

2. Introdução a Modelos Transformers

Davi Bezerra Barros

1. O que são Transformers

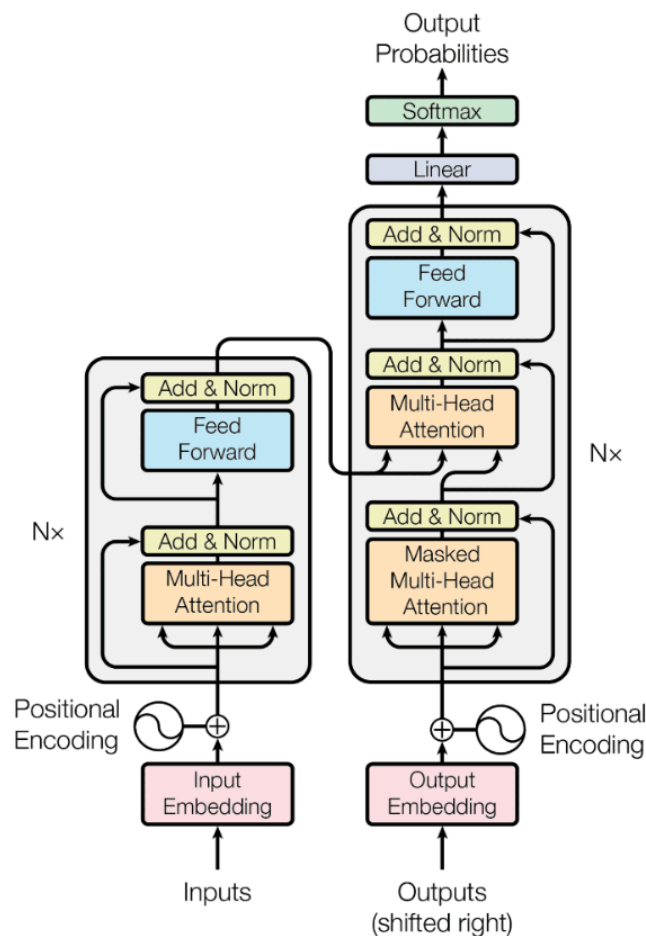
O Transformer é um modelo de rede neural proposto pelo Google em 2017, para resolver problemas de tradução neural; tarefas que transformam uma sequência de entrada em uma sequência de saída, que até o momento, eram realizadas pelas RNNs (Recurrent Neural Networks).

As RNNs não são tão eficientes para resolver tarefas de processamento de linguagem natural pois processam dados sequencialmente, e não permitem a utilização de GPUs modernas para processamento paralelo, o que torna o seu treinamento muito lento. Além disso, quanto maior a janela de contexto dos dados de entrada, menor a sua eficiência devido à distância entre os elementos relevantes, causando perda de informação.

A mudança na utilização de modelos RNNs para modelos Transformer em tarefas de NLP se baseia na solução destas duas limitações. Ao substituir os mecanismos de **recorrência** por mecanismos de **auto-atenção**, modelo Transformer é capaz de ponderar a importância de diferentes palavras em relação ao contexto da sentença e também pode processar todos os tokens de uma única vez, paralelizando o processamento e acelerando o tempo de treinamento em grandes conjuntos de dados.

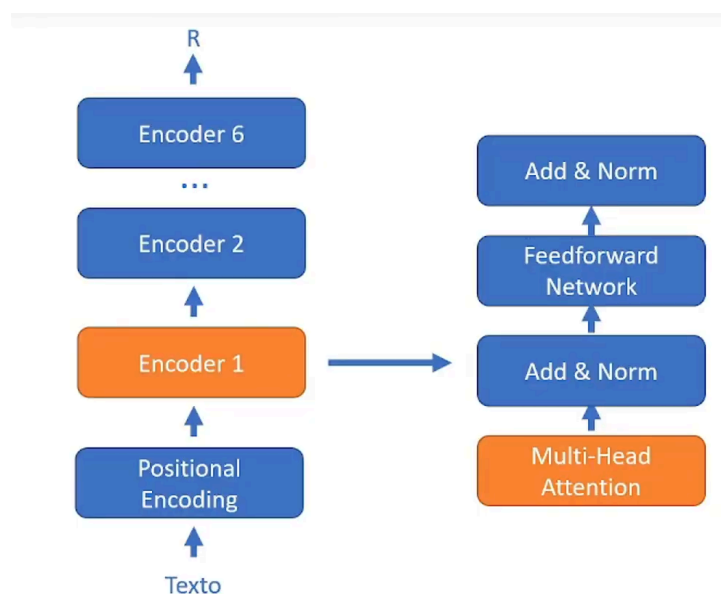
2. Funcionamento

Os Transformers consistem, primariamente, de **encoders** e **decoders** empilhados. Cada camada de encoder/decoder possui **mecanismos de auto-atenção**, redes **feed-forward**, **conexões residuais** e camadas de normalização.



2.1. Encoder

O encoder processa os dados de entrada em paralelo, e gera uma representação contextualizada dos tokens. A primeira etapa é a conversão dos tokens de entrada em vetores usando camadas de embedding, capturando seu significado semântico.



O encoder é formado por 6 blocos sequenciais, onde o primeiro recebe os vetores de embedding e passa ao bloco seguinte. Os blocos possuem o seguintes sub-componentes:

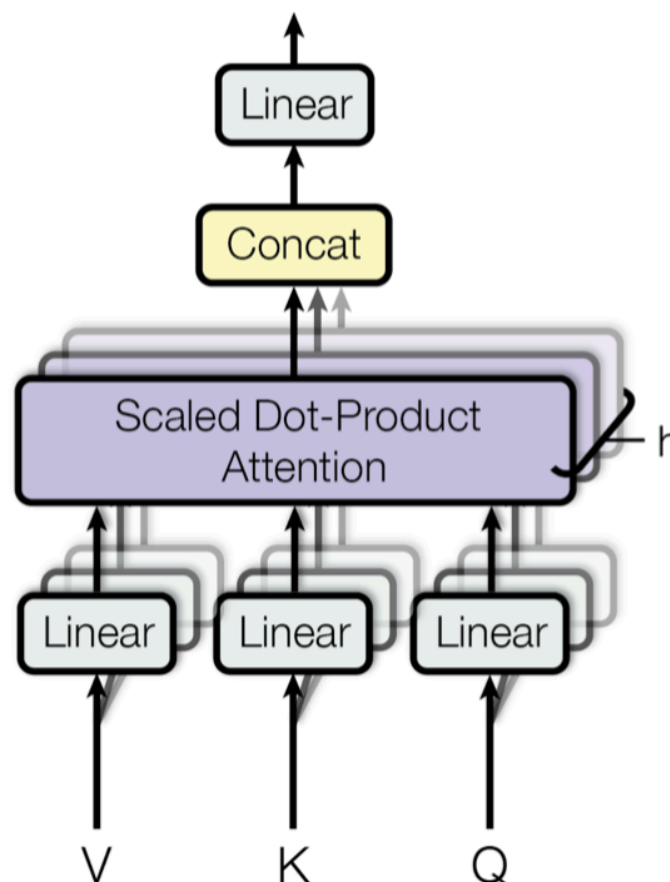
Positional Encoder

Como o transformer processa todos os tokens em paralelo e não sequencialmente como as RNNs, ele não possui informação sobre a posição de cada token na sentença. O **positional encoder** então adiciona aos embeddings as informações sobre a posição de cada token na entrada. Para isso, foram implementadas funções de semelhança senoidal e cossenoidal para codificar as posições e determinar quais palavras são próximas umas das outras. Os positional embeddings também podem ser aprendidos no treinamento via backpropagation.

Self-Attention e Multi-Head Attention

Self-Attention(Auto-Atenção) é o mecanismo que faz com que os tokens interajam com os outros tokens da sentença, permitindo que o modelo capture informações contextuais. Isso é feito através da representação dos tokens em três vetores:

- **Q (Query):** Vetor que representa a palavra ou token atual da sequência no mecanismo de Atenção.
- **K (Key):** Vetor que representa todos os outros tokens da sequência no mecanismo de atenção.
- **V (Value):** Vetor de valores associados às keys, usado para construir a saída da camada de atenção.



As etapas das operações no bloco de atenção são as seguintes:

1. Criação dos vetores Q, K e V: Os vetores são obtidos ao multiplicar os embeddings por matrizes de peso W^q , W^k e W^v , que em seguida são transformados linearmente.

2. Cálculo dos Escores de atenção: Após os vetores passarem pela camada linear, é feito um *produto escalar* entre os vetores Q e K, gerando uma matriz de escore. Esta matriz define quanta ênfase cada palavra deve ter sobre outras palavras, e cada palavra tem um escore em relação às outras palavras da sentença.

3. Escalonamento dos escores: A matriz de escores é escalonada por um fator de $1/\sqrt{dk}$, onde dk é a dimensão dos vetores Q e K . Isso é feito para estabilizar os gradientes durante o treinamento.

4. Aplicação de Softmax: Os escores são normalizados com a aplicação de uma função softmax, para obter os pesos de atenção, onde maiores escores se sobrepõem aos menores escores.

5. Combinação com o vetor V : O vetor resultante da função softmax é multiplicado com o vetor *value*, o ponderando com os pesos de atenção para formar a saída final do self-attention

Cada encoder é seguida de uma camada de normalização, e as saídas de cada sub-camada é adicionada à sua entrada, gerando uma conexão residual. Após 6 camadas (no transformer original), os dados passam por uma **Feed Forward Network** antes de ir para o decoder.

O primeiro módulo de self-attention processa toda a sentença de uma única vez, aplicando as operações de atenção em paralelo, em partes diferentes da sequência. Isso é chamado de **Multi-Head-Attention**.

2.2. Decoder

O decoder é responsável por gerar a sequência de saída token por token, de forma **autoregressiva**. Sua estrutura é semelhante à do encoder:

Máscara de Atenção (Masked Self-Attention):

Impede que o modelo veja tokens futuros durante o treinamento, garantindo que a previsão de cada token dependa apenas dos tokens anteriores.

Atenção Encoder-Decoder:

Conecta o decoder ao encoder, permitindo que o modelo focalize partes relevantes da entrada durante a geração. Usa as keys e values do encoder, mas as queries vêm do decoder.

Camadas Feed-Forward e Normalização:

Assim como no encoder, cada subcamada é seguida por normalização e conexões residuais para facilitar o treinamento.

3.0. BERT: Bidirectional Encoder Representation from Transformers

O BERT revolucionou o NLP ao pré-treinar um Transformer **bidirecional**, que captura contexto à esquerda e à direita de cada token. Diferente de modelos autoregressivos como o GPT, ele usa **apenas o encoder** e é otimizado para tarefas de compreensão.

3.1. Variantes

Como o Bert é um modelo transformer open-source, diversos desenvolvedores tiveram acesso a seu código e o modificaram de acordo com suas necessidades, dando origem a variações do BERT, pré treinados para tasks específicas. Abaixo estão algumas das variedades criadas.

Albert

RoBERTa: "Robusto Optimized BERT Approach", é uma variante do BERT criada pela META e treinada com um dataset 10 vezes maior que o BERT original. Sua arquitetura utiliza mascaramento dinâmico ao invés de estático, permitindo que o modelo aprendesse representações mais robustas.

ALBERT: "A Lite BERT" tem sua quantidade de parâmetros reduzida ao compartilhar pesos entre as camadas, otimizando o tempo de treinamento.

DistilBert: Variante criada para tornar o BERT mais acessível, ao torná-lo menor, mais leve e rápido. O DistilBert utiliza técnicas de destilação de conhecimento durante o pré-treino para reduzir seu tamanho em 40%, retraindo mais de 90% da sua capacidade.

ELECTRA: 'Efficiently Learning an Encoder that Classifies Token Replacement Accurately' é uma variante do BERT que aplica uma técnica de detecção de tokens substituídos para treinar de forma mais eficiente

4.0. Utilizando modelos pré treinados

Nesta seção do curso, o professor apresenta o HuggingFace, uma plataforma onde é possível encontrar modelos pré treinados para várias tarefas diferentes, de modelos mais simples a LLMs, que podem ser utilizados através do *pipeline* da biblioteca **transformers**. Os modelos explorados foram:

Question answering: Modelo baseado no BERT, treinado para responder perguntas a respeito de um texto ou corpus.

```
[ ] 1 qea = pipeline("question-answering",model="pierregruillou/bert-large-cased-squad-v1.1")

[ ] 1 texto = "Bruce Lee[b] (nascido Lee Jun Fan;[c] 27 de novembro de 1940 - 20 de julho de 1971) foi um ator, cineasta e filósofo chinês-americano."
    2 pergunta = "quem foi bruce lee?"
    3 resposta = qea(question=pergunta, context=texto)
    4 print("Pergunta: ", pergunta)
    5 print("Resposta:",resposta['answer'])
    6 print("Score:", resposta['score'])
```

➡ Pergunta: quem foi bruce lee?
Resposta: artista marcial, ator, cineasta e filósofo
Score: 0.22302116453647614

Preenchimento de lacunas:

Modelo baseado no BERT, treinado para preencher informações faltantes em textos e sentenças.

```
[ ] 1 #pipeline
    2 mascarar = pipeline("fill-mask",model="neuralmind/bert-base-portuguese-cased") # devo passar o tipo de tarefa a ser executada

[ ] 1 texto = mascarar("Existe uma chance do copo cair no [MASK]")
    2 for x in range(len(texto)):
    3     print(texto[x])
```

➡ {'score': 0.5944021344184875, 'token': 8105, 'token_str': 'chão', 'sequence': 'Existe uma chance do copo cair no chão'}
{'score': 0.04544356092810631, 'token': 2187, 'token_str': 'rio', 'sequence': 'Existe uma chance do copo cair no rio'}
{'score': 0.04219291731715202, 'token': 528, 'token_str': 'mar', 'sequence': 'Existe uma chance do copo cair no mar'}
{'score': 0.03341996297240257, 'token': 4848, 'token_str': 'fogo', 'sequence': 'Existe uma chance do copo cair no fogo'}
{'score': 0.02256273478269577, 'token': 14575, 'token_str': 'lixo', 'sequence': 'Existe uma chance do copo cair no lixo'}

Aplicação de resumos: Tarefa onde o modelo deve criar um resumo de um texto. Nesta tarefa o modelo não foi especificado, ficando a critério do pipeline escolher um modelo adequado.

```
[ ] 1 summarize = pipeline("summarization")
```

 **Mostrar saída oculta**

```
[ ] 1 text = """Caio Júlio César[a] (eGaius Julius Caesar[a] (12 July 100 BC - 15 March 44 BC) was a Roman general and :
2
3 In 60 BC, Caesar, Crassus, and Pompey formed the First Triumvirate, an informal political alliance that dominated
4
5 After assuming control of government and pardoning many of his enemies, Caesar set upon vigorous reform and build:
6
7 Caesar was an accomplished author and historian; much of his life is known from his own accounts of his military
8
9 """
```

```
1 resumo = summarize(text, max_length=50, min_length=50)
2 print(resumo)
```

 'summary_text': ' Julius Caesar was a Roman general and statesman . He was a member of the First Triumvirate . He led the

Geração de texto: Pra esta task, foi utilizado um modelo GPT2, treinado para gerar textos na língua portuguesa.

▼ Geração de Texto

```
[ ] 1 gerador = pipeline("text-generation", model="pierreguillou/gpt2-small-portuguese")
```

```
[ ] 1 texto="em sentido estrito, a ciência refere-se ao sistema de adquirir conhecimento baseado no método científico"
2 result = gerador(texto, do_sample=True)
3 print(result)
```


 ico ou na experiência científica, levando ao desenvolvimento da base do conhecimento. O conhecimento pode ou não ser esta

Aplicação com OpenAI: Nesta seção o professor ensina a gerar uma chave para a API da OpenAI, e ter acesso à sua família de modelos pré treinados. Os modelos da OpenAI, diferentemente dos modelos do HuggingFace, não são executados localmente, tendo um custo financeiro tanto para utilização quanto para fine-tuning, que varia conforme o modelo escolhido.

```
[ ] 1 from openai import OpenAI
```

```
[ ] 1 client = OpenAI(api_key="sk-proj-oIpLFe_irRy-XDrbDusGWWDpMi11HbgJyWzXex1CKkV5jIqFssqdv1hjV08JZI-LJ0
2 model = "gpt-3.5-turbo-0301"
```

```
1 response = client.chat.completions.create(
2     model = model,
3     messages = [
4         {"role": "user", "content": "Tell me a joke"}
5     ]
6 )
```

 **NotFoundError** Traceback (most recent call last)
<ipython-input-6-5aa67c67625c> in <cell line: 0>()
----> 1 response = client.chat.completions.create(
2 model = model,
3 messages = [
4 {"role": "user", "content": "Tell me a joke"}
5]
6)

↕ 4 frames ↕

```
/usr/local/lib/python3.11/dist-packages/openai/_base_client.py in _request(self, cast_to, options,
retries_taken, stream, stream_cls)
1021
1022     log.debug("Re-raising status error")
-> 1023     raise self._make_status_error_from_response(err.response) from None
1024
1025     return self._process_response(
```

Infelizmente a versão gratuita do GPT não garante acesso à sua API, e não pude testar os serviços da OpenAI. Apesar disso, o aprendizado é válido para futuras oportunidades de trabalhar com estes modelos.

5.0 Conclusão

Transformers redefiniram o NLP ao combinar paralelismo, atenção contextual e escalabilidade. Com frameworks como Hugging Face, qualquer desenvolvedor pode integrar modelos de última geração em aplicações reais, desde chatbots até sistemas de recomendação, causando uma verdadeira democratização do acesso aos modelos de rede neural.