Deep Learning in NLP

Iván Moreno (ivan@nieveconsulting.com)

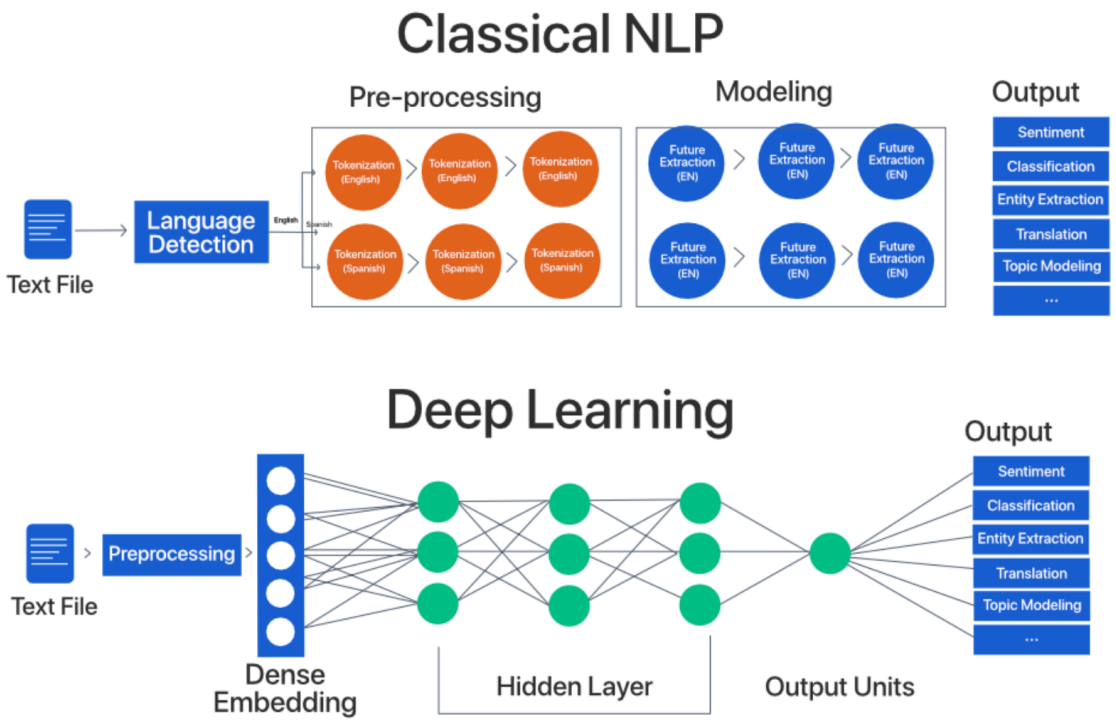
Table of Contents

1. Introduction
2. NLP Pipeline with Deep Learning
3. Recurrent Neural Networks (RNNs)
4. Transformers
5. Training of Large Language Models

Introduction

- **Deep Learning** has revolutionized Natural Language Processing (NLP) by enabling the development of powerful language models.
- There are various deep learning models used in NLP, including **Recurrent Neural Networks (RNNs)**, **Convolutional Neural Networks (CNNs)**, and **Transformers**.
- In this presentation, we will explore the role of deep learning models in NLP, focusing on Transformers and their applications.

NLP Pipeline with Deep Learning

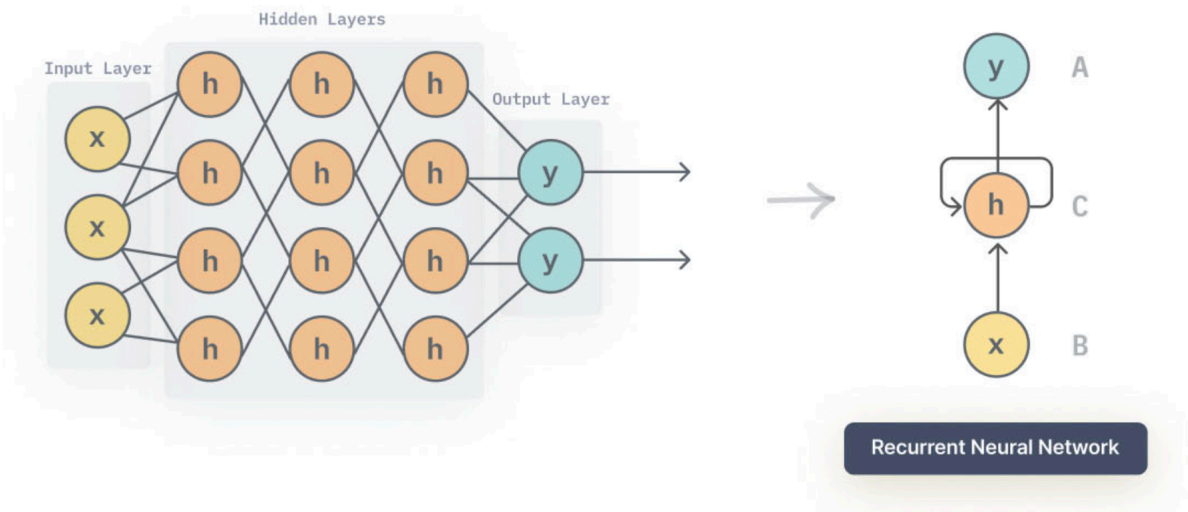


- In contrast to other modeling approaches, deep learning models learn complex patterns and representations *directly from data*.
- This means that, instead of manually engineering features, deep learning models can **automatically** extract relevant information from text data.

Why Different Architectures?

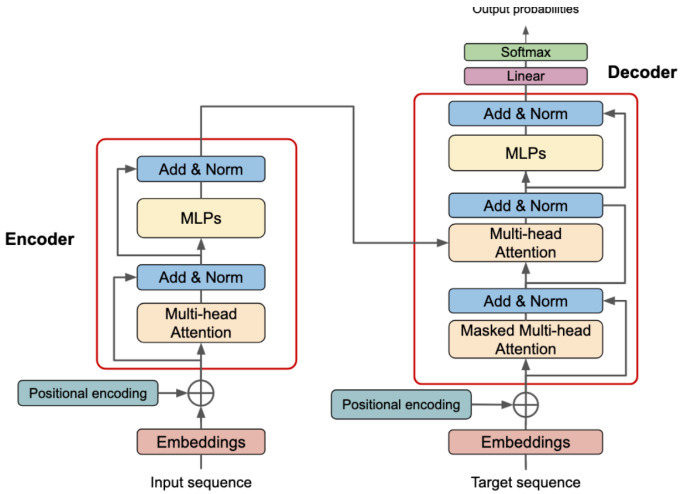
- **Traditional neural networks (MLPs)** process *fixed-size inputs*, making them unsuitable for sequential data. They, however, have been used in NLP for **downstream tasks** like text classification by combining a fixed-size representation of text (i.e., computed features), and then feeding it to an MLP.
- **Convolutional Neural Networks (CNNs)** are effective for image processing but less suited for sequential data. They have been adapted for NLP tasks by treating text as a 1D sequence.
- **Recurrent Neural Networks (RNNs)** were *developed to handle sequential data* by maintaining a hidden state that captures context. However, they are prone to vanishing/exploding gradients and slow training due to *sequential processing*.
- **Transformers** introduced a new architecture that revolutionized NLP by **enabling parallel processing** of tokens and capturing **long-range dependencies**. They are now the foundation of most state-of-the-art language models.

Recurrent Neural Networks (RNNs)



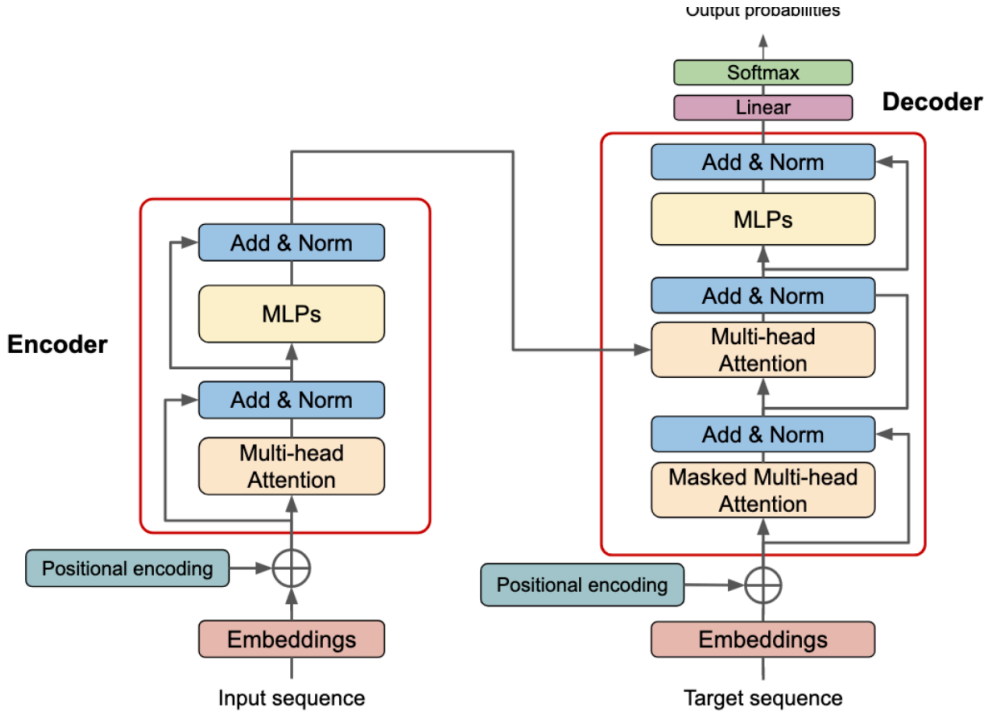
- Traditional models for sequential data processing.
- Process *one token at a time*, maintaining a **hidden state** that captures context.
- Prone to vanishing/exploding gradients and slow training due to sequential processing.
  - **Long Short-Term Memory (LSTM)** and **Gated Recurrent Unit (GRU)** are variants that address these issues by introducing gating mechanisms.
  - Gates control the flow of information, allowing the model to learn when to remember or forget information.

Transformers



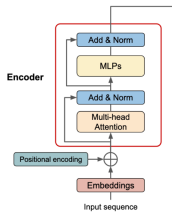
- **Transformers** are deep learning models designed to handle sequential data, such as natural language.
- Introduced by *Vaswani et al.* (2017) with the seminal paper *Attention is All You Need*.
  - Initially developed for machine translation, have since become the foundation of most state-of-the-art language models.
- Process all tokens **in parallel**, allowing for more efficient computation.
- **Attention** mechanism captures **long-range dependencies**, improving context understanding.

Transformer – Components



- The original transformer architecture, designed for machine translation, had *two main components*:
  1. **Encoder**: Processes the input sequence and generates a compressed representation.
  2. **Decoder**: Generates the output sequence based on the encoder's representation and the target sequence.

Encoder Architecture

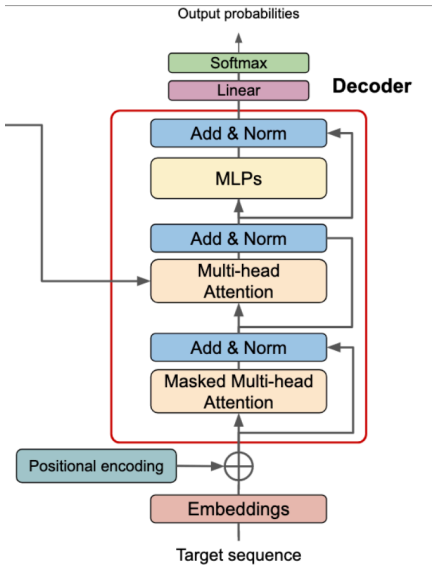


- Each encoder layer consists of three sub-layers:
  1. **Multi-Head Self-Attention Mechanism**: Allows the model to focus on different parts of the input simultaneously.
  2. **Layer Normalization**: Normalizes the output of each sub-layer. This helps stabilize the training process.
  3. **Feedforward (MLP)**: Processes information from the attention layer (feature expansion). Further processes and transforms the embeddings.

What is layer normalization?

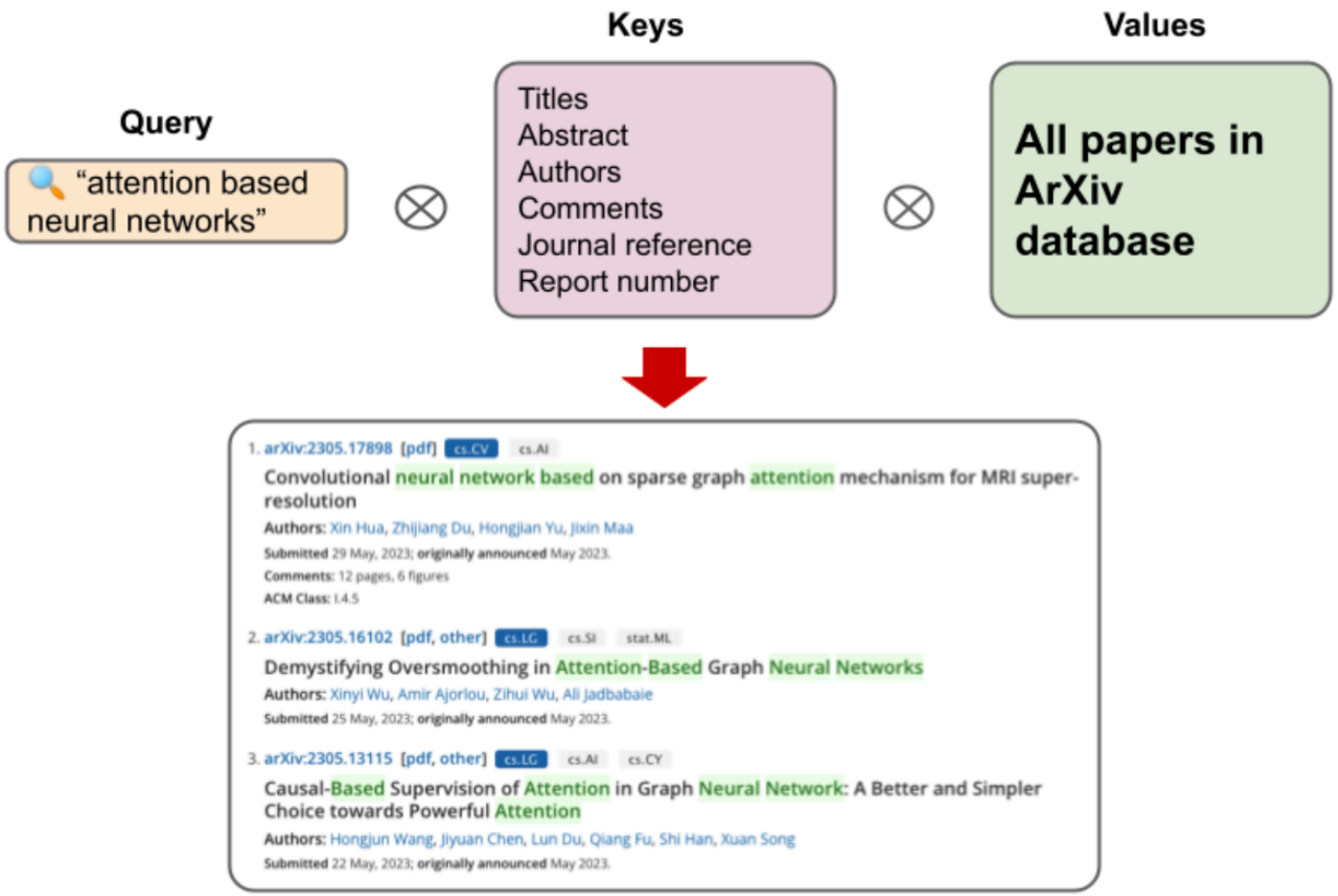
Layer normalization is a technique used to normalize the activations of a neural network layer. This is done in order to prevent the **“internal covariate shift”** problem, which can slow down training and make it difficult to converge. Layer normalization normalizes the activations of each layer across the feature dimension, ensuring that the mean and variance of the activations are consistent across training examples.

Decoder Architecture

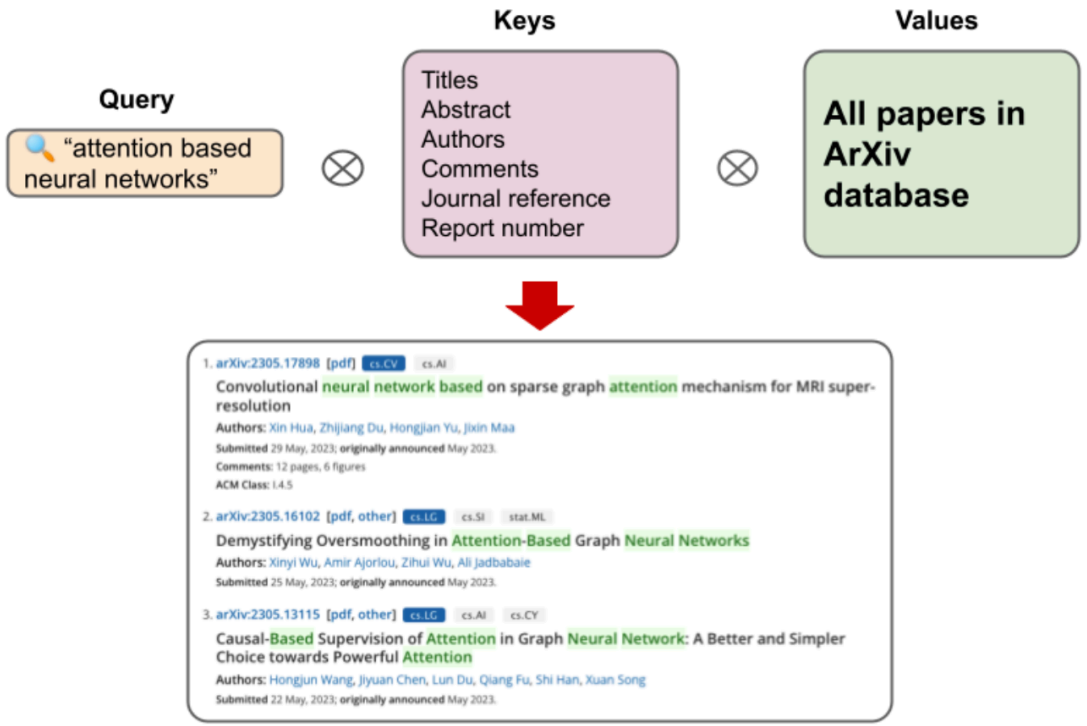


- *Almost identical* to the encoder, but with an *additional sub-layer* that performs multi-head attention over the encoder's output.
- The goal of the decoder is to generate the output sequence **token by token**, attending to the encoder's output and the previously generated tokens.
- In order to prevent the decoder from peeking at future tokens, the self-attention mechanism in the decoder is masked, only allowing the model to attend to tokens that have already been generated.

The Attention Function



- Attention measures the *similarity between two vectors* and returns the weighted similarity scores.
- The attention function takes three inputs: **Query (Q)**, **Key (K)**, and **Value (V)** vectors.



- Intuitively, query, key and values are terms commonly used in information retrieval systems.
  - **Query**: The search term (input) used to retrieve information.
  - **Key**: The index or identifiers of the documents in the database.
  - **Value**: All the information contained in a database.
  - **A search engine would use the query to find the most relevant documents (values) based on the keys.**

Self-Attention vs Cross-Attention

Depending on the origin of the query, key, and value vectors, attention mechanisms can be classified into self-attention and cross-attention.

- **Self-Attention**: The query, key, and value vectors are derived from the **same input sequence**.
- **Cross-Attention**: The query vector is derived from one input sequence, while the key and value vectors are derived from **another** input sequence.
- In the transformer architecture, *self-attention* is used to capture dependencies within the input sequence, while *cross-attention* is used to relate the input and output sequences in the decoder.

Mathematical Formulation

$$\text{Attention}(\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{softmax}\left(\frac{\mathrm{QK}^{\mathrm{T}}}{\sqrt{\mathrm{d_k}}}\right)\mathrm{V}$$

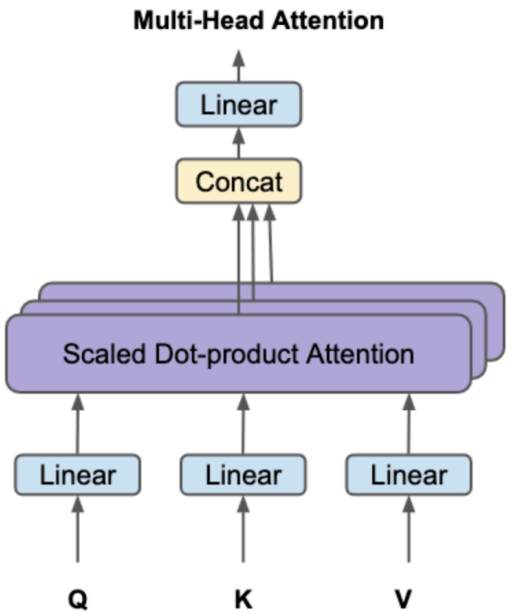
- The attention function computes the weighted sum of the values (V) based on the similarity between the query (Q) and key (K) vectors.
- The $\mathrm{softmax}$ function normalizes the similarity scores, ensuring that the weights sum to 1.
- The scaling factor $\sqrt{\mathrm{d_k}}$ is used to prevent the dot product of Q and K from becoming too large, which can lead to gradients vanishing during training.
  - This could happen when $\mathrm{d_k}$ is large, as the dot product grows with the dimensionality of the vectors.
- The output of the attention function is the *weighted sum of the values*, which is then passed through a feedforward neural network.

Parallel Processing

Transformer parallelization is *enabled* by the attention function.

Provided that both Q, K, and V are matrices, the attention function can be computed by **two matrix multiplications**.

Multi-Head Attention

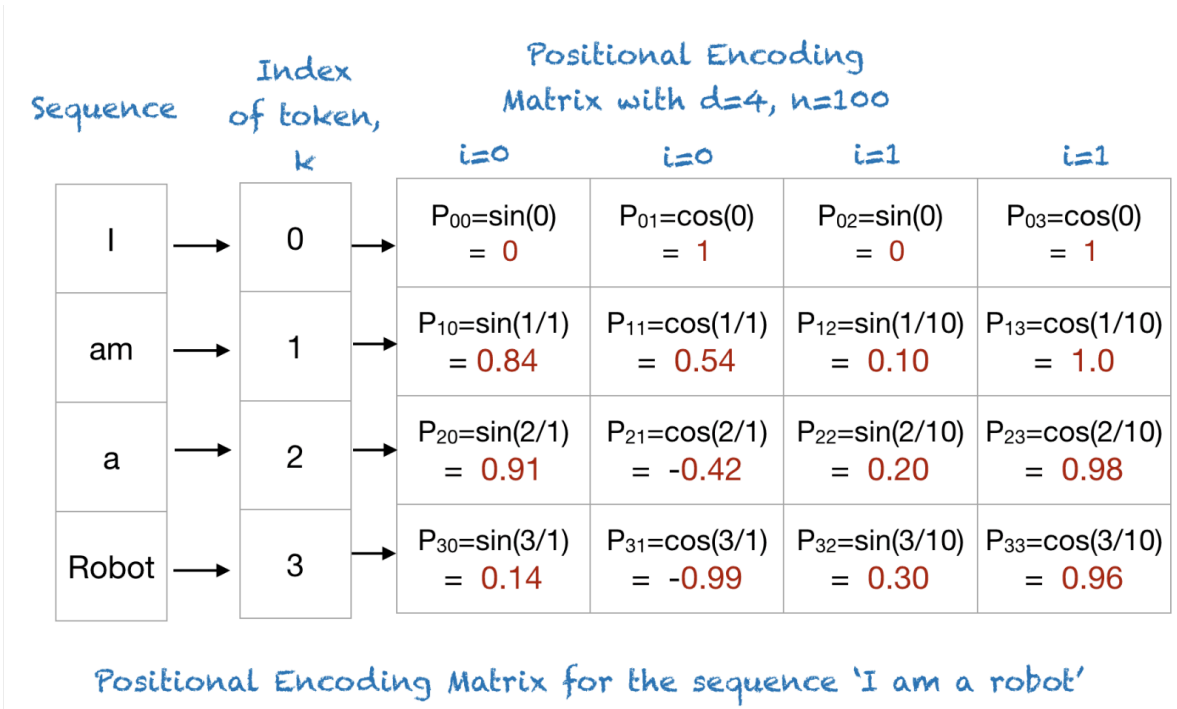


- Multi-head attention allows the model to *jointly attend to information from different representation subspaces at different positions*.
- Each head computes the attention function **in parallel**, using different learned linear projections of the query, key, and value vectors.
- The outputs of the multiple heads are **concatenated** and linearly transformed to produce the final output.

Visualization of Multi-Head Attention

- exBERT is a visualization tool that allows you to explore the *attention patterns learned by transformer models*.
- It shows **how different heads attend to different parts** of the input sequence, providing insights into the model's decision-making process.
- Available at exBERT.

Positional Encoding



Positional Encoding Matrix for the sequence 'I am a robot'

- **Transformers do not inherently understand the order of tokens in a sequence**, as they process tokens in parallel.
- Positional encoding is added to the input embeddings to provide information about the token's position in the sequence.
- Different positional encoding schemes can be used, such as $\sin$ and $\cos$ functions.

Transformer Variants

- **Encoder-Decoder Transformers**: Consist of an encoder and a decoder. It's the original transformer architecture used for machine translation.
- **Encoder-Only Transformers**: Used for tasks where the input and output sequences are of the same length, such as text classification.
  - When talking about same-length sequences, it means that the model processes each token independently and does not generate an output sequence.
- **Decoder-Only Transformers**: Used for tasks where the output sequence is generated token by token, such as text generation.

How can decoder-only transformers follow instructions?

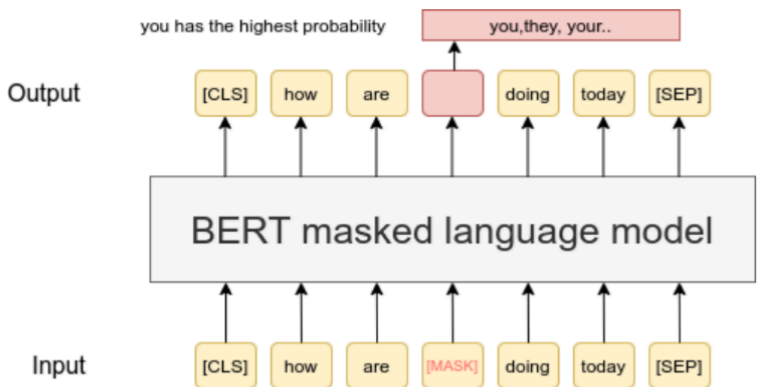
- In an encoder-decoder transformer, the input is first encoded into a fixed-size representation by the encoder (context vector), which is then used by the decoder to generate the output sequence.
- In a decoder-only transformer, the model generates the output sequence token by token, attending to the encoder's output and the previously generated tokens.

The self-attention mechanism within the decoder allows each token to attend to all previous tokens, including the instruction. This means that when generating a response, the model can effectively use the instruction as part of its context.

Which Transformer Variant to Use?

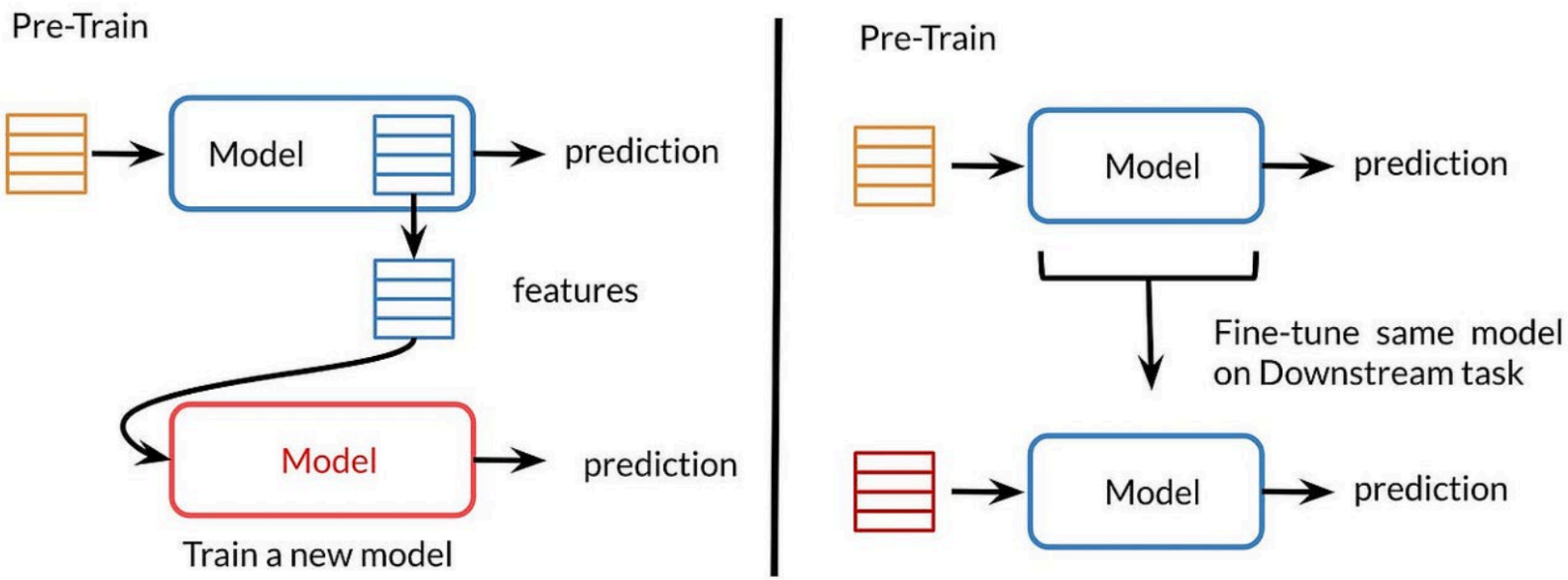
- **Encoder-Decoder Transformers**: Natural language understanding tasks like machine translation, summarization, and question answering.
- **Encoder-Only Transformers**: Text classification, sentiment analysis, and named entity recognition.
- **Decoder-Only Transformers**: Language modeling, text generation, and dialog systems.

Training of Large Language Models



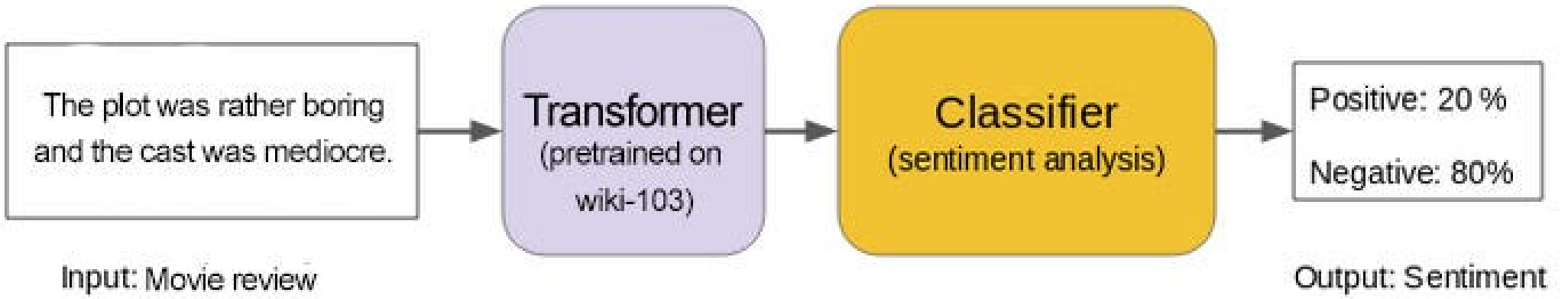
- LLMs are typically trained on large-scale datasets, in a **self-supervised** manner. This means that the model learns from the input data itself, without requiring explicit labels.
- Depending on the model variant, different training objectives can be used:
  - **Next Token Prediction**: The model is trained to predict the next token in the input sequence.
  - **Masked Language Modeling (MLM)**: The model is trained to predict masked tokens in the input sequence.
  - **Next Sentence Prediction (NSP)**: The model is trained to predict whether two input sentences are consecutive or not.

Transfer Learning with Pretrained Models



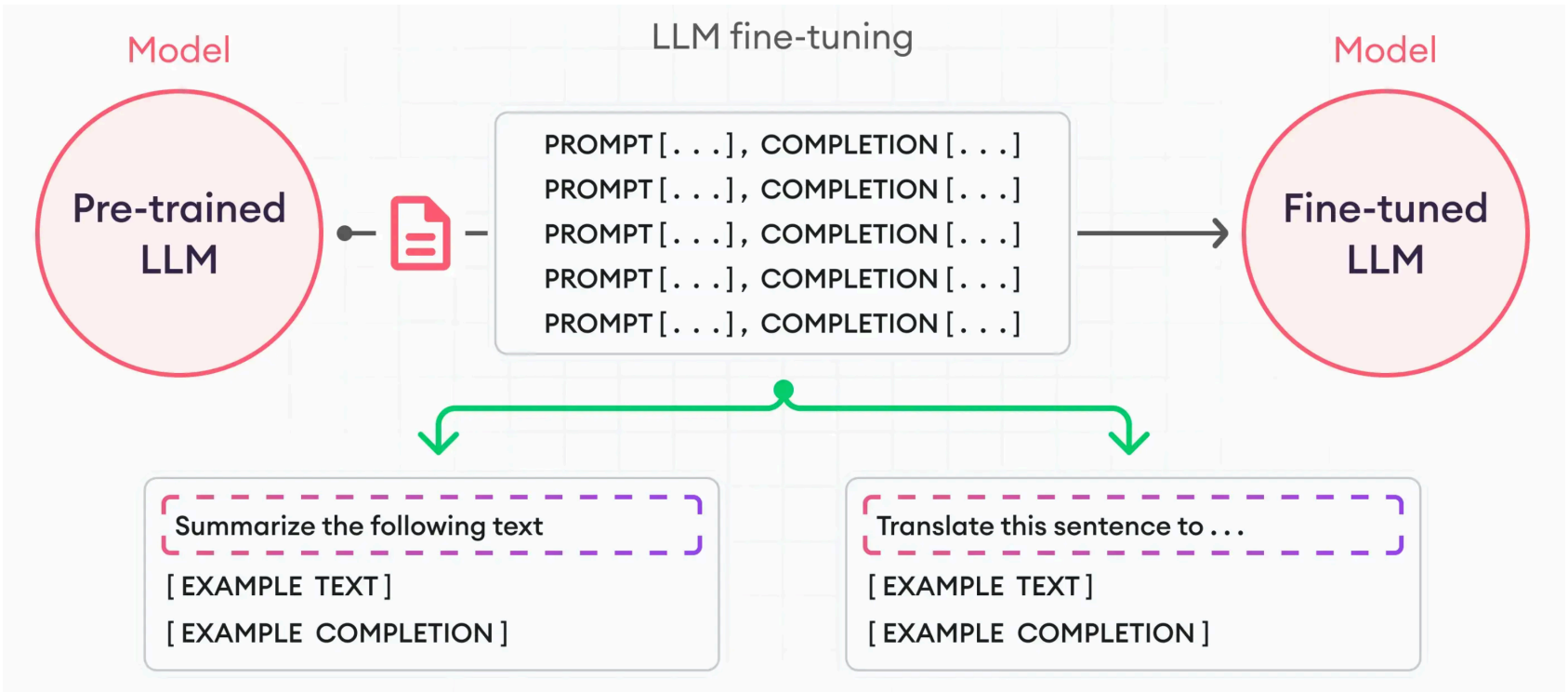
- Pretrained language models, such as **BERT**, **GPT-2**, and **RoBERTa**, have been trained on large-scale datasets and are available for transfer learning.
- Transfer learning involves **fine-tuning** the pretrained model on a specific downstream task, such as text classification or named entity recognition.
- By leveraging the knowledge learned during pretraining, transfer learning allows the model to achieve good performance on the downstream task with **less data** and **training time**.

Fine-Tuning of Pretrained Models



- Pretrained language models can be **fine-tuned** on downstream tasks by adding a task-specific head to the model.
- The pretrained weights are **frozen** during fine-tuning, and only the weights of the task-specific head are updated.
- Fine-tuning allows the model to adapt to the specific characteristics of the downstream task, improving performance.

Instruction Tuning



- A common challenge in NLP is **instruction following**, where the model is required to generate a sequence of actions based on a set of instructions.
- Pretrained transformers can be fine-tuned on instruction-following tasks by providing the instructions as input and training the model to generate the corresponding actions.
- The self-attention mechanism in transformers allows the model to capture dependencies between the instructions and the generated actions, enabling effective instruction following.