Capítulo 2

Deep Learning para Processamento de Linguagem Natural

Eduardo Soares de Paiva e Fernando Sola Pereira

Abstract

This chapter presents an overview of the main natural language processing techniques currently used. Studies demonstrate evolutions in textual representation and texts processing. For text representation, we demonstrated techniques to transform texts in numerical formats and to capturing semantic and syntactic information of words. Text processing researches also point to the use of deep learning algorithms. Thus, this chapter analyzes the forms of text representation, RNN and CNN networks for natural language processing, transformers architecture and the BERT model.

Resumo

Esse capítulo apresenta uma visão sobre as principais técnicas de processamento de linguagem natural utilizadas atualmente. Estudos demonstram evoluções tanto nas formas de representação textual, quanto no modo de processamento dos textos. Para a representação de texto, apresentamos as técnicas utilizadas para transformar textos em formatos numéricos e para capturar informações semânticas e sintáticas de palavras. Pesquisas em processamento de texto também apontam para o uso de algoritmos de aprendizado profundo. Sendo assim, esse trabalho analisa as formas de representação textual, as redes RNN e CNN para processamento de linguagem natural, a arquitetura transformers e o modelo BERT.

2.1. Introdução

Esse capítulo apresenta uma visão sobre as principais técnicas de processamento de linguagem natural com a utilização de *deep learning*.

Atualmente, a maioria dos dados disponíveis para análise encontra-se em formato textual. Logo, essa é uma área de pesquisa que tem recebido bastante atenção nos últimos anos. As atividades de processamento de linguagem natural podem ser utilizadas

para classificação textual [Paiva et al., 2021], análise de sentimentos [Yadav and Vishwakarma, 2020], tradução automática de textos [Hassan et al., 2018], sumarização de textos [Kosmajac and Kešelj, 2019], dentre outras.

Estudos demonstram evoluções nas formas de representação textual, e no modo de processamento dos textos. Quanto a representação textual, foram desenvolvidas técnicas capazes não só de representar os textos, como também de capturar informações semânticas e sintáticas das palavras [Mikolov et al., 2013b], [Mikolov et al., 2013a], [Pennington et al., 2014], [Devlin et al., 2019].

Já as pesquisas na área de processamento dos textos têm apontado para a utilização de algoritmos de deep learning (redes neurais profundas). Nesse contexto, as redes *Convolutional Neural Network-CNN* [Lecun et al., 1998], *Recurrent Neural Network-RNN* [Elman, 1990] e as baseadas na arquitetura Transformers [Vaswani et al., 2017] têm demonstrado bons resultados.

Sendo assim, esse trabalho faz um levantamento de tais técnicas a fim de fornecer um panorama sobre os principais direcionamentos utilizados no processamento de textos.

O restante desse Capítulo está dividido da seguinte forma: a Seção 2.2 apresenta um estudo sobre representação de palavras. As Seções 2.3, 2.4 e 2.5 descrevem as redes neurais convolucionais, as redes neurais recorrentes e a arquitetura *transformer*, respectivamente. Já a Seção 2.6 demonstra o funcionamento e utilização da arquitetura *Bidirectional Encoder Representations from Transformers* (BERT) e a Seção 2.7 apresenta outros modelos que tentam tratar limitações do modelo BERT. Finalmente, a Seção 2.8 faz uma breve conclusão do estudo.

2.2. Representação de Palavras

Os modelos de redes neurais não trabalham com o texto bruto, eles esperam como entrada vetores numéricos [Chollet, 2017]. Logo, antes de se iniciar o processamento de textos com redes neurais, realiza-se um processo de vetorização textual.

Atualmente, a forma mais usual de se representar palavras como vetores numéricos é pela utilização de *word embeddings*. *Word embedding* é uma forma de representação textual que utiliza vetores densos[1], de baixa dimensionalidade, cujos valores são aprendidos a partir do próprio texto. A *word embedding* utiliza a vizinhança de cada uma das palavras do texto para formular a representação das palavras.

Isso permite a criação de um vetor denso que representa a projeção de cada palavra. Dessa forma, a *word embedding* representa as coordenadas da palavra no espaço vetorial que foi aprendido a partir do texto. Sendo assim, os relacionamentos geométricos entre os vetores de palavras devem refletir os relacionamentos semânticos entre essas palavras [Goyal et al., 2018]. Utilizando-se essa abordagem, pode-se comparar a relação entre duas palavras quaisquer a partir da comparação dos vetores que as representam.

Esse tipo de representação surgiu da necessidade de se expressar as características de similaridade entre as palavras, de forma que isso pudesse ser aproveitado no contexto

[1]Um vetor denso é vetor cujas posições são preenchidas com valores diferentes de zero. Em contrapartida, um esparso é aquele cuja maioria das posições são preenchidas com o valor zero.

das aplicações.

Sendo assim, passou-se a explorar o conceito de modelagem estatística da linguagem. Esse modelo permite a previsão da palavra seguinte, levando-se em consideração as anteriores [Bengio et al., 2000]. Essa ideia é uma extensão dos modelos utilizados no tratamento de séries temporais, pois, da mesma forma que em sistemas lineares, o estado seguinte pode ser determinado pela combinação dos estados anteriores. Logo, pode-se prever a palavra seguinte de um determinado texto, com base nas palavras que ocorreram anteriormente nesse mesmo texto.

Essa constatação já havia sido feita por linguistas, Harris (1954) já afirma que as palavras vizinhas estavam relacionadas semanticamente, logo, palavras semelhantes ocorreriam em contextos semelhantes. Nesse mesmo sentido, Firth (1957) afirma que é possível conhecer uma palavra através das palavras que a acompanham (*"You shall know a word by the company it keeps!"*). Dessa forma, conclui-se que as palavras não ocorrem em contextos independentes, uma palavra sempre está relacionada com as palavras que vêm antes e depois.

Sendo assim, pode-se destacar duas características importantes do modelo de linguagem:

- A probabilidade de se encontrar uma palavra em um determinado texto é função da ocorrência de todas as palavras anteriores;
- Uma palavra sempre está relacionada com seus vizinhos.

Logo, a partir dessas premissas, formulou-se a seguinte expressão, indicada na Equação 1, para representar a probabilidade de ocorrência de uma determinada palavra em uma sequência de dados textuais. Ou seja, para se obter a probabilidade de ocorrência de uma palavra localizada na posição t, deve-se considerar a probabilidade de ocorrência de todas as palavras que apareceram nas $t-1$ posições anteriores (razão pela qual o valor de K, apresentado na Equação 1, varia entre 1 e $t-1$).

$$p(w^{(t)}) = \prod_{k=1}^{k=t-1} p(w^{(k)}|\{w^1,...,w^{k-1}\}) \qquad [Bengioet\ al.,\ 2000] \qquad (1)$$

Logo, a probabilidade de ocorrência de uma determinada palavra em um texto será igual ao produto das probabilidades de ocorrência das palavras que ocorreram antes dela nesse mesmo texto.

No entanto, essa é apenas uma formulação teórica, pois na maioria das situações não é possível utilizar todas as palavras anteriores (desde o início do texto) para se prever a próxima. Então, para tornar essa premissa viável de ser aplicada em situações práticas, faz-se uma aproximação, e em vez de se considerar todas as palavras para o cálculo da probabilidade da palavra seguinte, realiza-se esse cálculo com base em uma janela de tamanho fixo. Ou seja, calcula-se a probabilidade de uma palavra com base nas n palavras anteriores (onde n é o tamanho da janela a ser considerada). Dessa forma, a Equação 1 pode ser simplificada para a Equação 2:

$$p(w^{(t)}) \approx \prod_{k=t-n+1}^{k=t-1} p(w^{(k)}|\{w^{k-1},...,w^{k-n}\}) \qquad [Bengioet\ al.,\ 2000] \qquad (2)$$

Isso permite a modelagem de qualquer palavra do texto em função de um número fixo de parâmetros, o que torna tal representação muito mais concisa. A partir dessa consideração, torna-se possível desenvolver uma série de modelos de *word embeddings* que são utilizados para modelagem textual. Existem vários modelos de *word embeddings* que se utilizam dessa premissa. Esse estudo apresenta dois exemplos desses tipos de modelos: word2Vec [Mikolov et al., 2013b] e [Mikolov et al., 2013a] e GloVe [Pennington et al., 2014].

2.2.1. Word2Vec

Apesar de algumas tentativas anteriores, as primeiras iniciativas que apresentaram resultados satisfatórios, para a representação de palavras como vetores densos, foram as propostas em Mikolov et al. [2013b] e Mikolov et al. [2013a]. Esses trabalhos apresentam a representação distribuída de palavras obtida a partir da utilização de uma rede neural de duas camadas. Tal modelo ficou conhecido como o Word2Vec, dividindo-se em duas variantes: skip-gram e *Continuous Bag of Words* (CBOW).

- **Skip-gram**

O modelo skip-gram [Mikolov et al., 2013a] é a variante do word2vev que prevê o contexto baseado na palavra corrente, ou seja, dada uma palavra de entrada, ele tenta prever as palavras que aparecem antes e depois dessa palavra de entrada. Para isso, utiliza-se uma rede neural de duas camadas que é treinada com um determinado corpus, ou seja, o próprio texto já oferece a entrada (uma palavra central) e os valores de referência que devem ser utilizados no processo de treinamento (palavras vizinhas), constituindo-se assim em um processo auto supervisionado.

Logo, a entrada dessa rede neural é um vetor cujo tamanho é a palavra central, e a saída são as palavras vizinhas que se pretende prever. Nessa situação, a *word embedding* é representada pelos pesos da camada escondida da rede neural.

A Figura 2.1 ilustra a estrutura da rede neural utilizada para encontrar a representação das palavras empregando a variante skip-gram do modelo word2vec, sendo que, nesse caso, V é o tamanho do vocabulário[2], N é a quantidade de neurônios da camada escondida, x e y são vetores *one-hot encoding*[3] que representam as entradas e saídas do modelo, respectivamente, W_{VN} é a matriz de pesos entre a camada de entrada e a camada escondida, onde a i-esima linha da matriz representa o peso correspondente da i-esima palavra do vocabulário, a matriz W'_{VN} representa os pesos entre a camada escondida e a camada de saída e C está relacionado ao tamanho do contexto (quantidade de vizinhos da palavra alvo). Os vetores da word embedings estão contidos na matriz de pesos W_{VN}.

[2] Vocabulário: conjunto de palavras (ou tokens) que devem ser consideradas no processamento do texto.

[3] Vetor *one-hot encoding* é um vetor em que todas as suas posições são preenchidas com o valor zero, exceto a posição da palavra que ele está representando, que é preenchida com o valor 1.

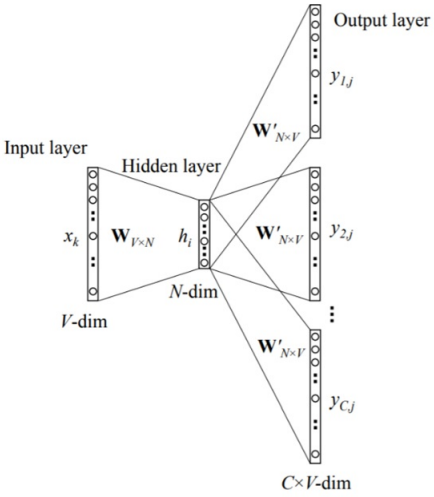

Figura 2.1. Estrutura de Funcionamento do word2vec - skip-gram. Fonte: [Rong, 2014].

Sendo assim, considerando a frase *"esse é um exemplo de sentença"*, dada a palavra central *"exemplo"*, o modelo tenta prever as palavras do contexto: *"esse"*, *"é"*, *"um"*, *"de"* e *"sentença"*.

O algoritmo funciona da seguinte forma: primeiro gera-se o vetor *one-hot encoding* da palavra central, depois obtém-se o vetor de pesos correspondente à palavra em questão (*word embeding* dessa palavra central). O passo seguinte é a geração do *score z*, sendo que esse *score* é obtido pela multiplicação do vetor de entrada pelo vetor de pesos (como a entrada é um vetor esparso, *one-hot encoding*, quando é feita a multiplicação da entrada pela matriz de pesos, apenas a linha correspondente à palavra de entrada em questão é considerada). Posteriormente é feito o cálculo da probabilidade das palavras de saída, para isso, utiliza-se a função softmax[4] ($y = softmax(z)$). Depois, compara-se o vetor de probabilidades geradas com o vetor de probabilidades reais. Isso possibilita a atualização dos pesos pelo método do gradiente e repete-se esse processo de forma iterativa até o fim do treinamento.

A probabilidade, obtida pela aplicação da função softmax, é dada pela Equação 3. Nessa Equação, υ_O indica a representação da palavra de saída ($w^{(t)}$) e μ_I é a representação da palavra de entrada ($w^{(t+k)}$).

$$p(w^{(t+k)}|w^{(t)}) = p(\upsilon_O|\mu_I) = \frac{exp(\upsilon_O^T \mu_I)}{\sum_{j=1}^{M} exp(\upsilon_j^T \mu_I)} \qquad [Mikolovet\ al.,\ 2013b] \quad (3)$$

Logo, obtêm-se a probabilidade de uma determinada palavra $(t+k)$, a partir de uma palavra t, pela utilização da função softmax aplicada ao produto de dois vetores u e v.

[4] A função softmax recebe como entrada um vetor de valores reais e fornece como saída um vetor com valores entre 0 e 1 e a soma desses valores é igual a 1.

- ***Continuous Bag of Words*** **(CBOW)**

O modelo CBOW [Mikolov et al., 2013a] funciona de forma análoga ao skip-gram. No entanto, nessa versão do word2vec a representação da palavra é obtida pelo ajuste dos parâmetros do modelo que tenta fazer a previsão de uma determinada palavra a partir do seu contexto (palavras que aparecem na sua vizinhança), conforme apresentado na Figura 2.2.

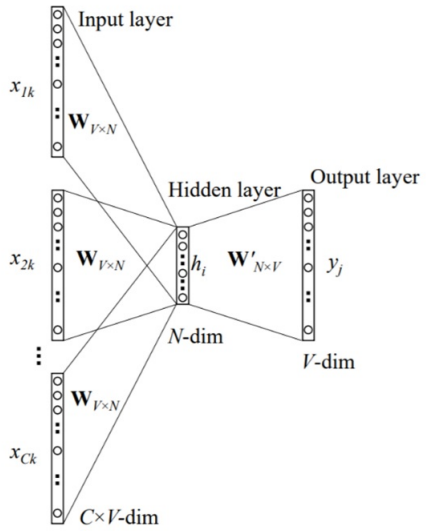

Figura 2.2. Estrutura de Funcionamento do word2Vec – CBOW. Fonte: [Rong, 2014].

Sendo assim, considerando a frase *"esse é um exemplo de sentença"*, dadas as palavras do contexto: *"esse"*, *"é"*, *"um"* e *"de"* e *"sentença"*, o modelo tenta prever a palavra central *"exemplo"*.

Nesse caso, as entradas são vetores *one-hot encoding*, que representam cada uma das palavras de contexto, e a saída é o vetor *one-hot encoding* que faz a previsão da palavra central. Cabe ressaltar que o algoritmo do CBOW funciona de forma análoga ao skip-gram, sendo que, a única diferença é que no caso do CBOW as entradas e saídas da rede neural são invertidas. Ou seja, as palavras de contexto funcionam como entrada da rede neural, e a palavra central é a saída da rede. Logo, essa palavra central funcionará como parâmetro de comparação para se realizar o ajuste dos pesos.

2.2.2. GloVe

O modelo Global Vectors for Word Representation - GloVe [Pennington et al., 2014] trabalha com as probabilidades de coocorrência de palavras em um texto para incorporá-las em vetores significativos. Ou seja, ele considera a frequência em que uma palavra j aparece no contexto de uma palavra i em todo o texto. Dessa forma, considera-se X a matriz de coocorrência e X_{ij} o número de vezes que a palavra j aparece no contexto da palavra i.

A probabilidade de coocorrência de uma palavra j ocorrer com uma palavra i é a razão entre o número de vezes que a palavra j ocorre no contexto da palavra i sobre o

número de vezes que qualquer palavra aparece no contexto da palavra *i*, conforme apresentada na Equação 4.

$$P_{ij} = P(j|i) = \frac{X_{ij}}{\sum_{i \in context} X_{ik}} \qquad [Penningtonet\ al.,\ 2014] \tag{4}$$

Dessa forma, o GloVe verifica a razão entre as probabilidades de coocorrências para extrair o significado interno das palavras. Pennington et al. (2014) utilizam a Tabela 2.1 para exemplificar essa ideia.

Probabilidade e Razão	k = solid	k = gas	k = water	k = fashion
P(k/ice)	$1,9 \times 10^{-4}$	$6,6 \times 10^{-5}$	$3,0 \times 10^{-3}$	$1,7 \times 10^{-5}$
P(k/steam)	$2,2 \times 10^{-5}$	$7,8 \times 10^{-4}$	$2,2 \times 10^{-3}$	$1,8 \times 10^{-5}$
P(k/ice)/P(k/steam)	8,9	$8,5 \times 10^{-2}$	1,36	0,96

Tabela 2.1. Probabilidades de Coocorrências [Pennington et al., 2014]

Pennington et al. (2014) explicam que as duas primeiras linhas da tabela mostram as probabilidades das palavras *"solid"*, *"gas"*, *"water"* e *"fashion"* ocorrerem no contexto das palavras *"ice"* e *"steam"*. Já a última linha mostra a razão de probabilidades.

Para palavras relacionadas a *"ice"*, mas não a *"steam"*, como *"solid"*, a razão é alta. Por outro lado, para palavras relacionadas a *"steam"*, mas não a *"ice"*, como *"gas"*, a razão é baixa e, para palavras relacionadas a ambos ou a nenhum deles, como *"water"* e *"fashion"*, respectivamente, a razão é próxima de 1.

Dado que as razões de probabilidades de coocorrência capturam informações relevantes sobre o relacionamento das palavras, o modelo GloVe visa obter uma função F que prevê essas proporções a partir de dois vetores de palavras w_i e w_j e de um vetor de palavra de contexto w_k, conforme apresentado na Equação 5.

$$F(w_i, w_j, \tilde{w_k}) = \frac{P_{ik}}{P_{jk}} \qquad [Penningtonet\ al.,\ 2014] \tag{5}$$

Sendo assim, o modelo aprende as representações dos vetores de palavras w_i, w_j e w_k para alimentar a função F e prever corretamente as proporções de probabilidades.

O GloVe prediz as palavras circundantes maximizando a probabilidade de uma palavra de contexto ocorrer, a partir de uma palavra central, utilizando para isso uma regressão logística.

2.3. *Convolution Neural Network*

As redes neurais convolucionais (CNNs) são largamente empregadas em atividades de visão computacional, obtendo excelentes resultados em tais tarefas [Krizhevsky et al., 2017], [Han et al., 2018]. Esse sucesso se dá principalmente pelo fato do seu processamento ser realizado de forma convolucional[5], o que é muito eficaz para casos em que os

[5]A convolução é uma operação matemática entre duas funções g e h, produzindo uma terceira função que normalmente é vista como uma versão modificada de uma das funções originais[Díaz-Guerra et al.,

dados tenham alguma relação topológica com os seus vizinhos, como ocorre em imagens. Tal propriedade permite a extração de características que são capazes de definir os dados que são utilizados como entrada.

Essa mesma propriedade, que torna as CNNs tão eficazes para tarefas de visão computacional, também é útil para a extração de características importantes em dados sequenciais, pois, via de regra, os dados pertencentes a uma sequência textual também compartilham características com seus elementos vizinhos. Um texto pode ser visto como uma sequência de palavras. Logo, assim como em uma imagem um pixel compartilha algumas características em comum com os seus pixels vizinhos, as palavras de um texto também guardam algum tipo de relação com as demais palavras que aparecem nas suas proximidades. Sendo assim, pode-se utilizar esse tipo de rede para treinar modelos capazes de desempenhar tarefas de processamento de linguagem natural, conforme pode ser observado em [Kim, 2014], [Zhang et al., 2015] e [Johnson and Zhang, 2015].

No entanto, enquanto nas tarefas de visão computacional os dados são tratados de forma bidimensional (largura e altura da imagem), nas tarefas de Processamento de Linguagem Natural com CNNs, os dados aparecem de forma unidimensional, onde a única dimensão tratada é a dimensão temporal. Logo, cada palavra do texto é considerada como o valor correspondente a um determinado instante de tempo. Sendo assim, para o caso de processamento de textos, ao invés de se utilizar convoluções 2D (como é o caso do processamento de imagens), utiliza-se convolução 1D.

A Figura 2.3 ilustra o processo de convolução 1D para uma determinada sequência de tokens. A convolução começa capturando os primeiros tokens da sequência da entrada, considerando-se o tamanho do kernel do filtro. A partir dessa captura, faz-se a multiplicação desses valores pelos filtros e obtém-se a saída correspondente a essa parte da entrada, sendo que esse processo se repete até que se chegue ao final da sequência de entrada. A CNN unidimensional é invariante para translações (assim como a bidimensional), o que significa que certas sequências podem ser reconhecidas mesmo que apareçam em posições diferentes. Isso pode ser útil para a identificação de determinados padrões no texto.

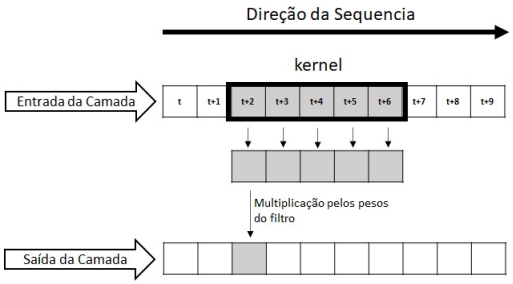

Figura 2.3. Processo de Convolução 1D. Fonte: Elaborada pelos autores.

Essa capacidade das camadas de convolução 1D reconhecerem padrões locais em uma sequência se dá pelo fato de que a mesma transformação de entrada é executada em cada janela. Logo, o padrão aprendido em uma determinada posição de uma sentença

2012].

pode ser reconhecido posteriormente em uma posição diferente. Por essa razão, diz-se que a convolução é invariante a translações. Dessa forma, caso se tenha um filtro sensível[6] a uma determinada sequência de palavras, ele será ativado sempre que tal sequência aparecer no texto, independentemente de sua posição dentro da frase.

Associada à camada de convolução, utiliza-se uma camada de *pooling*, cujo objetivo é diminuir a dimensionalidade dos dados de entrada e tornar o modelo pouco sensível a operações de rotação e de translação, fazendo uma espécie de subamostragem. A operação de *pooling* tem algumas variantes no critério de subamostragem, como por exemplo, *pooling* pela média ou *pooling* pelo valor máximo.

Sendo assim, as camadas de convolução e de *pooling* trabalham de forma combinada, com o objetivo de extrair um conjunto de características capazes de identificar algumas propriedades do conjunto de dados de entrada.

2.4. *Long Short-Term Memory Networks*

Uma das principais características das redes neurais convencionais (*feedforward*) é que elas não têm memória. Cada entrada apresentada a elas é processada independentemente, sem que nenhum estado seja mantido entre as entradas. Dessa forma, para se processar uma sequência de dados, é necessário mostrar a sequência inteira à rede de uma só vez, transformando-a em um único ponto de dado.

No entanto, muitas vezes é importante se ter alguma informação a respeito do que foi processado nos instantes anteriores, para utilizá-la no processamento atual. Essa característica é especialmente importante para o processamento de textos, visto que, muitas das vezes o significado de uma determinada palavra depende das palavras que foram processadas anteriormente.

As redes neurais recorrentes (*recurrent neural network* - RNN) [Elman, 1990] surgiram com o objetivo de preencher essa lacuna. Esse tipo de rede neural processa sequências de dados e mantêm as informações referentes a esse processamento. Nas RNNs há uma espécie de realimentação a partir da sua saída, o que permite com que as variáveis de estado possam ser mantidas para auxiliar nas próximas previsões.

No entanto, as redes neurais recorrentes tradicionais apresentam o problema da dissipação do gradiente (*vanish gradient*) e da explosão do gradiente [Pascanu et al., 2013]. Esse tipo de problema dificulta o processo de treinamento da rede, pois, durante a fase de retropropagação do gradiente, o sinal do gradiente acaba sendo multiplicado várias vezes pela matriz de peso associada às conexões entre os neurônios da camada recorrente. Isso faz com que o valor dessas multiplicações tenda para zero ou para valores muito altos, o que, em ambos os casos, é prejudicial para o processo de treinamento da rede.

Pascanu et al. (2013) demonstram esse problema. Basicamente, se a matriz for formada por valores pequenos (se o autovalor principal da matriz de peso for menor que 1), isso fará com que o valor do gradiente fique tão pequeno a ponto de paralisar o processo de treinamento. Esse fenômeno é conhecido como dissipação do gradiente

[6]Nesse contexto, entende-se como um filtro sensível, aquele que é capaz de capturar algum tipo específico de padrão nos dados que estão sendo processados.

(ou *vanishing gradient*). Por outro lado, se os pesos dessa matriz forem altos (autovalor principal da matriz de peso maior que 1), o sinal de gradiente pode ficar tão grande a ponto de fazer com que o processo de treinamento não convirja. Esse fenômeno é conhecido como explosão do gradiente (ou *exploding gradient*).

Essas limitações motivaram o surgimento das redes LSTM [Hochreiter and Schmidhuber, 1997]. Os neurônios de uma camada LSTM possuem uma célula de memória que é composta por quatro elementos principais: um portão de entrada, um portão com uma conexão auto-recorrente, um portão de esquecimento e um portão de saída. A conexão auto-recorrente tem um peso de 1 e permite que o estado da célula de memória permaneça constante de um passo para outro. Os portões servem para regular as interações entre a própria célula de memória. A porta de entrada pode permitir que o sinal recebido altere o estado da célula de memória ou bloqueie-o. Já a porta de saída pode permitir que o estado da célula de memória tenha efeito sobre outros neurônios ou não. O portão de esquecimento trata a conexão recorrente da célula de memória, permitindo que a célula se lembre ou esqueça do seu estado anterior, conforme necessário. A Figura 2.4 ilustra uma célula LSTM.

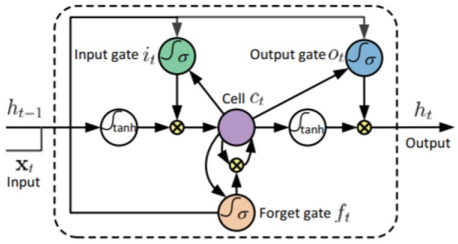

Figura 2.4. Neurônio LSTM. Fonte: [Zhu et al., 2016].

Esse tipo de rede não sofre com os problemas da degradação e explosão do gradiente pelo fato de se gerar um fluxo contínuo para o cálculo do gradiente, pois, o cálculo do gradiente em relação ao espaço interno de memória não é retropropagado para esses parâmetros. Dessa forma, o erro retropropagado passa direto pela camada, não desaparecendo, nem explodindo, para sequências mais longas. Essa característica faz com que essa topologia melhore os resultados quando se precisa de memórias mais longas[7].

Por tais características, as redes LSTMs vêm sendo largamente utilizadas nas tarefas de processamento de linguagem natural, pois a topologia da rede LSTM permite que sejam guardadas algumas informações referentes as palavras anteriores, sem sofrer os problemas de treinamento das RNNs tradicionais, o que pode ser útil para o processamento da palavra atual.

2.5. Arquitetura *Transformer*

Outra arquitetura de rede que tem ganhado relevância no contexto de processamento de linguagem natural é a arquitetura transformer, que é composta por duas partes: um codi-

[7]Entende-se como necessidade de memórias mais longas as situações em que o texto processado é muito grande e as células de memória precisam armazenar muitas informações relativas as palavras anteriores.

ficador e um decodificador.

Porém, o conceito de codificadores e decodificadores é anterior à arquitetura *transformer*, um exemplo disso é a arquitetura proposta em [Cho et al., 2014]. Cho et al. (2014) propõem duas redes neurais recorrentes (RNN) que atuam como um codificador e um decodificador.

O codificador mapeia uma sequência fonte de comprimento variável para um vetor de comprimento fixo e o decodificador mapeia essa representação vetorial de volta para uma sequência alvo de comprimento variável.

Na arquitetura proposta por [Cho et al., 2014], as duas redes (codificador e decodificador) são treinadas em conjunto para maximizar a probabilidade condicional da sequência de destino, dada uma sequência de origem.

Nessa arquitetura, o codificador é responsável por receber sequências textuais e o decodificador utiliza a saída desse codificador para produzir respostas para os textos de entrada. Esse princípio é útil para problemas de tradução automática de textos, pois, nesse tipo de problema, deseja-se mapear uma sequência de palavras (que forma uma sentença) em uma outra sequência de palavras que forma uma sentença com a mesma carga semântica da sequência de entrada, mas escrita em outro idioma.

Utilizando esse princípio de codificadores e decodificadores, Vaswani et al. (2017) propõem uma nova arquitetura, denominada *transformers*, que é centrada em mecanismos de atenção. A Figura 2.5 apresenta a arquitetura *transformers*, sendo que, a parte esquerda representa o codificador e a parte direita representa o decodificador.

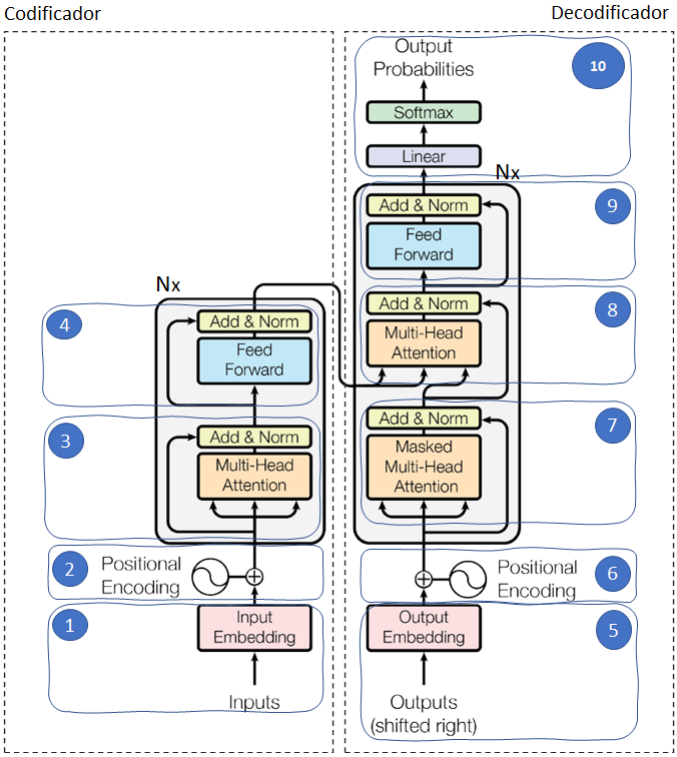

Figura 2.5. Arquitetura Transformer. Fonte: Adaptado de [Vaswani et al., 2017].

A arquitetura apresentada na Figura 2.5 recebe como entrada, na parte inicial do codificador, uma frase inteira, e fornece como saída, na parte final do decodificador a probabilidade das próximas palavras a serem preditas.

Cabe ressaltar que as saídas do decodificador também são utilizadas para retroalimentar o processo, funcionando também como entrada na parte inicial do decodificador. Dessa forma, o decodificador também recebe como entrada uma frase inteira. Ou seja, a frase é passada para o codificador, faz-se uma série de processamentos e depois vai para o decodificador. Então, o decodificador faz as previsões, que retornam as probabilidades de cada uma das palavras. Essas probabilidades são utilizadas para realimentar a entrada do decodificador. Ou seja, as palavras previstas são utilizadas como entrada para novas previsões.

As próximas subseções detalham o funcionamento de cada um dos componentes da arquitetura *transformer*.

2.5.1. *Input Embedding*

A subcamada *Input Embedding* é a camada de entrada do codificador e tem a função de receber as palavras de entrada e convertê-las em vetores. Essa subcamada é representada pela estrutura marcada com o número 1 na Figura 2.5.

2.5.2. *Positional Encoding*

O *Positional Encoding* é utilizado para marcar a posição das palavras dentro das frases. As arquiteturas baseadas em convoluções (CNNs) ou em recorrências (RNNs) não precisam desse tipo de recurso. No caso das convoluções, as palavras sequenciais ficam juntas no mesmo filtro, ou seja, o posicionamento das palavras é mantido. As RNNs também mantêm a ordem das palavras, de acordo com a ordem de entrada dessas palavras. Sendo assim, essas duas arquiteturas permitem a manutenção da ordem sequencial das palavras.

Já na arquitetura *transformer*, faz-se necessário um marcador de posição, que aplica uma fórmula matemática, na matriz de *embbeding*, para que seja possível a identificação da posição de cada palavra no texto.

Vaswani et al. (2017) resolvem essa questão utilizando as funções seno e cosseno para gerar diferentes frequências para a codificação posicional (PE). O *Positional Encoding*, representado pela estrutura de número 2 na arquitetura apresentada na Figura 2.5, cria outra matriz de *embbeding* adicionando um valor posicional, que marca a ordem das palavras. As Equações 6 e 7 indicam como se dá a obtenção do *positional encoding*.

$$PE_{(pos,\ 2i)} = \sin\left(\frac{Pos}{100000^{2i/d_{model}}}\right) \qquad [Vaswaniet\ al.,\ 2017] \tag{6}$$

$$PE_{(pos,\ 2i+1)} = \cos\left(\frac{Pos}{100000^{2i/d_{model}}}\right) \qquad [Vaswaniet\ al.,\ 2017] \tag{7}$$

Como pode ser observado, para as posições pares $(2i)$ utiliza-se a função seno, e para as posições ímpares $(2i+1)$ utiliza-se a função cosseno.

Logo, a função dessa subcamada é adicionar um valor de codificação posicional ao *embedding* de entrada, a fim de possibilitar a recuperação da posição da palavra dentro da frase.

2.5.3. *Multi-Head Attention*

As subcamadas *Multi-Head Attention*, representadas pelos números 3 e 8 na Figura 2.5, recebem como entrada um vetor de embedding que é dividido em três fluxos iguais. Logo, essa camada recebe três entradas, sendo que essas entradas são idênticas e representam os vetores de palavras da frase de entrada.

Esse componente é responsável por implementar o mecanismo de atenção. O mecanismo de atenção é um recurso que permite que uma determinada região do texto receba um "foco" maior, fazendo com que as demais informações fiquem "desfocadas". Vaswani et al. (2017) definem esse tipo de mecanismo como uma função que realiza o mapeamento de uma *Query* (Q) e um conjunto de pares de chave-valor (K,V) para uma saída. A *Query* (Q), as chaves (K), os valores (V) e a saída são matrizes formadas pelos vetores das palavras da frase de entrada. A saída é calculada como uma soma ponderada dos valores, em que o peso atribuído a cada valor é calculado por uma função de compatibilidade da *Query* com a chave correspondente.

Essa atividade é executada por uma estrutura denominada *Scaled dot-product attention*, cuja funcionalidade é descrita abaixo

- ***Scaled dot-product attention***

O *scaled dot-product attention* trata a relação da frase original com suas próprias palavras. Sendo assim, esse mecanismo utiliza duas sentenças iguais A e B, e calcula como cada elemento de A está relacionado a cada elemento de B. Após isso, recombina-se A de acordo com essa relação. Ou seja, esse tipo de mecanismo identifica palavras relacionadas dentro de uma mesma frase.

Sendo assim, a partir de uma sequência A (com um contexto B), obtém-se uma nova sequência na qual cada elemento é uma mistura dos elementos de A que estão relacionados com B. Assim, tem-se uma matriz que faz a combinação da frase com ela mesma.

Dessa forma, um produto escalar indica a similaridade entre dois vetores (no caso duas palavras). Ou seja, considerando u e v dois vetores de *embeddings* que representam duas palavras em uma mesma frase, quanto mais distante u está de v, menor a similaridade entre essas palavras. Da mesma forma, quanto mais próximos estes vetores estiverem, maior é a similaridade entre as palavras. Logo, a partir do produto escalar, realizado para todos os pares de palavras da frase, tem-se a relação existente entre cada par de palavra dessa frase.

A Equação 8 representa a operação executada pelo *scaled dot-product attention*, sendo que, Q, K e V são iguais e representam a frase de entrada.

$$Attention(Q,K,V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad [Vaswaniet\ al.,\ 2017] \qquad (8)$$

A Figura 2.6 representa graficamente a Equação 8. Nesse mecanismo, primeiro faz-se a multiplicação de Q e K (QK^T, sendo que K^T representa a transposta da matriz K). No denominador da função softmax, d indica a dimensão do *embbeding* que representa os tokens. Essa divisão é utilizada para alterar a escala do resultado da multiplicação das matrizes. Posteriormente, aplica-se a função softmax (que fornece um valor entre 0 e 1). Por fim, faz-se a multiplicação pela matriz V.

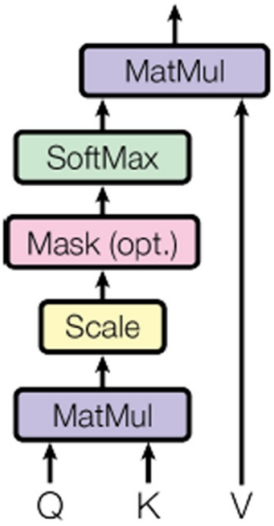

Figura 2.6. *Scaled dot-product attention*. Fonte: [Vaswani et al., 2017].

Na Equação 8, a multiplicação de Q por K indica a autocorrelação das palavras da frase (pois, Q e K representam uma mesma frase). Logo, essa multiplicação retorna uma matriz que indica como as palavras da frase estão relacionadas entre si. O retorno da multiplicação QK é uma matriz, que é multiplicada por V e origina uma nova matriz.

Para exemplificar a forma como o *scaled dot-product attention* funciona, utilizou-se frase: “eu gosto de café”. Considerando que cada uma das palavras está sendo representada por um vetor de 5 posições, os vetores das palavras podem ser ilustrados conforme os vetores apresentados na Figura 2.7(a). Já a frase completa é representada pela junção desses vetores, conforme apresentado na Figura 2.7(b). Ou seja, a frase será representada por uma matriz com 4 linhas (cada uma referente a uma das palavras da frase) e com 5 colunas (cada coluna referente a uma das dimensões utilizadas para representar as palavras). Cabe ressaltar que as 3 entradas do *scaled dot-product attention* são cópias dessa matriz que representa a frase inteira e essas entradas são chamadas de Q, K e V.

A primeira parte do *scaled dot-product attention* faz a multiplicação entre as matrizes Q e K^T. Nessa multiplicação, cada linha da matriz Q é associada com uma coluna da matriz K^T. Ou seja, cada vetor que representa uma palavra na matriz Q é multiplicado

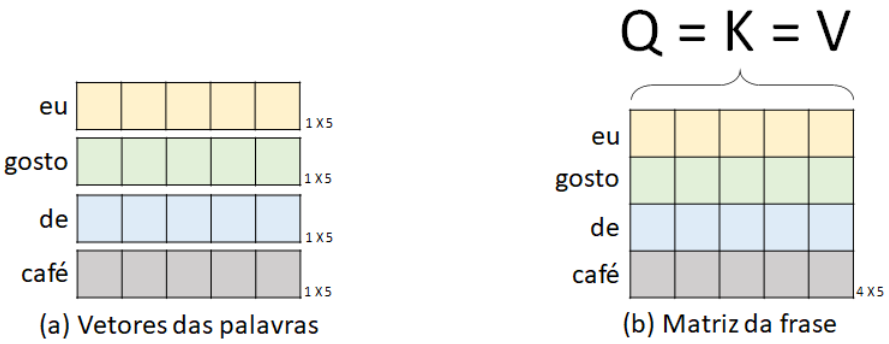

Figura 2.7. Representação da Frase. Fonte: Elaborada pelos autores.

por um vetor que representa uma palavra na matriz K^T, sendo que, o resultado dessa multiplicação expressa o grau de associação de cada uma das palavras com todas as demais palavras que compõem a frase de entrada. Dessa forma, ao final tem-se como resultado uma matriz quadrada, cuja dimensão coincide com o número de palavras da frase. Essa matriz de saída fornece o auto relacionamento das palavras de uma mesma frase. Assim, torna-se possível identificar palavras que possuem ligações maiores e sentidos semelhantes. A Figura 2.8 ilustra esse processo.

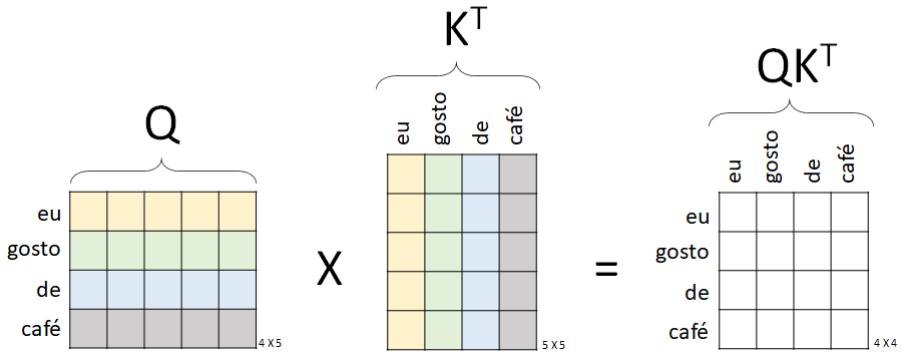

Figura 2.8. Cálculo do relacionamento entre as palavras. Fonte: Elaborada pelos autores.

O resultado da multiplicação é dividido pela raiz da dimensão e aplica-se a função $softmax$, fazendo assim, que todos os valores fiquem compreendidos entre 0 e 1. O passo final é a multiplicação do resultado da função $softmax$ com a matriz V. Ou seja, a saída do *scaled dot-product attention* é uma matriz com as mesmas dimensões da matriz de entrada. A Figura 2.9 ilustra esse processo.

- **Projeções Lineares no *Multi-Head attention Layer***

Vaswani et al. (2017) identificaram que, ao invés de executarem uma única função de atenção com todo o texto de entrada, seria melhor trabalhar com projeções lineares dessa entrada. Sendo assim foi proposta a utilização de um número h de projeções (que são aplicadas nos textos de entrada), conforme demonstrado na Figura 2.10.

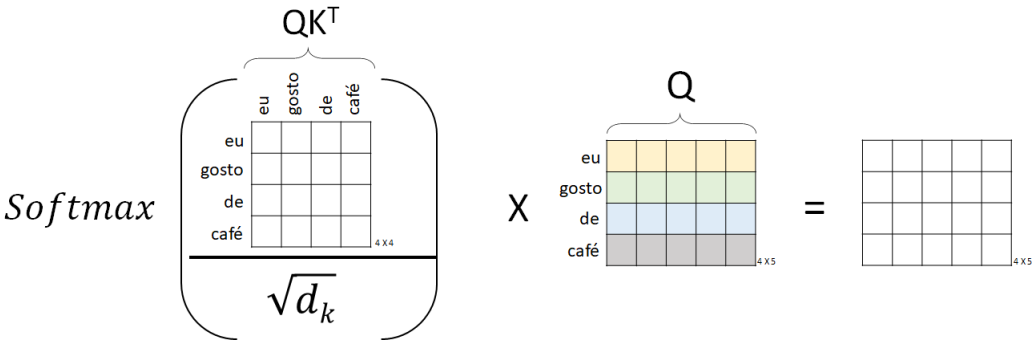

Figura 2.9. Saída do *scaled dot-product attention*. Fonte: Elaborada pelos autores.

O conceito de projeções lineares, permite a aplicação de mecanismos de atenção em diferentes subespaços. Ou seja, diferentes projeções lineares possibilitam o processamento de informações de diferentes subespaços em diferentes posições.

Dessa forma, antes de se aplicar o produto escalar, realiza-se a divisão das entradas em subespaços. O objetivo dessa divisão é fazer com que a relação entre as palavras fique mais forte. Isso ocorre pelo fato do produto escalar ser aplicado em um conjunto menor de palavras. Logo, quando se tem palavras relacionadas, o cálculo de similaridade entre as palavras retorna um valor maior, deixando mais evidente a relação entre as palavras.

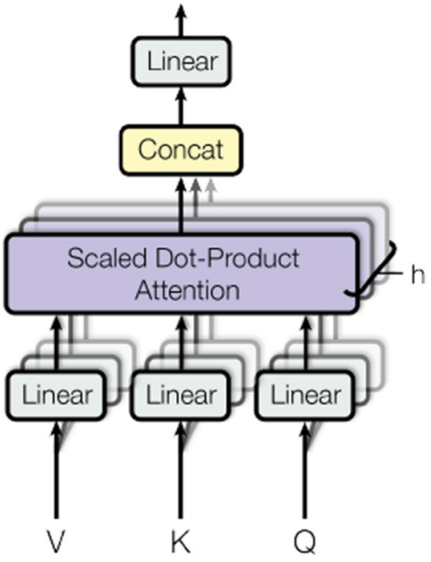

Figura 2.10. Projeções Lineares no *Multi-Head attention Layer*. Fonte: [Vaswani et al., 2017].

Caso essas projeções não fossem realizadas, a relação entre as palavras seria mais fraca, pois mais palavras seriam levadas em consideração durante o cálculo de similaridade.

2.5.4. *Feedforward Network*

As camadas *Feed Forward*, representadas pelo número 4 na Figura 2.5, são responsáveis pela aplicação de duas transformações lineares (duas camadas densas), conforme pode ser observado na Equação 9.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \qquad [Vaswaniet\ al.,\ 2017] \tag{9}$$

2.5.5. Camada de Normalização (*Add & Norm*)

Cada subcamada de atenção e cada subcamada *Feed Forward* do *transformer* é seguida por uma subcamada de normalização (*Add & Norm*). Essas subcamadas aparecem após várias estruturas da arquitetura *transformers*, como pode ser observado nas estruturas representadas pelos números 3, 4, 7, 8 e 9 na Figura 2.5.

O objetivo da camada *Add & Norm* é evitar que as informações da camada anterior sejam esquecidas, o que ajuda durante o processo de *backpropagation*.

Como pode ser observado na arquitetura, a camada *Add & Norm* sempre recebe uma cópia das frases, antes delas entrarem nos mecanismos de atenção (ou nas camadas *Feed Forward*). Isso permite que, além dos valores processados, os valores originais também sejam mantidos. Dessa forma, tem-se a versão original e a versão transformada dos textos.

Essa camada de normalização contém uma função de adição e um processo de normalização de camada. A função *Add* processa as conexões residuais que vêm da entrada da subcamada. Essa operação garante que informações críticas não sejam perdidas. A Equação 10 descreve o funcionamento dessa camada, sendo que, x representa o texto original e $sublayer(x)$ representa a saída da camada imediatamente anterior à camada de normalização (uma camada de atenção ou *Feed Forward*).

$$LayerNorm(x + Sublayer(x)) \qquad [Vaswaniet\ al.,\ 2017] \tag{10}$$

2.5.6. *Masked Multi-head Attention*

O *Masked Multi-head Attention* é uma estrutura que só está presente no decodificador. Essa estrutura funciona de maneira similar ao *Multi-head Attention*, a única diferença é que ela aplica uma máscara na matriz de entrada. A função dessa máscara é mascarar as palavras localizadas após a palavra que está sendo tratada. Dessa forma, a arquitetura aprende a prever essas palavras futuras que foram mascaradas.

Cabe ressaltar que a subcamada *Masked Multi-head Attention*, representada pelo número 7 na Figura 2.5, também é seguida por uma subcamada de normalização, que funciona de forma idêntica à descrita na Subseção 2.5.5.

2.5.7. Última camada Linear e Camada Softmax

Na saída do decodificador há uma última camada densa, essa camada tem o tamanho do vocabulário. Finalmente, o último componente da arquitetura é uma função softmax, que

é aplicada com o objetivo de gerar as probabilidades de cada uma das palavras. Essas estruturas são representadas pelo número 10 na Figura 2.5.

2.6. *Bidirectional Encoder Representations from Transformers* (BERT)

BERT (*Bidirectional Encoder Representations from Transformers*) [Devlin et al., 2019] é um modelo de representação de linguagem baseado na arquitetura *transformer* [Vaswani et al., 2017]. Esse modelo é considerado bidirecional pelo fato das palavras serem representadas com base tanto no contexto à esquerda quanto no contexto à direita. Tais representações são obtidas a partir de treinamentos em bases de dados textuais não rotuladas.

A partir do modelo do BERT, surgiu uma série de modelos que apresentam algumas variantes em relação ao modelo original: ROBERTA [Liu et al., 2019], ALBERT [Lan et al., 2020], DistilBERT [Sanh et al., 2019] e etc. No entanto, este estudo ficará restrito ao BERT.

2.6.1. Modelos Anteriores

Durante o processo de evolução das técnicas de processamento de linguagem natural, muitos estudos apresentaram resultados satisfatórios, inspirando assim o surgimento de novos modelos. Dentre os modelos que precederam o BERT pode-se destacar o Elmo [Peters et al., 2018] e o OpenAI GPT [Radfort et al., 2018].

- **ELMO**

O ELMO [Peters et al., 2018] não implementa o conceito de *transformers*, nem de transferência de aprendizagem. Esse modelo apresenta duas características principais: é considerado pseudo bidirecional e utiliza redes neurais recorrentes.

A arquitetura ELMO é classificada como pseudo bidirecional porque, apesar de considerar tanto o contexto à esquerda quanto o contexto à direita das palavras, ela não considera a sentença inteira, de uma única vez. A sentença não passa por um núcleo único, que processa todo o texto de forma conjunta. Sendo assim, a sentença é dividida em duas partes: uma à esquerda da palavra que está sendo considerada e outra à direita dessa palavra.

A Figura 2.11 ilustra a arquitetura ELMO, sendo que, cada elemento E_i representa uma frase de entrada. Como pode ser observado, cada frase é repassada tanto para a parte esquerda quanto para a parte direita da arquitetura. Ao final do processo, os resultados desses dois núcleos de processamento são concatenados em uma única saída.

Cabe ressaltar que as setas internas nos dois núcleos de processamento, apresentados na Figura 2.11, representam a direção em que as informações são processadas.

O fato dessa arquitetura utilizar, células recorrentes (mais especificamente células LSTM) representa uma desvantagem para esse tipo de processamento. Essa desvantagem ocorre porque as LSTMs fazem sucessivas chamadas recursivas, e à medida que as camadas ficam mais profundas a informação tende a ser perdida.

- **OpenAI GPT**

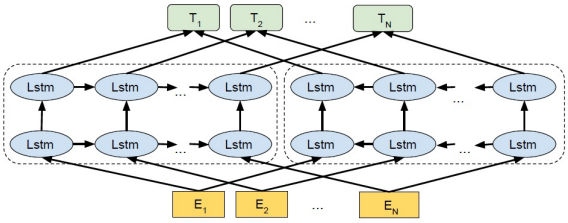

Figura 2.11. Arquitetura ELMO. Fonte: [Devlin et al., 2019].

A arquitetura OPEN AI GPT [Radfort et al., 2018], apresentada na Figura 2.12, é baseada em *transformers*. No entanto, ela só considera o contexto à esquerda dos *tokens*. Sendo assim, a principal deficiência desse modelo é que ele não implementa o conceito bidirecional.

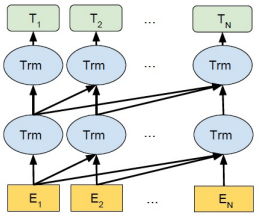

Figura 2.12. Arquitetura GPT-1. Fonte: [Devlin et al., 2019].

2.6.2. Arquitetura BERT

O BERT [Devlin et al., 2019] combina as características dos modelos anteriores: utiliza a arquitetura transformer e é bidirecional. A bidirecionalidade é obtida pelo fato das frases serem processadas tanto em relação aos dados à esquerda quanto à direita. Dessa forma, durante o treinamento, uma frase é passada tanto para a direita quanto para a esquerda, porém em um mesmo centro de processamento.

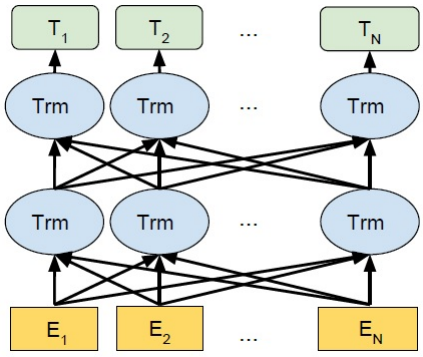

Figura 2.13. Arquitetura BERT. Fonte: [Devlin et al., 2019].

O BERT é composto por uma série de codificadores (*transformers*) empilhados. Dessa forma, o BERT utiliza apenas parte da arquitetura *transformer*. Esses codificadores empilhados permitem que as sentenças sejam codificadas de maneira mais efetiva, possi-

bilitando uma melhor composição dessas sentenças, de acordo com as relações entre as palavras.

O BERT possui duas arquiteturas possíveis, o *BERT BASE* e o *BERT LARGE*, sendo que a principal diferença entre essas arquiteturas é o número de codificadores empilhados. O *BERT BASE* é constituído por uma sequência de 12 codificadores, enquanto o *BERT LARGE* é composto por 24 codificadores empilhadas.

Apesar dessa diferença principal, as variantes do BERT podem ser definidas pela combinação de 3 parâmetros: *L*, *H* e *A*. O parâmetro *L* indica o número de camadas de codificadores. Já o parâmetro *H* indica a dimensão dos *embbedings* gerados (número de unidades na última camada oculta da arquitetura) e o parâmetro *A* indica o número de *self-attentions heads*. A Figura 2.14 apresenta a composição dos modelos *Base* e *Large* do BERT. Cabe ressaltar que o modelo *Base* possui cerca de 110 milhões de parâmetros, enquanto o modelo *Large* 340 milhões.

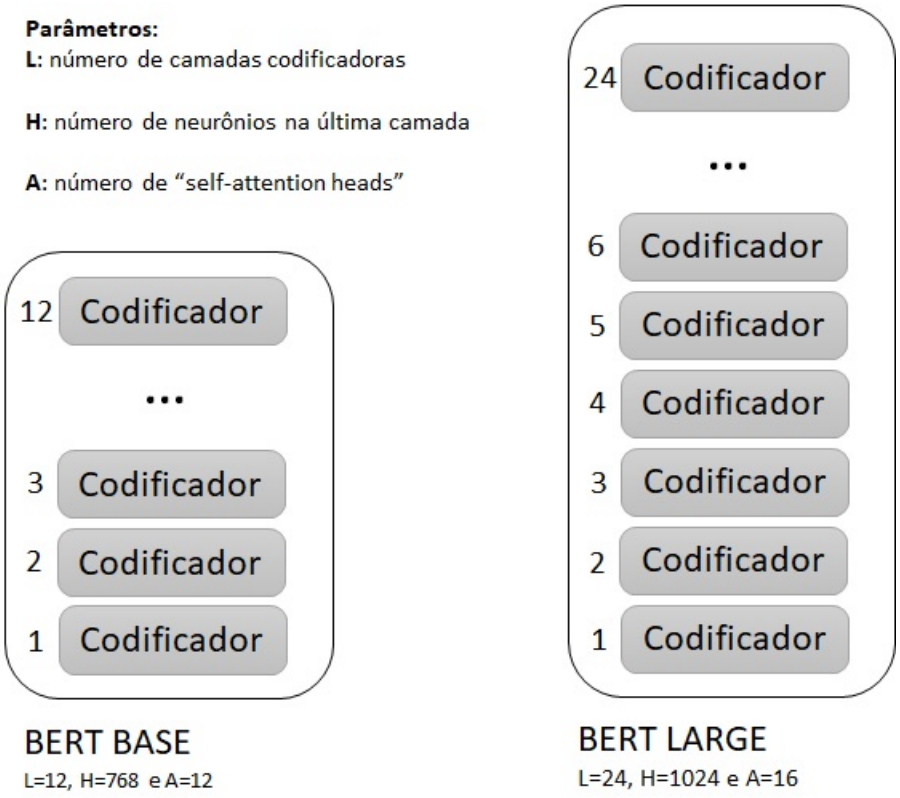

Figura 2.14. composição dos modelos Bert. Fonte: Elaborada pelos autores.

2.6.3. Treinamento do BERT

O modelo original do BERT foi pré-treinado utilizando o *BookCorpus* (corpus com textos de livros com cerca de 800 milhões de palavras) e a Wikipedia em inglês (com cerca de 2,5 milhões de palavras). O pré-treinamento utilizou 16 TPUs[8] e demorou cerca de 3

[8]Tensor Processing Unit-TPU é uma unidade de processamento de tensores desenvolvidas pelo Google para acelerar o processamento em aplicações de deep learning.

dias. Ou seja, esse processo é complexo e exige um grande poder computacional para executá-lo.

O pré-treinamento do BERT é composto por duas tarefas: o *Masked Language Modeling* (MLM) e o *Next Sentence Prediction* (NSP).

- ***Masked Language Modeling* (MLM)**

Na tarefa *Masked Language Modeling*, realiza-se um mascaramento aleatório de uma percentagem do *tokens* de entrada, para que o modelo possa aprender como predizê-los. Dessa forma, substitui-se a palavra a ser predita, colocando-se o *token [MASK]* no lugar, e tenta-se obter o *embedding* desse *token* na saída, conforme ilustrado na Figura 2.15.

Devlin et al. (2019) deixam claro que, diferentemente dos auto-encoders de eliminação de ruído Vincent et al. [2008], o objetivo dessa tarefa não é reconstruir toda a sequência de palavras de entrada, mas apenas fazer a previsão da palavra mascarada.

Cabe ressaltar que, esse token $[MASK]$ não é empregado em todas as iterações, ele é utilizado em apenas 15% das palavras. Ou seja, apenas 15% das palavras são substituídas, sendo que, tal substituição obedece a seguinte regra: em 80% dos casos substitui-se pelo token $[MASK]$, em 10% dos casos substitui-se por um token aleatório, e em 10% dos casos, não se modifica o token.

Devlin et al. (2019) informam que a vantagem desse procedimento é que o codificador do *transformer* não sabe quais palavras ele terá que prever, e nem quais foram substituídas por palavras aleatórias, então ele é forçado a manter uma representação contextual distributiva de cada *token* de entrada. Além disso, como a substituição aleatória ocorre em uma pequena percentagem dos *tokens*, isso não prejudica a capacidade do modelo compreender a linguagem.

A Figura 2.15 ilustra a tarefa de treinamento denominada MLM. Nessa Figura, utiliza-se uma frase como entrada e aplica-se um mascaramento aleatório. Sendo assim, no exemplo em questão, a palavra correr foi substituída pelo token $[MASK]$, logo, deseja-se fazer a previsão dessa palavra. Ao final do processo aplica-se uma camada totalmente conectada (*feed forward neural network*) seguida de uma função softmax, para gerar a probabilidade para cada uma das classes (possíveis *tokens*). Logo, tem-se um problema de classificação em que se retorna uma probabilidade para cada uma dos *tokens* do vocabulário, sendo que, o objetivo é que o *token* mascarado (correr) seja o *token* com maior probabilidade.

- ***Next Sentence Prediction* (NSP)**

A segunda tarefa de treinamento do BERT é a Predição da Próxima Sentença. A entrada consiste em duas frases e o modelo deve dizer se a segunda sentença é sequência da primeira ou não.

Dessa forma, essa tarefa consiste em uma classificação binária, sendo que, sempre que a segunda sentença é sequência da primeira o rótulo é positivo, e no caso contrário o rótulo é negativo.

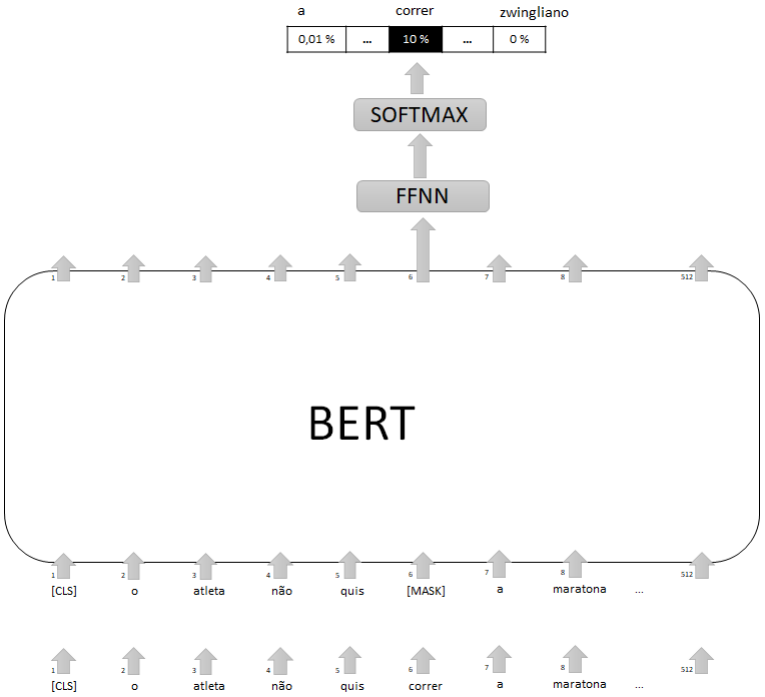

Figura 2.15. Treinamento "Masked Language Modeling". Fonte: Elaborada pelos autores.

A tarefa utiliza dois tokens especiais: $[CLS]$ e $[SEP]$. O $[CLS]$ é um token de classificação binário que é adicionado ao início da primeira frase para prever se a segunda frase é sequência da primeira. Já o $[SEP]$ é um token de separação que sinaliza o fim de uma frase. Sendo assim, a primeira frase é separada da segunda pelo token $[SEP]$.

As duas tarefas (MLM e NSP) são executadas simultaneamente, sendo que, as sequências de entrada devem ser previamente preparadas. A Figura 2.16 ilustra esse preparo.

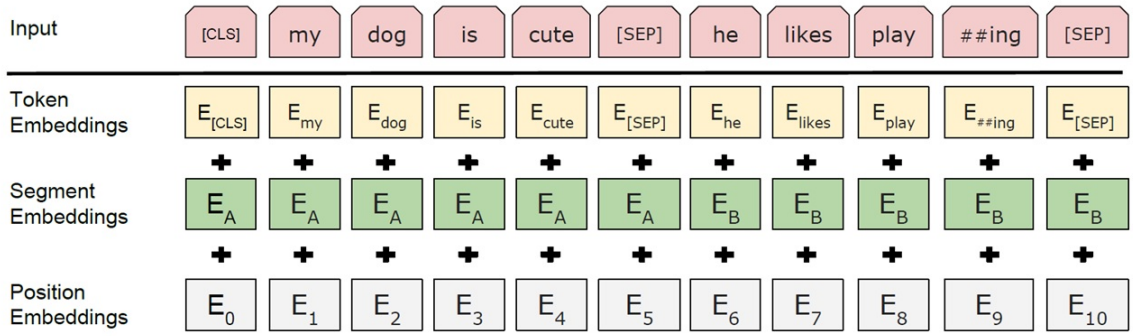

Figura 2.16. Entrada do modelo BERT. Fonte: [Devlin et al., 2019].

Conforme pode ser observado, os *embeddings* de entrada são obtidos pela soma dos *tokens embeddings* com os *segment embeddings* e os *embeddings* de codificação posicional. Essas operações podem ser resumidas da seguinte forma:

1. A sequência de palavras é dividida em *tokens*,

2. Substitui-se aleatoriamente alguns tokens pelo token $[MASK]$.

3. Um *token* de classificação $[CLS]$ é inserido no início de uma das sentenças e um *token* $[SEP]$ separa a primeira sentença da segunda.

4. O *embedding* das sentenças é adicionado ao *embedding* de *tokens*, de modo que a primeira frase tenha um valor de *embedding* diferente da segunda frase.

5. A codificação posicional é aprendida, sendo que, nesse caso não se utiliza a codificação posicional seno-cosseno do *transformer* original.

Na tarefa de NSP, busca-se um entendimento de nível superior, passando do nível de palavras para o nível de frases. Dessa forma, o modelo torna-se adequado para outros tipos de tarefas como por exemplo perguntas e respostas. A Figura 2.17 ilustra esse treinamento.

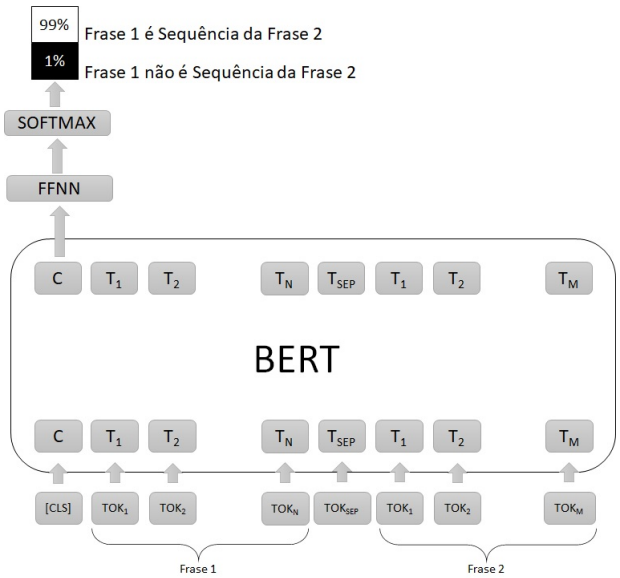

Figura 2.17. Treinamento "Next Sentence Prediction". Fonte: Elaborada pelos autores.

2.6.4. *Fine-tuning*

A grande vantagem do BERT é a possibilidade de transferência de aprendizagem. Dessa forma, uma vez que se tenha o modelo pré-treinado, esse pode ser utilizado em diversas tarefas de PLN, como por exemplo: classificação textual, verificação de semelhança entre sentenças, sistemas de perguntas e respostas e etc.

Para isso, deve-se realizar uma tarefa de *fine-tuning*, a fim de adaptar o modelo original a uma tarefa específica. Nesse sentido, Sun et al. (2019) investigam como maximizar a utilização do BERT para a tarefa de classificação de textos.

O modelo BERT recebe como entrada sequências de 512 *tokens* e fornece como saída vetores de 768 ou 1024 posições[9]. Sendo assim, a tarefa de *fine-tuning* consiste na inclusão de novas camadas ao modelo, a fim de adaptar a estrutura do modelo original ao problema em questão. Durante o *fine-tuning* também é realizado o treinamento com os dados específicos do problema em questão.

Sun et al. (2019) defendem que, quando se adapta o BERT para uma tarefa de PLN em um domínio específico, deve-se buscar uma estratégia de *fine-tuning* apropriada. Dessa forma, os autores sugerem 3 métodos de *fine-tuning*.

- ***Fine-tuning Strategies*:** quando se ajusta o BERT para uma tarefa específica, existem muitas formas de utilizar o BERT. Sun et al. (2019) argumentam que as diferentes camadas do BERT capturam diferentes níveis de informações semânticas e sintáticas. Dessa forma, torna-se necessário saber qual camada é a melhor para uma determinada tarefa. Os autores defendem ainda que, além da definição da camada, deve-se também escolher de forma adequada a taxa de aprendizagem e a estratégia de otimização.
- **Pré-treinamento adicional:** o BERT é treinado em textos de domínio geral (genéricos), com distribuições de dados diferentes da distribuição dos dados dos textos da tarefa de domínio específico, que se deseja adaptar. Sendo assim, Sun et al. (2019) sugerem um pré-treinamento adicional, com os dados referentes ao domínio da tarefa específica.
- ***Multi-task fine-tuning*:** Sun et al. (2019) defendem que, quando há várias tarefas disponíveis em um domínio específico, uma questão interessante é verificar se há benefícios no *fine-tuning* do BERT em todas as tarefas simultaneamente.

Sun et al. (2019) citam a limitação do BERT de trabalhar com tamanhos máximos de textos de até 512 tokens e sugerem dois métodos para solucionar essa dificuldade: o método do truncamento e o método hierárquico.

O método do truncamento consiste em truncar o texto de forma que esse passe a ter apenas 510 tokens (reserva-se dois tokens, para indicar o início e o final do texto). Para esse truncamento, Sun et al. (2019) citam 3 estratégias distintas: manter os 510 tokens iniciais, manter os 510 tokens finais ou seleciona os 128 primeiros tokens e os 382 tokens finais.

Nos métodos hierárquicos, Sun et al. (2019) propõem a divisão do texto de entrada em grupos de 510 tokens. Dessa forma, torna-se possível apresentar ao modelo cada um desses grupos de forma separada. Assim, obtém-se a representação de cada um desses grupos (de 510 tokens) e em seguida combina-se essas representações a fim de se obter uma representação final.

Sun et al. (2019) também citam o problema do esquecimento catastrófico, apresentado por McCloskey and Cohen (1989) . Esse tipo de problema ocorre em situações

[9] O modelo *BERT BASE* possui 768 neurônios na última camada oculta, e por tal razão a sua saída é um vetor de 768 valores. Já o *BERT LARGE*, apresenta com saída um vetor de 1024 posições, visto que, ele possui 1024 neurônios na última camada oculta.

de transferência de aprendizado, em que o conhecimento pré-treinado é apagado durante a aprendizagem de novos conhecimentos.

2.7. Outros Modelos

Um dos grandes avanços introduzidos pela arquitetura *transformers* foi a proposta de um mecanismo de atenção, que pode ser avaliado em paralelo para cada *token* da sequência de entrada, eliminando assim a dependência sequencial que ocorria em redes neurais recorrentes, como a LSTM.

No entanto, a limitação desse mecanismo é seu alto custo computacional. O fato de todos os *tokens* se relacionarem com todos os demais *tokens* da sequência torna quadrática a complexidade desse mecanismo (complexidade $O(n^2)$) .

Por tal razão, modelos derivados da arquitetura *transformers* possuem tamanhos máximos de entrada bem restritos. Um exemplo disso é o modelo BERT, visto anteriormente, que só trabalha com textos de até 512 *tokens*.

Sendo assim, passou-se a pesquisar formas de se alcançar os benefícios trazidos pelos mecanismos de atenção, introduzidos pela arquitetura *transformers*, sem o alto custo computacional.

Nesse sentido, foram propostas algumas soluções que diminuíam a complexidade do mecanismo de atenção. Dentre essas propostas estão as arquiteturas *Longformer* [Beltagy et al., 2020] e *Big Bird* [Zaheer et al., 2020], que possuem complexidade linear e por isso aceitam textos maiores.

2.7.1. *Longformer: The Long-Document Transformer*

Beltagy et al. (2020) definem alguns padrões de atenção que possibilitam a matriz de atenção escalar linearmente com o tamanho da sequência de entrada. Os padrões de atenção propostos foram a janela deslizante, a janela deslizante dilatada e a atenção global, que são apresentados na Figura 2.18.

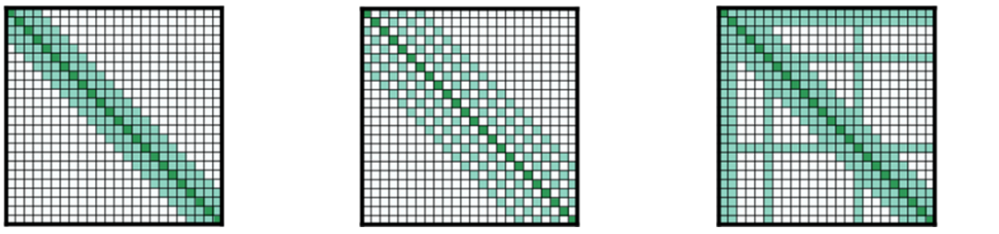

Figura 2.18. Padrões de atenção da arquitetura LongFormer. Fonte: Adaptado de [Beltagy et al., 2020]

A janela deslizante é o padrão que considera um tamanho de janela arbitrário w, e cada token se relaciona apenas com os w tokens mais próximos dele ($w/2$ à esquerda e $w/2$ à direita). Essa modificação faz com que a complexidade do mecanismo de atenção caia para $O(n)$. Ou seja, passa-se de uma complexidade quadrática para uma complexidade linear.

Os autores defendem a utilização de diferentes valores de w para camadas distintas, pois isso pode propiciar um equilíbrio entre a eficiência e a capacidade de representação do modelo.

Beltagy et al. (2020) também propõem a janela deslizante dilatada, com o objetivo de aumentar o campo receptivo sem aumentar a computação. Essa janela funciona de forma análoga a citada anteriormente, porém, com lacunas de dilatação.

Dessa forma, torna-se possível atingir um número maior de tokens, sem prejudicar a complexidade do algoritmo. Isso ocorre porque as lacunas deixam os campos respectivos mais amplos, mantendo o número de operações matemáticas.

Apesar das soluções anteriores resolverem o problema da complexidade do mecanismo de atenção, elas não têm condições de tratar dependências de longa duração. Ou seja, elas não permitem a identificação de relacionamentos entre *tokens* que estejam muito distantes no texto.

Sendo assim, para tratar esse tipo de situação, os autores propõem também a utilização do mecanismo de atenção tradicional. No entanto, a ideia não é calcular a relação entre todos os *tokens*, como acontece em [Vaswani et al., 2017]. Nesse caso, apenas alguns *tokens* pré-selecionados tem a sua atenção totalmente calculada.

Os autores ressaltam que essa operação de atenção é simétrica: ou seja, um *token* com uma atenção global atende a todos os *tokens* na sequência e todos os *tokens* na sequência atendem a ele.

Porém, como o número desses *tokens* selecionados não depende do tamanho da sequência (ou seja, a quantidade de *tokens* que recebem a atenção global não varia com o tamanho de n), a complexidade desse mecanismo de atenção continua sendo $O(n)$.

Sendo assim, o *Longformer* trabalha com mecanismos de atenção que combinam janelas deslizantes com a aplicação de atenção global para alguns *tokens* selecionados. Essa combinação consegue resultados semelhantes aos dos *transformers* originais, porém, com complexidade bem menor.

2.7.2. *Big Bird: Transformers for Longer Sequences*

Zaheer et al. (2020) também propõem um mecanismo de atenção esparsa que reduz a complexidade quadrática dos mecanismos originais para uma complexidade linear. Esse modelo pode ser entendido como uma combinação de 3 tipos de mecanismos de atenção: atenção aleatória, janela de atenção e atenção global. A Figura 2.19 ilustra o funcionamento desses mecanismos.

A atenção aleatória é uma proposta em que um determinado *token* só se conecta com alguns *tokens* aleatoriamente selecionados, ao invés de se conectar com todos os demais *tokens* da sequência, como ocorre no mecanismo de atenção original.

Sendo assim, supondo que sejam selecionados r *tokens* para se conectarem com um determinado *token* (por exemplo $r = 2$), a complexidade deixa de ser $O(n^2)$ e passa a ser $O(r * n)$. Porém, como r é um valor constante, pode-se dizer que a complexidade é $O(n)$.

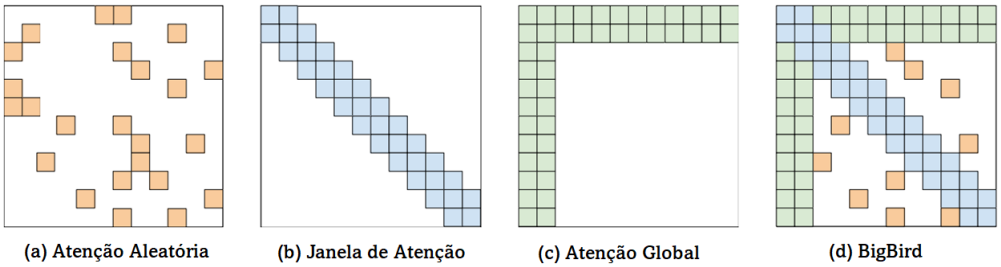

Figura 2.19. Mecanismo de atenção do BigBird. Fonte: Adaptado de [Zaheer et al., 2020]

Ou seja, para cada *token* da sequência seleciona-se um número aleatório de *tokens* com quem ele irá se relacionar, sendo que esse número aleatório é fixo, não dependendo assim do tamanho da sequência.

Já a janela aleatória funciona de maneira análoga à janela deslizante proposta por Beltagy et al. [2020]. Nesse tipo de mecanismo, cada *token* se comunica com um número fixo w de vizinhos mais próximos. Logo, a complexidade passa de $O(n^2)$ para $O(w * n)$. No entanto, como w também é fixo, pode se dizer que a complexidade é $O(n)$.

Por fim, na atenção global, seleciona-se alguns *tokens* que se conectam a todos os demais *tokens*. Ou seja, essa estrutura também é similar à empregada em [Beltagy et al., 2020].

O Big Bird une essas três estruturas a fim de propor uma aproximação para o mecanismo de atenção original. Ou seja, busca-se resultados semelhantes com menos complexidade. Essas alterações permitem a utilização de sequências de textos maiores.

2.8. Conclusão

Esse Capítulo apresentou um estudo sobre as principais técnicas de deep learning para o processamento de linguagem natural. No entanto, esse assunto é muito amplo e constantemente novas técnicas estão sendo propostas. Dessa forma, esse trabalho não teve a pretensão de esgotar esse assunto, sendo apenas um ponto de partida para estudos mais profundos nessa área.

Referências

Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The long-document transformer. *ArXiv*, abs/2004.05150.

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

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Stroudsburg, PA, USA. Association for Computational Linguistics.

Chollet, F. (2017). *Deep learning with Python*. Simon and Schuster, Shelter Island, NY 11964, 1 edition.

Devlin, J., Chang, M. W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1(Mlm):4171–4186.

Díaz-Guerra, L., Daher Adegas, F., Stoustrup, J., and Monros, M. (2012). Adaptive control algorithm for improving power capture of wind turbines in turbulent winds. In *2012 American Control Conference (ACC)*, pages 5807–5812.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Firth, J. R. (1957). A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

Goyal, P., Pandey, S., and Jain, K. (2018). *Deep Learning for Natural Language Processing*. Apress, Berkeley, CA.

Han, D., Liu, Q., and Fan, W. (2018). A new image classification method using CNN transfer learning and web data augmentation. *Expert Systems with Applications*, 95:43–56.

Harris, Z. S. (1954). Distributional Structure. *WORD*, 10(2-3):146–162.

Hassan, H., Aue, A., Chen, C., Chowdhary, V., Clark, J., Federmann, C., Huang, X., Junczys-Dowmunt, M., Lewis, W., Li, M., et al. (2018). Achieving human parity on automatic chinese to english news translation. *arXiv preprint arXiv:1803.05567*.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

Johnson, R. and Zhang, T. (2015). Effective use of word order for text categorization with convolutional neural networks. In Mihalcea, R., Chai, J. Y., and Sarkar, A., editors, *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 103–112. The Association for Computational Linguistics.

Kim, Y. (2014). Convolutional neural networks for sentence classification. In Moschitti, A., Pang, B., and Daelemans, W., editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1746–1751. ACL.

Kosmajac, D. and Kešelj, V. (2019). Automatic text summarization of news articles in serbian language. In *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6. IEEE.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2020). Albert: A lite bert for self-supervised learning of language representations. *ArXiv*, abs/1909.11942.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 24(C):109–165.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient Estimation of Word Representations in Vector Space. *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems*.

Paiva, E., Paim, A., and Ebecken, N. (2021). Convolutional neural networks and long short-term memory networks for textual classification of information access requests. *IEEE Latin America Transactions*, 19(5):826–833.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *30th International Conference on Machine Learning, ICML 2013*, number PART 3, pages 2347–2355. PMLR.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Stroudsburg, PA, USA. Association for Computational Linguistics.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. In *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, volume 1, pages 2227–2237, Stroudsburg, PA, USA. Association for Computational Linguistics.

Radfort, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). (OpenAI Transformer): Improving Language Understanding by Generative Pre-Training. *OpenAI*, pages 1–10.

Rong, X. (2014). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108.

Sun, C., Qiu, X., Xu, Y., and Huang, X. (2019). How to Fine-Tune BERT for Text Classification? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11856 LNAI(2):194–206.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems*, 2017-Decem(Nips):5999–6009.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 1096–1103, New York, New York, USA. ACM Press.

Yadav, A. and Vishwakarma, D. K. (2020). Sentiment analysis using deep learning architectures: a review. *Artificial Intelligence Review*, 53(6):4335–4385.

Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. (2020). Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*, volume 33, pages 17283–17297.

Zhang, X., Zhao, J. J., and LeCun, Y. (2015). Character-level convolutional networks for text classification. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657.

Zhu, W., Lan, C., Xing, J., Zeng, W., Li, Y., Shen, L., and Xie, X. (2016). Co-occurrence Feature Learning for Skeleton based Action Recognition using Regularized Deep LSTM Networks. *30th AAAI Conference on Artificial Intelligence, AAAI 2016*, pages 3697–3703.