UNIVERSIDADE FEDERAL DA GRANDE DOURADOS

Adriano Souza Arruda

Representação de Palavras para Aplicações de Aprendizado Profundo no Processamento de Linguagem Natural

Dourados – MS

2019

Adriano Souza Arruda

Representação de Palavras para Aplicações de Aprendizado Profundo no Processamento de Linguagem Natural

Trabalho de Conclusão de Curso de graduação apresentado para obtenção do título de Bacharel em Sistemas de Informação pela Faculdade de Ciências Exatas e Tecnologia da Universidade Federal da Grande Dourados.

Orientador: Prof.° Dr. Joinvile Batista Junior

Dourados – MS
2019

Adriano Souza Arruda

Representação de Palavras para Aplicações de Aprendizado Profundo no Processamento de Linguagem Natural

Trabalho de Conclusão de Curso aprovado como requisito para obtenção do título de Bacharel em Sistemas de Informação na Universidade Federal da Grande Dourados, pela comissão formada por:

Orientador Prof. Dr. Joinvile Batista Junior
FACET – UFGD

Prof. Me. Anderson Bessa da Costa
FACET – UFGD

Prof. Me. Murilo Táparo
FACET – UFGD

Dourados, 22 de novembro de 2019

Dados Internacionais de Catalogação na Publicação (CIP).

A778r Arruda, Adriano Souza

Representação de Palavras para Aplicações de Aprendizado Profundo no Processamento de Linguagem Natural [recurso eletrônico] / Adriano Souza Arruda. -- 2019.

Arquivo em formato pdf.

Orientador: Joinvile Batista Junior.

TCC (Graduação em Sistemas de Informação)-Universidade Federal da Grande Dourados, 2019.

Disponível no Repositório Institucional da UFGD em:
https://portal.ufgd.edu.br/setor/biblioteca/repositorio

1. aprendizado de máquina. 2. aprendizado profundo. 3. processamento de linguagem natural. 4. rede neural profunda. 5. análise de sentimentos. I. Batista Junior, Joinvile . II. Título.

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

RESUMO

O Processamento de Linguagem Natural (PLN) tem se beneficiado fortemente da utilização de Aprendizado Profundo (*Deep Learning*). Para que sentenças de linguagem natural sejam processadas por redes neurais, é necessário que as palavras de sentença sejam representadas por valores numéricos. Neste trabalho, são analisados três sistemas que suportam a representação de palavras em formato numérico: Word2Vec, ELMo e BERT. Foi montada uma rede neural na qual as camadas iniciais são mapeadas a partir de modelos pré-treinados desses sistemas e as camadas finais utilizam a representação de palavras para realizar a tarefa de PLN: análise de sentimentos. Foi realizado um estudo das arquiteturas envolvidas dos três sistemas escolhidos e feita uma análise do desempenho da rede utilizada como referência para comparação dos três sistemas.

Palavras-Chave: Aprendizado de máquina, aprendizado profundo, processamento de linguagem natural, rede neural profunda, análise de sentimentos.

ABSTRACT

Natural Language Processing (NLP) has benefited greatly from the use of Deep Learning. For natural language sentences to be processed by neural networks, sentence words must be represented by numeric values. In this work, three systems that support the representation of words in numerical format are analyzed: Word2Vec, ELMo and BERT. A neural network was set up in which the initial layers are mapped from pre-trained models of these systems and the final layers use the word representation to perform the NLP task: sentiment analysis. A study of the involved architectures of the three chosen systems was performed and an analysis of the performance of the network used as reference for comparison of the three systems was made.

Keywords: Machine learning, deep learning, natural language processing, deep neural network, sentiment analysis.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 Introdução

Neste capítulo serão apresentados o contexto histórico, a motivação, o objetivo geral, os objetivos específicos, a metodologia adotada e o conteúdo do trabalho proposto.

1.1 Histórico e Motivação

Recentemente o Processamento de Linguagem Natural (PLN) tem se beneficiado das técnicas de Aprendizado Profundo que em geral tem possibilitado melhorias em relação ao estado da arte. Atualmente as tarefas utilizam um conceito conhecido como Representação de Palavras (*Word Embeddings*), que foi abordado pela primeira vez no trabalho "A Neural Probabilistic Language Model" (Bengio et al. 2003). Esse conceito tornou-se popular através do Word2Vec (Mikolov et al. 2013), um modelo que utiliza redes neurais artificiais (RNA) para gerar vetores de palavras compostos por números reais. O resultado consiste em um dicionário no qual palavras com semelhança sintática e semântica estão mapeadas próximas umas das outras.

Apesar do grande impacto e avanço do Word2Vec em relação as técnicas de PLN, sua utilização não permite tratar a polissemia. O presente trabalho aborda duas propostas que se propõem a resolver esse problema. A primeira delas é um modelo de linguagem conhecido como ELMo (Embeddings from Language Models) (Peters et al., 2018). Este modelo é uma aplicação de aprendizado profundo que utiliza Redes Neurais Profundas (RNPs), e ao contrário do Word2Vec, é capaz de gerar, em tempo de execução, diferentes representações para uma mesma palavra. Essa caraterística permitiu que o modelo alcançasse o estado da arte em uma ampla gama de tarefas, como resposta a perguntas, análise de sentimentos e reconhecimento de entidades nomeadas (Peters et al., 2018).

A segunda proposta trata-se do BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2018), um modelo de linguagem baseado na arquitetura Transformer (Vaswani et al., 2017) que superou as representações do ELMo e alcançou o estado da arte em onze tarefas de PLN (Devlin et. al, 2018).

Dado a importância e os recentes avanços de PLN, foi realizado um estudo de caso sobre três modelos de aprendizado de máquina (Word2Vec, ELMo e BERT) e suas principais arquiteturas. O treinamento foi feito sob uma base de dados rotulada na tarefa de análise de sentimentos e ao final os modelos foram analisados comparativamente.

1.2 Objetivos do Trabalho

1.2.1 Objetivo Geral

Estudar propostas de representação de palavras que viabilizam aplicações de aprendizado profundo para processamento de linguagem natural.

1.2.2 Objetivos específicos

- Descrever as principais arquiteturas utilizadas nas propostas estudadas para representação de palavras em redes neurais. A saber: Redes Neurais Convolucionais (CNN), Redes Neurais Recorrentes (RNR), Redes *Highways* e Transformer.
- Efetuar uma análise comparativa de implementações disponíveis das propostas estudas a partir de uma aplicação de processamento de linguagem natural, especificamente análise de sentimentos.

1.3 Metodologia Adotada

A metodologia adotada neste trabalho consiste na pesquisa exploratória dentre três modelos de linguagem utilizados no processamento de linguagem natural. Para tanto, será realizado um estudo detalhado de cada artigo e das principais arquiteturas utilizadas.

Como estudo de caso, a tarefa escolhida consiste em buscar por uma base de dados rotulada, realizar o treinamento, teste e incorporar os modelos em uma rede neural para de análise de sentimentos. Ao final, será realizado um comparativo dos resultados.

1.4 Conteúdo do Trabalho

O presente trabalho possui a seguinte estrutura:

- **Capítulo 2 – Fundamentação Teórica:** é apresentado as principais arquiteturas de redes neurais profundas e explicado conceitos teóricos e práticos de três modelos de aprendizado de máquina.
- **Capítulo 3 – Desenvolvimento do Trabalho Proposto:** é detalhado como os três modelos de aprendizado de máquina foram treinados e testados, além de uma análise comparativa entre eles.
- **Capítulo 4 – Considerações Finais:** são apresentadas as principais conclusões e contribuições deste trabalho, além de sugestões de trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo serão apresentadas as principais arquiteturas utilizadas nas três propostas estudadas: Redes Neurais Convolucionais, Redes Neurais Recorrentes, Redes Highways, Transformer. Adicionalmente, serão abordados os conceitos de cada um dos modelos estudados: Word2Vec, ELMo e BERT.

2.1 Redes Neurais Convolucionais (CNN)

Rede Neural Convolucional (CNN - *Convolutional Neural Network*) é uma classe de rede que foi inicialmente projetada para classificação de imagens, mas posteriormente foi aplicada em outros domínios, como processamento de linguagem natural.

A principal inovação das CNNs está na capacidade de aprender automaticamente e em paralelo um grande número características sem qualquer supervisão humana. Essa independência de um conhecimento prévio e do esforço humano tornou a CNN uma das principais e mais populares arquiteturas de aprendizado profundo.

Conforme Sumit (2018) CNNs são capazes de capturar com sucesso as dependências espaciais e temporais de imagem através da aplicação de filtros. Em relação as redes neurais tradicionais, a CNN realiza um melhor ajuste ao conjunto de dados devido à redução no número de parâmetros, além da reutilização dos pesos.

Todos os modelos CNN seguem uma arquitetura semelhante ao da Figura 1.

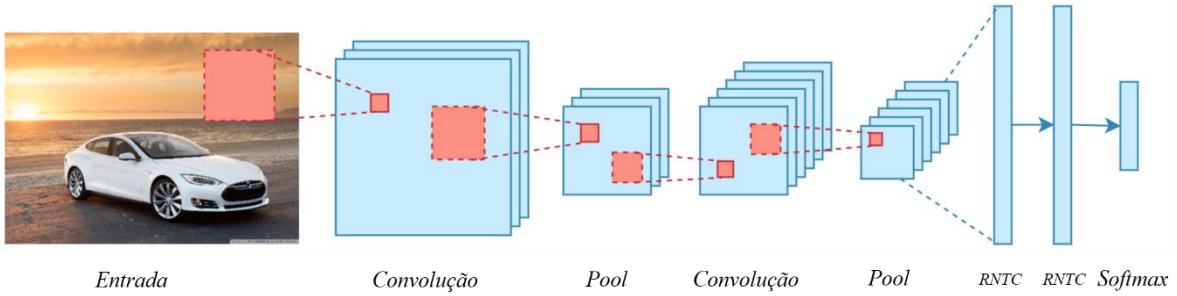


Figura 1: Arquitetura base para uma Rede Neural Convolucional
Fonte: Adaptado (ARDEN, 2017)

Dado um objeto alvo, neste caso uma imagem. É realizado uma convolução em série, posteriormente operações de *pool*, seguido por uma rede neural formada por camadas totalmente conectadas (RNTC, figura 1). Em tarefas de classificação com várias classes, a saída é dada por uma camada *softmax*.

2.1.1 Camada de Convolução

O principal componente da CNN é a camada convolucional. Convolução é uma operação matemática para mesclar dois conjuntos de informações. No caso da Figura 1Figura 2, a convolução é aplicada nos dados de entrada usando um filtro de convolução para produzir um mapa de características.

Um filtro é tipicamente uma matriz de pesos $M \times N$ que é aplicada sobre uma entrada para capturar características. Normalmente é inicializado seguindo alguma heurística e atualizado por retropropagação (*backpropagation*). Este mecanismo consiste em atualizar os filtros comparando a diferença entre a saída real e a saída esperada (KIM, 2019).

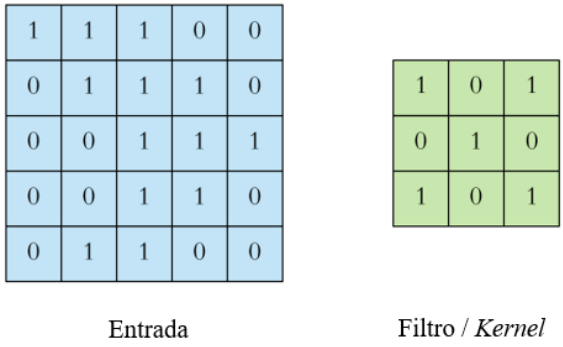


Figura 2: Exemplo de filtro para convolução
Fonte: Arden (2017)

Como um computador não efetua cálculos sobre pixels ou valores alfanuméricos, é necessário converter a entrada para uma matriz de números. O lado esquerdo da Figura 2 trata-se da entrada para uma camada de convolução e à direita o filtro, também chamado de *kernel*.

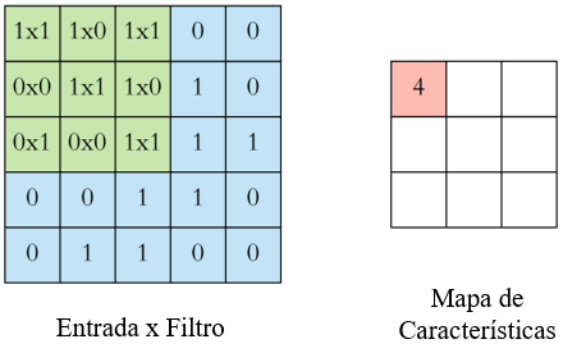


Figura 3: Aplicação de filtro sobre uma entrada
Fonte: Arden (2017)

Uma operação de convolução é realizada “deslizando” o filtro ao longo da entrada. Para cada posição deslizada, efetua-se a multiplicação elemento por elemento, somando o resultado.

Na Figura 3, a região superior esquerda no qual ocorrem as multiplicações (área verde) corresponde a uma operação de convolução, conhecida como campo receptivo. O resultado da primeira operação, neste caso 4, é adicionado ao mapa de características. Em seguida, o filtro é deslizado para a direita executando a mesma operação, no qual o resultado também adicionando ao mapa. Esta sequência é repetida até que toda a entrada tenha sido percorrida. O resultado final é mostrado na Figura 4.

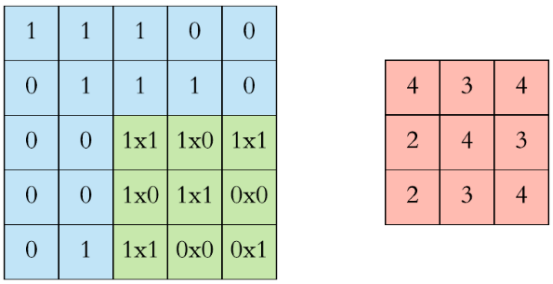


Figura 4: Resultado da convolução
Fonte: Arden (2017)

O exemplo supracitado trata-se de uma operação de convolução 2D. No entanto, para classificação de imagens, as convoluções são realizadas em 3D para representar a altura, largura e profundidade. Cada profundidade corresponde aos canais de cores RGB.

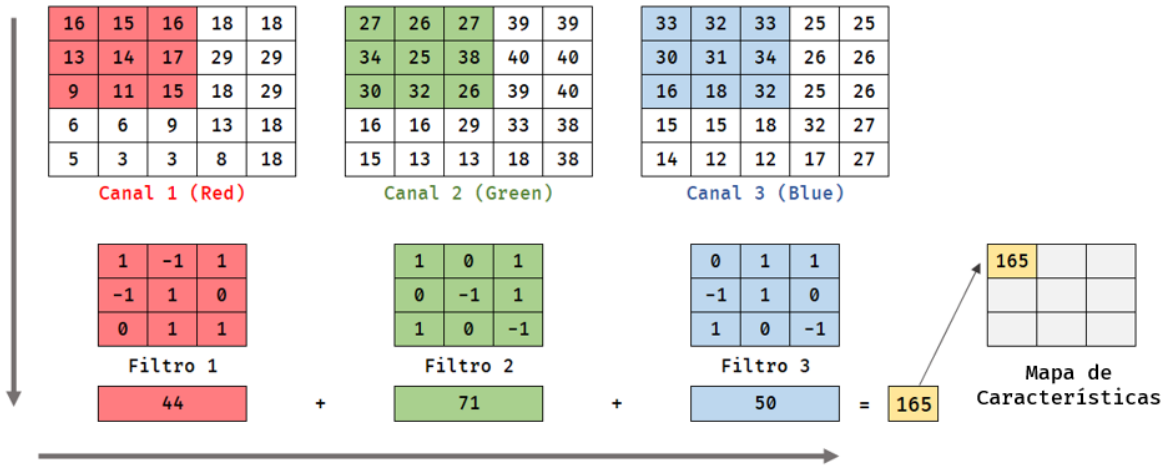


Figura 5: Convolução 3D
Fonte: Do autor

Conforme ilustrado na Figura 5, a principal diferença em relação a convolução 2D está na aplicação de filtros para cada camada RGB. O resultado das convoluções são somados e adicionado ao mapa de características. Esta sequência de etapas é repetida até que as janelas deslizem por toda a entrada.

2.1.2 Camada Pooling

Afim de reduzir a complexidade das redes neurais e acelerar o treinamento, os dados passam por camadas de *pooling* que reduzem a altura e largura das matrizes, mantendo a profundidade. O tipo mais comum é o *max pooling*, que simplesmente captura o valor máximo da janela.

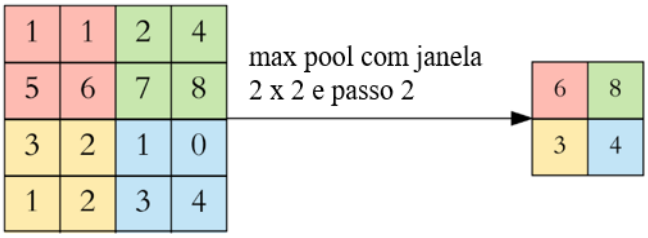


Figura 6: Aplicação do max pooling
Fonte: Arden (2017)

Na Figura 6 foi aplicado a técnica de *max pooling* em uma janela 2x2 com passo 2. Cada cor indica uma janela diferente. Considere passo como o número de pixels deslocados sobre a matriz de entrada.

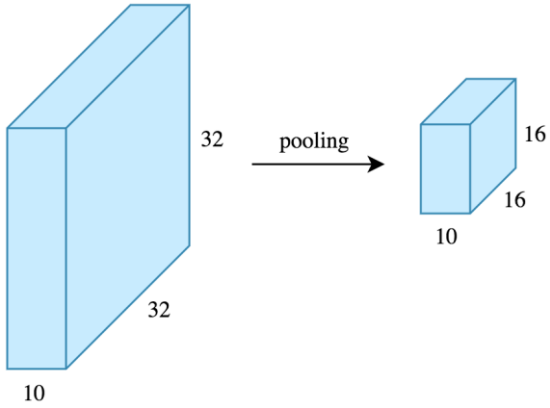


Figura 7: Exemplo de pooling em uma imagem 32 x 32 x 10
Fonte: Arden (2017)

Na Figura 7 o lado esquerdo representa uma entrada de tamanho 32 x 32 x 10. Ao aplicar o *pooling* com os mesmos parâmetros da Figura 6, o resultado será um mapa de características com dimensões 16 x 16 x 10. A altura e a largura do mapa de características são reduzidas pela metade, mas a profundidade mantida. Esta redução é muito importante, pois redes neurais profundas lidam com milhões de pesos de alto custo computacional.

Nas arquiteturas CNN, o *pooling* é normalmente realizado com janelas 2 x 2, passo (*stride*) 2 e sem preenchimento (*padding*). Enquanto a convolução é feita com janelas 3 x 3, passo 1 e com preenchimento. Quando um filtro não se encaixa perfeitamente na imagem de entrada é aplicado a técnica de preenchimento, que consiste em adicionar uma borda preenchida com zeros (ARDEN, 2017).

2.2 Redes Neurais Recorrentes

No clássico exemplo para o mercado de ações, toda previsão depende do histórico passado. Desta forma, é necessário que haja persistência. Para resolver problemas como este, foram modeladas as Redes Neurais Recorrentes (RNN – *Recurrent Neural Networks*), que em termos gerais são laços (*loops*) que capturam persistência. Elas foram projetadas para reconhecer padrões em sequências de dados, como texto, caligrafia, fala ou quaisquer outros dados de séries temporais. Esta arquitetura leva em consideração o tempo, a sequência e possuem uma dimensão temporal (CHRISTOPHER, 2015).

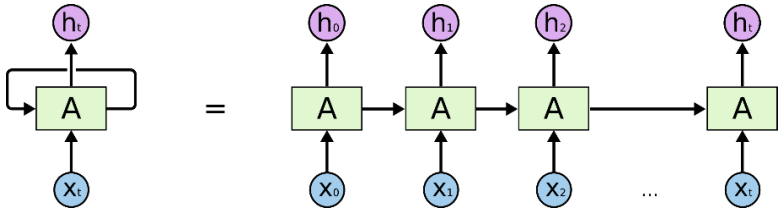


Figura 8: RNN desmembrada
Fonte: Christopher (2015)

Na Figura 8, do lado esquerdo, parte da rede neural $\boldsymbol{A}$, analisa uma entrada $\boldsymbol{x_t}$ e gera um valor $\boldsymbol{h_t}$. O laço permite que as informações sejam passadas de uma etapa da rede para a próxima. Este processo é ilustrado do lado direito, no qual há várias cópias da rede, cada uma passando dados para o sucessor.

Em teoria, redes neurais recorrentes também seriam capazes de lidar com dependências de longo prazo. Mas na prática, devido a problemas de desaparecimento do gradiente (*vanishing gradient*) e explosão do gradiente (*exploding gradient*), isto não acontece. O problema foi profundamente abordado em Bengio, Simard e Frasconi (BENGIO, SIMARD e FRASCONI, 2019) e (JOSEF, 1991).

O desaparecimento do gradiente ocorre quando uma rede neural recorrente é incapaz de propagar informações de gradiente úteis da saída de volta para a entrada. Já a explosão do gradiente ocorre quando gradientes de erro se acumulam resultando em muitas atualizações nos pesos durante o treinamento. Estes problemas fazem com que os modelos sejam instáveis e incapazes de aprender com os dados de treinamento (BROWNLEE, 2019).

O gradiente descendente (gradient descent) é um método utilizado na otimização para encontrar o mínimo local de uma função de perda (*loss function*). Um algoritmo de aprendizagem de máquina normalmente inicia em um ponto e a partir desse ponto é calculado o seu gradiente, de modo que seja possível minimizar o erro. De modo geral, o cálculo do gradiente é realizado derivando a função de perda em um determinado ponto, essa derivada aponta para onde a função está crescendo (KRISTEN, 2019).

Suponha que um comerciante queira prever as vendas de material escolar. Provavelmente as vendas irão atingir picos apenas duas vezes no ano (início e fim de cada

semestre letivo). Neste cenário é preciso ter um enorme contexto para que vendas antigas sejam incluídas. Infelizmente, à medida que essas lacunas aumentam, as RNNs tornam-se incapazes de aprender a conectar informações, conforme ilustrado na Figura 9.

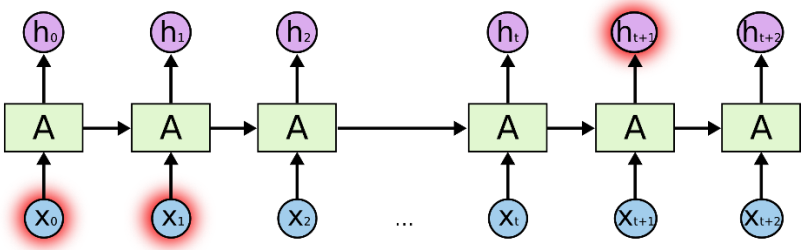


Figura 9: Ilustração de RNNs com dependências de longo prazo
Fonte: Christopher (2015)

Para que fosse possível lidar com dependências de longo prazo, foi criada uma arquitetura de memórias de longo prazo, conhecida como LSTM (Long Short Term Memory). Este modelo foi proposto por Hochreiter e Schmidhuber (HOCHREITER e SCHMIDHUBER, 1997) e posteriormente refinado em diversos trabalhos científicos.

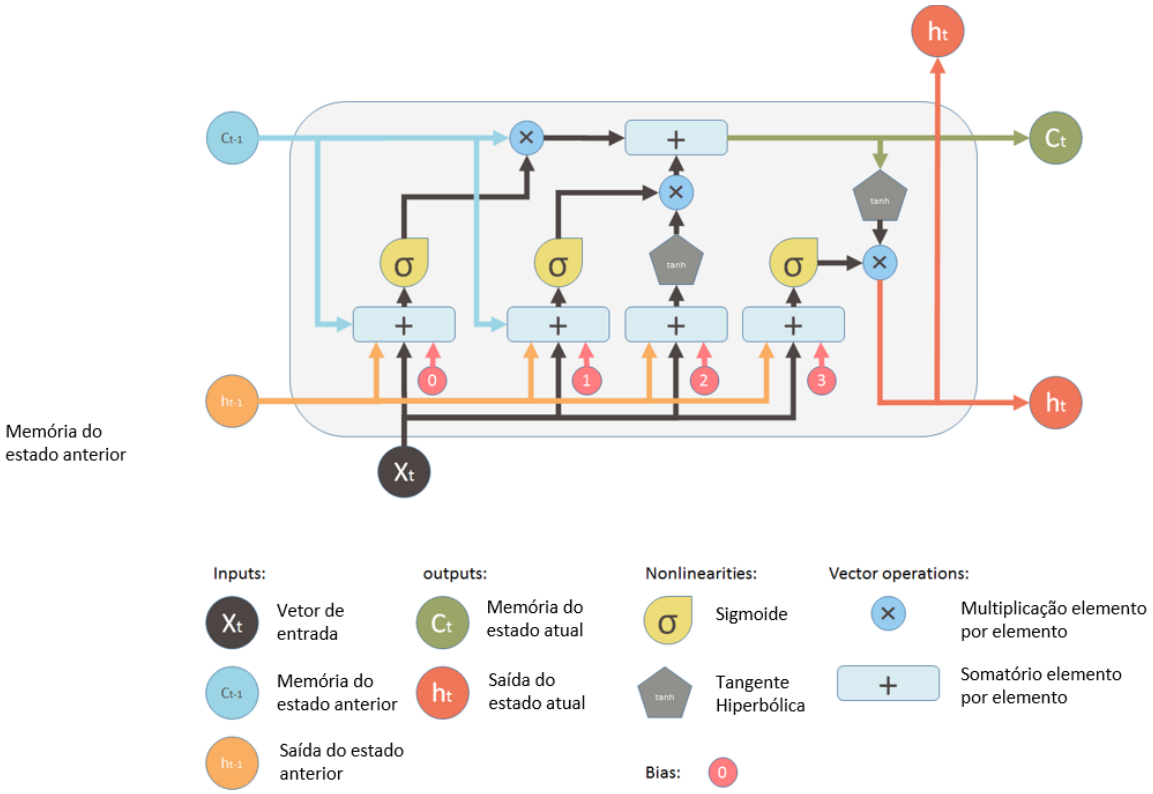


Figura 10: Ilustração de uma unidade LSTM
Fonte: Yan (2016)

Na Figura 10, temos a ilustração de uma unidade LSTM que aceita três entradas, sendo x_t a entrada do tempo atual, h_{t-1} a saída do estado anterior e c_{t-1} a memória do estado anterior. Quanto as saídas, h_t é a saída da rede atual e c_t é a memória da unidade atual.

O comportamento da memória interna c_t é regulado por estruturas chamadas de portões ou *gates*, em inglês. Eles são compostos de uma função de ativação sigmoide e uma operação de multiplicação por pontos, conforme ilustrado na Figura 11.

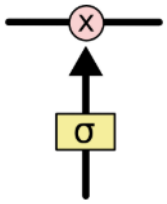


Figura 11: Ilustração de um gate
Fonte: Christopher (2015)

A primeira tarefa de um LSTM é decidir quais informações serão mantidas e quais serão descartadas. O componente responsável por esta decisão chamasse *forget gate* em que é comparado a saída do estado anterior h_{t-1} com a entrada do estado atual x_t, produzindo uma saída para cada número no estado da memória anterior c_{t-1}. Se o valor for 0, a informação da memória anterior será descartada, caso contrário, ela será processada para compor a informação da memória no estado atual.

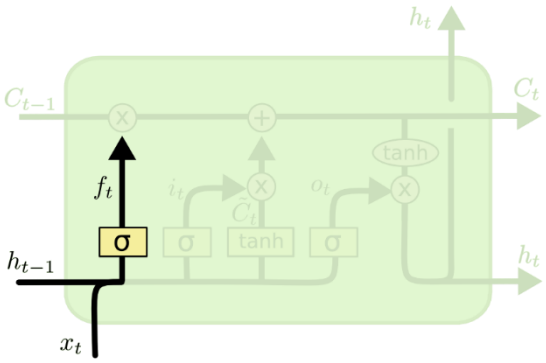


Figura 12: Ilustração do forget gate (portão do esquecimento)
Fonte: Christopher (2015)

O próximo passo é decidir quais novas informações serão armazenadas no estado da célula. Primeiro, uma função de ativação sigmoide i_t chamada *input gate* decide quais valores serão atualizados. Em seguida, uma função tangente hiperbólica ($tanh$) cria um vetor de novos valores candidatos, $\tilde{C}_t$, que podem ser adicionados ao estado. Na próxima etapa, ambos são combinados para atualizar o estado atual.

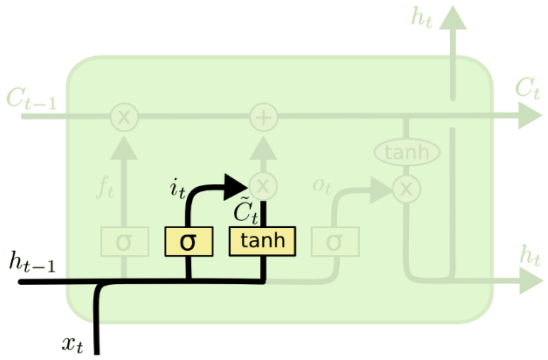


Figura 13: Ilustração do input gate e tanh
Fonte: Christopher (2015)

Na etapa final (Figura 14), é aplicado uma função de ativação sigmoide O_t que determina quais partes do estado farão parte da saída. O estado atual c_t é modificado por *tanh* (para manter os valores entre -1 e 1) e depois multiplicado pela saída do *gate sigmoide* O_t para produzir somente o que foi decidido.

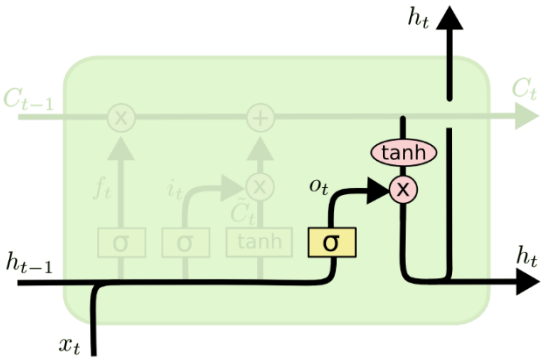


Figura 14: Saída de uma rede LSTM
Fonte: Christopher (2015)

2.3 Redes Highway

A profundidade das redes neurais é crucial para obter um melhor resultado. No entanto, quanto mais profundo é o modelo, mais difícil é o treinamento. Como alternativa, foi proposta uma nova arquitetura denominada *Highway Networks* inspirada em redes LSTM de conexões residuais, regulando quanto do sinal de entrada é adicionado à saída (SRIVASTAVA, GREFF e SCHMIDHUBER, 2015).

Considere uma rede neural de 4 camadas como exemplo. Em uma configuração residual, além de passar a saída da camada 1 para a camada 2, é necessário também adicionar as saídas da camada 1 às saídas da camada 2. Seja cada camada denotada por $f(x)$, em uma rede convencional temos $y = f(x)$. No entanto, em uma rede residual, $y = f(x) + x$.

Todas as fórmulas citadas nesta seção, tem as seguintes características: letras em negrito indicam vetores e matrizes; letras maiúsculas em itálico indicam funções de transformação; **0** e **1** denotam vetores de zeros e uns, respectivamente; o operador (·) é usado para indicar a multiplicação elemento por elemento, ou seja, elemento por elemento (*element-wise*).

Uma rede neural simples de *feedforward* consiste em L camadas, no qual a camada l^{th} ($l \in \{1, 2, \dots, L\}$) aplica uma transformação não linear H (parametrizada por $W_{H,1}$) em sua entrada x_1 para produzir sua saída y_1. Assim, x_1 é a entrada para a rede e y_L é a saída da rede. O índice da camada e o viés *(bias)* foram omitidos para facilitar o entendimento:

$$y = H(x, \boldsymbol{W_H})$$

Equação 1

Na Equação 1 H é geralmente uma transformação afim seguida por uma função de ativação não linear, mas pode assumir outras formas, como convolucionais ou recorrentes. Para uma rede *highway*, são definidas duas transformações $T(x, \boldsymbol{W_T})$ e $C(x, \boldsymbol{W_c})$ de forma que:

$$y = H(x, \boldsymbol{W_H}) \cdot T(x, \boldsymbol{W_T}) + x \cdot C(x, \boldsymbol{W_c})$$

Equação 2

Na Equação 2, T representa um Portão de Transformação (*Transform Gate*) que expressa a quantia da saída que é transformada pela entrada. Por outro lado, C representa um Portão de Transporte (*Carry Gate*) que expressa a quantia da saída que é transportada pela entrada. Para simplicidade, $C = 1 - T$, sendo:

$$y = H(x, \boldsymbol{W_H}) \cdot T(x, \boldsymbol{W_T}) + x \cdot (1 - T(x, \boldsymbol{W_T}))$$

Equação 3

Há condições para valores de T específicos:

$$y = \begin{cases} x, & se\ T(x, \boldsymbol{W_T}) = \boldsymbol{0} \\ H(x, \boldsymbol{W_H}), & se\ T(x, \boldsymbol{W_T}) = \boldsymbol{1} \end{cases}$$

Equação 4

Quando $T = 0$, a entrada é passada diretamente como saída, o que cria uma rodovia (*highway*) de informações. Esta característica originou o termo "*Highway Networks*". Quando T = 1, é utilizado a entrada transformada, ativada e não linear como saída.

A dimensionalidade de $x, y, H(x, \boldsymbol{W_H})$ e $T(x, \boldsymbol{W_T})$ deve ser a mesma para que a Equação 2 seja válida. Para alterar o tamanho da representação, pode-se substituir x por $\hat{x}$ obtido por subamostragem ou preenchimento com zero (*zero-padding*) x. Outra alternativa é

usar uma camada simples, sem *highways*, para alterar a dimensionalidade, que é uma estratégia utilizada e sugerida pelos autores.

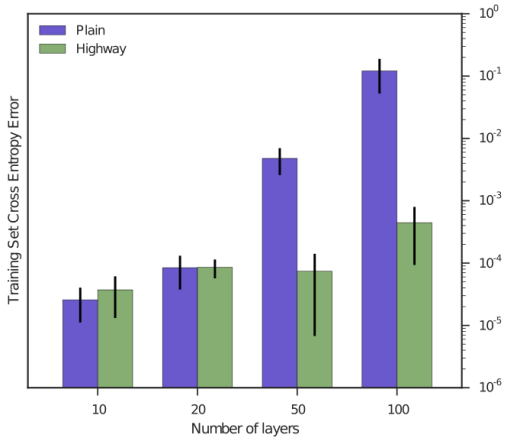


Figura 15: Comparativo da otimização entre redes simples e highways de várias profundidades.
Fonte: Srivastava, Greff e Schmidhuber (2015, p. 3)

Na Figura 15 é mostrado o desempenho das principais configurações em até 100 camadas. Nota-se que redes simples se tornam muito mais difíceis de otimizar com o aumento da profundidade, enquanto redes *highways* mantém um equilíbrio.

2.4 Transformer

As redes neurais recorrentes foram por muito tempo a principal maneira de capturar dependência entre sequências. No entanto, elas são de natureza sequencial, o que dificulta o processamento paralelo e as tornam ineficientes em GPUs modernas (WANG, 2019). Além disso, mesmo utilizando LSTMs, há uma certa dificuldade em lidar com dependências de longo prazo, como alternativa, foi proposto pelo Google AI uma arquitetura chamada Transformer (Vaswani et al., 2017), publicado no artigo “Attention Is All You Need”. Esta arquitetura surgiu com o propósito de auxiliar na transdução de sequencias e tem sido utilizada com sucesso em diversos modelos de última geração.

“O Transformer é o primeiro modelo de transdução que depende inteiramente da autoatenção para calcular representações da entrada e saída sem usar RNNs ou convolução alinhados em sequência” (Vaswani et al., 2017).

Dado uma sequência de entrada, em um transformer, cada sentença de entrada passa por bloco codificador (*encoder*) e é transformada ou traduzida por um bloco decodificador (*decoder*). Estes blocos são compostos por uma pilha de tamanho fixo, conforme ilustrado na Figura 16.

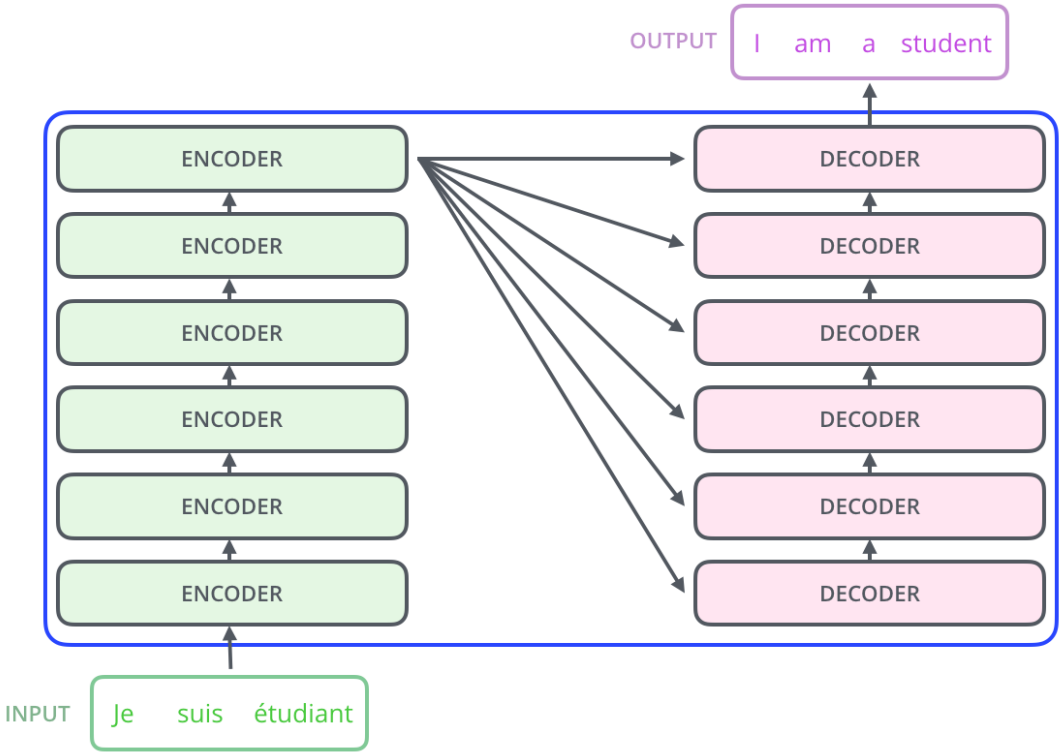


Figura 16: Visão alto nível de um Transformer
Fonte: Alammar (2018)

Os codificadores são idênticos em estrutura e não compartilham pesos. As entradas fluem através de uma camada de auto atenção (*self-attention)*. Este componente ajuda o codificador encontrar palavras chave e será descrito mais adiante. A segunda camada é uma rede neural *feedforward* simples.

O decodificador é semelhante ao codificador, mas possui uma camada de atenção extra que ajuda a se concentrar em partes relevantes da sentença de entrada.

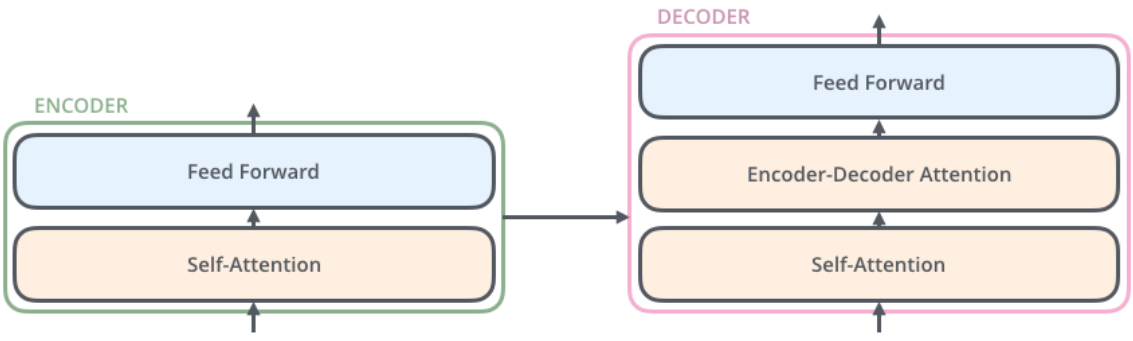


Figura 17: Estrutura de um codificador e decodificador
Fonte: Alammar (2018)

A atenção transforma a sentença de entrada em um vetor de palavras e, em seguida, faz correspondências, identificando o contexto relevante. Esta técnica é muito útil na tradução automática.

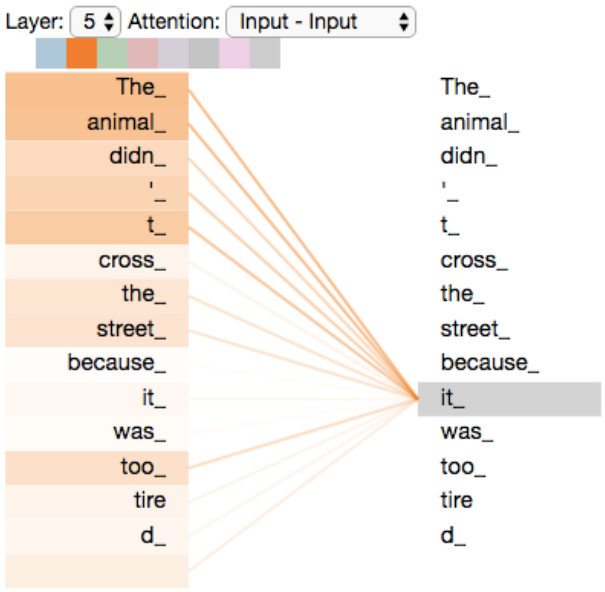


Figura 18: Ilustração do mecanismo de auto atenção
Fonte: Alammar (2018)

Na Figura 18, a palavra “it” está sendo codificada na camada número 5 e do lado esquerdo, as diferentes tonalidades indicam o valor da “atenção”. Desta forma, podemos notar que o foco está em “the” e “animal”.

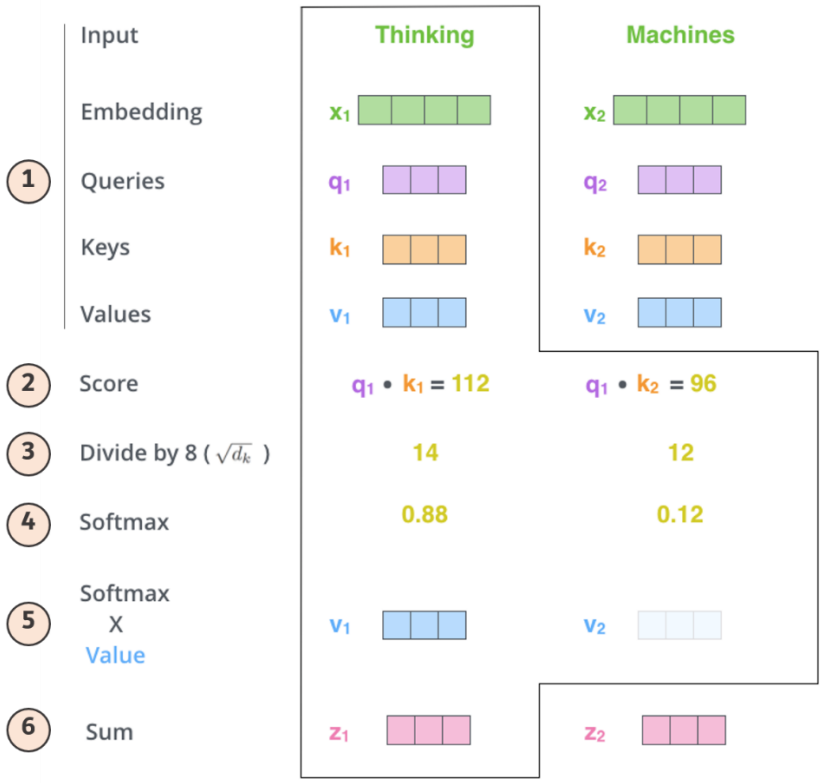


Figura 19: Cálculo da pontuação de atenção
Fonte: Alammar (2018)

Conforme ilustrado na Figura 19, o cálculo da atenção pode ser dividido em 6 etapas principais:

1. Para cada palavra, criar um vetor de consulta (*query*), um vetor de chave (*key*) e um vetor de valor (*value*). Esses vetores são obtidos multiplicando a representação da palavra por três matrizes de pesos aprendidas durante o treinamento.
2. Calcular uma pontuação marcando a palavra alvo em cada posição da sequência de entrada. Este cálculo consiste no produto escalar entre o vetor de consulta e o vetor chave da palavra a ser pontuada. Portanto, a pontuação da palavra na primeira posição será o produto escalar de q_1 e k_1 e a pontuação da palavra na segunda posição será o produto escalar de q_1 e k_2.
3. Dividir as pontuações por 8 (raiz quadrada da dimensão dos vetores chave).
4. Envolver o resultado através de uma operação *softmax* que normaliza os valores.
5. Multiplicar cada vetor de valor (v) pela pontuação *softmax*. A ideia consiste em manter a pontuação das palavras a serem focadas e minimizar as que forem irrelevantes (multiplicando por números muito pequenos, como 0,001).
6. O último passo é somar os vetores de valor ponderados, isso produz a saída da camada de auto atenção.

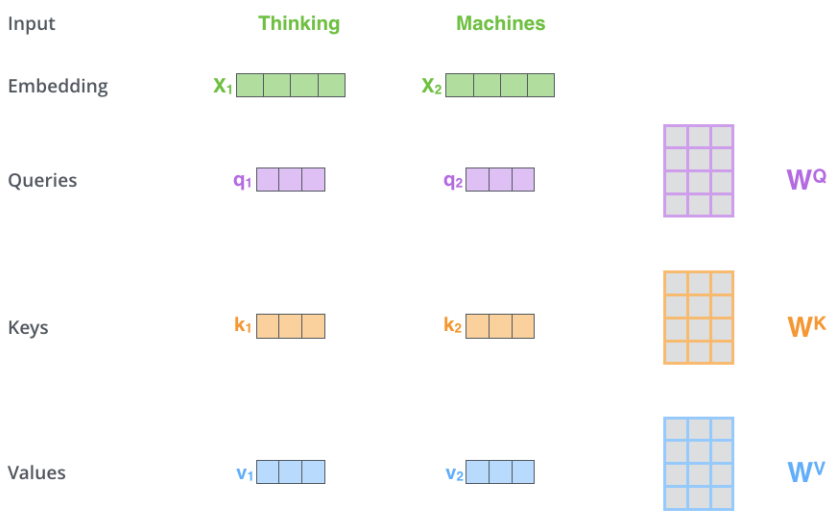


Figura 20: Primeira etapa do cálculo da atenção
Fonte: Alammar (2018)

A primeira etapa da Figura 19 é ilustrado na Figura 20, no qual x_1 é multiplicado pelas matrizes de pesos W_q, W_k e W_v, produzindo os vetores q_1, k_1 e v_1, que são consulta, chave e valor, respectivamente. Este procedimento é repetido para x_2.

Os autores aperfeiçoaram ainda mais a camada de auto atenção, adicionando um mecanismo chamado *Multi-Head Attention*, ou atenção de várias cabeças, em português. Esta

abordagem expande a capacidade do modelo de se concentrar em diferentes posições, fornecendo vários subespaços de representação (ALAMMAR, 2018). Para o artigo apresentado, são 8 matrizes de ponderação (consulta, chave e valor) geradas aleatoriamente para cada codificador e decodificador.

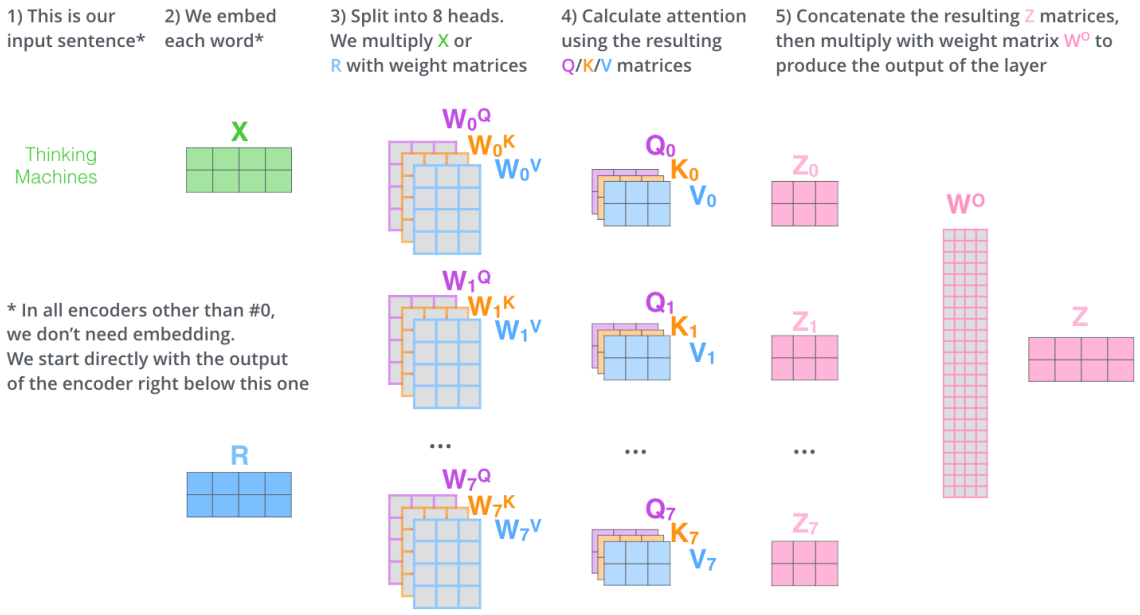


Figura 21: Etapas na camada de atenção com várias cabeças
Fonte: Alammar (2018)

O cálculo das representações para a camada *Multi-Head Attention* é ilustrado na Figura 21 e possui as seguintes etapas:

1. Receber uma sequência de entrada.
2. Gerar uma representação X no primeiro codificador e utilizar a saída R nas camadas subsequentes.
3. Criar 8 *heads* e multiplicar X ou R com as matrizes de pesos.
4. Calcular a atenção usando as matrizes Q, K, V.
5. Concatenar as matrizes Z resultantes e multiplicar pela matriz de pesos W^O para produzir a saída da camada atual. Este procedimento é necessário porque a camada *feedforward* espera uma única matriz para cada palavra.

Para que as camadas do transformer saibam a ordem correta das palavras em uma sequência, é adicionado um vetor de posição. A ideia consiste em fornecer distâncias significativas entre as representações durante a atenção. Se assumirmos que uma determinada representação tem uma dimensionalidade 4, as codificações seriam:

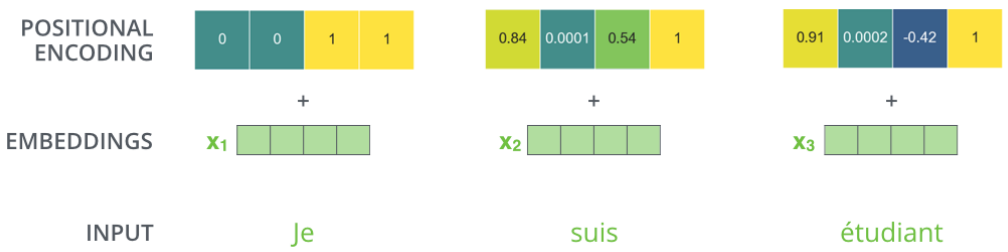


Figura 22: Exemplo de uma codificação posicional
Fonte: Alammar (2018)

Após compreender o cálculo e a importância das representações geradas na camada de atenção com várias cabeças é possível compreender a estrutura dos transformers ilustrado na Figura 23.

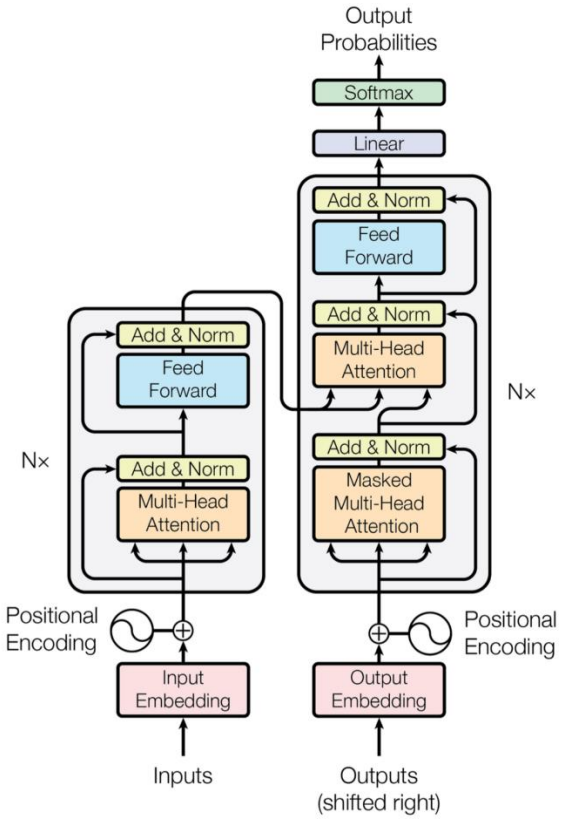


Figura 23: Arquitetura Transformer
Fonte: Vaswani et al. (2017)

O bloco codificador (lado esquerdo) e o bloco decodificador (lado direito) são empilhados em Nx camadas idênticas. No caso do artigo, este hiperparâmetro tem valor 6. A saída de cada subcamada passa por um processo de normalização que está descrito em Ba, Kiros e Hinton (2016).

O codificador é composto por duas subcamadas. O primeiro é um mecanismo de auto atenção com várias cabeças e o segundo é uma rede *feedforward* simples e totalmente conectada. Uma conexão residual é anexada em cada subcamada, seguida pela normalização,

produzindo a saída $LayerNorm(x + Sublayer(x))$, sendo x a representação de cada *token* e $Sublayer(x)$ a função implementada pela própria subcamada. Para o correto funcionamento das conexões residuais, todas as subcamadas do modelo, além das camadas de representação, produzem saídas de dimensão $d_{model} = 512$.

O decodificador é semelhante ao codificador, mas possui uma subcamada extra que executa atenção múltipla sobre a saída da pilha do codificador. Essa máscara, combinada com o fato de que as saídas são compensadas por uma posição, garante que as previsões para a posição i possam depender apenas das saídas conhecidas em posições inferiores a i.

A saída da rede possui duas camadas, uma linear e outra *softmax*. A camada Linear é uma rede neural simples e totalmente conectada que projeta o vetor produzido pela pilha de decodificadores em um vetor chamado de *logits*. A dimensão deste vetor corresponde ao número de palavras aprendidas durante o treinamento e cada célula contém à pontuação de uma palavra. A camada *softmax* transforma as pontuações em probabilidades, todas positivas e somam 1. A célula com a maior probabilidade é escolhida e a palavra associada a ela é produzida como saída para um intervalo de tempo.

2.5 Word2Vec

O Word2Vec (Mikolov et. al, 2013) é um modelo caracterizado por uma rede neural rasa de duas camadas capaz de transformar entradas de texto (*corpus*) em representações vetoriais que descrevem o significado sintático e semântico das palavras. Tais mapeamentos são popularmente conhecidos como *Word Embeddings* e pode ser traduzido como casamento ou representação de palavras.

Embora o Word2Vec utilize uma rede neural de duas camadas, a saída produzida (dicionário) pode ser incorporada por redes neurais profundas. A diferença entre as abordagens é ilustrada na Figura 24.

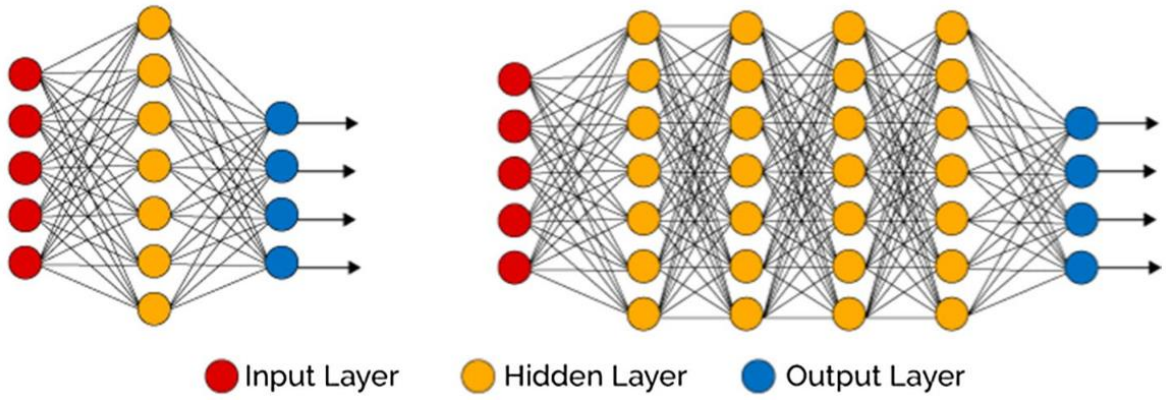


Figura 24: Redes Neurais Rasas x Redes Neurais Profundas
Fonte: Johnson e Khoshgoftaar (2019)

O treinamento do modelo proposto é realizado de maneira semi-supervisionada e os relacionamentos capturados exploram a probabilidade de coocorrência das palavras em relação ao contexto. Cada representação é distribuída em um espaço vetorial e palavras semanticamente semelhantes são mapeadas próximas umas das outras. Para exemplificar o processo, suponha o seguinte *corpus*:

"O gato é peludo"
"O gato subiu no telhado"
"O gato caiu do telhado"

O conjunto de treinamento pode ser divido em *tokens* (palavras e pontuações, por exemplo) e classificado seguindo algum padrão, neste exemplo, ordem lexicográfica. O resultado é um vocabulário com todos os elementos do *corpus*:

["caiu", "do", "é", "gato", "no", "o", "peludo", "subiu", "telhado"]

Este padrão permite gerar um vetor *one-hot* em que a posição correspondente a uma palavra é preenchida com 0, e todo o restante com 1. Como exemplo, a palavra "telhado" tem a seguinte representação:

[0 0 0 0 0 0 0 0 1]

Neste cenário, considere V como o tamanho do vocabulário e N como a dimensão das palavras representadas. A camada de entrada do Word2Vec tem dimensão V e recebe um vetor *one-hot* como entrada; A camada oculta tem dimensão N e a camada de saída tem dimensão V, mas ao invés de 0's e 1's, os vetores expressam probabilidades de cada palavra no vocabulário. Os pesos que interligam a camada de entrada à camada oculta, e a camada oculta à camada de saída, podem ser representados pelas matrizes $WI_{V\times N}$ e $WO_{N\times V}$, respectivamente (ver Figura 25).

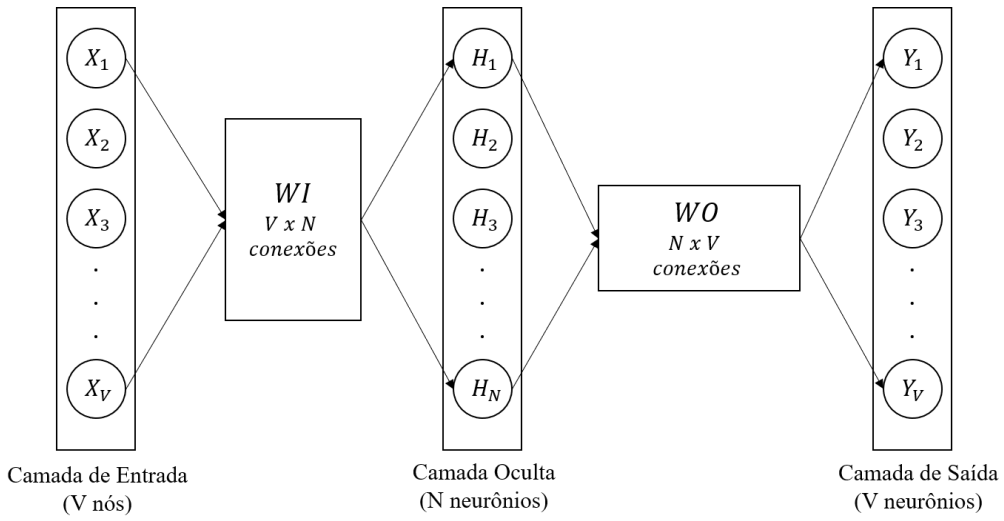


Figura 25: Arquitetura do Word2Vec
Fonte: Krishan (2015)

Como ilustração, suponha que durante o treinamento as matrizes de pesos WI e WO tenham dimensão $N = 3$ e com os seguintes valores:

$$WI = \begin{pmatrix} 0.473 & 0.424 & 0.819 \\ 0.219 & 0.282 & 0.009 \\ 0.186 & 0.559 & 0.786 \\ 0.100 & 0.975 & 0.593 \\ 0.799 & 0.247 & 0.917 \\ 0.893 & 0.579 & 0.772 \\ 0.686 & 0.232 & 0.106 \\ 0.053 & 0.493 & 0.684 \\ 0.327 & 0.015 & 0.855 \end{pmatrix}$$

$$WO = \begin{pmatrix} 0.028 & 0.308 & 0.912 & 0.270 & 0.146 & 0.947 & 0.427 & 0.081 & 0.166 \\ 0.009 & 0.056 & 0.224 & 0.341 & 0.989 & 0.441 & 0.945 & 0.480 & 0.680 \\ 0.830 & 0.655 & 0.657 & 0.066 & 0.765 & 0.718 & 0.246 & 0.113 & 0.843 \end{pmatrix}$$

A forma como uma palavra entra na rede para ser processada é dada por um vetor *one-hot* no qual a posição correspondente a palavra no vocabulário é preenchida com 1 e todo o restante com 0. Como exemplo, suponha a submissão das palavras “gato” e “caiu” para que a rede aprenda a relação entre elas. Nesse caso, dado o vetor *one-hot* da palavra “gato”, é necessário verificar a probabilidade da palavra “caiu” a partir do vetor de saída. O vetor de entrada é dado por:

$$X = [0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0]$$

Ao realizarmos o cálculo da representação $G = X \times (WI \times WO)$ para gato, obtemos, aproximadamente, o seguinte resultado:

$$X = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$WI \times WO =$

0.697	0.706	1.064	0.326	1.115	1.223	0.804	0.334	1.057	0
0.016	0.089	0.269	0.156	0.318	0.338	0.362	0.154	0.236	1
0.663	0.604	0.812	0.293	1.182	0.988	0.802	0.373	1.074	2
0.504	0.473	0.699	0.399	1.432	0.950	1.109	0.543	1.179	3
0.786	0.861	1.386	0.360	1.063	1.524	0.800	0.287	1.074	4
0.671	0.813	1.451	0.489	1.294	1.656	1.118	0.437	1.193	5
0.109	0.294	0.747	0.271	0.411	0.828	0.538	0.179	0.361	6
0.574	0.492	0.608	0.228	1.019	0.759	0.657	0.319	0.921	7
0.719	0.661	0.863	0.150	0.717	0.930	0.364	0.131	0.785	8

$G = X \times (WI \times WO) =$

0.504	0.473	0.699	0.399	1.432	0.950	1.109	0.543	1.179

Em termos práticos, multiplicar o vetor *one-hot* pelo resultado da operação $WI \times WO$ permite selecionar a linha que contém a representação da palavra desejada.

Através da função *softmax*, o Word2Vec transforma os valores da camada de saída em um conjunto de probabilidades. Desta forma, para calcular a probabilidade da palavra "caiu" dado a ocorrência da palavra "gato", temos a seguinte equação:

$$P(w_k) = \frac{e^{w'_k}}{\sum_{n=1}^{V} e^{w'_n}}$$

Equação 5

onde w'_k é o valor da palavra w_k ("caiu") na camada de saída. O resultado da equação é dado por:

$$P(\text{"caiu"} \mid \text{"gato"}) \simeq \frac{e^{1.432}}{21.539} \simeq 0.194$$

Para gerar o dicionário de palavras o Word2Vec implementa duas arquiteturas. A primeira, conhecida como *skip-gram*, consiste em percorrer as palavras de cada sentença e usar a palavra atual *w* para prever seus vizinhos, ou seja, o contexto. A segunda, conhecida como *Continuous Bag-of-Words* (CBOW), utiliza cada um desses contextos para prever a palavra atual *w*.

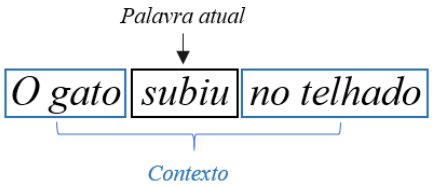


Figura 26: Palavra x Contexto
Fonte: Do autor.

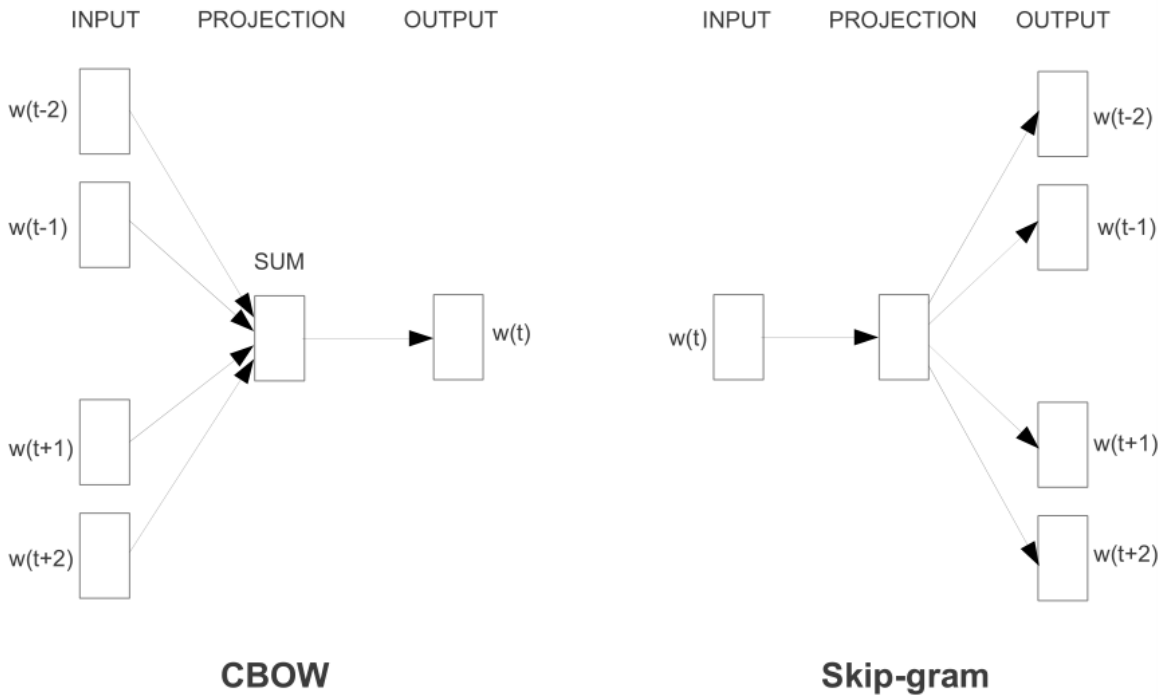


Figura 27: Arquiteturas do Word2Vec
Fonte: Mikolov et al. (2013)

Caso o contexto de uma palavra seja muito grande, além de permitir a captura de termos irrelevantes, o custo computacional pode ser muito elevado. Para evitar este problema, o Word2Vec implementa o hiperparâmetro tamanho da janela. Por simplicidade, na Figura 27, o tamanho da janela é 2, mas geralmente está entre 5 e 10.

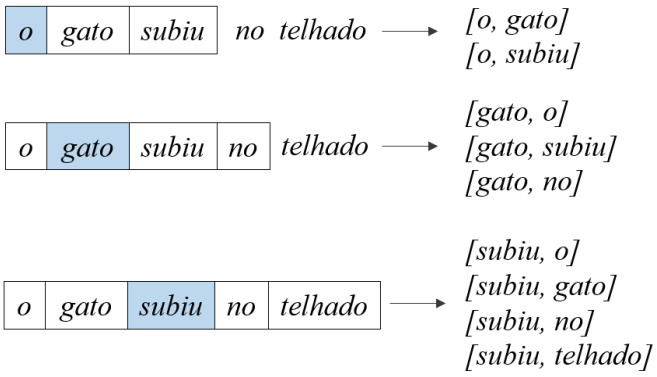


Figura 28: Ilustração da janela de contexto com tamanho 2
Fonte: Do autor

Para atualizar os pesos da rede as matrizes WI e WO são inicializadas seguindo algum padrão e atualizadas através do mecanismo *backpropagation*. Esse método consiste em recalcular os pesos com base no erro, ou seja, a diferença entre a saída da rede e a saída esperada em cada caso de treino. Para isso, é definido uma função de perda que é derivada com relação a cada peso em busca do mínimo global.

Embora utilize redes neurais rasas, os modelos do Word2Vec são capazes de capturar características altamente relevantes e com baixo custo computacional. Em um conjunto bem treinado de dados, é possível empregar operações algébricas que evidenciam regularidades linguísticas:

$$\text{vetor}(\text{“rei”}) - \text{vetor}(\text{“homem”}) + \text{vetor}(\text{“mulher”}) \simeq \text{vetor}(\text{“rainha”})$$

Neste clássico exemplo é possível extrair a seguinte relação: *“homem está para mulher assim como rei está para rainha”* [1].

2.6 ELMo – Representação de Modelos de Linguagem

O ELMo (Embeddings from Language Models) (Peters et al., 2018) é um modelo de linguagem profundo capaz de capturar características complexas do uso de palavras, por exemplo, sintaxe e semântica e como esses usos variam em diferentes contextos, ou seja, para modelar a polissemia (ALLENNLP, 2018). Sua aplicação aprimora o desempenho em diversas tarefas, como resposta a perguntas, rotulação de papéis semânticos, análise de sentimentos e reconhecimento de entidades nomeadas.

O artigo oficialmente publicado na NAACL (North American Chapter of the Association for Computational Linguistics) foi premiado como destaque e causou enorme impacto na comunidade, pois o modelo proposto revolucionou a forma de representar palavras, além de alcançar o estado da arte em todas as tarefas na qual foi testado.

Considere as seguintes frases: “fui ao banco solicitar empréstimo” e “o banco da praça estava sujo”. Este é um claro exemplo de polissemia em que uma palavra pode ter múltiplos significados. As representações de palavras tradicionais, como Word2Vec e GLoVe, geram o mesmo vetor para "banco" nas duas frases. Portanto, não conseguem capturar o contexto em que a palavra foi usada. As representações ELMo resolvem esse problema, pois levam em consideração toda a sentença de entrada. Desta forma, o termo “banco” teria diferentes vetores para diferentes contextos.

A ideia principal do ELMo pode ser dividida em três etapas:

1. O modelo de linguagem bidirecional (biLM) é pre-treinado de forma não supervisionada em um conjunto universal de dados.
2. É feito um ajuste fino (*fine tuning*), no qual o modelo é treinado de maneira supervisionada em um *corpus* específico.
3. O modelo final é usado para gerar, em tempo de execução, uma representação dependente de contexto.

[1] Mais exemplos podem ser encontrados na página oficial: https://code.google.com/archive/p/word2vec/

2.6.1 Modelo de Linguagem Bidirecional

A abordagem tradicional na construção de modelos de linguagem consiste em prever uma palavra, considerando as palavras anteriores. Neste cenário, pode-se utilizar uma Rede Neural Recorrente (RNR) que fornece um contexto ilimitado à esquerda. Porém, para capturar o real significado de uma palavra é primordial efetuar uma análise em ambos os sentidos. Para solucionar este problema, foi criado o Modelo de Linguagem Bidirecional, popularmente conhecido como biLM (Bidirectional Language Model). Este componente é o pilar para as representações ELMo.

O biLM é completamente independente de tarefa, sendo treinado de maneira não supervisionada. Esta característica é fundamental, pois a quantidade de dados não rotulados é vasta e não requer grande esforço humano.

Considere uma entrada com n *tokens* $(t_1, t_2, \dots, t_n)$, o modelo de linguagem para frente (*forward)* analisa palavras antes do *token* alvo, ou seja, calcula a probabilidade do *token* t_k, dado o histórico $(t_1, \dots, t_{k-1})$:

$$p(t_1, t_2, \dots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, \dots, t_{k-1})$$

Equação 6

O modelo de linguagem para trás (*backward)* faz o processo inverso, ou seja, prevê o *token* anterior, dado o contexto futuro:

$$p(t_1, t_2, \dots, t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, \dots, t_N)$$

Equação 7

Um biLM combina os modelos de linguagem *forward* e *backward*. A fórmula elaborada pelos autores, maximiza conjuntamente a probabilidade de log para cada uma das direções. Na Equação 8, os parâmetros foram amarrados em ambas as direções para a representação de *token* (Θ_x) e a camada *Softmax* (Θ_s), enquanto os demais (estado oculto, gate e memória) são separados.

$$\sum_{k=1}^{N} (log\ p(t_k \mid t_1, \dots, t_{k-1}; \Theta_x, \overrightarrow{\Theta}\ LSTM, \Theta_s) + log\ p(t_k \mid t_{k+1}, \dots, t_N; \Theta_x, \overleftarrow{\Theta}\ LSTM, \Theta_s)$$

Equação 8

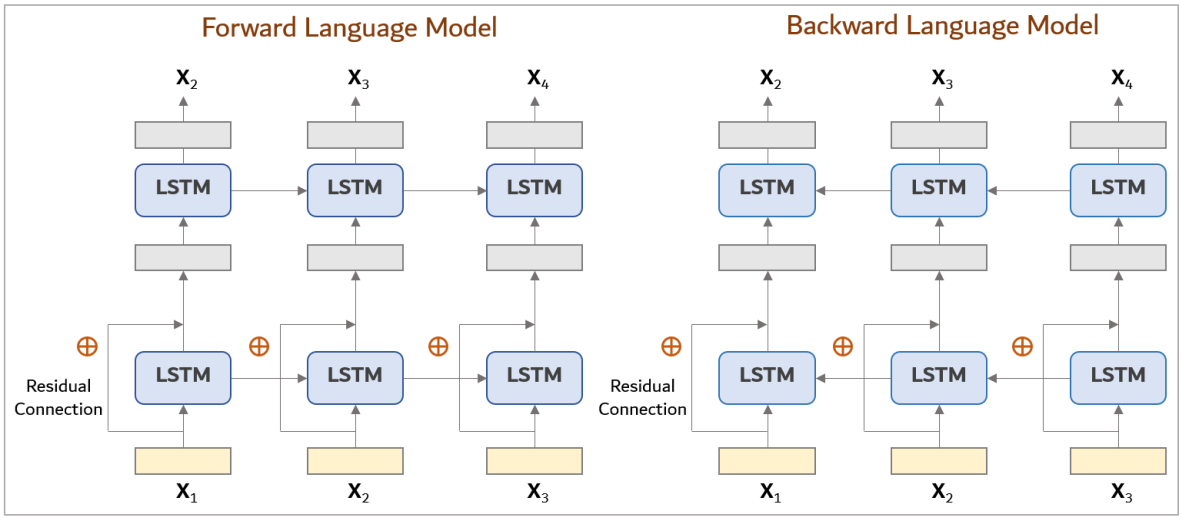


Figura 29: Modelo de Linguagem Bidirecional
Fonte: Createmomo (2018)

O modelo final usa duas camadas LSTM bidirecionais, com 4096 unidades e 512 dimensões, além de uma conexão residual entre a primeira e a segunda camada (Figura 29). A intuição de alto nível é que as conexões residuais ajudam os modelos profundos a treinar com mais êxito e com menor custo computacional.

2.6.2 Representação Contextualizada de Palavras

O ELMo empilha todos os estados ocultos entre as camadas, aprendendo uma combinação linear específica de tarefas. Para cada token t_k, um biLM calcula um conjunto de $2L + 1$ representações, sendo L o número de camadas LSTM e 1 a saída da camada independente de contexto. Na *Equação 9* $h_{k,0}^{LM}$ é a camada *token* e $h_{k,j}^{LM} = [\overrightarrow{h}_{k,j}^{LM}; \overleftarrow{h}_{k,j}^{LM}]$ para cada camada biLSTM.

$$R_k = \{x_k^{LM}, \overrightarrow{h}_{k,j}^{LM}, \overleftarrow{h}_{k,j}^{LM} \mid j = 1, \dots, L\} = \{h_{k,j}^{LM} \mid j = 0, \dots, L\}$$

Equação 9

Para inclusão em uma tarefa final, o ELMo combina todas as camadas de R em um único vetor, $ELMO_k = E(R_k; \Theta_e)$. De maneira geral, é calculado uma ponderação específica de tarefas em todas as camadas do biLM.

$$ELMo_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_k^{task} \; h_{k,j}^{LM}$$

Equação 10

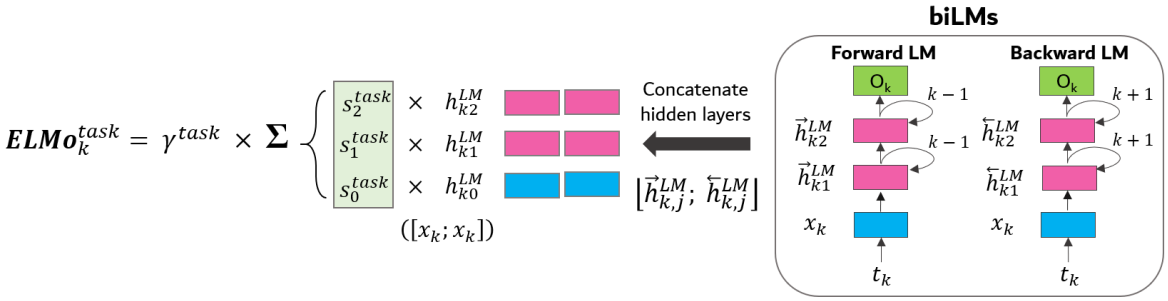


Figura 30: Ilustração para cálculo da representação ELMO
Fonte: Shuntaro (2018)

Na Equação 10, o índice k corresponde ao índice da palavra e j ao índice da camada na qual o estado oculto está sendo extraído. Desta forma, $h_{k,j}$ é a saída do *j-ésimo* LSTM para o *token* k e s_j é o peso de $h_{k,j}$ no cálculo da representação para o *token* k. O parâmetro s^{task} são pesos *softmax* normalizados e γ^{task} permite que o modelo final dimensione todo o vetor ELMo. O parâmetro escalar γ é considerado parte essencial no processo de otimização, pois sem ele o desempenho é ruim e o treinamento pode falhar completamente.

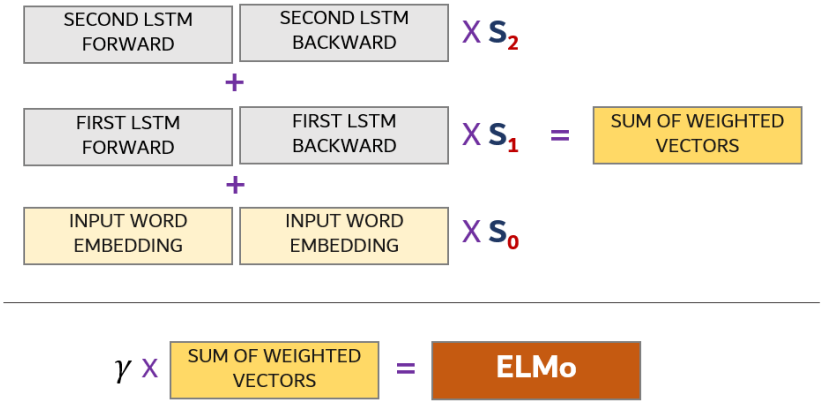


Figura 31: Etapas para o cálculo das representações ELMo
Fonte: Adaptado de Createmomo (2018)

Na Figura 31, a saída para cada camada oculta é concatenada e multiplicada por pesos de uma tarefa específica. Ao final, os vetores ponderados são somados e multiplicados pelo parâmetro escalar γ, que formam a representação ELMo.

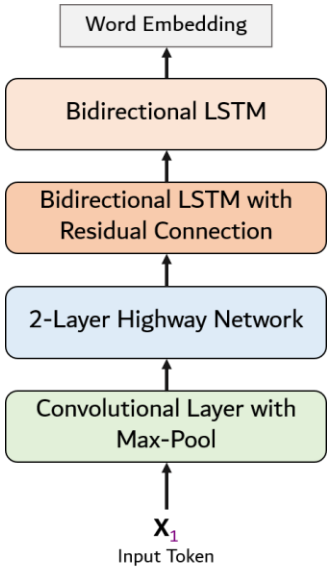


Figura 32: Visão alto nível da arquitetura ELMo
Fonte: Do autor.

Conforme descrito no artigo e ilustrado na Figura 32, o modelo final usa camadas duas camadas biLSTM com 4096 neurônios, uma projeção de 512 dimensões e uma conexão residual da primeira à segunda camada. A representação insensível ao contexto utiliza uma camada CNN com filtros convolucionais de 2048 caracteres *n-gram*, seguidos por duas camadas *highways* (SRIVASTAVA, GREFF e SCHMIDHUBER, 2015) e uma projeção linear de 512 dimensões. Como resultado, o biLM fornece três representações para cada *token*, incluindo aqueles fora do conjunto de treinamento. Isto é possível através da rede neural convolucional puramente baseada em caracteres, que permite à rede usar pistas morfológicas para formar representações robustas de *tokens*.

TASK	PREVIOUS SOTA		OUR BASELINE	ELMO + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

Tabela 1: Comparativo dos resultados obtidos
Fonte: Peters et al. (2018)

Conforme demonstrado na Tabela 1, a adição do ELMo melhorou significativamente o estado da arte em diversas tarefas. Para tal comparativo, é levado em consideração a pontuação F_1, que combina precisão e recall, indicando a qualidade geral do modelo.

2.6.3 Modelos pré-treinados

O ELMo disponibiliza vários modelos pré-treinados no idioma inglês. Cada modelo é especificado com dois arquivos separados, um arquivo de opções no formato JSON e outro com os hiperparâmetros e um arquivo no formato hdf5 com os pesos do modelo.

Modelo	**Parâmetros (Milhões)**	**Tamanho LSTM Oculto / Tamanho Saída**	**Camadas Highway**	**SRL F1**
Pequeno	13,6	1024/128	1	83,62
Médio	28,0	2048/256	1	84,04
Original	93,6	4096/512	2	84,63
Original (5,5B)	93,6	4096/512	2	84,93

Tabela 2: Comparativo entre os modelos ELMo pré-treinados.[2] A coluna "SRL F1" contém a pontuação F1 obtida na tarefa SRL (Semantic Role Labeling) para rotulagem de funções semânticas.

Todos os modelos descritos na Tabela 2, exceto o "Original (5.5B)", foram treinados no conjunto de dados "*1 Billion Word Benchmark*", com aproximadamente 800 milhões de *tokens* extraídos de notícias do WMT 2011. O modelo ELMo 5.5B foi treinado em um conjunto de dados de 5,5 bilhões de *tokens* do Wikipédia (1,9B) e todas as notícias monolíngues do WMT 2008-2012 (3.6B). Nos testes realizados pelos autores, o modelo 5.5B tem desempenho pouco superior em relação ao original, por isso é recomendado o uso do modelo padrão.

2.7 BERT - Bidirectional Encoder Representations from Transformers

Inicialmente os modelos de linguagem eram treinados da esquerda para a direita ou da direita para a esquerda, o que tornam suscetíveis a erros devido à perda de informações. O ELMo tentou corrigir o problema ao treinar e concatenar a saída de dois modelos de linguagem nos contextos a esquerda e a direita. Embora tenha sido uma grande evolução, esta abordagem possui algumas limitações. Como alternativa, foi proposto o BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2018), um modelo de linguagem profundamente bidirecional baseado na arquitetura Transformer (Vaswani et al., 2017).

Conforme (HOREV, 2018), em sua concepção original, o Transformer inclui dois mecanismos separados: um codificador que lê a entrada de texto e um decodificador que produz uma previsão. Como o objetivo do BERT é gerar um modelo de linguagem, apenas o mecanismo do codificador é necessário.

[2] Material para cada modelo pré-treinado disponível em https://allennlp.org/elmo

Assim como no ELMo, o BERT pode ser dividido em três etapas principais:

1. O modelo de linguagem profundamente bidirecional é pre-treinado de forma não supervisionada em um conjunto universal de dados.
2. É realizado um ajuste fino (*fine tuning*), no qual o modelo é incorporado em uma tarefa específica e treinado de maneira supervisionada.
3. O modelo final é usado para gerar, em tempo de execução, uma representação dependente de contexto.

2.7.1 Arquitetura do Modelo

A arquitetura do BERT consiste em um codificador bidirecional de várias camadas baseado no Transformer. Segundo os autores, a implementação é idêntica ao demonstrado no artigo original (seção 2.4) e foi dividida em dois tamanhos:

- $BERT_{BASE}: L = 12, H = 768, A = 12, Total\ Parameter = 110M$
- $BERT_{LARGE}: L = 24, H = 1024, A = 16, Total\ Parameter = 340M$

O número de blocos do Transformer é denotado como L, o estado oculto como H e o número de "cabeças" de atenção como A. Para ambas variações, o tamanho do *feedforward* foi definido como $4H$, ou seja, 3072 para $H = 768$ e 4096 para $H = 1024$.

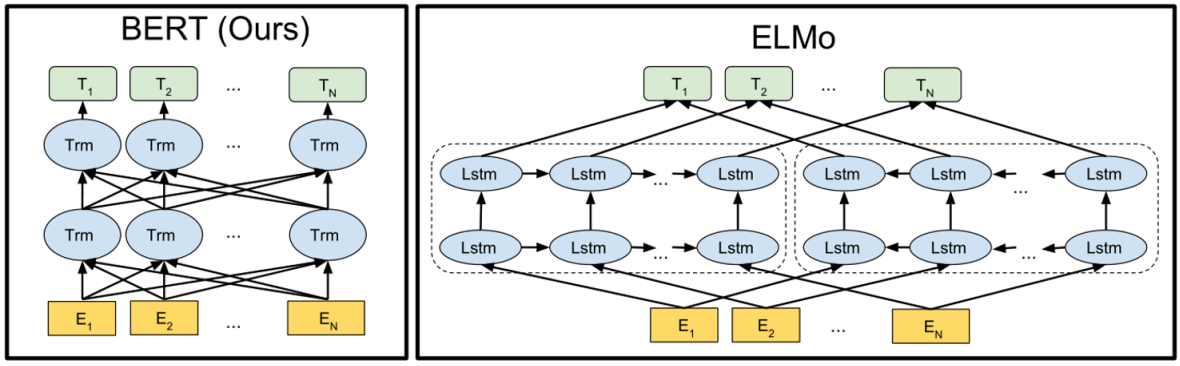


Figura 33: Comparativo entre o BERT e o ELMo
Fonte: Adaptado de Devlin et al. (2018)

Na Figura 33 as setas indicam o fluxo de informações entre as camadas. E_x é o *token* de entrada e T_x a representação contextualizada final para cada *token* de entrada. Trm é o transformer e $Lstm$ a memória de longo prazo. A partir deste comparativo podemos notar que: o BERT é profundamente bidirecional, pois cada camada captura os contextos à esquerda e a direita; o ELMo é superficialmente bidirecional, pois os contextos à esquerda e a direita são capturados por camadas individuais e concatenados na saída.

Para gerar uma representação contextualizada a cada *token* de entrada, os autores descrevem as seguintes características:

- Foi utilizado um vocabulário com 30.000 *tokens* do WordPiece (Wu et al., 2016).
- As representações posicionais aprendidas têm comprimento máximo de 512 tokens.
- O primeiro *token* de cada sequência é sempre [CLS]. O estado oculto final (saída do Transformer) correspondente a este *token* é usado como a representação de sequência agregada para tarefas de classificação. Para tarefas de não classificação, esse vetor é ignorado.
- Os pares de frases são agrupados em uma única sequência, mas separados pelo *token* especial [SEP]. Além disso, é adicionado uma frase *A* e *B* aprendidas e incorporadas a cada token da primeira e segunda frase, respectivamente.
- Para entradas de sentença única, é usado apenas a frase *A* incorporada.

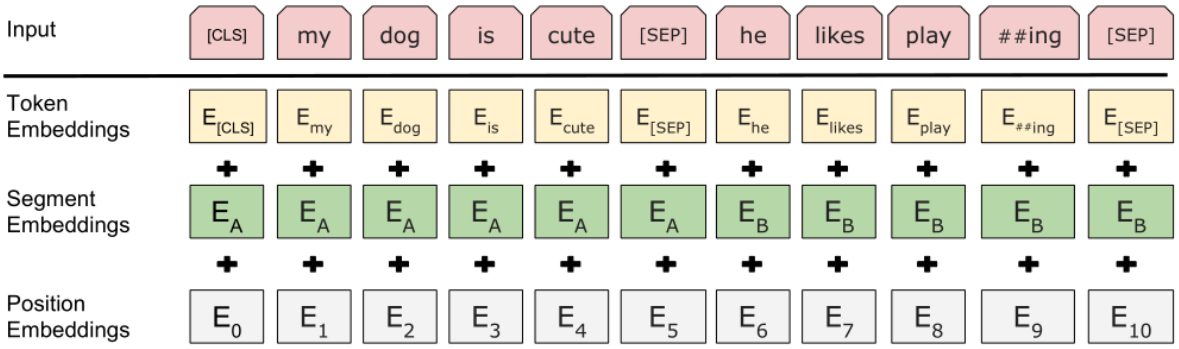


Figura 34: Representação de uma entrada com BERT
Fonte: Devlin et al. (2018)

Conforme ilustrado na Figura 34, cada entrada é caracterizada pela combinação de 3 representações:

- **Position Embeddings:** São representações posicionais aprendidas pelo BERT para expressar a posição das palavras em uma frase.
- **Segment Embeddings:** Como o BERT aceita pares de frases como entrada ele aprende uma representação única para que seja possível distingui-las. Todos os *tokens* marcados como E_A pertencem à sentença A e todos os *tokens* marcados como E_B pertencem a sentença B.
- **Token Embeddings:** São representações aprendidas para um *token* específico no vocabulário do WordPiece.

2.7.2 Pré-treinamento

Para realizar o pré-treinamento do BERT são utilizadas duas estratégias de previsão descritas nas seções 2.7.3.1 e 2.7.3.2.

2.7.3 Modelos pré-treinados

Os autores do BERT disponibilizaram os seguintes modelos pré-treinados:

Description	**Layers**	**Hidden States**	**Attention Heads**	**Parameters (millions)**
BERT-Base, Uncased	12	768	12	110M
BERT-Large, Uncased	24	1024	16	340M
BERT-Base, Cased	12	768	12	110M
BERT-Large, Cased	24	1024	16	340M
BERT-Base, Multilingual Cased with 104 languages	12	768	12	110M
BERT-Base, Multilingual Uncased with 102 languages	12	768	12	110M
BERT-Base, Chinese	12	768	12	110M

Tabela 3: Modelos pré-treinados - BERT

BERT-Base e BERT-Large se referem ao tamanho da arquitetura; Uncased significa que o texto foi convertido para minúsculas antes da tokenização no WordPiece; Cased significa que letras maiúsculas e minúsculas foram preservadas; Multilingual é utilizado para tarefas multilinguagem (tradução automática, por exemplo).

2.7.3.1 Modelo de Linguagem Mascarado

O modelo de linguagem mascarado (Masked LM ou MLM) substitui aleatoriamente uma porcentagem de *tokens* por [MASK] e tenta prever o valor original com base no contexto fornecido pelas palavras não mascaradas. Os autores destacaram que esta abordagem possui duas desvantagens:

A primeira é que o *token* [MASK] nunca será visto durante o ajuste fino. Para atenuar o problema, 15% dos *tokens* são escolhidos aleatoriamente e ao invés de sempre substituir por [MASK], o gerador de dados irá:

1. 80% do tempo substituir as palavras pelo token [MASK]. Exemplo: “meu cachorro é peludo” → “meu cachorro é [MASK]”.
2. 10% do tempo substituir a palavra por outra aleatoriamente. Exemplo: “meu cachorro é peludo” → “meu cachorro é maçã”.
3. 10% do tempo manter a palavra inalterada. O objetivo é influenciar a representação para a palavra observada. Exemplo: “meu cachorro é peludo” → “meu cachorro é peludo”.

Como destacado pelos autores, o codificador do transformer não sabe quais palavras serão solicitadas a prever ou quais foram substituídas por outras aleatoriamente, portanto, é forçado a manter uma representação contextual distributiva de cada *token* de entrada. Além disso, como a substituição aleatória ocorre em apenas 1,5% de todos os *tokens*, não prejudica a capacidade de entendimento do modelo.

A segunda desvantagem é que um MLM converge mais devagar que um modelo de linguagem da esquerda para a direita, porém os resultados obtidos superam o alto custo de treinamento.

2.7.3.2 Previsão da Próxima Frase

Como os modelos de linguagem não são capazes de capturar diretamente o relacionamento entre duas frases, o BERT implementa um modelo chamado Next Sentence Prediction (previsão da próxima frase) que mostrou ser fundamental nas tarefas de resposta a perguntas (QA) e inferência de linguagem natural (NLI).

No processo de treinamento, o modelo recebe pares de frases como entrada e aprende a prever se a segunda frase é a frase subsequente no conjunto de dados original. Ao escolher as frases A e B para cada exemplo de pré-treinamento, 50% do tempo B é a próxima frase e 50% uma frase aleatória do corpus. A suposição é que a sentença aleatória será desconectada da primeira sentença.

O seguinte exemplo é dado pelos autores:

$$InputSentence = [CLS]\ the\ man\ went\ to\ [MASK]\ store\ [SEP]$$
$$he\ bought\ a\ gallon\ [MASK]\ milk\ [SEP]$$
$$SentenceLabel = \boldsymbol{IsNext}$$

$$InputSentence = [CLS]\ the\ man\ [MASK]\ to\ the\ store\ [SEP]$$
$$penguin\ [MASK]\ are\ flight\ \#\#less\ birds\ [SEP]$$
$$SentenceLabel = \boldsymbol{NotNext}$$

$IsNext$ diz que a sentença B é a correta, e $NotNext$ uma sequência aleatória. Ao escolher as frases $NotNext$ de forma completamente aleatória, o modelo final pré-treinado atingiu entre 97% e 98% de precisão (Devlin et al., 2018).

3 Desenvolvimento do Trabalho Proposto

Durante o processo de desenvolvimento foi realizado um levantamento e estudo dos artigos publicados para cada um dos três modelos abordados neste trabalho. Após o entendimento e resumo dos principais conceitos, características e nível de complexidade, foi decidido começar o desenvolvimento pelo ELMo. A justificativa é que o Word2Vec trata-se de um modelo relativamente simples e consagrado na literatura, enquanto o BERT é o mais recente e complexo. Desta forma, a escolha se justifica pelo fato do ELMo estar em um nível intermediário dentre os três modelos.

Para iniciar a implementação do ELMo foi revisado o material de referência disponível no GitHub [3]. O código foi escrito na linguagem Python sob o apoio de uma biblioteca de alto nível chamada TensorFlow.

O TensorFlow é uma biblioteca Python desenvolvida pela Google que reúne vários modelos e algoritmos de aprendizado de máquina. Sua estrutura é composta por grafos, onde cada nó representa uma operação matemática e cada conexão é uma matriz multidimensional, conhecida como tensor (YEGULALP, 2019).

Após um longo período necessário para entender e se familiarizar com a estrutura do código e organização dos pacotes, foram realizadas tentativas de pré-treinamento do modelo sobre uma base de dados conhecida como “Yelp Dataset” que contém comentários sobre restaurantes e comerciantes dos EUA. O procedimento resultou nas seguintes dificuldades:

- incompatibilidade entre versões diferentes do Python e as bibliotecas do projeto;
- falha na instalação de dependências, sendo necessário utilizar um sistema de gerenciamento de pacotes conhecido como Anaconda Cloud;
- interrupções durante a execução devido à falta de memória (posteriormente este problema foi sanado aumentando a capacidade de 8GB para 16GB).

Após sanar os problemas, notou-se que o treinamento leva muito tempo para ser concluído. Como alternativa, foi analisado a documentação do ELMo e BERT em busca de informações a respeito do custo computacional envolvendo os modelos de linguagem recentes. Conforme esperado, as seguintes informações foram extraídas:

- ELMo [4]: O modelo original foi pré-treinado em 3 GTX 1080 por 10 *epochs*, levando cerca de duas semanas.
- BERT [5]: Os modelos foram pré-treinados em aproximadamente 4 dias utilizando entre 4 a 16 TPUs. TPU é uma unidade de processamento tensorial com alta

[3] https://github.com/allenai/bilm-tf
[4] https://github.com/allenai/bilm-tf/blob/master/README.md
[5] https://github.com/google-research/bert/blob/master/README.md

capacidade de processamento desenvolvida e otimizada especificamente para o TensorFlow.

Além disso, utilizando a mesma fonte, os autores do BERT afirmam que a maioria dos pesquisadores dificilmente irão precisar pré-treinar o modelo a partir do zero. Desta forma, levando em consideração a complexidade e a necessidade em comparar os modelos diante de um mesmo cenário, a decisão foi utilizar os modelos pré-treinados sugerido pelos autores.

Antes de incorporar o ELMo, foi preciso escolher uma base de dados rotulada que será descrita na seção 3.1. E posteriormente construir uma rede neural simples destinada a classificação de sentimentos:

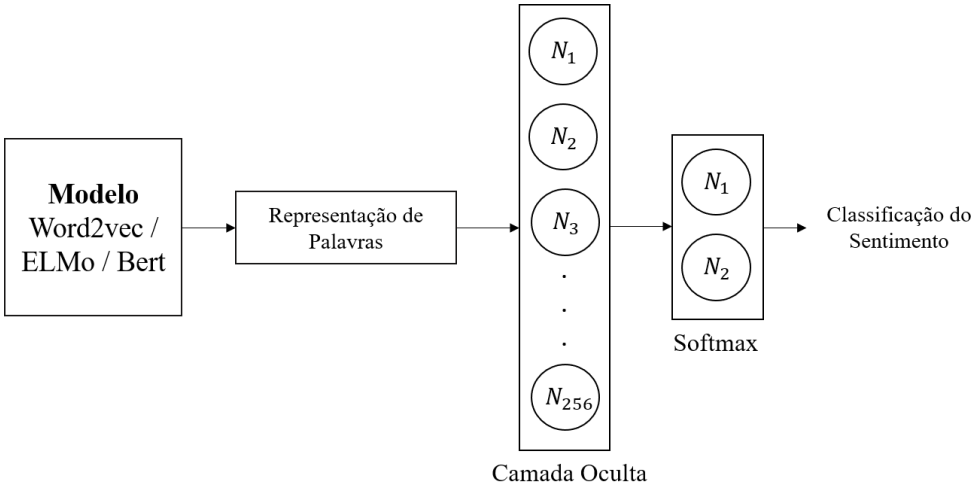


Figura 35: Estrutura da rede para análise de sentimentos

Na Figura 35 o ELMo e o BERT se referem a uma rede neural pré-treinada com seus respectivos pesos e hiperparâmetros. No caso do Word2Vec, existe uma camada adicional responsável por receber os dados de entrada (vetores de palavras). A camada oculta possui 256 neurônios totalmente conectados e a camada de saída 2 neurônios ativados por uma função *softmax*. Os hiperparâmetros serão descritos na seção 3.2.

Entender e implementar cada componente de uma rede neural profunda é uma tarefa complexa que exige cuidado e experiência por parte do desenvolvedor. Desta forma, para reduzir a complexidade e facilitar os testes, foi utilizado o Keras, uma biblioteca Python de alto nível, com suporte ao TensorFlow e que permite o uso e a criação de algoritmos de aprendizado de máquina. Seu objetivo é acelerar e facilitar a implementação de algoritmos, principalmente com redes neurais, no qual componentes podem ser adicionados e removidos com poucas linhas de código (HELLER, 2019).

Para facilitar a incorporação os autores do ELMo disponibilizaram os modelos pré-treinados no TensorFlow Hub, uma biblioteca para publicação, descoberta e consumo de modelos de aprendizado de máquina. Cada módulo consiste em um grafo TensorFlow

juntamente com os pesos e demais configurações da rede. Neste repositório foi publicado somente o modelo original treinado no Wikipedia [6].

Para processar as entradas, o módulo ELMo possui duas assinaturas:

- default: aceita uma lista de frases completas;
- tokens: aceita uma lista de frases tokenizadas.

Quanto as saídas, é possível capturar:

- word_emb: representações de palavras baseadas em caracteres;
- lstm_outputs1: a saída do primeiro LSTM;
- lstm_outputs2: a saída do segundo LSTM;
- elmo: soma ponderada das 3 camadas do modelo;
- default: um conjunto de médias fixo de todas as representações de palavras contextualizadas.

Para os testes realizados neste trabalho, foram definidas as assinaturas *default* tanto para entrada quanto para saída. Como o Keras e o TensorFlow são compatíveis, foi possível baixar o módulo diretamente do TensorFlow Hub e incorporar o modelo na rede desenvolvida para análise de sentimentos.

Após concluir o treinamento e o teste da rede com o ELMo, o próximo passou foi realizar os mesmos procedimentos com o BERT. O modelo pre-treinado escolhido possui apenas 12 camadas (BERT-Base Uncased) [7], isto porque a versão de 24 camadas (BERT-Large) exige um hardware mais robusto.

Embora os modelos do BERT também estejam disponíveis no TensorFlow Hub, houve problemas ao tentar configurá-los. Pois diferentemente do ELMo, cada frase deve ser pré-processada e convertida em um formato específico. Por se tratar de uma tarefa que causa confusões, os autores disponibilizaram um arquivo de exemplo no repositório oficial [8].

Em termos gerais é necessário criar uma instância do objeto tokenizer responsável por converter frases de entrada em uma sequência de *tokens*, além de gerar identificadores para cada elemento do vocabulário. Após este procedimento, cada frase é convertida em três vetores distintos:

- input_ids: uma sequência de números inteiros identificando cada token gerado pelo tokenizer;
- input_masks: cada posição referente a um *token* é preenchida com zero;
- segment_ids: para entradas de frase única, caso deste trabalho, todas as posições são preenchidas com zero; para entradas com duas frases, a posição de cada token

[6] https://tfhub.dev/google/elmo/2
[7] https://tfhub.dev/s?q=bert
[8] https://github.com/google-research/bert/blob/master/run_classifier.py

da primeira frase é preenchida com zero e a posição de cada token da segunda frase é preenchida com 1.

O tamanho dos três vetores descrito acima deve ser igual. Neste trabalho são 256 posições e as não utilizadas são preenchidas com zero.

Com relação ao Word2Vec não é preciso incorporá-lo a uma rede neural, isto porque sua saída é armazenada em um arquivo conhecido como dicionário de palavras. Desta forma, foi necessário adicionar uma camada de entrada responsável por receber os vetores e repassa-los para a camada oculta.

Para equilibrar o cenário foi utilizado um dicionário de palavras extraído a partir de um treinamento realizado no Wikipedia, chamado Wikipedia2vec[9]. O arquivo final foi gerado por uma rede que utiliza janela de tamanho 10 e vetores de dimensão 500.

Após finalizar o treinamento e teste da rede utilizando o dicionário de palavras, foram constatadas diferenças significativas quando os dados para treinamento e teste são escolhidos aleatoriamente. Afim de solucionar este problema, foram realizadas pesquisas em que a maior parte dos autores sugerem utilizar métodos de validação cruzada. A justificativa é que a base de dados escolhida é relativamente pequena e possui discrepância entre o número de rótulos positivos e negativos. Além disso, como cada amostra passa tanto pelo treinamento quanto pelo teste, obtemos um modelo menos tendencioso ao realizar a estimativa final, o que permite realizar um comparativo mais próximo da realidade.

A técnica consiste em dividir o conjunto de dados em k folds e percorrer um loop com k iterações. Para cada iteração é realizado o treinamento sobre $k - 1$ *folds* e as amostras que restaram são utilizadas para teste. As métricas são armazenadas e o procedimento é repetido até que cada *fold* sirva como conjunto de teste. Ao final, é calculado a média das pontuações gravadas que representa o desempenho final do modelo. O fluxo realizado neste trabalho pode ser observado na Figura 36.

[9] http://wikipedia2vec.s3.amazonaws.com/models/en/2018-04-20/enwiki_20180420_500d.txt.bz2

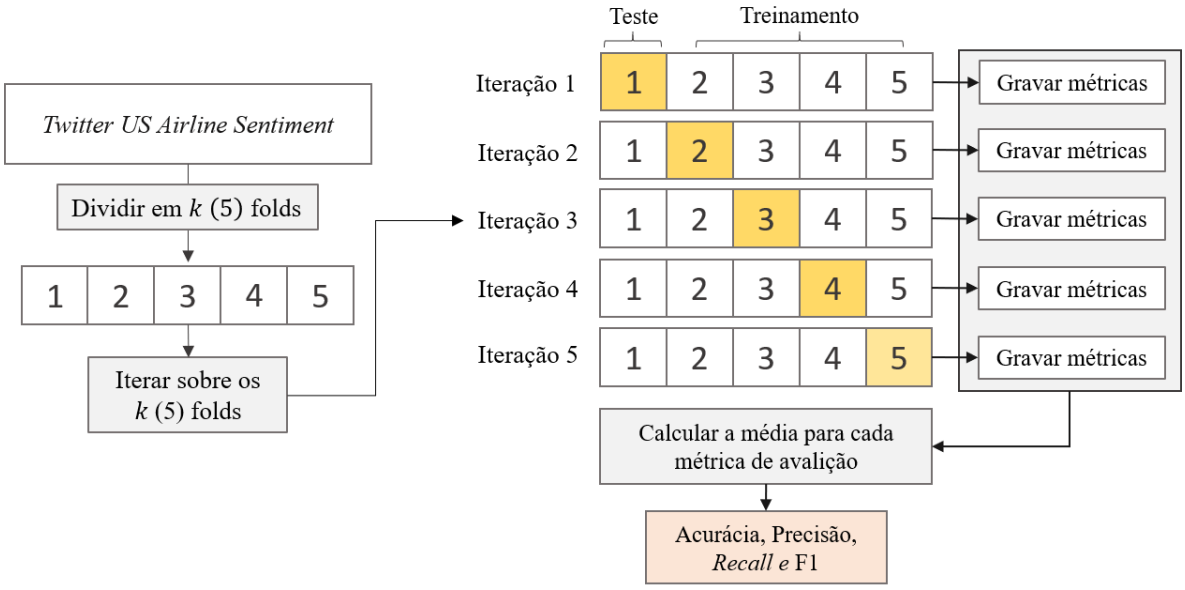


Figura 36: Validação cruzada realizada pelo autor
Fonte: Do autor.

3.1 Pré-processamento

A base de dados escolhida para treinamento e classificação de sentimentos foi retirada do Kaggle como "*Twitter US Airline Sentiment*" [10] e contém comentários que expressam o sentimento de passageiros em relação as principais companhias aéreas dos EUA.

O Kaggle é uma comunidade para cientistas de dados atualmente mantida pela Google. Sua plataforma online é tida como *crowdsourcing* (terminologia para produção de conhecimento coletivo) e tem como objetivo atrair, nutrir, treinar e desafiar cientistas de dados de todo o mundo para resolver problemas de ciência de dados, aprendizado de máquina e análise preditiva (USMANI, 2017).

[10] https://www.kaggle.com/crowdflower/twitter-airline-sentiment/

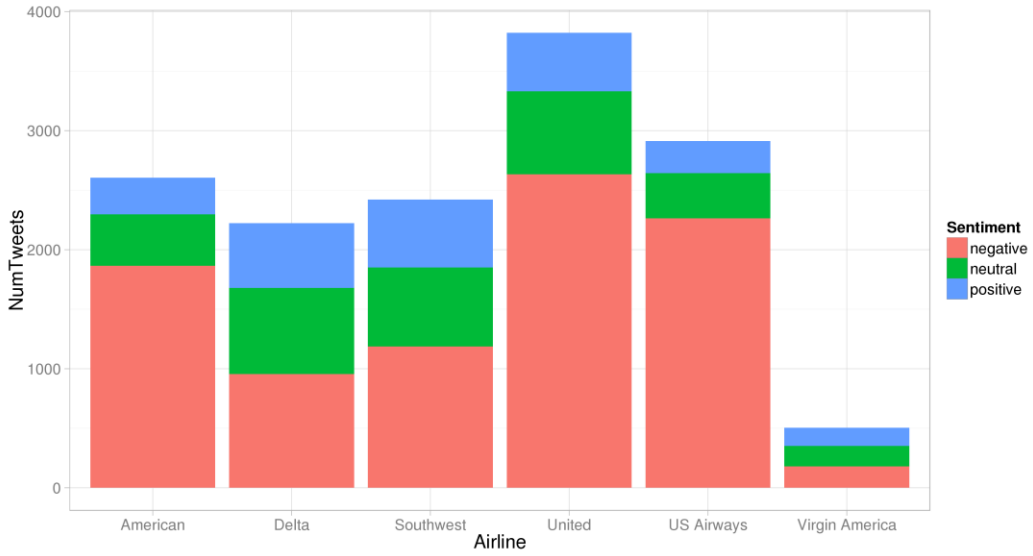


Figura 37: Classificação dos tweets - Twitter US Airline Sentiment [11]

Os dados foram coletados a partir de fevereiro de 2015 e publicados originalmente pela Crowdflower na biblioteca "*Data for Everyone*". Os colaboradores da Kaggle incluíram um arquivo SQLite e realizaram ajustes na formatação do arquivo CSV. Cada amostra contém se o sentimento do tweet foi positivo, negativo ou neutro para seis companhias aéreas. Uma abstração pode ser observada na Figura 37.

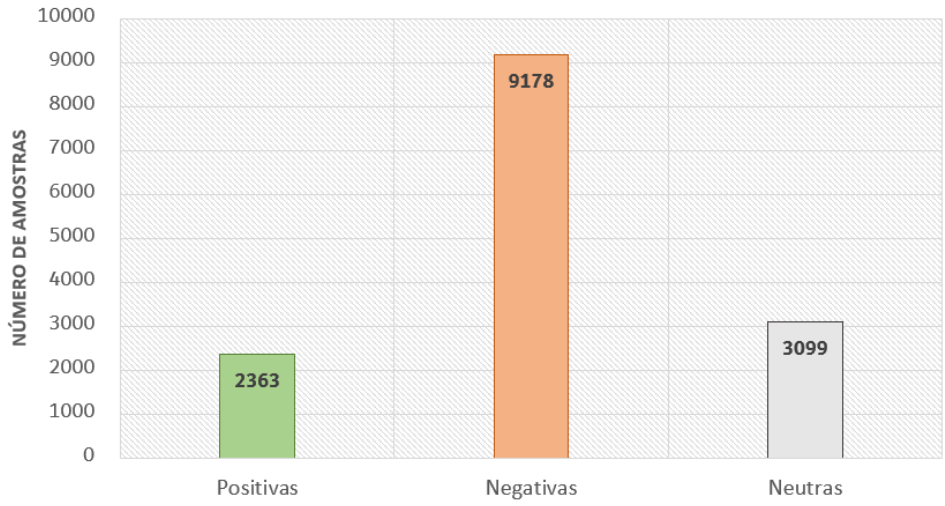


Figura 38: Divisão das amostras - Twitter US Airline Sentiment
Fonte: Do autor.

A contagem dos rótulos é de 2.363 comentários positivos, 9.178 comentários negativos e 3.099 comentários neutros. Para simplificar o comparativo entre os modelos e reduzir o tempo de processamento, os rótulos neutros foram removidos.

[11] https://www.kaggle.com/crowdflower/twitter-airline-sentiment

A qualidade de qualquer algoritmo para aprendizado de máquina começa pela injeção de dados limpos e que sejam representativos. Desta forma, para garantir a qualidade dos modelos, foram realizados os seguintes procedimentos:

- Remoção de hiperligações (*links*)
- Substituição de contrações do inglês, por exemplo: $cant't \rightarrow can\ not$
- Remoção de espaços que se repetem duas ou mais vezes por apenas um
- Remoção de caracteres especiais, como: !@#$%^&* ><

3.2 Hiperparâmetros

Como forma de efetuar análise comparativa em um cenário semelhante, foram utilizados os mesmos hiperparâmetros para cada modelo durante o treinamento. O conceito será abordado nas subseções a seguir e um resumo pode ser observado logo abaixo:

- número de *epochs*: 5
- *batch Size* (tamanho do lote): 128
- função de perda: entropia cruzada categórica
- regularizador L2: 0,001
- otimizador: Adam
    - learning rate (α): 0,001
    - beta1 (β_1): 0,9
    - beta2 (β_2): 0,999
    - épsilon (ϵ): 10^{-8}

A escolha dos parâmetros do Adam se baseia nas configurações testadas e sugeridas pelos autores.

3.2.1 Número de *epochs* e *batch size*

Devido a limitações de memória, não é recomendado propagar todas as amostras por uma rede neural, principalmente em grandes conjuntos de dados. Desta forma, os dados são divididos em lotes (*batch*) que cabem na memória de um computador. Quando todos os lotes são propagados e o treinamento é concluído, temos o conceito de *epochs*, um hiperparâmetro que controla o número de passagens completas pelo conjunto de treinamento.

3.2.2 Entropia cruzada categórica

A perda de entropia cruzada mede o desempenho de um modelo de classificação cuja saída é uma probabilidade entre 0 e 1.

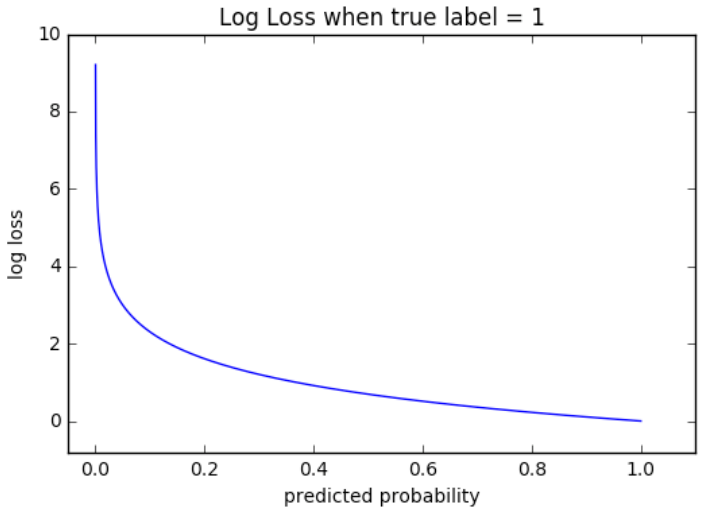


Figura 39: Gráfico para entropia cruzada categórica [12]

Conforme ilustrado na Figura 39, à medida que a probabilidade prevista se aproxima de 1, a perda diminui lentamente, no entanto, à medida que a probabilidade prevista diminui, a perda aumenta rapidamente. Desta forma, caso seja predito uma probabilidade de 0,020 quando o rótulo real é 1, temos um alto valor de perda. Em um modelo perfeito (algo intangível) a perda é 0.

3.2.3 Regularização L2

A regularização é uma técnica utilizada para reduzir a complexidade de um modelo. Isso é feito penalizando a função de perda para resolver o problema de *overfitting*, no qual o modelo se adapta muito bem aos dados de treinamento, mas não consegue generalizar nos dados de avaliação.

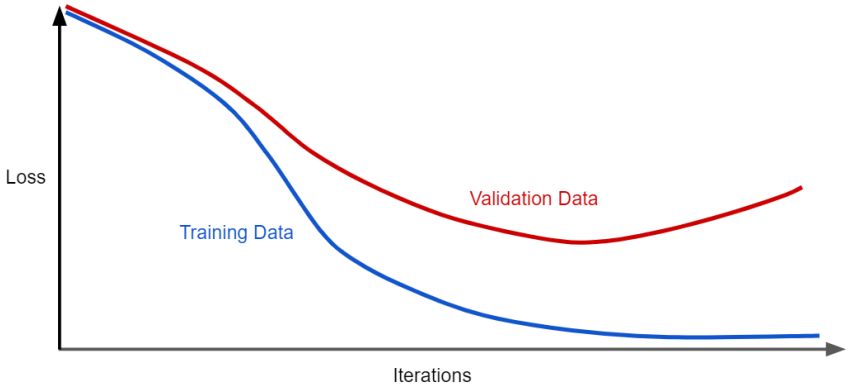


Figura 40: Perda nos conjuntos de treinamento e validação [13]

[12] https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
[13] https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization

Na Figura 40 temos a ilustração de um modelo em que a função de perda diminui gradualmente durante o treinamento, mas que aumenta durante a validação. Isso mostra que o modelo está adaptando-se demais aos dados de treinamento. Uma forma de evitar o excesso de ajustes consiste em penalizar modelos complexos utilizando o princípio de regularização.

No presente trabalho foi utilizado a regularização L2 que força os pesos a decair em direção a zero, mas não exatamente zero.

3.2.4 Taxa de aprendizado

A taxa de aprendizado é um hiperparâmetro que controla o quanto os pesos de uma rede serão ajustados em relação a perda de gradiente. Desta forma, afeta a rapidez com que um modelo converge para um mínimo local. Uma taxa de aprendizado muito baixa torna o aprendizado da rede muito lento, ao passo que uma taxa de aprendizado muito alta provoca oscilações no treinamento e impede a convergência do processo de aprendizado. A Figura 41 demonstra os diferentes cenários ao configurar a taxa de aprendizado.

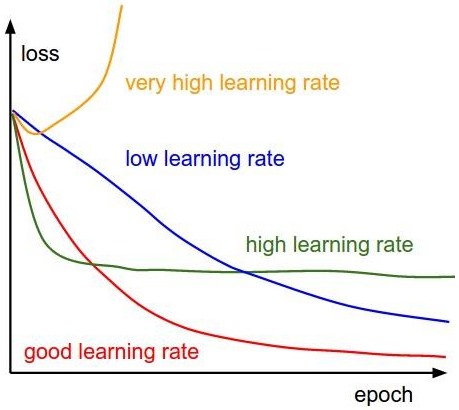
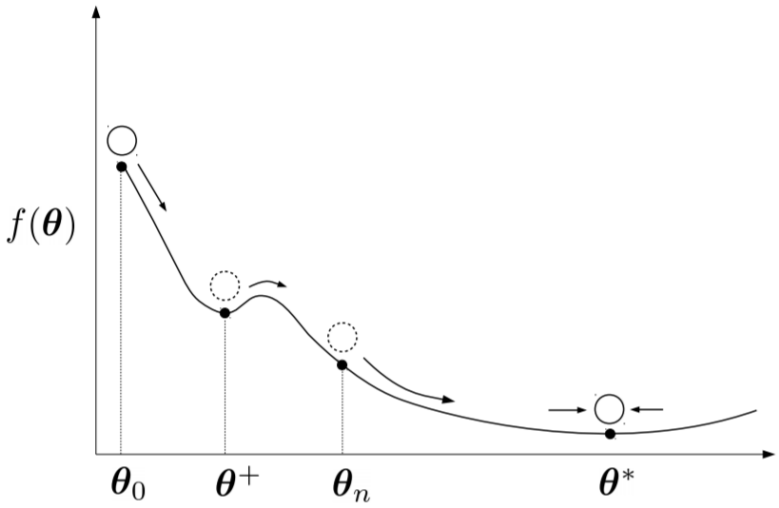


Figura 41: Efeito de várias taxas de aprendizagem durante a convergência
Fonte: Karpathy (2018)

Não existe um padrão ao definir a taxa de aprendizado, desta forma é necessário escolher um valor aleatoriamente ou baseado em experiencias anteriores e ir ajustando de acordo com os resultados.

3.2.5 Otimizador Adam

Adam (*Adaptive Moment Estimation*) (KINGMA; BA, 2015) é um algoritmo de otimização adaptativo e robusto desenvolvido especificamente para o treinamento de redes neurais profundas. Ele se comporta como uma bola pesada com atrito, pois como a média é baseada em gradientes anteriores ele ultrapassa os mínimos locais pequenos e pode encontrar mínimos planos que generalizam bem. A Figura 42 mostra a dinâmica do HBF (Heavy Ball with Friction), onde a bola estabiliza em um mínimo plano (HEUSEL et al., 2018).



- *Figura 42: Analogia entre Adam e uma bola pesada com atrito*
  Fonte: Heusel et al. (2018)

A taxa de aprendizado é mantida para cada peso da rede e se adaptada durante o treinamento. Em termos de configuração, o Adam possui os seguintes hiperparâmetros:

- alfa (α): conhecido como taxa de aprendizado, o alfa representa a proporção em que os pesos da rede são atualizados.
- beta1 (β_1): taxa de decaimento para estimativas do primeiro momento;
- beta2 (β_2): taxa de decaimento para as estimativas do segundo momento;
- épsilon (ϵ): um número muito pequeno que impede qualquer divisão por zero.

Os autores citam as seguintes vantagens ao utilizar o Adam:

- computacionalmente eficiente;
- poucos requisitos de memória;
- adequado para problemas grandes em termos de dados ou parâmetros;
- adequado para problemas com gradientes muito ruidosos ou esparsos;
- hiperparâmetros requerem pouco ajuste.

3.3 Resultados obtidos

Tendo como base a tarefa de análise de sentimentos, nesta seção será mostrado e analisado os resultados para cada modelo abordado neste trabalho. As métricas de avalição utilizadas são *accuracy* (acurácia), *precision* (precisão), *recall* (cobertura) e *F1-Score*.

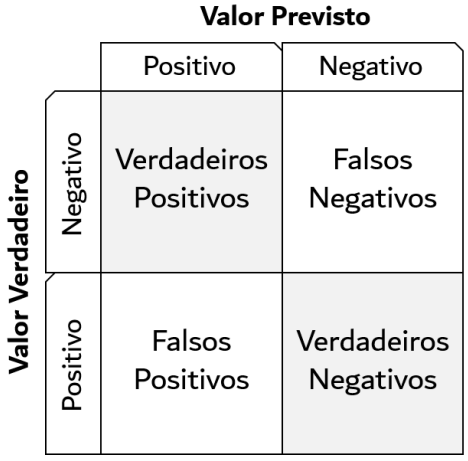


Figura 43: Matriz de confusão

As pontuações são calculadas a partir de uma estrutura conhecida como matriz de confusão. Esta matriz representa a relação entre o valor verdadeiro de uma amostra e o valor efetivamente previsto:

- Verdadeiros positivos representam a classificação correta da classe positivo.
- Falsos negativos representam o erro em que o modelo previu a classe negativo quando o valor esperado era a classe positivo.
- Falsos positivos representam o erro em que o modelo previu a classe positivo quando o valor esperado era a classe negativo.
- Verdadeiros negativos representam a classificação correta da classe negativo.

A medida de acurácia quantifica a performance geral do modelo, ou seja, dentre todas as classificações, quantas o modelo classificou corretamente:

$$Acurácia = \frac{VP + VN}{VP + VN + FP + FN}$$

A precisão representa a quantidade de classificações corretas da classe positivo:

$$Precisão = \frac{VP}{VP + FP}$$

O *Recall* representa a quantidade de acertos para situações onde a classe positiva é esperada:

$$Recall = \frac{VP}{VP + FN}$$

O F1-Score é uma média entre precisão e *recall*. O intuito é trazer um número único que indique a qualidade geral do modelo:

$$\frac{2 \times Precisão \times Recall}{Precisão + Recall}$$

A coluna “Tempo” descrita nas tabelas 5, 6 e 7 representa o tempo total desde o início do treinamento até o fim da predição. Para facilitar a compreensão, as unidades de medida estão em horas (h), minutos (m) e segundos (s).

Todos os testes foram realizados em uma máquina com as seguintes configurações:

- Processador Intel Core i5-7400 3GHz
- Memória RAM Kingston 2x8GB 2133Mhz
- Placa de vídeo Galax GTX 1060 6GB

Dado a complexidade da tarefa e o fato de *tweets* normalmente conterem muitos ruídos, o Word2Vec obteve excelentes pontuações, ainda mais se considerarmos que é gerado somente uma representação para cada palavra. Um ponto importante a ser observado é que houve uma oscilação considerável entre diferentes *folds*, o que ressalta ainda mais a importância de se realizar validação cruzada.

Fold	**Acurácia**	**Precisão**	**Recall**	**F1**	**Tempo (s)**
1	0,90	0,85	0,82	0,84	13,00
2	0,88	0,87	0,79	0,82	12,00
3	0,85	0,85	0,81	0,83	12,00
4	0,93	0,88	0,75	0,79	12,00
5	0,86	0,75	0,88	0,79	12,00
Média	0,88	0,85	0,81	0,82	12,00
				TOTAL	**61,0**

Tabela 4: Métricas de avalição - Word2Vec
Fonte: Do autor.

Analisando a Tabela 5, podemos notar que houve uma grande evolução do ELMo em relação ao Word2Vec. O modelo além de obter uma pontuação F1 maior, teve variações bem discretas entre cada *fold*.

Fold	**Acurácia**	**Precisão**	**Recall**	**F1**	**Tempo (m)**
1	0,91	0,88	0,82	0,84	36,38
2	0,90	0,87	0,86	0,87	36,27
3	0,88	0,87	0,87	0,87	36,38
4	0,93	0,81	0,88	0,84	35,59
5	0,94	0,87	0,88	0,87	36,26
Média	0,91	0,87	0,87	0,87	36,17
				TOTAL	**217,05**

Tabela 5: Métricas de avaliação - ELMo
Fonte: Do autor.

Traçando um comparativo com relação ao ELMo, o BERT obteve melhores resultados, mas com uma pequena diferença. Um ponto importante a ser observado é que o BERT foi testado utilizando a versão com apenas 12 camadas, enquanto que no artigo original foi utilizado a versão com 24 camadas. Desta forma, a pontuação F1 deverá aumentar.

A partir da Tabela 6 podemos notar que o tempo de processamento é extremamente elevado, o que ressalta a complexidade do modelo.

Fold	**Acurácia**	**Precisão**	**Recall**	**F1**	**Tempo (h)**
1	0,94	0,93	0,86	0,89	12,70
2	0,93	0,90	0,90	0,90	12,06
3	0,92	0,91	0,92	0,92	12,07
4	0,93	0,82	0,88	0,84	12,17
5	0,95	0,89	0,90	0,89	12,11
Média	0,93	0,90	0,90	0,89	12,22
				TOTAL	**61,11**

Tabela 6: Métricas de avalição - BERT
Fonte: Do autor.

A Figura 44 ilustra um comparativo entre os três modelos e os valores representam a média final das pontuações para cada um deles.

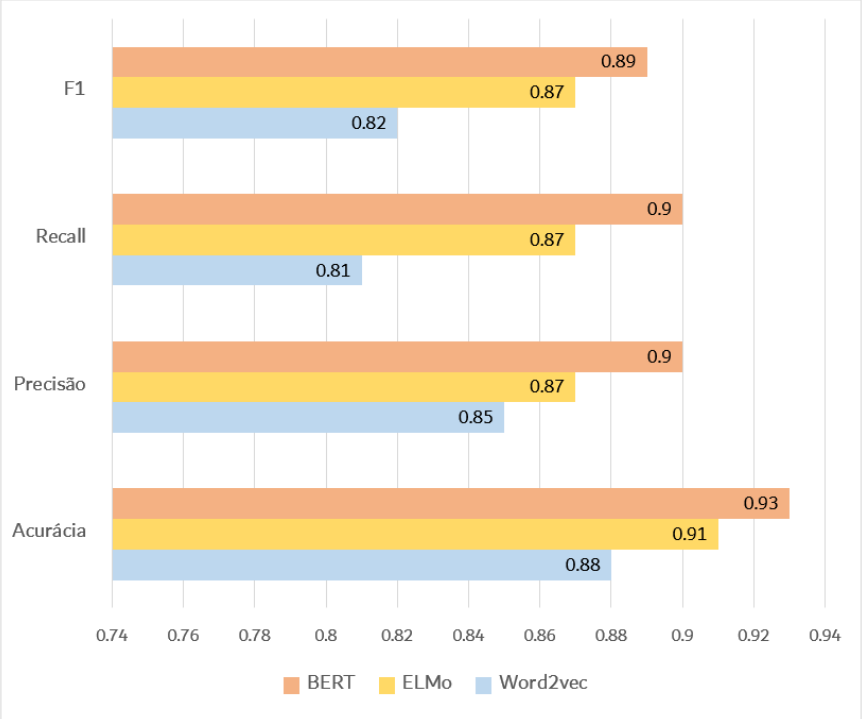


Figura 44: Comparativo entre os modelos Word2Vec, ELMo e BERT
Fonte: Do autor.

4 Considerações Finais

Neste capítulo são apresentadas as conclusões finais, bem como as dificuldades encontradas e sugestões de trabalhos futuros.

4.1 Conclusões

Para que seja possível a utilização de Aprendizado Profundo em atividades de processamento de linguagem e necessário a conversão de palavras em representação numérica. Nesse trabalho foram estudados três sistemas que suportam essa representação. O primeiro é de grande relevância na área devido ao impacto que causou em relação a sua capacidade de capturar relações de similaridade entre palavras. Os demais se propõem a resolver um problema de polissemia não abordado no primeiro. Nesse trabalho, foi realizado uma análise comparativa entre as três propostas estudadas. Com relação aos resultados obtidos, podemos chegar às seguintes conclusões:

- O Word2Vec é um modelo simples e com baixo custo computacional, mas capaz de capturar características complexas do uso de palavras. É uma excelente opção para tarefas em que a polissemia não esteja fortemente presente ou quando o hardware é um fator limitante.
- O ELMo, além de tratar a questão da polissemia, apresenta melhorias na qualidade de predição em relação ao Word2Vec. Ele pode ser aplicado em diversas tarefas de aprendizado de máquina, e dado sua complexidade, é um modelo consideravelmente rápido, conforme os resultados comparativos de tempo de processamento apresentados na seção 3.3.
- Com a utilização do sistema BERT, foram obtidos os melhores resultados em relação as métricas de avaliação utilizadas para comparação dos modelos testados. Sua qualidade de predição pode ser otimizada com expansão do número de camadas utilizadas em sua rede. No entanto, é necessário ter uma máquina com alta capacidade de processamento, pois o custo de treinamento é muito alto.

4.2 Dificuldades Encontradas

Durante o desenvolvimento do trabalho proposto foram encontradas as seguintes dificuldades:

- Os artigos de referências utilizados para estudo e análise das propostas do ELMo e BERT não fornecem informações detalhadas necessários para o entendimento das arquiteturas propostas. Foi necessário recorrer a outras fontes e ao código das implementações utilizadas para um entendimento detalhado das propostas.

- A limitação de hardware contribuiu para um tempo de processamento elevado, principalmente para o BERT, tornando o teste de novas configurações de rede inviável.

4.3 Trabalhos Futuros

No decorrer deste trabalho foi possível identificar algumas questões e dificuldades que podem ser investigadas em trabalhos futuros:

- Aplicar novas técnicas de limpeza de texto afim de reduzir o ruído e aprimorar o treinamento.
- Treinar o modelo BERT-Large utilizando a Google Cloud TPU (Tensor Processing Unit), uma unidade de processamento tensorial disponibilizada em nuvem, desenvolvida e otimizada especificamente para o TensorFlow.
- Efetuar ajustes nos hiperparâmetros para cada modelo de linguagem e alterar a estrutura da rede para análise de sentimentos em busca de melhores resultados.
- Incorporar os modelos abordados neste trabalho em outras tarefas de processamento de linguagem natural, como rotulagem de papéis semânticos, marcação de partes da fala (*part-of-speech tagging*) e respostas a perguntas.

5 Referências

ALAMMAR, J. The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning), 2018. Disponivel em: <http://jalammar.github.io/illustrated-bert/>. Acesso em: 9 nov. 2019.

ALAMMAR, J. The Illustrated Transformer. **Jalammar**, 2018. Disponivel em: <http://jalammar.github.io/illustrated-transformer/>. Acesso em: 23 out. 2019.

ALLENNLP. ELMo. **AllenNLP**, 2018. Disponivel em: <https://allennlp.org/elmo>. Acesso em: 7 out. 2018.

ARDEN, D. Applied Deep Learning - Part 4: Convolutional Neural Networks. **towardsdatascience**, 2017. Disponivel em: <https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>. Acesso em: 9 out. 2019.

BA, J. L.; KIROS, J. R.; HINTON, G. E. Layer Normalization. **arXiv preprint arXiv: 160706450**, 2016.

BENGIO, Y. et al. A Neural Probabilistic Language Model. **Journal of machine learning research**, v. 3, p. 1137-1155.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning Long-Term Dependencies with Gradient Descent is Dificult. **IEEE transactions on neural networks**, v. 5, n. 2, p. 157-166, mar. 2019.

BROWNLEE, J. How To Get Baseline Results And Why They Matter. **Machine Learning Mastery**, 2017. Disponivel em: <https://machinelearningmastery.com/how-to-get-baseline-results-and-why-they-matter/>. Acesso em: 4 out. 2019.

BROWNLEE, J. A Gentle Introduction to k-fold Cross-Validation. **Machine Learning Mastery**, 2018. Disponivel em: <https://machinelearningmastery.com/k-fold-cross-validation/>. Acesso em: 9 nov. 2019.

BROWNLEE, J. A Gentle Introduction to Exploding Gradients in Neural Networks. **Machine Learning Mastery**, 2019. Disponivel em: <https://machinelearningmastery.com/exploding-gradients-in-neural-networks/>. Acesso em: 11 nov. 2019.

BROWNLEE, J. How to Fix the Vanishing Gradients Problem Using the ReLU. **Machine Learning Mastery**, 2019. Disponivel em: <machinelearningmastery>. Acesso em: 11 nov. 2019.

CHRISTOPHER, O. Understanding LSTM Networks. **Colah**, 2015. Disponivel em: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. Acesso em: 12 out. 2019.

CREATEMOMO. Super Machine Learning Revision Notes. **CreateMomo**, 2018. Disponivel em: <https://createmomo.github.io/2018/01/23/Super-Machine-Learning-Revision-Notes/>. Acesso em: 11 out. 2019.

DEVLIN, J. et al. BERT: Pre-training of Deep Bidirectional Transformers for. **arXiv preprint arXiv:1810.04805**, 2018.

HELLER, M. What is Keras? The deep neural network API explained. **Info World**, 2019. Disponivel em: <https://www.infoworld.com/article/3336192/what-is-keras-the-deep-neural-network-api-explained.html>. Acesso em: 9 nov. 2019.

HEUSEL, M. et al. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. **arXiv:1706.08500 [cs.LG]**, 2018. 6626-6637.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, v. 9, n. 8, p. 1735-1780, 1997.

HOREV, R. BERT Explained: State of the art language model for NLP. **Towards Data Science**, 2018. Disponivel em: <https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270>. Acesso em: 27 out. 2019.

JOHNSON, J. M.; KHOSHGOFTAAR, T. M. Survey on deep learning with class. **Journal of Big Data**, v. 6, n. 1, 2019.

JOSEF, H. Untersuchungen zu dynamischen neuronalen Netzen. **Diploma, Technische Universität München**, v. 91, n. 1, 15 jun. 1991.

KARPATHY, A. Convolutional Neural Networks for Visual Recognition. **CS231n**, 2018. Disponivel em: <http://cs231n.github.io/neural-networks-3/>. Acesso em: 7 nov. 2019.

KIM, S. A Beginner's Guide to Convolutional Neural Networks (CNNs). **Towards Data Science**, 2019. Disponivel em: <https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnns-14649dbddce8>. Acesso em: 29 nov. 2019.

KINGMA, D. P.; BA, J. L. Adam: A Method for Stochastic Optimization. **arXiv preprint arXiv:1412.6980**, 2014.

KIRSTEN, G. A evolução da descida de gradiente e seus otimizadores. **Medium**, 2019. Disponivel em: <https://medium.com/@gabrielkirsten/a-evolu%C3%A7%C3%A3o-da-descida-de-gradiente-e-seus-otimizadores-680c835c1b4f>. Acesso em: 29 nov. 2019.

KRISHAN. Words as Vectors, 2015. Disponivel em: <https://iksinc.online/tag/word2vec/>. Acesso em: 28 out. 2019.

MIKOLOV, T. et al. Efficient Estimation of Word Representations in Vector Space. **arXiv preprint arXiv:1301.3781**, 2013.

PESCE, A. Natural Language Processing in the kitchen. **Datadesk**, 2013. Disponivel em: <http://datadesk.latimes.com/posts/2013/12/natural-language-processing-in-the-kitchen/>. Acesso em: 14 nov. 2019.

PETERS, M. E. et al. Semi-supervised sequence tagging with bidirectional language models. **arXiv:1705.00108**, 2017.

PETERS, M. E. et al. Deep contextualized word representations. **arXiv:1802.05365**, 2018.

PETERS, M. E. et al. Tensorflow implementation of contextualized word representations from bi-directional language models. **Github**, 2018. Disponivel em: <https://github.com/allenai/bilm-tf>. Acesso em: 10 jan. 2019.

SAS. Natural Language Processing (NLP): What it is and why it matters. **SAS**, 2019. Disponivel em: <https://www.sas.com/en_us/insights/analytics/what-is-natural-language-processing-nlp.html>. Acesso em: 11 14 2019.

SHUNTARO, Y. A Review of Deep Contextualized Word Representations. **Slideshare**, 2018. Disponivel em: <https://pt.slideshare.net/shuntaroy/a-review-of-deep-contextualized-word-representations-peters-2018>. Acesso em: 19 out. 2019.

SRIVASTAVA, R. K.; GREFF, K.; SCHMIDHUBER, J. Training Very Deep Networks. **arXiv:1507.06228**, 2015. p. 2377-2385.

SUMIT, S. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. **towardsdatascience**, 2018. Disponivel em: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>. Acesso em: 11 out. 2019.

USMANI, Z.-U.-H. What is Kaggle, Why I Participate, What is the Impact? **Kaggle**, 2017. Disponivel em: <https://www.kaggle.com/getting-started/44916>. Acesso em: 1 nov. 2019.

VASWANI, A. et al. Attention Is All You Need. **Advances in neural information processing systems**, 2017. 5998-6008.

WANG, Z. et al. R-Transformer: Recurrent Neural Network. **arXiv preprint arXiv:1907.05572**, 7 dez. 2019.

WU, Y. et al. Google's Neural Machine Translation System: Bridging the Gap. **arXiv preprint arXiv:1609.08144**, 2016.

YAN, S. Understanding LSTM and its diagrams. **Medium**, 2016. Disponivel em: <https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714>. Acesso em: 12 out. 2019.

YEGULALP, S. What is TensorFlow? The machine learning library explained. **Info World**, 2019. Disponivel em: <https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html>. Acesso em: 9 nov. 2019.