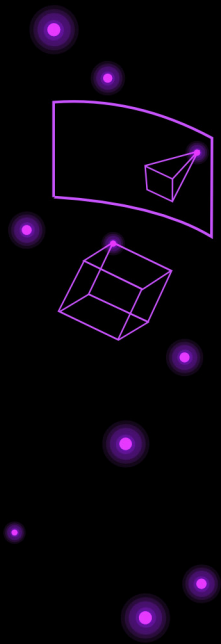
CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

Arquiteturas Profundas para Processamento de Linguagem Natural

Autoria:

Rafael Teixeira Sousa

Manoel Verissimo dos Santos Neto



Organizadores:

Deborah Silva Alves Fernandes

Taciana Novo Kudo

Renata Dutra Braga

Cristiane Bastos Rocha Ferreira

Arlindo Rodrigues Galvão Filho

Cegraf UFG

Arquiteturas Profundas para Processamento de Linguagem Natural

Autoria:
Rafael Teixeira Sousa
Manoel Verissimo dos Santos Neto

Organizadores:
Deborah Silva Alves Fernandes
Taciana Novo Kudo
Renata Dutra Braga
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

2024

Revisão Técnica
Deborah Silva Alves Fernandes

Revisão Editorial
Ana Laura de Sene Amâncio Zara Brisolla

Capa
Iuri Vaz Miranda

Editoração Eletrônica
Layane Grazielle Souza Dias
Luma Wanderley de Oliveira

https://doi.org/10.5216/SOU.arq.ebook.978-85-495-1079-2/2024

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Souza, Rafael Teixeira
Arquiteturas profundas para processamento de linguagem natural [livro eletrônico] / Rafael Teixeira Souza, Manoel Verissimo dos Santos Neto ; organização Deborah Silva Alves Fernandes...[et al.]. -- 1. ed. -- Goiânia, GO : Cegraf UFG, 2025.
PDF

"Edição de 2024".
Outros organizadores: Taciana Novo Kudo, Renata Dutra Braga, Cristiane Bastos Rocha Ferreira, Arlindo Rodrigues Galvão Filho.
Bibliografia.
ISBN 978-85-495-1079-2

1. Linguagem 2. Processamento de dados linguísticos I. Santos Neto, Manoel Verissimo dos. II. Fernandes, Deborah Silva Alves. III. Kudo, Taciana Novo. IV. Braga, Renata Dutra. V. Ferreira, Cristiane Bastos Rocha. VI. Galvão Filho, Arlindo Rodrigues.

25-256794 CDD-400

Índices para catálogo sistemático:

1. Linguagem 400

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Arquiteturas Profundas para Processamento de Linguagem Natural

Instituições responsáveis

Universidade Federal de Goiás (UFG)

Centro de Competência Embrapii em Tecnologias Imersivas, denominado AKCIT (Advanced Knowledge Center for Immersive Technologies)

Centro de Excelência em Inteligência Artificial (CEIA)

Instituições financiadoras

Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii)

Governo do Estado de Goiás

Empresas parceiras do AKCIT

Apoio

Universidade Federal de Goiás (UFG)

Pró-Reitoria de Pesquisa e Inovação (PRPI-UFG)

Instituto de Informática (INF-UFG)

Lista de Abreviaturas e Siglas

BERT	Bidirectional Encoder Representations from Transformers - Representações de Codificadores Bidirecionais de Transformadores
BiLSTM	Bidirectional Long Short-Term Memory Network - Rede Bidirecional de Memória de Longo e Curto Prazo
CNNs	Redes Neurais Convolucionais
ELMo	Embeddings from Language Models - Incorporações de Modelos de Linguagem
FFNN	Feed-Forward Neural Networks - Redes Neurais Diretas
FLAN	Fine-tuned Language Net - Rede de Idiomas Aprimorada
GLoVe	Global Vectors for Word Representation - Vetores Globais para Representação de Palavras
GLUE	General Language Understanding Evaluation - Avaliação Geral da Compreensão Linguística
GPT	Generative Pre-trained Transformer - Transformador Generativo Pré-treinado
GPU	Graphics Processing Units - Unidades de Processamento Gráfico
GRU	Gated Recurrent Unit - Unidade Recorrente Fechada
IA	Inteligência Artificial
KNN	K-Nearest Neighbors - K-Vizinhos Mais Próximos
LLM	Large Language Model - Grandes Modelos de Linguagem
LSTM	Long Short-Term Memory - Memória de Longo Prazo
MLM	Masked Language Modeling - Modelagem de Linguagem Mascarada
MLP	Multi Layer Perceptron - Perceptron Multicamadas

NER	Named Entity Recognition - Reconhecimento de Entidades Nomeadas
NSP	Next Sentence Prediction - Previsão da Próxima Frase
PLN	Processamento de Linguagem Natural
Q&A	Perguntas e Respostas
RNN	Rede Neural Recorrente
SQuAD	Stanford Question Answering Dataset - Conjunto de Dados de Resposta a Perguntas de Stanford
STLR	Sloped Triangular Learning Rate - Taxa de Aprendizado Triangular Inclinada
TPU	Tensor Processing Units - Unidades de Processamento de Tensor
UFG	Universidade Federal de Goiás
UFLM-FiT	Universal Language Model Fine-tuning - Ajuste Fino do Modelo de Linguagem Universal

Listas de Figuras, Quadros e Tabelas

Sumário

Apresentação

Prezado(a) Participante,

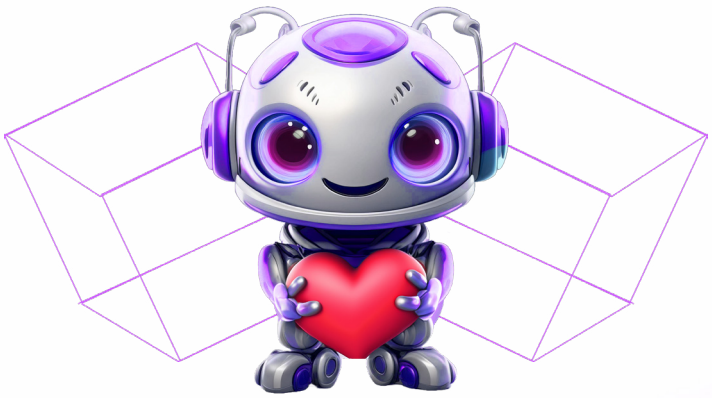
Seja bem-vindo(a) ao Microcurso **Arquiteturas Profundas para Processamento de Linguagem Natural (PLN)**!

Este Microcurso faz parte da Coleção Formação e Capacitação do Centro de Competências Imersivas, uma parceria entre a Embrapii e a Universidade Federal de Goiás (UFG).

PLN e Redes Neurais se cruzaram por vários anos na história da Inteligência Artificial (IA), na tentativa de criar representações que consigam compreender a linguagem e assim solucionar tarefas que lidam com textos. Para entendermos melhor tal evolução, é necessário compreender como os métodos clássicos de PLN evoluíram à medida que novas arquiteturas neurais foram sendo propostas.

Partindo do *Word2Vec*, que atua como uma arquitetura neural que representa palavras, combinado com Redes Neurais Recorrentes (RNNs), que são capazes de compreender uma sequência de palavras, até os Grandes Modelos de Linguagem (Modelos de Linguagem de Grande Porte ou *Large Language Models* - LLM), que tanto representam palavras como também realizam a compreensão de textos em uma mesma arquitetura, temos um histórico de diferentes evoluções que serão explorados neste Microcurso.

Dessa forma, neste Microcurso, serão abordadas, em detalhes, as diversas arquiteturas propostas, seus desdobramentos e aplicações, nos trazendo do PLN clássico, pautado no *Machine Learning* clássico, ao PLN contemporâneo baseado em *Deep Learning* e o Aprendizado de Representações.



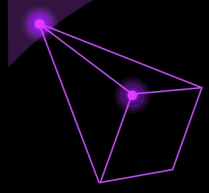
Desejamos um excelente estudo!!!

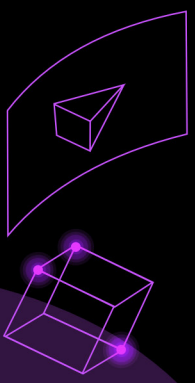
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Arquiteturas Profundas para Processamento de Linguagem Natural

Unidade I

Redes Neurais Recorrentes





Unidade I: Redes Neurais Recorrentes

Para entendermos a evolução do PLN baseado em *deep learning,* primeiro, precisamos compreender a necessidade que nos levou à tal evolução. Inicialmente, é preciso refletir sobre a abordagem clássica. Imaginemos um problema de classificação de sentimentos de avaliações de restaurantes utilizando *bag-of-words* (considerando *stopwords* já removidas) e um classificador, como, por exemplo, um *K-Nearest Neighbors* (KNN). Se considerarmos dois possíveis sentimentos (positivo e negativo) e um dicionário com quatro diferentes palavras (comida, boa, ruim, não) podemos ter os seguintes exemplos de sentenças, descritos na Tabela 1, a seguir.

Tabela 1 - Exemplo de *bag-of-words* em duas sentenças diferentes

Sentença	Comida	Boa	Ruim	Não	Sentimento
A comida estava boa, não estava ruim	1	1	1	1	Positivo
A comida estava ruim, não estava boa	1	1	1	1	Negativo

Fonte: autoria própria.

Ao lermos as duas sentenças na Tabela 1, somos capazes de compreender que possuem sentido contrário, mas ao observar como o *bag-of-words* as representa, podemos ver que teriam a mesma representação. A representação com *bag-of-words* aponta somente a ocorrência da palavra, assim, ambas as sentenças acabariam sendo consideradas equivalentes por um KNN. A partir disso, onde está o problema: no *bag-of-words* ou no KNN? De certa forma, em ambos. Tanto o *bag-of-words* não possui capacidade de expressar as palavras conforme seu contexto e ordem de ocorrência, como também o KNN não tem capacidade de superar esses problemas. Esse arranjo de técnicas considera que as palavras apenas ocorrem, como se todas estivessem em um mesmo saco, ocorrendo em um mesmo momento, sem compreensão de ordem.

Se alterarmos o *bag-of-words* para o *Word2Vec*, uma abordagem mais recente baseada em redes neurais, teríamos uma série de vetores densos. Porém, da mesma

forma, teríamos uma sequência de vetores semelhantes, pois as duas sentenças possuem as mesmas palavras. Assim, a única caraterística capaz de diferenciar as duas sentenças é compreender a ordem em que elas ocorrem, abordando, assim, a sentença como uma sequência de símbolos ordenados.

Um *Multi-Layer Perceptron*, como o utilizado pelo *Word2Vec*, não é capaz de compreender ordem, pois trata as suas entradas como partes de uma mesma amostra. Para tratar problemas com dados sequenciais, foram propostas as RNNs. Tal abordagem foi responsável pelas primeiras soluções relevantes de PLN baseadas em redes neurais por combinar a representação vetorial, como a proposta pelo *Word2Vec*, com uma arquitetura que é capaz de processar e compreender dados sequenciais.

Na Figura 1, a seguir, podemos ver uma linha do tempo com as diferentes arquiteturas neurais que serão exploradas no decorrer deste Microcurso e as diferentes eras. No início, temos o período do PLN clássico, onde surgiram as primeiras alternativas neurais, mas que ainda não eram completamente exploradas. Na sequência, a evolução do *Deep Learning* em PLN, com as primeiras arquiteturas profundas. Por fim, temos a era dos Grande Modelos de Linguagem, que se sustentam na evolução da era anterior para alcançar resultados significativamente superiores.

Figura 1 - Linha do tempo da evolução das arquiteturas neurais para Processamento de Linguagem Natural

Fonte: autoria própria.

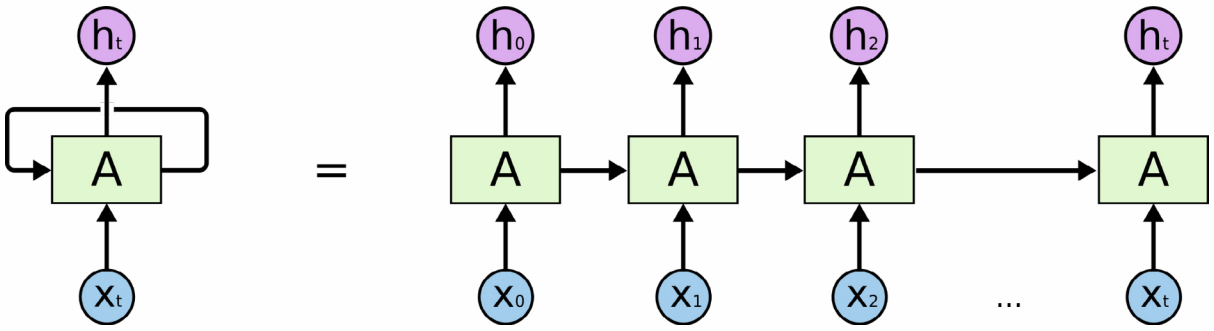
1.1 Redes Neurais Recorrentes (RNNs)

As primeiras alternativas de RNNs foram propostas por Jordan e Elman, em 1986 e 1990, respectivamente. Apesar de serem propostas diferentes, ambas são bem parecidas, por isso, é comum creditar a criação das RNNs a ambos autores. As duas propostas seguem um mesmo princípio: um neurônio artificial que seja capaz de se realimentar com a própria saída à medida que recebe novas entradas. Na equação, a seguir, e na Figura 1, é detalhado o funcionamento de uma RNN de Elman, onde as setas indicam o fluxo de informações.

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

$$y_t = \sigma_y(W_y h_t + b_y)$$

Figura 2 - Estrutura fundamental de uma Rede Neural Recorrente (RNN)



Fonte: adaptada de Wikimedia Commons (2024).

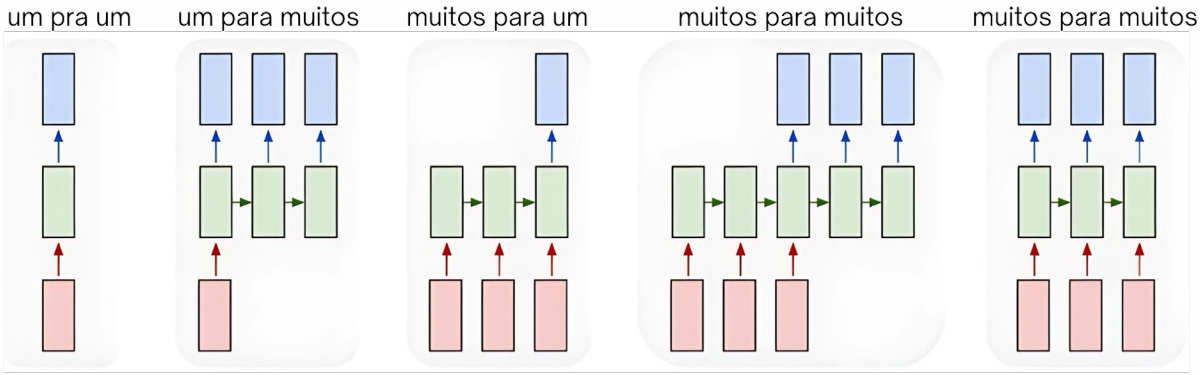
Um dos principais elementos da RNN é o seu estado interno que é definido por uma matriz de pesos multiplicada pela entrada , um bias , o estado interno do tempo anterior e sua função de ativação representada por σ, que comumente é a função tangente hiperbólica. Após definido o estado interno temos outra matriz de pesos e bias que são usados para definir a saída da rede . Sendo assim, a arquitetura é semelhante a um *perceptron*, que define o novo estado interno, e outro *perceptron*, que usa desse estado para definir a saída.

Os pesos e *bias* são utilizados para processar cada uma das *t* entradas. O *t* aqui é usado em uma alusão a tempo, que, no caso de textos, pode ser encarado como a posição das palavras em relação a uma sentença ou texto. Como a mesma matriz de pesos é utilizada no processamento de todas as entradas, a equação recorre a cada entrada, sendo assim, chamada de recorrente. O estado interno é um vetor usado como um espécie de memória da rede, onde, a cada entrada, é alterado e usado para influenciar a rede no instante seguinte. Dessa forma, o *h* se torna, ao fim da execu-

ção, um *embedding* que incorpora toda a sequência de entrada, representando a interpretação e projeção da sentença em um novo espaço latente, que tenta resolver o problema a partir da sua compreensão.

A partir de uma RNN, é possível solucionar o exemplo citado anteriormente, pois, ao receber as duas sentenças "A comida estava boa, não estava ruim" e "A comida estava ruim, não estava boa", por terem uma ordem diferente vão gerar dois estados *h* diferentes. Assim, a RNN consegue, de certa forma, resolver problemas de PLN quando combinada com representações vetoriais de palavras como o *Word2Vec*. Por serem capazes de receber infinitas entradas e gerar infinitas saídas, as RNNs possuem diferentes arranjos possíveis de utilização, descritos a seguir e exemplificados na Figura 3:

Figura 3 - Diferentes arranjos de utilização de uma Rede Neural Recorrente (RNN), onde cada bloco vermelho simboliza uma entrada, azul uma saída e verde uma recorrência da RNN



Fonte: adaptada de Li; Li; Gao (2023).

- **Um para um:** uma RNN aplicada com apenas uma entrada e uma saída é equivalente a inutilizar o estado interno *h*, tornando-se, assim, equivalente a uma simples camada de um *Multi Layer Perceptron* (MLP).
- **Um para muitos:** se, a partir de uma entrada, gerarmos várias saídas sem novas entradas, apenas reutilizando e alterando o seu estado interno, teremos, assim, uma geração condicional de dados. Um exemplo desse caso é gerar uma descrição de uma imagem, pois dada uma única imagem é necessário gerar várias palavras que a descreve.
- **Muitos para um:** nesse caso, temos diversos problemas quando é necessário ler sentenças ou textos para se gerar uma única resposta ao final. Análise de sentimentos, classificação de textos e detecção de fraudes são exemplos de problemas que necessitam resumir textos em apenas uma única resposta.

» **Muitos para muitos:** quando se tem uma sequência de entrada e uma sequência de saída, podemos ter dois cenários: (i) muitos para muitos, onde a saída depende da total compreensão da entrada, como, por exemplo, em tradução automática, onde é necessário ler toda a entrada para definir como redigir a saída; (ii) muitos para muitos sincronizados, onde, a cada entrada, temos uma saída. Nesse caso, são problemas mais complexos como geração de legendas para vídeos ou classificação de vídeos quadro-a-quadro.

Este modelo original de RNN, apesar de ser teoricamente muito capaz, possui um ponto fraco significativo que é seu procedimento de treino. Para treinar uma RNN, é necessário calcular o gradiente a partir do erro calculado com relação a cada saída esperada e, então, aplicar o *backpropagation*, que é a retropropagação desse gradiente para corrigir os seus pesos. O problema é que ao executar o *backpropagation* para cada saída, é necessário corrigir as matrizes de pesos (*W* referente a entrada e *U* ao estado *h*) a cada tempo anteriormente executado para gerar tal saída. Como as matrizes recorrem no momento do *feedfoward* (processamento das entradas para gerar as saídas) no ajuste dos pesos também são ajustadas múltiplas vezes, gerando um produto de múltiplos gradientes. Caso o vetor gradiente tenha valores maiores que 1, isso resultará no *gradient exploding* (ou gradientes explosivos, em português), que é a ocorrência de um gradiente muito grande que pode fazer a otimização divergir. Caso os valores do gradiente sejam menores que 1, ocorrerá o *vanish gradient*, ou o sumiço do gradiente, de maneira semelhante ao que ocorre quando se encadeia várias camadas em um MLP. Por conta desse ponto fraco, as RNNs originais de Elman e Jordan possuem aplicações bem limitadas, por ser demasiadamente difícil treiná-las para lidar com entradas maiores que 50 palavras.

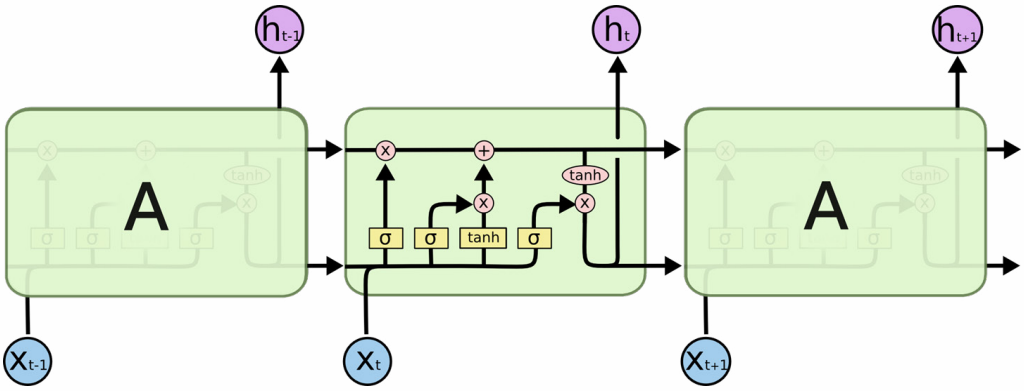
Ao tentar aprimorar as RNNs e permitir a compreensão de longas sequências de dados, várias arquiteturas foram propostas, dentre elas a mais relevante é a *Long Short-Term Memory* (LSTM) proposta por Hochreiter e Schmidhuber em 1997.

1.2 *Long Short-Term Memory*

A LSTM surge como uma alternativa recorrente para evitar o problema do *vanish/exploding gradient* e aumentar a expressividade das RNNs. Tal rede é baseada em *gates* (portões ou portas) que atuam como seletores de passagem de informação. Diferente do modelo original da RNN, onde toda informação que adentra a rede, obrigatoriamente altera o estado interno, e tal alteração é sempre propagada para os instantes de tempo seguintes, a LSTM usa de seus *gates* para regular o fluxo de informação durante a recorrência. Os quatro *gates* que compõem a LSTM são: *forget*,

input, *update* e *output*. Além disso, ela também implementa o *cell state C*, que atua como um estado interno alternativo que não é exposto pela saída. Na Figura 4, está exposto o fluxo das informações com os *gates* em amarelo, com suas respectivas funções de ativação, e operações de soma e multiplicação em rosa.

Figura 4 - Estrutura de uma *Long Short-Term Memory* (LSTM)

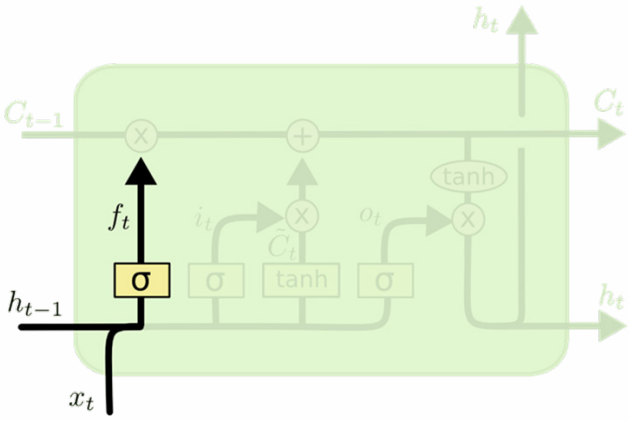


Fonte: adaptada de Oinkina; Hakyll (2015).

***Forget Gate*:** ao regular o que será esquecido (ou descartado) pelo estado interno, a LSTM tem a capacidade de alterar o estado interno em um primeiro momento, antes de definir sua saída. Esse portão gera saídas entre 0 e 1 que são multiplicadas ponto a ponto ao *cell estate*, assim, quando sua saída é 1 irá manter e quando 0 irá descartar parte do estado *C*. Na Figura 5, podemos visualizar o fluxo das entradas e saídas do *forget gate*.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figura 5 - *Forget Gate*



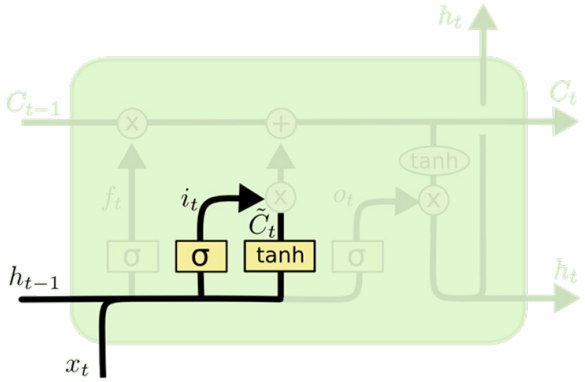
Fonte: adaptada de Oinkina; Hakyll (2015).

Input e Update Gate: o portão de entrada e *update* atuam juntos para a partir do estado *h* passado definir um novo estado *C*. A atualização deste estado é feita pelo multiplicação entre a saída do *gate input* e do *update*, para, logo em seguida, realizar a soma ponto a ponto com o estado *C* (nesse momento já alterado pelo *forget gate*). Nas Figuras 6 e 7, é possível visualizar a combinação do *input* e *update gate* e seu efeito no *cell state*.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

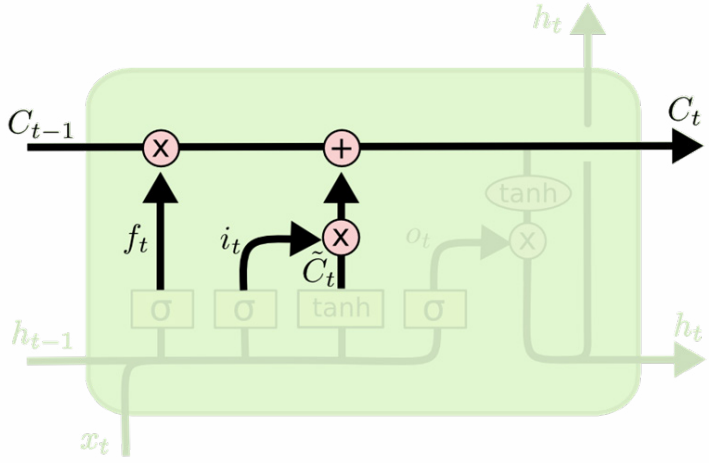
Figura 6 - *Input e Update Gate*



Fonte: adaptada de Oinkina; Hakyll (2015).

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

Figura 7 - Atualização do estado C a partir dos *gates Forget, Input* e *Update*

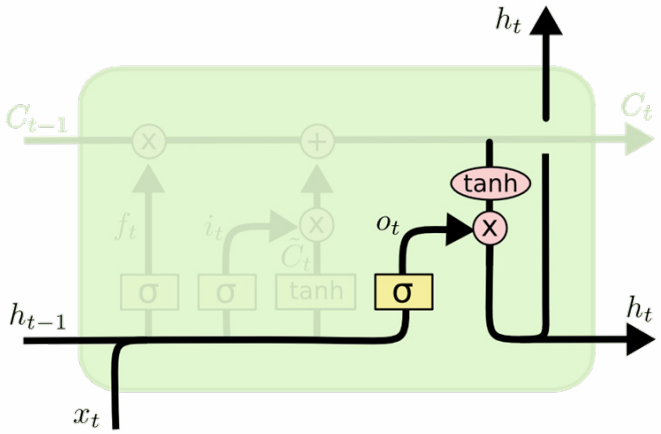


Fonte: adaptada de Oinkina; Hakyll (2015).

Output Gate: após atualizado o estado *C*, a LSTM realiza a definição da saída. O *output gate* calcula a saída da LSTM, operando com a entrada e estado *h* anterior e o produto é multiplicado com o estado *C*, atualizado pelos *gates* anteriores. O novo

estado *h* é usado, então, como saída da rede e repassado para o próximo instante de tempo. Na Figura 8 está exposto o processo de geração do novo estado *h*, a partir do *output gate*.

Figura 8 - *Output Gate*

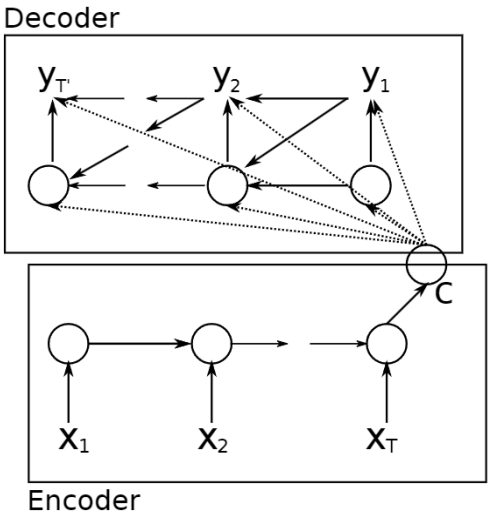


Fonte: adaptada de Oinkina; Hakyll (2015).

Com as inovações propostas, a LSTM consegue ser capaz de lidar melhor com o *vanish/exploding gradient* e, assim, lidar com entradas mais longas, tornando-se uma ferramenta importante no desenvolvimento do *Deep Learning* voltado a dados sequenciais com arranjo muitos para um. Além do PLN, a LSTM possibilitou avanços na área de processamento de fala no problema de identificação de fala e transcrição (Haşim, 2015); e também na área de visão computacional com trabalhos significativos que combinaram Redes Neurais Convolucionais (CNNs) e LSTM para descrição automática de imagens (Vinyals, 2015).

No PLN, as RNNs, como a LSTM, se destacaram principalmente a partir do ano de 2014, quando no trabalho intitulado "*Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*" foi proposta uma arquitetura com duas camadas recorrentes para tradução automática. O Seq2Seq, como ficou conhecido, é uma arquitetura que realiza o mapeamento de sequências em sequências com um formato de *auto-encoder*. A arquitetura usa de um princípio semelhante ao do *Word2Vec*, onde se tem um *encoder* e um *decoder*; o *encoder* tem a função de comprimir as entradas em um espaço latente e o *decoder* é responsável por decodificar de volta ao domínio original das entradas. Nesse caso, o Seq2Seq propõe um *encoder* em um idioma e um *decoder* em outro, com o espaço latente sendo responsável por comprimir e representar a informação de forma que seja independente de idioma. Na Figura 9, a arquitetura é apresentada, onde podemos ver como duas camadas separadas de RNNs interagem com as entradas (x) e saídas (y) e o vetor condicional *c* sendo repassado entre elas.

Figura 9 - Arquitetura do Seq2Seq



Fonte: adaptada de Cho (2014).

No mesmo trabalho, também é proposta uma alternativa à LSTM. Nomeada posteriormente de *Gated Recurrent Unit* (GRU), a unidade é uma RNN que combina o *input* e *forget gate* em um só *gate* e elimina o *cell state C*, possuindo apenas o estado interno *h*.

O Seq2Seq abriu um universo de possibilidades para problemas com arranjo do tipo muitos para muitos, especialmente problemas que envolvem compreensão e geração de texto, áudio, vídeo e séries temporais. Em PLN, problemas como sumarização de textos, Perguntas e Respostas (Q&A) e *chatbots* tiveram seus primeiros avanços significativos em direção às redes neurais com aplicações do Seq2Seq.

Apesar das vantagens e oportunidades, o Seq2Seq também possui algumas limitações que são importantes de serem compreendidas, sendo as duas principais a limitação da comunicação *encoder* e *decoder* e a dificuldade de treino. A comunicação *encoder-decoder,* por depender apenas do estado interno da camada do *encoder* para transferir informações para o *decoder*, o tamanho do estado interno é um limitante na expressividade da arquitetura toda. Aumentar o estado interno de RNNs significa aumentar, também, o número de pesos a serem treinados, pois todas possuem complexidade quadrática (h^2). Além disso, apesar da LSTM e GRU reduzirem os problemas de treinamento, esses ainda não são completamente solucionados, pois o *vanish/exploding* ainda ocorre em longas sequências. Ao adicionar um *decoder* à arquitetura do Seq2Seq, torna o treino ainda mais complexo por ser necessário calcular o erro no *decoder* e realizar o *backpropagation* até o *encoder*.

Na tentativa de solucionar tais problemas no Seq2Seq, foi proposto, em 2015, o trabalho intitulado "*Neural machine translation by jointly learning to align and trans-*

late", responsável por uma enorme mudança no PLN. Nesse trabalho, foi proposto o que inicialmente foi chamado de alinhamento, um mecanismo que é capaz de alinhar os estados internos do *encoder* com os instantes de tempo do *decoder*. Dessa forma, em vez de comprimir toda a informação do *encoder* no seu último estado interno, o *decoder* passa a ser capaz de observar o estado interno em todos os seus instantes de tempo e construir, assim, um vetor de contexto. Tal método foi posteriormente chamado de mecanismo de atenção e deu origem a uma nova família de arquiteturas independentes de recorrência que serão detalhadas na Unidade II deste Microcurso.

Antes de passarmos para a próxima Unidade, vamos na seção seguinte experimentar o processo de treinar algumas RNNs na tarefa de classificação de textos para identificação de discurso de ódio.

1.3 Classificação de sentenças com Redes Neurais Recorrentes (RNNs)

Notebook Colab

1. Objetivo:

- Neste notebook iremos ver como uma RNN e LSTM podem ser implementadas em um problema de classificação de texto. Iremos explorar suas qualidades e defeitos a partir de um conjunto de dados real.

2. Organização do ambiente

Primeiro iremos instalar um pacote usado para baixar conjuntos de dados do site HuggingFace. Após instalação e necessário reiniciar o ambiente

```
[ ] !pip install datasets keras==2.15.0 tensorflow==2.15.0
```

```
Collecting datasets
  Downloading datasets-2.21.0-py3-none-any.whl.metadata (21 kB)
Collecting keras==2.15.0
  Downloading keras-2.15.0-py3-none-any.whl.metadata (2.4 kB)
```

continua

```
Collecting tensorflow==2.15.0
  Downloading tensorflow-2.15.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.4 kB)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (1.6.3)
Requirement already satisfied: flatbuffers>=23.5.26 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (0.2.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (3.11.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (18.1.1)
Collecting ml-dtypes~=0.2.0 (from tensorflow==2.15.0)
  Downloading ml_dtypes-0.2.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (20 kB)
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (1.26.4)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (3.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (24.1)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (3.20.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (71.0.4)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (2.4.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (4.12.2)
Collecting wrapt<1.15,>=1.11.0 (from tensorflow==2.15.0)
  Downloading wrapt-1.14.1-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (6.7 kB)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (0.37.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.15.0) (1.64.1)
Collecting tensorboard<2.16,>=2.15 (from tensorflow==2.15.0)
  Downloading tensorboard-2.15.2-py3-none-any.whl.metadata (1.7 kB)
```

continua

```
Collecting tensorflow-estimator<2.16,>=2.15.0 (from tensorflow==2.15.0)
  Downloading tensorflow_estimator-2.15.0-py2.py3-none-any.whl.metadata
(1.3 kB)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/
dist-packages (from datasets) (3.15.4)
Collecting pyarrow>=15.0.0 (from datasets)
  Downloading pyarrow-17.0.0-cp310-cp310-manylinux_2_28_x86_64.whl.meta-
data (3.3 kB)
Collecting dill<0.3.9,>=0.3.0 (from datasets)
  Downloading dill-0.3.8-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/
dist-packages (from datasets) (2.1.4)
Requirement already satisfied: requests>=2.32.2 in /usr/local/lib/py-
thon3.10/dist-packages (from datasets) (2.32.3)
Requirement already satisfied: tqdm>=4.66.3 in /usr/local/lib/python3.10/
dist-packages (from datasets) (4.66.5)
Collecting xxhash (from datasets)
  Downloading xxhash-3.5.0-cp310-cp310-manylinux_2_17_x86_64.manyli-
nux2014_x86_64.whl.metadata (12 kB)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.16-py310-none-any.whl.metadata (7.2 kB)
Requirement already satisfied: fsspec<=2024.6.1,>=2023.1.0 in /usr/local/
lib/python3.10/dist-packages (from fsspec[http]<=2024.6.1,>=2023.1.0->d
atasets) (2024.6.1)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/
dist-packages (from datasets) (3.10.3)
Requirement already satisfied: huggingface-hub>=0.21.2 in /usr/local/lib/
python3.10/dist-packages (from datasets) (0.23.5)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/
dist-packages (from datasets) (6.0.2)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/
python3.10/dist-packages (from astunparse>=1.6.0->tensorflow==2.15.0)
(0.44.0)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/
python3.10/dist-packages (from aiohttp->datasets) (2.3.5)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (6.0.5)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (1.9.4)
```

continua

```
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/
python3.10/dist-packages (from aiohttp->datasets) (4.0.3)

Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/
lib/python3.10/dist-packages (from requests>=2.32.2->datasets) (3.3.2)

Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/
dist-packages (from requests>=2.32.2->datasets) (3.7)

Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py-
thon3.10/dist-packages (from requests>=2.32.2->datasets) (2.0.7)

Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py-
thon3.10/dist-packages (from requests>=2.32.2->datasets) (2024.7.4)

Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/
lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15->tensor-
flow==2.15.0) (2.27.0)

Requirement already satisfied: google-auth-oauthlib<2,>=0.5 in /usr/lo-
cal/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15->tensor-
flow==2.15.0) (1.2.1)

Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/py-
thon3.10/dist-packages (from tensorboard<2.16,>=2.15->tensorflow==2.15.0)
(3.6)

Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0
in /usr/local/lib/python3.10/dist-packages (from tensor-
board<2.16,>=2.15->tensorflow==2.15.0) (0.7.2)

Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/py-
thon3.10/dist-packages (from tensorboard<2.16,>=2.15->tensorflow==2.15.0)
(3.0.3)

Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/
python3.10/dist-packages (from pandas->datasets) (2.8.2)

Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/
dist-packages (from pandas->datasets) (2024.1)

Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/py-
thon3.10/dist-packages (from pandas->datasets) (2024.1)

Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/
lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensor-
board<2.16,>=2.15->tensorflow==2.15.0) (5.4.0)

Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/
lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensor-
board<2.16,>=2.15->tensorflow==2.15.0) (0.4.0)

Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/
python3.10/dist-packages (from google-auth<3,>=1.6.3->tensor-
board<2.16,>=2.15->tensorflow==2.15.0) (4.9)

Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/
lib/python3.10/dist-packages (from google-auth-oauthlib<2,>=0.5->tensor-
board<2.16,>=2.15->tensorflow==2.15.0) (1.3.1)

Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/lo-
cal/lib/python3.10/dist-packages (from werkzeug>=1.0.1->tensor-
board<2.16,>=2.15->tensorflow==2.15.0) (2.1.5)

Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /usr/local/
lib/python3.10/dist-packages (from pyasn1-modules>=0.2.1->google-au-
```

continua

```
th<3,>=1.6.3->tensorboard<2.16,>=2.15->tensorflow==2.15.0) (0.6.0)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/py-
thon3.10/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauth-
lib<2,>=0.5->tensorboard<2.16,>=2.15->tensorflow==2.15.0) (3.2.2)
Downloading keras-2.15.0-py3-none-any.whl (1.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 17.0 MB/s eta
0:00:00
Downloading tensorflow-2.15.0-cp310-cp310-manylinux_2_17_x86_64.manyli-
nux2014_x86_64.whl (475.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 475.2/475.2 MB 2.9 MB/s eta
0:00:00
Downloading datasets-2.21.0-py3-none-any.whl (527 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 527.3/527.3 kB 14.7 MB/s eta
0:00:00
Downloading dill-0.3.8-py3-none-any.whl (116 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 8.3 MB/s eta
0:00:00
Downloading ml_dtypes-0.2.0-cp310-cp310-manylinux_2_17_x86_64.manyli-
nux2014_x86_64.whl (1.0 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 17.5 MB/s eta
0:00:00
Downloading pyarrow-17.0.0-cp310-cp310-manylinux_2_28_x86_64.whl (39.9
MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 39.9/39.9 MB 14.6 MB/s eta
0:00:00
Downloading tensorboard-2.15.2-py3-none-any.whl (5.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 69.8 MB/s eta
0:00:00
Downloading tensorflow_estimator-2.15.0-py2.py3-none-any.whl (441 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 442.0/442.0 kB 31.0 MB/s eta
0:00:00
Downloading wrapt-1.14.1-cp310-cp310-manylinux_2_5_x86_64.manylinux1_
x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (77 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 77.9/77.9 kB 6.9 MB/s eta
0:00:00
Downloading multiprocess-0.70.16-py310-none-any.whl (134 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 7.6 MB/s eta
0:00:00
Downloading xxhash-3.5.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
x86_64.whl (194 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.1/194.1 kB 15.4 MB/s eta
0:00:00
Installing collected packages: xxhash, wrapt, tensorflow-estimator, pyar-
row, ml-dtypes, keras, dill, multiprocess, tensorboard, datasets, tensor-
flow
  Attempting uninstall: wrapt
```

continua

```
    Found existing installation: wrapt 1.16.0
    Uninstalling wrapt-1.16.0:
      Successfully uninstalled wrapt-1.16.0
  Attempting uninstall: pyarrow
    Found existing installation: pyarrow 14.0.2
    Uninstalling pyarrow-14.0.2:
      Successfully uninstalled pyarrow-14.0.2
  Attempting uninstall: ml-dtypes
    Found existing installation: ml-dtypes 0.4.0
    Uninstalling ml-dtypes-0.4.0:
      Successfully uninstalled ml-dtypes-0.4.0
  Attempting uninstall: keras
    Found existing installation: keras 3.4.1
    Uninstalling keras-3.4.1:
      Successfully uninstalled keras-3.4.1
  Attempting uninstall: tensorboard
    Found existing installation: tensorboard 2.17.0
    Uninstalling tensorboard-2.17.0:
      Successfully uninstalled tensorboard-2.17.0
  Attempting uninstall: tensorflow
    Found existing installation: tensorflow 2.17.0
    Uninstalling tensorflow-2.17.0:
      Successfully uninstalled tensorflow-2.17.0
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the fol-
lowing dependency conflicts.
cudf-cu12 24.4.1 requires pyarrow<15.0.0a0,>=14.0.1, but you have pyarrow
17.0.0 which is incompatible.
ibis-framework 8.0.0 requires pyarrow<16,>=2, but you have pyarrow 17.0.0
which is incompatible.
tensorstore 0.1.64 requires ml-dtypes>=0.3.1, but you have ml-dtypes
0.2.0 which is incompatible.
tf-keras 2.17.0 requires tensorflow<2.18,>=2.17, but you have tensorflow
2.15.0 which is incompatible.
Successfully installed datasets-2.21.0 dill-0.3.8 keras-2.15.0 ml-
dtypes-0.2.0 multiprocess-0.70.16 pyarrow-17.0.0 tensorboard-2.15.2 ten-
sorflow-2.15.0 tensorflow-estimator-2.15.0 wrapt-1.14.1 xxhash-3.5.0
```

Neste notebook usaremos o Keras, que é uma biblioteca integrada ao Tensorflow ou Pytorch e que nos permite implementar novas redes neurais de forma bastante simplificada.

Seu uso é principalmente recomendado para prototipação de novas arquiteturas.

3. Imports

Para iniciar vamos importar as bibliotecas que iremos usar no notebook

- NLTK: Vamos usar o tokenizador de texto da biblioteca chamado punkt
- Keras: Implementar a rede recorrente
- gensim: Carregar Word2Vec

Por fim vamos definir o seed do keras. Seed é uma forma de controlar as funções aletórias. Como ao criar uma nova rede neural ele é iniciada com pesos aleatórios o seed fixo vai garantir que todas as vezes que executarmos o notebook teremos o mesmo ponto de partida.

```
[ ] import collections

    import datasets
    import matplotlib.pyplot as plt
    import numpy as np
    import tqdm
    import string
    import pandas as pd

    from gensim.models import *

    import nltk
    from nltk.tokenize import word_tokenize
    nltk.download('punkt')

    import keras
    from keras.preprocessing.text import Tokenizer
    from keras.preprocessing.sequence import pad_sequences

    from gensim.models import KeyedVectors

    # Define a seed para garantir o mesmo resultado a cada execução
    keras.utils.set_random_seed(812)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

4. Word2Vec

Aqui faremos o *download* do Word2Vec treinado em português.

Os *embeddings* foram treinados pelo NILC (Núcleo Interinstitucional de Linguística Computacional do ICMC - USP) em um conjunto de diversos textos em português e também estão disponíveis no link.

No bloco de código abaixo é feito o *download*, extração do arquivo zip e carregamento dos embeddings usando a biblioteca gensim.

```
[ ] # Download do arquivo
    !wget -O "w2v.zip" "http://143.107.183.175:22980/download.php?file=embeddings/word2vec/skip_s100.zip"
    # Extração do zip
    !unzip "w2v.zip"
    # Aqui os pesos são carregado com a biblioteca gensim
    word2vec = KeyedVectors.load_word2vec_format("skip_s100.txt")

    # Definição do tamando dos embeddings. É necessário manter apenas 100 dimensões pois o arquivo baixado possui 100 dimensões
    EMB_DIM = 100
```

```
--2024-08-20 12:22:32--  http://143.107.183.175:22980/download.php?file=embeddings/word2vec/skip_s100.zip
Connecting to 143.107.183.175:22980... connected.
HTTP request sent, awaiting response... 200 OK
Length: 337179242 (322M) [application/octet-stream]
Saving to: ‘w2v.zip’

w2v.zip             100%[===================>] 321.56M  11.2MB/s    in 30s

2024-08-20 12:23:02 (10.7 MB/s) - ‘w2v.zip’ saved [337179242/337179242]

Archive:  w2v.zip
  inflating: skip_s100.txt
```

5. Dataset - HateBR - Offensive Language and Hate Speech Dataset in Brazilian Portuguese

Como conjunto de dados iremos usar um *dataset* brasileiro de classificação de discurso de ódio. O *dataset* está disponível abertamente e inclui 7000 exemplos anotados extraídos da rede social Twitter(atual X).

Aqui é feito o *download* dos 2 arquivos principais, conjunto de treino e de teste.

```
[ ] train_dataset, test_dataset = datasets.load_dataset("ruanchaves/hatebr",
    split=["train", "test"])
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.
py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your set-
tings tab (https://huggingface.co/settings/tokens), set it as secret in
your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to ac-
cess public models or datasets.
  warnings.warn(
Downloading data: 100% 264k/264k [00:00<00:00, 389kB/s]
Downloading data: 100% 68.2k/68.2k [00:00<00:00, 411kB/s]
Downloading data: 100% 81.0k/81.0k [00:00<00:00, 133kB/s]
Generating train split: 100% 4480/4480 [00:00<00:00, 64197.51 examples/s]
Generating validation split: 100% 1120/1120 [00:00<00:00, 36347.19 exam-
ples/s]
Generating test split: 100% 1400/1400 [00:00<00:00, 39383.66 examples/s]
```

O *dataset* possui a variável 'instagram_comments' contendo o texto, 'offensive_language' com o rótulo sobre ser ou não conteúdo ofensivo. As outras variáveis são rótulos mais específicos, como 'offensiveness_levels' que indica o nível de ofensa, e categorias de ofensas como 'homophobia', 'racism', e 'sexsim'

Iremos usar nessa demonstração apenas o rótulo principal que indica linguagem ofensiva ou não.

```
[ ] pd.DataFrame(train_dataset)
```

6. Pré-processamento

Para preparar os dados para o treino é feito um pré-processamento simples.

- Conversão dos textos para caixa baixa.
- Remoção da pontuação.
- Tokenização convertendo em uma lista de palavras.

```
[ ] train_data = []
    test_data = []

    # Trata conjunto de treino
    for data in train_dataset:
      # Separa entrada (text) do rótulo (y) com base na variável 'offensive_
    language'
      text = data['instagram_comments']
      y = data['offensive_language']

      # Transforma em caixa baixa
      s = str(text).lower()

      # Remove pontuação
      table = str.maketrans({key: None for key in string.punctuation.re-
    place("'", "")})
      s = s.translate(table)
      s = s.translate(str.maketrans({"'" : " "}))

      # Adiciona ao vetor de entradas tratadas
      train_data.append([s, y])

    # Trata conjunto de teste
    for data in test_dataset:
      # Separa entrada (text) do rótulo (y)
      text = data['instagram_comments']
      y = data['offensive_language']

      # Transforma em caixa baixa
      s = str(text).lower()
```

continua

```
    # Remove pontuação
    table = str.maketrans({key: None for key in string.punctuation.re-
place("'", "")})
    s = s.translate(table)
    s = s.translate(str.maketrans({"'" : " "}))

    # Adiciona ao vetor de entradas tratadas
    test_data.append([s, y])

# Transforma os vetores em DataFrames
train_data = pd.DataFrame(train_data, columns=['input', 'label'])
test_data = pd.DataFrame(test_data, columns=['input', 'label'])

# Aplica o tokenizador nas entradas (input)
train_data['input'] = train_data['input'].apply(lambda x: word_tokenize(x))
test_data['input'] = test_data['input'].apply(lambda x: word_tokenize(x))
```

Podemos visualizar o dataframe com os dados já pré-processados em amostras. A coluna *input* possui a lista de tokens, ainda em linguagem natural, e a coluna *label* armazena a resposta de cada amostra em um tipo lógico.

```
[ ] train_data.head()
```

	input	label
0	[este, lixo]	True
1	[comunista, safada]	True
2	[comunista, lixo]	True
3	[esquerdista]	True
4	[oportunista, essa, corruptaagora, todos, os, ...	True

7. Preparo dos dados

Agora precisamos converter as palavras em identificadores inteiros, pois as bibliotecas de redes neurais possuem implementadas camadas que fazem o mapeamento dos *embeddings* a partir de inteiros. Essa formatação torna o código mais efi-

ciente pois representamos a palavra da menor forma possível e apenas no momento do *feedfoward* da rede que ela será substituida por um vetor denso como o de um Word2Vec.

```
[ ] # Identifica todas as palavras únicas do dataset de treino
    all_training_words = [word for tokens in train_data["input"] for word in
    tokens]
    TRAINING_VOCAB = sorted(list(set(all_training_words)))
```

```
[ ] # Definir o tamanho máximo das sentenças
    MAX_LEN = 30


    # Aqui iremos ensinar o Tokenizer a mapear as palavras do dataset para
    que ele aprenda a quebrar as palavras em tokens
    tokenizer = Tokenizer(num_words=len(TRAINING_VOCAB), lower=True, char_
    level=False)
    tokenizer.fit_on_texts(train_data["input"].tolist())


    # Aqui convertemos o conjunto de treino e teste para inteiros
    training_sequences = tokenizer.texts_to_sequences(train_data["input"].
    tolist())
    test_sequences = tokenizer.texts_to_sequences(test_data["input"].
    tolist())


    # Por fim vamos fazer o padding que é o processo de adicionar 0s na frase
    para torna todas em uma lista do mesmo tamanho.
    # A referência usada é o MAX_LEN, frases maiores que MAX_LEN são cortadas
    e assim terão no máximo 30 tokens, enquanto frases menores também ficaram
    com 30 por conta do padding.
    train_hate_data = pad_sequences(training_sequences, maxlen=MAX_LEN)
    test_hate_data = pad_sequences(test_sequences, maxlen=MAX_LEN)


    # Agora precisamos ver qual inteiro está sendo usado para cada palavra e
    usar isto para ajustar o mapeamento do Word2vec
    train_word_index = tokenizer.word_index
    print('Número de palavras únicas %s.' % len(train_word_index))

    Número de palavras únicas 9221.
```

Após definir o mapeamento das palavras precisamos criar a matriz de pesos a partir do Word2Vec treinado. Para isso vamos pegar cada uma das palavras do voca-

bulário e usar o objeto do Word2Vec para identificar o vetor denso que a representa.

Caso a palavra não tenha vetor referente será adicionado um vetor aleatório. Isso é necessário porque o *dataset* possui alguns emojis, neologismos e termos usados no Twitter conforme exemplos apresentados pelo *print*.

```
[ ] # Cria matriz numpy para receber os pesos com base no total de palavras
    no dicionário e tamanho dos embeddings (dicionario, embs)
    train_embedding_weights = np.zeros((len(train_word_index)+1, EMB_DIM))

    # Para cada palavra no dicionário pega o vetor correspondente
    for word,index in train_word_index.items():
      if word in word2vec:
        # Se houver vetor pré-treinado o capturamos do objeto word2vec
        train_embedding_weights[index,:] = word2vec[word]
      else:
        # Aqui é para caso a palavra não existe nos vetores pré-treinados.
    isso ocorre com neologismos e emojis
        if(index < 500):
          # Dentre as 500 primeiras palavras avaliadas imprimimos as inexis-
    tentes para demonstração
          print("Palavra não encontrada: ", word)

        # Caso não tenhamos vetor pré-treinado será usado um vetor aleatório
        train_embedding_weights[index,:] = np.random.rand(EMB_DIM)
    print(train_embedding_weights.shape)
```

```
Palavra não encontrada:  joicehasselmannoficial
Palavra não encontrada:  globolixo
Palavra não encontrada:  “
Palavra não encontrada:  ”
Palavra não encontrada:  à
Palavra não encontrada:  👏👏👏
Palavra não encontrada:  👏👏👏👏
Palavra não encontrada:  BR
Palavra não encontrada:  😂😂😂
Palavra não encontrada:  2022
Palavra não encontrada:  👏👏
Palavra não encontrada:  😭
Palavra não encontrada:  👏👏👏👏👏
Palavra não encontrada:  bolsonarosp
(9222, 100)
```

Checando o *dataset* podemos ver os estágios de pré-processamento. Inicialmente a lista de tokens conhecidos, depois uma sequência de identificadores dos tokens e por fim uma sequência de inteiros com comprimento fixo.

```
[ ] train_data["input"][0]

['este', 'lixo']
```

```
[ ] training_sequences[0]

[98, 80]
```

```
[ ] train_hate_data[0]

array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 98, 80], dtype=int32)
```

8. RNN

Agora iremos implementar um modelo de RNN simples usando o Keras.

Neste modelo temos:

- Uma camada *Embedding* que é usada para receber a matriz dos vetores do Word2Vec;
- Uma camada *Simple*RNN que é uma RNN de Elman simples com estado interno *h* de tamanho 8;
- Uma camada Dense com um neurônio que irá pegar o estado interno final após ler a sentença e irá responder um valor entre 0 e 1 para classificar a sentença. O Modelo é compilado e será otimizado usando a entropia cruzada como função de erro e acurácia como métrica a ser acompanhada.

Documentação do Keras:

- Modelo Sequential
- Camadas disponíveis

```
ermitir que seja retreinada

 a camada anterior tem 8 saídas e irá construir os 8 pesos para tratar as entradas

ito eficiente) e a métrica a ser acompanhada (acurácia)
```

Next Clique aqui para avançar o código

Aqui executamos o treino do modelo com 10 épocas e *batch* de 32 sentenças. O treino possui o processo de validação feito com 20% das sentenças de treino que serão separadas para teste a cada época.

- Época é o nome que damos a uma volta inteira pelo conjunto de dados. Dessa forma, 10 épocas significam que cada exemplo, ao fim, será usado 10 vezes para ajustar a rede.
- *Batchs* são lotes de dados. Ao invés de usarmos o *dataset* todo, nós processamos 32 exemplos, calculamos o erro e já fazemos o ajuste dos pesos com o *backpropagation* e a resposta do *batch*.

Exercício 1: Tente alterar os parâmetros de *batch* e épocas para ver o comportamento do treinamento mudar.

```
[ ] model.fit(train_hate_data, train_data['label'], epochs=10, batch_size=32,
    validation_split=0.2)
```

continua

```
Epoch 1/10
112/112 [==============================] - 6s 30ms/step - loss: 0.6564 -
accuracy: 0.6138 - val_loss: 0.8921 - val_accuracy: 0.3471
Epoch 2/10
112/112 [==============================] - 3s 30ms/step - loss: 0.5705 -
accuracy: 0.7104 - val_loss: 0.9703 - val_accuracy: 0.3248
Epoch 3/10
112/112 [==============================] - 3s 24ms/step - loss: 0.5042 -
accuracy: 0.7598 - val_loss: 1.0057 - val_accuracy: 0.3739
Epoch 4/10
112/112 [==============================] - 2s 17ms/step - loss: 0.4604 -
accuracy: 0.7927 - val_loss: 0.8647 - val_accuracy: 0.5424
Epoch 5/10
112/112 [==============================] - 2s 16ms/step - loss: 0.4333 -
accuracy: 0.8069 - val_loss: 1.0125 - val_accuracy: 0.5011
Epoch 6/10
112/112 [==============================] - 2s 16ms/step - loss: 0.4188 -
accuracy: 0.8167 - val_loss: 0.9294 - val_accuracy: 0.5424
Epoch 7/10
112/112 [==============================] - 2s 15ms/step - loss: 0.4090 -
accuracy: 0.8287 - val_loss: 1.0386 - val_accuracy: 0.4844
Epoch 8/10
112/112 [==============================] - 2s 15ms/step - loss: 0.4016 -
accuracy: 0.8253 - val_loss: 0.8261 - val_accuracy: 0.6261
Epoch 9/10
112/112 [==============================] - 3s 24ms/step - loss: 0.3927 -
accuracy: 0.8242 - val_loss: 0.8943 - val_accuracy: 0.5926
Epoch 10/10
112/112 [==============================] - 3s 27ms/step - loss: 0.3885 -
accuracy: 0.8334 - val_loss: 1.0879 - val_accuracy: 0.4788

<keras.src.callbacks.History at 0x7fa5f4d4fc70>
```

Ao fim podemos ver o erro do modelo (loss) e seu desempenho.

Se pegarmos um exemplo do conjunto de teste podemos testar nossa RNN ao realizar uma predição.

```
[ ] test_data.iloc[30]
```

	30
input	[realmente, uma, verdadeira, hipócrita]
label	True

dtype: object

```
[ ] model.predict(test_hate_data[30].reshape(1, -1))
```

```
1/1 [==============================] - 0s 19ms/step
array([[0.8587818]], dtype=float32)
```

Nesse caso a saída esperada é um valor maior que 0.5, que representa uma alta probabilidade de ser um caso de discurso de ódio.

Exercício 2: Tente executar a predição de alguns outros exemplos do conjunto de teste.

9. Visualização do estado interno

Agora iremos fazer um experimento, vamos alterar a rede para que a camada RNN retorne todos os estados internos e não apenas o último.

```
[ ] model.layers[1].return_sequences = True
```

Aqui criamos uma função com o Keras que irá passar as entradas apenas até a RNN (get_h_state) e outra que irá pegar cada um destes estados internos pela camada Dense final do modelo (get_output).

Perceba que aqui não iremos realizar nenhum treinamento. O objetivo é apenas de visualizar a saída das camadas interiores do modelo já treinado anteriormente.

```
[ ] from keras import backend as K

    # with a Sequential model
    get_h_state = K.function([model.layers[0].input],
                                       [model.layers[1].output])
    layer_output = get_h_state(test_hate_data[30].reshape(1, -1))

    get_output = K.function([model.layers[2].input],
                                       [model.layers[-1].output])
    model_outputs = get_output(layer_output[0][0])[0].reshape(30)
```

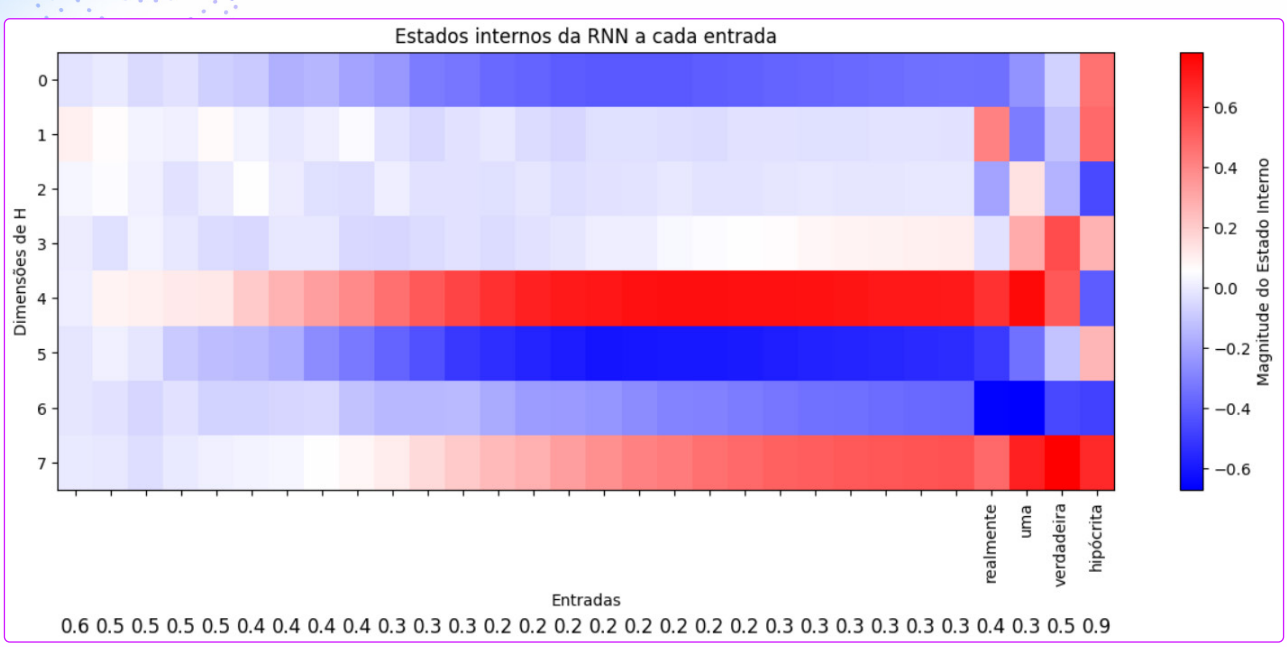
Agora iremos usar as duas funções para visualizar as mudanças do estado interno ao ler cada uma das entradas de um dos exemplos de teste.

Abaixo está também a saída da rede a cada palavra, assim podemos ver como a resposta final é construída ao ler a sentença.

```
[ ]
    def plot_internal_states(states, outputs, inputs):
        plt.figure(figsize=(15, 5))
        plt.imshow(states[0].T, aspect='auto', cmap='bwr')
        plt.colorbar(label='Magnitude do Estado Interno')
        plt.xlabel('Entradas')
        plt.ylabel('Dimensões de H')
        plt.title('Estados internos da RNN a cada entrada')

        # Marcar os inputs no eixo x
        plt.xticks(ticks=np.arange(len(inputs)), labels=inputs, rotation=90)

        for t in range(outputs.shape[0]):
            plt.text(t, 10, f'{outputs[t]:.1f}', ha='center', va='cen-
    ter', color='black', fontsize=12, bbox=dict(facecolor='white', edgecol-
    or='none', pad=0.5))

        plt.show()

    plot_internal_states(layer_output[0], model_outputs, tokenizer.sequenc-
    es_to_texts(test_hate_data[30].reshape(-1, 1)))
```

continua

Figura 10 - Estados internos da RNN a cada entrada I



Fonte: Autoria própia

Visualizar estados internos é uma tarefa complexa. Não é possível compreender exatamente como se comporta cada dimensão de um estado de uma rede já treinada, mas como estamos tratando de um problema binário é interessante visualizar a mudança nos estados a cada para inserida.

Podemos ver que ao ler a palavra “hipócrita” a rede passa a ter significativas mudanças no estado interno e assume uma saída positiva, pois considerando as palavras anteriores realmente ainda não era possível determinar como positivo ou negativo. Fica claro que a palavra se torna um elemento chave para a interpretação da rede.

No bloco abaixo temos uma caixa de entrada dinâmica onde podemos avaliar frases originais que não estão do dataset e visualizar o comportamento e saída da RNN ao processar a entrada.

No exemplo já inserido podemos ver que a palavra “safado” ao ocorrer antes já define um estado positivo e a “hipócrita” apenas a reforça.

Insira algum texto de exemplo para ver a dinâmica dos estados internos

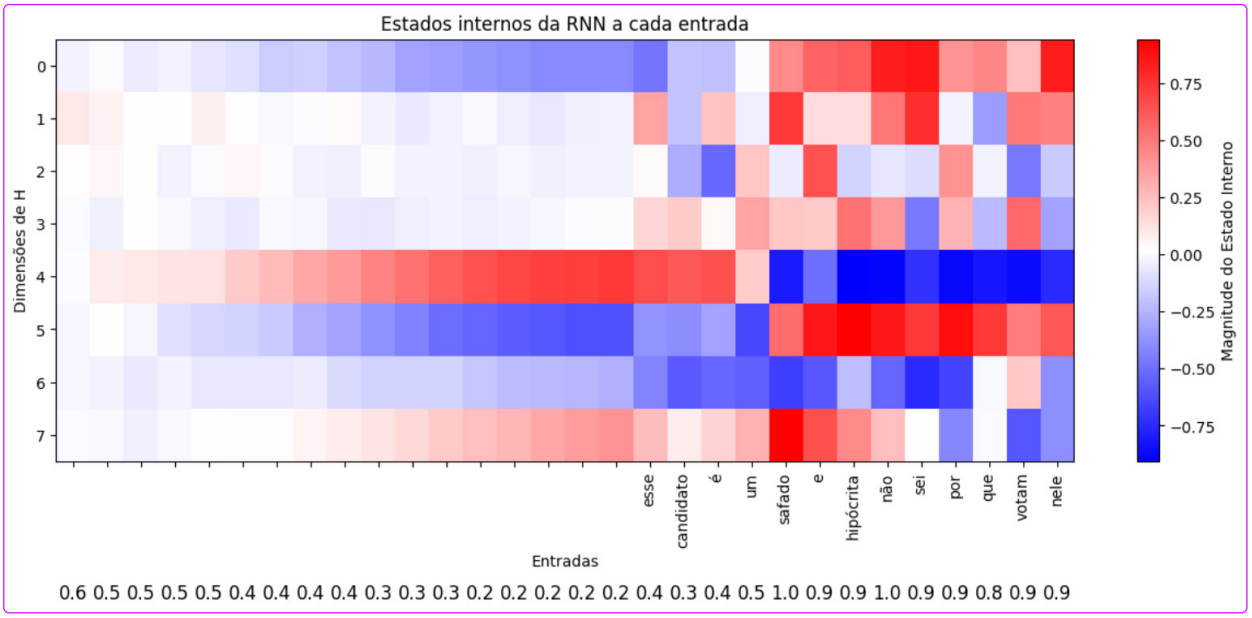
```
[ ] # @title Insira algum texto de exemplo para ver a dinâmica dos estados
    internos
    input = "Esse candidato é um safado e hipócrita, não sei por que votam
    nele" # @param {type:"string"}


    s = str(input).lower()
    table = str.maketrans({key: None for key in string.punctuation.re-
```

continua

```
    place("'", "")})
    s = s.translate(table)
    s = s.translate(str.maketrans({"'" : " "}))
    input_tokens = word_tokenize(s)
    input_tokenized = tokenizer.texts_to_sequences([input_tokens])
    input_padded = pad_sequences(input_tokenized, maxlen=MAX_LEN)

    layer_output = get_h_state(input_padded.reshape(1, -1))
    model_outputs = get_output(layer_output[0][0])[0].reshape(30)
    plot_internal_states(layer_output[0], model_outputs, tokenizer.sequenc-
    es_to_texts(input_padded.reshape(-1, 1)))
```

```
input:
" Esse candidato é um safado e hipócrita, não sei por que votam nele "
```

Figura 11 - Estados internos da RNN a cada entrada II



Fonte: Autoria própria

Agora iremos treinar uma LSTM no mesmo problema em questão. Usando Keras a única diferença é trocar a camada SimpleRNN por LSTM. Assim iremos criar uma camada LSTM com estado interno de tamanho 8.

```
[ ] from keras.layers import LSTM


    model_lstm = Sequential()
    model_lstm.add(Embedding(input_dim=len(train_word_index)+1,
```

continua

```
                    output_dim=EMB_DIM,
                    input_length=MAX_LEN,
                    weights=[train_embedding_weights],
                    trainable=False))
model_lstm.add(LSTM(8))
model_lstm.add(Dense(1, activation='sigmoid'))

model_lstm.compile(optimizer='adam', loss='binary_crossentropy', met-
rics=['accuracy'])

model_lstm.fit(train_hate_data, train_data['label'], epochs=10, batch_
size=32, validation_split=0.2)
```

```
Epoch 1/10
112/112 [==============================] - 5s 9ms/step - loss: 0.6411 -
accuracy: 0.6172 - val_loss: 0.9026 - val_accuracy: 0.1931
Epoch 2/10
112/112 [==============================] - 1s 6ms/step - loss: 0.5513 -
accuracy: 0.7363 - val_loss: 0.9680 - val_accuracy: 0.4018
Epoch 3/10
112/112 [==============================] - 1s 5ms/step - loss: 0.4507 -
accuracy: 0.8114 - val_loss: 1.0100 - val_accuracy: 0.4609
Epoch 4/10
112/112 [==============================] - 1s 5ms/step - loss: 0.4008 -
accuracy: 0.8354 - val_loss: 1.0376 - val_accuracy: 0.4743
Epoch 5/10
112/112 [==============================] - 1s 6ms/step - loss: 0.3738 -
accuracy: 0.8451 - val_loss: 1.0801 - val_accuracy: 0.5033
Epoch 6/10
112/112 [==============================] - 1s 5ms/step - loss: 0.3563 -
accuracy: 0.8502 - val_loss: 0.9644 - val_accuracy: 0.5580
Epoch 7/10
112/112 [==============================] - 1s 5ms/step - loss: 0.3402 -
accuracy: 0.8599 - val_loss: 1.0217 - val_accuracy: 0.5123
Epoch 8/10
112/112 [==============================] - 1s 5ms/step - loss: 0.3288 -
accuracy: 0.8644 - val_loss: 0.9873 - val_accuracy: 0.5614
Epoch 9/10
112/112 [==============================] - 1s 6ms/step - loss: 0.3200 -
accuracy: 0.8638 - val_loss: 0.7752 - val_accuracy: 0.6719
Epoch 10/10
112/112 [==============================] - 1s 5ms/step - loss: 0.3068 -
accuracy: 0.8783 - val_loss: 1.0290 - val_accuracy: 0.5781
<keras.src.callbacks.History at 0x7fa57ec293f0>
```

A LSTM irá provavelmente obter um melhor resultado. Porém, para realmente conseguir um resultado superior o ideal é aumentar o número de épocas e o tamanho do estado interno para fazer com que a rede aprenda melhor sobre os dados.

Exercício 3: Tente alterar esses parâmetros de treino para alcançar melhores resultados.

Exercício 4: Para explorar melhor o comportamento das RNNs que tal tentar refazer o todo o processo com algum dos diferentes rótulos que o dataset fornece.

Desafio: Implemente uma arquitetura com múltiplas camadas para alcançar um desempenho superior. Para isso leia a documentação da ferramenta Sequential e tente adicionar mais camadas recorrentes ou Dense.

https://keras.io/guides/sequential_model/

Referências

Keras: https://keras.io

RNN no Keras: https://keras.io/api/layers/recurrent_layers/simple_rnn/

LSTM no Keras: https://keras.io/api/layers/recurrent_layers/lstm/

Dataset de linguagem ofensiva: https://github.com/franciellevargas/MOL

SAIBA MAIS...

Acesse **Understanding LSTM Networks** (Oinkina; Hakyll, 2015).

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

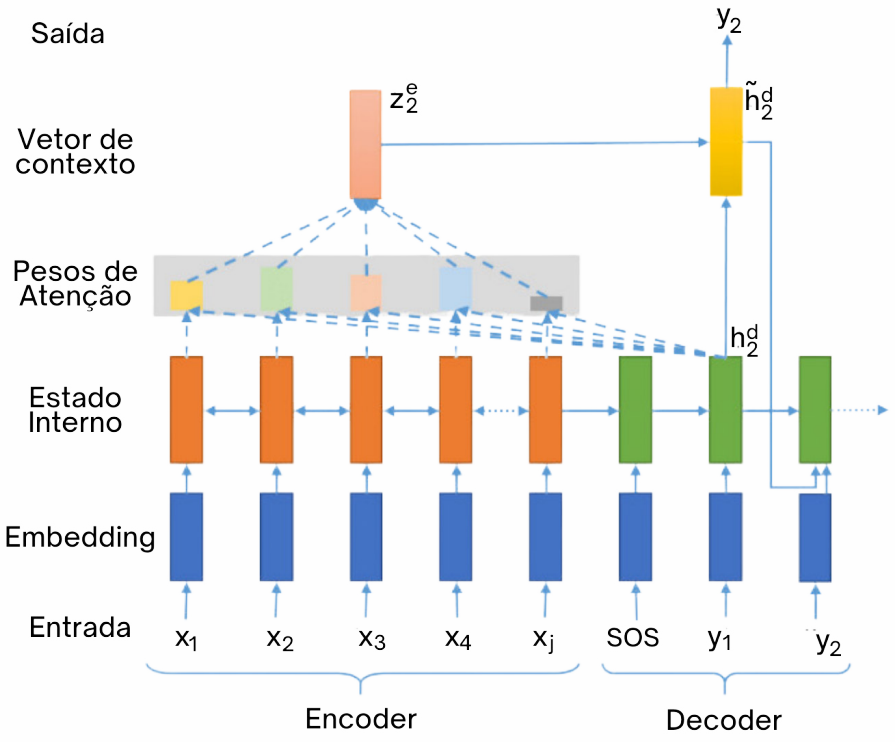
Arquiteturas Profundas para Processamento de Linguagem Natural

Unidade II

Redes Neurais baseadas em atenção: transformers

O Seq2Seq foi responsável por trazer problemas complexos do mundo do PLN para o *Deep Learning* apresentando uma arquitetura de alta capacidade (Figura 10). Ao evoluir para o Seq2Seq com atenção (ou alinhamento), a área foi apresentada a um conceito que iria revolucionar a elaboração de arquiteturas.

Figura 12 - Arquitetura do Seq2Seq com atenção

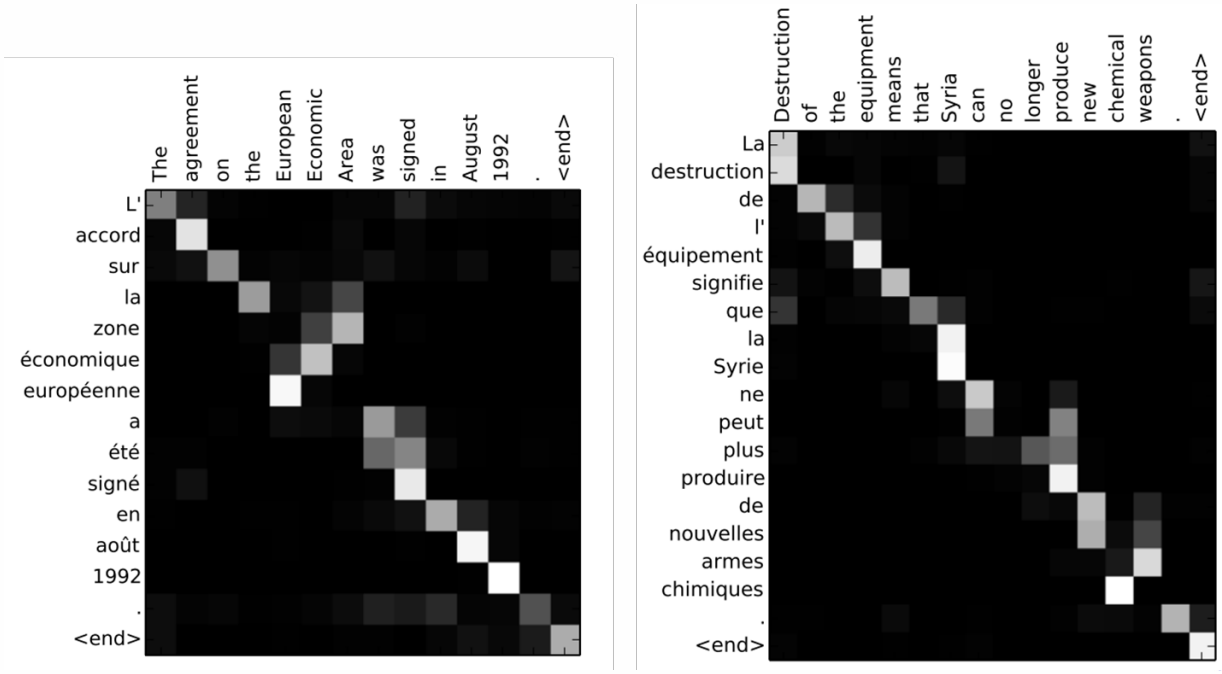


Fonte: adaptada de Shi (2020).

O mecanismo de atenção trouxe para o Seq2Seq a capacidade de ponderar os estados internos do *encoder* de maneira inteligente. Ao usar um pequeno MLP treinável para definir pesos que são usados para calcular a soma ponderada dos estados *h*, a arquitetura passa a ser capaz de filtrar os momentos mais relevantes do *encoder*. Tal mecanismo pode ser visualizado na Figura 10 (acima) que mostra o vetor de contexto sendo gerado a partir dos estados internos. As principais vantagens ao introduzir a atenção são:

- **Dependências de longo alcance:** o uso da atenção retira a necessidade da RNN armazenar informações que possam estar no início da entrada até no tempo final, pois o *decoder* pode capturar a influência das entradas iniciais sem a influência das seguintes.
- **Interpretação e a transparência:** ao observar os pesos atribuídos pela rede aos estado internos, é possível visualizar o funcionamento das arquiteturas. Mesmo que tal interpretação seja limitada e possa ser confusa, foi um passo importante na compreensão do funcionamento das arquiteturas em problemas complexos. Na Figura 11 (abaixo) temos um exemplo de visualização das atenções em um problema de tradução, onde temos uma sentença em francês no eixo Y e a tradução em inglês no eixo X. Os tons de cinza indicam a intensidade da atenção, quanto mais claro, maior a atenção do modelo naquele instante de tempo para gerar cada tradução.
- **Alinhamento de palavras:** em tarefas como tradução, o avanço foi significativo, pois a atenção permite à arquitetura alinhar palavras equivalentes em dois idiomas na hora de realizar a tradução.
- **Melhor treinamento:** ao permitir que o *backpropagation* seja realizado do *decoder* direto para todos os estados internos do *encoder*, a atenção suaviza o problema do *vanish gradient*.

Figura 13 - Visualização dos pesos do mecanismo de atenção em exemplos de tradução

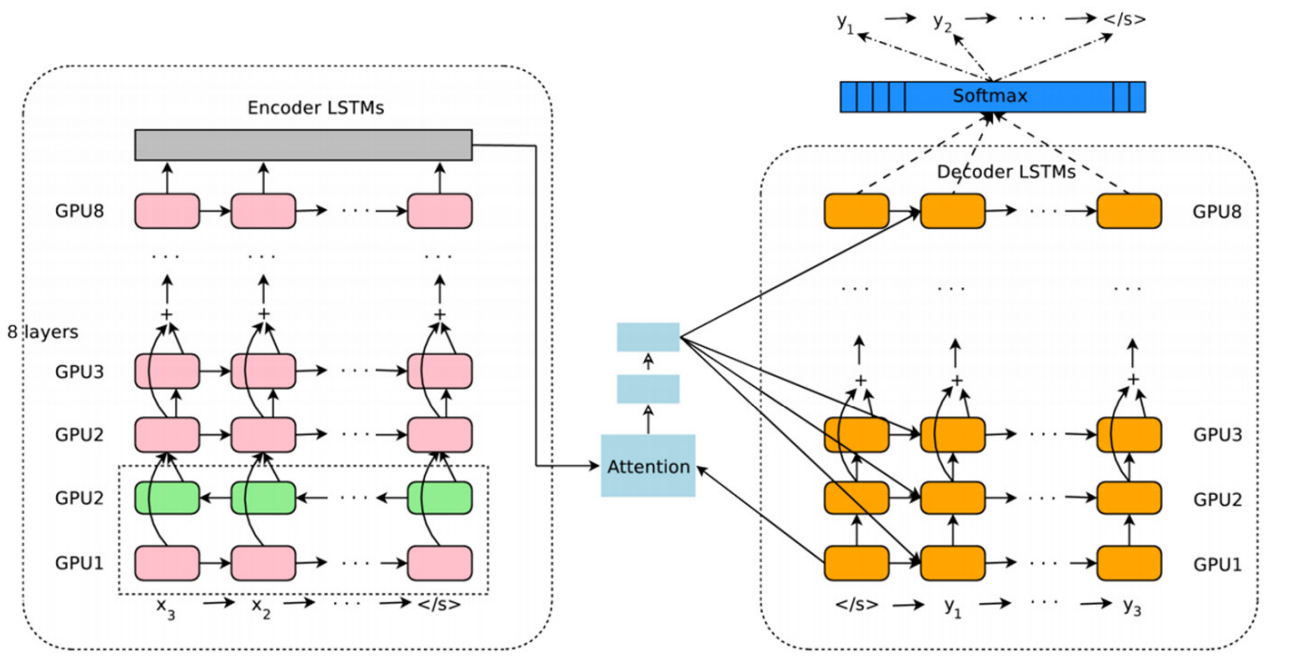


Fonte: adaptada de Bahdanau (2015).

Um marco do desenvolvimento de soluções com uso do Seq2Seq foi o *"Google's Neural Machine Translation with Joint Representation"*, que apresentou um novo

sistema de tradução elaborado pelo Google® com uso de múltiplas camadas de LSTMs e mecanismo de atenção (Figura 12). Tal trabalho evoluiu nos anos seguintes para uma solução multilingual e aprimorou o serviço de tradução do Google® de maneira significativa.

Figura 14 - Arquitetura da *Google's Neural Machine Translation* (GNMT) para tradução



Fonte: adaptada de Wu (2016).

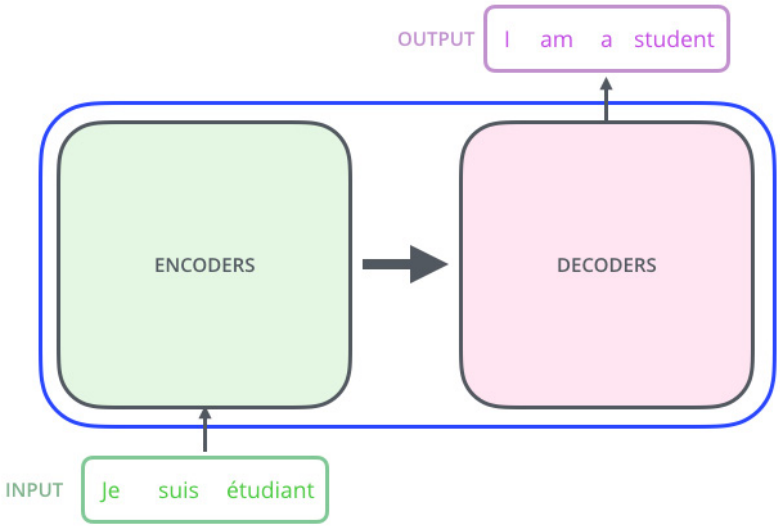
Após o impacto do conceito da atenção, foi proposto pela própria equipe de pesquisa do Google®, em 2017, o artigo *"Attention is all you need"*, que introduziu a arquitetura *Transformer,* uma nova abordagem baseada apenas em operações de atenção. Diferente da atenção utilizada no Seq2Seq, que é uma atenção cruzada entre *encoder* e *decoder*, o *Transformer* aplica a chamada autoatenção (*self-attention*), que realiza a operação de cruzamento entre as entradas com elas mesmas. Nas seções seguintes, iremos destrinchar o funcionamento de cada parte da arquitetura.

2.1 *Transformer*

Após a introdução do *Word2Vec*, podemos dizer que a arquitetura *Transformer* foi a maior revolução na construção de arquiteturas neurais para PLN. A arquitetura funciona a partir de uma combinação de autoatenção combinada com camadas *feedforward* (semelhantes a camada de um MLP), sem utilizar de recorrências. Para

entendermos a arquitetura, primeiro, precisamos considerar que ela mapeia sequências em sequência, em um formato *encoder-decoder*, de forma semelhante a um Seq2Seq (Figura 13).

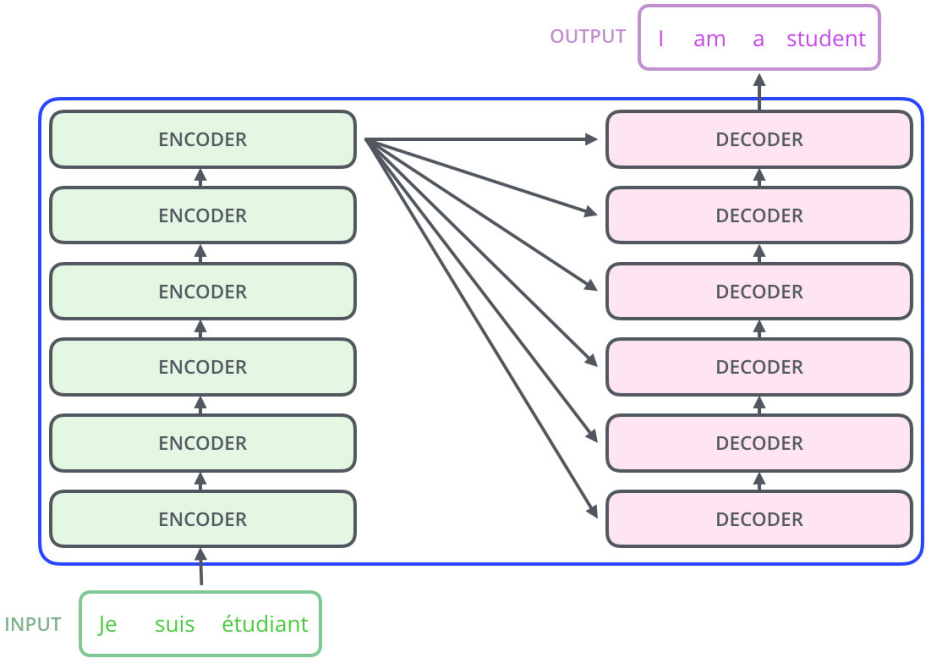
Figura 15 - *Transformer* como *encoder-decoder*



Fonte: adaptada de Alammar (2018).

Se olharmos mais a fundo, veremos que tanto o *encoder* como *decoder* são compostos por múltiplas camadas. Essas estruturas são chamadas de blocos e são diferentes nas duas partes do modelo (Figura 14).

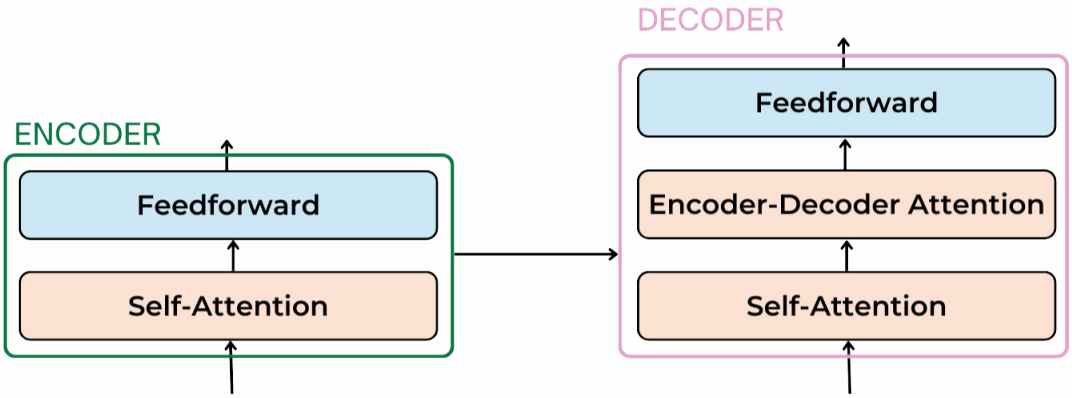
Figura 16 - Blocos de um *encoder* e *decoder* do *Transformer*



Fonte: adaptada de Alammar (2018).

Cada um desses blocos é composto por uma camada de autoatenção e uma *feedforward*, exceto no *decoder* onde temos uma autoatenção, uma autoatenção que cruza os dados com o *encoder* e, por fim, a *feedforward*. Como pode ser visto na Figura 15, nenhum dos blocos possui RNNs, sendo assim não-recorrente.

Figura 17 - Bloco do *encoder* e *decoder*



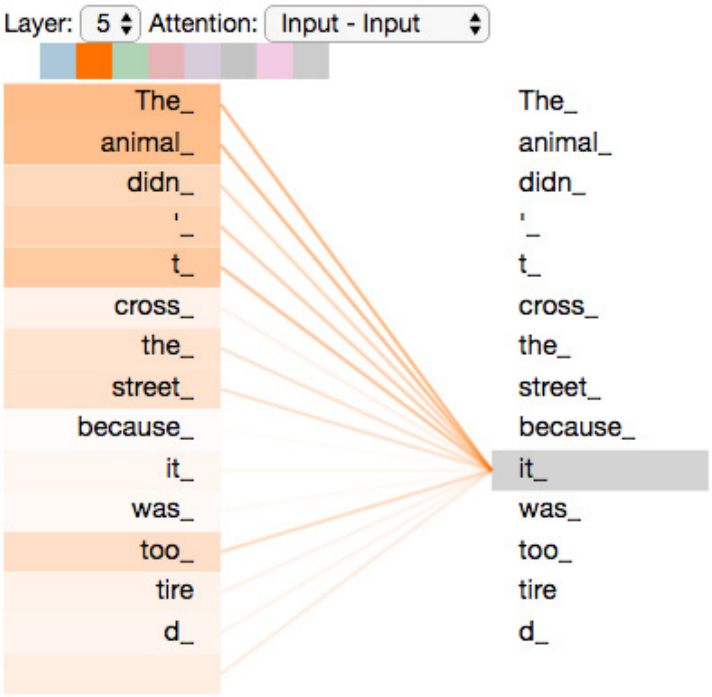
Fonte: adaptada de Alammar (2018).

A camada de *feedforward* é semelhante a uma camada de um MLP e serve para adicionar uma camada de operação não-linear, dando mais capacidade ao modelo. Sobre a autoatenção, iremos analisar com mais detalhes a partir de um exemplo.

2.1.1 Autoatenção

A operação de autoatenção segue o mesmo conceito do alinhamento proposto ao Seq2Seq, porém é realizado o alinhamento das entradas com as próprias entradas. A ideia fundamental por trás da autoatenção é de que cada palavra em uma sentença possui significado construído a partir das palavras que a acompanham, assim, a compreensão de todo o texto vem da compreensão de tais relações. Como no exemplo exposto na Figura 16, considerando os pesos de uma autoatenção, a palavra "*it*" possui forte relação com "*The animal*" por ser uma referência ao sujeito da frase. Dessa forma, ao encadear diversas camadas de autoatenção, é possível criar algo semelhante a uma árvore de análise sintática, onde a representação criada na rede neural é capaz de abstrair tais relações em longos textos. Para essa e outras visualizações interativas, acesse este *link*.

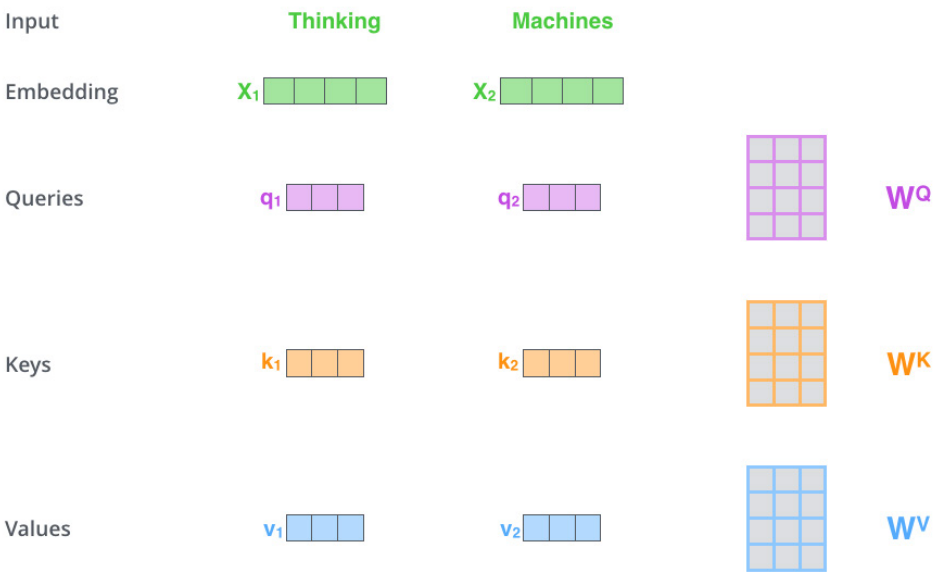
Figura 18 - Exemplo de autoatenção



Fonte: adaptada de Alammar (2018).

Para explicarmos a fundo a autoatenção utilizada nos *Transformers*, vamos usar o exemplo a seguir (Figura 17) com a sentença em inglês "*Thinking Machines*".

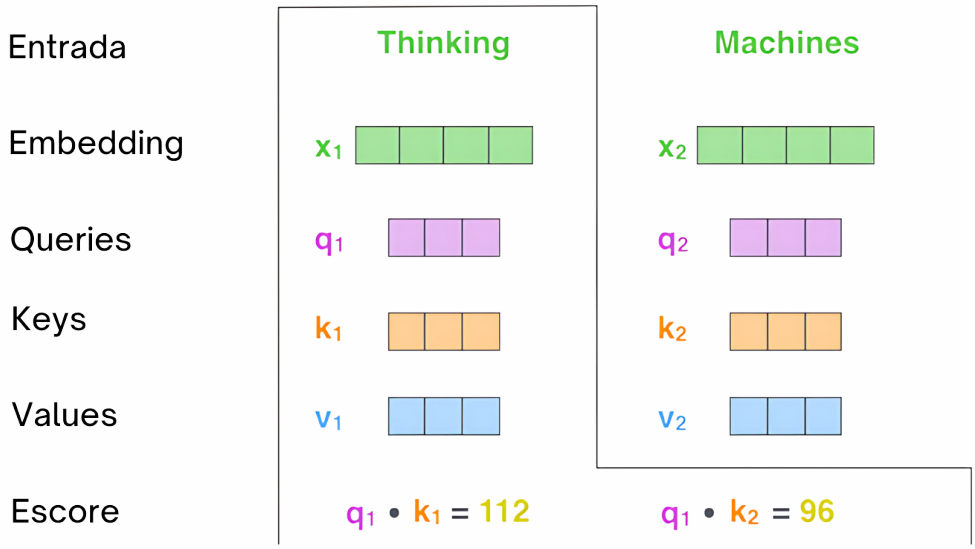
Figura 19 - Exemplo da criação do *Query*, *Key* e *Value*



Fonte: adaptada de Alammar (2018).

Para cada palavra inserida na arquitetura, são criados três diferentes vetores chamados de *Query*, *Key* e *Value*. Os nomes são uma analogia a sistemas de busca de informação onde *Query* (pode ser traduzido como consulta) é a busca a ser feita, como ao usar o Google® o que é digitado na barra de busca é a *Query*. O *Key* pode ser visto como um identificador das informações dos elementos buscados, como o título de um *site*, seu conteúdo e suas imagens, cada um é um *Key* diferente. Já o *Value* é o valor encontrado, então se *Key* pode ser o título de um *site*, cada *site* tem seu valor diferente para essa chave. Na autoatenção, cada um dos três elementos são gerados para cada uma das palavras de entrada por uma camada *feedforward*, dessa forma o modelo tem liberdade para gerar três vetores independentes entre eles, como demonstrado na Figura 18, onde q1 é a *Query* de x1 e q2 a de x2.

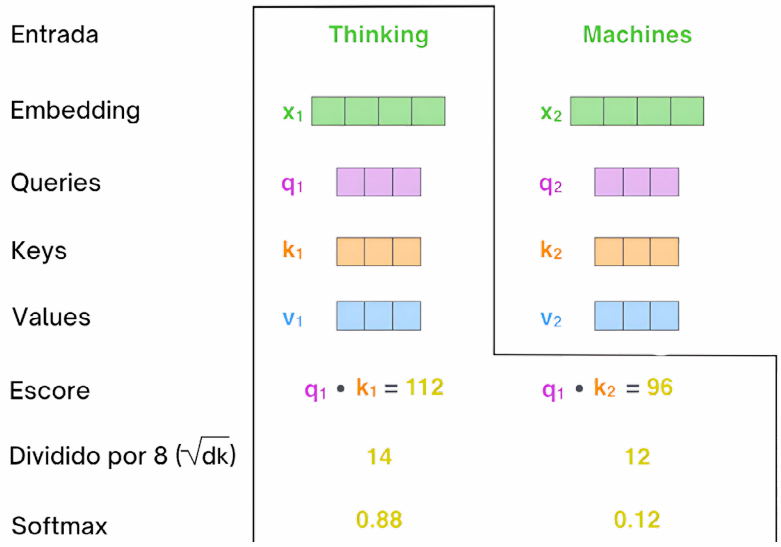
Figura 20 - Operação entre *Query* e *Key*



Fonte: adaptada de Alammar (2018).

Após gerar os três vetores, os *Queires* e *Keys* são multiplicados para ponderar o quão relevante cada palavra é em relação a todas as outras. Para iss, o *Query* da palavra *"Thinking"* é multiplicado pelo *Key* também de *"Thinking"* e pelo *Key* de *"Machines"*. Esse escore calculado é considerado a atenção bruta, como na Figura 19.

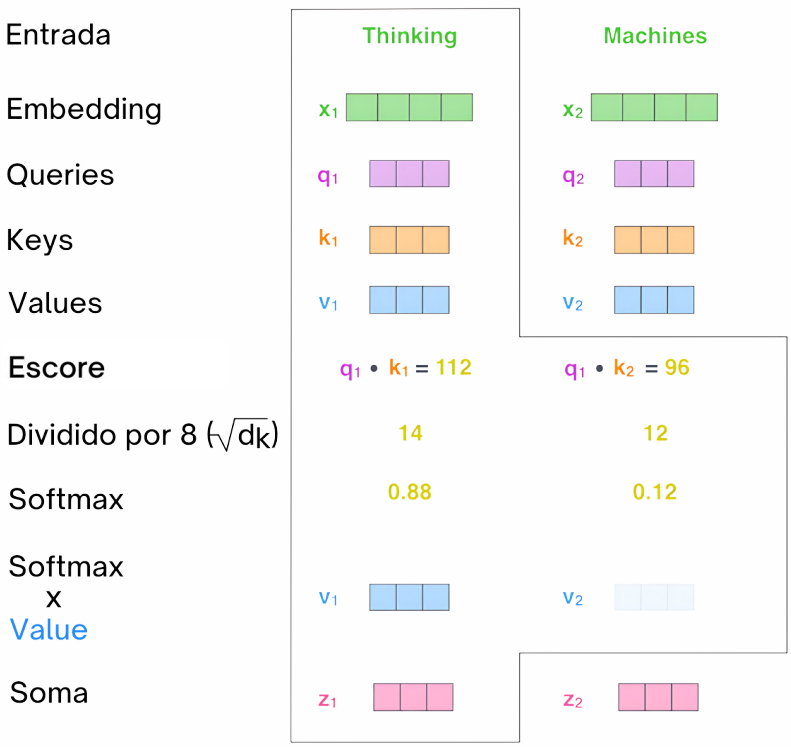
Figura 21 - Normalização da atenção



Fonte: adaptada de Alammar (2018).

Em seguida, os escores são normalizados, primeiro, a partir da raiz da dimensão do vetor Key, que, por padrão, costuma ser 64, dessa forma a raiz quadrada é 8. Isso é feito para melhorar a estabilidade do treino. Por fim, é feita uma normalização *Softmax* em todos os escores para que todos sejam positivos, com soma total 1 e que as palavras mais relevantes se destaquem em relação a outras.

Figura 22 - Operação dos *Values*

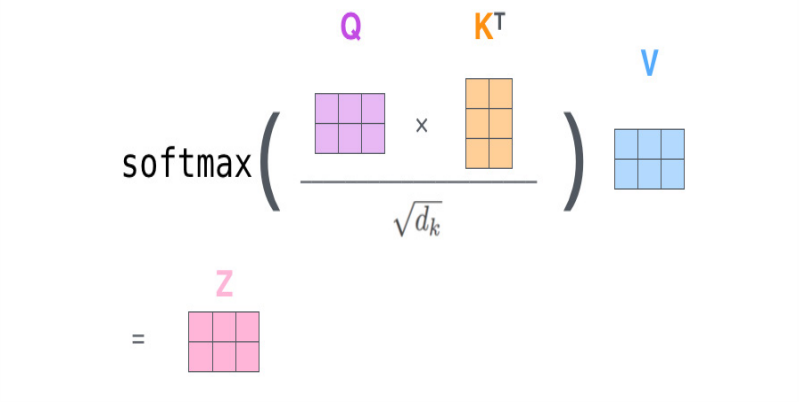


Fonte: adaptada de Alammar (2018).

Logo em seguida, o valor pós *Softmax* é multiplicado aos *Values* de suas respectivas palavras. Com isso, palavras menos relevantes terão vetores menores. Após isso, os *Values* ponderados são somados formando o vetor *z* que representa o produto de

todo o processo da autoatenção, como demonstrado na Figura 21. Esse processo é feito para todas as palavras de entrada, que pode ser resumido como uma série de operações matriciais (Figura 21).

Figura 23 - Equação matricial da autoatenção

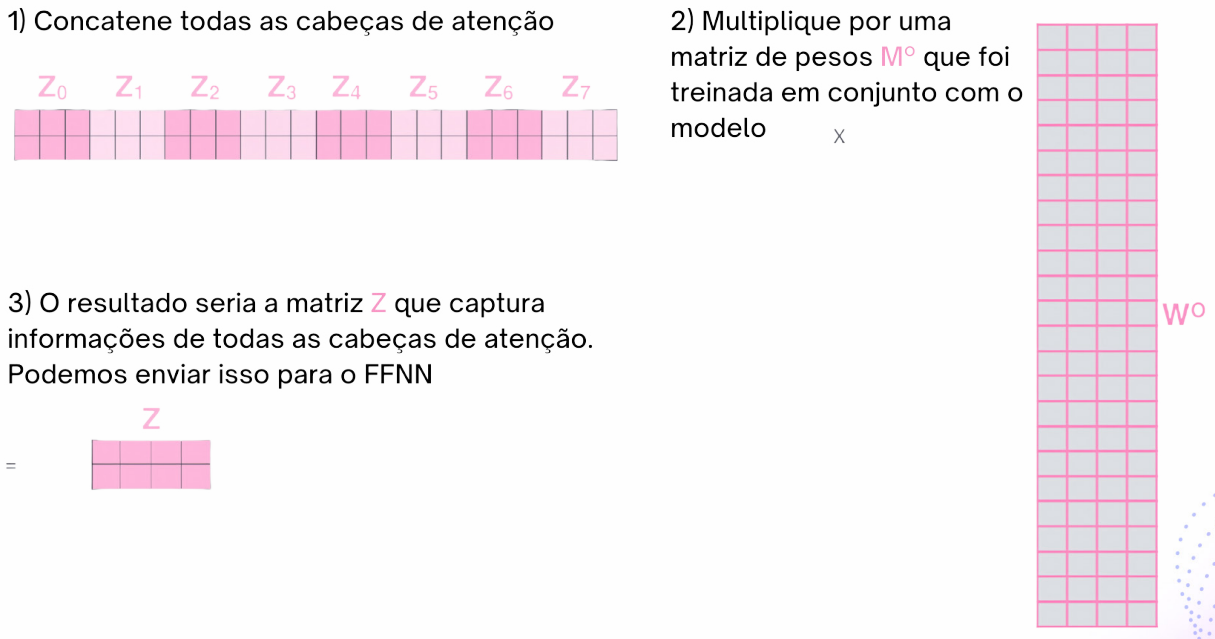


Fonte: adaptada de Alammar (2018).

2.1.2 Cabeças de Autoatenção

Todo esse processo descrito anteriormente é apenas uma autoatenção. A arquitetura *Transformer* usa múltiplas cabeças de autoatenção *(Multi-Head Attention)* de forma que temos várias repetições do mesmo processo. Por padrão, são usadas oito cabeças de autoatenção onde cada uma possui seus parâmetros independentes, gerando assim oito vetores *z* para cada entrada.

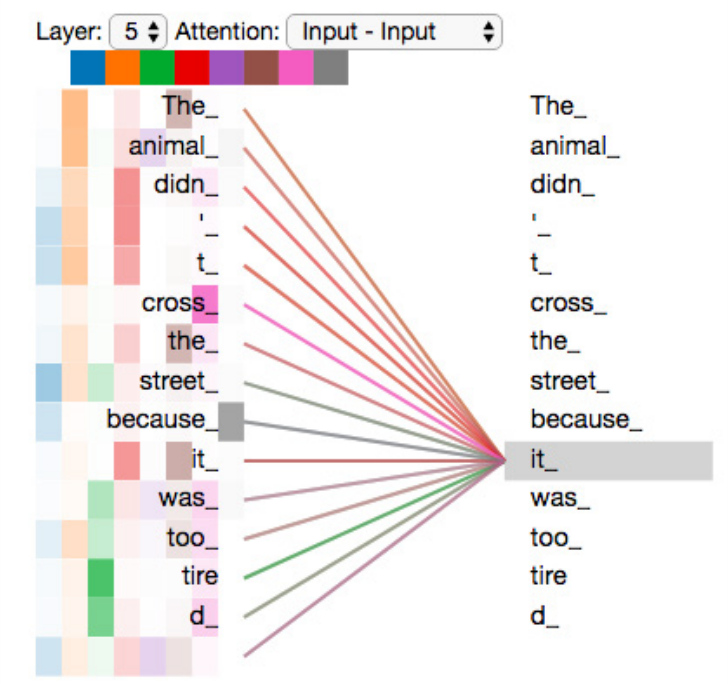
Figura 24 - Combinação das múltiplas cabeças de autoatenção



Nota: FFNN: *Feed-Forward Neural Networks*. Fonte: adaptada de Alammar (2018).

Ao gerar todos os vetores *z*, uma camada final realiza a combinação de todas as saídas para um único vetor *z*, como demonstrado na Figura 22. Inicialmente, o exemplo foi mostrado apenas como uma cabeça de autoatenção, ao olhar todas as oito na Figura 20 abaixo podemos ver que o processamento é bem mais complexo. As cores na imagem indicam cada uma das cabeças de atenção e assim podemos ver que apenas uma das cabeças (a laranja) está capturando a relação de "it" com "The animal". O conceito das múltiplas cabeças é justamente este: permitir que a arquitetura seja capaz de capturar diversas relações simultaneamente.

Figura 25 - Exemplo de autoatenção com todas as cabeças

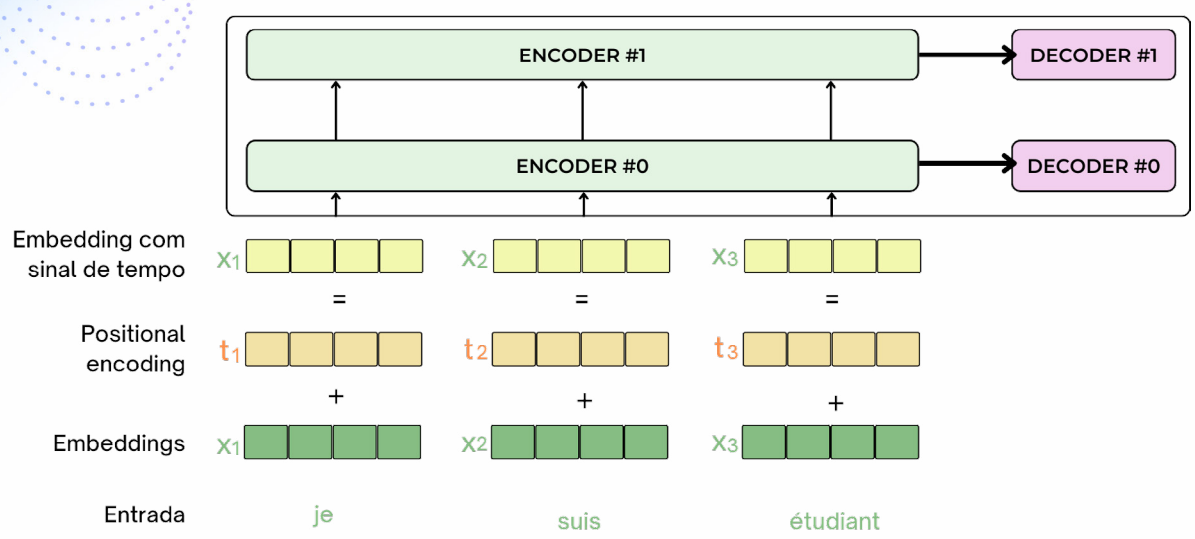


Fonte: adaptada de Alammar (2018).

2.1.3 *Positional Encoding*

Um elemento não menos importante no *Transformer* é que as camadas de autoatenção executam apenas operações de *feedforward*, dessa forma, não existe compreensão de ordem como discutido no início deste *ebook*. Para solucionar tal problema, foi proposto o *Positional Encoding*.

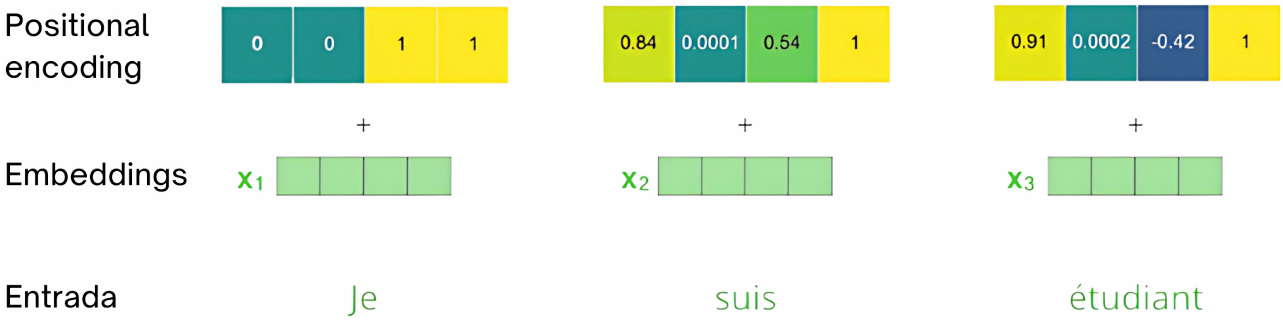
Figura 26 - *Positional Encoding*



Fonte: adaptada de Alammar (2018).

Ao *embedding* de cada palavra, é adicionado um vetor que descreve a posição em que tal palavra ocorre. Esses vetores de posição são obtidos a partir de funções seno e cosseno, discretizadas a cada posição. Como na Figura 25 a seguir, onde cada coluna é uma palavra e seu vetor de posição, podemos ver que cada vetor é único e possui a propriedade de ser possível calcular quantas posições se passam entre um e outro.

Figura 27 - *Positional Encoding* em cada palavra

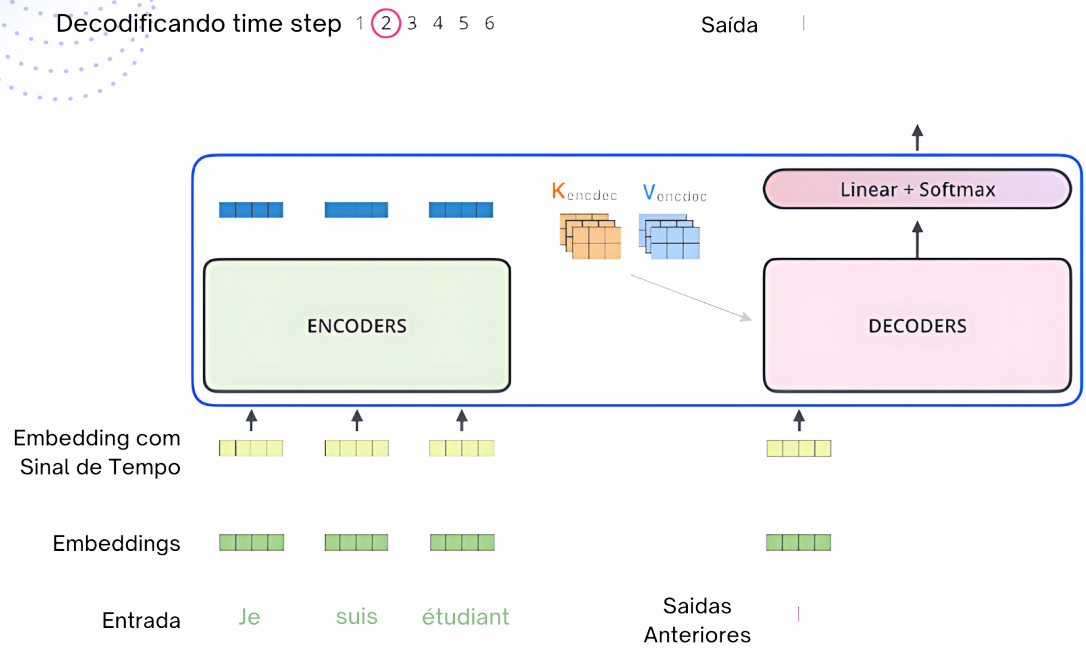


Fonte: adaptada de Alammar (2018).

2.1.4 Autoatenção *Encoder-decoder*

A explicação dada até agora é apenas da autoatenção comum, mas o *Transformer* possui também outro tipo que é a atenção *encoder-decoder*. Por ser uma arquitetura proposta para tradução é importante considerar a comunicação entre *encoder* e *decoder*. Para isso, o *decoder* possui camadas especiais, onde o *Query* vem das palavras já geradas e *Key* e *Value* são obtidos do *encoder*, cruzando, assim, as informações e permitindo gerar a tradução com base no idioma de entrada. Na Figura 26, podemos ver um exemplo dessa interação entre *encoder* e *decoder*.

Figura 28 - Autoatenção *encoder-decoder*



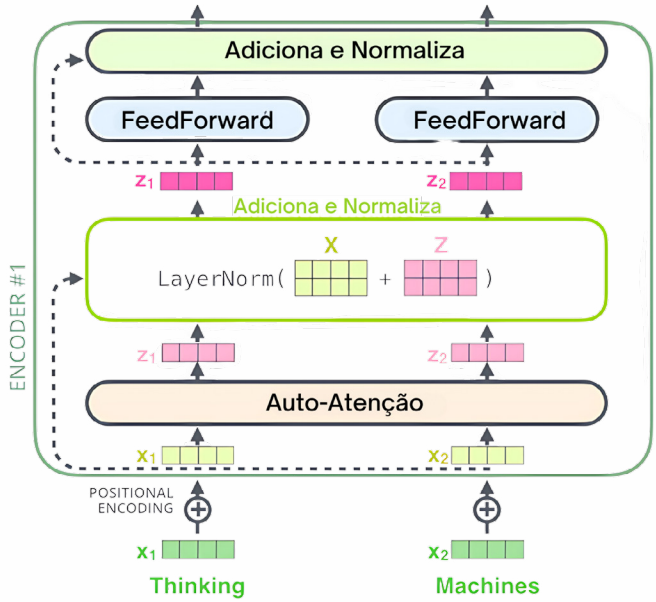
Fonte: adaptada de Alammar (2018).

A operação de autoatenção apresentou um significativo avanço, pois não possui os problemas de *vanish/exploding* como em uma RNN, de forma que os pesos não reocorrem. Outro ponto importante é que, em uma RNN, é necessário calcular cada tempo na devida ordem, impedindo a paralelização da sua execução, enquanto que a autoatenção por ser uma única operação de matrizes e pode ser executada em paralelo, permitindo, assim, a implementação de treinamento distribuído em várias Unidades de Processamento Gráfico (GPUs).

2.1.5 Conexões Residuais

Semelhante às CNNs e também às LSTM de maior porte, a arquitetura *Transformer* possui conexões residuais que conectam a entrada de cada camada com sua saída. Isso é feito tanto na autoatenção quanto na *feedforward*. Essas operações são usadas para dar mais estabilidade ao processo de treino, reduzindo o tempo necessário para se obter bons resultados e são conectadas como mostradas na Figura 27.

Figura 29 - Conexões residuais e normalização

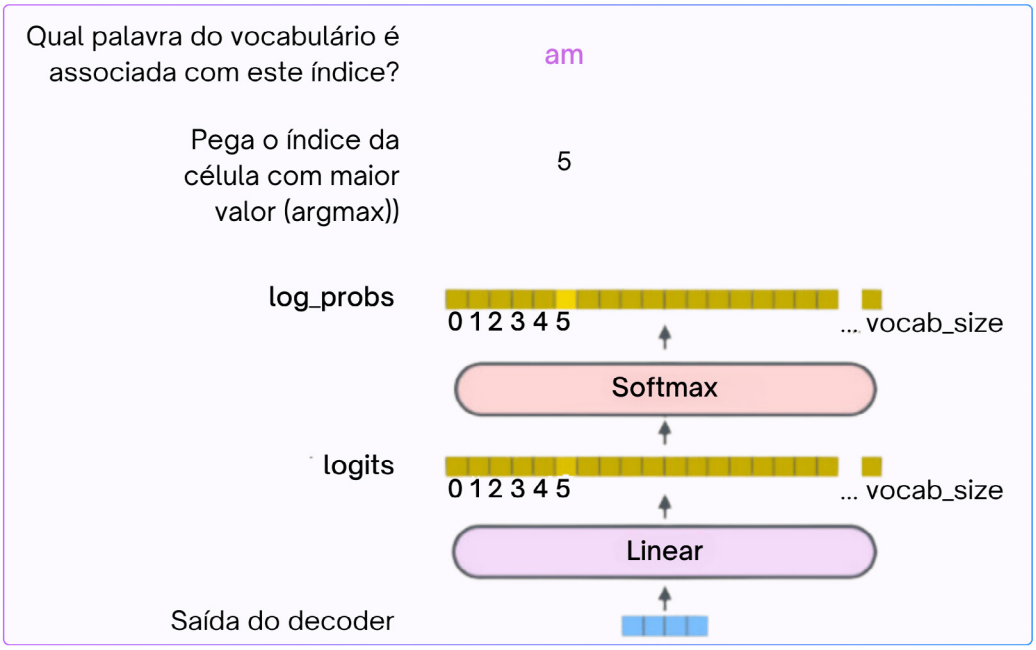


Fonte: adaptada de Alammar (2018).

2.1.6 Camada Final

No fim da arquitetura, temos uma camada *feedforward* (como pode ser observado na Figura 27) que realiza a predição da palavra a ser gerada. Essa camada possui o tamanho do vocabulário e após a normalização *Softmax* gera uma probabilidade associada a cada palavra possível. Na Figura 28, é demonstrado como a palavra com maior probabilidade é escolhida como resposta final.

Figura 30 - Camada final e processo de resposta da arquitetura

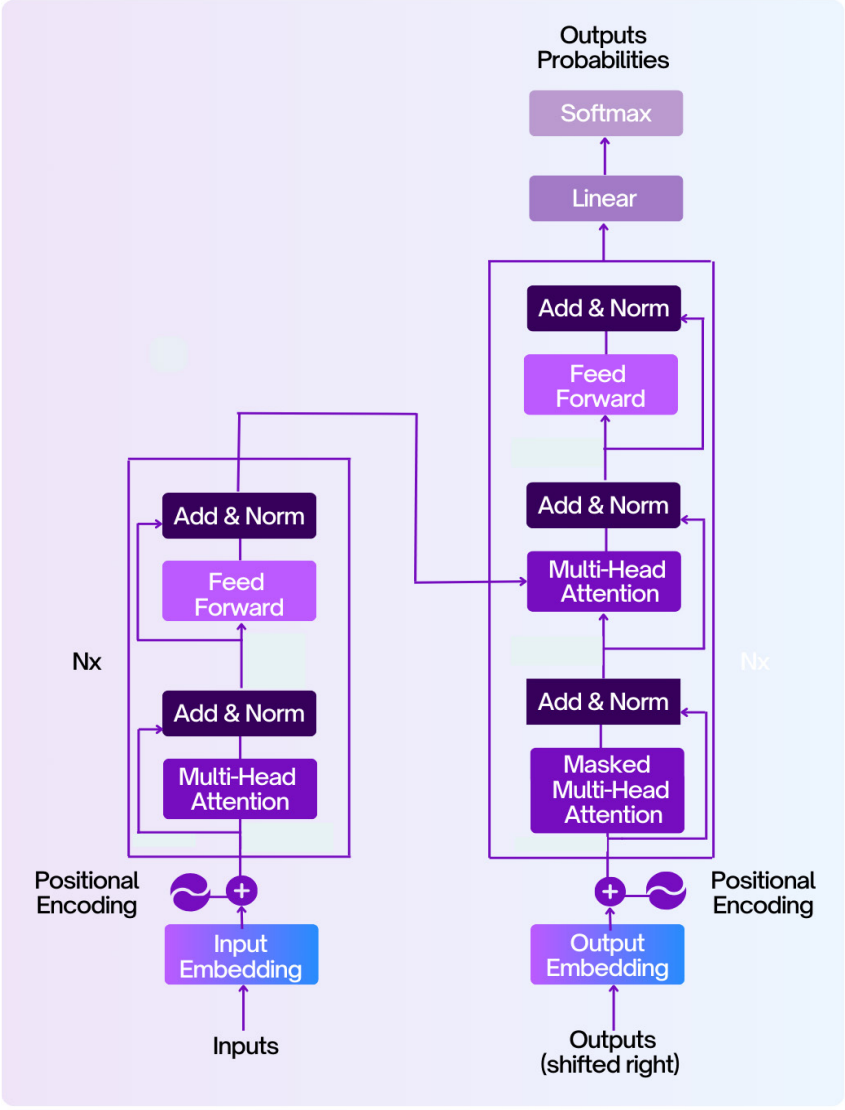


Fonte: adaptada de Alammar (2018).

2.1.7 Treinamento e Uso do *Transformer*

Por fim, podemos observar toda a arquitetura como uma estrutura única na Figura 29. A rede, então, foi treinada inicialmente na tarefa de tradução de textos e obteve performance significativamente superior a todas as alternativas baseadas em LSTMs. Isso chamou atenção não apenas para a tarefa de tradução, mas também para os estudos voltados à representação de palavras e sentenças. Durante o mesmo período (entre 2017 e 2018), existiam diversas pesquisas em torno de modelos que com uso de RNNs, que buscavam criar representações contextuais de palavras, algo como o *Word2Vec*, mas com capacidade de considerar uma janela de contexto em torno da palavra. Porém, após a publicação da arquitetura *Transformer* esta se mostrou uma ótima alternativa às RNNs em termos de representação. Na Unidade a seguir, iremos discutir os estudos realizados que usam de RNNs para a representação e como esses evoluíram para o uso de *Transformers*.

Figura 31 - Arquitetura *Transformer*



Fonte: autoria própria.

SAIBA MAIS...

- The Illustrated Transformer (Alamar, 2018).
- Neural machine translation with a Transformer and Keras (The TensorFlow Authors, 2024).
- The Annotated Transformer (Klein *et al.*, 2017)
- Deep Learning (Goodfellow; Bengio; Courville, 2016).

Arquiteturas Profundas para Processamento de Linguagem Natural

Unidade III

Histórico ULMFiT, GPT, Elmo

Unidade III: Histórico ULMFiT, GPT, Elmo

Como apresentado nas Unidades anteriores, o desenvolvimento de aplicações de PLN com Redes Neurais usaram por anos (principalmente entre 2013 e 2017) modelos baseados em representações de palavras, como o *Word2Vec*, combinadas com RNNs. No entanto, esses modelos apresentavam uma limitação significativa: a dificuldade de considerar o contexto dinâmico em que as palavras são usadas.

A ideia de pré-treinamento, inicialmente popularizada em Redes Neurais no campo da visão computacional com o ImageNet e CNNs pré-treinadas, inspirou abordagens semelhantes no PLN. Grandes modelos de CNN eram pré-treinados em um grande conjunto de dados de imagens, em uma tarefa mais simples, que consistia em classificar as imagens com base no seu conteúdo. Isso permitiu aos modelos aprender características visuais gerais que podiam ser aplicadas a diversas tarefas específicas após um ajuste fino *(fine-tuning)* relativamente pequeno na rede. Essa abordagem permitiu economizar recursos computacionais e melhorar o desempenho em tarefas específicas.

Inspirados por tal abordagem surgiram modelos pré-treinados de representação de palavras com base em contexto, como o *Universal Language Model Fine-tuning* (ULMFiT) e o *Embeddings from Language Models* (ELMo). Esses modelos, que são baseados em RNNs, introduziram uma nova era de representações contextuais, onde o significado de uma palavra é ajustado com base no seu uso específico em uma sentença. Posteriormente, a evolução desses modelos levou ao desenvolvimento do *Bidirectional Encoder Representations from Transformers* (BERT), que aperfeiçoou ainda mais a incorporação do contexto ao utilizar a arquitetura *Transformer*. Essa progressão, do *Word2Vec* ao BERT, ilustra a transformação fundamental na maneira como a linguagem é modelada e processada, refletindo um movimento contínuo em direção a representações mais contextualmente ricas e semanticamente precisas.

Nesta Unidade, os modelos ULMFiT e ELMo serão explorados, discutindo-se sobre suas arquiteturas, processo de treino e utilização. A Unidade seguinte será focada no BERT com suas aplicações e como ele inspirou o surgimento dos Grande Modelos de Linguagem.

Um **Modelo de Linguagem** é um tipo de sistema de inteligência artificial projetado para entender, gerar ou prever texto em linguagem natural, com base em gran-

des conjuntos de dados textuais. Ele é caracterizado pela capacidade de capturar padrões linguísticos e semânticos a partir de dados, sendo usado em várias tarefas de NLP. Já o *Large Language Model* (LLM) é uma versão significativamente maior, treinada em enormes volumes de dados e composta por bilhões ou até trilhões de parâmetros. O que os diferencia é a escala: enquanto modelos de linguagens menores podem realizar tarefas específicas com eficiência, os LLMs têm maior capacidade de generalização, oferecendo respostas mais complexas, com maior fluidez e precisão, graças ao seu treinamento massivo em dados diversos e sua arquitetura extensa.

3.1 *Universal Language Model Fine-tuning for Text Classification* (ULMFiT)

O ULMFiT, introduzido por Jeremy Howard e Sebastian Ruder em 2018, revolucionou a forma como modelos de linguagem podem ser aplicados a tarefas específicas com eficiência. Antes do ULMFiT, os LMs eram, geralmente, treinados do zero para cada tarefa específica, o que demandava uma quantidade significativa de dados rotulados e recursos computacionais. O ULMFiT propôs um processo em três etapas: pré-treino de um LM, ajuste fino em um *corpus* específico e, finalmente, ajuste fino na tarefa específica.

O ULMFiT utiliza uma arquitetura baseada em LSTM, sendo assim uma RNN. As LSTMs são eficazes em capturar dependências de longo prazo em sequências de dados, o que é crucial para tarefas de PLN.

3.1.1 Pré-treino do Modelo de Linguagem

O pré-treino é a fase mais crucial no ULMFiT, pois permite que o modelo aprenda uma representação rica da linguagem a partir de um grande *corpus* de texto não rotulado. Esse pré-treino consiste em várias etapas, descritas a seguir.

3.1.1.1 Treinamento em um *Corpus* Grande e Genérico

O modelo de linguagem é inicialmente treinado em um *corpus* grande e genérico, como o WikiText-103, que contém artigos da *Wikipedia*. O objetivo desse treinamento é ensinar ao modelo a previsão da próxima palavra em uma frase, uma tarefa conhecida como modelagem de linguagem. A função de perda utilizada é a de entropia cruzada, que mede a diferença entre a distribuição predita pelo modelo e a distribuição real das palavras.

3.1.1.2 Técnicas de Regularização e Otimização

Diversas técnicas de regularização são aplicadas para evitar o *overfitting* durante o pré-treino. Entre elas estão a regularização de *dropout*, a ativação gradual de camadas (*gradual unfreezing*), e o uso de múltiplas taxas de aprendizado (*triangular learning rates*):

- **Dropout:** é uma técnica na qual, durante o treinamento, alguns neurônios são desativados aleatoriamente, o que ajuda a tornar o modelo mais robusto;
- **Ativação gradual de camadas:** envolve descongelar as camadas do modelo gradualmente durante o ajuste fino, começando pelas camadas superiores;
- **Múltiplas taxas de aprendizado:** permitem que diferentes camadas do modelo sejam treinadas com diferentes velocidades, otimizando o processo de aprendizagem.

3.1.1.3 Captura de Dependências de Longo Alcance

As LSTMs utilizadas no ULMFiT são particularmente eficazes em capturar dependências de longo alcance nas sequências de texto, o que é essencial para a compreensão do contexto completo em tarefas de PLN. O pré-treino permite que o modelo desenvolva um entendimento profundo das estruturas sintáticas e semânticas da linguagem.

3.1.2 Ajuste Fino no Corpus Específico (LM)

Após o pré-treino em um *corpus* genérico, o modelo é ajustado em um *corpus* específico relacionado à tarefa de destino. Esse estágio envolve:

- **Transferência de aprendizado:** o modelo pré-treinado é adaptado para o novo *corpus*, permitindo que ele aproveite o conhecimento adquirido durante o pré-treino. Esse ajuste fino (LM) no *corpus* específico ajuda o modelo a adaptar-se aos padrões linguísticos e ao vocabulário específico do domínio;
- **Congelamento e descongelamento de camadas:** inicialmente, as camadas inferiores do modelo são congeladas (não são atualizadas) e apenas as camadas superiores são ajustadas. Gradualmente, as camadas inferiores são descongeladas e ajustadas para otimizar a performance no novo *corpus*.

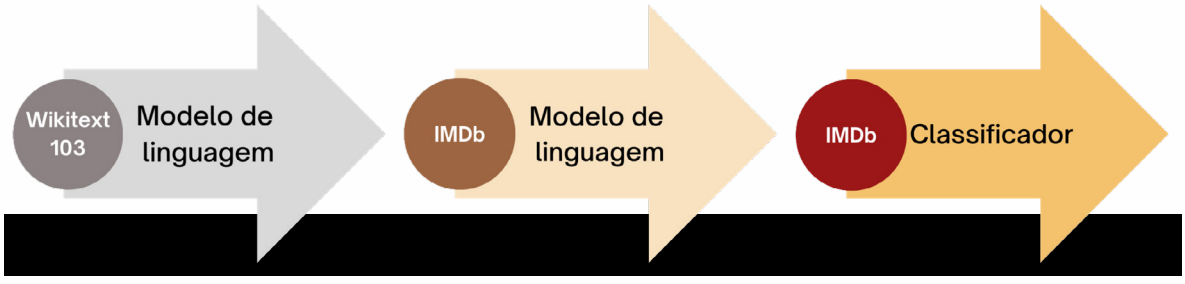
3.1.3 Ajuste Fino da Tarefa

O último estágio envolve ajustar o modelo na tarefa específica, como classificação de texto, detecção de sentimentos ou qualquer outra tarefa de PLN. Esse ajuste inclui:

- **Adição de camadas de classificação:** camadas são adicionadas ao modelo para realizar a tarefa específica, como uma camada *Softmax* para classificação de texto. Essas camadas adicionais são treinadas juntamente com o restante do modelo.
- **Treinamento supervisionado:** o modelo é treinado utilizando dados rotulados específicos da tarefa, ajustando os parâmetros para minimizar a função de perda específica da tarefa. Durante esse treinamento, técnicas de regularização continuam a ser aplicadas para evitar o *overfitting*.
- **Avaliação e ajuste final:** o desempenho do modelo é avaliado utilizando métricas específicas da tarefa, como acurácia, precisão, *recall,* e *F1-score*. Baseados na avaliação, ajustes finais são realizados para otimizar a performance do modelo.

Os passos do treinamento do ULMFiT serão apresentados nas três figuras, a seguir, de maneiras diferentes para auxiliá-lo na compreensão. Na primeira, Figura 30, há um exemplo simplificado do fluxo do treinamento com bases de dados. De forma resumida, temos três fases: na primeira, há pré-treino do modelo que, nesse caso, foi realizado com o Wikitext103[1]; na segunda há o *fine-tuning* empregando o IMDb *dataset*[2] e, no terceiro, há a classificação de dados do conjunto IMDb *dataset*.

Figura 32 - Treinamento ULMFiT



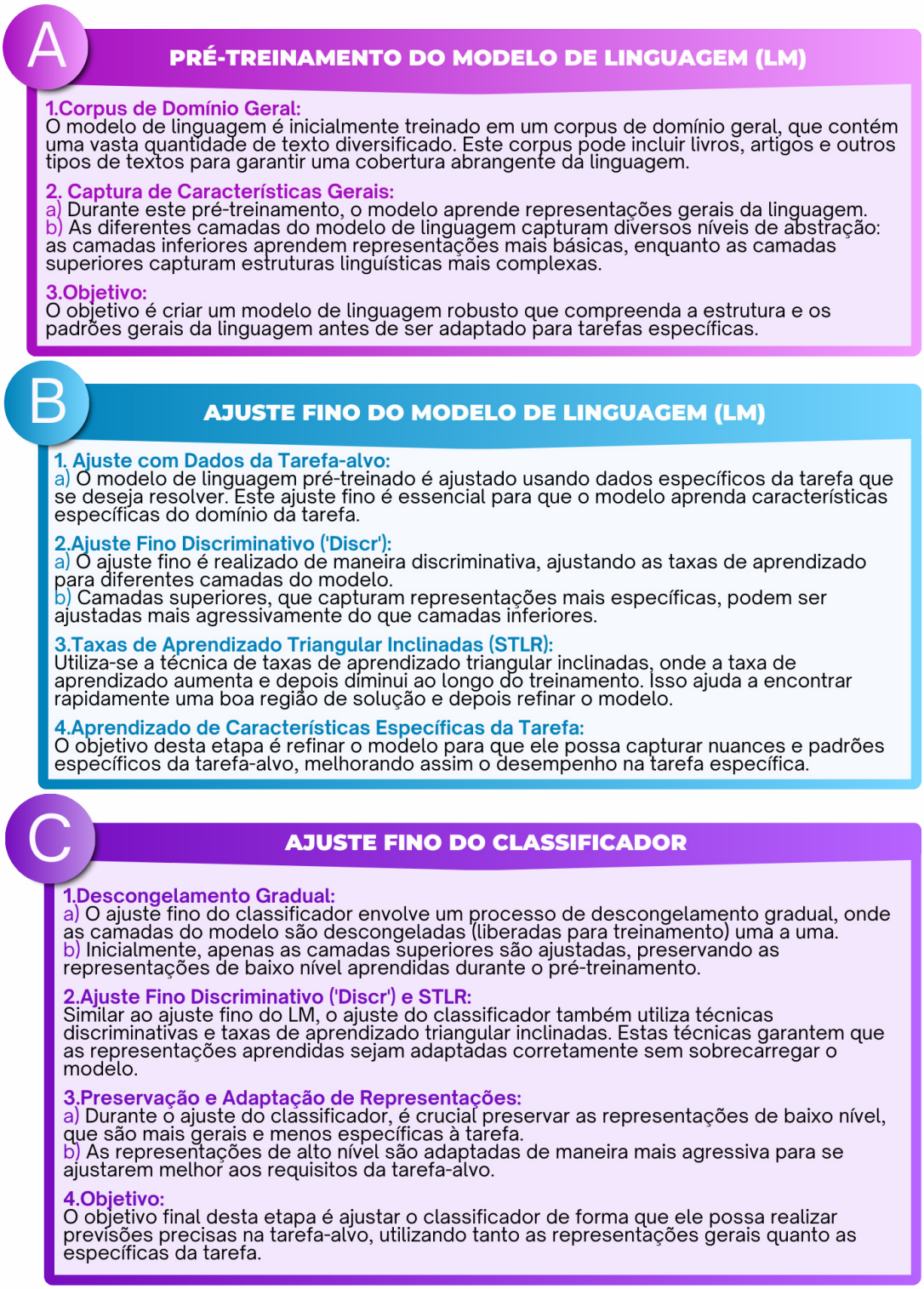
Fonte: Guillou (2018).

1 Trata-se de um conjunto de dados criado para treinar modelos de linguagem. Ele é composto por artigos da Wikipedia e contém cerca de 103 milhões de palavras.

2 Trata-se de um conjunto de dados que contém 50.000 resenhas de filmes, sendo 25.000 positivas e 25.000 negativas. É geralmente empregado ajuste de modelos em treinamento para posterior classificação de resenhas positivas ou negativas.

Os passos do processo de treinamento do ULMFiT, em três etapas principais, são detalhados na Figura 31, os quais são cruciais para adaptar o modelo às tarefas específicas de PLN.

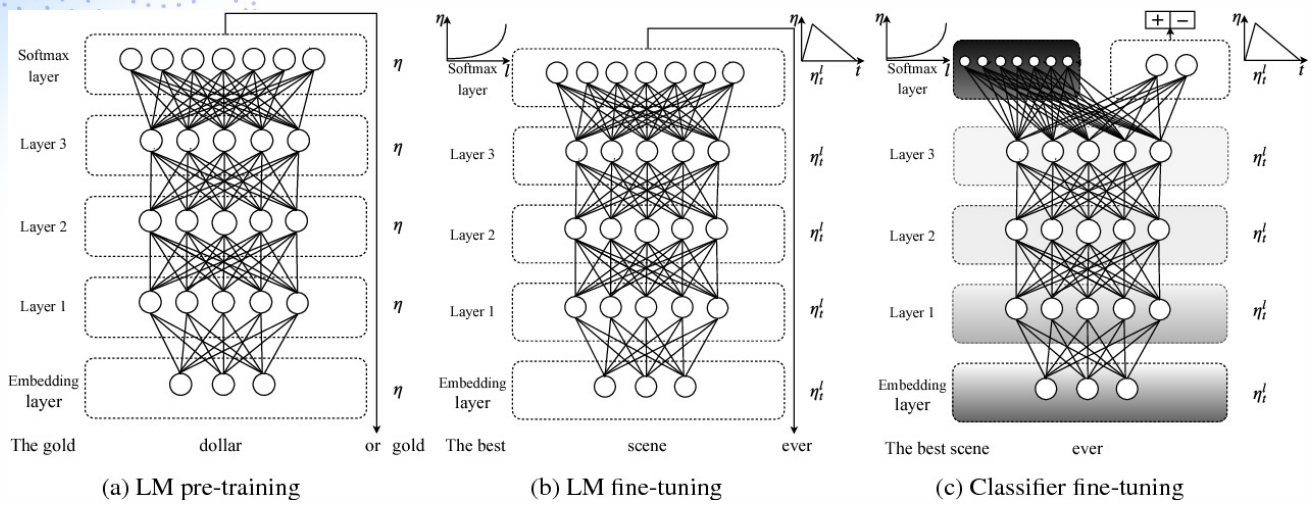
Figura 33 - Descrição do treinamento do ULMFiT



STRL: *Sloped Triangular Learning Rate*. Fonte: autoria própria.

Na Figura 32, é possível observar esse mesmo processo sendo realizado por meio da arquitetura do modelo, demonstrando a complexidade envolvida por trás de cada etapa.

Figura 34 - Detalhes do modelo ULMFiT



Fonte: adaptada de Howard; Ruder (2018).

3.1.4 Vantagens do ULMFiT

O ULMFiT demonstrou ser altamente eficiente em tarefas de classificação de texto. Entre as principais vantagens desse modelo, destacam-se a sua capacidade de adaptação a domínios específicos com poucos dados anotados e a sua eficácia em reduzir a necessidade de grandes quantidades de dados para obter bons resultados. Aplicações típicas do ULMFiT incluem análise de sentimentos e classificação de tópicos.

- **Eficiência de dados:** ULMFiT mostrou ser altamente eficiente, necessitando de menos dados rotulados para atingir alta performance em tarefas de classificação específicas.
- **Flexibilidade:** o método de ajuste fino permite que o mesmo modelo pré-treinado seja adaptado para uma ampla variedade de tarefas de classificação de PLN.
- **Redução de custos:** ao reutilizar modelos pré-treinados, o ULMFiT reduz significativamente os custos computacionais associados ao treinamento de modelos do zero para cada tarefa.

3.2 *Embeddings from Language Models* (ELMo)

O ELMo, introduzido por Matthew Peters *et al.*, em 2018, trouxe um avanço significativo no campo do PLN ao introduzir *embeddings* contextuais, que consideram o contexto completo de uma palavra em uma frase para gerar suas representações vetoriais. Antes do ELMo, as representações de palavras como *Word2Vec* e *Global*

Vectors for Word Representation (GloVe)[3], eram estáticas, ou seja, uma palavra tinha a mesma representação independentemente do contexto no qual ela aparecia. O ELMo mudou isso ao gerar *embeddings* dinâmicos, ajustados de acordo com o contexto.

O ELMo utiliza uma arquitetura baseada em RNNs bidirecionais (*Bidirectional Long Short-Term Memory Network* - BiLSTM). Essa arquitetura permite que o modelo capture tanto o contexto anterior quanto o posterior de uma palavra, o que é crucial para entender o significado completo das palavras em diferentes contextos.

3.2.1 Pré-treino do Modelo de Linguagem

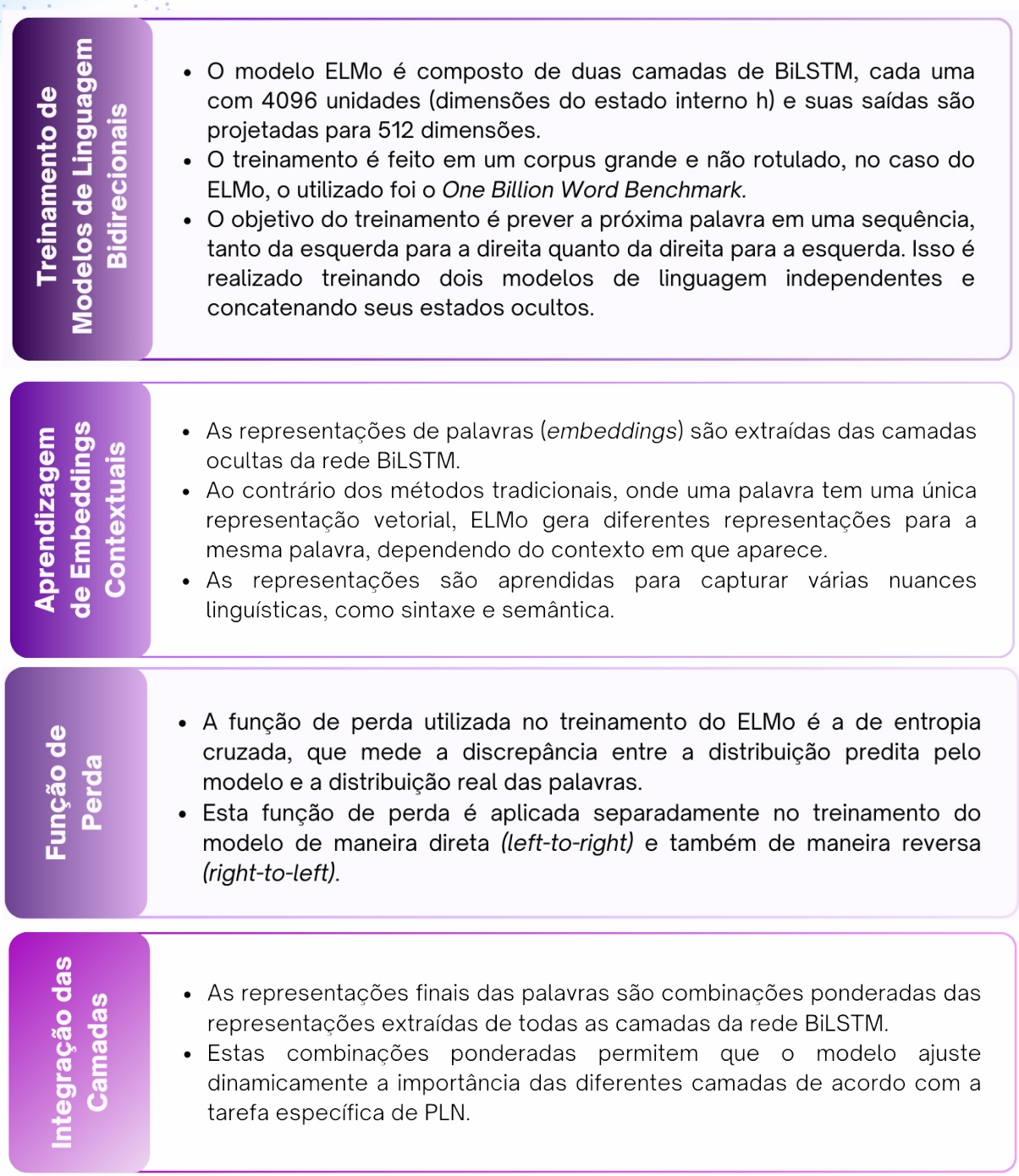
O pré-treino do modelo de linguagem é a fase mais crítica no ELMo, onde o modelo é treinado em um grande *corpus* de texto para aprender representações contextuais. Vale ressaltar que um modelo de linguagem é um modelo treinado para compreender e gerar texto natural. Ele aprende as estruturas e padrões da linguagem analisando grandes volumes de texto e capturando as relações entre palavras e frases.

O ELMo utiliza uma RNNs bidirecional, na qual o processamento da sequência de texto ocorre em duas direções, treinamento do modelo de maneira direta (*left-to-right*) e também de maneira reversa (*right-to-left*). No treinamento direto, a rede é treinada para prever a próxima palavra em uma frase considerando apenas as palavras anteriores. No treinamento reverso, além de olhar para o contexto da esquerda, ou seja, para as palavras anteriores, o modelo é treinado para tentar prever as palavras anteriores com base nas palavras seguintes.

O pré-treino envolve várias etapas a serem detalhadas a seguir (Figura 33).

3 O GloVe é um algoritmo de aprendizado usado em PLN para gerar representações vetoriais estáticas (ou *embeddings*) de palavras. Esses *embeddings* são gerados com base na ocorrência das palavras em um grande corpus de texto. Esse algoritmo parte do princípio de que o significado de uma palavra pode ser capturado observando a frequência com que ela aparece ao lado de outras palavras em um determinado contexto.

Figura 35 - Etapas para pré-treino do modelo de linguagem



3.2.2 Aplicações das Representações ELMo

As representações ELMo são utilizadas como características adicionais em diversos modelos de PLN para melhorar seu desempenho. Elas podem ser incorporadas em arquiteturas baseadas em redes neurais, como LSTMs ou *Transformers*, bem como em modelos tradicionais de aprendizado de máquina.

Após o pré-treino, as representações ELMo são ajustadas em tarefas específicas de PLN. Esse ajuste envolve:

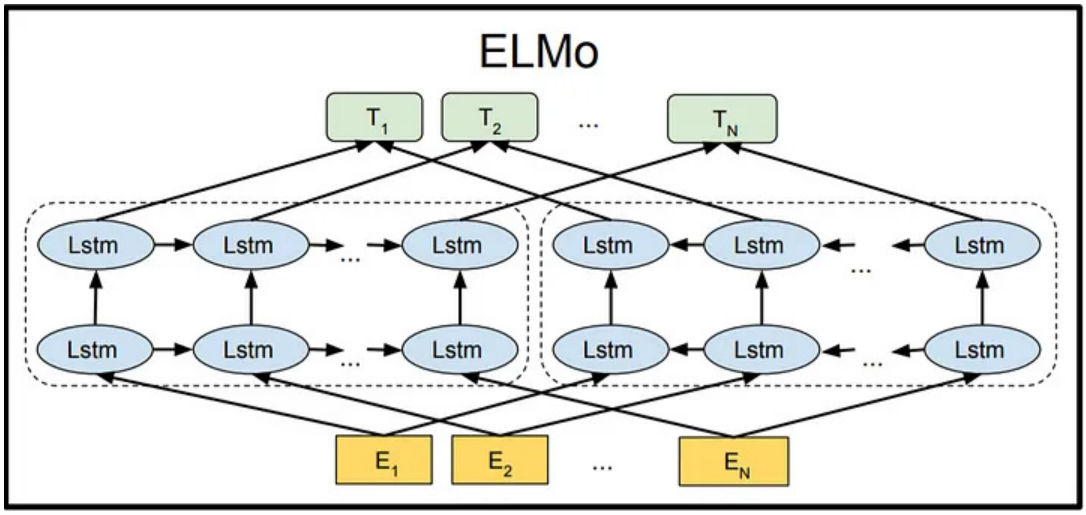
1. **Extração de embeddings contextuais:** durante a aplicação em tarefas específicas, as representações ELMo são extraídas de cada palavra em um texto, considerando seu contexto completo. Essas representações são então utilizadas como entradas adicionais para modelos de PLN, enriquecendo a informação disponível para a tarefa.

2. **Integração com modelos de tarefas específicas:** as representações ELMo podem ser concatenadas com outras características de entrada, como *embeddings* estáticos ou características manuais. Modelos de tarefas específicas, como Reconhecimento de Entidades Nomeadas (*Named Entity Recognition* - NER), análise de sentimentos ou resolução de correferências, utilizam essas representações enriquecidas para melhorar seu desempenho.

3. **Treinamento supervisionado:** o modelo final que integra as representações ELMo é treinado utilizando dados rotulados específicos da tarefa. Esse treinamento supervisionado ajusta os parâmetros do modelo para otimizar a performance na tarefa específica, utilizando técnicas de regularização e de ajuste fino.

A Figura 34, é mostrada uma representação da arquitetura do modelo ELMo e seus componentes estão descritos na Tabela 2.

Figura 36 - Arquitetura ELMO



Fonte: Tsang (2022).

Tabela 2 - Descrição da representação da arquitetura ELMo

Item	Descrição	Representação
T (Tokens)	Cada token T_i representa uma palavra ou subpalavra na frase de entrada.	Caixas em verde $T_1, T_2, ..., T_n$
LSTM Layers (camadas)	Na figura temos duas camadas de LSTMs bidirecionais empilhadas. Os nós ovais representam as camadas LSTM com suas iterações explícitas: • A primeira camada de LSTMs bidirecionais processa a sequência de tokens e produz uma representação para cada token considerando o contexto anterior e posterior. • A segunda camada de LSTMs bidirecionais toma as representações da primeira camada e refina ainda mais o contexto. • As setas bidirecionais indicam que as LSTMs estão processando as informações em ambas as direções (da esquerda para a direita e da direita para a esquerda).	Elipses em azul
E *(Embeddings)*	Cada *embedding* de saída E_i é uma combinação das representações das duas camadas de LSTMs para o token correspondente T_i. Esses *embeddings* são dinâmicos, ou seja, são dependentes do contexto da palavra na frase.	Caixas em amarelo $E_1, E_2, ..., E_n$

Fonte: autoria própria.

3.3 Evolução dos Modelos de Linguagem

A evolução dos modelos de linguagem natural tem sido marcada por sucessivos avanços na capacidade de compreensão e geração de texto, começando com o ULMFiT, que demonstrou o potencial do *fine-tuning* universal para tarefas variadas de NLP. Em seguida, o BERT trouxe uma inovação ao introduzir o pré-treinamento bidirecional, permitindo uma compreensão mais profunda do contexto em textos. O modelo T5 deu um passo adiante ao reformular todas as tarefas de NLP como problemas de geração de texto, estabelecendo uma nova perspectiva no processamento de linguagem. A arquitetura *Generative Pre-trained Transformer* (GPT), com seu foco autoregressivo, elevou ainda mais o padrão na geração de texto, destacando-se especialmente com o GPT-3, pela sua capacidade de produzir respostas altamente coerentes e detalhadas. Posteriormente, o *Fine-tuned Language Net* (FLAN) T5 surgiu como uma evolução do T5, aprimorando a adaptação de tarefas por meio de instruções e consolidando a versatilidade desses modelos no cenário atual de NLP. Na Figura 35, temos uma linha do tempo que demonstra essa evolução.

Figura 37 - Linha do tempo para os

MODELOS DE LINGUAGEM

2018

Março: ELMo (*Embeddings from Language Models*)

- **Contribuição:** Introdução de *embeddings* contextualizados, permitindo que a representação de uma palavra varie conforme seu contexto na frase.
- **Impacto:** Significativo avanço em tarefas de PNL, melhorando a precisão em *benchmarks* (processo de avaliação comparativa que mede o desempenho de modelos em *datasets* específicos) como SQuAD e GLUE.

Abril: ULMFiT *(Universal Language Model Fine-tuning)*

- **Contribuição:** Proposta de um método de ajuste fino universal que permite que um modelo de linguagem pré-treinado seja adaptado para várias tarefas de PNL com poucos dados.
- **Impacto:** Estabeleceu uma abordagem eficiente para transfer *learning* em PNL, simplificando o treinamento de modelos para tarefas específicas.

Junho: GPT *(Generative Pre-trained Transformer)*

- **Contribuição:** Introdução do GPT, um modelo baseado em *Transformer* pré-treinado em um grande corpus de texto e ajustado para tarefas específicas.
- **Impacto:** Mostrou que modelos pré-treinados podem ser adaptados para diversas tarefas de PNL, estabelecendo uma nova referência para geração de texto.

2019

Fevereiro: BERT *(Bidirectional Encoder Representations from Transformers)*

- **Contribuição:** Introdução de um modelo bidirecional que considera o contexto de ambas as direções (esquerda e direita) para pré-treinamento.
- **Impacto:** Estabeleceu novos padrões de desempenho em vários *benchmarks* de PNL, incluindo SQuAD, GLUE e outros.

Novembro: GPT-2

- **Contribuição:** Expansão do GPT com 1,5 bilhões de parâmetros, demonstrando capacidades impressionantes de geração de texto e compreensão de linguagem.
- **Impacto:** Levantou questões sobre o uso ético e seguro de modelos de linguagem poderosos devido à sua capacidade de gerar texto convincente.

2020

Junho: T5 *(Text-to-Text Transfer Transformer)*

- **Contribuição:** Introdução de uma abordagem *"text-to-text"* unificada para todas as tarefas de PNL, utilizando a arquitetura *Transformer*.
- **Impacto:** Simplificou a abordagem para resolver múltiplas tarefas de PNL com um único modelo, demonstrando alto desempenho em diversos *benchmarks*.

Junho: GPT-3

- **Contribuição:** Expansão significativa do GPT com 175 bilhões de parâmetros, demonstrando capacidades avançadas de compreensão e geração de texto com pouca necessidade de ajuste fino.
- **Impacto:** Estabeleceu novos padrões de desempenho em tarefas de PNL e levantou debates sobre o impacto e as implicações éticas de modelos de linguagem tão poderosos.

2021

Dezembro: FLAN-T5 *(Fine-tuned LAnguage Net T5)*

- **Contribuição:** Adaptação do T5 com *fine-tuning* em instruções múltiplas para melhorar a generalização e a robustez em tarefas de PNL.
- **Impacto:** Melhorou a capacidade do T5 de generalizar para novas tarefas não vistas durante o treinamento, aumentando a utilidade prática do modelo.

3.4 Características dos Modelos de Linguagem

Na Tabela 3, é fornecida uma visão geral das principais características dos modelos ULMFiT e ELMo, destacando-se suas diferenças em termos de arquitetura, método de treinamento e aplicação.

Tabela 3 - Comparativo dos modelos ULMFiT e ELMo

Característica	ULMFiT	ELMo
Ano de Lançamento	2018	2018
Arquitetura	*Transfer Learning (Fine-Tuning)*	*Embeddings* Contextuais Profundos
Tamanho do Modelo	Aproximadamente 24 milhões de parâmetros	Aproximadamente 93 milhões de parâmetros
Método de Treinamento	Pré-treino em grandes volumes de texto seguido de *fine-tuning* específico em tarefas	Pré-treino em grandes volumes de texto para criar *embeddings* contextuais a partir de uma rede bidirecional LSTM
Pré-treino	Usando o modelo AWD-LSTM em textos grandes e variados	Utiliza LSTMs bidirecionais para gerar *embeddings* contextuais a partir de texto
Fine-Tuning	Sim, permite ajuste fino em tarefas específicas	Não, os *embeddings* são usados diretamente em tarefas *downstream*
Aplicação	Classificação de texto, análise de sentimentos, resposta a perguntas	Diversas tarefas de PLN como etiquetagem de sequências, resposta a perguntas, análise de sentimentos
Resultados Alcançados	Melhoria significativa em tarefas de PLN com pouca quantidade de dados	Resultados avançados em tarefas de PLN ao considerar o contexto da palavra em toda a frase
Vantagens	Adaptabilidade a tarefas específicas com pouco dado, simplicidade no *fine-tuning*	Captura de significados contextuais ricos para melhor entendimento do texto
Desvantagens	Necessidade de re-treinar para cada tarefa específica	Maior complexidade computacional e necessidade de maior poder computacional para treinar e usar
Principais Contribuições	Popularizou o uso de transferência de aprendizado em PLN	Introduziu a ideia de *embeddings* contextuais, influenciando modelos subsequentes

Fonte: autoria própria.

3.5 Exemplos Práticos

Nos Quadros 1 e 2 abaixo, serão apresentadas as aplicações práticas de ULMFiT e ELMo em tarefas de PLN, destacando-se suas principais vantagens.

Quadro 1 - Aplicações práticas do modelo ULMFiT

ULMFiT
Tarefa: Classificação de Texto
• Aplicação: Imagine que você tem um grande número de avaliações de produtos online e deseja determinar se cada avaliação é positiva ou negativa. • Como Utilizar: ULMFiT pode ser usado para esta tarefa começando com um modelo pré-treinado em um grande corpus de texto geral, como a Wikipedia. Em seguida, você ajusta o modelo usando um conjunto de dados específico de avaliações de produtos. Após esse ajuste fino, o modelo se torna capaz de classificar novas avaliações como positivas ou negativas com alta precisão. • Benefícios: A abordagem ULMFiT é vantajosa porque permite ajustar o modelo para o domínio específico das avaliações de produtos, resultando em um desempenho superior, mesmo com uma quantidade limitada de dados específicos.
Tarefa: Análise de Sentimentos
• Aplicação: Em um contexto de análise de mídias sociais, como Twitter, você pode querer entender o sentimento geral sobre uma nova campanha de marketing. • Como Utilizar: Você pode usar ULMFiT para treinar um modelo que classifica tweets em categorias de sentimento, como positivo, negativo ou neutro. Isso é feito ajustando o modelo pré-treinado com tweets rotulados para sentimentos específicos. • Benefícios: ULMFiT permite adaptar o modelo para captar nuances específicas da linguagem usada nas mídias sociais, oferecendo percepções mais precisas sobre o sentimento público.

Fonte: autoria própria.

Quadro 2 - Aplicações práticas do modelo ELMo

ELMo
Tarefa: Reconhecimento de Entidades Nomeadas (NER)
• Aplicação: Suponha que você esteja trabalhando com documentos legais e precise identificar e classificar automaticamente nomes de pessoas, organizações e locais mencionados nos textos. • Como Utilizar: ELMo pode ser utilizado para esta tarefa fornecendo *embeddings* contextuais para cada palavra no texto. Esses *embeddings* são então usados por um modelo de NER para identificar e classificar as entidades nomeadas. • Benefícios: ELMo captura o contexto de cada palavra de forma dinâmica, melhorando a precisão na identificação de entidades, especialmente em textos complexos como documentos legais.

Fonte: autoria própria.

SAIBA MAIS...

- Let's learn about Universal Language Model Fine-tuning, ULMFiT (Ha, 2022).
- Guide to Learn ELMo for Extracting Features from Text (Joshi, 2024).
- Language Modeling II: ULMFiT and ELMo (Deshpande, 2020).

Arquiteturas Profundas para Processamento de Linguagem Natural

Unidade IV

BERT com Hugging Face

Unidade IV: BERT com *Hugging Face*

Lançado em 2018 pela *Google AI Research*®, o BERT marcou uma virada significativa na evolução dos modelos de PLN. Antes do BERT, muitos modelos de linguagem eram limitados a contextos unidirecionais, o que restringia sua capacidade de compreender o significado das palavras de maneira mais completa.

4.1 Introdução

Introduzido em 2018, o BERT transformou a forma como os modelos de linguagem compreendem o contexto das palavras, utilizando uma arquitetura bidirecional baseada em *Transformers*. Ao permitir que cada palavra em uma frase seja analisada em relação ao seu contexto completo, o BERT trouxe avanços significativos em diversas aplicações de PLN, como classificação de texto, tradução e resposta a perguntas, estabelecendo novos padrões de desempenho e inspirando uma série de variantes e melhorias subsequentes.

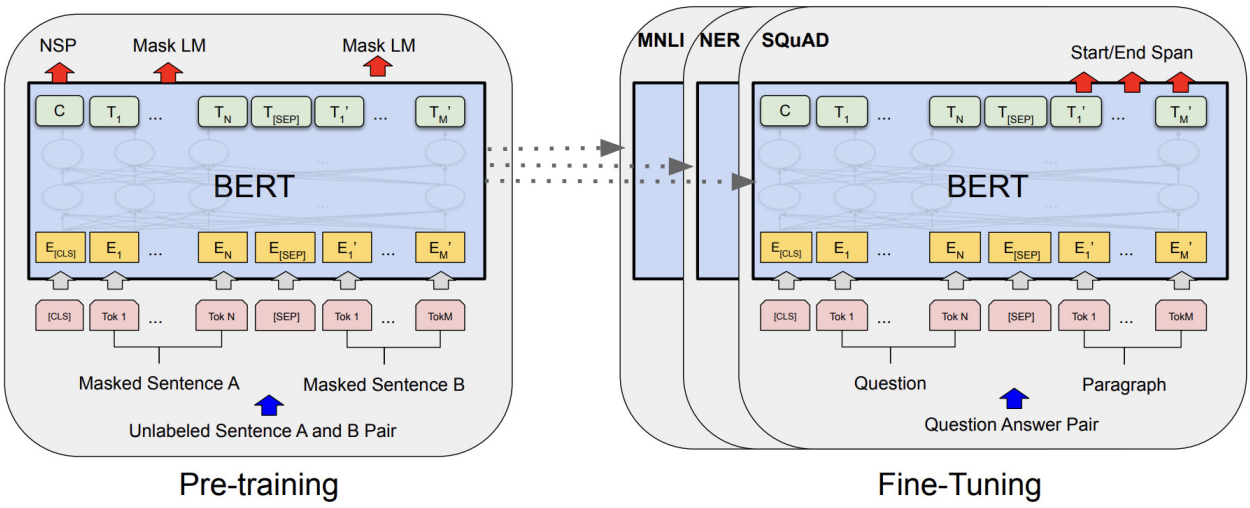
BERT introduziu a inovação da atenção bidirecional, permitindo que o modelo analisasse o texto a partir de ambos os lados de uma palavra simultaneamente. Esse avanço foi possível graças à arquitetura *Transformer*, originalmente proposta por Vaswani *et al*. (2017), que revolucionou o campo ao permitir um processamento mais eficaz e contextualizado das sequências de texto. A introdução do BERT não apenas melhorou o desempenho em uma ampla gama de tarefas de PLN, mas também estabeleceu um novo padrão de referência, inspirando o desenvolvimento de várias outras arquiteturas e variantes que ampliaram ainda mais as capacidades dos modelos de linguagem.

Desde seu lançamento, BERT estabeleceu novos padrões de desempenho em vários *benchmarks* de PLN, como o *General Language Understanding Evaluation* (GLUE) e o *Stanford Question Answering Dataset* (SQuAD). Seu impacto foi tão significativo que inspirou uma série de variantes e melhorias, incluindo modelos, que serão comentados mais adiante, como: RoBERTa, ALBERT e DistilBERT, que adaptam e expandem as capacidades do modelo original.

4.2 Arquitetura do BERT

A principal inovação do BERT reside em sua arquitetura bidirecional baseada em *Transformers*. Antes do BERT, muitos modelos de PLN, como os baseados em LSTMs e GRUs, processavam o texto de maneira unidirecional ou com capacidades bidirecionais limitadas. O BERT, no entanto, utiliza a atenção bidirecional plena, permitindo que cada palavra em uma frase seja considerada em relação ao contexto completo, melhorando significativamente a compreensão semântica. Na Figura 36, a seguir, está ilustrada a arquitetura de pré-treino e *fine-tuning* do BERT. Nas subseções seguintes (4.2.1 e 4.2.2), será comentado sobre seus componentes.

Figura 38 - Pré-treino e *fine-tuning* para BERT



Fonte: adaptada de Devlin *et al.* (2019).

4.2.1 Pré-treinamento

O BERT foi pré-treinado em um grande *corpus* de texto que inclui o *BookCorpus* (800 milhões de palavras) e a *Wikipedia* em inglês (2,5 bilhões de palavras). O processo de pré-treino envolve duas tarefas principais: *Masked Language Modeling* (MLM ou Mask LM) e *Next Sentence Prediction* (NSP). No MLM, algumas palavras da frase são mascaradas (temporariamente escondidas do modelo) e o modelo deve prever essas palavras com base no contexto bidirecional. No NSP, o modelo aprende a entender a relação entre pares de sentenças, determinando se uma frase segue a outra.

4.2.1.1 *Masked Language Modeling* (MLM)

O objetivo do MLM é treinar o modelo para prever palavras mascaradas em uma frase, o que permite ao BERT aprender representações contextuais profundas. Ao aprender a prever palavras mascaradas, o BERT captura o significado de uma palavra considerando todo o seu contexto (tanto à esquerda quanto à direita), o que melhora significativamente a compreensão semântica. Método do MLM:

1. Em cada sequência de entrada, 15% das palavras são selecionadas aleatoriamente para serem mascaradas.
2. Das palavras selecionadas:
    a. 80% são substituídas pelo *token* [MASK] .
    b. 10% são substituídas por uma palavra aleatória.
    c. 10% permanecem inalteradas (isso ajuda o modelo a manter a capacidade de reconhecer palavras corretamente).
3. O modelo é então treinado para prever as palavras originais que foram mascaradas, baseando-se no contexto fornecido pelas palavras não mascaradas ao redor.

4.2.1.2 *Next Sentence Prediction* (NSP)

O objetivo do NSP é treinar o modelo para entender a relação entre pares de sentenças, o que é crucial para tarefas como resposta a perguntas e classificação de parágrafos. Ao aprender a prever se duas sentenças são sequenciais, o BERT desenvolve uma compreensão melhor da coerência entre sentenças, melhorando seu desempenho em tarefas que dependem de múltiplas sentenças. Método do NSP:

1. O modelo recebe pares de sentenças como entrada.
2. Em 50% dos casos, a segunda sentença realmente segue a primeira no *corpus* original (par verdadeiro).
3. Nos outros 50% dos casos, a segunda sentença é uma sentença aleatória do *corpus* (par falso).
4. O modelo é treinado para prever se a segunda sentença é uma continuação da primeira (par verdadeiro) ou não (par falso).

4.2.2 *Fine-tuning* do BERT

O objetivo do *fine-tuning* é ajustar o modelo BERT pré-treinado para uma tarefa específica de PLN, como classificação de texto, resposta a perguntas, análise de sentimentos, entre outras. Isso é feito treinando o modelo com um conjunto de dados rotulados específico para a tarefa desejada.

Figura 39 - Procedimento para *fine-tuning* do BERT

Preparação dos Dados:

a) Entrada:
- As entradas para o modelo durante o *fine-tuning* são formatadas de maneira semelhante ao pré-treino. Cada entrada é uma sequência de tokens, incluindo tokens especiais [CLS] e [SEP].

b) Tokens Especiais:
- [CLS]: Este token é adicionado no início de cada sequência e sua representação final é usada para tarefas de classificação.
- [SEP]: Este token é usado para separar duas sentenças em tarefas que envolvem pares de sentenças.

Estrutura do Modelo:

a) Camadas de Tarefa Específica:
- Durante o *fine-tuning*, uma ou mais camadas adicionais são adicionadas ao topo do modelo BERT para a tarefa específica. Por exemplo, uma camada de classificação softmax é adicionada para tarefas de classificação de texto.

Processo de Treinamento:

a) Inicialização:
- Os pesos do modelo BERT são inicializados com os valores do modelo pré-treinado.

b) Treinamento:
- O modelo é treinado em um conjunto de dados rotulado específico para a tarefa. Durante o treinamento, os pesos de todas as camadas do BERT, bem como das camadas específicas da tarefa, são ajustados.

c) Função de Perda:
- A função de perda depende da tarefa específica. Por exemplo, para classificação de texto, é usada a entropia cruzada.

Ajuste dos Hiperparâmetros:

a) Taxa de Aprendizado:
- A taxa de aprendizado é geralmente menor durante o *fine-tuning* do que durante o pré-treino para evitar grandes atualizações nos pesos.

b) *Batch Size*:
- O tamanho do lote pode variar dependendo da tarefa e dos recursos computacionais disponíveis.

c) Número de Épocas:
- O número de épocas de treinamento é ajustado com base no desempenho do modelo no conjunto de validação.

O processo de *fine-tuning* do BERT é crucial para adaptar o modelo pré-treinado a tarefas específicas de PLN. Ele envolve a preparação dos dados de entrada, a adição de camadas de tarefa específica, o treinamento do modelo com ajuste dos pesos e a otimização dos hiperparâmetros. Esse processo permite que o BERT alcance desempenho competitivo em diversas aplicações de PLN. O procedimento para o treinamento pode ser realizado conforme os passos a seguir (Figura 37).

4.3 Hugging Face Transformers

A biblioteca *Hugging Face Transformers* é uma das ferramentas mais populares e poderosas no campo do PLN. Ela foi desenvolvida pela *Hugging Face* (plataforma *open-source* que facilita o uso de modelos de NLP, como *Transformers*) e fornece implementações de última geração para diversos modelos de linguagem baseados em *Transformers*, como BERT, GPT-2, RoBERTa, T5, entre muitos outros. É versátil e facilita a implementação, seus benefícios para o uso serão comentados abaixo e incluem: o acesso a modelos de última geração, facilidade de uso, flexibilidade, uma grande comunidade de suporte, e integração com outras ferramentas populares de *machine learning*. Suas utilizações abrangem uma ampla gama de tarefas de PLN, incluindo classificação de texto, geração de texto, tradução, resposta a perguntas, sumarização e NER, tornando-a uma escolha essencial para desenvolvedores e pesquisadores no campo de PLN.

Estão listados, a seguir, alguns benefícios do uso dessa biblioteca:

- **Acesso a modelos de última geração:** oferece uma vasta coleção de modelos pré-treinados que podem ser facilmente utilizados para diversas tarefas de PLN. Isso inclui modelos como BERT, GPT-2, T5, DistilBERT, entre outros, que são considerados estado da arte em diversas aplicações de PLN.
- **Facilidade de uso:** a *Hugging Face Transformers* é conhecida por sua simplicidade e facilidade de uso. Mesmo sem um conhecimento profundo de *machine learning*, os desenvolvedores podem rapidamente implementar e utilizar modelos de PLN. Ela fornece interfaces de alto nível que tornam o processo de integração de modelos em projetos muito mais simples.
- **Flexibilidade:** é altamente flexível e pode ser usada para uma ampla gama de tarefas de PLN, incluindo classificação de texto, tradução, geração de texto, resposta a perguntas, sumarização e muitas outras. Ela permite o *fine-tuning* de modelos para tarefas específicas, adaptando-os às necessidades particulares dos projetos.
- **Comunidade e suporte:** a *Hugging Face* possui uma grande comunidade de usuários e desenvolvedores que contribuem com tutoriais, exemplos de código, e soluções para problemas comuns. Além disso, a documentação

da biblioteca é extensa e bem estruturada, facilitando a aprendizagem e a implementação dos modelos.

- **Integração com outras ferramentas:** a biblioteca integra-se bem com outras ferramentas e *frameworks* populares de *machine learning*, como *PyTorch* e *TensorFlow*. Isso permite aos desenvolvedores aproveitar o melhor dos dois mundos, combinando a flexibilidade e a capacidade de personalização do *PyTorch* com a eficiência e a escalabilidade do *TensorFlow*.
- **Classificação de texto:** pode ser usada para tarefas de classificação de texto, como análise de sentimentos, classificação de tópicos e identificação de spam. Modelos pré-treinados podem ser ajustados *(fine-tuning)* com conjuntos de dados rotulados específicos para melhorar a precisão da classificação.
- **Geração de texto:** a *Hugging Face Transformers* é amplamente utilizada para geração de texto, como na criação de assistentes virtuais, *chatbots* e sistemas de resposta automática. Modelos como GPT-2 e GPT-3 são particularmente eficazes nessa tarefa, produzindo texto coerente e contextualmente relevante.
- **Tradução automática:** suporta modelos de tradução que podem ser usados para traduzir texto entre diferentes idiomas. Modelos como MarianMT e T5 podem ser utilizados para criar sistemas de tradução automática com alta qualidade.
- **Resposta a perguntas:** utilizando modelos como BERT e RoBERTa, a biblioteca permite a criação de sistemas de resposta a perguntas que podem ler um documento e responder perguntas específicas sobre seu conteúdo. Isso é particularmente útil em assistentes virtuais, mecanismos de busca e sistemas de suporte ao cliente.
- **Sumarização de texto:** modelos como BERT e T5 são utilizados para criar resumos automáticos de textos longos. Isso é útil em contextos como análise de notícias, resumos de documentos e sínteses de relatórios extensos.
- **NER:** a biblioteca permite a identificação e classificação automática de entidades nomeadas (como nomes de pessoas, organizações e locais) em textos. Isso é valioso em aplicações de mineração de texto, análise de documentos e sistemas de busca.

4.4 Modelos Derivados do BERT

Desde a introdução do BERT em 2018, o campo do PLN tem experimentado avanços significativos. O BERT estabeleceu novos padrões de desempenho em diversas tarefas de PLN, inspirando a criação de várias variantes e modelos derivados. Cada um desses modelos foi desenvolvido para atender a necessidades específicas, seja para melhorar a eficiência, reduzir a complexidade ou adaptar-se a domínios específicos.

Os modelos derivados do BERT trazem inovações e ajustes que otimizam seu desempenho em diferentes contextos. Alguns modelos focam na compactação e eficiência, como o DistilBERT e o ALBERT, enquanto outros, como o RoBERTa, aprimoram o treinamento para alcançar maior precisão. Além disso, existem versões especializadas para idiomas e domínios específicos, como o BERTimbau para o português brasileiro e o BioBERT para textos biomédicos.

A seguir, apresentamos um quadro comparativo (Tabela 4), destacando as principais características dos modelos derivados do BERT, tamanho, velocidade, precisão e as tarefas recomendadas para cada um, proporcionando uma visão clara das adaptações e melhorias implementadas em cada variante.

Tabela 4 - Principais características dos modelos derivados do BERT

MODELO	DESCRIÇÃO	TAMANHO DO MODELO	VELOCIDADE	PRECISÃO	TAREFAS
BERT	Modelo original com atenção bidirecional, pré-treinado em uma grande quantidade de texto.	Base: 110M parâmetros, Large: 340M parâmetros	Médio	Alta	Classificação de texto, NER, resposta a perguntas
DistilBERT	Versão compacta e otimizada do BERT, 60% menor e 60% mais rápido, com 97% da precisão do BERT.	66M parâmetros	Alta	Alta	Tarefas que requerem baixa latência e menos recursos
RoBERTa	Modelo robusto otimizado do BERT, treinado com mais dados e técnicas de regularização aprimoradas.	Base: 125M parâmetros, Large: 355M parâmetros	Médio	Muito alta	Classificação de texto, NER, análise de sentimentos
BERTimbau	Versão do BERT adaptada para o português brasileiro, treinada em um grande corpus de texto em português.	Base: 110M parâmetros	Médio	Alta	Tarefas de PNL específicas para o português
ALBERT	Versão leve do BERT com eficiência de memória aprimorada e arquitetura modificada, utilizando fatorização de parâmetros e repetição de camadas.	Base: 12M parâmetros, Large: 18M parâmetros	Muito alta	Alta	Tarefas de PNL com limitações de memória
XLNet	Modelo que combina as vantagens do BERT e Transformer-XL, utilizando uma abordagem autoregressiva permutacional para pré-treino.	Base: 110M parâmetros, Large: 340M parâmetros	Baixa	Muito alta	Modelagem de linguagem, classificação de texto, NER
ERNIE	Modelo desenvolvido pela Baidu, treinado para capturar conhecimento semântico mais profundo, utilizando tarefas de pré-treino específicas para incorporar conhecimento externo.	Base: 110M parâmetros	Médio	Alta	Análise de sentimentos, resposta a perguntas, NER
SpanBERT	Variante do BERT que se foca em prever spans de texto em vez de palavras individuais, melhorando o desempenho em tarefas de compreensão de leitura e NER.	Base: 110M parâmetros	Médio	Alta	Compreensão de leitura, NER, resposta a perguntas
BioBERT	Versão do BERT especializada em textos biomédicos, treinada em corpora de literatura biomédica.	Base: 110M parâmetros	Médio	Alta	Análise de texto biomédico, NER biomédico, QA biomédico
SciBERT	Modelo adaptado para textos científicos, treinado em artigos científicos de diversas áreas.	Base: 110M parâmetros	Médio	Alta	Análise de texto científico, NER científico, QA científico

Fonte: autoria própria.

4.5 Exemplos Práticos

Para uma compreensão prática e detalhada das aplicações de PLN, recomendamos acessar o notebookque desenvolvemos. Este *notebook* demonstra passo a passo como realizar o *fine-tuning* do modelo BERT para análise de sentimentos em português, além de exemplos práticos de NER e tradução automática utilizando modelos pré-treinados da biblioteca *Hugging Face Transformers*.

Notebook Colab

Processamento de Linguagem Natural utilizando a biblioteca Hugging Face Transformers

Objetivos de Aprendizagem

Neste *Notebook*, vamos explorar exemplos reais de soluções para tarefas de Processamento de Linguagem Natural (NLP) utilizando a biblioteca Hugging Face Transformers. Nosso primeiro exemplo será o *fine-tuning* do modelo BERT para a análise de sentimentos em português. Em seguida, apresentaremos dois exemplos utilizando apenas modelos pré-treinados com os *pipelines* da Hugging Face. O segundo exemplo demonstrará o uso de *Named Entity Recognition* (NER), e o terceiro exemplo abordará a tradução automática. Estes exemplos práticos fornecerão uma visão abrangente de como aplicar técnicas avançadas de NLP para resolver problemas do mundo real.

2. Fine-Tuning do BERT para Análise de Sentimentos

Neste tópico, vamos realizar o *fine-tuning* do modelo BERT para a tarefa de análise de sentimentos utilizando a biblioteca Hugging Face Transformers. Vamos usar um *dataset* de comentários sobre compras em português.

Importante: Para o treinamento deve ser usada a rumtime com **GPU**.

2.1 Instalação das Dependências

Vamos começar instalando as bibliotecas necessárias.

```
[ ] !pip install transformers
    !pip install datasets
    !pip install torch
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.10/
dist-packages (4.44.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/
dist-packages (from transformers) (3.15.4)
Requirement already satisfied: huggingface-hub<1.0,>=0.23.2 in /usr/lo-
cal/lib/python3.10/dist-packages (from transformers) (0.24.6)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/
dist-packages (from transformers) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (24.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/
dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (2024.5.15)
Requirement already satisfied: requests in /usr/local/lib/python3.10/
dist-packages (from transformers) (2.32.3)
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (0.4.4)
Requirement already satisfied: tokenizers<0.20,>=0.19 in /usr/local/lib/
python3.10/dist-packages (from transformers) (0.19.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/
dist-packages (from transformers) (4.66.5)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/py-
thon3.10/dist-packages (from huggingface-hub<1.0,>=0.23.2->transformers)
(2024.6.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/
lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.23.2->trans-
formers) (4.12.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/
lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/
dist-packages (from requests->transformers) (3.8)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py-
thon3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py-
thon3.10/dist-packages (from requests->transformers) (2024.8.30)
Collecting datasets
  Downloading datasets-2.21.0-py3-none-any.whl.metadata (21 kB)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/
dist-packages (from datasets) (3.15.4)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/
dist-packages (from datasets) (1.26.4)
```

continua

```
Collecting pyarrow>=15.0.0 (from datasets)
  Downloading pyarrow-17.0.0-cp310-cp310-manylinux_2_28_x86_64.whl.meta-
data (3.3 kB)
Collecting dill<0.3.9,>=0.3.0 (from datasets)
  Downloading dill-0.3.8-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/
dist-packages (from datasets) (2.1.4)
Requirement already satisfied: requests>=2.32.2 in /usr/local/lib/py-
thon3.10/dist-packages (from datasets) (2.32.3)
Requirement already satisfied: tqdm>=4.66.3 in /usr/local/lib/python3.10/
dist-packages (from datasets) (4.66.5)
Collecting xxhash (from datasets)
  Downloading xxhash-3.5.0-cp310-cp310-manylinux_2_17_x86_64.manyli-
nux2014_x86_64.whl.metadata (12 kB)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.16-py310-none-any.whl.metadata (7.2 kB)
Requirement already satisfied: fsspec<=2024.6.1,>=2023.1.0 in /usr/local/
lib/python3.10/dist-packages (from fsspec[http]<=2024.6.1,>=2023.1.0->d
atasets) (2024.6.1)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/
dist-packages (from datasets) (3.10.5)
Requirement already satisfied: huggingface-hub>=0.21.2 in /usr/local/lib/
python3.10/dist-packages (from datasets) (0.24.6)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/
dist-packages (from datasets) (24.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/
dist-packages (from datasets) (6.0.2)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/
python3.10/dist-packages (from aiohttp->datasets) (2.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (6.0.5)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/py-
thon3.10/dist-packages (from aiohttp->datasets) (1.9.4)
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/
python3.10/dist-packages (from aiohttp->datasets) (4.0.3)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/
lib/python3.10/dist-packages (from huggingface-hub>=0.21.2->datasets)
(4.12.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/
lib/python3.10/dist-packages (from requests>=2.32.2->datasets) (3.3.2)
```

continua

```
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/
dist-packages (from requests>=2.32.2->datasets) (3.8)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py-
thon3.10/dist-packages (from requests>=2.32.2->datasets) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py-
thon3.10/dist-packages (from requests>=2.32.2->datasets) (2024.8.30)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/
python3.10/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/
dist-packages (from pandas->datasets) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/py-
thon3.10/dist-packages (from pandas->datasets) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/
dist-packages (from python-dateutil>=2.8.2->pandas->datasets) (1.16.0)
Downloading datasets-2.21.0-py3-none-any.whl (527 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 527.3/527.3 kB 7.1 MB/s eta
0:00:00
Downloading dill-0.3.8-py3-none-any.whl (116 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 5.7 MB/s eta
0:00:00
Downloading pyarrow-17.0.0-cp310-cp310-manylinux_2_28_x86_64.whl (39.9
MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 39.9/39.9 MB 10.8 MB/s eta
0:00:00
Downloading multiprocess-0.70.16-py310-none-any.whl (134 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 6.3 MB/s eta
0:00:00
Downloading xxhash-3.5.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
x86_64.whl (194 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.1/194.1 kB 8.6 MB/s eta
0:00:00
Installing collected packages: xxhash, pyarrow, dill, multiprocess, data-
sets
  Attempting uninstall: pyarrow
    Found existing installation: pyarrow 14.0.2
    Uninstalling pyarrow-14.0.2:
      Successfully uninstalled pyarrow-14.0.2
ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the fol-
lowing dependency conflicts.
cudf-cu12 24.4.1 requires pyarrow<15.0.0a0,>=14.0.1, but you have pyarrow
17.0.0 which is incompatible.
ibis-framework 8.0.0 requires pyarrow<16,>=2, but you have pyarrow 17.0.0
which is incompatible.
Successfully installed datasets-2.21.0 dill-0.3.8 multiprocess-0.70.16
pyarrow-17.0.0 xxhash-3.5.0
```

continua

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/
dist-packages (2.4.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/
dist-packages (from torch) (3.15.4)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/
lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/
dist-packages (from torch) (1.13.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/
dist-packages (from torch) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/
dist-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/
dist-packages (from torch) (2024.6.1)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/py-
thon3.10/dist-packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/py-
thon3.10/dist-packages (from sympy->torch) (1.3.0)
```

2.2 Configuração de variáveis globais

Vamos configurar as variáveis de parâmetros para serem utilizadas durante o código.

```
[ ] # model_id = "neuralmind/bert-base-portuguese-cased"
    model_id = "adalbertojunior/distilbert-portuguese-cased"
    max_length= 512
    num_labels = 3
    batch_size = 28
    results_path = "./results"
    pretrained_path = "./sentiment-analysis-bert-portuguese"
```

2.3 Importação das Bibliotecas

Vamos importar as bibliotecas necessárias para carregar o *dataset*, tokenizar os textos, configurar o modelo BERT, e realizar o treinamento e avaliação.

```
[ ] from transformers import BertTokenizer, BertForSequenceClassification,
    Trainer, TrainingArguments, DataCollatorWithPadding
```

continua

```
from datasets import load_dataset, load_metric, DatasetDict
from torch.utils.data import DataLoader, SequentialSampler
import torch
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
classification_report
```

2.4 Carregar o Dataset

Vamos carregar um dataset de comentários de pedidos em português. Para este exemplo, vamos utilizar o *dataset* ***verissimomanoel/olist_customers_review*** da Hugging Face, que contém comentários rotulados para análise de sentimentos. No dataset já tem uma parte de treino ***train*** e outra de para teste ***test***. Contudo precisamos dividir o treino mais uma vez para ter uma parte para validação ***val*** que será usada durante o treino.

```
[ ] # Carrega o dataset que tem train e test
    dataset = load_dataset("verissimomanoel/olist_customers_review", trust_
    remote_code=True)

    # Divide o treino em 80% e 20%, sendo os 80 para treino e os 20 para val-
    idação
    ds_train_split = dataset["train"].train_test_split(test_size=0.2)

    # Monta o dataset com todas as partes train, test e val
    dataset = DatasetDict({
        "train": ds_train_split["train"],
        "test": dataset["test"],
        "val": ds_train_split["test"],
    })

    # Separa as partes do dataset
    train_dataset = dataset['train']
    test_dataset = dataset['test']
    val_dataset = dataset['val']
```

continua

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.
py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your set-
tings tab (https://huggingface.co/settings/tokens), set it as secret in
your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to ac-
cess public models or datasets.
  warnings.warn(
Downloading readme: 100% 579/579 [00:00<00:00, 4.80kB/s]
Downloading data: 100% 1.56M/1.56M [00:00<00:00, 3.16MB/s]
Downloading data: 100% 397k/397k [00:00<00:00, 826kB/s]
Generating train split: 100%33395/33395 [00:00<00:00, 499942.11 examples/s]
Generating test split: 100%  8349/8349 [00:00<00:00, 206939.16 examples/s]
```

2.5 Visualização do *Dataset*

Vamos visualizar a estrutura do dataset carregado. Para tal, as sequências de código [6] e [7] abaixo mostrarão, na forma de gráfico, os conjuntos de dados para treino, teste e validação.

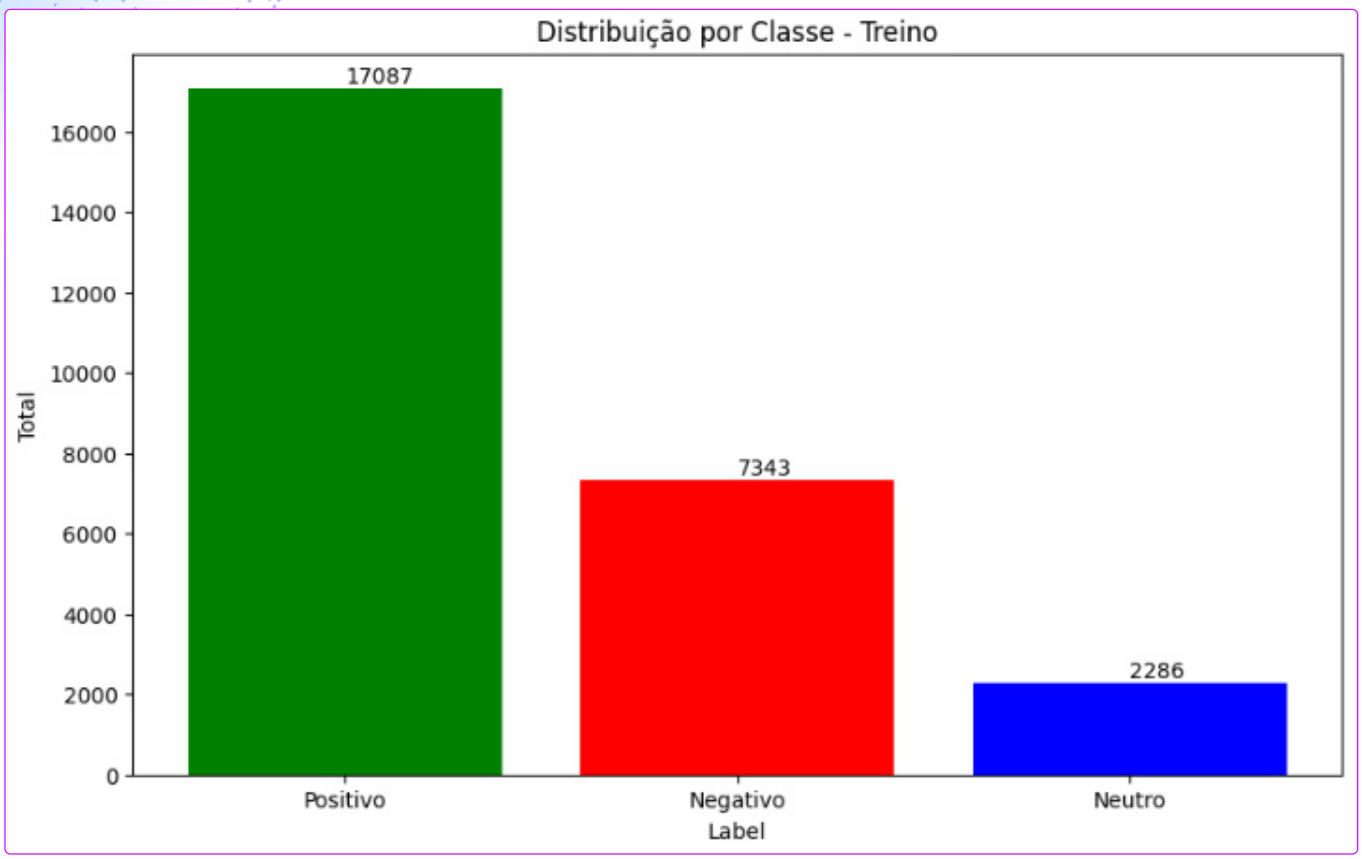
```
[ ] print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 26716
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 8349
    })
    val: Dataset({
        features: ['text', 'label'],
        num_rows: 6679
    })
})
```

```
[ ] def show_info_dataset(dataset, title):
        # Converter o dataset para um DataFrame do pandas
        df = dataset.to_pandas()

        # Contar as ocorrências na coluna 'label'
        label_counts = df['label'].value_counts()

        # Mapeamento dos labels para nomes
        label_names = {0: 'Negativo', 1: 'Positivo', 2: 'Neutro'}

        # Obter os nomes das labels
        labels = [label_names[label] for label in label_counts.index]

        # Definir as cores para cada label
        colors = ['green', 'red', 'blue']

        # Plotar o gráfico de barras
        plt.figure(figsize=(10, 6))
        bars = plt.bar(labels, label_counts, color=colors)

        # Adicionar os totais em cima das barras
        for bar in bars:
            yval = bar.get_height()
            plt.text(bar.get_x() + bar.get_width()/2, yval, int(yval),
    va='bottom')  # va: vertical alignment

        # Configurar o título e os rótulos dos eixos
        plt.title(title)
        plt.xlabel('Label')
        plt.ylabel('Total')
        plt.show()
```

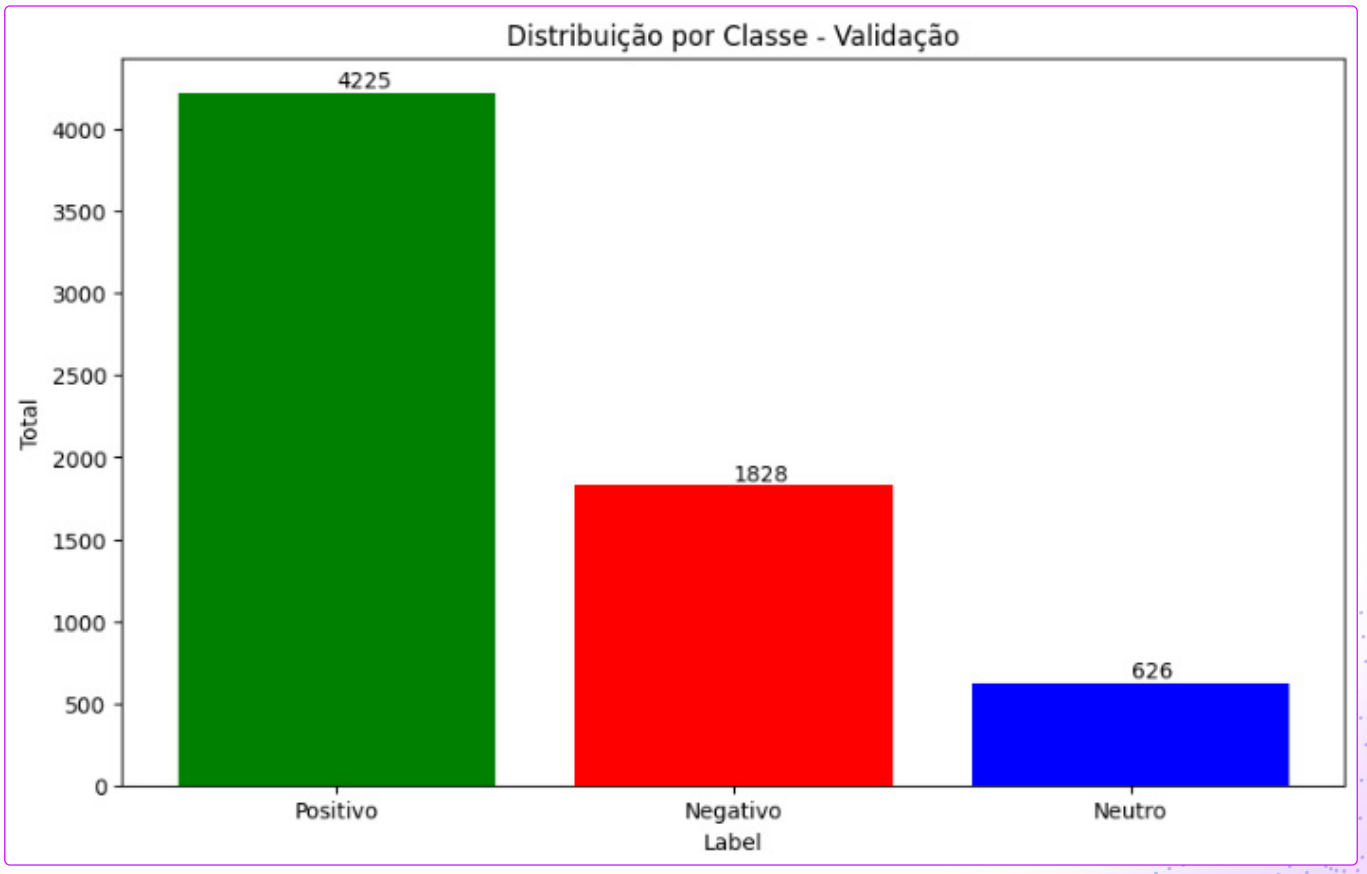
```
[ ] show_info_dataset(train_dataset, 'Distribuição por Classe - Treino')
    show_info_dataset(val_dataset, 'Distribuição por Classe - Validação')
    show_info_dataset(test_dataset, 'Distribuição por Classe - Teste')
```

Figura 40- Distribuição por Classe - Treino



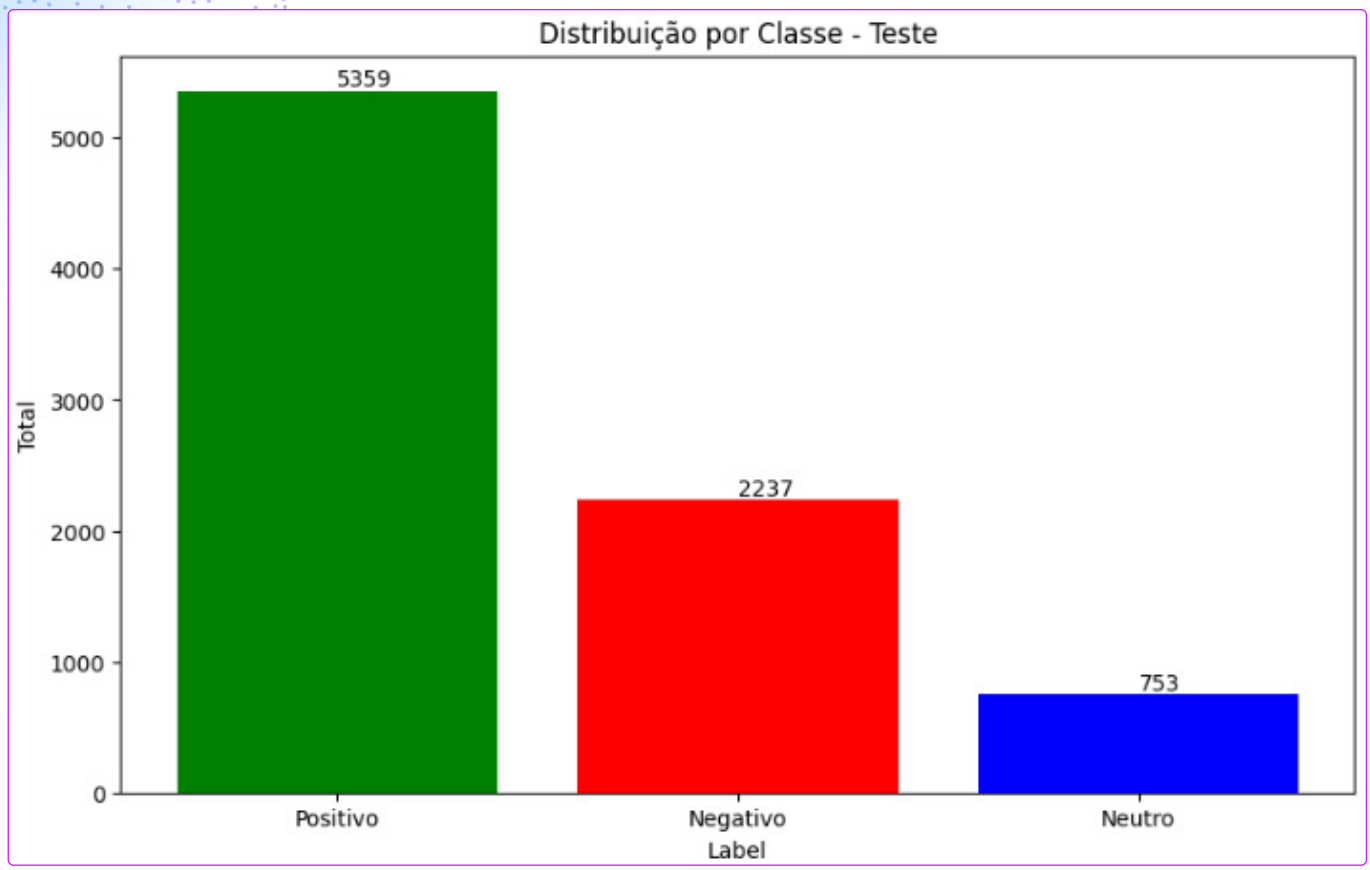
Fonte: autoria própria.

Figura 41 - Distribuição por Classe - Validação



Fonte: autoria própria.

Figura 42 - Distribuição por Classe - Teste



Fonte: autoria própria.

2.6 Tokenização do Dataset

Vamos carregar o tokenizer do BERT e usá-lo para tokenizar os textos no *dataset*. Vamos usar o modelo ***neuralmind/bert-base-portuguese-cased***, que é um BERT treinado em português.

O comando train_dataset.shuffle().select(range(5000)) é responsaável pelo embaralhamento e seleção de amostras. Neste caso está selecionando as primeiras 5000 amostras do conjunto de dados embaralhado.

O comando train_dataset.map(tokenize_function, batched=True) realiza a tokenização no conjunto de dados de treinamento (train_dataset) e usa a função map() para aplicar a função de tokenização a cada lote de amostras do conjunto de dados. Com batched=True, a função é aplicada em lotes de amostrase não em uma amostra por vez, isso torna o processo mais eficiente.

```
[ ] tokenizer = BertTokenizer.from_pretrained(model_id)
    data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

    def tokenize_function(examples):
      return tokenizer(examples['text'], padding='max_length', truncation=True,
    max_length=max_length)
```

continua

```
# Reduz o tamanho dos datasets apenas para conseguir rodar no Google Colab por menos tempo, para rodar com o dataset completo só comentar as próximas 3 linhas
train_dataset = train_dataset.shuffle().select(range(5000))
test_dataset = test_dataset.shuffle().select(range(1000))
val_dataset = test_dataset.shuffle().select(range(800))

train_dataset = train_dataset.map(tokenize_function, batched=True)
test_dataset = test_dataset.map(tokenize_function, batched=True)
val_dataset = val_dataset.map(tokenize_function, batched=True)
```

2.7 Formatando o Dataset

Vamos definir o formato dos datasets para que o *Trainer* da Hugging Face possa processá-los corretamente.

- **set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])**: Este método converte os datasets para o formato do PyTorch (tensors) para que possam ser usados diretamente em redes neurais. A função especifica que apenas as colunas input_ids, attention_mask e label serão mantidas no formato final.
- input_ids: Contém os identificadores numéricos que representam as palavras ou tokens da entrada.
- attention_mask: Indica quais tokens são relevantes (1) e quais são padding (0), para que o modelo saiba onde prestar atenção.
- label: São as etiquetas associadas a cada exemplo, que o modelo deve prever (por exemplo, para uma tarefa de classificação).

```
[ ] train_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
    test_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
    val_dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
```

2.8 Configuração do Modelo BERT

Vamos configurar o modelo BERT para a tarefa de classificação de sequência. Neste caso, estamos utilizando a versão **neuralmind/bert-base-portuguese-cased** do modelo.

```
[ ] model = BertForSequenceClassification.from_pretrained(model_id, num_
    labels=num_labels)
```

2.9 Função de Avaliação

Vamos definir uma função de avaliação para calcular a precisão do modelo durante a avaliação. Utilizaremos a métrica de precisão (**accuracy**) fornecida pela biblioteca **datasets**.

```
[ ] metric = load_metric("accuracy", trust_remote_code=True)

    def compute_metrics(eval_pred):
        logits, labels = eval_pred
        # Transforma os logits (saída do modelo) em previsões de classe,
    atribuindo a cada exemplo a classe com a
        # maior probabilidade, ou seja, a que tem o maior valor de logit.
        predictions = torch.argmax(torch.tensor(logits), dim=-1)

        return metric.compute(predictions=predictions, references=torch.
    tensor(labels))
```

2.10 Configuração dos Argumentos de Treinamento

Vamos definir os parâmetros de treinamento, incluindo a taxa de aprendizado, tamanho do batch, número de épocas, e a estratégia de avaliação.

```
[ ] training_args = TrainingArguments(
    output_dir=results_path,              # Diretório de saída para os resultados
  evaluation_strategy="epoch",      # Estratégia de avaliação (avaliar a cada época)
      learning_rate=3e-5,                                  # Taxa de aprendizado
      per_device_train_batch_size=batch_size,     # Tamanho do batch de treino
     per_device_eval_batch_size=batch_size,     # Tamanho do batch de avaliação
     num_train_epochs=3,                        # Número de épocas de treinamento
      weight_decay=0.01,                                   # Decaimento de peso
    )
```

2.11 Treinamento do Modelo

Vamos criar um objeto ***Trainer*** com o modelo, dados de treino e validação, e os argumentos de treinamento definidos nos comandos acima. Em seguida, vamos iniciar o treinamento do modelo.

```
[ ] trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        data_collator=data_collator,
        compute_metrics=compute_metrics,
    )

    trainer.train()
```

2.12 Avaliação do Modelo - *Evaluate*

Vamos avaliar o modelo no conjunto de teste (com o comando trainer.evaluate) e exibir a precisão com a função show_info logo abaixo.

```
[ ] results = trainer.evaluate(eval_dataset=test_dataset)
    print(f"Acurácia no conjunto de teste: {results['eval_accuracy']}")
```

2.13 Avaliação do Modelo - *Predict*

Vamos avaliar o modelo no conjunto de teste e exibir a matriz de confusão.

```
[ ] raw_pred, _, _ = trainer.predict(test_dataset=test_dataset)
```

```
[ ] def show_info(y_true, y_pred, title='Confusion matrix', cmap='Blues'):
        target_names = ['Negativo', 'Positivo', 'Neutro']
        print(classification_report(y_true, y_pred, target_names=target_
    names))

        plt.figure(figsize=(16, 10))
        cm = confusion_matrix(y_true, y_pred)
```

continua

```
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_la-
bels=target_names)
    fig, ax = plt.subplots(figsize=(10,10))
    disp.plot(ax=ax, xticks_rotation='vertical', cmap=plt.cm.Blues,val-
ues_format='g')
    plt.show()
```

```
[ ] # Pré-processar previsões brutas
    y_pred = np.argmax(raw_pred, axis=1)
    y_true = test_dataset["label"]

    show_info(y_true, y_pred)
```

2.14 Salvar o Modelo

Vamos salvar o modelo treinado e o tokenizer para uso futuro.

```
[ ] model.save_pretrained(pretrained_path)
    tokenizer.save_pretrained(pretrained_path)
```

2.15 Predição de Novos Exemplos

Vamos definir uma função para prever o sentimento de novas frases usando o modelo treinado. Em seguida, vamos testar o modelo com uma nova frase e exibir o resultado.

```
[ ] def predict_sentiment(text):
        # Certificar que o modelo e os inputs estão no mesmo dispositivo (CPU
    ou GPU)
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model.to(device)

        # Tokeniza o texto, aplica padding e truncamento, converte para ten-
    sor PyTorch e move os dados para o dispositivo especificado.
        inputs = tokenizer(text, return_tensors="pt", padding=True, trunca-
    tion=True, max_length=512).to(device)
        model.eval()
        with torch.no_grad():
            # Executa o modelo com as entradas fornecidas, passando os ten-
    sores de input como argumentos para gerar as previsões ou saídas do mod-
    elo.
```

continua

```
        outputs = model(**inputs)

    # Aplica a função softmax as saídas do modelo para converter os va-
lores em probabilidades, normalizadas ao longo da última dimensão.
    probs = torch.nn.functional.softmax(outputs.logits, dim=-1)

    # Retorna a maior probabilidade
    return probs.argmax().item()

example_text = "Eu adorei esse filme! Foi fantástico."
predicted_label = predict_sentiment(example_text)
sentiment = ['Negativo', 'Positivo', 'Neutro']
print(f"Sentimento previsto: {sentiment[predicted_label]}")
```

2.16 F1 *Score* - Importância em Datasets Desbalanceados

O F1 *Score* é uma métrica usada para avaliar a performance de um modelo de classificação, especialmente quando lidamos com datasets desbalanceados. Para entender o F1 Score, é importante conhecer alguns conceitos básicos:

1. **Acurácia** (*Accuracy*): Métrica de avaliação utilizada para medir a proporção de previsões corretas em relação ao total de previsões feitas por um modelo de classificação. Ela é definida pela fórmula:

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN}$$

Onde:

- **TP** (*True Positives*): Verdadeiros Positivos
- **TN** (*True Negatives*): Verdadeiros Negativos
- **FP** (*False Positives*): Falsos Positivos
- **FN** (*False Negatives*): Falsos Negativos

2. **Precisão *(Precision)***: É a proporção de verdadeiros positivos (TP) entre todas as instâncias que o modelo previu como positivas. Em outras palavras, é a quantidade de previsões corretas de uma classe específica em relação ao total de previsões feitas para essa classe.

$$\text{Precisão} = \frac{TP}{TP + FP}$$

onde FP são os falsos positivos.

3. **Revocação** (*Recall*): É a proporção de verdadeiros positivos entre todas as instâncias que são realmente positivas. Ou seja, é a quantidade de previsões corretas de uma classe específica em relação ao total de instâncias reais dessa classe.

$$\text{Revocação} = \frac{TP}{TP + FN}$$

onde FN são os falsos negativos.

4. **F1 Score**: É a média harmônica entre a Precisão e a Revocação. A média harmônica é usada aqui, porque penaliza valores extremos, garantindo que o F1 *Score* será baixo se um dos dois (Precisão ou Revocação) estiver baixo.

$$\text{F1 Score} = 2 \times \frac{\text{Precisão} \times \text{Revocação}}{\text{Precisão} + \text{Revocação}}$$

Por que usar o F1 Score em datasets desbalanceados?

Em datasets desbalanceados, onde uma classe é muito mais frequente do que outra, métricas como a acurácia podem ser enganosas. Por exemplo, se temos 95% das instâncias de uma classe e apenas 5% de outra, um modelo que sempre prevê a classe majoritária terá alta acurácia, mas não será útil para detectar a classe minoritária.

O F1 Score é importante porque leva em consideração tanto a Precisão quanto a Revocação. Em um cenário desbalanceado, isso ajuda a fornecer uma visão mais equilibrada da performance do modelo, destacando se ele é capaz de identificar a classe minoritária com precisão e frequência suficientes.

Assim, o F1 Score é particularmente útil quando a prioridade é garantir que tanto a taxa de detecção dos positivos (revocação) quanto a qualidade das detecções positivas (precisão) são importantes, o que é frequentemente o caso em situações desbalanceadas.

2.17 Exercícios *Fine-Tuning*

1. Alterar alguns parâmetros de treinamento, como: batch_size, learning rate e número de épocas. Avaliar qual o impacto negativo ou positivo na alteração desses parâmetros.
2. Utilizar alguma técnica de balanceamento de dataset e avaliar os resultados, Ex.: *Oversampling* e *Undersampling*.

3. BERT para NER

Neste tópico, vamos demonstrar como usar a biblioteca Hugging Face Transformers e *pipelines* de um modelo já treinado para realizar *Named Entity Recognition* (NER) em textos em português. Utilizaremos um modelo pré-treinado adequado para a tarefa de NER.

3.1 Configuração de variáveis globais

Vamos configurar as variáveis de parâmetros para serem utilizadas durante o código.

```
[ ] # Seleciona a versão do modelo que será utilizada
    model_id = "lfcc/bert-portuguese-ner"
```

3.2 Importação das Bibliotecas

Vamos importar as bibliotecas necessárias para carregar o *pipeline* de NER e o *dataset* em português.

```
[ ] from transformers import pipeline, AutoModelForTokenClassification, Auto-
    Tokenizer
```

3.3 Carregar o Modelo BERT para NER

Vamos carregar o *pipeline* de NER usando um modelo pré-treinado disponível na Hugging Face. Utilizaremos o modelo **xlm-roberta-base**, que é adequado para NER em português.

Ao carregar o *pipeline* NER, usando o aggregation_strategy=”simple” o modelo adotará a estratégia de agregaçao para unir tokens que pertençam a uma mesma entidade nomeada.

```
[ ] # Carregando o modelo e tokenizer
    model = AutoModelForTokenClassification.from_pretrained(model_id)
    tokenizer = AutoTokenizer.from_pretrained(model_id)

    # Carregando o pipeline de NER
    ner_pipeline = pipeline("ner", model=model, tokenizer=tokenizer, aggrega-
    tion_strategy="simple")
```

3.4 Extraindo Entidades de Texto

Vamos utilizar o pipeline de NER para identificar entidades nomeadas em alguns exemplos de texto em português.

```
[ ] # Definindo alguns exemplos de texto em português
    examples = [
        "Paulo viajou para o Estados Unidos.",
        "Marie Curie foi uma cientista polonesa que realizou pesquisas pio-
    neiras sobre radioatividade.",
        "Fernando Henrique Cardoso foi o primeiro presidente eleito após a
    ditadura no Brasil.",
        "Petrobras foi fundada em 3 de outubro de 1953"
    ]

    # Realizando NER nos exemplos de texto
    for example in examples:
        ner_results = ner_pipeline(example)
        print(f"Texto: {example}")
        print("Entidades Nomeadas:")
        for entity in ner_results:
            print(f" - {entity['word']}: {entity['entity_group']} ({enti-
    ty['score']*100:.2f}%)")
        print()
```

4. Tradução Automática usando Hugging Face Transformers

Neste tópico, vamos demonstrar como usar a biblioteca Hugging Face Transformers e *pipelines* de um modelo já treinado para realizar a tradução automática de textos em português para o inglês. Utilizaremos um modelo pré-treinado adequado para a tarefa de tradução.

4.1 Importação das Bibliotecas

Vamos importar as bibliotecas necessárias para carregar o pipeline de tradução.

```
[ ] from transformers import pipeline
```

4.2 Carregar o Modelo de Tradução

Vamos carregar o *pipeline* de tradução usando um modelo pré-treinado disponível na Hugging Face. Utilizaremos o modelo **Helsinki-NLP/opus-mt-pt-en**, que é adequado para tradução do português para o inglês.

```
[ ] # Carregando o pipeline de tradução
    translation_pipeline = pipeline("translation_en_to_pt", model="Helsinki-NLP/opus-mt-tc-big-en-pt")
```

4.3 Exemplos de Tradução

Vamos utilizar o *pipeline* de tradução para traduzir alguns exemplos de texto em português para o inglês.

```
[ ] # Definindo alguns exemplos de texto em português
    examples = [
        "I love learning about natural language processing.",
        "The BERT model was developed by Google AI Research.",
        "Machine translation is a challenging and interesting task."
    ]

    # Realizando a tradução dos exemplos de texto
    for example in examples:
```

continua

```
    translation = translation_pipeline(example)
    print(f"Texto original: {example}")
    print(f"Tradução: {translation[0]['translation_text']}\n")
```

4.4 Exercício - Desafio

Para finalizar a seção do BERT do nosso curso fica o exercício desafio que será divido em três partes:

» Utilizar o *dataset* **hate-speech-portuguese/hate_speech_portuguese** e dividi-lo em 3 partes **train**, **test** e **val.**

» Usar o *dataset* dividido e avaliar somente a parte do **val**, rodar a predição no modelo **adalbertojunior/distilbert-portuguese-cased** e avaliar a métrica **F1 Score** que deve ser calculada usando o **load_metric** (https://huggingface.co/docs/evaluate/choosing_a_metric) do Hugging Face.

» Realizar um *fine-tunning* para esse *dataset* e avaliar a métrica **F1 Score** comparando. Compare o resuldado desse modelo com o do passo anterior e veja qual ficou melhor.

4.6 Recursos Adicionais Oferecidos pela *Hugging Face*

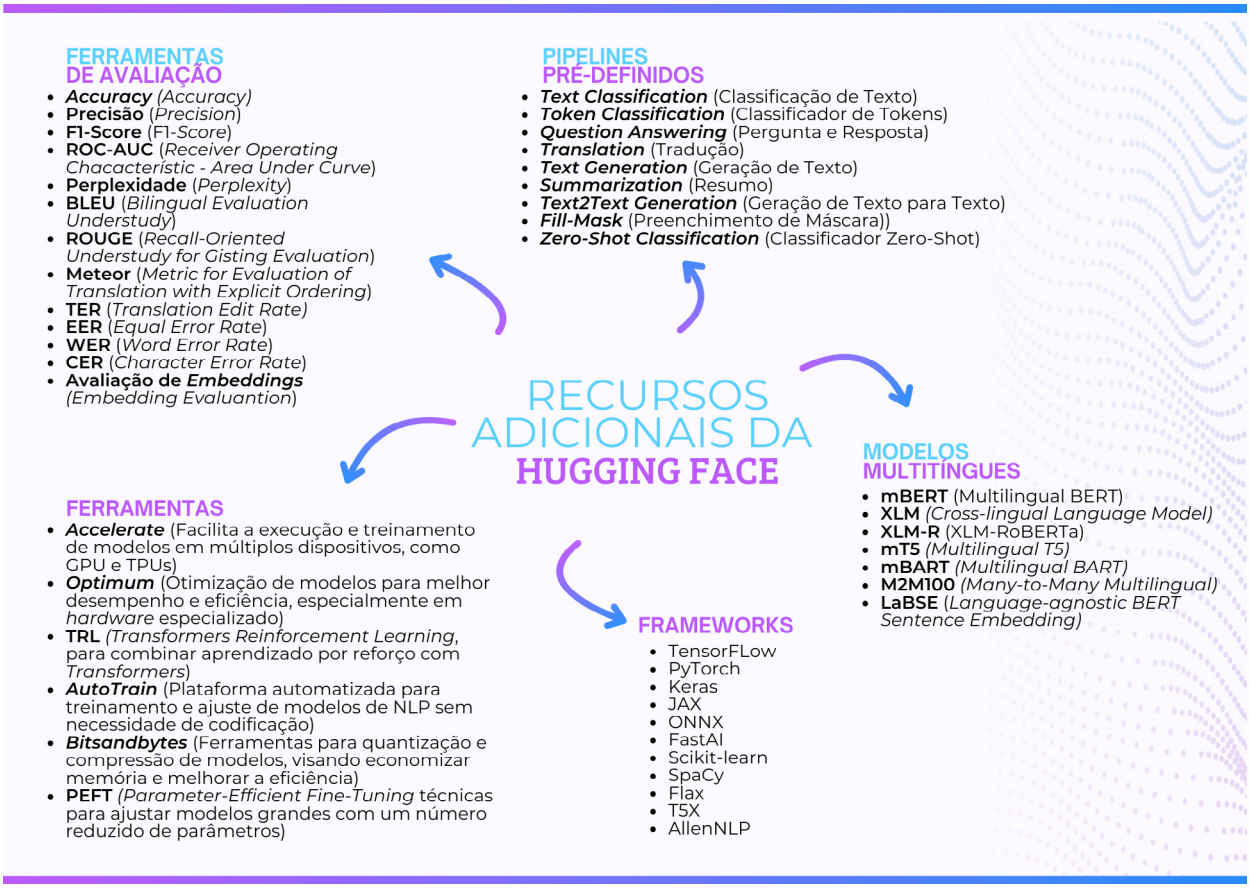
A *Hugging Face* é conhecida por oferecer uma vasta gama de recursos e ferramentas que facilitam o desenvolvimento e a implementação de modelos de PLN. Na Figura 38, é apresentado um mapa mental que oferece uma visão geral desses recursos adicionais divididos em várias categorias importantes. Destacam-se a versatilidade e a abrangência das ferramentas que facilitam a vida dos desenvolvedores e pesquisadores na área de NLP.

No centro do mapa mental, temos “Recursos Adicionais da *Hugging Face*”, que se ramifica em quatro categorias principais:

1. **Pipelines pré-definidos:** facilita a execução de tarefas específicas de NLP, como classificação de texto, tradução, geração de texto, entre outras.
2. **Modelos multilíngues:** inclui modelos treinados para suportar múltiplos idiomas, como mBERT, XLM-R, e mT5, permitindo a criação de aplicações globais e inclusivas.
3. **Ferramentas de avaliação:** conjunto de métricas e ferramentas, como Precisão, *Recall*, *F1-Score*, BLEU, entre outras, que ajudam a avaliar e melhorar a performance dos modelos.
4. **Frameworks:** suporte a diversos *frameworks* populares como *TensorFlow*, *PyTorch*, *Keras* e outros, proporcionando flexibilidade e facilidade de integração.

Além dos recursos adicionais apresentados no mapa mental a seguir (Figura 38), a *Hugging Face* oferece **ferramentas** especializadas como *Accelerate*, *Optimum*, *TRL, AutoTrain, Bitsandbytes* e *PEFT*, que ajudam a otimizar e personalizar os modelos conforme as necessidades específicas dos desenvolvedores.

Figura 43 - Recursos adicionais *Hugging Face*



Fonte: autoria própria.

4.7 Desafios e Limitações Associados ao Uso do BERT

O uso do modelo BERT revolucionou o campo do PLN com sua capacidade de entender o contexto bidirecional em textos. No entanto, essa poderosa ferramenta também apresenta desafios significativos. Abaixo, na Figura 39 está um infográfico que ilustra as três principais limitações associadas ao uso do BERT: o alto custo computacional, a necessidade de grandes conjuntos de dados de treinamento e as considerações éticas relacionadas ao viés nos dados. Esse infográfico fornece uma visão clara e concisa das dificuldades enfrentadas por pesquisadores e desenvolvedores ao trabalhar com o BERT, ajudando a entender melhor os recursos necessários e as precauções que devem ser tomadas para minimizar seus impactos negativos.

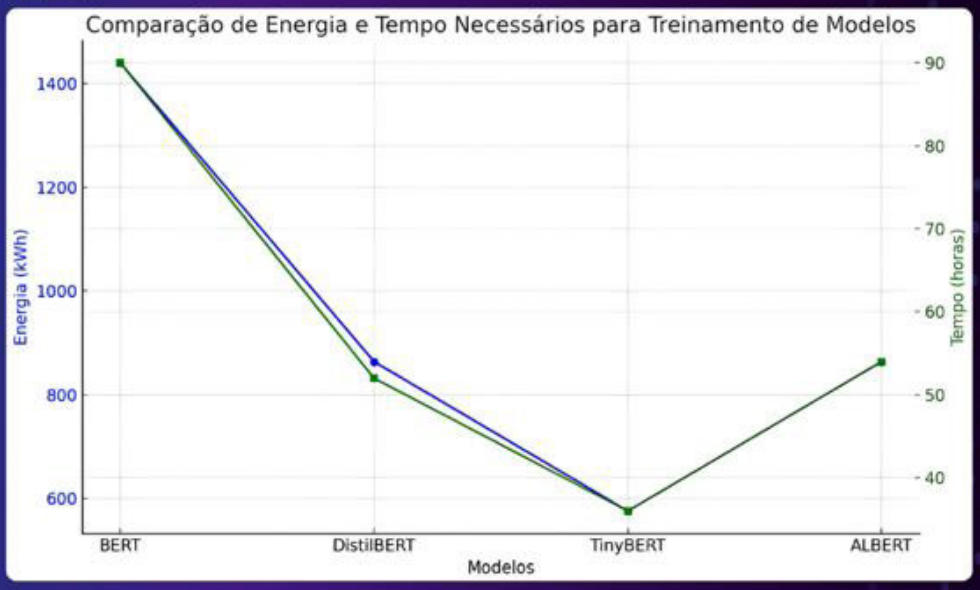
Figura 44 -

LIMITAÇÕES DO USO DO BERT

1 Custo Computacional

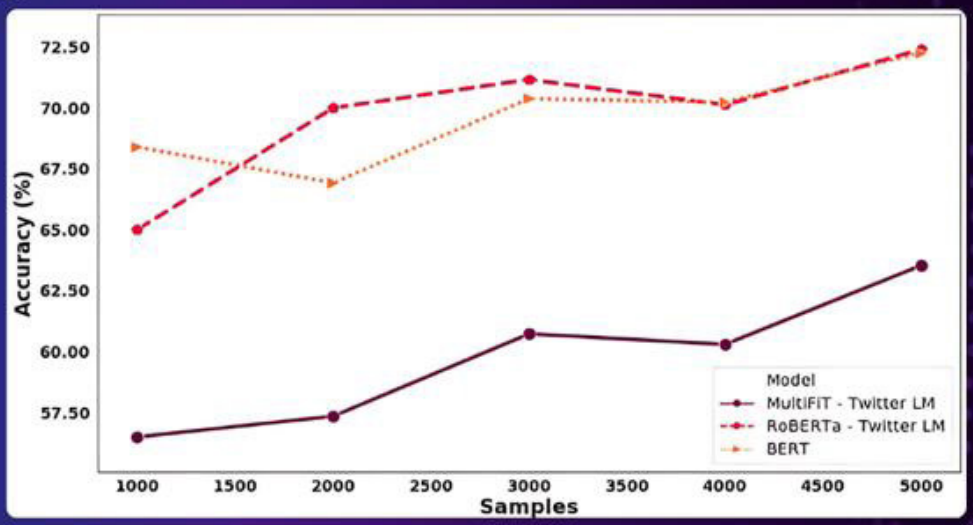
Custo Computacional Elevado

- O BERT requer recursos computacionais significativos para treinamento e inferência.



2 Necessidade de Grandes Conjuntos de Dados

- BERT precisa de grandes volumes de dados anotados para treinamento eficaz.



3 Considerações Éticas e Viés nos Dados

- A presença de vieses nos dados de treinamento pode levar a resultados tendenciosos e discriminatórios.

Fonte: adaptada de dos Santos Neto; da Silva; da Silva Soares (2024).

SAIBA MAIS...

- Hugging Face
- BERT language model (Hashemi-Pour e Lutkevich).
- BERT Explained: State of the art language model for NLP (Horev, 2018).

CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Arquiteturas Profundas para Processamento de Linguagem Natural

Unidade V

GPT, T5 e FLAN-T5

Unidade V: GPT, T5 e FLAN-T5

Nesta Unidade, serão exploradas a série de modelos GPT, desenvolvidos pela *OpenAI*, que revolucionaram o campo do PLN, e também os modelos T5 e FLAN-T5 desenvolvidos pela *Google AI Research*®.

Será apresentada uma visão geral do modelo GPT, destacando seus princípios fundamentais e inovações técnicas. Será possível entender como o GPT utiliza a arquitetura *Transformer* para capturar o contexto bidirecional das palavras e gerar texto de maneira coerente e fluida. Discutiremos também os desafios enfrentados na construção dos primeiros modelos e como esses desafios foram superados. Além disso, serão examinados os processos de pré-treinamento e *fine-tuning* do GPT. Explicaremos como o modelo é inicialmente treinado em grandes *corpora*[4] de texto para aprender representações linguísticas gerais e, posteriormente, ajustado para tarefas específicas utilizando conjuntos de dados menores e mais focados. Esse processo permite que o GPT seja altamente adaptável a diversas tarefas de PLN, como tradução, resumo de texto e resposta a perguntas.

Para os modelos T5 e FLAN-T5, serão comentadas as características de cada arquitetura bem como exemplos práticos de uso.

5.1 *Generative Pre-trained Transformer* (GPT)

Desde a sua introdução, os modelos GPT têm se destacado por sua capacidade de gerar texto de alta qualidade e compreender contextos complexos, tornando-se ferramentas indispensáveis para diversas aplicações de PLN. Desenvolvido pela *OpenAI*, esse modelo representa um dos avanços mais significativos no campo do PLN.

O GPT utiliza como base a arquitetura do *Transformer*, que por sua vez utiliza mecanismos de autoatenção para atribuir pesos diferentes a diferentes palavras em uma frase, permitindo que o modelo foque nas partes mais relevantes do contexto. Para esse modelo, foi utilizada a variante de atenção chamada "*masked self-attention*", que impede que a previsão de uma palavra dependa de palavras futuras, forçando o modelo a prever a próxima palavra com base apenas nas palavras anteriores.

4 *Corpora* é o plural de *corpus*. Um **conjunto de dados textuais** ou uma **coleção de textos** usados para treinamento, avaliação ou análise de modelos de linguagem é chamado de *corpus*. **Várias coleções** de textos ou conjuntos de dados textuais são chamados de *corpora*.

Com suas várias iterações (GPT, GPT-2, GPT-3), mostrou uma capacidade sem precedentes para gerar texto coerente, realizar tarefas de compreensão de linguagem e até mesmo resolver problemas complexos. Sua abordagem baseia-se no pré-treino em uma vasta quantidade de texto não rotulado, seguido de um ajuste fino em tarefas específicas.

5.1.1 Pré-treino do Modelo GPT

O pré-treino é a fase mais crucial no GPT, onde o modelo aprende representações ricas da linguagem a partir de um grande *corpus* de texto. O pré-treino envolve várias etapas detalhadas:

1. **Corpus de treinamento:** o GPT é pré-treinado em um *corpus* massivo e diversificado, contendo texto extraído de livros, artigos, *sites* da *web* etc. O objetivo é expor o modelo a uma ampla variedade de estilos e tópicos de linguagem. Por exemplo, o GPT-3 foi treinado no conjunto de dados "*Common Crawl*", que abrange uma vasta quantidade de texto extraído da *web*.

2. **Modelo Transformer:** o modelo utilizado é um *Transformer* com 12 camadas. Esses são conhecidos por sua capacidade de capturar dependências de longo alcance em texto, graças ao uso de mecanismos de atenção. Cada camada do *Transformer* consiste em uma subcamada de atenção multi-cabeça e uma subcamada de *feedforward* totalmente conectada. Essas subcamadas são equipadas com conexões residuais e normalização de camada.

3. **Tarefa de modelagem de linguagem:** a tarefa principal durante o pré-treino é a modelagem de linguagem autoregressiva, onde o modelo é treinado para prever a próxima palavra em uma sequência, dadas as palavras anteriores. A função de perda utilizada é a de entropia cruzada, que mede a discrepância entre a distribuição predita pelo modelo e a distribuição real das palavras.

4. **Treinamento em escala massiva:** o GPT é treinado usando grandes quantidades de recursos computacionais, incluindo múltiplas GPUs ou *Tensor Processing Units* (TPUs), para processar o enorme volume de dados. O treinamento é distribuído e paralelizado para acelerar o processo e lidar com a escala massiva dos dados.

5. **Técnicas de regularização:** diversas técnicas de regularização são aplicadas para evitar o *overfitting* durante o pré-treino. Entre elas estão o *dropout*, a normalização de camadas e a penalização de entropia para regularização dos pesos. O *dropout* é utilizado para desativar aleatoriamente neurônios durante o treinamento, ajudando a prevenir o overfitting e melhorar a generalização do modelo.

5.1.2 Evolução do GPT (1, 2, 3)

Desde a introdução do GPT-1, em 2018, a série de modelos GPT desenvolvida pela OpenAI tem revolucionado o campo do PLN. Cada versão sucessiva trouxe avanços significativos em termos de capacidade, desempenho e aplicabilidade, estabelecendo novos padrões para a geração de texto e outras tarefas de PLN.

Na Tabela 5, a seguir, apresentamos um comparativo que destaca as principais características e evoluções dos modelos GPT-1, GPT-2 e GPT-3. Ela fornece uma visão detalhada das diferenças entre as três versões, incluindo o tamanho do modelo, os *corpora* de treinamento, as capacidades de geração de texto e as aplicações práticas. Essa comparação ajuda a entender como cada evolução contribuiu para aprimorar a habilidade dos modelos em compreender e gerar texto de maneira mais coerente e eficaz, ampliando as possibilidades de uso em diversas áreas.

Tabela 5 - Evoluções do Modelo GPT

Característica	GPT-1	GPT-2	GPT-3
Ano de Lançamento	2018	2019	2020
Tamanho do Modelo	117 milhões de parâmetros	1,5 bilhões de parâmetros	175 bilhões de parâmetros
Corpus de Treinamento	Livros (BookCorpus)	Vários datasets (8 milhões de documentos, 40GB de texto)	Vários datasets (570GB de texto)
Arquitetura	*Transformer* Unidirecional	*Transformer* Unidirecional	*Transformer* Unidirecional
Tarefas de PNL	Geração de Texto, Tradução, Resumo, Perguntas e Respostas	Geração de Texto, Tradução, Resumo, Perguntas e Respostas, e mais	Geração de Texto, Tradução, Resumo, Perguntas e Respostas, Programação, e mais
Contexto de Entrada	512 tokens	1024 tokens	2048 tokens
Capacidade de Geração	Coerência em textos curtos	Melhor coerência e capacidade de manter contexto em textos mais longos	Capacidade de manter contexto em textos muito longos e maior fluidez na geração
Desempenho	Bom em tarefas de PNL específicas	Excelente em diversas tarefas de PNL	Supera os modelos anteriores em quase todas as tarefas de PNL
Disponibilidade	Código e modelo disponíveis na OpenAI	Inicialmente restrito, depois disponibilizado em versões menores	Acesso via API da OpenAI, modelo completo não disponível publicamente
Exemplos de Aplicações	Assistentes virtuais, Tradução automática	Chatbots avançados, Geração de conteúdo, Assistentes de código	Assistentes virtuais avançados, Geração de código, Análise de sentimentos

Fonte: autoria própria.

5.2 T5 e FLAN-T5

Nesta seção, exploraremos os modelos T5 e FLAN-T5, duas variantes poderosas no campo do PLN. Desenvolvidos para abordar uma ampla gama de tarefas de PLN por meio de uma abordagem unificada, esses modelos têm demonstrado desempenhos impressionantes em diversas aplicações práticas.

5.2.1 Arquitetura do Modelo T5

O ***T**ext-**T**o-**T**ext **T**ransfer **T**ransformer* (T5), desenvolvido pela equipe do *Google AI Research*®, é baseado na arquitetura *Transformer*, uma estrutura de *Deep Learning* que revolucionou o campo de PLN. A principal inovação do T5 é sua abordagem "*text-to-text*", onde todas as tarefas de PLN são formuladas como tarefas de transformação de texto. Isso inclui tradução, sumarização, classificação, resposta a perguntas e muitas outras.

Principais características:

- **Fundamentos:** baseado na arquitetura *Transformer*, o T5 converte todas as tarefas de PLN em um formato de texto para texto, onde tanto a entrada quanto a saída são sequências de texto.
- **Treinamento:** inclui uma fase de pré-treinamento em um *dataset* diversificado, seguida de um *fine-tuning* específico da tarefa, permitindo que o modelo se adapte a uma ampla gama de aplicações.
- **Estrutura:** utiliza uma combinação de blocos de *encoder* e *decoder* do *Transformer*, otimizando o processo de aprendizado e a geração de respostas.

O T5 alcançou resultados de ponta em vários *benchmarks*[5] de PLN, como GLUE, SuperGLUE, e outros. Esses *benchmarks* são usados para avaliar o desempenho de modelos de linguagem natural em uma variedade de tarefas de compreensão de linguagem, sendo que SuperGLUE é uma versão mais desafiadora e avançada do GLUE.

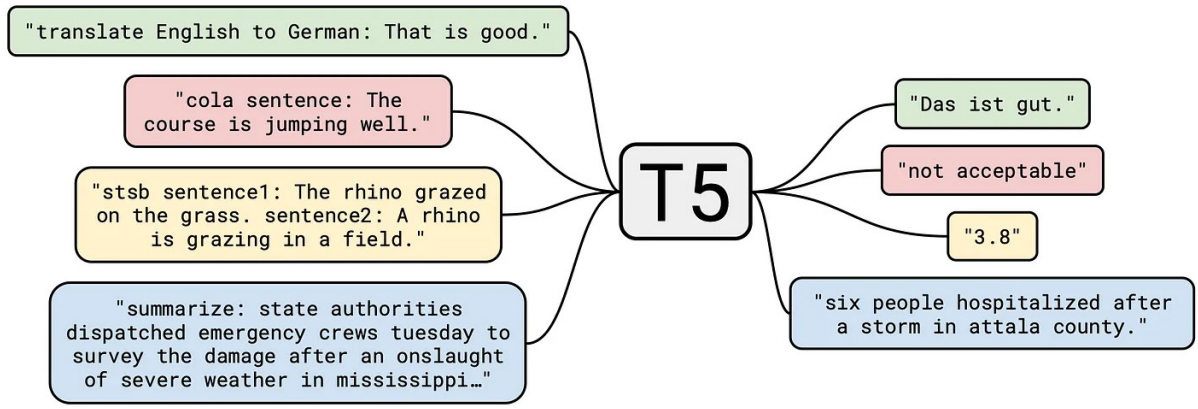
Na Figura 40, é ilustrado o funcionamento do modelo T5, que reformula diversas tarefas de NLP como problemas de conversão de texto em texto. Isso significa que tanto as entradas quanto as saídas do modelo são representadas em formato textual, independentemente da tarefa a ser realizada. Cada cor representa uma tarefa específica, conforme explicado abaixo:

- **Tradução (verde):** essa entrada pede ao T5 para traduzir uma frase do inglês para o alemão. A estrutura da entrada já inclui a tarefa (tradução) e a língua de destino. A saída é a frase traduzida.
- **Classificação de frases (rosa):** essa entrada refere-se à tarefa de verificar a aceitabilidade gramatical da frase em questão. O modelo classifica se a frase é "*acceptable*" ou "*not acceptable*".

5 *Benchmarks*: processo de avaliação comparativa que mede o desempenho de modelos em *datasets* específicos.

- **Similaridade de frases (amarelo):** nesse caso, o T5 está avaliando a similaridade semântica entre duas frases e atribui uma pontuação que indica o quão semelhantes as duas sentenças são (em uma escala de 0 a 5).
- **Sumarização (azul):** a entrada inclui uma longa descrição de um evento e a tarefa do T5 é gerar um resumo conciso e informativo desse texto. A saída resume o conteúdo principal em uma frase curta.

Figura 45 - *Framework text-to-text* T5



Fonte: adaptada de Raffel *et al.* (2020).

5.2.2 Arquitetura do Modelo FLAN- T5

O FLAN-T5 é uma variante do T5 que incorpora técnicas de *fine-tuning* para melhorar o desempenho em tarefas específicas de PLN. Esse modelo é treinado usando a técnica FLAN. O FLAN envolve o ajuste fino de modelos com base em instruções (*instruction-tuning*). Desenvolvido pelo *Google AI Research*®, o FLAN-T5 é projetado para ser mais robusto e generalizável.

Principais características:

- **Aprimoramentos no T5:** incorpora ajustes finos em *datasets* de múltiplas tarefas de PLN simultaneamente, utilizando uma abordagem de treinamento mais flexível e menos dependente de dados anotados especificamente para tarefas.
- **Generalização:** projetado para melhorar a capacidade de generalização do modelo T5, o FLAN-T5 demonstra maior eficácia em tarefas de PLN que não foram explicitamente incluídas durante o treinamento.

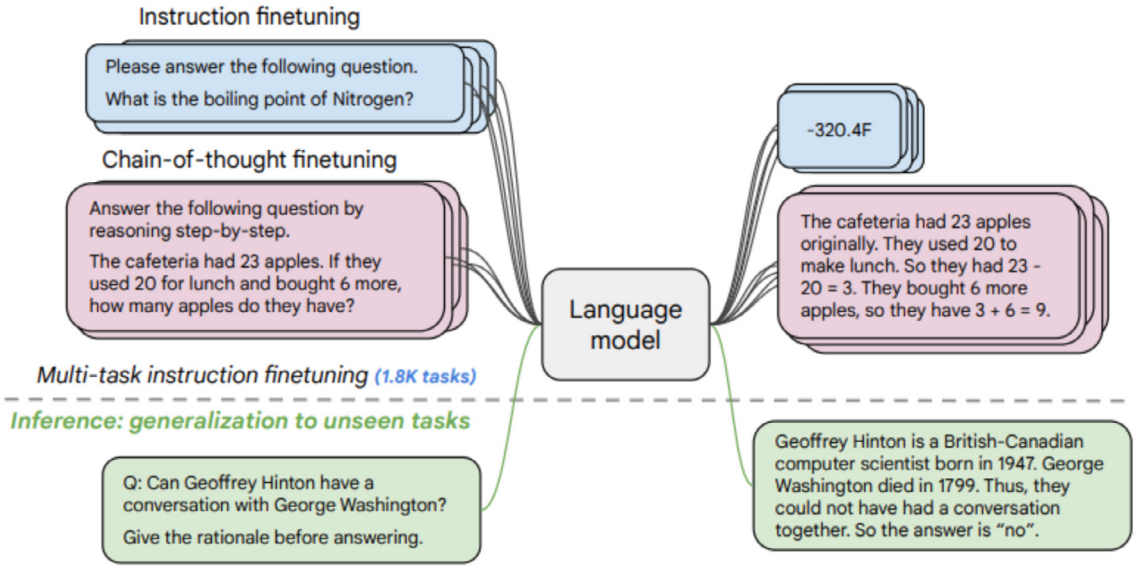
Na imagem da Figura 41, são ilustradas diferentes formas de ajuste fino (*fine-tuning*) aplicadas do modelo FLAN-T5, com foco em três principais abordagens: ajuste fino por instrução, ajuste fino com cadeia de raciocínio e ajuste fino multi-tarefa. Segue uma breve descrição para as três formas:

- ***Instruction fine-tuning* (ajuste fino por instrução):** esse tipo de ajuste fino envolve fornecer uma instrução clara ao modelo para realizar uma tarefa específica, como responder a perguntas factuais. O modelo foi ajustado para reconhecer que a entrada é uma pergunta e dar uma resposta factual direta.

- ***Chain-of-thought fine-tuning* (ajuste fino com cadeia de raciocínio):** neste caso, o ajuste fino envolve a solicitação de uma explicação passo a passo para chegar à resposta. Em vez de apenas fornecer o resultado final, o modelo apresenta o raciocínio completo, decompondo a questão em partes menores e explicando o processo de cálculo.

- ***Multi-task instruction fine-tuning* (ajuste fino multi-tarefa por instrução):** o ajuste fino multitarefa permite ao modelo lidar com uma variedade de tarefas ao mesmo tempo, aplicando instruções que vão além de apenas responder à pergunta, pedindo também justificativas. O modelo reconhece que precisa fornecer uma explicação antes de dar a resposta final. Esse tipo de ajuste permite ao modelo generalizar para novas tarefas, até mesmo aquelas não vistas durante o treinamento.

Figura 46 - *Framework* ajuste fino FLAN-T5



Fonte: adaptada de Chung *et al.* (2022).

5.2.3 Aplicações Práticas dos Modelos T5 e FLAN-T5

O FLAN-T5 pode ser usado em várias tarefa de NLP, seguem alguns exemplos a seguir:

1. **Tradução Automática:** tanto o T5 quanto o FLAN-T5 são capazes de realizar traduções de alta qualidade entre diversos idiomas, graças à sua abordagem generalista e à capacidade de entender nuances linguísticas.

2. **Sumarização de textos:** os modelos são excepcionais em condensar informações, produzindo resumos concisos que mantêm os pontos críticos dos textos originais.
3. **Geração de texto:** utilizados para criar conteúdo escrito variado, desde artigos informativos até narrativas criativas.
4. **Classificação de sentimento:** eficientes em identificar e classificar as emoções subjacentes em textos, uma aplicação valiosa para análises de mercado e *feedback* de clientes.
5. **Resposta a perguntas:** capazes de compreender e processar perguntas para fornecer respostas precisas e informativas.

5.3 Exemplos Práticos do GPT-2 e FLAN-T5

Para uma compreensão prática e detalhada das aplicações de PLN, recomendamos acessar o *notebook* que desenvolvemos. Este *notebook* demonstra passo a passo como realizar a geração de texto utilizando o GPT-2, assim como a sumarização de textos com os modelos GPT-2 e FLAN-T5. Embora esses modelos sejam eficazes para várias tarefas de NLP, eles possuem limitações quando comparados aos modelos de linguagem mais recentes, como GPT-3 e GPT-4. Os modelos mais recentes, com um número significativamente maior de parâmetros e treinados em *datasets* mais extensos e diversos, apresentam uma capacidade superior de manter coerência em textos longos, compreender contextos complexos e gerar respostas mais precisas e relevantes.

Notebook Colab

1. Objetivos de Aprendizagem

Neste notebook, exploraremos técnicas de Processamento de Linguagem Natural (NLP) utilizando modelos avançados de geração e sumarização de texto. O objetivo é aprender a aplicar modelos pré-treinados, como o GPT-2 e o T5, em tarefas de geração e resumo de texto, tanto em inglês quanto em português.

2. Geração de Texto usando GPT-2

Neste exemplo, vamos demonstrar como integrar o GPT-2 em uma aplicação para gerar texto automaticamente. Este tipo de aplicação pode ser útil para empresas que desejam automatizar a criação de conteúdo, como artigos, descrições de produtos, ou postagens em mídias sociais.

2.1 Instalação das Dependências

Vamos começar instalando as bibliotecas necessárias.

```
[ ] !pip install transformers
    !pip install torch
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.10/
dist-packages (4.42.4)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/
dist-packages (from transformers) (3.15.4)
Requirement already satisfied: huggingface-hub<1.0,>=0.23.2 in /usr/lo-
cal/lib/python3.10/dist-packages (from transformers) (0.23.5)
Requirement already satisfied: numpy<2.0,>=1.17 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (1.25.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (24.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/
dist-packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (2024.5.15)
Requirement already satisfied: requests in /usr/local/lib/python3.10/
dist-packages (from transformers) (2.31.0)
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/py-
thon3.10/dist-packages (from transformers) (0.4.3)
Requirement already satisfied: tokenizers<0.20,>=0.19 in /usr/local/lib/
python3.10/dist-packages (from transformers) (0.19.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/
dist-packages (from transformers) (4.66.4)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/py-
thon3.10/dist-packages (from huggingface-hub<1.0,>=0.23.2->transformers)
(2024.6.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/
lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.23.2->trans-
formers) (4.12.2)
```

continua

```
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/
lib/python3.10/dist-packages (from requests->transformers) (3.3.2)

Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/
dist-packages (from requests->transformers) (3.7)

Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py-
thon3.10/dist-packages (from requests->transformers) (2.0.7)

Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py-
thon3.10/dist-packages (from requests->transformers) (2024.7.4)

Requirement already satisfied: torch in /usr/local/lib/python3.10/
dist-packages (2.3.1+cu121)

Requirement already satisfied: filelock in /usr/local/lib/python3.10/
dist-packages (from torch) (3.15.4)

Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/
lib/python3.10/dist-packages (from torch) (4.12.2)

Requirement already satisfied: sympy in /usr/local/lib/python3.10/
dist-packages (from torch) (1.13.1)

Requirement already satisfied: networkx in /usr/local/lib/python3.10/
dist-packages (from torch) (3.3)

Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/
dist-packages (from torch) (3.1.4)

Requirement already satisfied: fsspec in /usr/local/lib/python3.10/
dist-packages (from torch) (2024.6.1)

Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch)

  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_
x86_64.whl.metadata (1.5 kB)

Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch)

  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_
x86_64.whl.metadata (1.5 kB)

Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch)

  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_
x86_64.whl.metadata (1.6 kB)

Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch)

  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl.
metadata (1.6 kB)

Collecting nvidia-cublas-cu12==12.1.3.1 (from torch)

  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.
whl.metadata (1.5 kB)

Collecting nvidia-cufft-cu12==11.0.2.54 (from torch)

  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.
whl.metadata (1.5 kB)

Collecting nvidia-curand-cu12==10.3.2.106 (from torch)

  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.
whl.metadata (1.5 kB)

Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch)

  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_
x86_64.whl.metadata (1.6 kB)
```

continua

```
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch)
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_
x86_64.whl.metadata (1.6 kB)
Collecting nvidia-nccl-cu12==2.20.5 (from torch)
  Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl.
metadata (1.8 kB)
Collecting nvidia-nvtx-cu12==12.1.105 (from torch)
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl.
metadata (1.7 kB)
Requirement already satisfied: triton==2.3.1 in /usr/local/lib/py-
thon3.10/dist-packages (from torch) (2.3.1)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolv-
er-cu12==11.4.5.107->torch)
  Downloading nvidia_nvjitlink_cu12-12.5.82-py3-none-manylinux2014_
x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/py-
thon3.10/dist-packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/py-
thon3.10/dist-packages (from sympy->torch) (1.3.0)
Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl
(410.6 MB)
Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.
whl (14.1 MB)
Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.
whl (23.7 MB)
Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_
x86_64.whl (823 kB)
Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl
(731.7 MB)
Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl
(121.6 MB)
Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl
(56.5 MB)
Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.
whl (124.2 MB)
Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.
whl (196.0 MB)
Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl
(176.2 MB)
Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99
kB)
Downloading nvidia_nvjitlink_cu12-12.5.82-py3-none-manylinux2014_x86_64.
whl (21.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.3/21.3 MB 26.2 MB/s eta
0:00:00
```

continua

```
    Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12,
    nvidia-nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-cuda-run-
    time-cu12, nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-
    cu12, nvidia-cusparse-cu12, nvidia-cudnn-cu12, nvidia-cusolver-cu12
    Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cup-
    ti-cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-run-
    time-cu12-12.1.105 nvidia-cudnn-cu12-8.9.2.26 nvidia-cufft-cu12-11.0.2.54
    nvidia-curand-cu12-10.3.2.106 nvidia-cusolver-cu12-11.4.5.107 nvid-
    ia-cusparse-cu12-12.1.0.106 nvidia-nccl-cu12-2.20.5 nvidia-nv-
    jitlink-cu12-12.5.82 nvidia-nvtx-cu12-12.1.105
```

2.2 Importação das Bibliotecas

Vamos importar as bibliotecas necessárias para carregar o *pipeline* de geração de texto.

- *pipeline* - cria pipelines para processamento simplificando tarefas como análise de sentimentos, classificação e resumo de textos entre outros.
- set_seed - útil para garantir a reprodutibilidade dos resultados. Em operações que envolvem aleatoriedade (inicialização de pesos de modelos, embaralhamento de dados), há a garantia de ter o mesmo comportamento a cada execução.

```
[ ] from transformers import pipeline, set_seed
    import torch
```

2.3 Carregamento do Pipeline de Geração de Texto

Vamos carregar o *pipeline* de geração de texto utilizando o modelo pierreguillou/gpt2-small-portuguese. Utilizamos a entrada “device=0 if torch.cuda.is_available() else -1” no comando para informar que o modelo será executado usando a CPU. Há opçoes que podem ser adotadas para uso de GPU caso exista.

```
[ ] model_name = "pierreguillou/gpt2-small-portuguese"

    # Carregando o pipeline de geração de texto
    generator = pipeline("text-generation", model=model_name, device=0 if
    torch.cuda.is_available() else -1)
```

2.4 Função para Gerar Texto Automaticamente

Vamos definir uma função que utiliza o pipeline do GPT-2 para gerar texto a partir de um *prompt* fornecido. As entradas para generator que é o pipeline de geração de texto:

- *prompt*: Será fornecido a função em comando no próximo bloco de código;
- *max_length*: tamanho máximo da entrada a ser gerada, neste caso foi definido 200 tokens;
- *num_return_sequences*: Define a quantidade de seqências a serem retornadas pelo gerador.
- *no_repeat_ngram_size*: Utilizado para impedir que n-grams sejam repetidos no texto, neste caso foram utilizados como entrada os bigramas.
- *early_stopping*: assim que a sequência for csiderada adequada pelo modelo, ele para a geração de texto.

```
[ ] def generate_text(prompt, generator, max_length=200):
        # Gerar o texto utilizando o pipeline GPT-2
        responses = generator(prompt, max_length=max_length, num_return_se-
    quences=1, no_repeat_ngram_size=2, early_stopping=True)

        # Obter o texto gerado
        generated_text = responses[0]['generated_text']
        return generated_text
```

2.5 Exemplos de Geração de Texto

Vamos definir alguns prompts e utilizar o GPT-2 para gerar textos automáticos a partir desses prompts.

```
[ ] prompts = [
        "A inteligência artificial está transformando o setor de saúde ao",
        "As tendências de tecnologia para 2023 incluem",
        "A sustentabilidade nas empresas é importante porque",
        "A transformação digital nas pequenas empresas",
        "O futuro do trabalho remoto será influenciado por"
    ]

    # Gerando e exibindo textos para cada prompt
```

continua

```
for prompt in prompts:
    generated_text = generate_text(prompt, generator)
    print(f"Prompt: {prompt}")
    print(f"Texto Gerado: {generated_text}\n")
```

```
Prompt: A inteligência artificial está transformando o setor de saúde ao
Texto Gerado: A inteligência artificial está transformando o setor de
saúde ao nível técnico, isto é, se transforma os hospitais de ponta em
clínicas, laboratórios e centros de treinamento. A pesquisa nessa área
está também presente em outros setores da Saúde, como no setor da edu-
cação. Os principais centros estão dentro dos estados da Bahia e do Pará.


As agências de pesquisa estão presentes nas mais variadas atividades de
produção e pesquisa envolvendo pessoas, empresas e os governos. Para o
trabalho de controle dessas agências, é necessário fazer investimentos
estratégicos nos setores de Ciência, Tecnologia e Inovação. Estas agên-
cias também implementam um projeto de avaliação que visa medir o retor-
no de um certo estímulo, sendo que esse valor está na relação do fator
a respeito do investimento. Tais investimentos tendem a ser metas, por
exemplo, de curto prazo, e podem ser de longo prazo. Além disso, são
desenvolvidos programas em que os programas sejam seguidos com resultados
significativos. O principal programa de monitoramento, em conjunto com
várias agências dos Estados-membros, está disponível para o público


Prompt: As tendências de tecnologia para 2023 incluem
Texto Gerado: As tendências de tecnologia para 2023 incluem a necessidade
de uma nova camada central de energia e redução de riscos de prolifer-
ação. O governo espera desenvolver essa tecnologia ao redor do mundo, a
fim de assegurar que as centrais de transporte de alimentos não precisam
ficar no controle destas tecnologias. A adoção de um sistema piloto em
todo o mundo está contribuindo para reduzir a pressão sobre importações
de combustíveis.


A tecnologia é um grande problema para muitas organizações e outras orga-
nizações. Como consequência, grandes empresas, como o GE, e grandes com-
panhias emergentes como a Samsung têm o interesse em desenvolver técnicas
como "scale de combustível", para que não seja necessário deixar de oper-
ar as infraestruturas necessárias para garantir sua cadeia de suprimen-
tos. As empresas de origem europeia como Toyota são especialmente famosas
por desenvolver tecnologias em larga escala, com potencial para levar à
revolução de forma rápida e barata. Isso cria uma grande possibilidade de
criação de empresas em grande escala. Esta mudança permitiria que empre-
sas mais rentáveis pudessem usar o investimento de mais de $100 bilhões
de dólares


Prompt: A sustentabilidade nas empresas é importante porque
Texto Gerado: A sustentabilidade nas empresas é importante porque pro-
move, promove e fortalece as condições de sustentabilidade. O processo de
```

continua

criação de um sistema de governança que garante a qualidade de vida dos cidadãos foi desenvolvido e implementado pelo Ministério da Integração Nacional, órgão vinculado ao Ministério do Desenvolvimento Rural e Peshina, em junho de 1987. A partir do segundo trimestre do mesmo ano, a organização lançou um relatório de acompanhamento aos gestores públicos sobre a sustentabilidade dos diferentes níveis de atividade. No mesmo período, foram adotados novos Planos de Acompanhamento Nacional (NAOPS), o qual definiu o sistema que está em fase de implementação.

A responsabilidade pela gestão dos resíduos urbanos é fundamental para promover o desenvolvimento econômico e social do município, principalmente através da política de transparência do Poder Legislativo municipal. Embora o Poder Executivo deva ser mantido como o chefe administrativo-administrativo da cidade, todos os prefeitos e vereadores são eleitos para mandatos de quatro anos. Ainda que a Câmara Municipal do Recife seja composta por representantes da classe executiva (ass

Prompt: A transformação digital nas pequenas empresas

Texto Gerado: A transformação digital nas pequenas empresas pode ser crucial. A tecnologia, aliada à redução da necessidade de mão-de-obra qualificada e às habilidades digitais, é essencial para a qualidade de vida e melhorar a infraestrutura produtiva dessas pequenas cidades.

O desenvolvimento de tecnologias digitais na educação, saúde e ambiente urbano são cada vez mais importantes, além da qualificação profissional, como acesso a serviços de consultoria, gestão integrado, políticas públicas e iniciativas de melhoria de qualidade no desempenho das empresas. Atualmente, os grandes desafios da sociedade que envolvem a digitalização em massa são inúmeras, e são, portanto, bastante diversificados. Assim, a globalização gerou mudanças que mudaram o curso da tecnologia na sociedade, trazendo novos desafios à produção e transformação na tecnologia da educação. O desafio de criar e manter tecnologias na produção, de forma acessível à comunidade, não é apenas a questão básica da informação, mas a tecnologia produzida e distribuída de modo que todas possam se identificar com a oferta da demanda. Os desafios do mercado global da Educação, Saúde

Prompt: O futuro do trabalho remoto será influenciado por

Texto Gerado: O futuro do trabalho remoto será influenciado por um futuro distante.

O ponto de partida é um mundo onde os humanos e robôs estão criando suas próprias invenções. A combinação desses dois meios é tão importante como um ponto comum do jogo. O jogo, por trás do próximo produto a ser desenvolvido, tem um "efeito futuro", é o conceito da liberdade criativa que os computadores controlam de maneira rápida. Assim, o objetivo das tecnologias de hoje é evitar que a humanidade crie ou descuida os objetos que o computador constrói. Além disso, as pessoas têm liberdade na escolha de qual objeto ele se tornará um computador, de modo que as próximas tecnologias não necessariamente melhoram o nível de segurança de um próximo computador.

continua

```
    Uma das principais mudanças no campo de jogos em meados da era 50 a.C. é
    a introdução do uso do CAD/CES para criar novas tecnologias (e não máqui-
    nas). Essa mudança abriu a margem de alguns jogos já terem apresentado
    alguma falha. Por exemplo, a série
```

3. Sumarização de Texto usando GPT-2

Neste exemplo, vamos demonstrar como integrar o GPT-2 para realizar sumarização de texto. Embora o GPT-2 não seja especificamente treinado para sumarização, podemos adaptá-lo fornecendo um prompt que oriente o modelo a gerar um resumo do texto fornecido.

3.1 Carregamento do Pipeline de Geração de Texto

Vamos carregar o *pipeline* de geração de texto utilizando o modelo pierreguillou/gpt2-small-portuguese.

```
[ ] # Carregando o pipeline de geração de texto
    generator = pipeline("text-generation", model=model_name, device=0 if
    torch.cuda.is_available() else -1)


    # Definindo uma semente para reprodutibilidade
    set_seed(42)
```

3.2 Função para Sumarizar Texto

Vamos definir uma função que utiliza o *pipeline* do GPT-2 para gerar resumos de textos longos. Utilizaremos um *prompt* específico para orientar o modelo a gerar um resumo.

```
[ ] def summarize_with_gpt2(text, generator, max_length=150):
        prompt = f"Resuma o seguinte texto:\n\n{text}\n\nResumo:"
        response = generator(prompt, max_length=max_length, num_return_se-
    quences=1, no_repeat_ngram_size=2, early_stopping=True)
        summarized_text = response[0]['generated_text'].replace(prompt, '').
    strip()
        return summarized_text
```

3.3 Exemplos de Textos para Sumarização

Vamos definir alguns exemplos de textos longos e utilizar o modelo para gerar seus resumos.

```
[ ] texts = [
        """A inteligência artificial (IA) está transformando o setor de saúde
    de várias maneiras. Desde a análise de grandes volumes de dados de paci-
    entes para prever doenças até a personalização de tratamentos com base em
    dados genéticos, a IA está revolucionando como os cuidados de saúde são
    prestados. Além disso, tecnologias como machine learning e deep learning
    estão sendo usadas para desenvolver novas drogas e identificar potenciais
    tratamentos para doenças complexas. No entanto, a adoção da IA na saúde
    também enfrenta desafios, incluindo preocupações com a privacidade dos
    dados dos pacientes e a necessidade de regulamentações claras para garan-
    tir o uso ético da tecnologia.""",
        """As tendências de tecnologia para 2023 incluem avanços significati-
    vos em várias áreas. A inteligência artificial e o machine learning con-
    tinuam a ser tendências importantes, com aplicações que vão desde a au-
    tomação de processos até a criação de novas formas de interatividade com
    os usuários. A computação quântica, embora ainda em seus estágios inici-
    ais, promete revolucionar a forma como processamos informações. Além dis-
    so, a 5G está se expandindo globalmente, proporcionando velocidades de
    internet mais rápidas e mais estáveis. Outras tendências incluem a real-
    idade aumentada e virtual, que estão sendo cada vez mais utilizadas em
    setores como educação, entretenimento e saúde.""",
        """A sustentabilidade nas empresas é um tema cada vez mais impor-
    tante, à medida que as organizações reconhecem a necessidade de minimizar
    seu impacto ambiental. A implementação de práticas sustentáveis não só
    ajuda a proteger o meio ambiente, mas também pode resultar em economias
    significativas de custos e melhorar a imagem da empresa junto aos consum-
    idores. Entre as práticas mais comuns estão a redução do consumo de en-
    ergia, a reciclagem de materiais e a escolha de fornecedores que adotam
    práticas sustentáveis. No entanto, a transição para a sustentabilidade
    pode ser desafiadora e requer um compromisso a longo prazo e a partici-
    pação de todas as partes interessadas da empresa."""
    ]

    # Gerando e exibindo resumos para cada texto
    for text in texts:
        summary = summarize_with_gpt2(text, generator)
        print(f"Texto Original: {text}")
        print(f"Resumo: {summary}")
        print()
```

continua

```
Truncation was not explicitly activated but `max_length` is provided a specific value, please use `truncation=True` to explicitly truncate examples to max length. Defaulting to 'longest_first' truncation strategy. If you encode pairs of sequences (GLUE-style) with the tokenizer you can select this strategy more precisely by providing a specific strategy to `truncation`.

/usr/local/lib/python3.10/dist-packages/transformers/generation/configuration_utils.py:588: UserWarning: `num_beams` is set to 1. However, `early_stopping` is set to `True` -- this flag is only used in beam-based generation modes. You should set `num_beams>1` or unset `early_stopping`.

  warnings.warn(

Texto Original: A inteligência artificial (IA) está transformando o setor de saúde de várias maneiras. Desde a análise de grandes volumes de dados de pacientes para prever doenças até a personalização de tratamentos com base em dados genéticos, a IA está revolucionando como os cuidados de saúde são prestados. Além disso, tecnologias como machine learning e deep learning estão sendo usadas para desenvolver novas drogas e identificar potenciais tratamentos para doenças complexas. No entanto, a adoção da IA na saúde também enfrenta desafios, incluindo preocupações com a privacidade dos dados dos pacientes e a necessidade de regulamentações claras para garantir o uso ético da tecnologia.

Resumo: como a humanidade evoluiu, cada vez mais novas tecnologias podem ser aplicadas na medicina


Texto Original: As tendências de tecnologia para 2023 incluem avanços significativos em várias áreas. A inteligência artificial e o machine learning continuam a ser tendências importantes, com aplicações que vão desde a automação de processos até a criação de novas formas de interatividade com os usuários. A computação quântica, embora ainda em seus estágios iniciais, promete revolucionar a forma como processamos informações. Além disso, a 5G está se expandindo globalmente, proporcionando velocidades de internet mais rápidas e mais estáveis. Outras tendências incluem a realidade aumentada e virtual, que estão sendo cada vez mais utilizadas em setores como educação, entretenimento e saúde.

Resumo: Tecnologias Emergentes de TI são empregadas em diversos outros setores para desenvolvimento de aplicações web. Uma


Texto Original: A sustentabilidade nas empresas é um tema cada vez mais importante, à medida que as organizações reconhecem a necessidade de minimizar seu impacto ambiental. A implementação de práticas sustentáveis não só ajuda a proteger o meio ambiente, mas também pode resultar em economias significativas de custos e melhorar a imagem da empresa junto aos consumidores. Entre as práticas mais comuns estão a redução do consumo de energia, a reciclagem de materiais e a escolha de fornecedores que adotam práticas sustentáveis. No entanto, a transição para a sustentabilidade pode ser desafiadora e requer um compromisso a longo prazo e a participação de todas as partes interessadas da empresa.

Resumo: O ambiente é como qualquer outro meio de ser usado, não é o primeiro mas que deve ser
```

4. Sumarização de Texto usando o T5 em Português

Neste exemplo, vamos demonstrar como integrar o modelo T5 adaptado ao português para realizar sumarização de texto. Este tipo de aplicação pode ser útil para empresas que desejam automatizar a criação de resumos de documentos, artigos ou relatórios.

4.1 Carregamento do *Pipeline* de Sumarização

Vamos carregar o *pipeline* de sumarização utilizando o modelo unicamp-dl/ptt-5-base-portuguese-vocab.

```
[ ] # Carregando o pipeline de sumarização
    summarizer = pipeline(
        "summarization",
        model="rhaymison/flan-t5-portuguese-small-summarization",
        tokenizer="rhaymison/flan-t5-portuguese-small-summarization",
        device=0 if torch.cuda.is_available() else -1
    )
```

```
config.json: 100% 1.55k/1.55k [00:00<00:00, 120kB/s]
model.safetensors: 100% 308M/308M [00:12<00:00, 23.2MB/s]
generation_config.json: 100% 112/112 [00:00<00:00, 8.77kB/s]
tokenizer_config.json: 100% 20.8k/20.8k [00:00<00:00, 1.32MB/s]
tokenizer.json: 100% 2.42M/2.42M [00:01<00:00, 1.82MB/s]
special_tokens_map.json: 100% 2.54k/2.54k [00:00<00:00, 182kB/s]
```

4.2 Função para Sumarizar Texto

Vamos definir uma função que utiliza o *pipeline* de sumarização para gerar resumos de textos longos em português.

```
[ ] def summarize_text(text, summarizer, max_length=100, min_length=30):
        # Gerar o resumo utilizando o pipeline de sumarização
        summary = summarizer(text, max_length=max_length, min_length=min_
    length, do_sample=False)
```

continua

```
    # Obter o texto resumido
    summarized_text = summary[0]['summary_text']
    return summarized_text
```

4.3 Exemplos de Textos para Sumarização

Vamos definir alguns exemplos de textos longos em português e utilizar o modelo FLAN-T5 para gerar seus resumos.

```
[ ] texts = [
    """A inteligência artificial (IA) está transformando o setor de saúde de várias maneiras. Desde a análise de grandes volumes de dados de pacientes para prever doenças até a personalização de tratamentos com base em dados genéticos, a IA está revolucionando como os cuidados de saúde são prestados. Além disso, tecnologias como machine learning e deep learning estão sendo usadas para desenvolver novas drogas e identificar potenciais tratamentos para doenças complexas. No entanto, a adoção da IA na saúde também enfrenta desafios, incluindo preocupações com a privacidade dos dados dos pacientes e a necessidade de regulamentações claras para garantir o uso ético da tecnologia.""",
    """As tendências de tecnologia para 2023 incluem avanços significativos em várias áreas. A inteligência artificial e o machine learning continuam a ser tendências importantes, com aplicações que vão desde a automação de processos até a criação de novas formas de interatividade com os usuários. A computação quântica, embora ainda em seus estágios iniciais, promete revolucionar a forma como processamos informações. Além disso, a 5G está se expandindo globalmente, proporcionando velocidades de internet mais rápidas e mais estáveis. Outras tendências incluem a realidade aumentada e virtual, que estão sendo cada vez mais utilizadas em setores como educação, entretenimento e saúde.""",
    """A sustentabilidade nas empresas é um tema cada vez mais importante, à medida que as organizações reconhecem a necessidade de minimizar seu impacto ambiental. A implementação de práticas sustentáveis não só ajuda a proteger o meio ambiente, mas também pode resultar em economias significativas de custos e melhorar a imagem da empresa junto aos consumidores. Entre as práticas mais comuns estão a redução do consumo de energia, a reciclagem de materiais e a escolha de fornecedores que adotam práticas sustentáveis. No entanto, a transição para a sustentabilidade pode ser desafiadora e requer um compromisso a longo prazo e a participação de todas as partes interessadas da empresa."""
    ]

    # Gerando e exibindo resumos para cada texto
    for text in texts:
        summary = summarize_text(text, summarizer)
        print(f"Texto Original: {text}")
```

continua

```
        print(f"Resumo: {summary}")
        print()
```

```
Texto Original: A inteligência artificial (IA) está transformando o setor de saúde de várias maneiras. Desde a análise de grandes volumes de dados de pacientes para prever doenças até a personalização de tratamentos com base em dados genéticos, a IA está revolucionando como os cuidados de saúde são prestados. Além disso, tecnologias como machine learning e deep learning estão sendo usadas para desenvolver novas drogas e identificar potenciais tratamentos para doenças complexas. No entanto, a adoção da IA na saúde também enfrenta desafios, incluindo preocupações com a privacidade dos dados dos pacientes e a necessidade de regulamentações claras para garantir o uso ético da tecnologia.

Resumo: IA transforma setor de sade de várias maneiras. Desde a análise de grandes volumes de pacientes para prever doenças até a personalizaço de tratamentos com base em dados genéticos, a IA está revolucionando como os cuidados de


Texto Original: As tendências de tecnologia para 2023 incluem avanços significativos em várias áreas. A inteligência artificial e o machine learning continuam a ser tendências importantes, com aplicações que vão desde a automação de processos até a criação de novas formas de interatividade com os usuários. A computação quântica, embora ainda em seus estágios iniciais, promete revolucionar a forma como processamos informações. Além disso, a 5G está se expandindo globalmente, proporcionando velocidades de internet mais rápidas e mais estáveis. Outras tendências incluem a realidade aumentada e virtual, que estão sendo cada vez mais utilizadas em setores como educação, entretenimento e saúde.

Resumo: 5G: computaço quântica promete revolucionar a forma como processamos alertas. A 5G está se expandindo globalmente, proporcionando velocidades de internet mais rápidas e mais estáveis. Outras tendências incluem a realidade aumentada


Texto Original: A sustentabilidade nas empresas é um tema cada vez mais importante, à medida que as organizações reconhecem a necessidade de minimizar seu impacto ambiental. A implementação de práticas sustentáveis não só ajuda a proteger o meio ambiente, mas também pode resultar em economias significativas de custos e melhorar a imagem da empresa junto aos consumidores. Entre as práticas mais comuns estão a redução do consumo de energia, a reciclagem de materiais e a escolha de fornecedores que adotam práticas sustentáveis. No entanto, a transição para a sustentabilidade pode ser desafiadora e requer um compromisso a longo prazo e a participação de todas as partes interessadas da empresa.

Resumo: Sustentabilidade nas empresas: entenda o que é a medida que organizaçes reconhecem a necessidade de minimizar seu impacto ambiental. A transiço para a suposta no só ajuda a proteger o meio ambiente, mas também
```

5. Exercício

Para finalizar a seção do GPT e T5 do nosso curso fica o exercício desafio que será divido em quatro partes:

1. Utilizar algum parágrafo da sua escolha do link (https://pt.wikipedia.org/wiki/GPT-3).
2. Realizar o resumo do parágrafo escolhido com o GPT2.
3. Realizar o resumo do parágrafo escolhido com o T5.
4. Escreva seu comentários de qual foi o modelo que melhor realizou a tarefa de sumarização segundo a sua avaliação.

SAIBA MAIS...

- What is GPT? GPT-3, GPT-4, and more explained (Schulze, 2024).
- FLAN-T5 Tutorial: guide and fine-tuning (Keita, 2023).

CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Arquiteturas Profundas para Processamento de Linguagem Natural

Unidade VI

Encerramento

Unidade VI: Encerramento

Neste Microcurso, exploramos toda a evolução das arquiteturas neurais voltadas ao PLN. Começamos com RNNs, discutimos as vantagens e defeitos das arquiteturas recorrentes. Passamos pelos *Transformers* e o mecanismo de atenção. E, por fim, estudamos as arquiteturas que surgiram nos anos recentes com base em *Transformers*, passando pelo BERT e GPT e refletindo sobre como essas arquiteturas revolucionaram a forma de se resolver problemas em NLP.

Estudar o surgimento e funcionamento dessas arquiteturas tem como objetivo principal compreender como elas interagem com a linguagem natural, entendendo assim suas capacidades e limitações.

Para quem deseja ir além e estudar mais sobre as arquiteturas, procure compreender os detalhes mais minuciosos de como elas foram treinadas e as técnicas de otimização empregadas. Para isso, recomendamos as seguintes referências:

- **"*Deep Learning*",** de Ian Goodfellow, que discute a evolução das redes neurais para *Deep Learning* de forma detalhada;
- **"*Natural language processing with transformers*",** de Lewis Tunstall, que apresenta a arquitetura *Transformer* e a revolução que a arquitetura causou na área de NLP.

Além disso, sugerimos explorar suas diversas variações, como algumas que foram citadas no decorrer desse *e-book*.

Referências

ALAMMAR, Jay. The Illustrated Transformer. 2018. Disponível em: https://jalammar.github.io/illustrated-transformer/. Acesso em: 12 ago. 2024.

BAHDANAU, D.; CHO, K.; BENGIO, Y.. Neural machine translation by jointly learning to align and translate. **arXiv preprint arXiv:1409.0473**, 2014. DOI: https://doi.org/10.48550/arXiv.1409.0473.

BROWN, T. B.. Language models are few-shot learners. **NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing System**, New York, USA, 2020.

CHO, K. *et al.*. Learning phrase representations using RNN encoder-decoder for statistical machine translation. **arXiv preprint arXiv:1406.1078**, 2014. DOI: https://doi.org/10.48550/arXiv.1406.1078.

CHUNG, Hyung Won et al. Scaling instruction-finetuned language models. **Journal of Machine Learning Research**, v. 25, n. 70, p. 1-53, 2024.

DESHPANDE, Mandar. Language Modeling II: ULMFiT and ELMo. **Towards Data Science**, 2020. Disponível em: https://towardsdatascience.com/language-modelingii-ulmfit-and-elmo-d66e96ed754f. Acesso em: 12 jul. 2024.

DEVLIN, J. *et al.*. **BERT:** pre-training of deep bidirectional transformers for language understanding. Minnesota, USA: Association for Computational Linguistics, 2019.

DOS SANTOS NETO, M.V.; DA SILVA, N.F.F.; DA SILVA SOARES, A.. A survey and study impact of tweet sentiment analysis via transfer learning in low resource scenarios. **Lang Resources & Evaluation**, v. 58, p. 133-174, 2024. DOI: https://doi.org/10.1007/s10579-023-09687-8.

ELMAN, J. L.. Finding structure in time. **Cognitive Science**, San Diego, USA, 1990.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. Deep Learning. Cambridge: **MIT Press**, 2016. Disponível em: http://www.deeplearningbook.org. Acesso em: 12 nov. 2024.

GUILLOU, Pierre. Democratizando o Deep Learning. E depois? **Slideshare**, 2018. Disponível em: https://pt.slideshare.net/slideshow/democratizando-o-deep-learning-e-depois-pierre-guillou/87408000. Acesso em: 12 jul. 2024.

GRAVES, A.. **Supervised sequence labelling with recurrent neural networks**. Springer Berlin, Heidelberg, 2012.

HA, Ashley. Let's Learn About Universal Language Model Fine-Tuning (ULMFiT) Through Practical Applications. **Medium**, 2022. Disponível em: https://medium.com/@ashleyha/lets-learn-about-universal-language-model-fine-tuning-ulmfit-through-practical-applications-fea0aed2cf96. Acesso em: 27 jul. 2024.

HARVARD NLP. *The Annotated Transformer*. 2018. Disponível em: https://nlp.seas.harvard.edu/2018/04/03/attention.html. Acesso em: 12 nov. 2024.

HASHEMI-POUR, Cameron; LUTKEVICH, Ben. BERT language model. **TechTarget**, SearchEnterpriseAI, 2024. Disponível em: https://www.techtarget.com/searchenterpriseai/definition/BERT-language-model#:~:text=BERT%2C%20which%20stands%20for%20Bidirectional,calculated%20based%20upon%20their%20connection. Acesso em: 21 jul. 2024.

HOWARD, J.; RUDER, S.. Universal language model fine-tuning for text classification. ***In:* Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics,** v. 1, Long Papers, p. 328–339, Melbourne, Australia. DOI: https://doi.org/10.18653/v1/P18-1031.

HOREV, Rani. BERT Explained: State of the art language model for NLP. **Towards Data Science**, 2018. Disponível em: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270. Acesso em: 12 nov. 2024.

HUGGING FACE. *Hugging Face*. Disponível em: https://huggingface.co/. Acesso em: 12 jul. 2024.

JORDAN, M. I.. Serial order: a parallel distributed processing approach. **Advances in Psychology**, v. 121, p. 471-95, 1987. DOI: https://doi.org/10.1016/S0166-4115(97)80111-2.

JOSHI, Prateek. *Guide to Learn ELMo for Extracting Features from Text*. Analytics Vidhya. **Analytics Vidhya**, 2024. Disponível em: https://www.analyticsvidhya.com/blog/2019/03/learn-to-use-elmo-to-extract-features-from-text/. Acesso em: 12 nov. 2024.

KEITA, Zoumana. FLAN-T5 Tutorial: Guide and Fine-Tuning. **DataCamp**, 2023. Disponível em: https://www.datacamp.com/tutorial/flan-t5-tutorial. Acesso em: 12 nov. 2024.

KLEIN, Guillaume *et al*. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In: **Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)**, 2017. Disponível em: https://doi.org/10.18653/v1/P17-4012. Acesso em: 12 nov. 2024.

PETERS, M. E.. *et al*. Deep contextualized word representations. **New Orleans: Association for Computational Linguistics**, 2018.

OINKINA; HAKYLL. Understanding LSTM Networks. **Colah's blog**, 2015. Disponível em: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Acesso em: 12 ago. 2024.

RADFORD, A. *et al.*. Improving language understanding by generative pre-training. Preprint, San Francisco: **OpenAI**, 2018. Disponível em: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. Acesso em: 09 nov. 2024.

RAFFEL, Colin et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. **Journal of Machine Learning Research**, v. 21, p. 1-67, 2020.

SAK, Hasim. *et al*. Google voice search: faster and more accurate. **Research Blog**, [Internet], Google Speech Team, 2015. Disponível em: https://research.google/blog/google-voice-search-faster-and-more-accurate/. Acesso em: 09 nov. 2024.

SCHULZE, Jessica. What Is GPT? GPT-3, GPT-4, and More Explained. **Coursera**, 2024. Disponível em: https://www.coursera.org/articles/what-is-gpt. Acesso em: 12 ago. 2024.

THE TENSORFLOW AUTHORS. Neural machine translation with a Transformer and Keras. **TensorFlow**, 2024. Disponível em: https://www.tensorflow.org/text/tutorials/transformer. Acesso em: 27 jul. 2024.

TSANG, Sik-Ho. *Review — ELMo: Deep Contextualized Word Representations*. 2022. Disponível em: https://sh-tsang.medium.com/review-elmo-deep-contextualized-word-representations-8eb1e58cd25c. Acesso em: 12 ago. 2024.

VASWANI, A. *et al.*. Attention is all you need. Advances in neural information processing systems. v. 30. **31st Conference on Neural Information Processing Systems** (NIPS 2017), Long Beach, EUA, 2017. Disponível em: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. Acesso em: 09 nov. 2024.

VINYALS, O. *et al*. Show and tell: A neural image caption generator. **Proceedings of the IEEE conference on computer vision and pattern recognition**, San Francisco, USA, 1996. Disponível em: https://doi.org/10.48550/arXiv.1411.4555. Acesso em: 09 nov. 2024.

AKCIT

CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

AKCIT CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

CEIA CENTRO DE EXCELÊNCIA EM INTELIGÊNCIA ARTIFICIAL

UFG UNIVERSIDADE FEDERAL DE GOIÁS

EMBRAPII

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO

GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

SEBRAE

GOVERNO DE GOIÁS O ESTADO QUE DÁ CERTO

FAPEG Fundação de Amparo à Pesquisa do Estado de Goiás

SOBRE O E-BOOK

Tipografia: Montserrat

Publicação: Cegraf UFG
Câmpus Samambaia, Goiânia - Goiás. Brasil. CEP 74690-900
Fone: (62) 3521-1358
https://cegraf.ufg.br