Capítulo 1

Processamento de Linguagem Natural via Aprendizagem Profunda

Bárbara Stéphanie Neves Oliveira, Luís Gustavo C. do Rêgo, Lucas Peres, Ticiana L. Coelho da Silva, José Antônio F. de Macêdo

Abstract

Humans need to communicate. Out of this basic need combined with the Web, a vast amount of text has been generated on a daily basis. Given the presence of a lot of information allocated in different resources, it becomes vital to enable machines to understand spoken and written texts. This chapter presents how Deep Learning techniques can solve Natural Language Processing (NLP) tasks (e.g., Text Classification and Sentence Summarization), aiming to benefit from the computational power currently available and the low need for feature engineering when using these models. Initially, some essential concepts about NLP and Deep Learning are presented. Then, different pre-processing and textual representation techniques are explained to be used as input in Deep Learning models. Finally, it is shown how to apply the knowledge acquired in real applications of NLP.

Resumo

Seres humanos precisam de comunicação. A partir da junção dessa necessidade básica com a Web, uma grande quantidade de texto tem sido gerada diariamente. Dada a presença de muitas informações alocadas em diferentes meios, torna-se vital permitir que máquinas compreendam textos falados e escritos. Este capítulo apresenta como técnicas de Aprendizagem Profunda podem ser utilizadas na resolução de tarefas de Processamento de Linguagem Natural (PLN), como Classificação e Sumarização de Sentenças, visando o benefício do poder computacional disponível atualmente e da baixa necessidade de engenharia de features na utilização destes modelos. Inicialmente, são apresentados alguns conceitos importantes sobre PLN e Aprendizagem Profunda. Em seguida, diferentes técnicas de pré-processamento e representação textuais são explicadas a fim de serem usadas como entrada em modelos de Aprendizagem Profunda. Por fim, é mostrado como aplicar os conhecimentos adquiridos em aplicações reais do PLN.

1.1. Introdução

A comunicação, como uma necessidade básica da condição humana, juntamente com a existência da *Web* permitem que uma vasta quantidade de textos escritos e falados seja gerada diariamente. Dado o conteúdo textual presente em mídias sociais, aplicativos de bate-papo, *e-mails*, análises de produtos, artigos de notícias, trabalhos de pesquisa e *ebooks*, tornou-se vital a existência de um processamento automático de textos a fim de oferecer assistência ou tomar decisões para diversas tarefas diárias.

A capacidade de entender textos ou áudios em linguagem natural por uma máquina é um problema que vem sendo investigado há muito tempo [Chollet 2021]. As primeiras tentativas de construção de sistemas de Processamento de Linguagem Natural (PLN, de *Natural Language Processing* ou NLP) foram feitas através da análise intrínseca de linguagens que são naturalmente moldadas por um processo de evolução (por isso o termo "natural"). O PLN moderno envolve não apenas a habilidade de entendimento de uma linguagem como também possibilita, de forma automática, a extração de informações por meio de tarefas, tais como Classificação de Textos, Reconhecimento de Entidades Nomeadas (NER, de *Named Entity Recognition*), Desambiguação do Sentido das Palavras, e *Part-of-Speech* (POS) *tagging*.

Modelos de Aprendizagem Profunda, do inglês *Deep Learning* (também conhecida como Aprendizado Profundo ou Redes Neurais Profundas), aprendem vários níveis de representação de complexidade/abstração dos dados de forma crescente. Vários fatores evidenciam porque esses modelos têm sido amplamente usados em tarefas de PLN: (i) exigem pouca engenharia de *features*; (ii) produzem representações vetoriais que capturam similaridades de unidades linguísticas (palavras, por exemplo) presentes em textos, permitindo que sistemas de PLN possuam uma espécie de dependência de conhecimento; (iii) permitem aprendizado não supervisionado ou semi-supervisionado, o que é importante quando se tem um grande volume de dados e nenhum rótulo; (iv) aprendem vários níveis de representação, possibilitando que o nível mais baixo geralmente possa ser compartilhado entre diferentes tarefas; e (v) naturalmente lidam com a recursividade da linguagem humana, sendo capazes de capturar informações de forma sequencial.

O uso de modelos de Aprendizagem Profunda no PLN iniciou-se com a investigação da capacidade de compreensão de linguagem por Redes Neurais Recorrentes (RNNs, do inglês *Recurrent Neural Networks*) e redes LSTM (*Longest Shortest Term Memory*) [Hochreiter and Schmidhuber 1997]. Essas duas arquiteturas dominaram o PLN de forma geral de 2015 a 2017, uma vez que processam textos de comprimento variável. Os modelos LSTM bidirecionais, em particular, definiram o estado da arte em muitas tarefas importantes, desde Sumarização de Textos até Tradução Automática. Contudo, por volta de 2017 e 2018, uma nova arquitetura surgiu para "substituir" as RNNs: o *Transformer* [Vaswani et al. 2017], que permitiu um progresso considerável do PLN em um curto período de tempo.

O objetivo deste capítulo não é expor todas essas aplicações e arquiteturas de forma abrangente. Em vez disso, o foco está em como aplicar de forma prática representações textuais existentes e obtidas através de técnicas de Aprendizagem Profunda para resolver problemas de PLN. Este capítulo aborda ainda as diferentes etapas de processamento de textos realizada antes de treinar uma rede neural para uma tarefa de PLN. Ao

longo do capítulo, o leitor irá se familiarizar com as diferentes representações textuais que existem na literatura, desde a forma mais tradicional por *bag-of-words*, até representações via *word embeddings*. Apresentadas as principais representações textuais, este capítulo explorará duas tarefas de PLN populares e representativas: Classificação de Textos (ou Análise de Sentimentos) e Sumarização de Sentenças, que basicamente classificam conteúdos de textos e abreviam certos tipos de frases, respectivamente.

Em suma, este capítulo está organizado da seguinte forma: a Seção 1.2 apresenta os conceitos preliminares para entendimento deste capítulo. A Seção 1.3 fornece um roteiro de como alimentar uma rede neural com textos para treinamento de uma tarefa de PNL. A Seção 1.4 introduz as várias formas de representação de textos em vetores numéricos presentes na literatura. A Seção 1.5 discute como construir uma boa representação textual em tarefas de PLN. A Seção 1.6 exemplifica, por meio de tarefas de PLN, o uso de redes neurais profundas como mecanismo de solução. Por fim, as Seções 1.7 e 1.8 apresentam alguns desafios de pesquisa e as considerações finais, respectivamente.

1.1.1. Tecnologias Utilizadas

Ao longo deste curso, muitas tecnologias foram empregadas no desenvolvimento dos códigos relacionados a cada um dos conceitos apresentados e das tarefas de PLN utilizadas como exemplos.

A linguagem *Python* na versão 3.8 foi utilizada em *notebooks Colaboratory* ou, apenas, *Colab*[1]. *Notebooks* são páginas *Web* que permitem a escrita e execução de código em um mesmo documento.

Para auxiliar no desenvolvimento do código, as principais bibliotecas foram utilizadas:

- ***Numpy***[2] adiciona suporte para trabalhar, de forma eficiente, com vetores e matrizes numéricas, bem como funções otimizadas para essas estruturas.
- ***Keras***[3] provê interfaces para a criação, treinamento e avaliação das principais estruturas de redes neurais artificiais. Foi construída utilizando como base a biblioteca *TensorFlow*[4].
- ***Scikit-learn***[5] contém os principais algoritmos para diferentes tarefas de Aprendizagem de Máquina, como Classificação e Regressão.
- ***Matplotlib***[6] e ***Seaborn***[7], utilizadas para construção de gráficos.

A biblioteca *spaCy*[8] e NLTK[9] também foram utilizadas, bem como o *framework*

[1] https://colab.research.google.com
[2] https://numpy.org/
[3] https://keras.io/
[4] https://www.tensorflow.org/
[5] https://scikit-learn.org/stable/
[6] https://matplotlib.org/
[7] https://seaborn.pydata.org/
[8] https://spacy.io/
[9] https://www.nltk.org/

flairNLP[10], para trechos de códigos que envolvem o uso de modelos de PLN mais modernos, além de outras bibliotecas tradicionais da linguagem *Python*, como `string` e `unicode`.

1.2. Preliminares

A presente seção apresenta um histórico de como diferentes arquiteturas de Aprendizagem Profunda se relacionam com problemas de PLN. Além disso, são descritos alguns conceitos básicos sobre redes neurais e algumas das arquiteturas mais úteis no contexto do Processamento de Linguagem Natural.

1.2.1. Breve Histórico do Uso de Aprendizagem Profunda no PLN

A linguagem natural é complexa, ambígua, e está em constante mudança. Se para um ser humano é preciso de tempo e estudo para entender o sistema de princípios, condições e regras que formam os elementos ou propriedades de todas as línguas, criar algoritmos que possam dar sentido à linguagem é uma das, se não a mais inerente e importante, frente do PLN.

Além disso, não existe um idioma universal. A linguagem também não é flexível à formalização, isto é, não é trivial criar um conjunto complexo de regras que defina inteiramente o Português, o Inglês, o Japonês, ou esses três idiomas juntos, por exemplo. Após várias décadas de esforço para a criação de regras manuais para certas línguas [Chollet 2021], hoje, o PLN usa técnicas de Aprendizagem Profunda e grandes conjuntos de dados para que seja possível não só entender uma linguagem, como também solucionar diferentes problemas, como:

- **Classificação de Texto.** *"Qual é o tema deste texto?", "Esta notícia é verdadeira ou falsa?"*
- **Reconhecimento de Entidade Nomeada.** *"Como extrair o nome de todas as pessoas e lugares presentes nesta página?"*
- ***Language Modeling*.** *"Qual deve ser a próxima palavra nesta frase incompleta?"*
- **Sumarização.** *"Como você resumiria este artigo em um parágrafo?"*

A Figura 1.1 resume a evolução da área de Processamento de Linguagem Natural durante a era da Aprendizagem Profunda. A partir dos anos 2000, redes neurais começaram a ser utilizadas para a tarefa de *Language Modeling*, que visa prever a próxima palavra em uma sentença tomando como base as palavras anteriores [Bengio et al. 2000b]. Já na última década, pesquisas em PLN tiveram avanços consideráveis principalmente a partir da introdução do algoritmo *Word2Vec* [Mikolov et al. 2013]. O modelo trouxe uma nova forma de representar relações e similaridades entre termos através de vetores de palavras, ao demonstrar que palavras similares estariam próximas umas às outras em um mesmo espaço vetorial (ou espaço/plano Euclidiano), de dimensão finita. Apesar de existirem modelos mais recentes, o *Word2Vec* ainda é uma escolha popular e amplamente utilizada hoje.

[10] `https://github.com/flairNLP/flair`

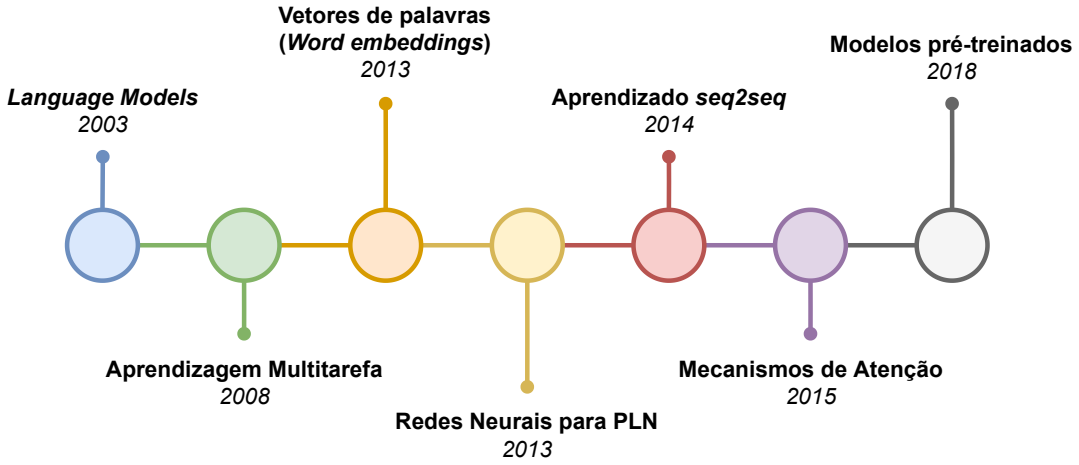

Figura 1.1. Avanços na área de PLN e Aprendizagem Profunda ao longo do tempo. Adaptado de [Louis 2020].

Durante o apogeu do *Word2Vec*, também em 2013, ocorreu o marco da adoção de modelos de redes neurais no PLN, em particular dois tipos bem definidos: Redes Neurais Recorrentes (RNNs, de *Recurrent Neural Networks*) e Redes Neurais Convolucionais (*Convolutional Neural Networks* ou CNNs). Graças a sua arquitetura, RNNs tornaram-se renomadas por lidar com sequências de entrada (como frases e parágrafos) de forma dinâmica e por possuir uma característica de recorrência, que permite representar a ordem das palavras e seu efeito em palavras anteriores e posteriores. Ao mesmo tempo, as redes CNNs que estavam começando a ser amplamente adotadas pela comunidade de Visão Computacional, também começaram a ser aplicadas em PLN. A vantagem de usar CNNs para lidar com sequências de texto é que elas são mais paralelizáveis do que RNNs.

Uma outra desvantagem das RNNs abre caminho para o problema da perda de informações através do cálculo dos gradientes. Como essas redes são muito profundas, os erros retro-propagados podem ser, potencialmente, números cada vez menores, resultando em um longo processo de treinamento ou simplesmente nenhum treinamento depois de um certo período de tempo. Um problema similar ocorre quando os erros são muito grandes, ocasionando no problema da explosão dos erros dos gradientes, fazendo com que os pesos não possuam nenhuma representação significativa. Para tentar mitigar os problemas das RNNs, unidades de memória de longo e curto prazo (*long short-term memory* ou LSTM) foram introduzidas. As conhecidas redes LSTM possuem a capacidade de detectar dependências de longo prazo nos dados. No contexto de PLN, isso significa que elas são capazes de analisar sentenças/textos cada vez maiores e não "esquecer" das representações aprendidas previamente.

Em 2014, [Sutskever et al. 2014a] propôs uma abordagem chamada de *sequence-to-sequence learning* (aprendizado de sequência a sequência ou, apenas, *seq2seq*), que mapeia uma sequência para outra usando uma rede neural. A tarefa de Tradução de Texto é uma referência clássica de *sequence-to-sequence learning*, por exemplo. O progresso

com essa abordagem foi tão significativo que o *Google* anunciou em 2016 que estava substituindo oficialmente seus modelos de Tradução Automática por um modelo neural *sequence-to-sequence*. Arquiteturas que utilizam redes neurais RNN ou LSTM para um problema *sequence-to-sequence*, quando recebem uma sequência como entrada, as transformam em um vetor (operação chamada de codificação), realizam as operações necessárias e transformam o vetor resultante em uma sequência (operação chamada de decodificação). Como essas operações são praticamente etapas bem definidas, esse modelo acabou sendo cunhado como codificador-decodificador[11] [Sutskever et al. 2014b].

Uma das barreiras encontradas para esse tipo de modelo é a sua limitação no tamanho das sequências das representações internas das palavras (vetores utilizados na codificação). O codificador comprime toda a sentença de origem em um único vetor, o que pode ser muito complicado de calcular porque o número de significados possíveis é infinito. Quando o codificador é forçado a colocar todas as informações em um único vetor, é provável que ele "esqueça" alguma coisa.

Mecanismos de atenção [Bahdanau et al. 2014] de certa forma tratam com esse problema e foram propostos tentando mimetizar a forma com que humanos lidam com frases em geral: atentando-se apenas para trechos específicos de uma sentença por vez e não ao texto inteiro. Nas RNNs, por exemplo, é considerado apenas o último estado do codificador no modelo codificador-decodificador (ou seja, apenas as últimas palavras de uma sequência de entrada); na abordagem utilizando mecanismos de atenção, todos os estados do codificador são considerados em cada etapa de decodificação. Dessa forma, é possível acessar informações sobre todos os elementos da sequência de entrada.

De fato, mecanismos de atenção extraem informações de toda a sequência de entrada a partir da média ponderada usada como uma espécie de pontuação (*score*) de todos os estados passados do codificador, permitindo que o decodificador atribua maior importância a um determinado elemento da entrada para cada elemento da saída. O mecanismo é potencialmente útil para qualquer tarefa que exija a tomada de decisões com base em certas partes da entrada. A limitação desses mecanismos de atenção está relacionada ao seu tempo de treinamento, uma vez que para analisar um determinado elemento de entrada, todos os seus elementos anteriores devem ter sido analisados, tornado o treinamento dessas redes demorado e computacionalmente custoso.

A arquitetura *Transformer* [Vaswani et al. 2017] (e suas variações) é o atual estado-da-arte para tarefas de PLN. Esse modelo foi pensado exclusivamente para a tarefa de Tradução de Texto e extrai informações de cada palavra utilizando mecanismos de auto-atenção (*self-attention*) para inferir a importância de todas as palavras de uma sequência em relação à palavra que está sendo analisada, sem nenhuma unidade de RNN envolvida na operação. Em suma, mecanismos de auto-atenção são usados para observar as palavras ao redor (contexto) em um texto, para obter representações de palavras mais contextualmente sensíveis. Modelos *Transformers* utilizam apenas os *scores* e funções de ativação para que a arquitetura como um todo possa ser altamente paralelizável e eficiente.

A última grande inovação no mundo do PLN são, sem dúvida, os grandes *Lan-*

[11] Também é conhecido por modelo *encoder-decoder* na literatura e será abordado com mais detalhes na Seção 1.6.1.

guage Models (LMs, ou Modelos de Linguagem) pré-treinados. Embora tenham sido propostos pela primeira vez em 2015 [Dai and Le 2015], apenas recentemente que mostraram uma grande melhoria em relação aos métodos de última geração para uma gama diversificada de tarefas. Os vetores de saída de LMs pré-treinados podem ser usados como *features* para um modelo final ou LMs pré-treinados podem ser ajustados aos dados da tarefa de destino, já que permitem um aprendizado eficiente com significativamente menos dados.

1.2.2. Conceitos Básicos de Redes Neurais

Imagine duas folhas de papel colorido: uma vermelha e outra azul. Coloque uma em cima da outra e amasse-as juntas em uma pequena bola. Considere que essa bola de papel amassada são os dados de entrada, e cada folha de papel é uma classe em um problema de classificação. O que uma rede neural deve fazer é descobrir uma forma de transformação da bola de papel que a desamassaria, de modo a tornar as duas classes separáveis novamente (veja a Figura 1.2). Com a Aprendizagem Profunda, isso seria implementado como uma série de transformações simples, como aquelas que poderiam ser aplicadas na bola de papel com os dedos, um movimento de cada vez.

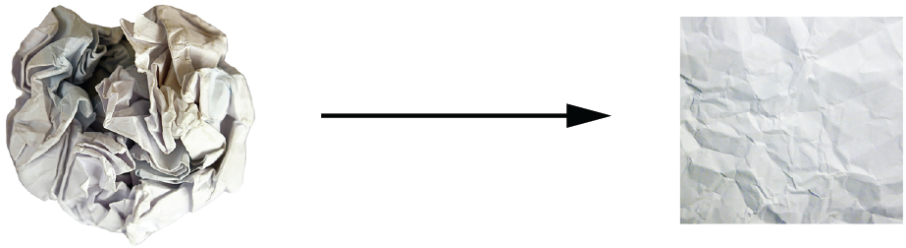

Figura 1.2. Desamassando uma variedade complicada de dados. Fonte: [Chollet 2021].

A Aprendizagem Profunda, também conhecida como Aprendizado Profundo (termos oriundos da expressão em inglês *Deep Learning*), é uma subárea do Aprendizado de Máquina que combina dezenas ou até centenas de camadas sucessivas de representações de processamento não linear para extrair e transformar atributos de entrada em modelos preditivos, chamados de redes neurais [Zhang et al. 2018]. Em suma, redes neurais profundas possuem uma estrutura matemática que aprende representações em vários estágios: as camadas mais próximas aos dados aprendem características simples, enquanto as de alto nível aprendem características mais complexas, sendo estas derivadas das camadas anteriores.

Apesar de modelos de Aprendizagem Profunda serem comumente chamados de redes neurais, estes métodos originaram-se como modelos de Aprendizagem de Máquina inspirados em neurônios biológicos. A flexibilidade dessas estruturas as tornaram ideais para tarefas complexas e, principalmente, que envolvem dados não estruturados, como imagens, sons, vídeos e textos.

As primeiras versões das redes neurais eram extremamente simples e capazes de realizar apenas cálculos lógicos básicos [McCulloch and Pitts 1943]. Com o passar dos anos, essas primeiras arquiteturas evoluíram para o *Perceptron* [Rosenblatt 1958]. A Figura 1.3 apresenta a unidade básica de um *Perceptron*, chamada de Unidade Lógica de

Limiar (do inglês, TLU). Essa unidade recebe entradas numéricas x e as associa a pesos w ao aplicar uma soma ponderada. Em seguida, uma função degrau é aplicada ao resultado dessa soma. A expressão $h_w(x) = \text{degrau}(z)$, onde $z = x^T w$ e $\text{degrau}(z)$ é a função de Heaviside, apresenta como as entradas são processadas em uma TLU.

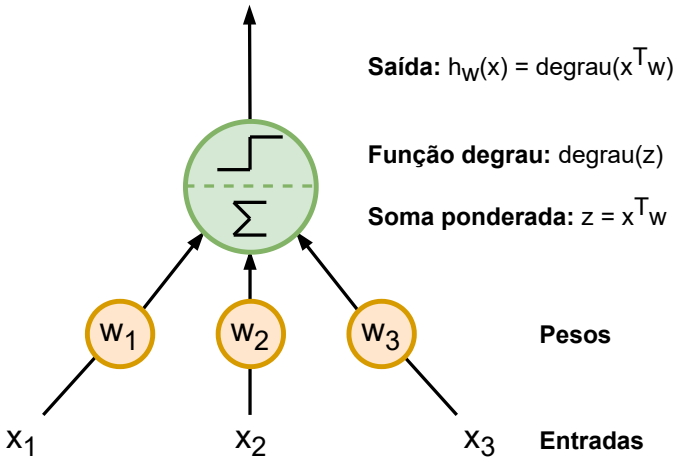

Figura 1.3. Unidade Lógica de Limiar (TLU). Adaptado de [Géron 2019].

Um *Perceptron* basicamente é uma estrutura formada por apenas uma camada de entrada e uma camada de TLUs, onde todas as TLUs estão ligadas a todas as entradas. O *Perceptron* também conta com um tipo adicional de neurônio chamado viés (*bias*), conectado a todas as TLUs. A Figura 1.4 mostra um exemplo de *Perceptron*. De forma similar à TLU, é possível calcular a saída de um *Perceptron* utilizando a fórmula $h_{W,b}(X) = \varphi(XW + b)$, onde W é a matriz dos pesos que ligam as entradas a todas as TLUs, X é a matriz de entradas, b é o valor do neurônio de viés, e φ a função de ativação das TLUs.

Para treinar um *Perceptron*, deve-se aplicar a seguinte regra de aprendizado após o fornecimento de uma instância dos dados de treinamento:

$$w_{i,j}^{proxima\_iteracao} = w_{i,j} + \eta(y_j - \widehat{y}_j)x_i \tag{1}$$

onde

- $w_{i,j}$ é o peso entre a entrada i e a TLU j,
- η é a taxa de aprendizagem,
- y_j é o valor real da saída,
- $\widehat{y}_j$ é o valor estimado pelo *Perceptron* como saída,
- x_i é o valor da entrada i.

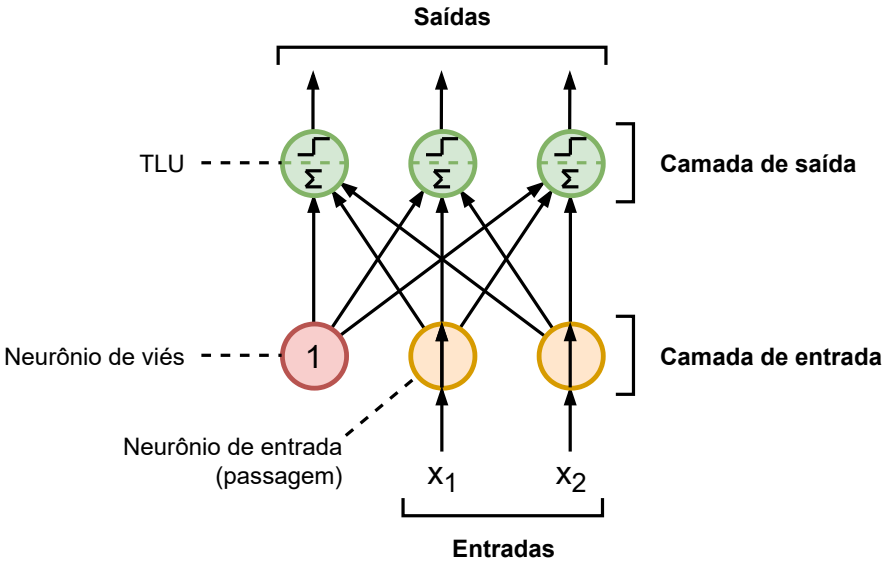

Figura 1.4. Arquitetura de um *Perceptron*. Adaptado de [Géron 2019].

O código do Programa 1.2.1 utiliza a implementação da biblioteca *Scikit-learn* de um *Perceptron* para classificar diferentes tipos da planta Íris.

Programa 1.2.1. Predição do tipo de planta Íris utilizando um *Perceptron* simples. Adaptado de [Géron 2019].

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2,3)]
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron()
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```

Uma das principais limitações dos *Perceptrons* é o fato de não conseguirem resolver problemas relativamente simples, como o problema de classificação da porta lógica XOR. O *Perceptron* Multicamada (do inglês, MLP) surge como uma evolução do *Perceptron* convencional. A estrutura de uma MLP é formada por uma camada de entrada, *n* camadas empilhadas de TLUs, chamadas de camadas ocultas (*hidden layers*), e uma camada adicional de TLUs que representa a saída da MLP. A Figura 1.5 apresenta uma MLP simples com uma camada oculta. Quando uma MLP possui mais do que duas camadas ocultas, é considerada uma DNN (sigla para a expressão Rede Neural Profunda em inglês).

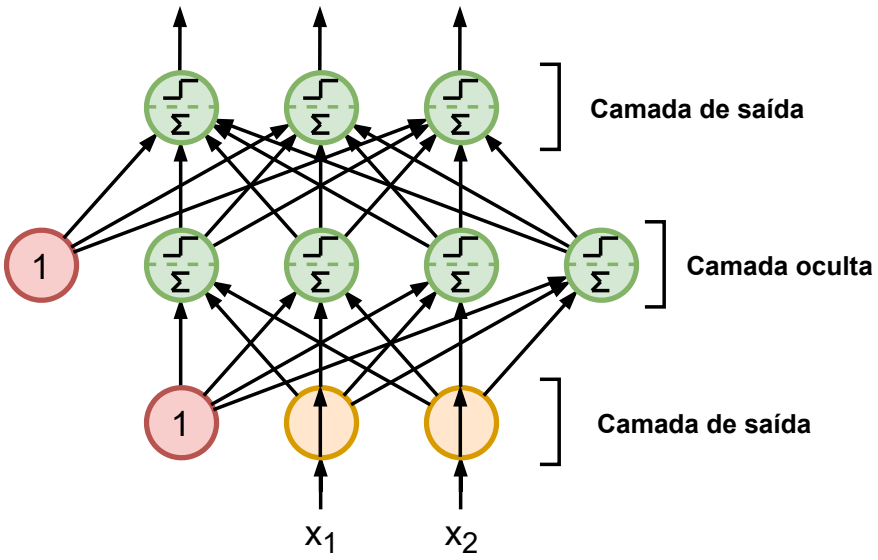

Figura 1.5. Arquitetura de um MLP com duas entradas e uma camada oculta. Adaptado de [Géron 2019].

Um dos conceitos mais importantes associados às DNNs é o algoritmo de retropropagação (*backpropagation*) [Rumelhart et al. 1985] utilizado para treiná-las. Esse algoritmo pode ser dividido em duas etapas: o *forward pass* e o *backward pass*. Na etapa de *forward pass*, a DNN recebe um *batch* (lote) de instâncias de dados e calcula seus resultados após a interação com todas as suas camadas. Em seguida, o algoritmo calcula o erro entre a saída gerada pela rede e a saída esperada presente nos dados, e utiliza esse resultado para calcular como cada peso da DNN contribui com esse erro encontrado. Por fim, o algoritmo executa um passo de gradiente descendente para atualizar todos os pesos da rede neural.

Um outro fator importante que diferencia um *Perceptron* de uma DNN são as funções de ativação utilizadas em cada neurônio (TLU): as redes profundas utilizam funções de ativação não lineares, como tangente hiperbólica e linear retificada (ReLU) para aproximar qualquer função contínua. O trecho de código a seguir exemplifica a utilização de uma DNN ao construir[12] uma rede capaz de classificar diferentes frases contendo, em geral, uma conotação de sentimentos positivos ou negativos.

O Programa 1.2.2 cria uma DNN com quatro camadas para o problema de Análise de Sentimentos de frases: a primeira recebe um vetor de TF-IDF (detalhes na Seção 1.4.1), que por sua vez é utilizado como entrada para a rede. A segunda e terceira camadas são criadas em um formato denso, isto é, com todos os seus neurônios conectados a todos os neurônios da camada anterior, com cinco (5) neurônios cada e função de ativação ReLU. Por fim, a quarta camada densa com dois (2) neurônios utiliza a função de ativação *softmax* para informar qual classe a frase de entrada tem mais chances de pertencer, se zero (0), para frases com sentimentos negativos, ou um (1), para frases com sentimentos positivos.

[12] A importação e divisão dos dados serão omitidos para que se possa focar apenas na DNN.

Programa 1.2.2. Criação, compilação e treinamento de uma rede neural profunda com duas camadas escondidas com cinco (5) neurônios cada.

```
import tensorflow as tf
from tensorflow import keras

model = keras.models.Sequential()
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(5, activation="relu"))
model.add(keras.layers.Dense(5, activation="relu"))
model.add(keras.layers.Dense(2, activation="softmax"))

model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd", metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=30)
```

Uma vez que a DNN é criada com a biblioteca *Keras*, é preciso configurar o modelo para treinamento através da função `compile()`, definindo, por exemplo, qual a função de perda (*loss*), otimizador e métricas que serão utilizadas. A entropia cruzada (do inglês, *cross-entropy*) foi utilizada no Programa 1.2.2 como função de perda, pois as classes possíveis de saída são discretas e exclusivas. Para treinar o modelo, é usado um gradiente descendente estocástico (parâmetro `"sgd"`). Por fim, como métrica de avaliação, utiliza-se nesse exemplo a acurácia (`"accuracy"`), finalizando a etapa de *backward pass*. Para treinar o modelo criado, basta chamar a função `fit()`, passando os dados utilizados e a quantidade de épocas que o modelo será otimizado. Um ciclo que passa pelos exemplos de treinamento da rede é chamada de época. As épocas são repetidas até ocorrer uma convergência, isto é, um ponto no treinamento após o qual as mudanças calculadas com a métrica de avaliação tornam-se menores e os erros produzidos pelo modelo (computados pela função de perda) são mínimos.

O *Perceptron* multicamada é indicado para tarefas onde têm-se, essencialmente, entradas de tamanho fixo e que resultam em saídas discretas. Para problemas com entradas em forma de sequências, como áudios, textos ou vídeos, essas arquiteturas mais simples não são as mais indicadas. Para esse tipo de problema com entradas muito longas, é comum o uso de arquiteturas chamadas de Redes Neurais Recorrentes.

A semelhança entre MLPs tradicionais e RNNs é que estas últimas também possuem conexões para neurônios de camadas anteriores. Essas conexões são capazes de modelar informações anteriores de uma determinada sequência de entrada. A Figura 1.6 (à esquerda) mostra um único neurônio, com uma ligação adicional da sua saída para sua entrada. Isso significa que a cada passo de tempo t, este neurônio recebe como entrada o trecho da sequência x_t e o trecho anterior x_{t-1} que vem do passo de tempo anterior $(t-1)$. Esse processo é realizado recursivamente até o passo de tempo $t = 0$. A Figura 1.6 (à direita) também ilustra como as autoconexões de um neurônio funcionam ao longo do tempo.

Da mesma forma em que é criada uma camada de neurônios "convencionais" em

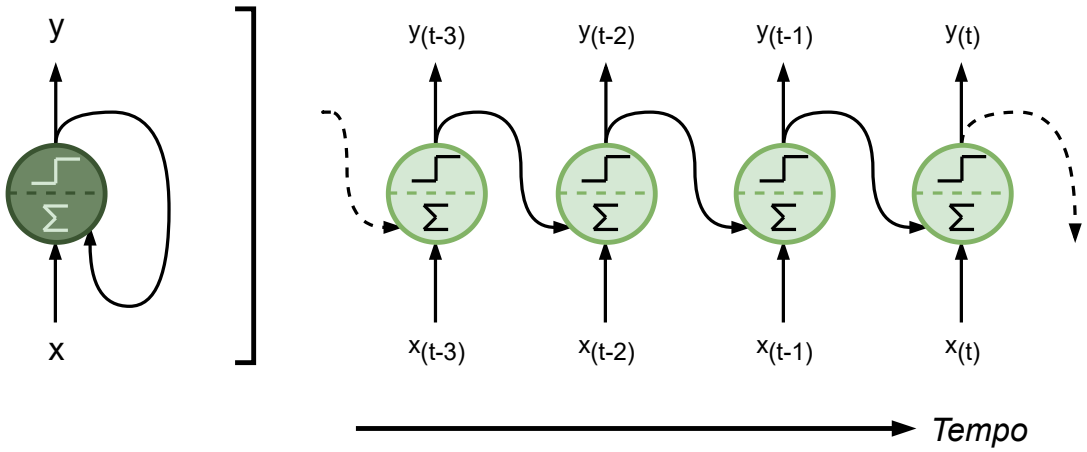

Figura 1.6. Unidade RNN (à esquerda) e unidade RNN desenrolada (à direita). Adaptado de [Géron 2019].

uma MLP, é possível criar uma camada de neurônios com novas autoconexões, como exemplificado na Figura 1.7.

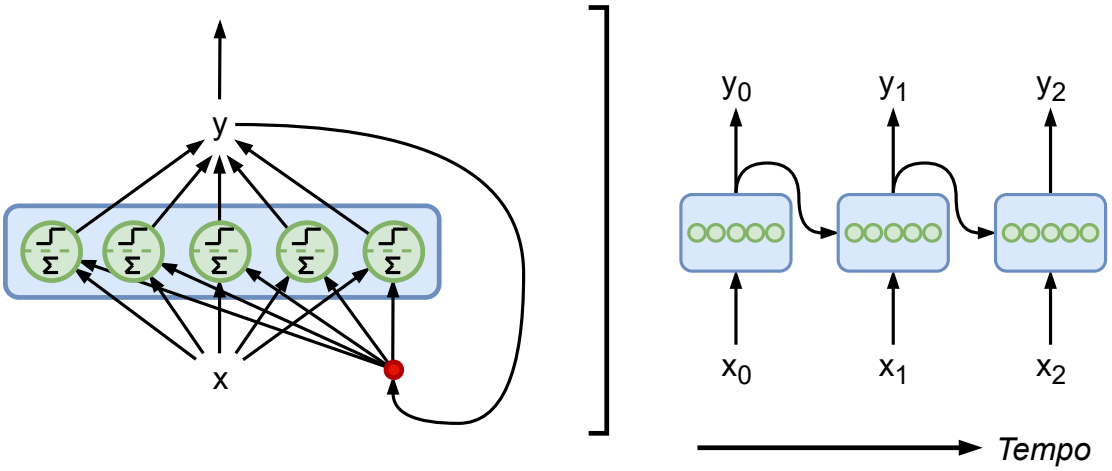

Figura 1.7. Rede neural com uma camada de unidades RNN (à esquerda) e a mesma rede desenrolada ao longo do tempo (à direita). Adaptado de [Géron 2019].

Para treinar uma RNN, deve-se aplicar a seguinte regra de aprendizado após fornecer todas as entradas do intervalo de tempo t dentro de uma matriz X_t:

$$Y_t = \phi([X_t Y_{t-1}]W + b) \tag{2}$$

onde

- Y_t é a matriz de saídas da RNN no intervalo de tempo t,

- X_t é a matriz de entradas da RNN no intervalo de tempo t,
- Y_{t-1} é a matriz de saídas da RNN no intervalo de tempo $t-1$,
- $W = \begin{bmatrix} W_x \\ W_y \end{bmatrix}$ é o empilhamento dos pesos relacionados às entradas X_t e às saídas Y_{t-1},
- b é o vetor de neurônios de viés.

Quando é preciso lidar com sequências longas, células RNN tendem a crescer bastante ao longo do tempo. Isso é um potencial problema, dado que torna o treinamento computacionalmente custoso, demorado e instável. Um outro problema que surge quando lida-se com sequências longas é a perda de informações do começo da sequência por parte da RNN. Um exemplo real disso ocorre quando uma rede tenta traduzir uma sentença muito longa e perde informações do começo da frase.

Para algumas tarefas onde a ordem das entradas (ordem das palavras em um texto, por exemplo) são importantes, o uso de RNNs bidirecionais (BiRNNs) podem trazer um ganho significativo à performance do modelo. Uma BiRNN é composta por duas RNNs convencionais, em que uma processa a sequência de entrada em sua ordem natural, e a adicional processa a sequência em sua forma inversa. No final, é feito uma operação de concatenação das representações das duas RNNs. Dessa forma, uma BiRNN consegue capturar padrões que uma RNN convencional não consegue sozinha.

Alguns novos tipos de células baseadas em RNNs foram desenvolvidas para mitigar esses problemas. Dentre elas, a mais famosa é a célula de LSTM. A Figura 1.8 apresenta, de forma simplificada, uma célula LSTM. Comparando-a com uma célula RNN, como a da Figura 1.6, células LSTM possuem algo extra, visto que adicionam uma divisão em seu estado: o vetor $h_{(t)}$ representa o estado de curto prazo e o vetor $c_{(t)}$ representa o estado de longo prazo.

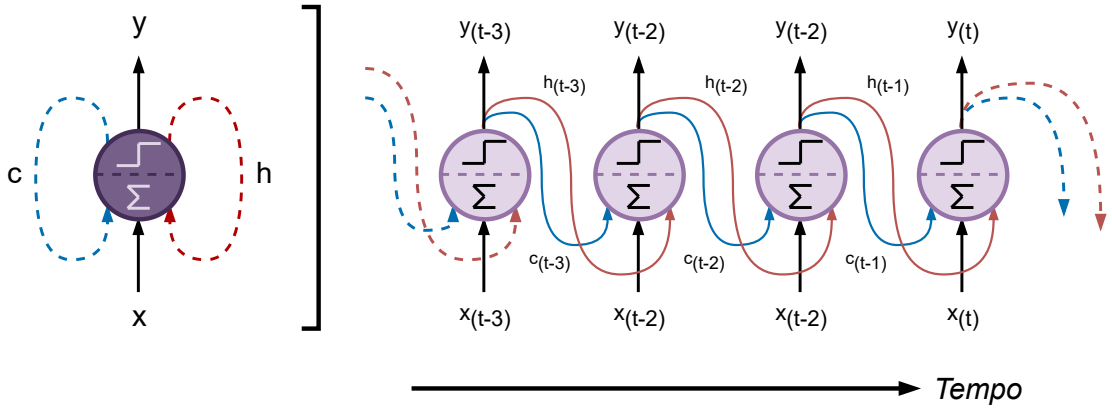

Figura 1.8. Unidade LSTM (à esquerda) e unidade LSTM desenrolada (à direita).

1.3. Preparação de Dados Textuais

Como o texto, em particular, origina a maior parte das comunicações e da produção cultural da humanidade, não é coincidência que ele também sirva de base para a criação de conjuntos de dados textuais[13], os quais são chamados de **corpus linguísticos**. Assim, em PLN, os modelos de Aprendizagem Profunda reconhecem padrões que possam ser aplicados a palavras, frases e/ou parágrafos inteiros.

Porém, não é possível alimentar modelos com palavras ou textos. Informações algébricas (ou seja, vetores numéricos) é que normalmente são extraídas de um corpus como forma de representar unidades linguísticas, especialmente, palavras. A representação vetorial de uma palavra pode ser obtida mediante um procedimento chamado de *text-vectorization* (vetorização do texto), que pode ser feito de várias maneiras. Dentre elas, segmentar o texto em palavras ou caracteres, ou até mesmo extrair *n-grams*[14], e transformar essas "partes" do texto em vetores [Chollet 2021].

Embora existam diversas formas de processar e preparar dados textuais para serem utilizados em modelos de Aprendizagem Profunda, quase todos seguem um mesmo padrão:

- Primeiramente, o texto deve ser **padronizado**, convertendo todas as letras em minúsculas[15] e removendo pontuações;
- Em seguida, é necessário dividir o texto em várias partes, chamados de *tokens*. Esse processo é conhecido como **tokenização**;
- Por fim, os *tokens* devem ser transformados em representações numéricas. É comum que sejam utilizadas estratégias de **indexação** e de ***embeddings***.

Seguindo esse roteiro, os dados em formato numérico podem ser usados para alimentar o modelo neural a ser treinado para alguma tarefa de PLN. A seguir, é explicada de forma mais detalhada cada caso.

1.3.1. Padronização

É comum que existam sentenças que apresentam uma mesma informação, mas que possuem estruturas diferentes. Suponha as seguintes frases: "*Olha só! Como PLN é legal! Estava estudando esses dias*", "*Olha só, como PLN é legal! Estive estudando esses dias*", e "*olha so como pln eh legal, estudei esses dias*". Apesar da informação passada ser a mesma, as frases podem muito bem ter sido escritas por três pessoas diferentes: uma mais animada, outra cética e outra mais informal, respectivamente.

Ainda utilizando o exemplo anterior, daria mais trabalho para um modelo aprender que *PLN* e *pln* tratam-se da mesma sigla, ou que *só* e *so* na verdade são um mesmo termo.

[13]Mesmo se a tarefa de PLN envolva fala ou áudios, uma transcrição é feita para que os arquivos de sons sejam convertidos em documentos textuais.

[14]*N-grams* (N-gramas) são grupos formados por *n* palavras ou caracteres consecutivos de uma determinada amostra de texto.

[15]Por convenção, a transformação de todas as letras em minúsculas é utilizada, mas a conversão para letras maiúsculas também é aceita.

A ideia principal da padronização é diminuir as diferenças entre textos que apresentam informações similares, permitindo assim que modelos não se preocupem em aprender tais detalhes.

Uma das estratégias mais básica e direta é, simplesmente, remover a pontuação e colocar os caracteres para minúsculo. Tal modificação tornam as frases bem mais semelhantes. Outra estratégia também comum é trocar as letras e vogais com acentos por suas versões sem acento, o que auxilia quando é trabalhado com dados de redes sociais, por exemplo, onde não é comum a formalidade de escrita. Uma forma simples de aplicação do processo de padronização é apresentada no Programa 1.3.1.

Programa 1.3.1. Padronização da sentença "*O gato caçou o rato!*".

```
import string
from unidecode import unidecode

def standardize(text):
    text = unidecode(text.lower())
    return "".join(char for char in text
        if char not in string.punctuation)

standardize("O gato caçou o rato!")
# 'o gato caçou o rato'
```

1.3.2. Tokenização

Uma vez padronizado, é necessário quebrar o texto em unidades a serem vetorizadas. Dividir um texto em unidades distintas (palavras, caracteres ou *n-grams*) nada mais é do que **tokenizar** este texto [Webster and Kit 1992]. Em geral, é mais comum dividir o texto em palavras ou *n-grams*.

Existem dois tipos principais de estratégias para formar *embeddings* (vetores): aquelas que utilizam a sequência de *tokens* para agregar uma informação de contexto e ordem (chamados de modelos sequenciais), e aqueles que importam-se apenas com a ocorrência e frequência dos *tokens* (chamados de modelos *bag-of-words*). Essas estratégias são apresentadas e explicadas com mais detalhes na Seção 1.4.

Para os modelos sequenciais, é comum que os *tokens* sejam criados a nível das palavras, para poder processar a formação do texto. Em contrapartida, para os modelos de *bag-of-words*, é comum que a tokenização seja feita a nível de *n-grams* a fim de contar/identificar a presença de termos e expressões. O processo de tokenização é representado de forma simplificada no Programa 1.3.2.

Entretanto, vale ressaltar que existem palavras que diferem de outras apenas por sufixos flexionais, modificando seu gênero, número, grau, etc., como em *presidente* e *presidenta*, *estava* e *estive*, *casa* e *casas*. Removendo esses elementos linguísticos e reduzindo as palavras aos seus radicais (*stem*), tem-se o processo chamado de *stemming* ou estemização. Para esse procedimento, existem funções prontas da biblioteca NLTK, conforme o código exemplo no Programa 1.3.3. Note como as frases distintas parecem

Programa 1.3.2. Padronização e tokenização da sentença "*O gato caçou o rato!*". Os *tokens* obtidos são a nível das palavras.

```
def tokenize(text):
    text = standardize(text)
    return text.split()

tokenize("O gato caçou o rato!")
# ['o', 'gato', 'caçou', 'o', 'rato']
```

ainda mais semelhantes se comparados os *stems* ao invés das próprias palavras.

Programa 1.3.3. Estemização das sentenças "*Os gatos caçaram os ratos*" e "*O gato caçou o rato*". Ambas retornam os mesmos *stems* quando aplicada a função `stem`: `o`, `gat`, `cac`, `o`, e `rat`.

```
from nltk.stem.snowball import PortugueseStemmer
stemmer = PortugueseStemmer()

def stem(text):
  return [stemmer.stem(w) for w in tokenize(text)]

stem("Os gatos caçaram os ratos")
# ['os', 'gat', 'cac', 'os', 'rat']

stem("O gato caçou o rato")
# ['o', 'gat', 'cac', 'o', 'rat']
```

Às vezes torna-se necessário não só remover sufixos flexionais, como também manter algumas dessas informações (tempo verbal, por exemplo) ou, até mesmo, mapear os *stems* para uma outra palavra que melhor os represente (o *stem* da palavra *casaram* poderia ser, por exemplo, *casar*, dando uma resposta diferente do *stem* de *casas*, que seria *casa*). Esse processo é conhecido como lematização (*lemmatization*). Para conseguir realizar esse processamento, são necessários modelos ou técnicas mais complexas. É possível realizar a lematização em Português com a biblioteca *spaCy* (Programa 1.3.4). Lembre-se que tanto *stems* como *lemmas* funcionam no final como *tokens*.

1.3.3. Indexação do Vocabulário

Uma vez que os *tokens* do texto são encontrados, é necessário codificá-los em uma representação numérica. Embora seja possível aplicar alguma função *hash* para os *tokens* e encontrar um número que os represente, é mais comum optar por construir um vocabulário de todos os termos presentes nos textos processados. Para cada *token*, atribui-se um número que serve como seu identificador (ou índice) dentro do vocabulário. Uma forma de implementar como obter um vocabulário pode ser encontrada no Programa 1.3.5.

Com o vocabulário construído, é possível utilizar o índice de cada termo e codificá-lo de forma vetorial utilizando uma estratégia chamada de codificação *one-hot* (também

Programa 1.3.4. Lematização das sentenças "*Amigos, amizade e carreira são importantes*" e "*Amigas, amizades e carreira é importante*". Os *tokens* obtidos com a função `lemma` para ambas as frases são `amigo`, `amizade`, `e`, `carreira`, `ser`, e `importante`.

```
import spacy

nlp = spacy.load('pt')

def lemma(text):
  doc = nlp(vectorizer.standardize(text))
  return [token.lemma_ for token in doc]

lemma("Amigos, amizade e carreira são importantes")
# ['amigo', 'amizade', 'e', 'carreira', 'ser',
 ↪  'importante']

lemma("Amigas, amizades e carreira é importante")
# ['amigo', 'amizade', 'e', 'carreira', 'ser',
 ↪  'importante']
```

Programa 1.3.5. Criação de vocabulário para um pequeno corpus formado pelas sentenças "*O gato caçou o rato*" e "*Gato e cachorro são animais*".

```
def make_vocab(texts):
    # O vocabulário é iniciado com 0 que indica strings
     ↪  vazias, e 1 que indica palavras desconhecidas
    vocab = {"": 0, "[UNK]": 1}
    for text in texts:
      text = standardize(text)
      tokens = tokenize(text)
      for token in tokens:
        if token not in vocab:
          vocab[token] = len(vocab)
    inverse_vocab = dict(
      (v, k) for k, v in self.vocab.items())

    return vocab, inverse_vocab

make_vocab(["O gato caçou o rato",
            "Gato e cachorro são animais"])
# {'': 0, '[UNK]': 1, 'o': 2, 'gato': 3, 'caçou': 4, 'rato':
 ↪  5, 'e': 6, 'cachorro': 7, 'são': 8, 'animais': 9}
```

conhecida como *dummy*). Um vetor *one-hot* possui o mesmo tamanho do vocabulário (ou seja, sua dimensão é igual a quantidade de *tokens*) e é preenchido com zeros, exceto na posição referente ao índice da palavra, onde contém o valor 1. O método de codificação *one-hot* é útil por alguns motivos: (i) todos os vetores estão normalizados; (ii) é fácil de descobrir a palavra que um vetor representa; e (iii) as operações vetoriais são mais

simples, permitindo trabalhar com matrizes esparsas[16]. Ainda assim, o uso de vetores *one-hot* não são a melhor forma de como representar palavras e textos.

O Programa 1.3.6 mostra como é possível implementar a construção dos vetores *one-hot* a partir de uma sentença exemplo.

Programa 1.3.6. Codificação *one-hot* da sentença "*O gato caçou o rato*".

```
import numpy as np

def one_hot_encoder(text, vocab):
    tokens = tokenize(text)
    vectors = np.zeros((len(tokens), len(vocab)))
    for i, token in enumerate(tokens):
      token_idx = vocab.get(token, 1)
      vectors[i, token_idx] = 1
    return vectors

one_hot_encoder("O gato caçou o rato", vocab)
# array([[0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
#        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
#        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
#        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
#        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]])
```

1.4. Representação Textual

O conceito de representação textual não baseia-se apenas em vetores, muito menos em vetores de palavras *one-hot*. Primeiramente, deve-se analisar como palavras formam uma sentença. Línguas diferentes ordenam palavras semelhantes ou não de maneiras diferentes e, apesar da ordem importar, sua relação com o significado não é tão direta. Como representar as palavras é o principal motivo pelo qual surgem diferentes tipos de arquiteturas de PLN, que podem ser agrupadas em três categorias principais:

1. **Representando palavras como conjuntos.** Trata-se da representação mais simples. Aqui a ordem das palavras simplesmente é descartada e o texto é tratado como um conjunto não ordenado de palavras, obtido essencialmente por modelos *bag-of-words*.

2. **Representando palavras como sequências.** Quando a ordem das palavras é considerada, elas devem ser processadas estritamente na ordem em que aparecem, uma de cada vez. Quanto a isto, os já citados vetores de palavras, ou *word embeddings*, são amplamente utilizados. Métodos específicos (*Word2Vec* e GloVe, por exemplo) e/ou arquiteturas de redes neurais (como RNNs e *Transformers*) são modelos de sequência de palavras cujas saídas geram *word embeddings*.

[16]Matriz esparsa é uma matriz onde a maioria dos valores é 0.

3. **Modelando as palavras.** Por possuir diferentes arquiteturas, a segunda abordagem ainda pode ser fragmentada em diversas outras. Uma delas, talvez a mais popular atualmente, relaciona-se a um conceito um pouco abstrato: o de como modelar uma linguagem.

Mas, o que significa "*modelar uma linguagem*"? Ou, ainda, o que significa "*modelar algo*"? Um bom modelo deve simular o comportamento do mundo real, entendendo quais eventos são mais prováveis de acontecer, isto é, prever o que acontece a seguir dada alguma **descrição de contexto**. Um modelo de linguagem não é diferente: um evento pode ser uma unidade linguística (texto, frase, *token*, ou símbolo), onde o objetivo final do modelo é estimar as probabilidades de cada unidade linguística ocorrer.

Assim, os chamados *Language Models* estimam a probabilidade de diferentes unidades linguísticas, como símbolos, *tokens*, e sequências de *tokens*. Algumas arquiteturas que usam *Transformers*, por exemplo, são LMs, podendo ou não funcionar do jeito tradicional, ao computar probabilidades. Na verdade, esse conceito é bastante conhecido e LMs estão em ação todos os dias, sejam em *sites* de busca, serviços de tradução, teclados de celulares, e até mesmo na correção de escrita feita em serviços de *e-mail*. A Figura 1.9 ilustra de forma didática o funcionamento de um LM.

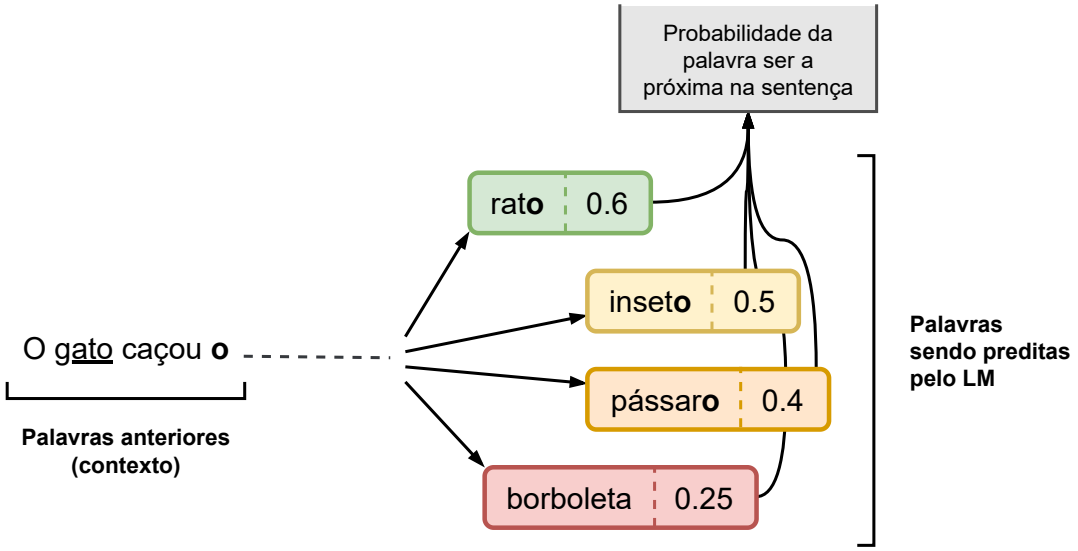

Figura 1.9. LM identificando qual a próxima palavra na frase "*O gato caçou o...*". À direita, estão listadas algumas das possíveis palavras juntamente com suas probabilidades. À esquerda, a palavra *gato* e o artigo *o* destacados indicam os termos mais importantes do "contexto" para predição. Nesse caso, a próxima palavra da sentença seria *rato*.

O escopo deste curso não inclui o aprofundamento dos tipos de LMs existentes utilizados para estimar probabilidades de palavras, pois seria necessário o estudo de diversos modelos estatísticos, além de conceitos da Teoria da Probabilidade, como Probabilidade Condicional. O que é importante que o leitor saiba[17] é que, dada uma palavra, o objetivo

[17] As informações apresentadas até aqui sobre LMs serão utilizadas nas próximas seções.

de um LM é identificar qual seria a próxima palavra, de acordo com o contexto da sentença. Porém, não só um LM pode atribuir probabilidades para próximas palavras, como também pode dar probabilidades a frases, como representado na Figura 1.10.

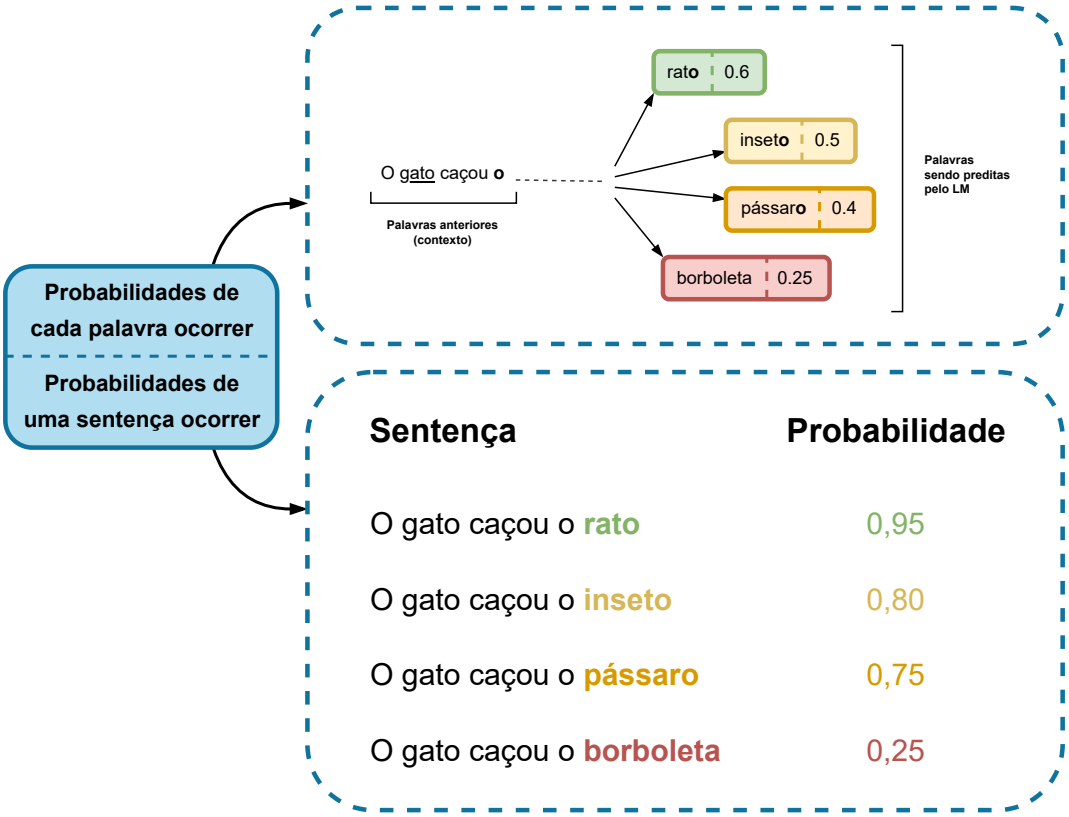

Figura 1.10. Além de palavras, LMs determinam qual sentença é mais adequada que outras com base no contexto.

Mesmo que modelos *bag-of-words* pareçam obsoletos se comparados com as soluções sofisticadas dos modelos de sequência não importa para qual tarefa ou conjunto de dados, isso definitivamente não é o caso: ambas as abordagens de como formar vetores de palavras descritas brevemente acima permanecem igualmente relevantes [Chollet 2021].

1.4.1. Representação por *Bag-of-Words*

A maneira mais simples de representar um texto é **descartar a ordem** e tratá-lo como um **conjunto de *tokens***. Existem diferentes formas de codificação com modelos *bag-of-words*, desde considerar palavras individuais (*unigrams*), até recuperar informações locais examinando grupos de sequências de *tokens* (*n-grams*).

***Unigrams* com codificação *multi-hot*.** Para exemplificação de um conjunto de *unigrams* (*bag-of-unigrams*), observe o Programa 1.4.1 que transforma a sentença "*o gato caçou o rato*" já padronizada.

A função `make_ngrams` presente no trecho de código do Programa 1.4.1 possui um nome genérico e para o parâmetro `n` da função está atribuído o valor 1. Repare que

Programa 1.4.1. *Bag-of-unigrams* da frase "*o gato caçou o rato*". Atente para o uso da função `make_ngrams`, que inicializa qualquer conjunto de *n-grams*.

```
sent = "o gato caçou o rato"

def make_ngrams(text, n=1):
  words = text.split()
  ngrams = [tuple(words[i:i + n]) for i in range(len(words) -
   ↪  n + 1)]
  return ngrams

unigrams = make_ngrams(sent, n=1)

print(unigrams)
# [('o',), ('gato',), ('caçou',), ('o',), ('rato',)]
```

unigrams nada mais são do que palavras únicas sem nenhuma ordem, e que a intenção é formar um conjunto apenas com as palavras distintas do texto.

Obtidos os *unigrams* da sentença exemplo, falta apenas aplicar a codificação *multi-hot*, também chamada de binária (*multi-hot binary enconding*). Diferentemente da codificação *one-hot*, a *multi-hot* bconsidera como entrada mais de um valor (no caso, mais de uma palavra). Sua funcionalidade será melhor explicada adiante porque, para *bag-of-unigrams*, o vetor resultante acaba sendo o mesmo que o da codificação *one-hot*.

Programa 1.4.2. Codificando os *unigrams* da sentença "*o gato caçou o rato*" com codificação *multi-hot* (função `multihot_encoder`).

```
from sklearn.preprocessing import MultiLabelBinarizer

def multihot_encoder(ngrams):
    mlb = MultiLabelBinarizer()
    binary_format = mlb.fit_transform(ngrams)
    print("Índices correspondentes de cada palavra =
     ↪  {}\n".format({k: v for k, v in
     ↪  enumerate(mlb.classes_)}))
    return binary_format

unigrams_vectors = multihot_encoder(unigrams)

print(unigrams_vectors)
# Índice correspondente de cada palavra = {0: 'caçou', 1:
 ↪  'gato', 2: 'o', 3: 'rato'}

# array([[0, 0, 1, 0],
#        [0, 1, 0, 0],
#        [1, 0, 0, 0],
#        [0, 0, 1, 0],
#        [0, 0, 0, 1]])
```

Para facilitar, a função `multihot_encoder` no Programa 1.4.2 faz uso do método `MultiLabelBinarizer`[18] da biblioteca *Scikit-learn*, comumente utilizado para codificação de *bag-of-words*. Em suma, o trecho de código do Programa 1.4.2 demonstra que a sentença seguiu com o padrão da codificação *one-hot*, onde cada palavra é representada como um único vetor preenchido por zeros, exceto as posições que indicam o índice correspondente, ditadas pelo valor 1.

***N-grams* com codificação *multi-hot*.** Dependendo do contexto, descartar a ordem das palavras pode não contribuir para o seu entendimento. Por exemplo, o termo *Bela Cruz* em um texto com descrições sobre cidades do estado do Ceará transmite um conceito bastante distinto do que o significado das palavras *bela* e *cruz* separadamente. Faz-se necessário inserir informações extras para uma representação de *bag-of-words* olhando para os *n-grams* em vez de palavras únicas (mais comumente, são utilizados *bigrams*). Com *bigrams* (*bag-of-bigrams*), a sentença do exemplo anterior no Programa 1.4.3 origina um diferente conjunto e diferentes vetores codificados.

Programa 1.4.3. *Bag-of-bigrams* e vetores resultantes da sentença "*o gato caçou o rato*" com codificação *multi-hot*.

```
bigrams = make_ngrams(sent, n=2)

print(bigrams)
# [('o', 'gato'), ('gato', 'caçou'), ('caçou', 'o'), ('o',
 ↪  'rato')]

bigrams_vectors = multihot_encoder(bigrams)

print(bigrams_vectors)
# Índices correspondentes de cada palavra = {0: 'caçou', 1:
 ↪  'gato', 2: 'o', 3: 'rato'}

# array([[0, 1, 1, 0],
#        [1, 1, 0, 0],
#        [1, 0, 1, 0],
#        [0, 0, 1, 1]])
```

O uso da codificação *multi-hot* torna-se mais intuitivo no Programa 1.4.3. O conjunto de *bigrams*, formado pelas tuplas `(o, gato)`, `(gato, caçou)`, `(caçou, o)`, e `(o, rato)`, e o vetor de saída resultante possuem quatro itens, e os índices de cada vetor seguem as mesmas posições que as palavras únicas da sentença (valor 0 para *caçou*, 1 para *gato*, e assim sucessivamente). Por isso que, para o primeiro vetor `[0, 1, 1, 0]`, correspondente à tupla `(o, gato)`, o valor 1 está presente nas posições 1 e 2, já que estas dizem respeito aos índices dos termos *gato* e *o*.

Por conta do parâmetro `n` em `make_ngrams`, é possível configurá-lo para retornar *n-grams* arbitrários além de *bigrams*: *trigrams*, 4-*grams*, 5-*grams*, etc.

[18] `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MultiLabelBinarizer.html`

***N-grams* com codificação TF-IDF.** Também é possível adicionar mais informações à representação por *bag-of-words* ao elencar quantas vezes cada palavra ou *n-grams* ocorrem no texto. Essa abordagem é possível pois, conforme um termo aparece abundantemente em um texto, entende-se que ele é crucial para o seu entendimento. Contar a frequência de palavras ou *n-grams* em um corpus também é importante, uma vez que é possível saber quais palavras repetem-se mais e quais pertencem a pequenos subconjuntos. Basicamente, o TF-IDF (*term frequency, inverse document frequency*) é uma métrica que une essas duas ideias.

No Programa 1.4.4, com a codificação TF-IDF, um termo foi "ignorado": a letra *o*. Isso ocorre porque o método `TfidfVectorizer`[19] usado possui uma propriedade oriunda desse tipo de codificação chamada de normalização. É por esse motivo que esse tipo de codificação também é conhecida como TF-IDF *normalization*. Como citado anteriormente, se um termo aparecer muitas vezes em um texto, ele provavelmente deve ser importante, mas será mesmo? Termos que aparecem em quase todos os textos (como "*o*" ou "*a*", considerados *stopwords*[20], não são particularmente informativos, enquanto os que são distintos ou que aparecem apenas em um pequeno subconjunto são, portanto, importantes. A normalização é efetuada porque o cálculo do TF-IDF pondera um determinado termo tomando como base a sua frequência (quantas vezes ele aparece em um texto), e dividindo-a por uma medida de frequência do documento, que estima a frequência com que o termo aparece em todo o corpus.

Programa 1.4.4. Codificação TF-IDF para os *bigrams* da sentença "*o gato caçou o rato*".

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_encoder = TfidfVectorizer(ngram_range=(1, 2))
# Limite inferior e superior do intervalo de n valores para os
 ↪  diferentes n-grams a serem extraídos (bigrams)

tfidf_ngrams_vectors = tfidf_encoder.fit_transform([sent])

feature_names = tfidf_encoder.get_feature_names_out()
dense = tfidf_ngrams_vectors.todense()

print("Índices correspondentes de cada palavra =
 ↪  {}".format({k: v for k, v in enumerate(feature_names)}))
# Índices correspondentes de cada palavra = {0: 'caçou', 1:
 ↪  'caçou rato', 2: 'gato', 3: 'gato caçou', 4: 'rato'}

print("Codificação por TF-IDF = {}".format(dense.tolist())
# Codificação por TF-IDF = [[0.4472136, 0.4472136, 0.4472136,
 ↪  0.4472136, 0.4472136]]
```

Por isso que no Programa 1.4.4 a letra *o* não foi considerada. Note que o vetor

[19] `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`

[20] Lista de palavras muito comuns de um idioma, mas pouco informativas, que podem ser ignoradas.

resultante possui valores iguais para cada *bigram*. Isso ocorre porque o método foi originalmente pensado para ser aplicado em um conjunto de dados completo, não apenas em uma frase. Veja a diferença da representação obtida no Programa 1.4.5 para um pequeno corpus.

Programa 1.4.5. Codificação TF-IDF para *unigrams* de um corpus exemplo com apenas duas frases. Observe que os termos *"o"* e *"e"* não foram considerados por conta da normalização. As posições dos que possuem valor 0 nos vetores indicam que o *token* não aparece na frase.

```
corpus = ["o gato caçou o rato",
          "gato e cachorros são animais"]

tfidf_encoder = TfidfVectorizer()
tfidf_corpus_vectors = tfidf_encoder.fit_transform(corpus)

feature_names = tfidf_encoder.get_feature_names_out()
dense = tfidf_corpus_vectors.todense()

print("Índices correspondentes de cada palavra =
 ↪  {}".format({k: v for k, v in enumerate(feature_names)}))
# Índices correspondentes de cada palavra = {0: 'animais', 1:
 ↪  'cachorros', 2: 'caçou', 3: 'gato', 4: 'rato', 5: 'são'}

print("Codificação por TF-IDF = {}".format(dense.tolist())
# Codificação por TF-IDF = [
    # [0.0, 0.0, 0.6316672017376245, 0.4494364165239821,
     ↪  0.6316672017376245, 0.0],
    # [0.534046329052269, 0.534046329052269, 0.0,
     ↪  0.37997836159100784, 0.0, 0.534046329052269]]
```

1.4.2. Representação por *Word Embeddings*

Diferentemente da representação por *bag-of-words*, *word embeddings* trabalham com **vocabulários** e consideram a **ordem das palavras**. Tal como abordado inicialmente em **Indexação do Vocabulário**, em PLN, um vocabulário contém palavras pré-selecionadas, as quais podem ser obtidas reunindo todas as palavras únicas de um corpus ou mesmo de um vocabulário preexistente. As palavras do vocabulário ficam dispostas em uma espécie de tabela de consulta (uma matriz, chamada de *look-up table*) que contém vetores de *word embeddings*. Como ilustrado na Figura 1.11, uma *word embedding* pode ser encontrada usando o índice da palavra no vocabulário ao procurar pelo mesmo índice na *look-up table*.

***Out-of-Vocabulary* (OOV).** Imagine que é preciso usar o vocabulário ilustrativo da Figura 1.11 para um corpus apenas sobre plantas. Isso é altamente comum no PLN[21], mas o que acontece caso o nome de uma planta não esteja no vocabulário sobre animais? Para dar conta de palavras desconhecidas, ou seja, que não estão no vocabulário (*out-of-vocabulary* ou OOV), geralmente é reservado um *token* especial `UNK` (de *unknown*, desconhecido).

[21] Justificativas do porque isso acontece serão explicadas mais a frente no texto.

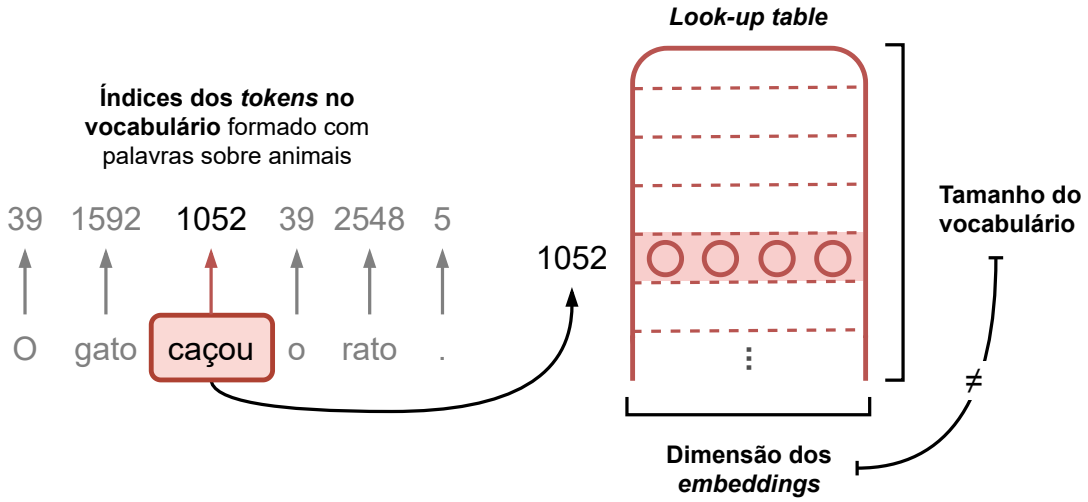

Figura 1.11. Ilustração que representa a relação dos índices de um vocabulário com *word embeddings*. O vocabulário em questão é formado com palavras que possuem nomes, características e comportamentos de animais. Os índices do vocabulário para os *tokens* da frase "*O gato caçou o rato.*" estão elencados à esquerda, juntamente com a palavra *caçou* e seu índice estando em destaque. À direita, é demonstrado que para o índice 1052, que representa a palavra *caçou*, existe uma *word embedding*. Neste cenário, o tamanho do vocabulário não é igual à dimensão (tamanho dos vetores) das *word embeddings*. Adaptado de [Voita 2020].

Alternativamente, *tokens* desconhecidos podem ser ignorados ou atribuídos para *embeddings* compostos apenas por zeros. Contudo, essa não é uma solução adequada para o problema de OOV [Torregrossa et al. 2021].

***Word embeddings* como símbolos discretos.** A maneira mais simples de obter *word embeddings* é representando palavras como vetores *one-hot*. No entanto, para vocabulários grandes, esses vetores também serão grandes: a dimensionalidade/tamanho do vetor é igual ao tamanho do vocabulário. Isso é indesejável na prática, uma vez que fornece vetores muito esparsos.

Neste curso, a codificação *one-hot* foi usada apenas para demostração de exemplos simples de como representar palavras. De agora em diante, o foco será nos tipos de representações mais efetivas e bem estabelecidas encontradas na literatura, uma vez que vetores *one-hot* são agnósticos às palavras.

***Word embeddings* como um espaço vetorial semântico.** Partindo do princípio de que palavras formam um espaço estruturado, isto é, que compartilham informações entre si, a hipótese distributiva (*distributional hyphothesis*) define que palavras que frequentemente aparecem em contextos semelhantes (que possuem um relacionamento semântico) também possuem significados semelhantes [Voita 2020]. Essa hipótese estabelece que capturar significado e capturar contextos são inerentemente a mesma coisa.

Técnicas de modelos de sequência inserem informações sobre contextos em *word embeddings*, e são capazes de identificar semelhanças entre palavras e mapeá-las em um

espaço vetorial de dimensão finita (espaço Euclidiano) [Chollet 2021]. Nesse espaço, a proximidade entre os vetores podem representar relações semânticas e sintáticas entre palavras de forma linear. Por exemplo, na Figura 1.12, a distribuição das palavras *rainha* e *rei* no espaço é (quase) a mesma que entre *mulher* e *homem*.

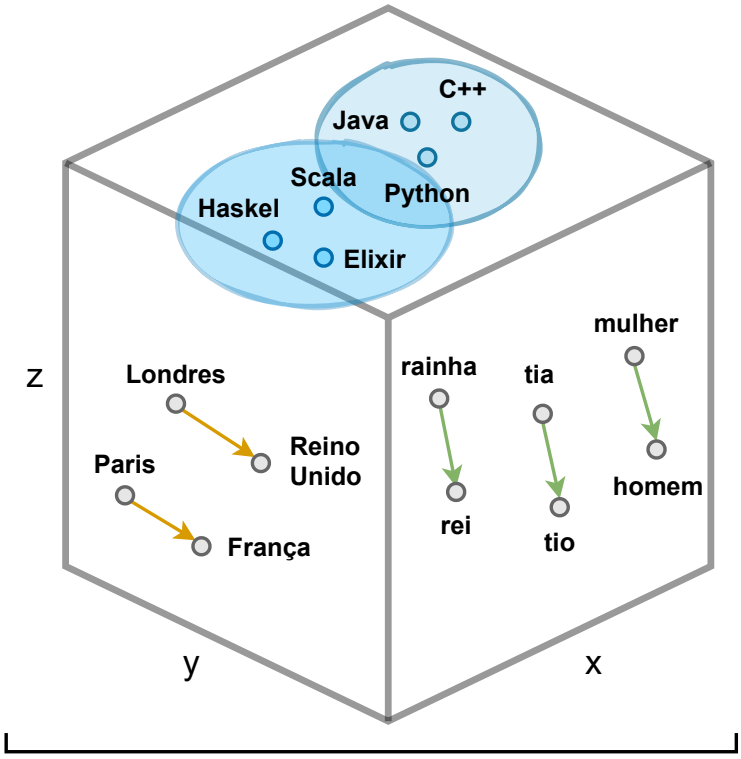

Figura 1.12. *Word embeddings* permitem que o significado seja representado quantitativamente através de um espaço vetorial. Quanto mais semelhante for o significado entre palavras, mais próxima será a representação no espaço, criando na figura, por exemplo, um *cluster* (grupo) de linguagens de *software* (em azul), que por sua vez contém subgrupos de linguagens de Programação Funcional (azul mais escuro) e Programação Orientada a Objetos (azul mais claro). Outra característica das *word embeddings* é que sua distância e direção no espaço mapeiam o significado: neste desenho, os vetores verdes representam o gênero, enquanto que os amarelos representam as relações entre país e capital.

O exemplo *mulher-homem* $\approx$ *rainha-rei* é provavelmente o mais popular, mas também existem muitas outras relações. Na Figura 1.13 estão alguns exemplos de relações similares que formam uma estrutura linear entre *country-capital* (país-capital, diagrama à esquerda) e relações sintáticas (gramaticais, diagrama à direita).

Word embeddings são representações vetoriais de palavras que conseguem exatamente isso: mapeiam a linguagem humana para um espaço geométrico estruturado. A representação obtida são de vetores densos, de baixa dimensionalidade e aprendidos através dos dados. Por fornecerem representações textuais eficientes, modelos que geram *word embeddings* semânticas são atualmente os mais utilizados no PLN. Atente-se apenas

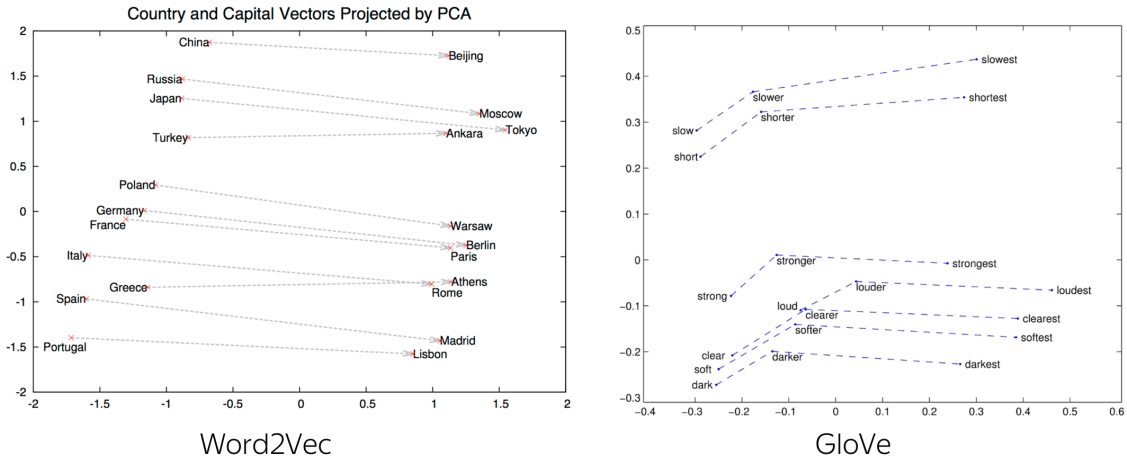

Figura 1.13. Exemplos ilustrativos em 2D de espaços vetoriais que mostram tipos diferentes de relações entre *word embeddings*. À esquerda, pares de palavras *country-capital* (país-capital) semanticamente similares computadas usando a representação do *Word2Vec*. À direita, palavras sintaticamente (variações de uma mesma palavra) similares computadas usando a representação do *GloVe*. Fonte: [Voita 2020].

para o fato de que esses modelos de sequência funcionam melhor quando muitos dados de treinamento estão disponíveis e quando cada texto é relativamente curto.

Os espaços vetoriais formados por *word embeddings* geralmente apresentam milhares de vetores interpretáveis e potencialmente úteis. Para técnicas de *word embeddings* que usam a hipótese distributiva, estas ainda podem ser separadas em dois grandes grupos: *word embeddings* estáticas (ou clássicas), e *word embeddings* contextuais. Antes de entender qual a diferença principal desses dois grupos, saiba que existem duas maneiras de obter *word embeddings*:

- **Aprender *word embeddings* em conjunto com a tarefa principal.** Nessa configuração, os vetores de palavras são iniciados de forma aleatória e, em seguida, aprendidos da mesma forma que são atualizados os pesos de uma rede neural. A Seção 1.6 possui exemplos práticos com problemas reais que demonstram essa forma em específico de gerar *word embeddings*.

- **Carregar *word embeddings* pré-treinadas.** Refere-se aos vetores que são pré-computados usando um modelo diferente do que o usado pela tarefa que se está tentando resolver, como também um corpus diferente com (provavelmente) muitos textos. Os nomes *Word2Vec* e *GloVe* presentes na parte inferior da Figura 1.13 tratam-se de dois modelos prestigiados que geram *word embeddings*. Em particular, vetores pré-treinados serão usados para demonstrarem a diferença entre *word embeddings* estáticas e contextuais nas próximas seções. Por tratar-se de uma forma eficaz e popular de como obter *word embeddings*, esses vetores pré-treinados são uma das principais causas de ocorrem problemas de palavras OOV em domínios específicos, por conta da divergência existente entre vocabulários.

1.4.3. Representação por *Word Embeddings* Estáticas

Como citado anteriormente, é possível aprender *word embeddings* juntamente com a tarefa de PLN, como também carregar vetores provenientes de um espaço de *word embeddings* pré-treinado. A principal lógica por trás do uso de *word embeddings* pré-treinadas no PLN[22] é que o espaço pré-formado é altamente estruturado e compõe-se de aspectos genéricos da estrutura da linguagem. Tais *word embeddings* são geralmente computadas usando estatísticas das ocorrências das palavras (observações sobre quais palavras co-ocorrem em sentenças ou documentos), usando uma variedade de técnicas, algumas envolvendo redes neurais, outras não.

A ideia de um espaço vetorial de *word embeddings* denso e de baixa dimensão, computado de forma não supervisionada, foi inicialmente explorada no início dos anos 2000 [Bengio et al. 2000a]. Embora só começou a popularizar-se em pesquisas e aplicações industriais após o lançamento de um dos mais famosos e bem-sucedidos modelos de *word embeddings*: o algoritmo *Word2Vec*[23] (citado na Figura 1.13), desenvolvido por Tomas Mikolov em 2013 [Mikolov et al. 2013].

Vetores de *word embeddings* pré-treinados estão disponíveis gratuitamente para carregamento e/ou *download* em diversos idiomas. *Word2Vec* é um deles. Outro modelo também renomado e presente na Figura 1.13, chamado de vetores globais para representação de palavras ou apenas GloVe[24], foi desenvolvido por pesquisadores da Universidade de *Stanford* em 2014 [Pennington et al. 2014].

Os vetores pré-treinados usados nesta seção para abordar assuntos previamente apresentados, como *look-up table* e OOV, além de explanar o que são *word embeddings* estáticas, originaram-se a partir de um outro modelo bem estabelecido: o *fastText*. Normalmente, uma *look-up table* atribui cada palavra para um vetor distinto. Por padrão, essas *word embeddings* não têm nenhuma ideia sobre suas sub-palavras. Por exemplo, a palavra *jogar* não "sabe" que possui relação com *jogador*, *jogando*, *jogado*, dentre outras flexões, pois todas as informações que esses vetores têm são do que aprenderam dos contextos. Uma das possíveis abordagens é compor um vetor de palavras a partir de vetores das suas sub-palavras.

Para isso, o modelo *fastText*[25] retorna, além de *word embeddings*, *embeddings* para *n-grams* de caracteres que também estão no vocabulário [Bojanowski et al. 2017]. Fornecer informações sobre sub-palavras possui algumas vantagens: (i) permite que o modelo saiba que diferentes *tokens* podem ser variações da mesma palavra; (ii) ajuda a representar palavras OOV com base em sua ortografia; e (iii) manipula erros ortográficos.

Note no Programa 1.4.6 como vetores pré-treinados com o *fastText* para o Português podem ser carregados facilmente com ajuda da biblioteca *flairNLP*, um *framework* de PLN projetado para facilitar o treinamento e a distribuição de modelos de rotulagem de sequência, classificação de texto e LMs [Akbik et al. 2019]. Inclusive, o *flairNLP* geralmente recomenda o uso de *word embeddings* do *fastText* ou GloVe.

[22]Mais detalhes são explanados na Seção 1.5.
[23]`https://code.google.com/archive/p/word2vec/`
[24]`https://nlp.stanford.edu/projects/glove`
[25]`https://fasttext.cc/`

Programa 1.4.6. Inicilizando *word embeddings* estáticas pré-treinadas do *fastText* para o Português com a biblioteca *flairNLP*.

```
from flair.embeddings import WordEmbeddings

# Inicializando as word embeddings estáticas pré-treinadas
# 'pt' = Português
ftext_embedding = WordEmbeddings('pt')

print("São embeddings estáticos? =
 ↪  {}".format(ftext_embedding.static_embeddings))
print("Tamanho do vocabulário = {}
 ↪  tokens".format(len(ftext_embedding.vocab)))
print("Dimensão dos embeddings =
 ↪  {}".format(ftext_embedding.embedding_length))
print("Tipos dos embeddings =
 ↪  {}".format(ftext_embedding.embedding_type))
# São embeddings estáticos? = True
# Tamanho do vocabulário = 592107 tokens
# Dimensão dos embeddings (tamanho) = 300
# Tipos dos embeddings = word-level
```

Dentro da variável `ftext_embedding` estão as *word embeddings* geradas a partir de milhões de *tokens* em Português, resultantes do corpora (conjunto de corpus) formado pelos dados do Wikipédia e *Common Crawl*[26]. O Programa 1.4.6 também possui algumas informações sobre os vetores carregados: tratam-se de *word embeddings* estáticas, com vocabulário extenso, de mais de 50.000 *tokens*, e com vetores de tamanho 300. É comum ver *word embeddings* estáticas de 256 dimensões, 512 ou até 1.024 dimensões ao lidar com vocabulários muito grandes [Chollet 2021]. O vocabulário do `ftext_embedding` pode ser facilmente acessado com o comando `ftext_embedding.vocab`, em que é retornado um dicionário com todos os *tokens*, iniciando do valor 0. Esse vocabulário serve apenas como base, e os valores das chaves (*tokens*) podem ser facilmente editados.

No *flairNPL*, é simples inicializar uma sentença e atribuir os *embeddings* correspondentes para cada palavra. A última linha de código no Programa 1.4.7 retorna um vetor de tamanho 300 que representa a palavra `gato`. O valor dois (2) reflete que `gato` é o segundo *token* da sentença "*O gato caçou o rato.*". Repare que a sentença não foi padronizada porque o foco aqui é verificar apenas a existência de *word embeddings* e de possíveis termos OOV, e não realizar um procedimento completo para treinamento de um modelo. Deste modo, nenhum dos *tokens* da sentença "*O gato caçou o rato.*" são OOV, ou seja, todos os *tokens* fazem parte do vocabulário do *fastText*.

Mas, afinal, o que são *word embeddings* estáticas? Basicamente, uma ***word embedding* estática** trata-se de uma representação de palavra que não varia entre contextos. Por exemplo, a palavra *bolo* em *"Pesquisei sobre receitas de bolo de laranja"* e *"Há um bolo de coisas sobre a mesa"* tem dois significados diferentes: a primeira indica uma comida e a segunda um amontoado de itens. Porém, uma *word embedding* estática for-

[26] `https://fasttext.cc/docs/en/crawl-vectors.html`

Programa 1.4.7. Trecho de código em que a frase "*O gato caçou o rato.*" é tokenizada e o *embedding* da palavra `gato` é acessado para o *fastText*.

```
from flair.data import Sentence

sent = Sentence("O gato caçou o rato.")

sent.tokens
# [Token[0]: "O",
#  Token[1]: "gato",
#  Token[2]: "caçou",
#  Token[3]: "o",
#  Token[4]: "rato",
#  Token[5]: "."]

ftext_embedding.embed(sent)

sent.get_token(token_id=2).embedding
```

nece a mesma representação para a palavra *bolo* em ambas as frases. O termo "estático" refere-se a esta particularidade: um vetor de palavras é sempre o mesmo, e pode ser *estaticamente* pré-computado e armazenado em uma *look-up table*. O Programa 1.4.8 realiza a comparação da palavra `bolo` nos diferentes contextos e demonstra que a *word embedding* é a mesma.

Sentence embeddings. Uma alternativa a limitação mencionada anteriormente é usar *sentence embeddings* [Feng et al. 2020, Cer et al. 2018, Le and Mikolov 2014] (veja alguns exemplos na Figura 1.14). Tais modelos tomam o texto como entrada, computam suas *word embeddings*, e geram uma saída com um único vetor de tamanho fixo. Isso não só ajuda o modelo a entender o contexto, como também a intenção e outras nuances presentes. Uma estratégia comum é usar uma operação de *pooling* para realizar a média das *word embeddings* de cada palavra presente numa sentença. Como nenhuma das tarefas de PLN abordadas neste curso presentes na Seção 1.6 fazem uso de *sentence embeddings*, para saber detalhes de quais são as possíveis técnicas para obtenção desses tipos de vetores, a biblioteca *flairNLP* ilustra e resume algumas delas[27].

1.4.4. Representação por *Word Embeddings* Contextuais

Sentence embeddings que derivam de representações fixas possuem alguns fatores que podem afetar sua efetividade, como o tamanho da sentença, a ordem das palavras, a presença de palavras que são dispensáveis ao contexto (como *stopwords*), e possíveis vieses presentes nos dados [Torregrossa et al. 2021]. Uma solução para esse problema e para *word embeddings* estáticas é computar representações que possam ser adaptáveis.

Em particular, ***word embeddings* contextuais** possuem propriedades variáveis e são diretamente obtidas a partir do contexto[28]. As informações úteis que formam uma

[27] `https://github.com/flairNLP/flair/blob/master/resources/docs/TUTORIAL_5_DOCUMENT_EMBEDDINGS.md`

[28] O contexto de uma palavra geralmente é composto pelas palavras que a cercam.

Programa 1.4.8. Demonstração simples de como *word embeddings* estáticas não variam entre contextos. Primeiramente, são instanciadas duas sentenças que possuem a palavra `bolo`, mas com significados diferentes: *"Pesquisei sobre receitas de bolo de laranja"* e *"Há um bolo de coisas sobre a mesa"*. Os índices para a palavra `bolo` em ambas as sentenças são cinco (5) e três (3), respetivamente. A partir dos índices, os *embeddings* são acessados e comparados, e é demonstrado que os vetores para a palavra `bolo` são os mesmos.

```
sent_1 = Sentence("Pesquisei sobre receitas de bolo de
 ↪  laranja")
sent_2 = Sentence("Há um bolo de coisas sobre a mesa")

ftext_embedding.embed([sent_1, sent_2])
# [Sentence: "Pesquisei sobre receitas de bolo de laranja",
#  Sentence: "Há um bolo de coisas sobre a mesa"]

sent_1.get_token(5), sent_2.get_token(3)
# (Token[4]: "bolo", Token[2]: "bolo")

bolo_emb_sent_1 = sent_1.get_token(token_id=5).embedding
bolo_emb_sent_2 = sent_2.get_token(token_id=3).embedding

print("Os embeddings são os mesmos? {}".format(bolo_emb_sent_1
 ↪  == bolo_emb_sent_2))
print("Índice da palavra 'bolo' =
 ↪  {}".format(ftext_embedding.vocab["bolo"]))
# Os embeddings são os mesmos? True
# Índice da palavra 'bolo' = 12713
```

word embedding contextual, em geral, são extraídas a partir de *Language Models* e lidam com os três problemas principais das *word embeddings* estáticas [Liu et al. 2019]: (i) *word embeddings* estáticas não tratam a polissemia (vide o exemplo anterior com a palavra polissêmica *bolo*); (ii) *word embeddings* contextuais permitem o uso de representação por *n-grams* de palavras; e (iii) *word embeddings* estáticas assumem que o significado de uma palavra depende de contextos similares, o que não é totalmente correto. Uma observação importante é que *word embeddings* contextuais ainda conseguem projetar palavras em um espaço vetorial semântico e finito.

Word embeddings contextuais são obtidas através de modelos pré-treinados (em sua maioria, *Language Models* pré-treinados), e em vez de representar palavras individualmente, representam palavras juntamente com o contexto em que são usadas. Ao treinar um LM, é obtido muito mais do que treinar apenas *word embeddings* clássicas, já que LMs processam não apenas palavras, mas frases, parágrafos, e textos completos, como ilustrado anteriormente na Figura 1.10 no início da seção sobre **Representação Textual**. Para verificar essa característica intrínseca e poderosa dos LMs, a mesma experimentação feita na Seção 1.4.3 é adaptada para uma das *word embeddings* contextuais mais conceituadas hoje: o modelo BERT.

O *Bidirectional Encoder Representations from Transformers* ou apenas BERT, foi introduzido em [Devlin et al. 2018] e, como o próprio nome diz, usa internamente a ar-

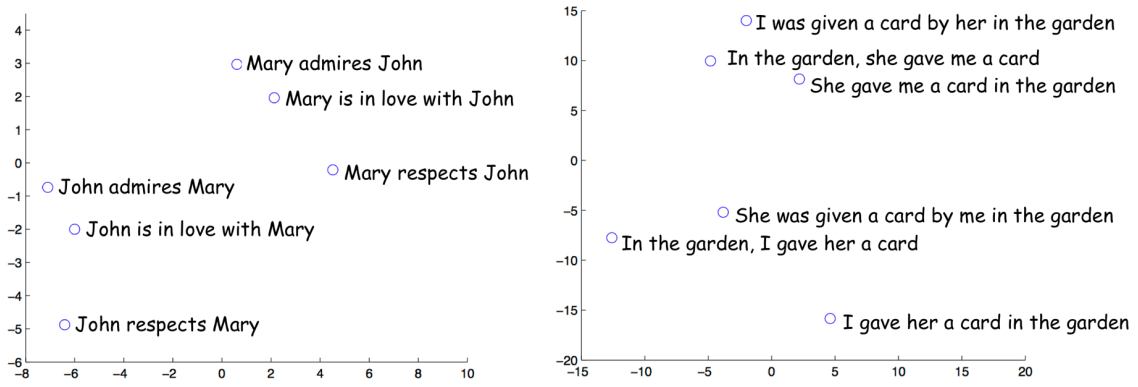

Figura 1.14. Representações das proximidades de *sentence embeddings* com significados semelhantes, mas estruturas diferentes. Fonte: [Voita 2020].

quitetura *Transformer*, um mecanismo de atenção que aprende relações contextuais entre palavras (ou sub-palavras) em textos. Em sua forma simples, o *Transformer* possui uma estrutura de codificador-decodificador. Como o objetivo do BERT é funcionar como um LM, apenas o codificador do *Transformer* é necessário. Ao contrário dos LMs tradicionais, que leem textos sequencialmente (da esquerda para a direita e depois da direita para a esquerda)[29], o codificador *Transformer* lê toda a sequência de palavras de uma só vez. Portanto, é considerado bidirecional, embora seja mais correto dizer que é não direcional.

Para contornar alguns outros problemas dos LMs tradicionais, BERT utiliza duas estratégias de treinamento: *Masked Language Modeling* (MLM ou LM Mascarado) e *Next Sentence Prediction* (NSP). Ambos MLM e NSP são treinados juntos, tendo como objetivo minimizar a função de perda. Por conta dessas características, em contraste com outros LMs, BERT atua não só como um substituto para *word embeddings* estáticas mas, também, dependendo da tarefa, pode ser usado como substituto para toda uma arquitetura específica.

Seguindo os passos do experimento feito anteriormente com *word embeddings* estáticas para as sentenças *"Pesquisei sobre receitas de bolo de laranja"* e *"Há um bolo de coisas sobre a mesa"*, em primeiro lugar, os vetores pré-treinados do BERT foram carregados. No Programa 1.4.9, são carregadas as representações do modelo BERT para o Português Brasileiro, ou BERTimbau[30] [Souza et al. 2020], com a classe `TransformerWordEmbeddings`[31].

[Souza et al. 2020] desenvolveram e disponibilizaram dois modelos BERTimbau: um chamado de *base*, com 12 camadas, dimensão de tamanho 768, 12 *attention heads* e 110 milhões de parâmetros; e o outro chamado de *large*, possuindo 24 camadas, dimensão

[29]Esse processo consiste no tradicional usado por BiRNNs e é importante porque o LM é testado a prever palavras considerando a ordem natural da sentença e depois a sua forma "inversa". Para esses dois caminhos, as representações obtidas das palavras são concatenadas em uma representação final.

[30]`https://github.com/neuralmind-ai/portuguese-bert/`

[31]`https://github.com/flairNLP/flair/blob/master/resources/docs/embeddings/TRANSFORMER_EMBEDDINGS.md`

Programa 1.4.9. Inicializando *word embeddings* contextuais pré-treinadas do BERTimbau *base* para o Português Brasileiro com a biblioteca *flairNLP*.

```
# Observe que a classe carregada é diferente da utilizada para
↪  as word embeddings estáticas
from flair.embeddings import TransformerWordEmbeddings

# Inicializando as word embeddings contextuais pré-treinadas
bert_embedding = TransformerWordEmbeddings(
'neuralmind/bert-base-portuguese-cased')

print("São embeddings estáticos? =
↪  {}".format(bert_embedding.static_embeddings))
print("Dimensão dos embeddings =
↪  {}".format(bert_embedding.embedding_length))
print("Tipos dos embeddings =
↪  {}".format(bert_embedding.embedding_type))
# São embeddings estáticos? = False
# Dimensão dos embeddings = 768
# Tipos dos embeddings = word-level
```

de tamanho 1.024, 16 *attention heads* e 330 milhões de parâmetros[32].

Observe que esses dois modelos possuem um número consideravelmente grande de parâmetros. Ao mesmo tempo, exigiram uma enorme quantidade de dados de treinamento e poderosos recursos de computação para garantir resultados promissores para o Português Brasileiro. BERTimbau foi treinado com o brWaC corpus[33], que contém 2,68 bilhões de *tokens* de mais de 3 milhões de textos, e é o maior corpus em Português disponível [Souza et al. 2020]. Além de seu tamanho, o brWaC é composto de documentos inteiros e sua metodologia garante alta diversidade de domínio e qualidade de conteúdo. A quantidade de parâmetros e dados, e o fato de precisarem de processadores avançados para treino do BERT e de outros LMs, tratam-se das principais limitações existentes hoje para obtenção de *word embeddings* contextuais [Neves Oliveira et al. 2022].

As representações carregadas no Programa 1.4.9 foram do modelo BERTimbau *base* (`bert-base-portuguese-cased`[34]), com dimensão 768. Os vetores resultantes das *word embeddings* dizem respeito a média das saídas das 12 camadas do *base*, conforme explicado na documentação do *flairNLP*. Uma restrição do BERT é que ele processa textos de duas formas distintas, com ou sem capitalização (`cased` ou `uncased`). Dependendo da tarefa, o uso de capitalização pode ser importante, como para o NER, já que entidades (nomes próprios que identificam algo ou alguém) tendem a aparecer com a letra inicial maiúscula. No caso, os modelos BERTimbau são apenas do tipo `cased`. Outra restrição do BERT é que ele apenas processa sentenças de até 512 *tokens*. Isso implica no fato de que ele não processa textos muito extensos.

[32] As variações do BERT em *base* e *large* foram introduzidas em seu *paper* [Devlin et al. 2018]. Em resumo, a diferença entre os dois está no número de camadas do codificador, sendo 12 para o *base* e 24 para o *large*.

[33] `https://www.inf.ufrgs.br/pln/wiki/index.php?title=BrWaC`

[34] `https://huggingface.co/neuralmind/bert-base-portuguese-cased`

Após o carregamento, seguindo o fluxo de experimentação, é verificado a *word embedding* da palavra `gato` presente na sentença "*O gato caçou o rato.*" (veja o Programa 1.4.10), sem padronização. Na verdade, não é recomendado a realização de padronização de texto quando se usa *word embeddings* contextuais, já que é necessário saber qualquer tipo de informação que possua dependência de domínio e que seja útil para predição do próximo termo, desde *stopwords* até mesmo pontuações.

Programa 1.4.10. Trecho de código em que a frase "*O gato caçou o rato.*" é tokenizada e o *embedding* da palavra `gato` é acessado para o BERTimbau *base*.

```
print(sent)
# Sentence: "O gato caçou o rato ."

bert_embedding.embed(sent)
gato_emb = sent.get_token(token_id=2).embedding

print("Tamanho do embedding da palavra gato =
 ↪  {}".format(len(gato_emb)))
# Tamanho do embedding da palavra gato = 768
```

Por fim, o Programa 1.4.11 verifica se as representações para a palavra polissêmica `bolo` são diferentes, o que é retornado que sim: as *word embeddings* são diferentes. Na literatura, os dois vetores distintos da palavra `bolo` são chamados de vetores de conceito (*concept vectors*) porque representam o conceito particular de uma palavra [Torregrossa et al. 2021]. De fato, BERT funciona a nível da palavra (`word-level`, Programa 1.4.9), e pode incorporar informações de sub-palavras, o que implica que lida com palavras OOV. Entretanto, somente se as palavras de contexto forem bem conhecidas, isto é, caso não apareçam raramente no corpus ou corpora usado para treinamento.

Programa 1.4.11. Demonstração simples de como *word embeddings* contextuais variam entre contextos. É demonstrado que os vetores para a palavra polissêmica `bolo` são diferentes para as sentenças *"Pesquisei sobre receitas de bolo de laranja"* e *"Há um bolo de coisas sobre a mesa"*.

```
bert_embedding.embed([sent_1, sent_2])
# [Sentence: "Pesquisei sobre receitas de bolo de laranja",
#  Sentence: "Há um bolo de coisas sobre a mesa"]

sent_1.get_token(5), sent_2.get_token(3)
# (Token[4]: "bolo", Token[2]: "bolo")

bolo_emb_sent_1 = sent_1.get_token(token_id=5).embedding
bolo_emb_sent_2 = sent_2.get_token(token_id=3).embedding

print("Os embeddings são os mesmos? {}".format(bolo_emb_sent_1
 ↪  == bolo_emb_sent_2))
# Os embeddings são os mesmos? False
```

1.5. O Que Faz uma Boa Representação Textual?

O fato de existirem dois grandes grupos principais de *word embeddings* trazem o seguinte questionamento: existe algum algoritmo para criação de *word embeddings* ideal que mapeie perfeitamente a linguagem humana e que possa ser usado para qualquer tarefa do Processamento de Linguagem Natural?

A resposta curta para essa pergunta é *não*. Como existem muitas linguagens diferentes, que não são semelhantes umas às outras, visto que uma língua é o reflexo de uma cultura e um contexto específico, pragmaticamente, o que faz um bom espaço vetorial de *word embeddings* depende muito da tarefa de interesse. A título de exemplo, *word embeddings* que funcionam bem para um modelo de classificação de resenhas de filmes em Inglês podem parecer totalmente diferentes das *word embeddings* ideais para um modelo de classificação de documentos jurídicos em Português, porque a importância de certas relações semânticas varia de tarefa para tarefa.

Isto posto, concluí-se que alguns fatores influenciam intrinsecamente nessa escolha:

- **Contexto ou domínio dos dados.** Realize os seguintes questionamentos: sobre o que é o corpus/corpora? É formado por resenhas de filmes, textos com *fake news*, ou são boletins policiais sobre roubos?
- **Tarefa de PLN.** Que tarefa e, consequentemente, que arquitetura será usada? Dependendo da tarefa, *word embeddings* podem ser unicamente essenciais para codificar as informações pertinentes, como *word embeddings* contextuais, ou podem não possuir por si só a capacidade de extrair informações, necessitando de um processo de codificação extra.
- **Quantidade dos dados.** Qual a quantidade de dados disponível? Caso exista uma quantidade razoavelmente grande de dados, é possível usar diretamente o corpus para gerar *word embeddings* específicas do domínio e, em seguida, aplicá-las na tarefa final. Agora, se não há uma quantidade suficiente de dados, é necessário carregar vetores originados previamente (*word embeddings* pré-treinadas), geralmente de um domínio genérico (como artigos de notícias), que é altamente estruturado e que exibe propriedades úteis ao capturar aspectos genéricos da estrutura da linguagem.

Sobre o último ponto elencado acima, vale destacar que a lógica por trás do uso de *word embeddings* ou modelos pré-treinados no PLN é de que, caso não haja dados disponíveis suficientes que possam ser usados para obter conhecimento linguístico por conta própria sobre os textos de um domínio, espera-se então que as informações necessárias sobre estes textos sejam genéricas. Nesse caso, faz sentido reutilizar o que foi aprendido pelas *word embeddings* pré-treinadas. Porém, deve-se sempre levar em consideração o tipo de corpus ou corpora usado para obter as *word embeddings* pré-treinadas, visto que nem sempre é tão "genérico" assim. Por exemplo, caso o conjunto de dados usado nas *word embeddings* pré-treinadas seja formado inerentemente por artigos de notícias, que de certa forma tendem a ser formais, e pretende-se usar esses vetores em mensagens de

textos que são mais informais, o modelo final pode não performar muito bem, visto que a linguagem é usada de forma distinta em cada um desses cenários.

Também é possível adaptar *word embeddings* pré-treinadas aos dados específicos de tarefas, realizando *fine-tuning* (ajuste) dos *embeddings* treinando-os com toda a rede responsável por resolver a tarefa. Isso pode trazer ganhos de desempenho, embora talvez não enormes [Voita 2020]. Todo esse processo tornou-se comum e possível graças a uma técnica chamada de *Transfer Learning* (Transferência de Aprendizagem). Atualmente, o *Transfer Learning* é a área de PLN mais popular tanto na academia quanto na indústria. A ideia geral por trás dessa técnica é de justamente "transferir conhecimento" de uma tarefa de origem para uma tarefa de interesse/destino [Voita 2020]. Por meio das *word embeddings* pré-treinadas, o conhecimento dos dados de treinamento é "transferido" para um modelo específico de uma certa tarefa.

A Tabela 1.1 descreve os diferentes métodos de representações por *bag-of-words* e *word embeddings* estáticas apresentados até aqui, juntamente com o BERT e outras *word embeddings* contextuais importantes, considerando cinco propriedades principais:

1. **Considera a ordem das palavras.** Como visto na seção sobre **Representação por *Bag-of-Words***, o modelo de *bag-of-bigrams* é bem limitado em relação a ordem das palavras num texto. A auto-atenção é um mecanismo de processamento de conjuntos e foca nas relações entre pares de elementos de sequência, assim, não sabe quais elementos ocorrem no início, no fim ou no meio de uma sequência. Apesar da auto-atenção fazer parte da composição de *Transformers*, e de outros modelos que usam tal arquitetura, como BERT e GPT-3 [Brown et al. 2020], todos estes consideram a ordem das palavras.

2. **Lida com palavras OOV.** O símbolo **—** destaca os métodos que não tratam OOV completamente. De fato, ELMo e BERT tratam palavras OOV apenas se os *tokens* em questão não forem raros.

3. **É contextual.** Propriedade que significa que a representação de uma palavra é obtida olhando para seu contexto, ou seja, quando é uma *word embedding* contextual.

4. **Possui treinamento auto-supervisionado.** Comumente, a tarefa usada para treinar uma *word embedding* é auto-supervisionada. Isso significa que uma tarefa supervisionada é extraída automaticamente de um conjunto de dados não rotulado.

1.6. Além da Representação Textual: Problemas de Aprendizagem Profunda em PLN

Na prática, todas as ferramentas necessárias para lidar com a maioria das tarefas do Processamento de Linguagem Natural foram apresentadas. Contudo, elas não foram vistas em ação. Nesta seção, a experiência é aprofundada a partir da abordagem de problemas mais tradicionais e bem-sucedidos do PLN, apresentados a seguir.

Tabela 1.1. Propriedades alto nível de várias técnicas de representações textuais. Adaptado de [Torregrossa et al. 2021]. Significado dos símbolos: (✓), a propriedade é aplicável à técnica; (—), a propriedade possui certas limitações; (×), a propriedade não é aplicável à técnica.

Técnica de word embedding	**Considera a ordem das palavras**	**Lida com palavras OOV**	**É contextual**	**Possui treinamento auto-supervisionado**
Bag-of-unigrams	×	×	×	×
Bag-of-bigrams	—	×	×	×
GloVe [Pennington et al. 2014]	✓	×	×	✓
Word2Vec [Mikolov et al. 2013]	✓	×	×	✓
fastText [Bojanowski et al. 2017]	✓	✓	×	✓
ELMo [Neumann et al. 2018]	✓	—	✓	✓
Flair [Akbik et al. 2018]	✓	✓	✓	✓
BERT [Devlin et al. 2018]	✓	—	✓	✓
GPT-3 [Brown et al. 2020]	✓	—	✓	✓

1.6.1. *Pipeline* de PLN

O codificador-decodificador é o paradigma da modelagem padrão para tarefas de *sequence-to-sequence*, conforme explicado brevemente na Seção 1.2.1. Um modelo *sequence-to-sequence* usa uma sequência como entrada (geralmente frases ou parágrafos) e a transforma em uma sequência completamente diferente. Essa é a tarefa central de muitas das aplicações mais bem-sucedidas do PLN, como Tradução Automática, Sumarização de Sentenças, *chatbots*, Geração de Textos, etc. A estrutura dessa modelagem consiste em dois componentes principais:

- **Codificador**, que constrói uma representação a partir da sequência de entrada e a passa ao decodificador. Depois que os *tokens* de entrada são mapeados em um espaço vetorial (*word embeddings* ou outros tipos de *embeddings*), os vetores passam por uma camada de codificador para compactar todas as informações em um vetor de comprimento fixo. Normalmente, um codificador é feito de redes do tipo RNN, como LSTMs.
- **Decodificador** usa a representação precedente do codificador para gerar a sequência de saída. A camada do decodificador produz a sequência de saída de *tokens* com base no vetor codificado. Essa camada também é geralmente construída com arquiteturas RNN.

Mesmo que uma tarefa não seja um problema *sequence-to-sequence*, a modelagem ainda é relevante e pode ser utilizada para resolver outros problemas. Por exemplo, a tarefa de NER não se encaixa nesse tipo de abordagem, mas a estrutura de codificador-decodificador é usada para resolvê-la, sendo a saída uma sequência de entidades para cada *token* da sentença de entrada.

A Figura 1.15 demonstra como um *pipeline* de um modelo de Aprendizagem Profunda pode ser construído para o PLN, e deve ser lida de baixo para cima. Inicialmente, os textos de entrada (frases, parágrafos ou documentos completos) precisam ser convertidos em alguma abordagem de representação textual. Essa etapa no *pipeline* é tradicionalmente conhecida como camada de *embedding* (*embedding layer*), e a maioria das aplicações modernas de PLN iniciam com esta camada. Em seguida, a representação em forma

de vetores é passada ao codificador. Boa parte das tarefas simples de PLN terminam aí, como Classificação de Texto, uma vez que, depois do codificador, não é necessário entender mais nada sobre a sequência dos *tokens* de entrada, apenas converter a saída do codificador para um valor fixo (como 0 ou 1). O decodificador torna-se necessário caso seja preciso literalmente decodificar as representações vetoriais aprendidas pelo codificador em textos ou classes.

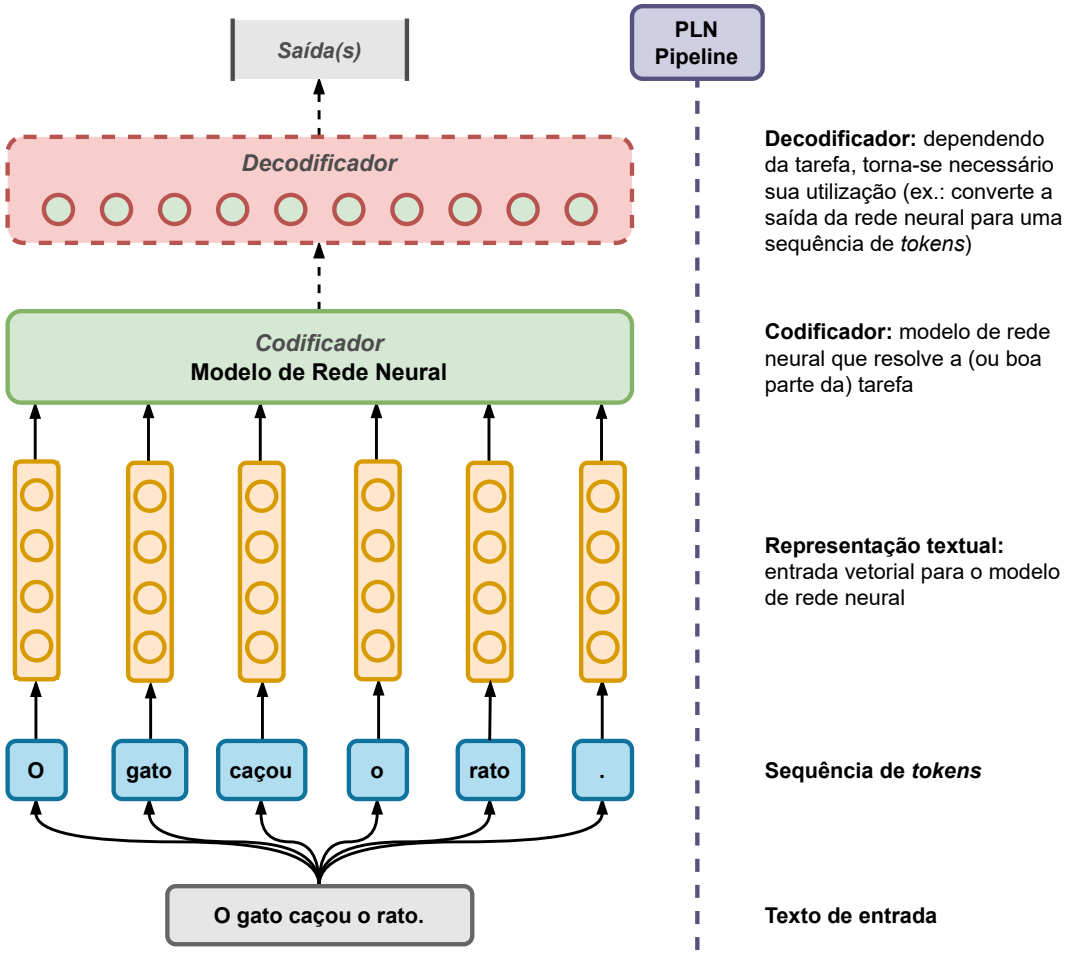

Figura 1.15. *Pipeline* geral de um modelo de Aprendizagem Profunda para o PLN.

Imagine o seguinte: o codificador recebe vetores que possuem internamente informações sobre as relações das palavras e usa-os por intermédio de uma rede neural para entender mais ainda sobre o seu significado. A saída desse codificador ainda serão vetores, e bem mais complexos. Então, como decodificar essa complexidade em algo que não seja vetorial, tal como aquele texto de entrada, e que ainda resolva a tarefa? Por isso que o decodificador é necessário. Um detalhe importante é que um decodificador precisa ser uma rede neural, diferente ou semelhante da usada pelo codificador, ou um modelo de Aprendizagem de Máquina. Isso explica porque, dependendo da tarefa de Classificação de Texto, um decodificador não torna-se necessário, já que não é fundamental o uso de

uma rede neural para converter vetores para valores inteiros.

Além da não utilização obrigatória do decodificador para determinadas tarefas, nem sempre essa estrutura é necessária quando a representação textual escolhida (vide o uso do BERT, explanado na Seção 1.4.4) é autossuficiente para conseguir resolver o problema. Porém, o *pipeline* ainda é válido para caso a representação textual aplicada não consiga obter informações específicas e suficientes sobre os textos de entrada. O *pipeline* apresentado é usado como base para resolução das tarefas de **Classificação de Textos** e **Sumarização de Sentenças**, descritas nas seções a seguir. Em cada tarefa, é destacado o que muda no *pipeline* geral, em relação aos tipos de *embeddings*, a rede que será usada no codificador, se possui um decodificador, etc.

1.6.2. Classificação de Textos

De certa forma, Classificação de Textos é uma das tarefas menos complexas do PLN: dado um texto, existe uma classe que pode ser atribuída a ele. Existem diversos exemplos para esse problema, como dizer se uma opinião sobre um produto é positiva ou negativa, identificar o tema de uma notícia, categorizar uma notícia como verdadeira ou falsa (*fake news*), etc.

Uma das aplicações mais recorrentes para o problema de Classificação de Textos é a Análise de Sentimentos, que busca identificar conteúdo de opinião, e determinar o sentimento, percepção ou atitude do público em relação ao alvo desta opinião. O IMDb (*Internet Movie Database*)[35] é um banco de dados *online* muito utilizado para o treinamento de modelos que resolvem esse tipo de problema. Um dos conjuntos de dados existentes baseados no IMDb é composto por resenhas de filmes em formato de texto e o sentimento associado à uma determinada resenha como classe (1 se for positivo e 0 se negativo).

A Figura 1.16 apresenta uma arquitetura simples para o treinamento de um modelo para a tarefa de Análise de Sentimentos, seguindo o *pipeline* básico da Figura 1.15. A sequência de *tokens* é codificada para *bag-of-bigrams*, em conformidade com o que foi explanado na Seção 1.4.2 e recomendado por [Chollet 2021].

Os *embeddings* gerados com *bag-of-bigrams* são utilizados por uma rede BiLSTM que fará o papel de codificador da resenha inicial. Uma vez que o modelo tenha sido treinado, uma etapa de decodificação é necessária para gerar a saída condizente com o problema inicial: como a definição da tarefa descreve que o sentimento é binário (positivo ou negativo), um *Perceptron* convencional é utilizado; caso a saída definida pertencesse a um conjunto finito de possibilidades, uma MLP poderia ser colocada como decodificador.

Todo o processo de leitura e padronização do conjunto de dados, montagem em código do modelo ilustrado na Figura 1.16, treinamento e avaliação, está documentado e descrito no repositório do curso `https://github.com/barbaraneves/nlp-course-jai22/tree/main/sec06-problemas-pln/classific-de-texto`.

[35] `https://keras.io/api/datasets/imdb/`

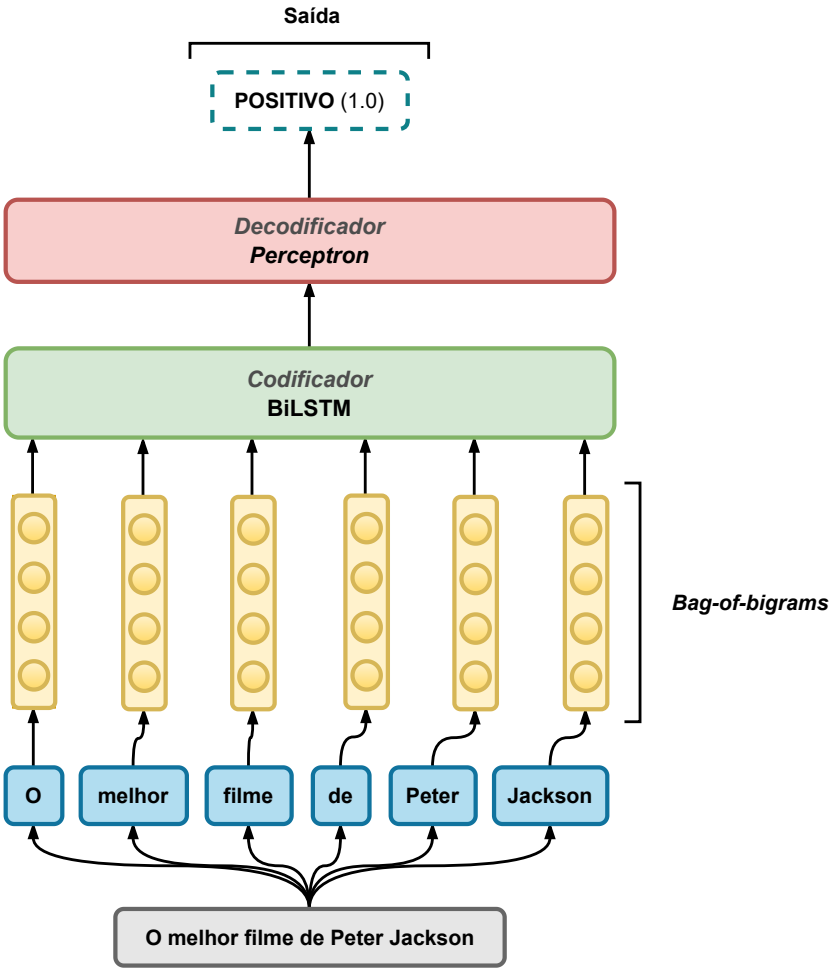

Figura 1.16. Modelo *baseline* para Classificação Binária de Textos. A sentença exemplo "*O melhor filme de Peter Jackson*" é classificada como uma opinião positiva (valor 1.0).

1.6.3. Sumarização de Sentenças

A tarefa de Sumarização de Sentenças [Wang et al. 2017] consiste em criar uma versão reduzida de uma determinada sentença original (com relação ao número de palavras) fornecida como entrada, de tal forma que a compressão gerada obedeça as duas seguintes regras: a sentença gerada deve manter todas as principais informações do texto original e deve ser gramaticamente correta. Chama-se de compressão válida as sentenças geradas que obedecem a essas regras.

Existem dois tipos distintos de sumarização de sentenças: extrativos ou abstrativos [Tas and Kiyani 2007]. A sumarização extrativa permite apenas a operação de remoção de palavras, sem alterar a ordem das palavras na compressão resultante com relação à sentença original. Por exemplo, a frase "*Eu amo muito chocolate branco.*" pode ser comprimida para "*Eu amo chocolate*". Nessa compressão, as palavras *muito* e *branco* foram removidas e a posição das palavras restantes com relação a sentença original foi mantida. A sumarização abstrativa gera compressões por meio da geração de linguagem natural a

partir da sentença original, sem restrições com relação à remoção, ou reordenamento de palavras ou inclusão de novos termos. Por exemplo, a manchete "*Milhares de caxemires cantando slogans pró-Paquistão no domingo participaram de um comício para receber de volta um líder separatista linha-dura que passou por tratamento de câncer em Mumbai.*" pode ter como compressão a sentença "*Milhares participam de comício para linha-dura da Caxemira*".

Com o crescimento do poder computacional nos últimos anos, o volume de textos disponíveis na *Web* e a popularização das redes neurais, diversos modelos de Aprendizagem Profunda foram propostos para resolver a tarefa de Sumarização de Sentenças de forma satisfatória. A Figura 1.17 apresenta uma das possíveis arquiteturas [Wang et al. 2017] que podem ser construídas para a tarefa de compressão de sentenças e que seguem o *pipeline* delineado na Figura 1.15. O conjunto de dados utilizado para resolução da tarefa de compressão é composto por 10 mil pares sentença/compressão extraídos do serviço *Google News* [Filippova et al. 2015].

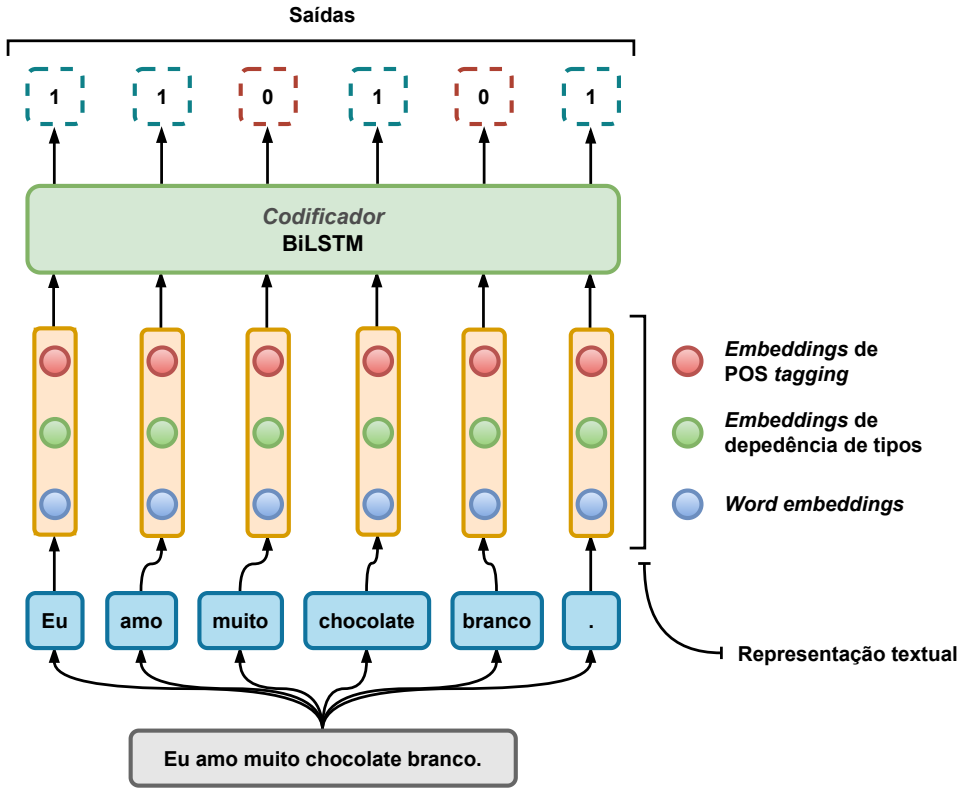

Figura 1.17. Arquitetura para o problema de Sumarização de Sentenças. É verificado como a sentença exemplo "*Eu amo muito chocolate branco.*" pode ser sumarizada: o valor 1 indica que o *token* deve permanecer, e o valor 0 indica que o *token* deve ser omitido da sumarização. Assim, "*Eu amo chocolate.*" trata-se da sentença sumarizada final.

Realizando a leitura de baixo para cima da Figura 1.17, tem-se a sentença "*Eu amo muito chocolate branco.*" que será sumarizada. A sentença passará por um processo de padronização (seguindo o fluxo apresentado na Seção 1.3), gerando uma sequência de *to-*

kens. Cada um dos *tokens* gerados é representado vetorialmente para serem fornecidos ao codificador por uma pilha de *embeddings*, composta por: *word embeddings*, *embeddings* de POS *tagging*, e *embeddings* de tipos de dependências das palavras. O codificador usa uma BiLSTM padrão para processar os vetores de *embeddings* sequencialmente nas duas direções da sentença. Também é responsabilidade do codificador predizer a classe de cada um dos *tokens* da sequência: 0, para caso o *token* deva ser omitido da sumarização, e 1, caso deva permanecer.

Observe que esse problema de sumarização não necessita de um decodificador, apesar de retornar uma sequência de números. Todo o processo de leitura e padronização do conjunto de dados, montagem em código do modelo ilustrado na Figura 1.17, treinamento e avaliação, está documentado e descrito no repositório do curso `https://github.com/barbaraneves/nlp-course-jai22/tree/main/sec06-problemas-pln/sumarizacao-de-sent`.

1.7. Desafios de Pesquisa

Enquanto que o Processamento de Linguagem Natural tenta melhorar continuamente sua capacidade de compreender computacionalmente palavras e sentenças, a linguagem humana é imensamente complexa, fluida e inconsistente, apresentando grandes obstáculos.

Por ser uma área relativamente jovem, há muito espaço dentro do PLN para engenheiros e empresas enfrentarem os inúmeros problemas não resolvidos que surgem com a implantação destes sistemas. Ao longo deste curso foi levantado que na *Web* existe uma espécie de sobrecarga de informações, que representa vastos conjuntos de dados. Porém, isso não ocorre apenas na *Web*.

Setor público. A organização deste setor também possui uma geração de dados textuais contínua. Nos últimos três anos, a Universidade Federal do Ceará por meio do *Insight Data Science Lab* trabalhou em parceria com alguns departamentos de Segurança Pública no Brasil para construir e implantar soluções para problemas relacionados ao PLN. Muitos desses problemas envolviam a análise e entendimento de Boletins de Ocorrência (BOs). A frequência de geração desse tipo de texto, principalmente em locais com alta ocorrência de crimes, torna inviável que profissionais desse domínio leiam e categorizem manualmente todos os documentos relatados.

Os projetos de Pesquisa e Desenvolvimento em parceria com alguns departamentos de Segurança Pública envolviam: (i) tarefa de Reconhecimento de Entidade Nomeada (NER), para extração das informações relevantes das narrativas de BOs, como vítimas, criminosos e testemunhas; (ii) problema de Classificação de Textos a partir das narrativas dos BOs, para categorizar em um tipo de crime, como furto, fraude ou roubo; (iii) e, dado um BO, como identificar outros que apresentem narrativas semelhantes na base de dados da Segurança Pública, de forma que sejam identificados crimes com o mesmo *modus operandi*. [Neves Oliveira et al. 2022, Zschornack R. S. et al. 2020, da Silva et al. 2019] são alguns dos trabalhos publicados nessa temática.

Covid-19. A pandemia da Covid-19 exigiu dos governos medidas eficientes e ágeis para mitigar os efeitos causados pela doença. O Plantão Coronavírus foi uma das soluções desenvolvidas e disponibilizadas para a população no estado do Ceará. A plataforma

possui tecnologia de *chatbot*, permitindo que o usuário mantenha um bate-papo com um sistema de Inteligência Artificial, podendo ser redirecionado para um atendimento virtual com um humano (médico ou enfermeiro).

Ainda voltado para o setor público, o modelo *Sintomatic* [Da Silva et al. 2020] foi desenvolvido para processar textos em linguagem natural provenientes do Plantão Coronavírus e detectando sintomas. Devido a possíveis mutações do vírus e consequente aparecimento de novos sintomas, o *Sintomatic* proporcionou avanços para entender melhor a doença no Ceará. Por possuir a capacidade de reconhecer novos padrões, o modelo identificou uma alta frequência de comportamentos psicológicos alterados, como ansiedade, angústia e tristeza, em usuários positivos ou não para a Covid-19. A ferramenta mostrou, portanto, a necessidade do estado ampliar o atendimento através do canal para o cuidado com a saúde mental da população.

Impacto social. O caso de sucesso mencionado anteriormente sobre o Plantão Coronavírus e o *Sintomatic* foi publicado no livro A Era dos Dados no Setor Público[36], uma parceria da Universidade Federal do Ceará, Íris - Laboratório de Inovação Dados do Governo do Estado do Ceará, *Social Good Brasil* e *AWS Institute*. Outras publicações nessa temática foram [Júnior et al. 2021] e [Mendes de Melo et al. 2021].

Outros desafios. Independente da origem e temática dos textos produzidos, a extração de conhecimento em textos traz vários desafios, tais como: o tempo de desenvolvimento necessário para criação de um sistema de PLN; modelos que devem ser capazes de usar o contexto das sentenças para compreender ambiguidades; dependendo do idioma, modelos de PLN podem ser completamente diferentes; como identificar vieses presentes em textos; e, como tratar palavras que possuem significados diferentes dependendo do contexto.

Esses desafios têm sido amplamente investigados pela comunidade em geral de PLN.

1.8. Considerações Finais

O presente capítulo apresentou como técnicas de Aprendizagem Profunda podem ser utilizadas para resolver diversos tipos de tarefas de Processamento de Linguagem Natural. As seções apresentaram inicialmente os fundamentos necessários sobre Aprendizagem Profunda e como os textos utilizados como fontes de dados devem ser processados para que possa isolar suas características mais importantes. Em seguida, foram mostradas as diversas formas de representar, vetorialmente, textos e suas palavras. Por fim, foram apresentadas duas arquiteturas para resolver os problemas de Classificação de Textos e Sumarização de Sentenças baseadas em um *pipeline* genérico que tarefas de PLN tendem a seguir.

Dada a expressividade da área de Processamento de Linguagem Natural, vários assuntos deixaram de ser abordados neste capítulo. Um dos aspectos que muitos pesquisadores vêm investigando é o treinamento de modelos para tarefas de PLN em domínios específicos e que possuem uma pequena quantidade de dados anotados. Para isso, técnicas de *Transfer Learning* e *Meta-Learning* [Hospedales et al. 2020] têm sido desenvolvidas.

[36] `https://irislab.ce.gov.br/wp-content/uploads/2021/09/LIVRO-DIGITAL-A-Era-dos-Dados-para-o-Setor-Pu%CC%81blico.pdf`

Algumas tarefas importantes também não foram abordadas, como Tradução de Textos, Reconhecimento de Entidades Nomeadas, e suas resoluções com arquiteturas mais modernas de representações textuais, como BERT, ELMo, e *Flair*.

Código do curso. Todos os códigos e ilustrações deste capítulo estão disponíveis no repositório `https://github.com/barbaraneves/nlp-course-jai22`.

Agradecimentos

Este trabalho foi apoiado pela Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico (FUNCAP), projeto Plataforma de Big Data para Acelerar a Transformação Digital do Estado do Ceará sob o número 04772551/2020 e Plataforma Cidadã sob o número 04772314/2020.

Referências

[Akbik et al. 2019] Akbik, A., Bergmann, T., Blythe, D., Rasul, K., Schweter, S., and Vollgraf, R. (2019). FLAIR: An Easy-to-Use Framework for State-of-the-Art NLP. In *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59.

[Akbik et al. 2018] Akbik, A., Blythe, D., and Vollgraf, R. (2018). Contextual String Embeddings for Sequence Labeling. In *Proceedings of the 27th international conference on computational linguistics*, pages 1638–1649.

[Bahdanau et al. 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Bengio et al. 2000a] Bengio, Y., Ducharme, R., and Vincent, P. (2000a). A Neural Probabilistic Language Model. *Advances in Neural Information Processing Systems*, 13.

[Bengio et al. 2000b] Bengio, Y., Ducharme, R., and Vincent, P. (2000b). A neural probabilistic language model. *Advances in Neural Information Processing Systems*, 13.

[Bojanowski et al. 2017] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching Word Vectors with Subword Information. *Transactions of the association for computational linguistics*, 5:135–146.

[Brown et al. 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language Models are Few-Shot Learners. *Advances in neural information processing systems*, 33:1877–1901.

[Cer et al. 2018] Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Céspedes, M., Yuan, S., Tar, C., et al. (2018). Universal sentence encoder. *arXiv preprint arXiv:1803.11175*.

[Chollet 2021] Chollet, F. (2021). *Deep Learning with Python*. Simon and Schuster.

[Da Silva et al. 2020] Da Silva, T. L. C., Ferreira, M. G. F., Magalhaes, R. P., De Macêdo, J. A. F., and da Silva Araújo, N. (2020). Rastreador de sintomas da COVID19. In *SBBD*.

[da Silva et al. 2019] da Silva, T. L. C., Magalhães, R. P., de Macêdo, J. A., Araújo, D., Araújo, N., de Melo, V., Olímpio, P., Rego, P. A., and Neto, A. V. L. (2019). Improving Named Entity Recognition using Deep Learning with Human in the Loop. In *EDBT*, pages 594–597.

[Dai and Le 2015] Dai, A. M. and Le, Q. V. (2015). Semi-Supervised Sequence Learning. *Advances in neural information processing systems*, 28.

[Devlin et al. 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*.

[Feng et al. 2020] Feng, F., Yang, Y., Cer, D., Arivazhagan, N., and Wang, W. (2020). Language-agnostic bert sentence embedding. *arXiv preprint arXiv:2007.01852*.

[Filippova et al. 2015] Filippova, K., Alfonseca, E., Colmenares, C. A., Kaiser, Ł., and Vinyals, O. (2015). Sentence compression by deletion with lstms. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 360–368.

[Géron 2019] Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. "O'Reilly Media, Inc.".

[Hochreiter and Schmidhuber 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hospedales et al. 2020] Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2020). Meta-Learning in Neural Networks: A Survey. *arXiv preprint arXiv:2004.05439*.

[Júnior et al. 2021] Júnior, V. O. D. S., Branco, J. A. C., De Oliveira, M. A., Da Silva, T. L. C., Cruz, L. A., and Magalhaes, R. P. (2021). A Natural Language Understanding Model COVID-19 based for chatbots. In *2021 IEEE 21st International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 1–7. IEEE.

[Le and Mikolov 2014] Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *ICML*, pages 1188–1196. PMLR.

[Liu et al. 2019] Liu, N. F., Gardner, M., Belinkov, Y., Peters, M. E., and Smith, N. A. (2019). Linguistic knowledge and transferability of contextual representations. *arXiv preprint arXiv:1903.08855*.

[Louis 2020] Louis, A. (2020). A Brief History of Natural Language Processing — Part 2.

[McCulloch and Pitts 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[Mendes de Melo et al. 2021] Mendes de Melo, S., Lima Férrer de Almeida, A., Almada Cruz, L., and Linhares Coelho da Silva, T. (2021). A chat recommender system for covid-19 support based in textual sentence embeddings. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 248–252.

[Mikolov et al. 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *arXiv preprint arXiv:1301.3781*.

[Neumann et al. 2018] Neumann, M. P. M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep Contextualized Word Representations. *arXiv preprint arXiv:1802.05365*.

[Neves Oliveira et al. 2022] Neves Oliveira, B. S., Fernandes de Oliveira, A., Monteiro de Lira, V., Linhares Coelho da Silva, T., and Fernandes de Macêdo, J. A. (2022). Held: Hierarchical entity-label disambiguation in named entity recognition task using deep learning. *Intelligent Data Analysis*, 26(3):637–657.

[Pennington et al. 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

[Rosenblatt 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

[Rumelhart et al. 1985] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

[Souza et al. 2020] Souza, F., Nogueira, R., and Lotufo, R. (2020). BERTimbau: Pretrained BERT Models for Brazilian Portuguese. In *Brazilian Conference on Intelligent Systems*, pages 403–417. Springer.

[Sutskever et al. 2014a] Sutskever, I., Vinyals, O., and Le, Q. V. (2014a). Sequence to Sequence Learning with Neural Networks. *Advances in neural information processing systems*, 27.

[Sutskever et al. 2014b] Sutskever, I., Vinyals, O., and Le, Q. V. (2014b). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

[Tas and Kiyani 2007] Tas, O. and Kiyani, F. (2007). A survey automatic text summarization. *PressAcademia Procedia*, 5(1):205–213.

[Torregrossa et al. 2021] Torregrossa, F., Allesiardo, R., Claveau, V., Kooli, N., and Gravier, G. (2021). A survey on training and evaluation of word embeddings. *International Journal of Data Science and Analytics*, 11(2):85–103.

[Vaswani et al. 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is All You Need. In *Advances in neural information processing systems*, pages 5998–6008.

[Voita 2020] Voita, E. (2020). NLP Course For You.

[Wang et al. 2017] Wang, L., Jiang, J., Chieu, H. L., Ong, C. H., Song, D., and Liao, L. (2017). Can syntax help? improving an lstm-based sentence compression model for new domains. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1385–1393.

[Webster and Kit 1992] Webster, J. J. and Kit, C. (1992). Tokenization as the initial phase in NLP. In *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*.

[Zhang et al. 2018] Zhang, L., Wang, S., and Liu, B. (2018). Deep learning for sentiment analysis: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1253.

[Zschornack R. S. et al. 2020] Zschornack R. S., F., Linhares Coelho da Silva, T., and Fernandes de Macêdo, J. A. (2020). Aspect term extraction using deep learning model with minimal feature engineering. In *International Conference on Advanced Information Systems Engineering*, pages 185–198. Springer.