Université Paris Dauphine PSL

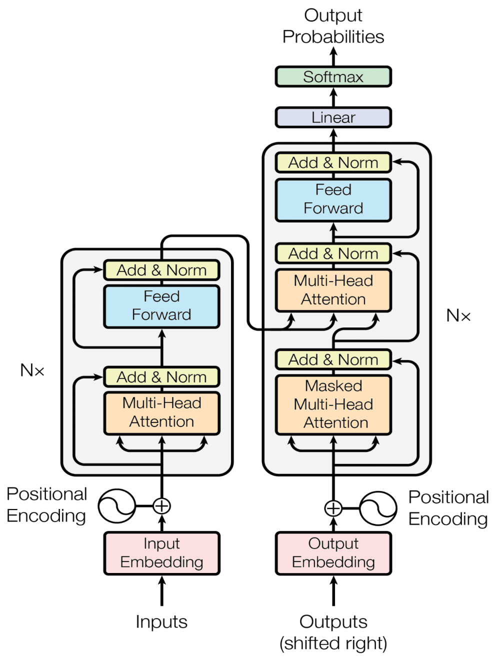
Campus de Tunis

From Attention to Instruction: Diving into Transformer Models and Prompt Design

Report elaborated by:

Chiheb GUESMI

M1 Artificial Intelligence Data Science



Academic Year

2024/2025

Contents

List of Figures

List of Tables

Introduction

In recent years, Transformers architectures have redefined the field of natural language processing (NLP), becoming the foundation of powerful large language models (LLMs) that drive applications ranging from chatbots to code generation tools. Although their performance is undeniable, their inner workings often remain opaque to newcomers and even practitioners such as myself.

As a master's student aiming to specialize in AI, I found myself repeatedly asking "What exactly makes LLMs so effective?" How are they built, trained, and directed? In short, I am trying to find out what is going on under the hood.

These questions motivated this report, which seeks to unpack the evolution from attention mechanisms to instruction-tuned models.

This report is structured into three distinct chapters. The first aims to introduce the reader to the foundations of Transformers models, the history of Natural Language Processing (NLP), and a variety of pipelines provided by transformers via Hugging Face.

The second chapter delves into the architectural blocks of Transformers as an attempt to summarize and simplify the landmark paper ***Attention Is All You Need*** by emphasizing the core elements and the relationship between them.

The third and last chapter introduces large language models (LLMs), their core concepts, and the way to master their prompts by treating the different elements and techniques.

Chapter 1

Foundations of Transformer Models

1.1 Introduction

In this chapter, we will delve into Transformer models by briefly uncovering their origin, architecture, and the key pipelines that define their use in modern NLP.

1.2 From NLP to LLMs

Natural language processing is not a new field, it actually appeared as a glimpse as early as the 1900's with the Swiss linguistics professor Ferdinand de Saussure[9]. However, he died before his theory was published. He was one of the very first to describe languages as systems, and that is what humanity started to understand after World War II.

In the 1950s, Alan Turing[10] introduced the concept of a 'thinking machine', and adding to the discoveries on how the human brain works through neural networks all of these events helped set the stage for modern AI and NLP to stand out.
So, what is NLP?

As the name says, this subfield of Artificial Intelligence allows computers to communicate with people using human language. It has revolutionized the field of artificial intelligence. NLP is not just about text [11]:

- **Content Categorization**: Document summary and anomaly detection.
- **Topic Modeling**: Extracts the main themes from text collections.
- **Contextual Extraction**: Retrieves structured data from text.
- **Sentiment Analysis**: Detects mood and opinions in text.
- **Text-to-Speech & Speech-to-Text**: Converts between voice and text.
- **Document Summarization**: Produces concise text summaries.
- **Machine Translation**: Translates text or speech between languages.

The revolutionary event that split the history of NLP is the introduction of Transformers. Before them, NLP went through the symbolic phase, based on handcrafted rules and grammar, followed by the statistical phase, where models like n-grams and Hidden Markov Models used probabilities but struggled with long-range context.

In the 2000s, neural networks came in, especially feed-forward networks combined with Word2Vec embeddings, which captured word meanings in dense vectors. Then in 2016, Transformers appeared, using self-attention mechanisms that allowed models to focus on different parts of the input simultaneously. This breakthrough paved the way for models like GPT-3 and T5, ushering in the era of Large Language Models (LLMs)[12].

1.3 Transformers

"Attention Is All You Need," the five words that changed the course of NLP and generative AI, As mentioned in the previous section, transformers have revolutionized the field of NLP not only with their unique architecture but also with their open-source pipelines.
Given the layered nature of the transformers architecture, we will dive deeper into it in the next chapter. In this section, we will cover some of the most well-known pipelines provided by Transformers through Hugging Face.

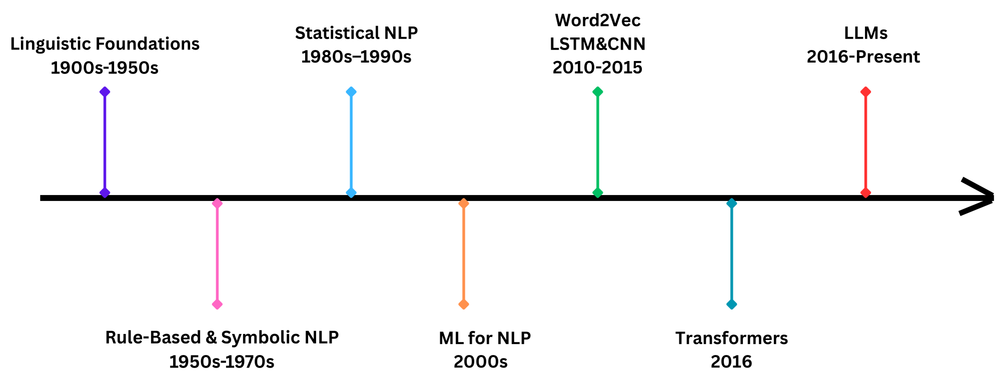
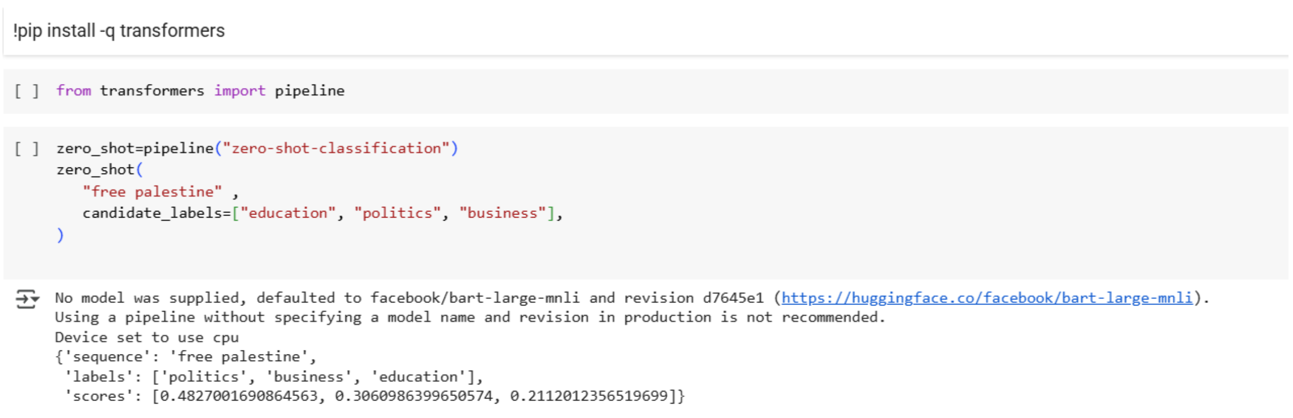


Figure 1.1: From NLP to LLMs Timeline

1.3.1 Pipelines

One of the core modules provided by the Hugging Face Transformers library is the ”pipeline()” function, we will try to dive into some of these pipelines. Based on the task and identifier, it lays out a pretrained and fine-tuned model specific to the very task.
As shown in Fig 1.2, this code emphasizes the zero-shot pipeline because there's no need to fine-tune it on our own data, by giving the desired label, it gives you back the probability scores.



```
!pip install -q transformers
```

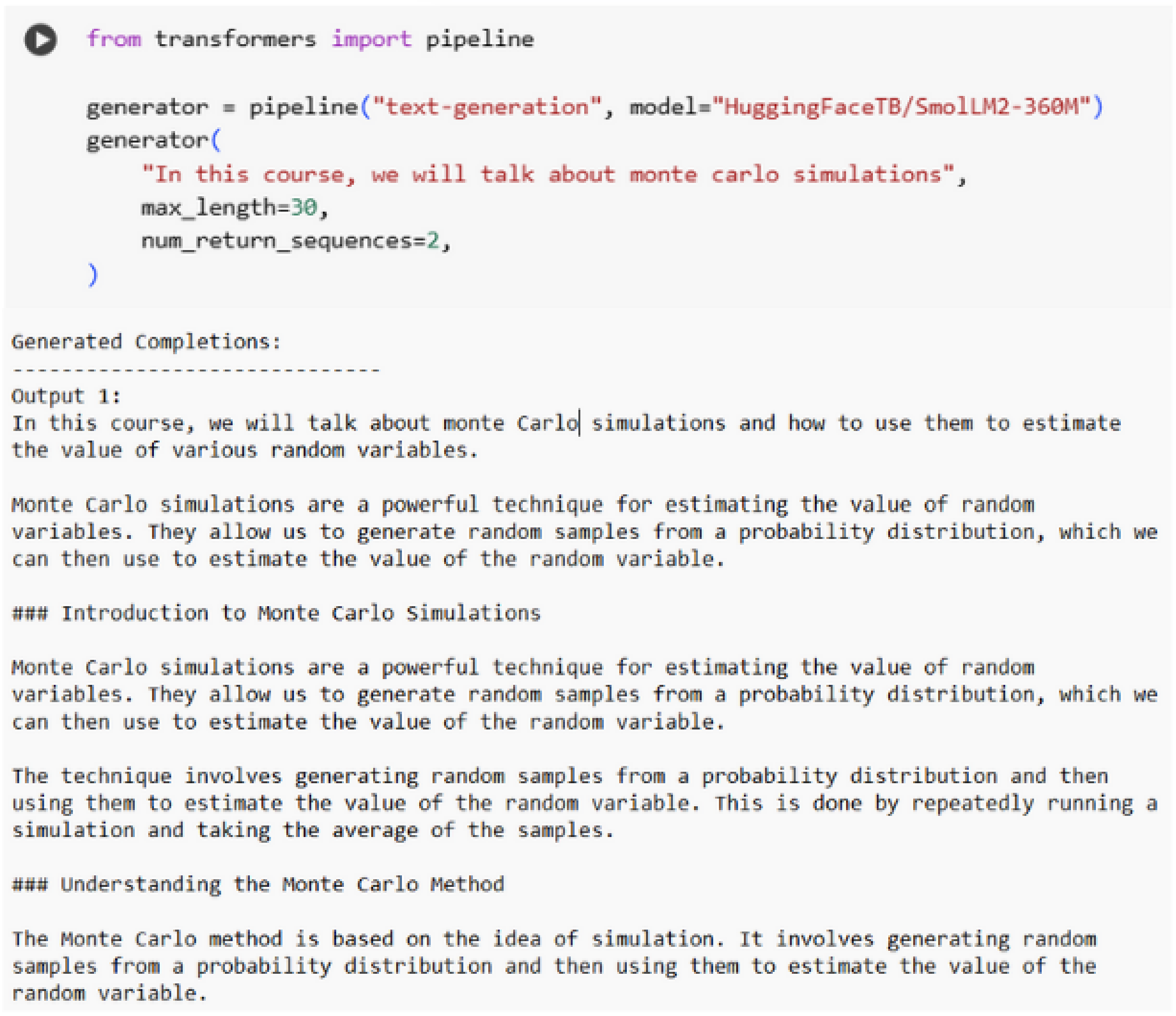
```
from transformers import pipeline
```

```
zero_shot=pipeline("zero-shot-classification")
zero_shot(
   "free palestine" ,
   candidate_labels=["education", "politics", "business"],
)
```

```
No model was supplied, defaulted to facebook/bart-large-mnli and revision d7645e1 (https://huggingface.co/facebook/bart-large-mnli).
Using a pipeline without specifying a model name and revision in production is not recommended.
Device set to use cpu
{'sequence': 'free palestine',
 'labels': ['politics', 'business', 'education'],
 'scores': [0.4827001690864563, 0.3060986399650574, 0.2112012356519699]}
```

Figure 1.2: Zero Shot Classification Pipeline.

Delving into the second example illustrated in Fig.1.3, which showcases the text generation pipeline, we just provide a prompt, select a model (refer to the Hugging Face documentation for the full list), and optionally adjust parameters like length or temperature (e.g., `max_length`, `temperature`)

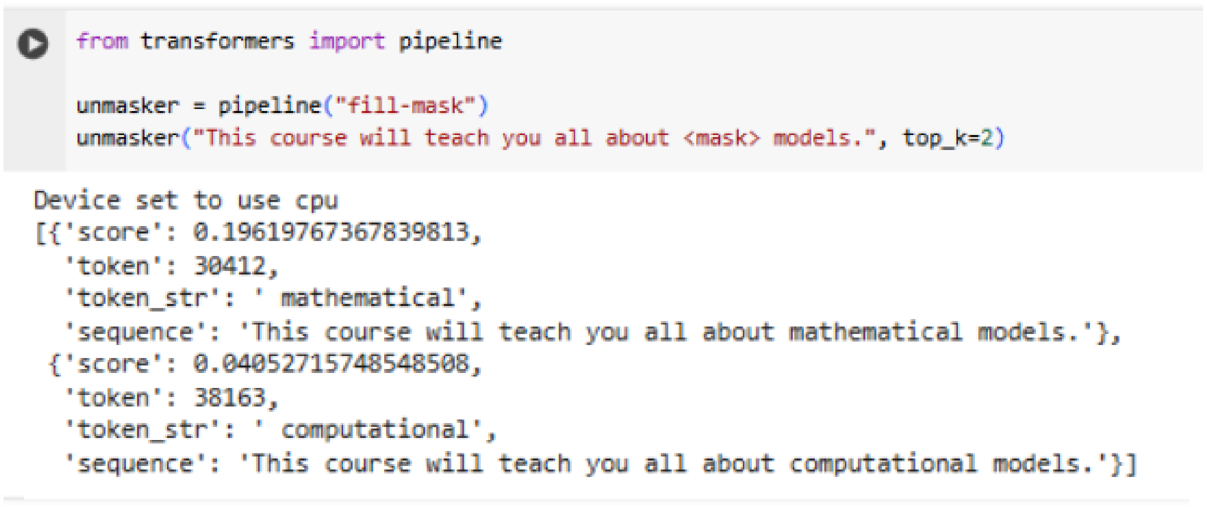


```
from transformers import pipeline

generator = pipeline("text-generation", model="HuggingFaceTB/SmolLM2-360M")
generator(
    "In this course, we will talk about monte carlo simulations",
    max_length=30,
    num_return_sequences=2,
)
```

```
Generated Completions:
------------------------------
Output 1:
In this course, we will talk about monte Carlo simulations and how to use them to estimate
the value of various random variables.

Monte Carlo simulations are a powerful technique for estimating the value of random
variables. They allow us to generate random samples from a probability distribution, which we
can then use to estimate the value of the random variable.

### Introduction to Monte Carlo Simulations

Monte Carlo simulations are a powerful technique for estimating the value of random
variables. They allow us to generate random samples from a probability distribution, which we
can then use to estimate the value of the random variable.

The technique involves generating random samples from a probability distribution and then
using them to estimate the value of the random variable. This is done by repeatedly running a
simulation and taking the average of the samples.

### Understanding the Monte Carlo Method

The Monte Carlo method is based on the idea of simulation. It involves generating random
samples from a probability distribution and then using them to estimate the value of the
random variable.
```

Figure 1.3: Text Generation Pipeline.

Moving on to the final pipeline example covered in this report, which is the fill-mask pipeline. The idea is pretty straightforward: it's all about filling in the blank. The model detects the missing part using the `<mask>` tag, as shown in Fig 1.4.



```python
from transformers import pipeline

unmasker = pipeline("fill-mask")
unmasker("This course will teach you all about <mask> models.", top_k=2)
```

```
Device set to use cpu
[{'score': 0.19619767367839813,
  'token': 30412,
  'token_str': ' mathematical',
  'sequence': 'This course will teach you all about mathematical models.'},
 {'score': 0.04052715748548508,
  'token': 38163,
  'token_str': ' computational',
  'sequence': 'This course will teach you all about computational models.'}]
```

Figure 1.4: Fill Mask Pipeline.

Hugging Face provides a variety of pre-trained and fine-tuned models throughout its pipelines. Down below is a table that summarizes the most used ones[13] .

Table 1.1: Hugging Face NLP Pipeline Cheat Sheet

Pipeline Task	**Identifier**	**What It Does**
Sentiment Analysis	`"sentiment-analysis"`	Classifies text as positive/negative
Text Classification	`"text-classification"`	General-purpose classification (multi-class)
Zero-Shot Classification	`"zero-shot-classification"`	Classifies into user-defined labels without retraining
Question Answering	`"question-answering"`	Answers questions based on a given context
Summarization	`"summarization"`	Summarizes long documents into shorter text
Translation	`"translation"`	Translates between languages
Text Generation	`"text-generation"`	Autocompletes text (GPT-style)
Text2Text Generation	`"text2text-generation"`	Translate, summarize, answer, etc. (Seq2Seq)
Named Entity Recognition	`"ner"`	Detects named entities like persons, locations, organizations
Conversational	`"conversational"`	Builds simple chatbot-style conversations
Feature Extraction	`"feature-extraction"`	Outputs hidden-state embeddings (for similarity, clustering, etc.)
Fill Mask	`"fill-mask"`	Predicts missing words in masked input

1.4 conclusion

In this chapter, we discovered a glimpse of the history of NLP from the early 1900s to the creation of Transformers, which paved the way for the rise of LLMs. We also saw some of the key Transformers pipelines to highlight how they simplify tasks. Now, it's time to reveal what happens behind the scenes of Transformers and see how they really work.

Chapter 2

Transformers Architecture

Transformers have revolutionized natural language processing (NLP) by replacing traditional sequence models like RNNs and LSTMs with a more powerful attention-based architecture. Before diving into this chapter, it's worth noting that its content can be seen as an elegant attempt to simplify and unify what was introduced in the renowned paper "***Attention Is All You Need***" [14].

2.1 Attention Mechanism

The answer to an efficient transformer lies in the name of its presented paper : *Attention is All You Need*, so in order to understand the transformers architecture we need to comprehend what the attention mechanism is:

The attention mechanism allows the model to focus on relevant parts of the input when generating each output token. Instead of processing inputs sequentially, attention allows for direct associations between any pair of words in the input, giving the model the capacity to capture long-range dependencies.

In simpler terms, attention can be thought of as a mechanism that allows each word to learn from the context of other words.

In order to better understand the concept of *self attention*, imagine that we want to translate the following sentence: *I did not stare at the lizard on the wall because i feared **it**.* (true story), the *self attention* mechanism helps the model to understand that "it" refers to the lizard and not the wall.

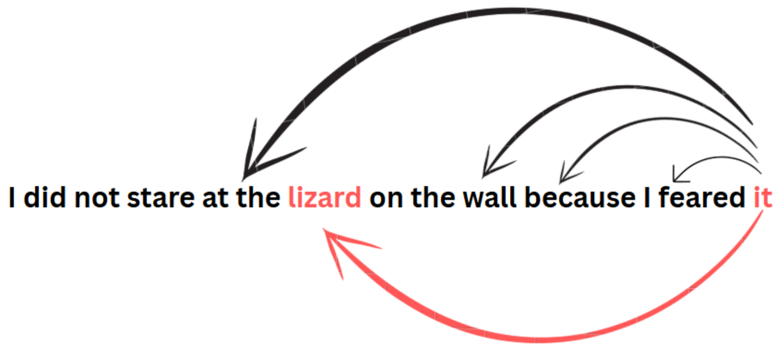

I did not stare at the lizard on the wall because I feared it

Figure 2.1: Self Attention Mechanism.

The key idea is to compute a weighted sum of values (**V**) based on the similarity between a query (**Q**) and a set of keys (**K**), using the formula:

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

where d_k is the dimension of the key vectors, used for scaling.
I came across an excellent explanation for the 3 vectors (Query, Key, Value): think of them as research of a LinkedIn profile. You search for a name (Query), the algorithm maps the query with the available profiles and companies in the database (Keys), and then it provides the profiles (Values).

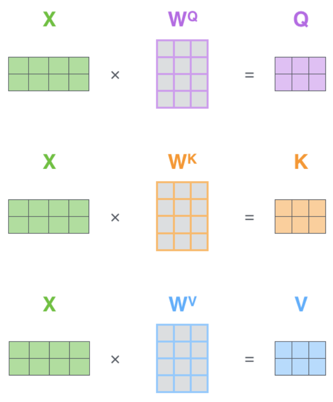


Figure 2.2: Query-Key-Value Illustration[1].

Let's try to walk through the story of what's really happening under the hood:
As shown in Fig 2.3 the **first** step is to embed the words of our input, if we have 2 words, then we get two vectors (a vector per word).

The **second** step is to create 3 vectors per word which are the Query, Key, Value vectors.
The **Third** step is to calculate the attention scores for each word, which is the dot product between its query and all keys.
Moving on to the **Fourth** step which is the scaling of the previous score by dividing each score by the square root of the key dimension d_k.
The **Fifth** step is to apply the softmax function in order to normalize the score and obtain a sum of 1. This paves the way to conclude the *self attention* by summing up the weighted value vectors in the **Sexth** step.

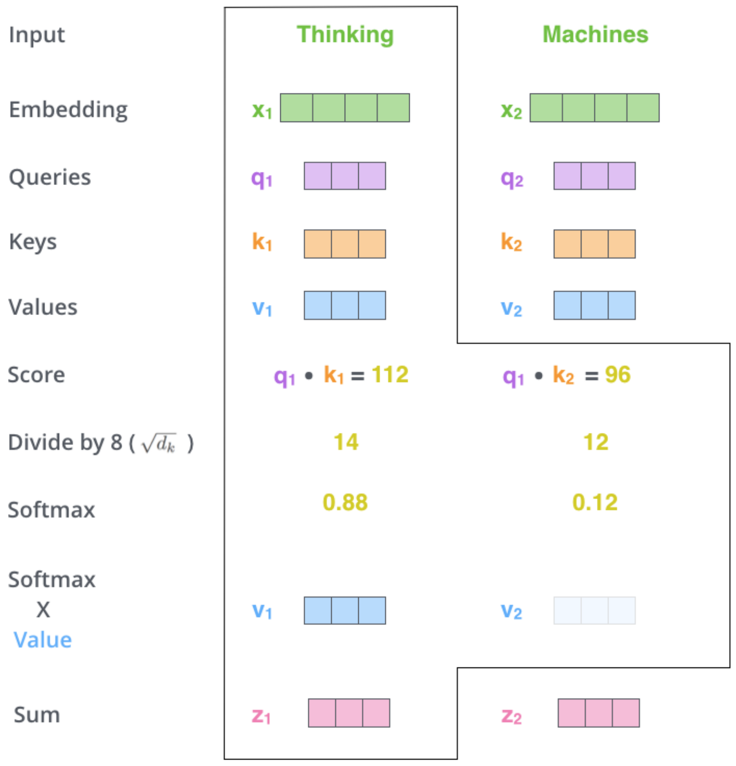


Figure 2.3: Self Attention Illustration[2].

Now that we understand the attention mechanism and some of its math, we can study the Transformers architecture and learn the way they function.

2.2 Core Blocks

In this section we will use Fig 2.5 as a reference in every step. At first sight we can distinguish two main blocks: an encoder on the left and a decoder on the right. The encoding component consists of a stack of encoders, and the decoding component mirrors it with an equal number of decoders. For example, if we want to translate the sentence ’ *je suis un étudiant* ’, it first passes through the encoder, then through the decoder to produce the output’ *I’m a student* ’.

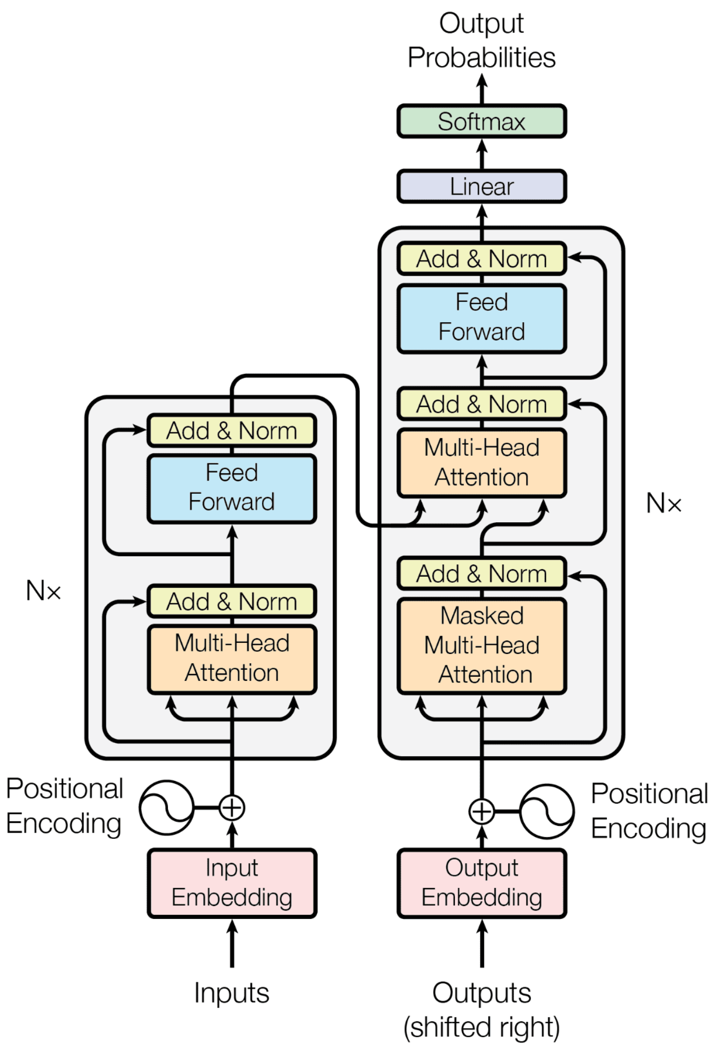


Figure 2.4: Transformers Architecture[3].

2.2.1 Embeddings

The very first step in this process is *the embedding* which is the transformation of a text into a vector. These vectors capture semantic similarity and are learned during training. Let x_i be the i-th token in the input sequence; the embedding layer transforms it into a vector $e_i \in \mathbb{R}^d$.

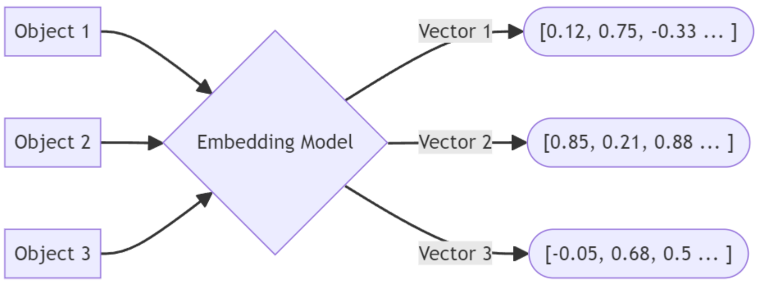


Figure 2.5: Embedding[4].

2.2.2 Positional Encoding

Transformers do not process tokens in order, thus embeddings include positional encodings to indicate token placements. The original study employs sinusoidal functions, namely a sine function for pair positions and a cosine function for unpaired ones:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

As illustrated in Fig2.6 these encodings are added to the embeddings to form the input to the encoder and decoder.

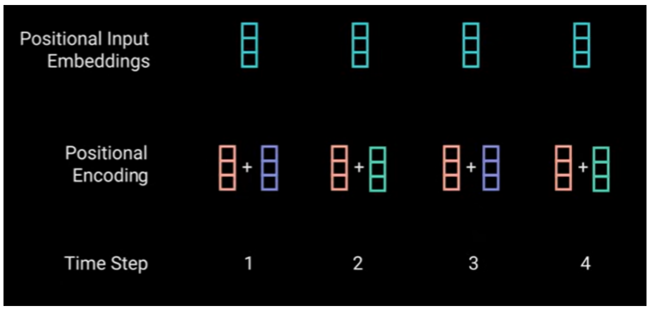


Figure 2.6: Positional Encoding[5].

2.2.3 Multi-head Attention

Moving forward to the next phase:

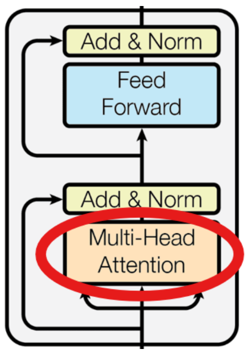


Figure 2.7: Multi-head Attention.

Building on the concept of Self attention outlined in 2.1, instead of computing a single attention function, multi-head attention divides the information into numerous heads. Each head develops a unique set of relationships. The outputs from all heads are concatenated and fed through a linear layer:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

where each head is:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

2.2.4 FFN + Residuals + LayerNorm

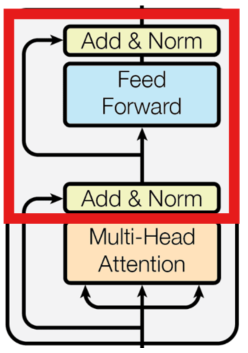


Figure 2.8: FFN + Residuals + LayerNorm.

After the multi-headed attention layer, each position in the sequence goes through a small neural network called a feed-forward network (FFN). This network is applied to each position separately and has the following form:

$$\mathrm{FFN}(x) = \mathrm{ReLU}(xW_1 + b_1)W_2 + b_2, \quad \text{with } \mathrm{ReLU}(z) = \max(0, z).$$

This means the input goes through two linear layers with a non-linearity (ReLU) in between. The ReLU function simply keeps only the positive parts of the data, making the model more flexible.

To help the model train better and faster, each layer (attention or FFN) is wrapped with two important techniques:

- **Residual connections**: Instead of just using the output of the layer, we add the original input back to it. This helps prevent the model from forgetting useful information. In simple terms: *output = input + layer result.*
- **Layer normalization**: After the residual connection, we normalize the result to make training more stable. This keeps the numbers in a controlled range and speeds up learning.

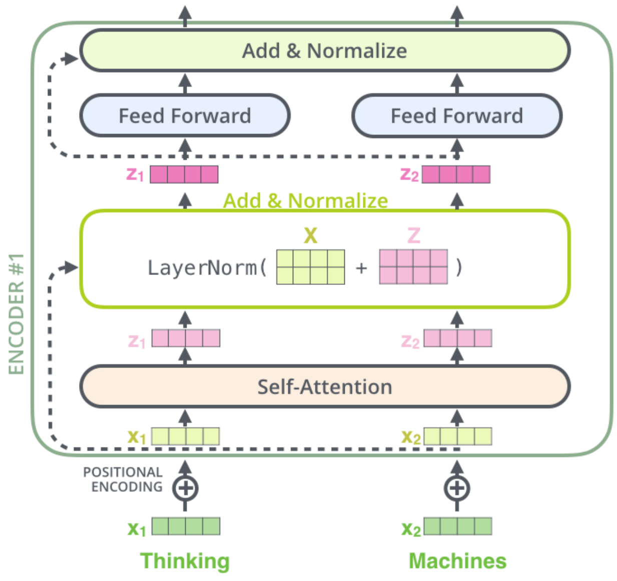


Figure 2.9: Feed-Forward Network with Residual and LayerNorm structure[6].

2.2.5 Output Embedding & Positional Encoding

As explained in 2.2.1 and 2.2.2, the output embedding and the positional encoding process are quite similar to the encoder.

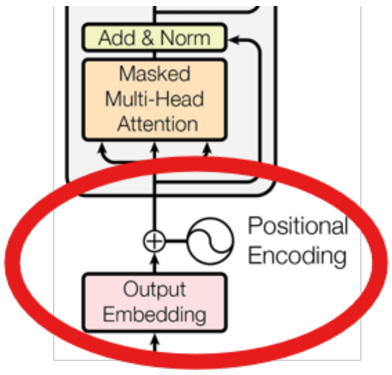


Figure 2.10: Output Embedding & Positional Encoding.

The decoder starts by transforming its input tokens into dense vectors using an embedding layer. Positional encodings are added to the embeddings to help the model understand the tokens order.

2.2.6 Decoder Multi-Headed Attention

As shown in Fig2.11 The decoder has two multi-headed attention layers:

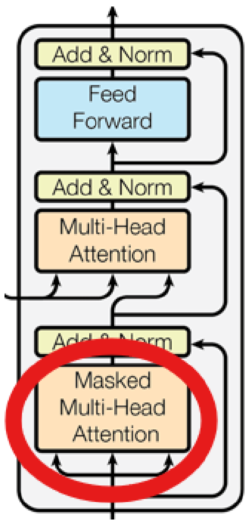


Figure 2.11: Decoder Multi-Headed Attention.

Each one computes attention scores. **However** the key **difference** compared to the encoder lies in the decoder's need to avoid seeing future words. For this reason, a mask—known as the look-ahead mask—is applied to prevent this from happening, as illustrated in Fig2.13.

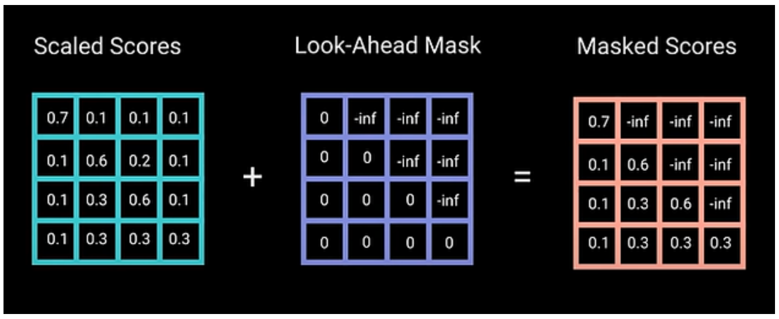


Figure 2.12: Look-Ahead Mask[5].

After passing through the first multi-head attention layer, the output goes through the feed-forward network (FFN), followed by the second attention block, which leads us towards the last step: The Linear Classifier.

2.2.7 Linear Classifier

In the decoder, the final layer is a linear transformation that has the size of classes , for example if we have 5000 words then logically we are going to have 5000 classes. This steps is followed by a softmax layer which produces a probability for each class, projecting the hidden state to vocabulary size V:

$$y = \text{softmax}(Wx + b)$$

The model then selects the most probable token based on this distribution.

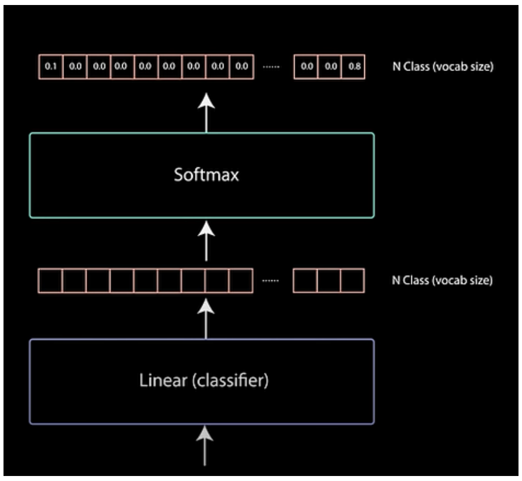


Figure 2.13: Linear Classifier[5].

2.3 Types of architecture

The Transformers architecture not only revolutionized the field of NLP and GenAI, but it also created three types of models, each with its unique utility, which we will discuss in this part.

2.3.1 Encoder-only (e.g., BERT)

Encoder-only models like BERT (Bidirectional Encoder Representations from Transformers) utilize just the encoder section of the Transformer architecture. These models aim to create deep contextual representations of text by attending to both sides, meaning each word is influenced by the words around it on both the left and right, also known as **bi-directionality**. This mechanism provides vectors that each one of them describes not only the word but also its meaning based on its neighbors. Giving the bi-directional property to encoder-only models, they are very efficient in tasks like finding the missing word in a sentence, like explained in Fig2.14.

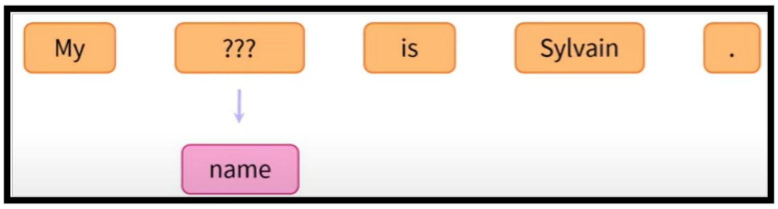


Figure 2.14: Encoder-Only Example[7].

This example can be mathematically expressed as the following:

$$P(y_t \mid y_{\neq t}) = \text{Encoder}(y_{\neq t})$$

- y_t: the token being predicted (the masked token)
- $y_{\neq t}$: all tokens in the sequence **except** the target token
- $\text{Encoder}(y_{\neq t})$: the encoder considers the full context (before and after t) to predict y_t.

Typical applications:

- Text classification

- Sentiment analysis
- Named entity recognition (NER)
- Question answering (extractive)

2.3.2 Decoder-only (e.g., GPT)

Decoder-only models, like GPT (Generative Pretrained Transformer), utilize only the decoder part of the Transformer. They are auto-regressive, meaning they generate text one token at a time by attending only to previous tokens

$$P(y_t \mid y_{<t}) = \text{Decoder}(y_{<t})$$

- y_t: the token predicted at position t
- $y_{<t}$: all tokens before position t
- $\text{Decoder}(y_{<t})$: the decoder-only Transformer uses the previous tokens to generate y_t.

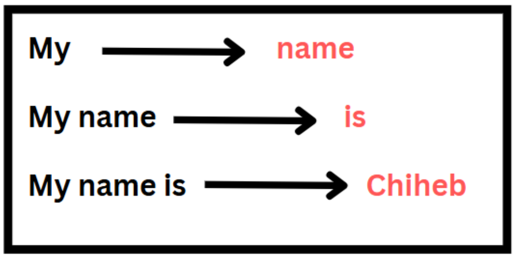


Figure 2.15: Generative Example with Decoder Only.

Typical applications:

- Text generation
- Dialogue systems (chatbots)
- Code generation
- Story or poetry writing

2.3.3 Encoder-Decoder (e.g., T5, Whisper)

Encoder-decoder models, such as T5 (Text-to-Text Transfer Transformer), combine both components:

$$P(y_t \mid y_{<t}, x) = \text{Decoder}(y_{<t}, \text{Encoded}(x))$$

- y_t : the token to be predicted at position t
- $y_{<t}$: all previously generated tokens $\{y_1, y_2, \ldots, y_{t-1}\}$
- x : the input sequence to the encoder
- $\text{Encoded}(x)$: the context representation of the input sequence from the encoder
- $\text{Decoder}(\cdot)$: the decoder function that uses previous tokens and encoder context to predict the next token
- $P(y_t \mid y_{<t}, x)$: the probability distribution over the vocabulary for the next token.

The bi-directional encoder transforms the input into a latent representation, which is then used by the auto-regressive decoder to generate the output. After generating the first token, the model no longer queries the encoder. Given its auto-regressive nature, the decoder continues the generation using its own previous outputs.

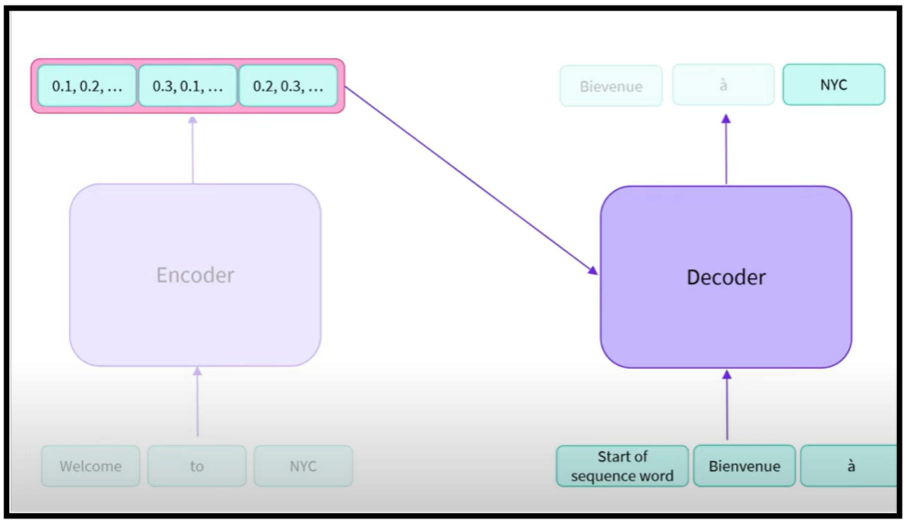


Figure 2.16: Encoder-Decoder Example Illustration[8].

Typical applications:

- Machine translation
- Text summarization
- Question answering (generative)

Down below is a synthetic table to sum up the 3 types of models:

Table 2.1: Summary of Transformer Model Types

Model Type	**Examples**	**Main Usage**
Encoder-only	BERT, RoBERTa	Understanding tasks (e.g., classification, NER, QA)
Decoder-only	GPT, GPT-2/3/4	Text generation, auto-completion
Encoder-Decoder	T5, BART	Seq2Seq tasks (e.g., translation, summarization)

2.4 Conclusion

In conclusion, this chapter has investigated the major components of the Transformer architecture, studied their linkages, and established the three model types based on this architecture. With this solid foundation, we are now ready to study the field of large language models (LLMs) and comprehend how they evolved from this design.

Chapter 3

Introduction to LLMs

3.1 Introduction

Nowadays, one of the most prevailing topics of discussion are large language models, or LLMs. LLMs are built on top of Transformers. As a matter of fact, most of today's popular LLMs such as BERT and GPT are direct implementations of specific Transformer configurations. And since we dived into the architecture of transformers in the previous chapter, now we have a head start in understanding LLMs.

This chapter will emphasize how those architecture blocks evolve into full-scale language models, what these models are capable of, and how to work with them efficiently.

3.2 What Large Language Models Are

LLMs are machine learning models that can comprehend and create human language. They can be thought of as a large neural network with billions of parameters. The core idea of LLMs is that the user provides its input, and the model attempts to finish it in the most likely way using patterns it has learned. Below is a table with the most commonly used LLMs and their characteristics. [15]:

Table 3.1: Popular LLMs, Their Creators, and Parameter Counts

Model	**Parameters**	**Company / Lab**
GPT-3	175B	OpenAI
GPT-4	more than 500B	OpenAI
LLaMA 2	7B / 13B / 70B	Meta AI
PaLM 2	340B	Google DeepMind
Mistral	7B	Mistral AI
Mixtral	12.9B (MoE)	Mistral AI
Command R+	35B	Cohere

LLMs are trained on massive text datasets from books, websites, and more. After training, they can answer questions, summarize content, translate text, and write code, all by predicting the next word, one token at a time.

3.3 Inference and Phases

Inference is the process of using a pretrained model to generate responses given an input prompt. It can be divided into two major phases[16]:

- **Pretraining:** This phase lays the foundation for the model by preparing the tokens, embedding them, and processing them for further analysis

Table 3.2: Key Steps in the Prefill Phase

Step	**Description**
Tokenization	Segmenting the input text into discrete units (tokens), which serve as the foundational elements the model can interpret.
Embedding Mapping	Converting tokens into dense vector representations that encapsulate semantic meaning and syntactic roles.
Contextual Processing	Passing embeddings through the model's neural architecture to generate context-aware representations.

- **Decode Phase:** After the prefill phase, the model enters the decode phase, which is when it begins generating text. The model generates one token at a time, with each new one based on the one before it(The auto-regressive mechanism). The decode phase contains several important steps that can be summarized in the following table:

Table 3.3: Key Steps in the Decode Phase

Step	**Description**
Attention Computation	The model looks at all previously generated tokens to understand the context.
Probability Calculation	It calculates the probability of each possible next token.
Token Selection	Based on the scores, the most likely token is selected.
Continue or Stop	The model checks whether to keep generating or stop.

3.4 Sampling Strategies

When large language models (LLMs) generate text, they do not just select the most likely word at each instance; this method would result in monotonous and repeating responses. Instead, they use sampling techniques to improve the diversity and human-like quality of the output. Some common strategies are in the next table [16]:

Table 3.4: Sampling and Decoding Strategies in LLMs

Component	**Description**
Token Decision	Raw logits are passed through: • **Temperature**: Controls randomness. • **Top-k**: Samples from top k tokens. • **Top-p**: Chooses tokens whose total prob. p.
Repetition Control	Discourages repetition using: • **Presence penalty**: Penalizes seen tokens. • **Frequency penalty**: Penalizes frequent tokens.
Length Control	• **Token limits**: Sets a hard cap on output. • **Stop sequences**: Stops at predefined phrases. • **EOS token**: Ends generation naturally.
Beam Search	Explores multiple output paths: • Keeps top n best sequences. • Tends toward high-probability, low-diversity results.

By tuning these settings, we can manage not only the length but also the tone of responses, whether we want them to be creative, professional, funny, etc.

3.5 Limitations of LLMs

As revolutionary and advanced as they may seem, LLMs will inevitably face some challenges that should be taken into consideration while developing and maintaining them. The most crucial challenges are:

- **Hallucination**: The model can start making stuff up and consider them true even after a negative comeback.
- **Lack of true understanding**: They mimic patterns, not reasoning.
- **Bias**: The LLMs reflect the data they were trained on; if that data is biased, the outputs are likely to be biased too.
- **High computational costs and environmental impact**: Training and running LLMs require massive computing resources, which consume a lot of energy and can harm the environment.
- **Prompt sensitivity**: LLMs can give different answers depending on how the question is phrased and may fail entirely with poorly structured prompts, a point that is going to be treated in the next section.

3.6 Prompt Engineering

Prompt engineering can be considered as the convention of how to talk to LLMs; every word can contribute to a different answer. That's why we should understand the different techniques of prompting.

3.6.1 Elements of a Prompt

A prompt should be composed of the following components[17]:

Table 3.5: Key Components of a Prompt

Component	**Description**
Instruction	A precise task or command given to the model.
Context	Additional information that helps steer the model toward better responses.
Input Data	The specific input or question for which a response is sought.
Output Indicator	Guidance on the expected format or type of output.

Lets treat this example in order to understand the concept of these elements:

Example Prompt:

- **Instruction:** Summarize the following text in two sentences.
- **Context:** This article discusses the impacts of climate change on agriculture.
- **Input Data:** *"Rising temperatures and unpredictable rainfall patterns threaten crop yields worldwide."*
- **Output Indicator:** Provide a concise summary.

3.6.2 Prompting Techniques

There are a lot of prompting techniques, each one can give a different response to a different task in order to guarantee the reliability and performance of LLMs[18].

Table 3.6: Common Prompt Engineering Techniques

Technique	**Description**
Zero-shot prompting	Asking the model to perform a task without any examples.
Few-shot prompting	Providing a few examples to guide the model's behavior.
Chain-of-thought prompting	Encouraging step-by-step reasoning by explicitly prompting for it.
Instruction tuning	Giving clear, structured instructions (e.g., "Summarize this article in 3 points").
Role prompting	Assigning the model a persona or role (e.g., "You are a helpful assistant…").
Output constraints	Specifying output format or limitations (e.g., "Answer in JSON format").
Decomposition prompting	Breaking a complex task into simpler, sequential subtasks.

Lets try to see some examples:

- **Zero-Shot Prompting:**

  Bad Prompt:

  `Translate this.`

  Improved Prompt:

  `Translate the following English sentence to French: ‘‘Where is the library?’’`

- **Few-Shot Prompting:**

  Bad Prompt:

  `Convert the sentences to French.`

Improved Prompt:

`English: "Good morning" → French: "Bonjour"`

`English: "How are you?" → French: "Comment ça va ?"`

`English: "Where is the library?" →`

- **Chain-of-Thought (CoT) Prompting:**

  Bad Prompt:

  `If a train leaves at 3 PM going 60 mph and another leaves at 4 PM going 80 mph, when will they meet?`

  Improved Prompt:

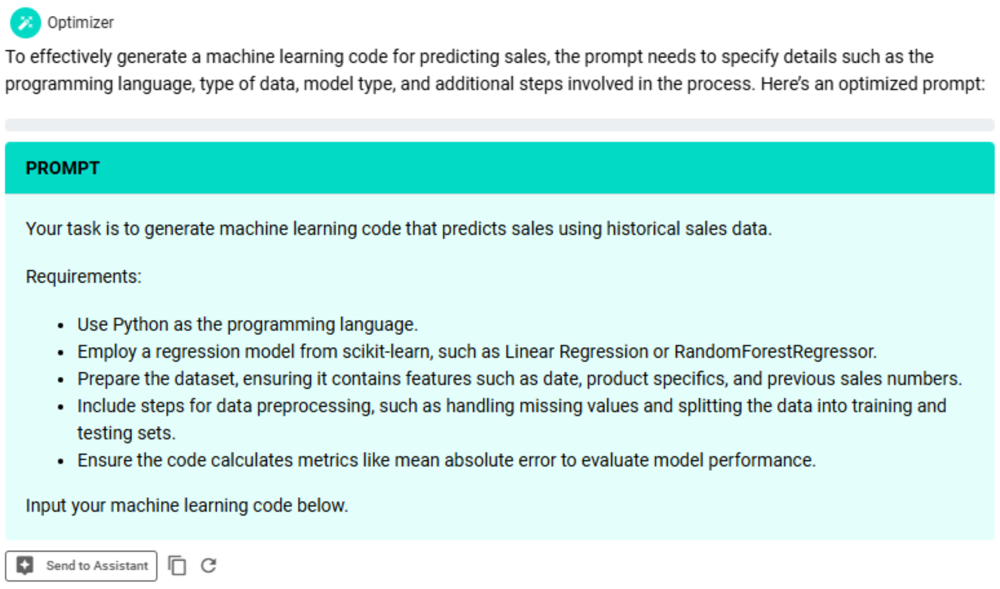
  `Let's think step by step. A train leaves at 3 PM traveling 60 mph. Another leaves at 4 PM traveling 80 mph. How long after 4 PM will they meet?`

While the prompting techniques differ, there are some ground rules that always stay efficient. Here are some of them:

- Be clear and specific in your instructions.
- Provide examples whenever possible.
- Specify the role or persona the model should assume (e.g., "Act as a teacher...").
- Break complex tasks into smaller, manageable steps.

A good prompt can make the difference between a vague answer and another helpful and precise one.

Here is one last example used by ***promptperfect.jina.ai***[19], a platform designed to optimize the user's prompt. I tried the following input: "***optimize this prompt : write a code of machine learning to predict sales***" And we can observe the big difference in Fig3.1 between the two prompts, which means a better output.



To effectively generate a machine learning code for predicting sales, the prompt needs to specify details such as the programming language, type of data, model type, and additional steps involved in the process. Here's an optimized prompt:

PROMPT

Your task is to generate machine learning code that predicts sales using historical sales data.

Requirements:

- Use Python as the programming language.
- Employ a regression model from scikit-learn, such as Linear Regression or RandomForestRegressor.
- Prepare the dataset, ensuring it contains features such as date, product specifics, and previous sales numbers.
- Include steps for data preprocessing, such as handling missing values and splitting the data into training and testing sets.
- Ensure the code calculates metrics like mean absolute error to evaluate model performance.

Input your machine learning code below.

Figure 3.1: Optimized Prompt with Promptperfet.jina.ai.

3.7 Conclusion

In this chapter we discovered the essential elements of large language models, as well as how different sorts of prompting affect the model's answer, and observed their impact on the model's response.

Conclusion

Restatement of Objectives

This work aimed to explore *the interplay between Transformers and Large Language Models (LLMs)*, focusing on *The attention mechanism, Transformers architecture, and prompting.*

Summary of Key Contributions

The research presented in this document can be summarized as follows:

- A detailed breakdown of the Transformer architecture, including self-attention and multi-head attention mechanisms.
- Comparison of encoder-only, decoder-only, and encoder-decoder models.
- A practical exploration of prompt engineering techniques with applications in LLMs.

Reflections and Insights

In this work, I aimed to explore how Transformer models function, beginning with their basic principles and ending with their application as Large Language Models (LLMs). Each chapter added to the last, slowly linking concepts, from the elegant mathematics of self-attention to the complex decoding methods of LLMs. Writing this report not only helped me understand the structure and functions of these models but also highlighted the significance of interpretability, efficiency, and responsible design.

Limitations

Despite the comprehensive nature of this work, it still presents some limitations to consider:

- **Limited scope:** Due to time and space, some topics like training methods and model fine-tuning were not fully covered.
- **No experiments:** The report focuses on explaining concepts and does not include any practical experiments or results.
- **Fast-changing field:** LLMs evolve quickly, so some parts of this report may become outdated soon.

Future Work

There are several directions to continue this work:

- **Practical implementation:** Try building a small Transformer model to apply what was learned.
- **Ethics and safety:** Look deeper into responsible use, fairness, and transparency in language models.
- **Real world use:** Explore how LLMs are used in apps like chatbots, assistants, or creative tools.
- **Further analysis:** Add comparisons, benchmarks, or visualizations to support the explanations.

Closing Remarks

When I started writing this report, my mind was filled with questions and curiosity. Exploring the architecture of Transformers allowed me to construct a coherent narrative and visualize the underlying processes. As Large Language Models continue to evolve and redefine the boundaries of human-computer interaction, understanding their internal mechanisms and promoting responsible usage becomes increasingly vital. I hope that this work offers a modest step toward fostering that understanding.

Bibliography

[1] Query-key-value illustration. https://jalammar.github.io/images/t/self-attention-matrix-calculation.png. Consulted on June 24th, 2025.

[2] Self attention illustration. https://jalammar.github.io/images/t/self-attention-output.png. Consulted on June 24th, 2025.

[3] Transformers architecture. https://machinelearningmastery.com/wp-content/uploads/2021/08/attention_research_1.png. Consulted on June 24th, 2025.

[4] Embedding. https://www.couchbase.com/blog/wp-content/uploads/2024/02/image1-1.png. Consulted on June 24th, 2025.

[5] Positional encoding. https://www.youtube.com/watch?v=4Bdc55j80l8&ab_channel=TheAIHacker. Consulted on June 24th, 2025.

[6] Feed-forward network with residual and layernorm structure. https://jalammar.github.io/images/t/transformer_resideual_layer_norm_2.png. Consulted on June 24th, 2025.

[7] Encoder-only illustration. https://youtu.be/MUqNwgPjJvQ?t=196, . Consulted on June 24th, 2025.

[8] Encoder-decoder illustration. https://youtu.be/0_4KEb08xrE?t=254, . Consulted on June 24th, 2025.

[9] Ferdinand de saussure. https://classiques.uqam.ca/collection_methodologie/saussure_ferdinand_de/saussure_ferdinand_de_photo/saussure_ferdinand_de_photo.html. Consulted on June 23rd, 2025.

[10] Alan turing. https://www.britannica.com/biography/Alan-Turing. Consulted on June 23rd, 2025.

[11] Nlp models. https://medium.com/@vaniukov.s/nlp-vs-llm-a-comprehensive-guide-to-unde Consulted on June 23rd, 2025.

[12] From nlp to llms history. https://medium.com/@vaniukov.s/nlp-vs-llm-a-comprehensive-guide-to-understanding-key-differences-0358f6571910. Consulted on June 23rd, 2025.

[13] Hugging face transformers pipelines. https://huggingface.co/learn/llm-course/chapter1/3?fw=pt#text-generation, . Consulted on June 23rd, 2025.

[14] Attention is all you need. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. Consulted on June 23rd, 2025.

[15] Llms and parameters counter. https://huggingface.co/models?pipeline_tag=text-generation&sort=trending. Consulted on June 25th, 2025.

[16] Llms inference and sampling. https://huggingface.co/models?pipeline_tag=text-generation&sort=trending, . Consulted on June 25th, 2025.

[17] Elements of a prompt. https://www.promptingguide.ai/introduction/elements, . Consulted on June 25th, 2025.

[18] Prompting techniques. https://huggingface.co/learn/llm-course/chapter1/8?fw=pt, . Consulted on June 25th, 2025.

[19] Prompt optimizer jinaai. https://promptperfect.jina.ai/interactive. Consulted on June 25th, 2025.