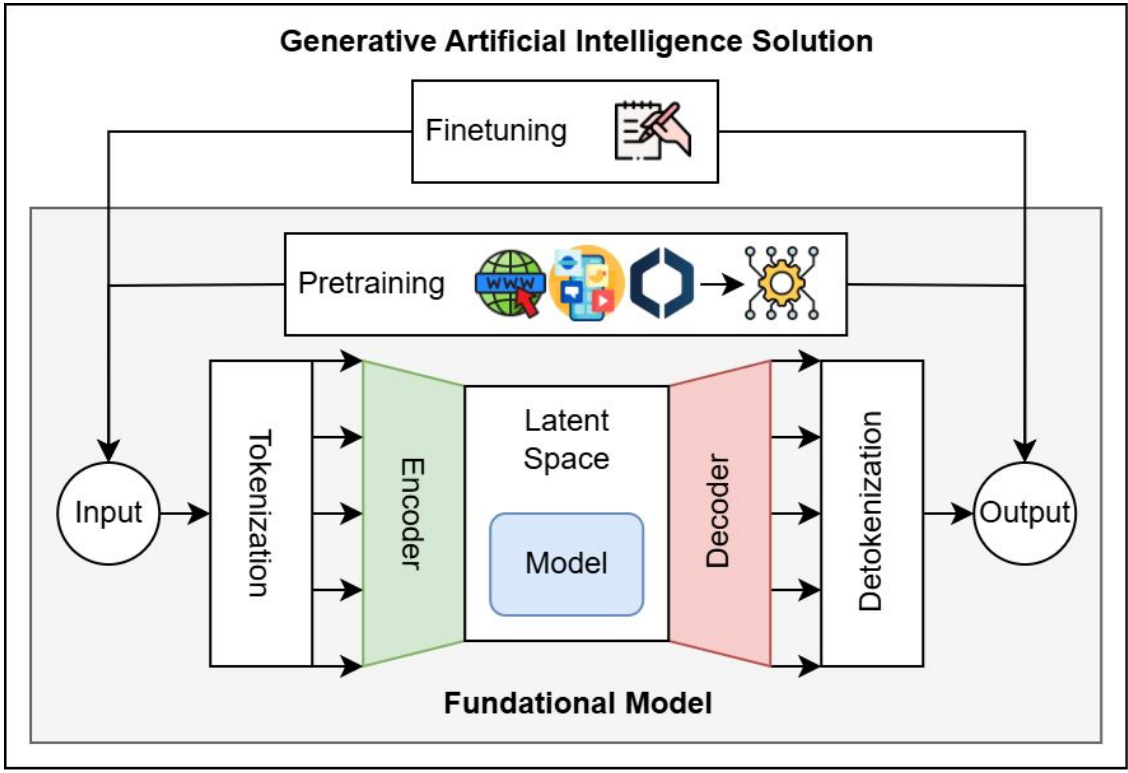
Inteligência Artificial Generativa Parte 2: Grandes Modelos de Linguagem

Thiago Brandenburg
2025/1

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Revisão

- Aula passada construímos um esquema de como funciona modelos generativos em alto nível.
- Nesta aula, iremos abordar a mesma estrutura, com foco em **Grandes Modelos de Linguagem (Large Language Models, sigla LLM)**

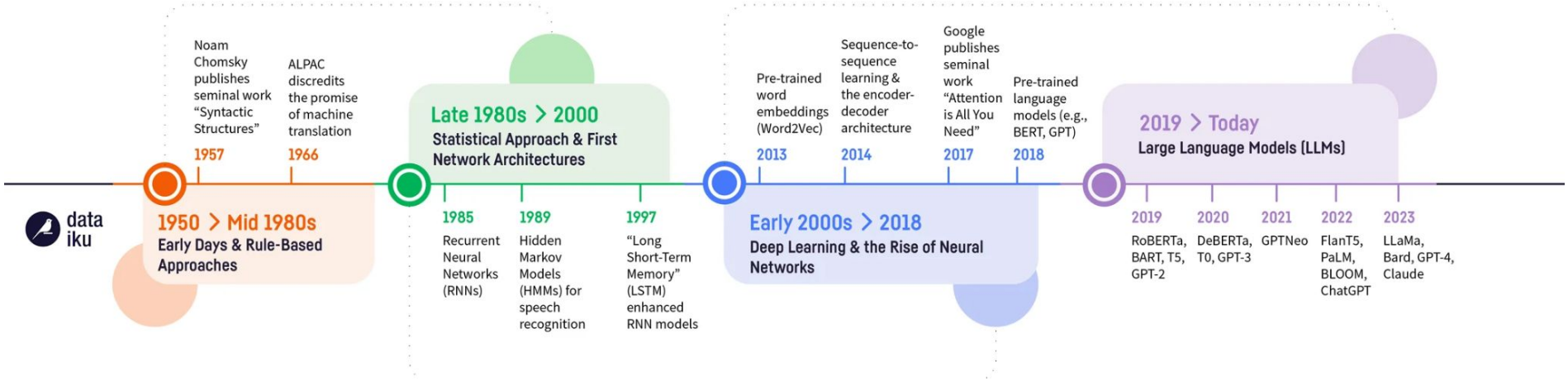


UDESC

Processamento de Linguagem Natural (PLN)

- Se considera o GPT1 (2018) o primeiro LLM, no entanto, os fundamentos explorados por ele são muito mais antigos, provenientes de uma sub-área da computação denominada **Processamento de Linguagem Natural (PLN).**

- PLN Inclui reconhecimento de fala, classificação de texto, compreensão de linguagem natural e geração de linguagem natural.

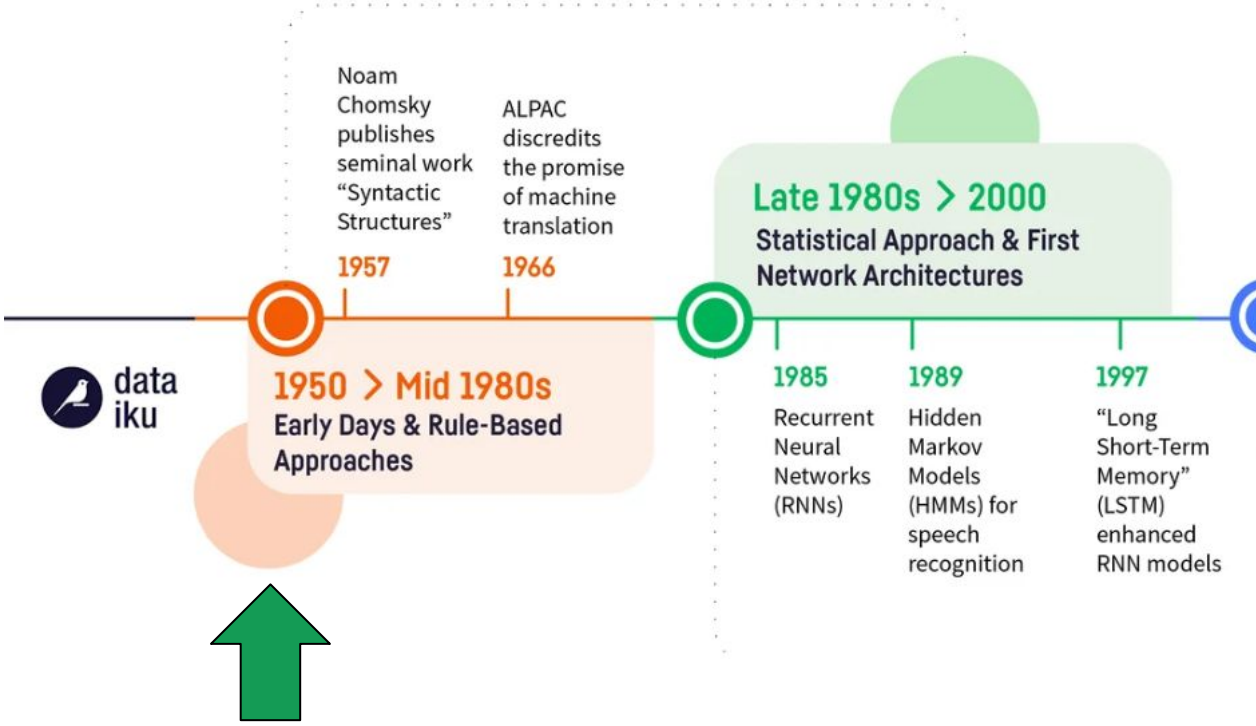


UDESC

Anos 1950: Início

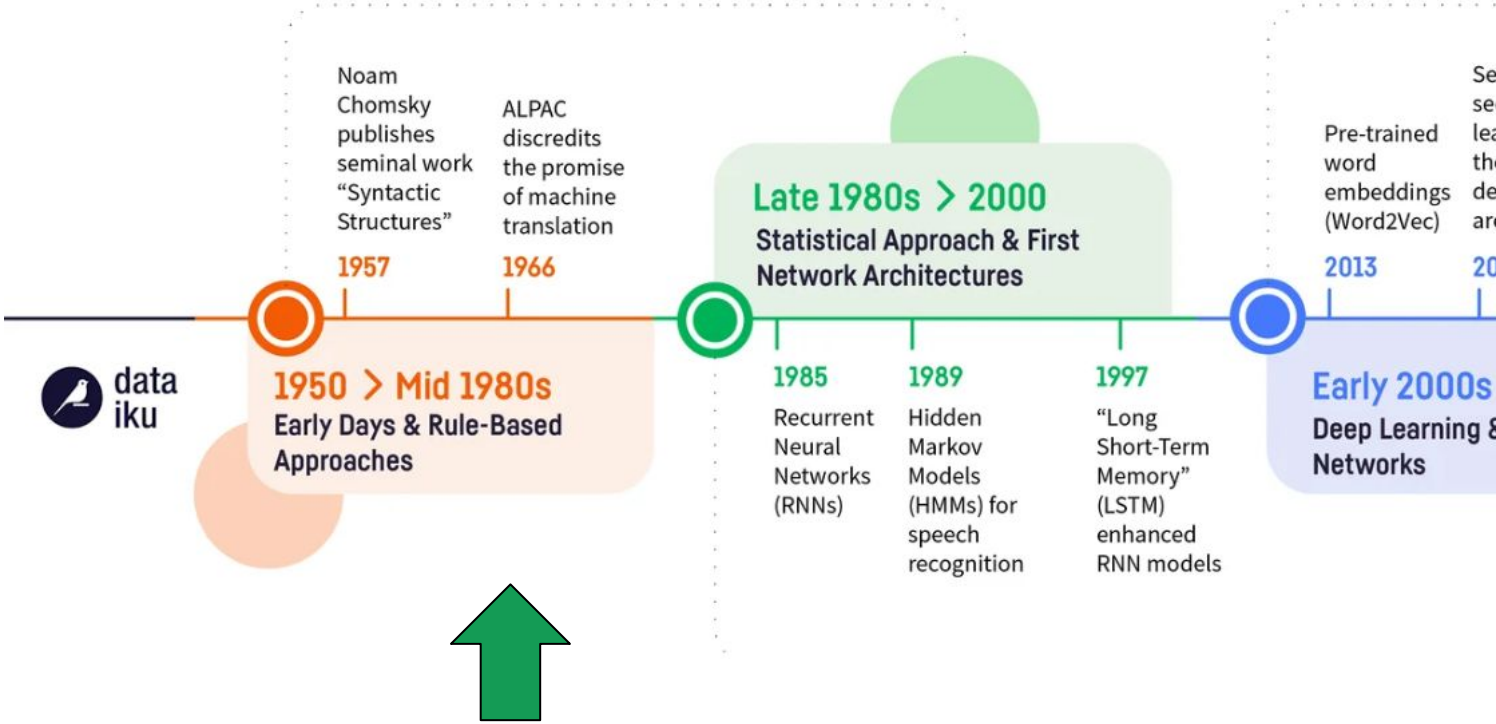
- Entusiasmo em relação a inteligência artificial.
  - Guerra Fria: Necessidade de traduzir textos de russo para inglês sistem
  - Ideia de que modelos seriam capazes de conversar com seres humanos
  - Representação da sintaxe das linguagens em estruturas interpretáveis p

- Primeiros chatbots:
  - SHRDLU, ELIZA.
  - Baseados em regras simples.



Anos 1950: Início

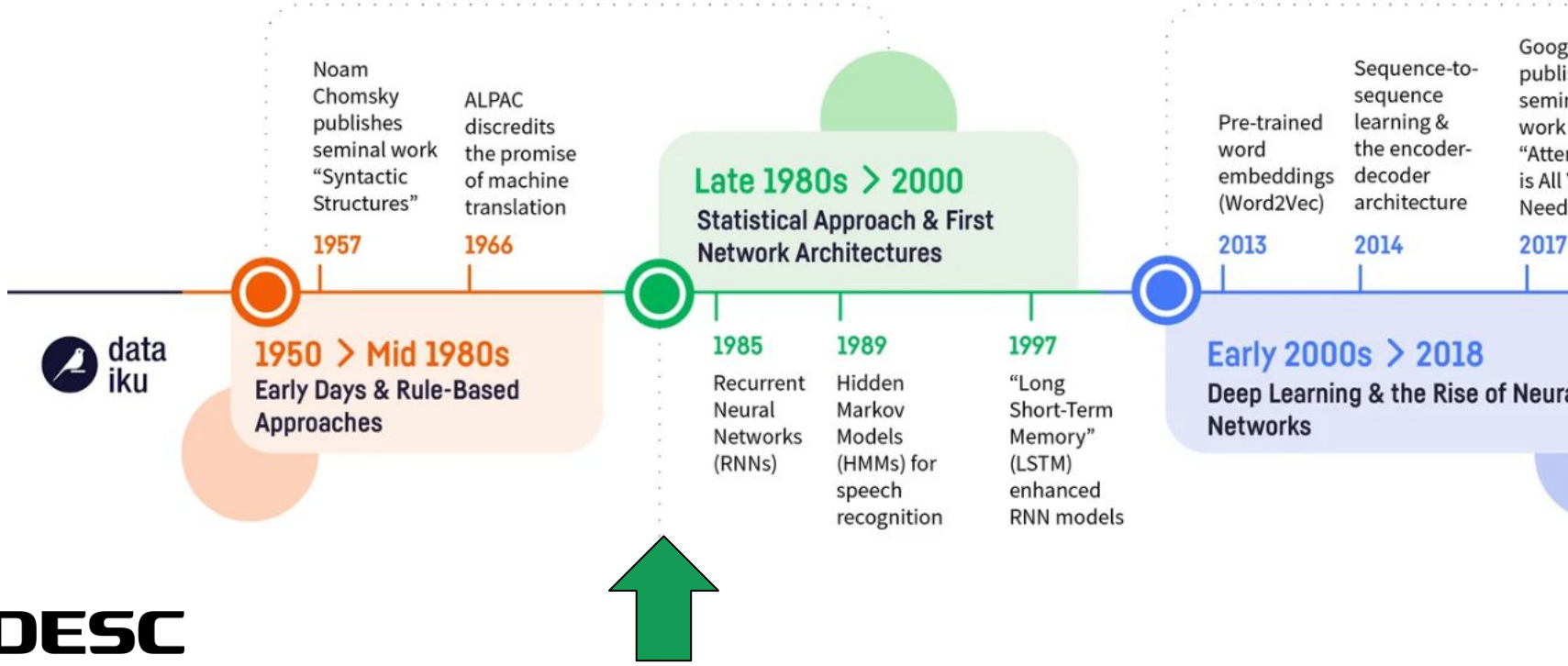
- Em 1964, é organizado o Automatic Language Processing Advisory Committee (ALPAC) pelo governo dos Estados Unidos para avaliar os investimentos do governo na área de tradução por máquina.
  - Em 1966 foi emitido o relatório:
    - Concluiu que ainda não era viável sistematizar tradução por computadores.
    - Congelou investimentos na área de PLN e IA por quase 20 anos (Inverno da IA).



Anos 1980: Modelos estatísticos

- Em meados dos anos 80, avanços na capacidade dos computadores permitiram o ressurgimento da pesquisa em PLNs, principalmente dentro da IBM.

- Modelos de linguagem deixam de ser baseados em regras, e viram **estatísticos**.

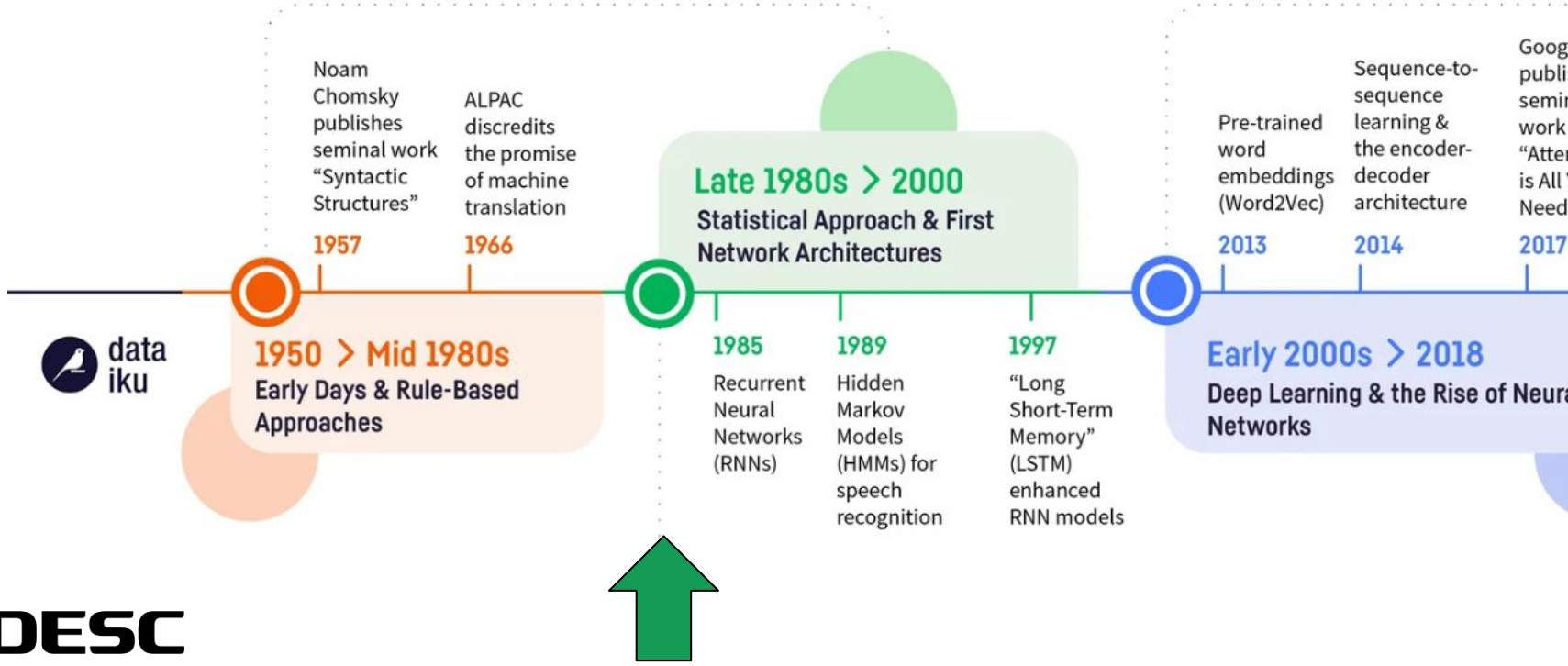


UDESC

Anos 1980: Modelos estatísticos

- Primeiros modelos **autoregressivos** linguagem treinados com **redes neurais artificiais**:
  - Prevem recursivamente a próxima palavra.
  - Janela de palavras anteriores como entrada.
  - Exemplo: Recurrent Neural Networks (RNN) e Long short-term memory (LSTMs).

- Limitações: Falta de dados, processamento

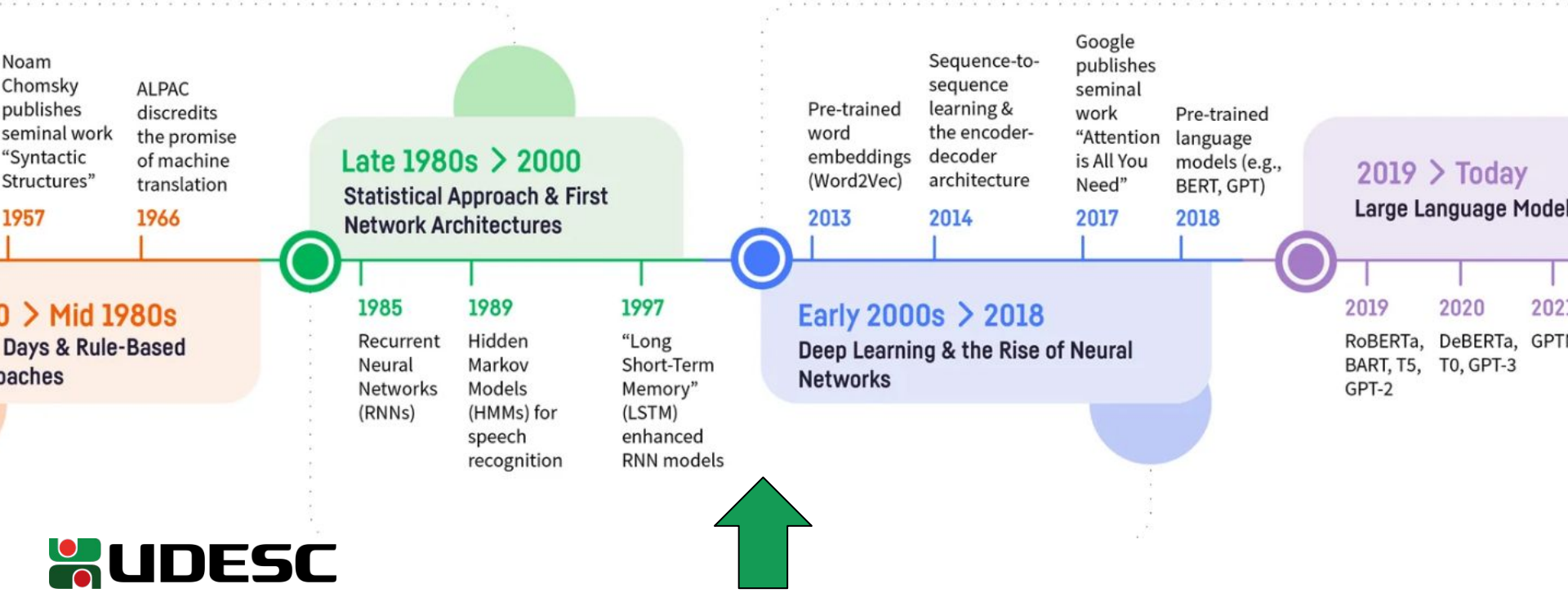


UDESC

Anos 2000

- A internet:
  - Quantidade massiva de textos para treino.
  - Soluções comerciais em web (Google Translate).
  - Ascensão de modelos estatísticos.

- Limitações: perda de contexto para textos longos, processamento.

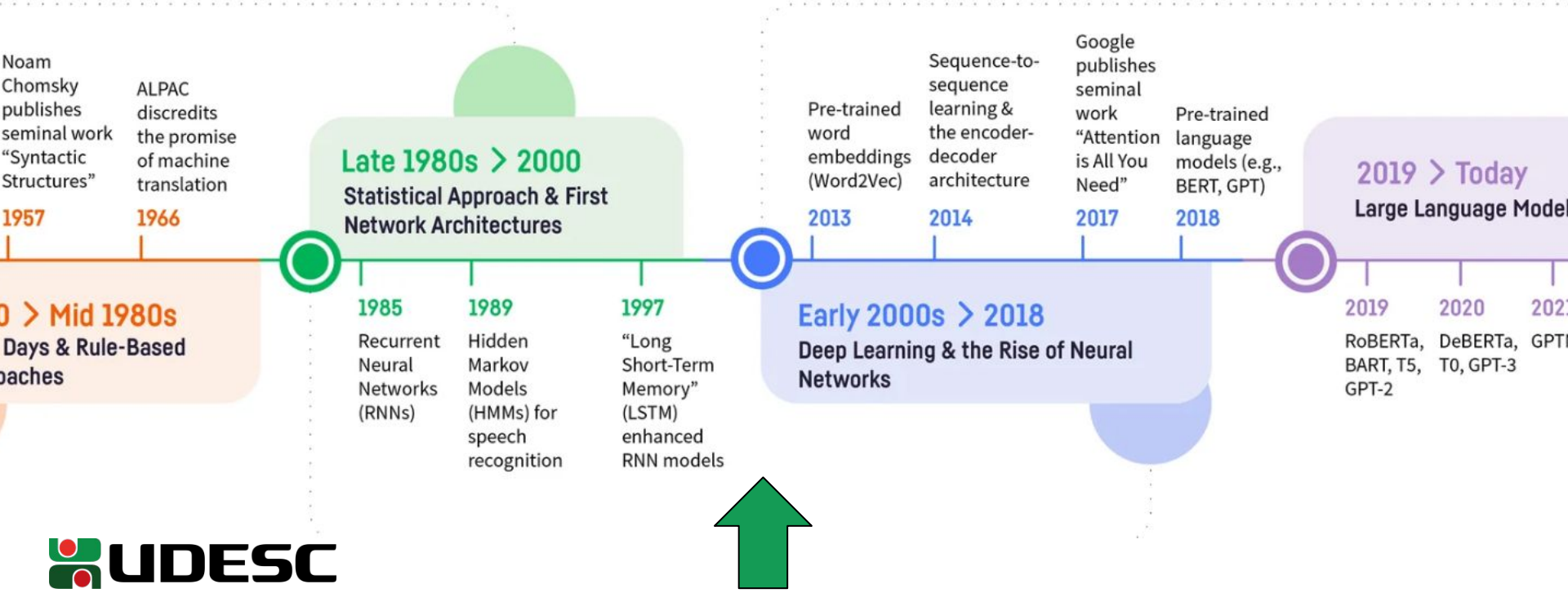


UDESC

Anos 2010

- Surge o conceito de **atenção** para solucionar o problema de perda de contexto.
  - A predição tem acesso a informações anteriores a janela.
  - Mecanismos de atenção são inseridos em RNNs (seq2seq, 2014) e em LSTMs (Google Neural Machine Translation, 2016).

- Problema: Processamento

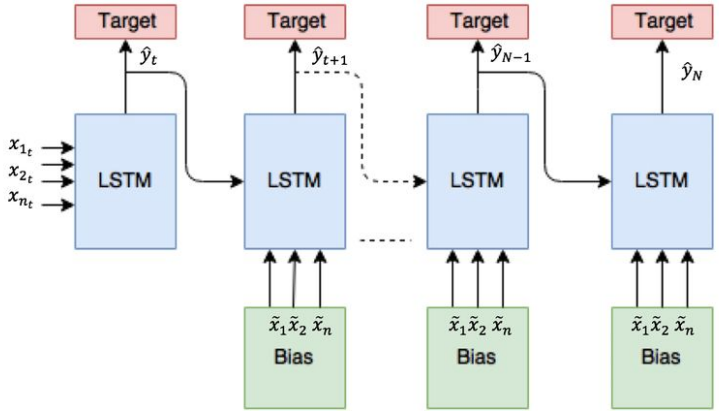


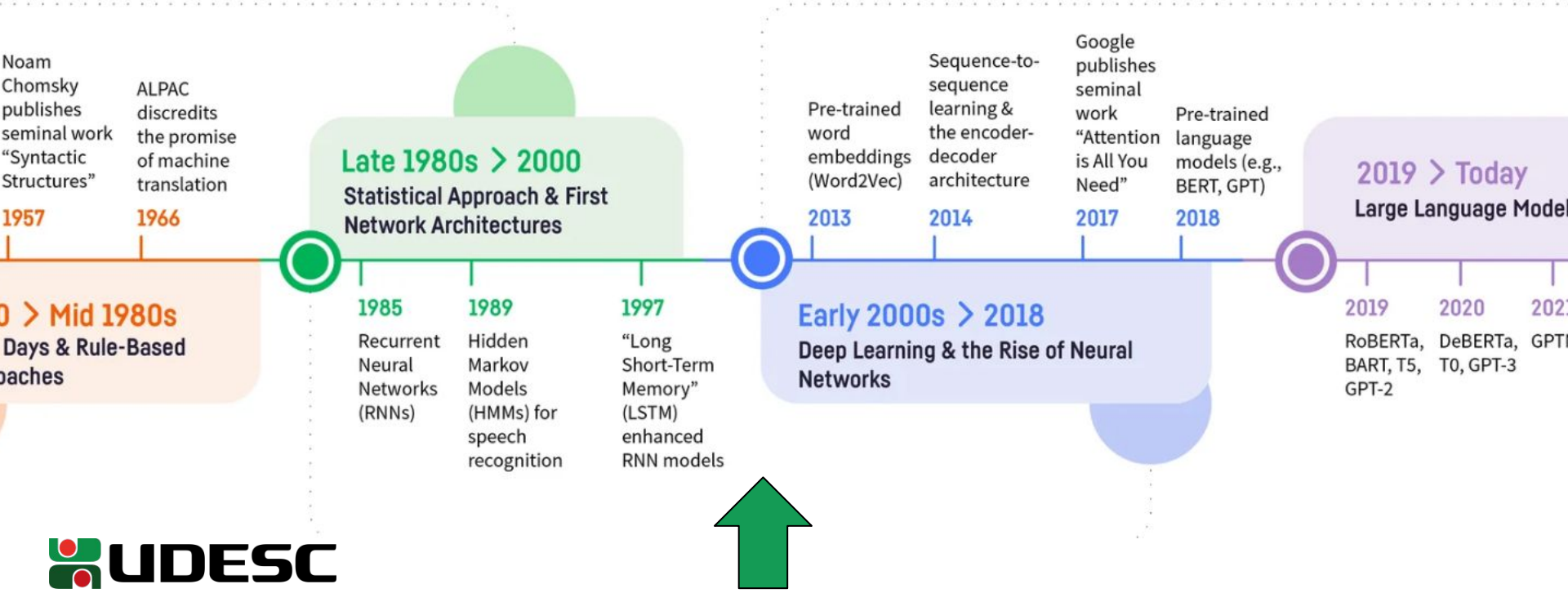
UDESC

Anos 2010

- Computação em **GPU** agora permite computação em larga escala de algoritmos paralelos.

- Algoritmos baseados em janela são sequenciais
  - Dificuldade de paralelização.
  - Como resolver?



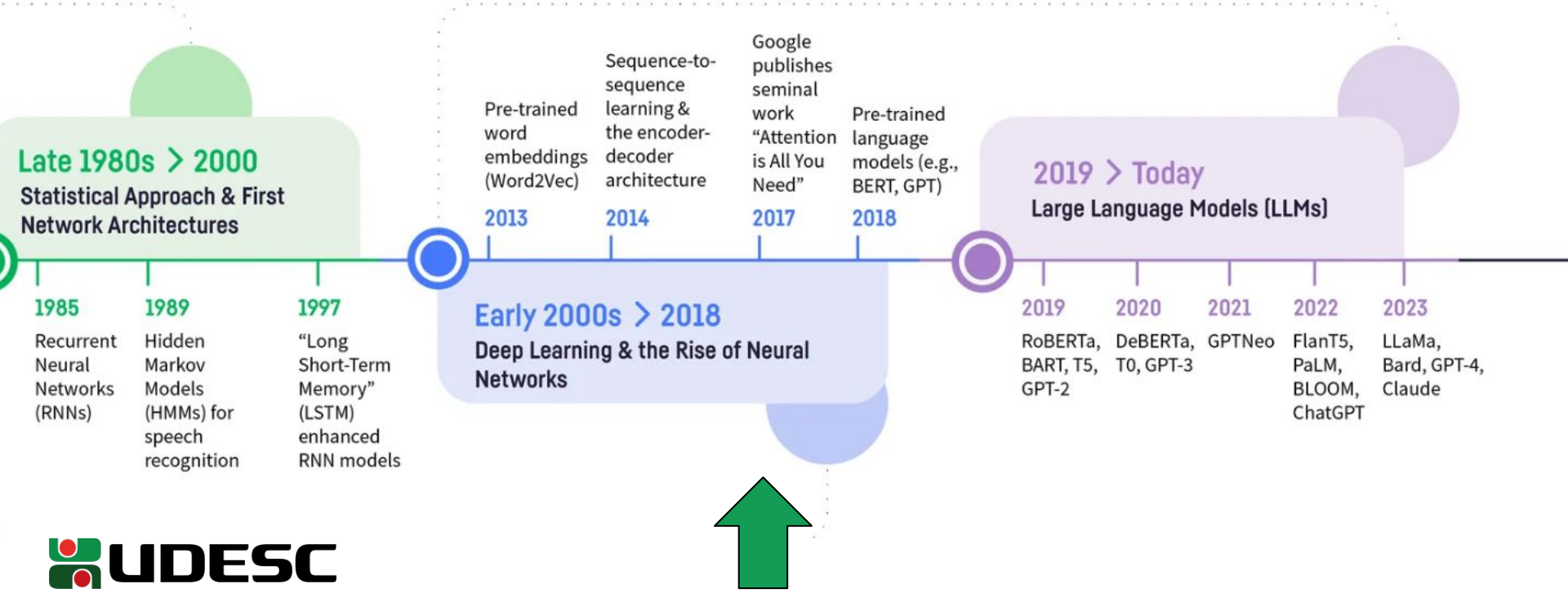


UDESC

2017: Attention is all you need

- Em 2017, a OpenAI lança o artigo “Attention is all you need”, revelando a **arquitetura transformer**.
  - Sem janelas, toda a entrada é processada de uma vez (paralelizável)
  - Contexto é preservado por meio de matrizes (self-attention)

- Modelos na arquitetura transformer são perceptivelmente mais inteligentes que de arquiteturas anteriores.

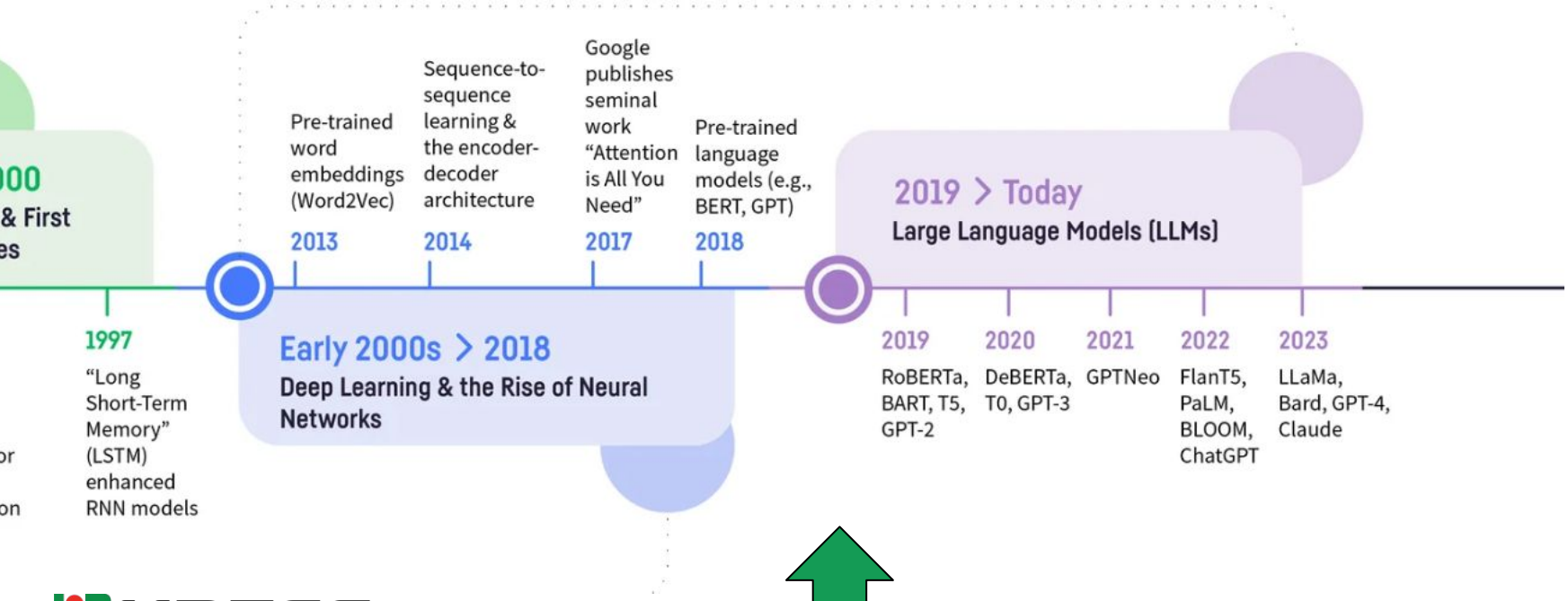


UDESC

Atualidade:

- Modelos generativos de texto baseados em Transformers dominam a área
  - GPT-4, Llama, Gemini, Deepseek V3, etc.

- Devido a importância na área, estudaremos a arquitetura do **Pre-Trained Generative Transformer (GPT)**



Revisão

1. **Inputs e outputs.**
2. **Tokenização.**
3. **Embedding.**
4. **Modelo**

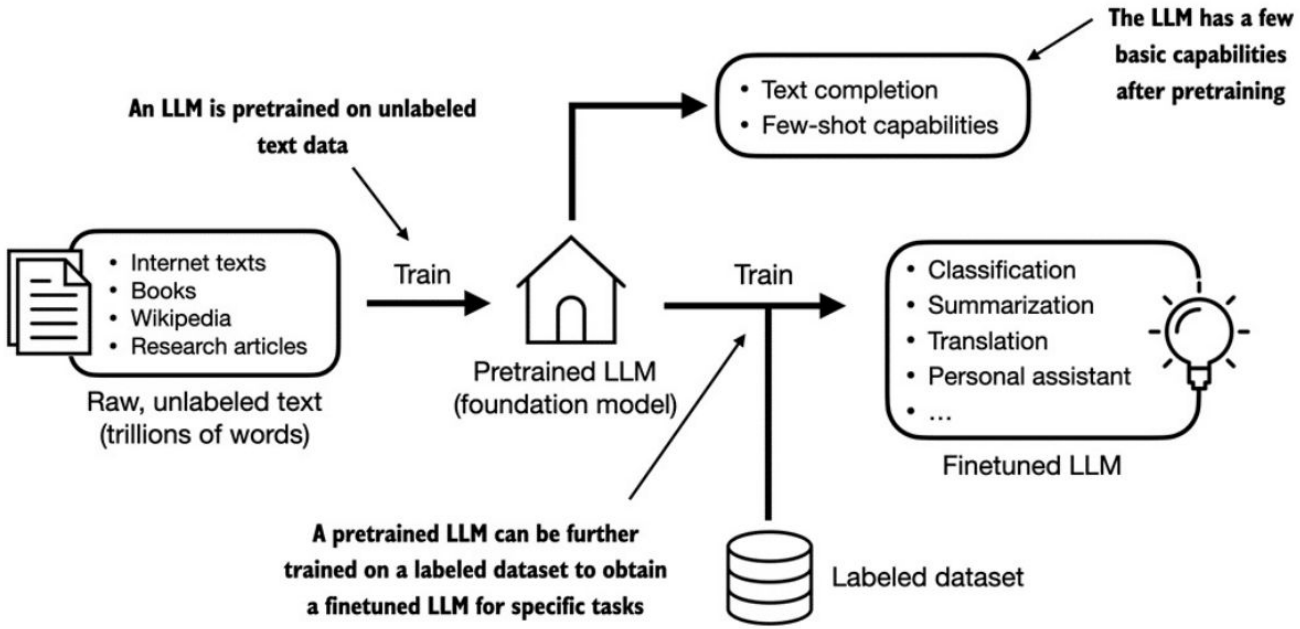
1. **Inputs e Outputs (IO):**
   a. Dados de texto.
   b. Autoregressão.
   c. Amostragem por janela
   d. Camada de Entrada: janela de tokens.
   e. Camada de Saída: Verossimilhança
   f. Métrica: Cross-entropia.
2. **Tokenização:**
   a. Byte-Pair tokenization
3. **Embedding**:
   a. Embedding de Tokenização.
   b. Embedding de posição.
4. **Modelo**:
   a. Transformer.
   b. Decoder-only.
   c. Encoder-only.
   d. Mecanismo de Atenção.

Índice

1. IO: Dados

- Sabemos que modelos generativos são treinados em duas etapas:
  - **Pre-training**: Grande base não estruturada, rotulagem auto-supervisionada, capacidades genéricas.
  - **Fine-tunning**: Pequena base rotulada, tarefas específicas.

- É preciso adotar uma estratégia para rotular os dados no pre-training do GPT.



UDESC

1. IO: Dados

- Sabemos que modelos generativos são treinados em duas etapas:
  - **Pre-training**: Grande base não estruturada, rotulagem auto-supervisionada, capacidades genéricas.
  - **Fine-tunning**: Pequena base rotulada, tarefas específicas.

- É preciso adotar uma estratégia para rotular os dados no pre-training do GPT.

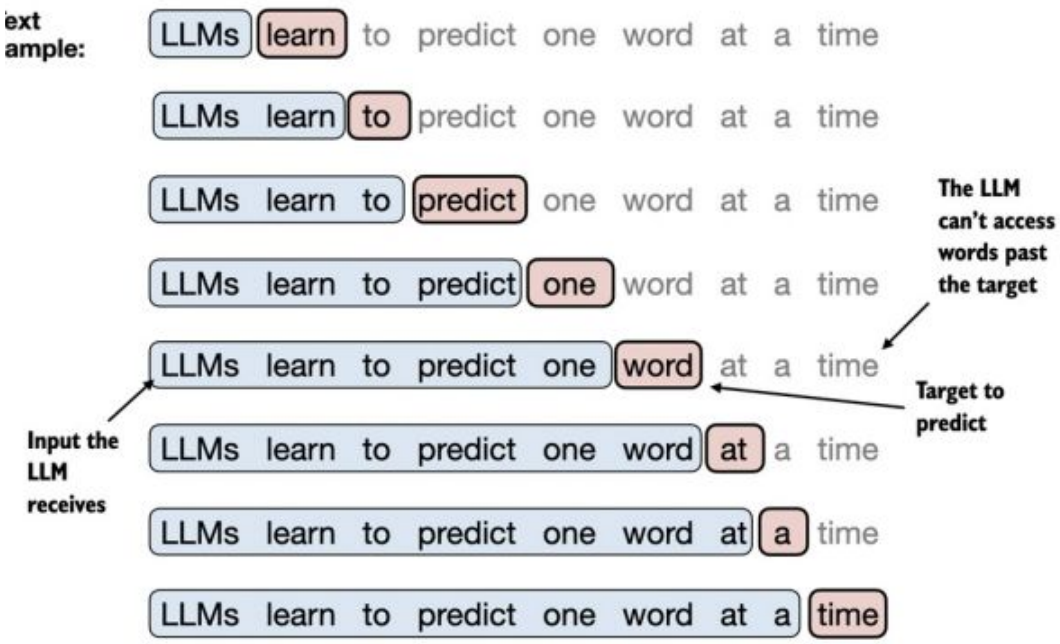
Dataset name	Dataset description	Number of tokens	Proportion in training data
CommonCrawl (filtered)	Web crawl data	410 billion	60%
WebText2	Web crawl data	19 billion	22%
Books1	Internet-based book corpus	12 billion	8%
Books2	Internet-based book corpus	55 billion	8%
Wikipedia	High-quality text	3 billion	3%

1. IO: Autoregressão

- O GPT é um modelo **autoregressivo**.

- Geramos os dados por **amostragem de janela deslizante**.

- **Parâmetros:**
  - **max_len**: número de tokens em uma janela
    - GPT: 512 tokens.
  - **stride**: deslocamento de uma janela para outra.
  - **batch**: número de entradas processadas de uma vez



UDESC

1. IO: exemplo de amostragem

- batch_size = 5, max_len = 3, stride = 1

```
X = [[1, 2, 3],   # "o rato roeu"
     [2, 3, 4],     # "rato roeu a"
     [3, 4, 5],     # "roeu a roupa"
     [4, 5, 6],     # "a roupa do"
     [5, 6, 7]]     # "roupa do rei"
y = [4, 5, 6, 7, 8]  # “a”, “roupa”, “do”, “rei”, “de”
```

- batch_size=5, max_len = 3, stride = 2

```
X = [[1, 2, 3],  # "o rato roeu"
     [3, 4, 5],    # "roeu a roupa"
     [5, 6, 7],    # "roupa do rei"
       [7, 8, 9],    # “rei de roma”
y = [4, 6, 8, <PAD>] # “a”, “do”, “de”, “”
```

o rato roeu a roupa do rei de roma

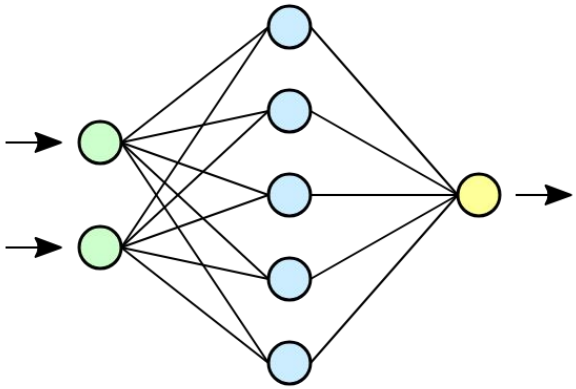
tokens = [1, 2, 3, 4, 5, 6, 7, 8, 9]

1. IO: arquitetura

Então esse é o formato de entrada e saída:

Entra

```
X = [[1, 2, 3], # "o rato roeu"
     [2, 3, 4],   # "rato roeu a"
     [3, 4, 5],   # "roeu a roupa"
     [4, 5, 6],   # "a roupa do"
     [5, 6, 7]]   # "roupa do rei"
```



Sai

```
y = [4, # “a”
       5, #“roupa”
       6, #“do”
       7, #“rei”
       8] #“de”
```

1. IO: arquitetura

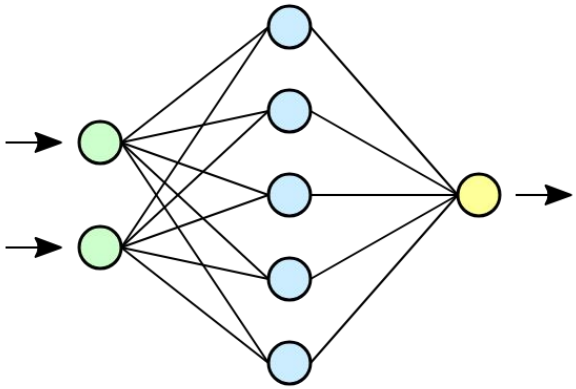
Então esse é o formato de entrada e saída.

ERRADO!

Pergunta: Qual é o problema com esse modelo?

Entra

```
X = [[1, 2, 3], # "o rato roeu"
     [2, 3, 4],   # "rato roeu a"
     [3, 4, 5],   # "roeu a roupa"
     [4, 5, 6],   # "a roupa do"
     [5, 6, 7]]   # "roupa do rei"
```

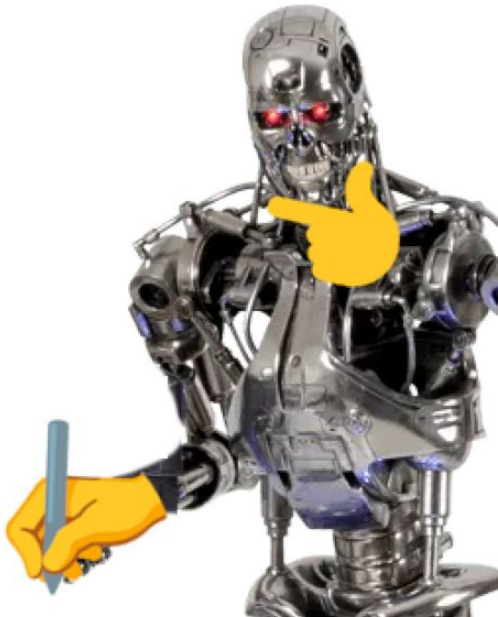


Sai

```
y = [4, # “a”
       5, #“roupa”
       6, #“do”
       7, #“rei”
       8] #“de”
```

1. IO: Estocasticidade

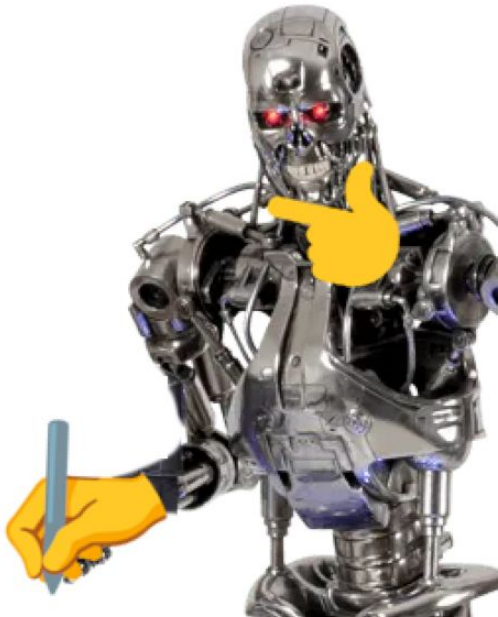
- Queremos gerar um modelo **estocástico**. Como fazer?



UDESC

1. IO: Estocasticidade

- Queremos gerar um modelo **estocástico**. Como fazer?
  - Precisamos ter mais de uma opção de saída.
    - Quantas?
  - Deve ser possível comparar as saídas de alguma forma.
    - Mais prováveis == maiores
  - Essa saída precisa ser construída em cima das saídas que já temos.
    - Re-rotular a base.



UDESC

1. IO: Função de probabilidade

Substituímos o token de saída por um vetor com **a probabilidade para todas as palavras possíveis (vocabulário)**.

o rato roeu a roupa do rei de roma

tokens = [1, 2, 3, 4, 5, 6, 7, 8, 9]

X = [[1,2,3,4]] # o, rato, roeu, a

y = [5] # roupa

y' = [0.0, 0.0, 0.0, 0.0, 1.0 , 0.0, 0.0, 0.0, 0.0]

y' indica que é 100% de chance de ser "roupa" e 0% para outras palavras.

UDESC

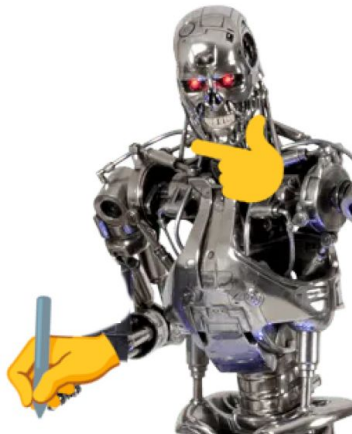
1. IO: Função de probabilidade

- Na prática, se treinarmos um modelo, a saída (y_pred) não vai encontrar um resultado exato (um token com 100%), linguagens variam!
  - Cada token vai ter alguma probabilidade.
  - Palavras mais prováveis terão valores maiores, e vice-versa.
  - A soma fecha 100%.
  - Multiplicar y' por y_pred' retorna a probabilidade.

y' = [0.0, 0.0, 0.0, 0.0, 1.0 , 0.0, 0.0, 0.0, 0.0]

y_pred' = [0.01, 0.02, 0.05, 0.10, 0.60, 0.05, 0.04, 0.02, 0.06, 0.05]

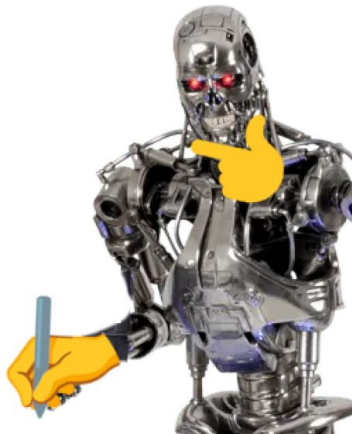
y' * y_pred' = p(y) = 0.6



UDESC

1. IO: Função de probabilidade

- E se temos uma **sequência de tokens?**



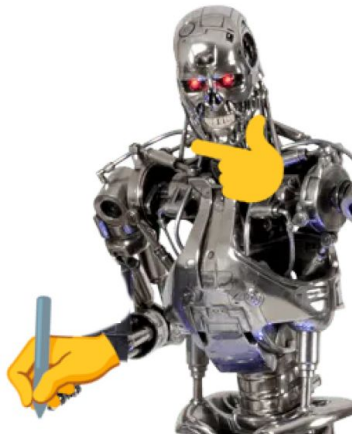
UDESC

1. IO: Função de probabilidade

- E se temos uma **sequência de tokens?**
  - A chance de uma sequência ser a correta é a multiplicação (p1*p2…*pn).
  - Aplicamos a função log e vira uma soma. **Por que fazemos isso?**

$L(y_1 \text{ à } y_n) = p1*p2…*pn.$

$\mathbf{L(y_1 \text{ à } y_n) = \sum y' \cdot \log(y\_pred')}$



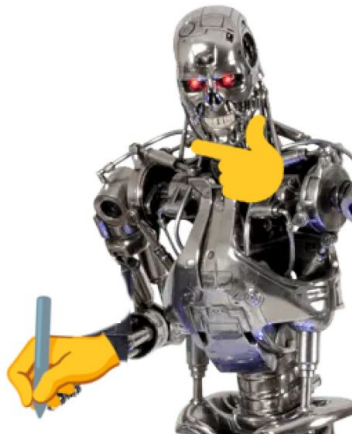
UDESC

1. IO: Função de probabilidade

- E se temos uma **sequência de tokens?**
  - A chance de uma sequência ser a correta é a multiplicação (p1*p2…*pn).
  - Aplicamos a função log e vira uma soma. **Por que fazemos isso?**
    - **Mais eficiente computacionalmente.**

$L(y_1 \text{ à } y_n) = p1*p2…*pn.$

$\mathbf{L(y_1 \text{ à } y_n) = \sum y' \cdot \log(y\_pred')}$

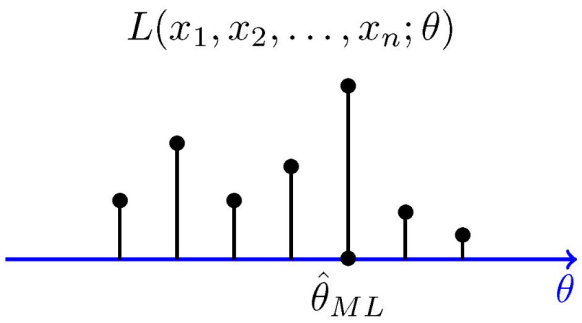


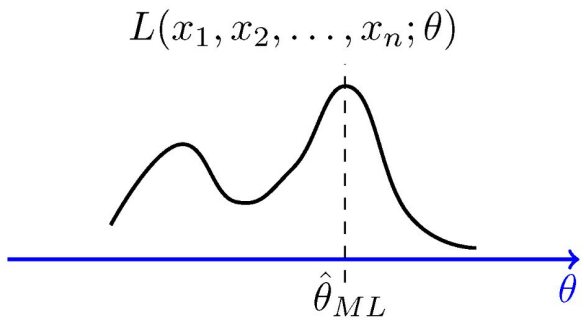
UDESC

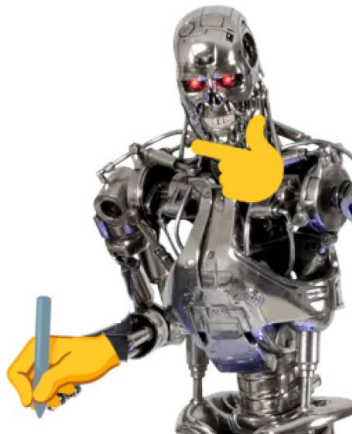
1. IO: Função de probabilidade

- O vetor de probabilidades de saída que criamos se chama **função de probabilidade de densidade.**
- Essa equação que soma probabilidades se chama **Log-Verosimilhança.**
- **Funções de log-verossimilhanças são as funções de perdas de LLMs**
  - GPT utiliza a entropia-cruzada como loss, uma construída utilizando a teoria de verossimilhança.

$$L(y_1 \text{ à } y_n) = \sum y' \cdot \log(y\_pred')$$





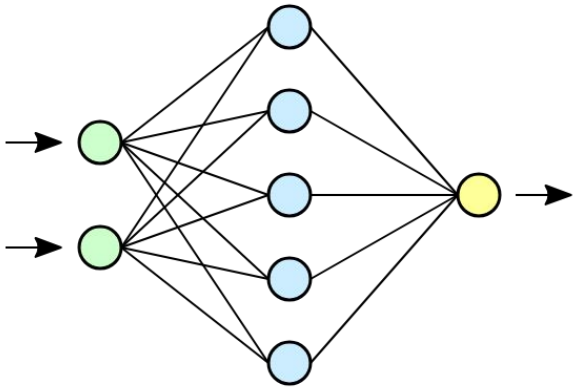


1. IO: Arquitetura

Tínhamos uma saída estática.

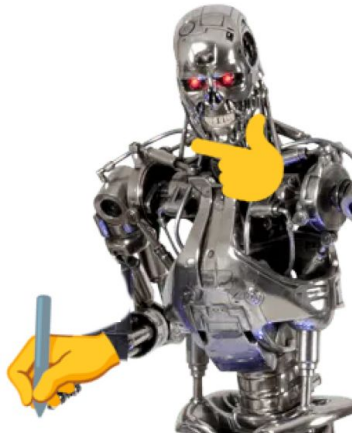
Entra

```
X = [[1, 2, 3], # "o rato roeu"
     [2, 3, 4],   # "rato roeu a"
     [3, 4, 5],   # "roeu a roupa"
     [4, 5, 6],   # "a roupa do"
     [5, 6, 7]]   # "roupa do rei"
```



Sai

```
y = [4, # “a”
     5, #“roupa”
     6, #“do”
     7, #“rei”
     8] #“de”
```

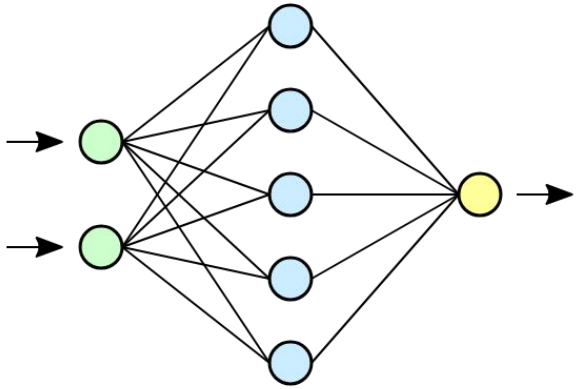


1. IO: Arquitetura

Agora temos uma saída de probabilidades:

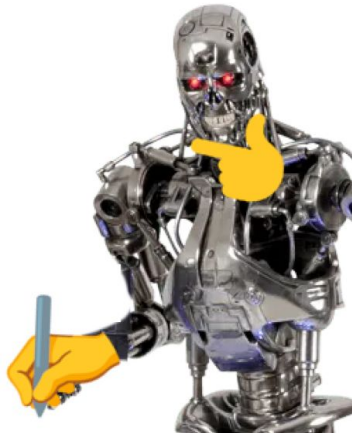
Entra

```
X = [[1, 2, 3], # "o rato roeu"
     [2, 3, 4],   # "rato roeu a"
     [3, 4, 5],   # "roeu a roupa"
     [4, 5, 6],   # "a roupa do"
     [5, 6, 7]]   # "roupa do rei"
```



Sai

```
y_pred_probs = [
          [0.05, 0.10, 0.60, 0.10, 0.10, ...],
          [0.04, 0.10, 0.05, 0.60, 0.10, ...],
          [0.03, 0.10, 0.05, 0.07, 0.60, ...],
          [0.02, 0.08, 0.05, 0.07, 0.06, ...],
          [0.01, 0.06, 0.05, 0.07, 0.05, ...],
        ]
```



UDESC

1. IO: Vantagens

- Podemos **escolher** a próxima palavra das probabilidades, não necessariamente o melhor valor vai gerar a melhor verosimilhança para a frase.
- Isso permite nosso LLM ser **estocástico**.

o rato roeu <?>
o rato roeu a -> 0.3
o rato roeu o -> 0.2

o rato roeu a roupa [0.3, 0.5]
o rato roeu o queijo [0.2, 0.6]

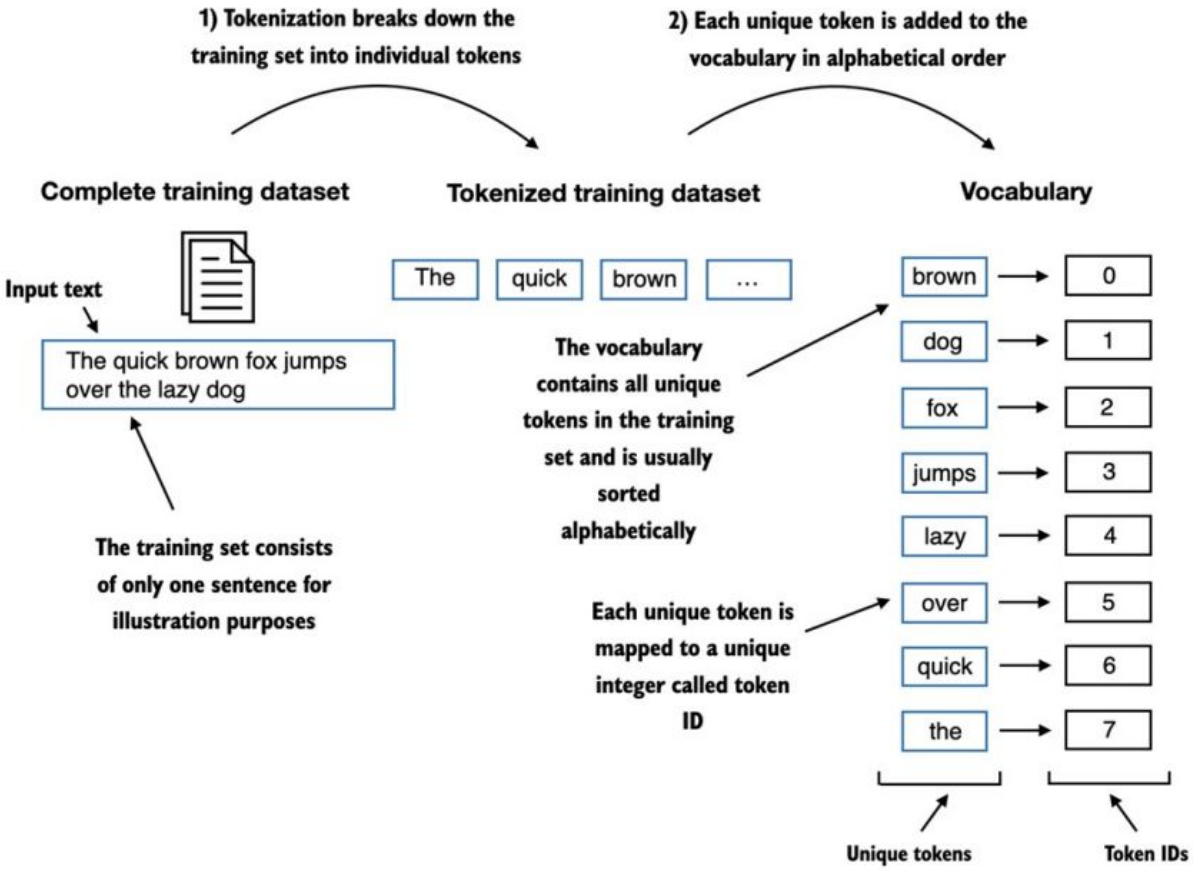
UDESC

Índice

2. Tokenização

- Perceba que todo o processo de amostragem e estruturação da base é feito em cima dos tokens, portanto um processo de **tokenização** é necessário.

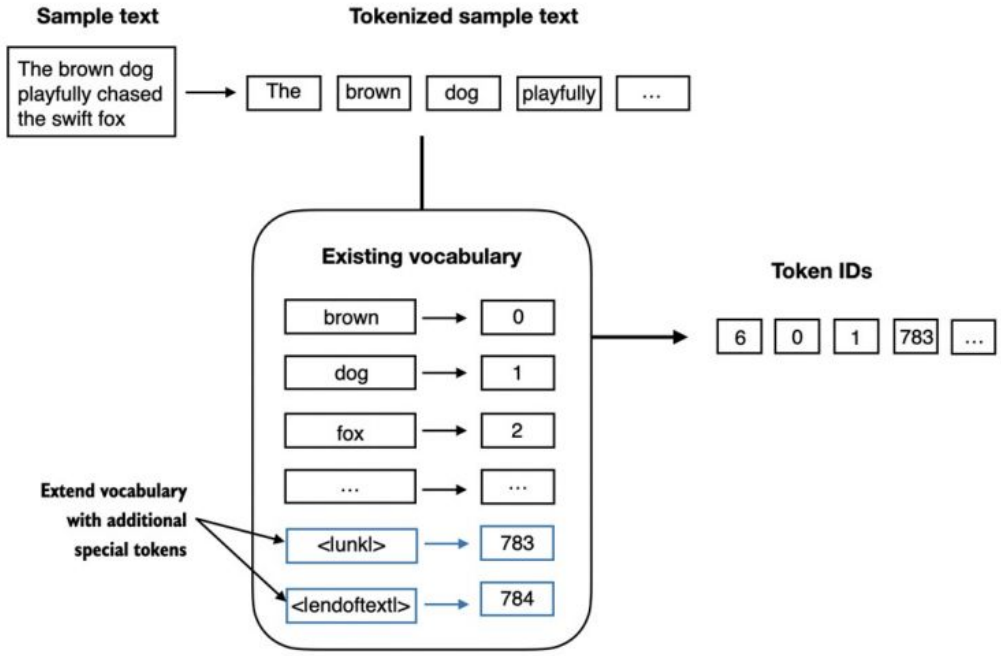
- Um processo de tokenização eficiente é converter as palavras em índices conforme aparecem na base de treino.

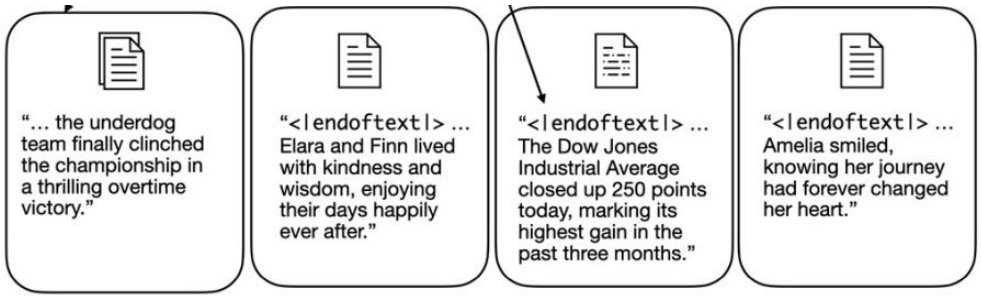


UDESC

2. Tokenização

- No processo de tokenização, ainda podemos adicionar **tokens especiais** que servem como identificadores de partes do texto.
  - **<unk>**: Identificador de token não reconhecido.
  - **<endoftext>**: Identificador de separação de um texto para outro no treinamento.
  - Outros caracteres que facilitem o fine tuning por exemplo.





UDESC

2. Tokenização

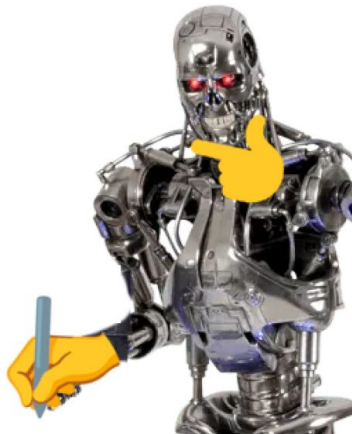
- A tokenização por palavras, é eficiente computacional.
  - Contudo, qual é o problema dessa abordagem?

2. Tokenização

- A tokenização por palavras, é eficiente computacional.
  - Contudo, qual é o problema dessa abordagem?

- A frequência do token <unk> se torna muito frequente:
  - Nomes próprios.
  - Variáveis.
  - Expressões
  - Dentre outras.

- Precisamos de um processo de tokenização que indexe palavras que não apareceram no treinamento.



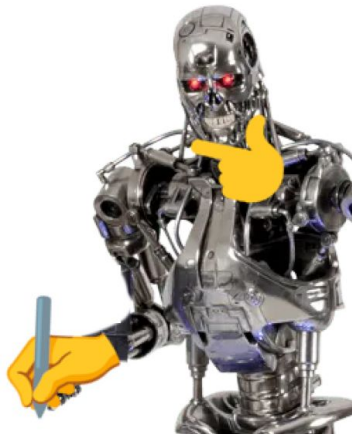
UDESC

2. Byte Pair Encoding (BPE)

LLMs usam um algoritmo de tokenização extremamente simples chamado Byte Pair Encoding:

1. Contamos todas as **sequências de 2 letras** dentro da base.
2. Pegamos a **sequência mais frequente** definimos como um novo token placeholder para ela.
3. Tokenizamos a base novamente com o novo token
4. Verificamos se atingimos o número máximo de tokens.
   a. Caso não: Voltamos ao passo 1.
   b. Caso sim: Encerramos a tokenização.

Geralmente o número máximo de tokens é tunado para não gerar tokens muito grandes.



UDESC

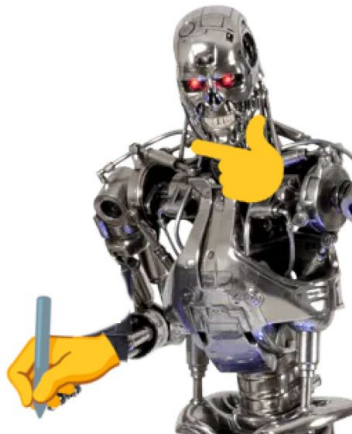
2. BPE: Exemplo

aaabdaaabac

[1, 1, 1, 2, 3, 1, 1, 1, 2, 1, 4]

Início

Léxico: a = 1, b =2, d =3, c=4



UDESC

2. BPE: Exemplo

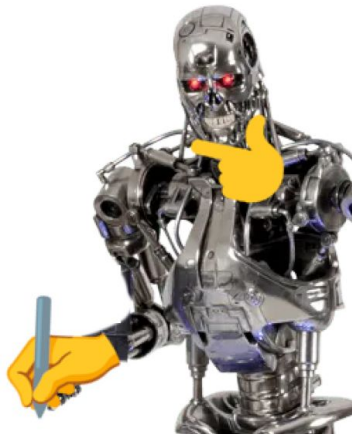
aaabdaaabac

[1, 1, 1, 2, 3, 1, 1, 1, 2, 1, 4]

Interação 1:

(1, 1) → 4 vezes
(1, 2) → 2 vezes
(2, 3) → 1 vez
(3, 1) → 1 vez
(2, 1) → 1 vez
(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4



UDESC

2. BPE: Exemplo

aaabdaaabac

[5, 1, 2, 3, 5, 1, 2, 1, 4]

Interação 1:

(1, 1) → 4 vezes

(1, 2) → 2 vezes

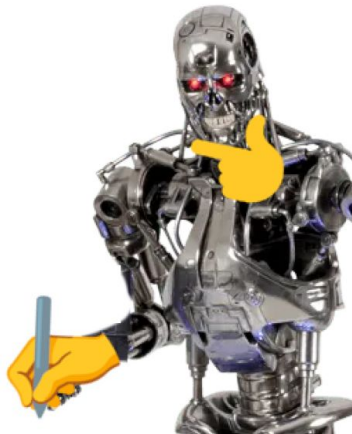
(2, 3) → 1 vez

(3, 1) → 1 vez

(2, 1) → 1 vez

(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4, **aa=5**



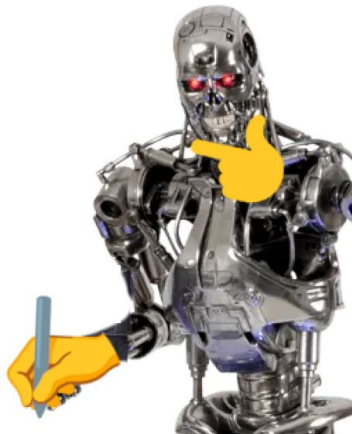
UDESC

2. BPE: Exemplo

aaabdaaabac
[5, 1, 2, 3, 5, 1, 2, 1, 4]

Interação 2:
(5, 1) → 2 vezes
(1, 2) → 2 vezes
(2, 3) → 1 vez
(3, 5) → 1 vez
(2, 1) → 1 vez
(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4, aa=5



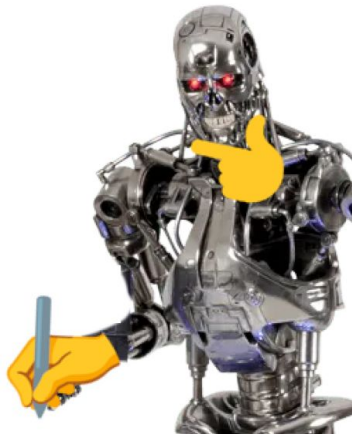
UDESC

2. BPE: Exemplo

aaabdaaabac
[5, 6, 3, 5, 6, 1, 4]

Interação 2:
(5, 1) → 2 vezes
(1, 2) → 2 vezes
(2, 3) → 1 vez
(3, 5) → 1 vez
(2, 1) → 1 vez
(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4, aa=5, ab=6



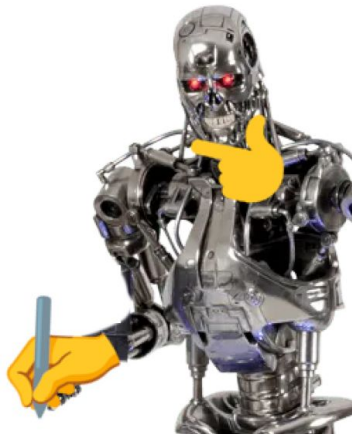
UDESC

2. BPE: Exemplo

aaabdaaabac
[5, 6, 3, 5, 6, 1, 4]

Interação 3:
(5, 6) → 2 vezes
(6, 3) → 1 vez
(3, 5) → 1 vez
(6, 1) → 1 vez
(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4, aa=5, ab=6



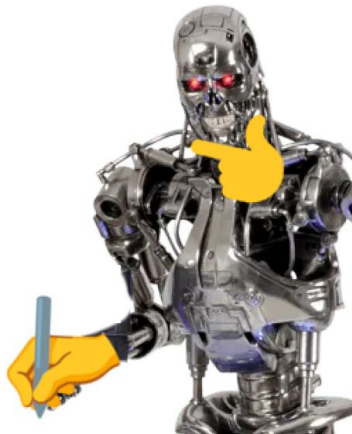
UDESC

2. BPE: Exemplo

aaabdaaabac
[7, 3, 7, 1, 4]

Interação 3:
(5, 6) → 2 vezes
(6, 3) → 1 vez
(3, 5) → 1 vez
(6, 1) → 1 vez
(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4, aa=5, ab=6, **aaab=7**
Se max = 7, pararia aqui.



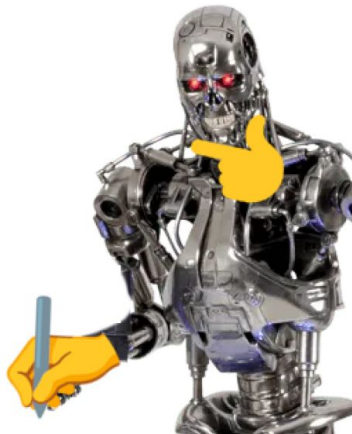
UDESC

2. BPE: Exemplo

aaabdaaabac
[7, 3, 7, 1, 4]

Interação 3:
(7, 3) → 1 vez
(3, 7) → 1 vez
(1, 4) → 1 vez

Léxico: a = 1, b =2, d =3, c=4, aa=5, ab=6, aaab=7
Possível condição de parada.
Poderíamos também continuar a execução.
Depende da abordagem.



Índice

1. **Inputs e Outputs (IO):**
    a. Dados de texto.
    b. Autoregressão.
    c. Amostragem por janela
    d. Camada de Entrada: janela de tokens.
    e. Camada de Saída: Função de densidade de probabilidade e verossimilhança

2. **Tokenização:**
    a. Algoritmo Byte-Pair tokenization

3. **Embeddings**:
    a. A arquitetura Transformer.
        i. Embedding de Tokenização.
        ii. Embeddings de Posição.
        iii. Embeddings contextuais.

UDESC

Somos todos adultos aqui...

- Aula que vem iremos falar sobre a parte a arquitetura de uma Transformer.

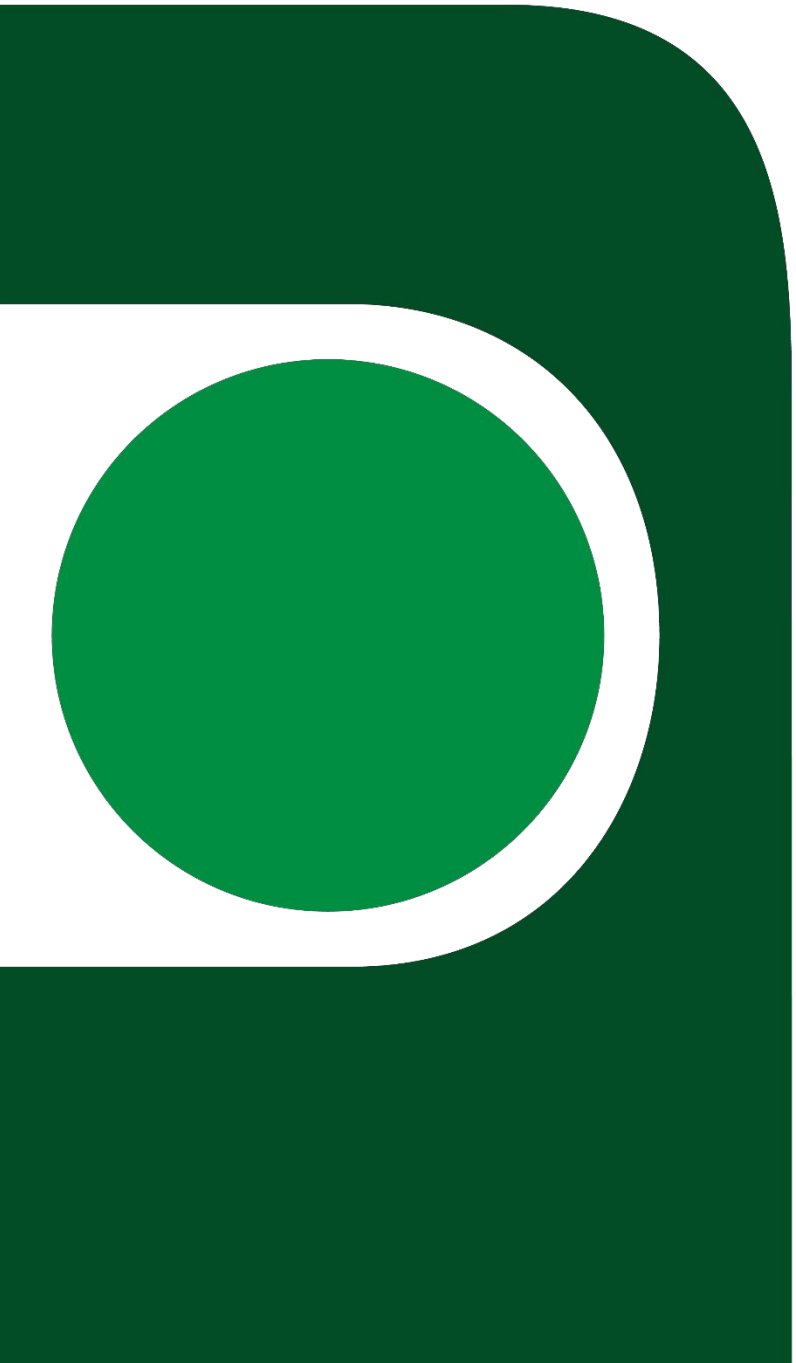
- Deixo um material sobre fine-tuning com feedback humano de um GPT como leitura:
  - Fine-Tuning Language Models from Human Preferences.

- Um resumo deste artigo foi feito na forma de um vídeo de comédia, entretanto muito explicativo.

Maximally bad output

Referências

- VASWANI, A. **Attention is all you need**. Advances in Neural Information Processing Systems, 2017.
- MURPHY, Kevin P. **Probabilistic machine learning: an introduction**. MIT press, 2022.
- RADFORD, A. **Improving language understanding by generative pre-training**. 2018.
- **Generative Models vs Discriminative Models: Which One to Choose?**. Turing, disponível em: <https://www.turing.com/kb/generative-models-vs-discriminative-models-for-deep-learning>. Acesso em 20-08-2024.
- GOZALO-BRIZUELA, Roberto; GARRIDO-MERCHAN, Eduardo C. **ChatGPT is not all you need. A State of the Art Review of large Generative AI models.** arXiv preprint arXiv:2301.04655, 2023.
- FOSTER, David. **Generative deep learning**. " O'Reilly Media, Inc.", 2022.



Obrigado

UDESC – Universidade do Estado de Santa Catarina

Thiago Brandenburg

thiago.brandenburg@edu.udesc.br