

Open in app ↗

Medium

 Search Write

# Understanding Next Token Prediction: Concept To Code: 1st part!



Akash Kesrwni

Follow

7 min read · Sep 8, 2023



100



2



*Note: We're going to develop a deep-dive understanding of the mechanism of the next token Prediction with all concepts & code. I just Break down this blog into 3 subparts to get an depth overview of it! Gonna attach the link of all parts at the end of the subblogs.*

Despite all that has been accomplished with large language models (LLMs), the underlying concept that powers all of these models is simple — we just need to accurately predict the next token! Though some may (reasonably) argue that recent research on LLMs goes beyond this basic idea, next token prediction still underlies the pre-training, fine-tuning (depending on the variant), and inference process of all causal language models, making it a fundamental and important concept for any LLM practitioner to understand.

| *"It is perhaps surprising that underlying all this progress is still the original autoregressive mechanism for generating text, which makes token-level decisions* |

| *one by one and in a left-to-right fashion.”*

## Relevant Background Concepts

Before delving into the main topic of this overview, there are several key concepts that we must grasp. In this section, we will provide a brief overview of these essential ideas and offer links for further reading on each of them.

### 1)Transformer Architecture:

First and foremost, it's crucial to have a solid understanding of the transformer architecture, especially the decoder-only variant. Fortunately, we have extensively covered these concepts in previous discussions:

- Learn about the [Transformer Architecture](#)
- Explore: [Decoder-Only Transformers](#)

### 2) Self-Attention:

To comprehend the inner workings of the transformer architecture, we must also grasp the concept of self-attention and its pivotal role. Specifically, the large causal language models we'll be examining employ a specific form of self-attention known as [multi-headed causal self-attention](#).

### 3) Training Neural Networks with PyTorch:

The code examined in this overview is written in PyTorch and heavily relies on distributed training techniques, such as distributed data-parallel (DDP) training. To get a grasp of the fundamentals of PyTorch and distributed training, consider reviewing the following resources:

- Understanding Neural Networks in PyTorch
- PyTorch Distributed Training Overview
- Leveraging Distributed Data-Parallel in PyTorch

#### 4) Automatic Mixed Precision (AMP) Training:

In addition to basic and distributed neural network training in PyTorch, we will also encounter the use of *automatic mixed precision (AMP)* training. AMP selectively adjusts the precision of computations within the neural network during training, switching between full precision (float32) and half-precision (*float16 or bfloat16*) to enhance efficiency. For a more detailed and practical understanding of AMP, please refer to this resource [\[link\]](#).

#### 5) Deep Learning Basics:

This overview assumes a foundational understanding of neural networks, including their training and application. To acquire this knowledge, I highly recommend the “*Practical Deep Learning for Coders*” course offered by fast.ai. This course is frequently updated and is, in my opinion, the best practical introduction to deep learning available [\[link\]](#).

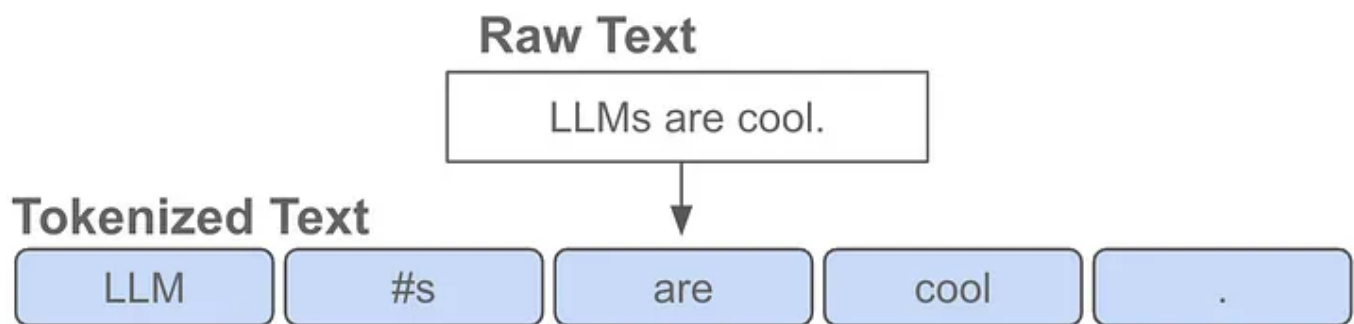
By familiarizing ourselves with these fundamental concepts and resources, we will be well-prepared to delve deeper into the main topic of this overview.

## Understanding Next Token Prediction

Let's dive into the concept of next-token prediction, which is the core task for causal language models. To understand this, we'll start with some fundamental ideas about tokens and how they relate to language models.

## Tokens and Vocabulary:

When we talk about tokens, think of them as individual units of text, like words or even smaller parts of words. The first thing we do when using a language model is to take a piece of text and break it down into these discrete tokens. This process is called tokenization, and it's essential for language models to understand and work with text effectively: See below for an example.



*To perform tokenization*, we use a specialized tool called a tokenizer. This tokenizer has been trained on a large amount of text from various sources to learn a fixed and specific set of tokens. This set of tokens is what we call the vocabulary of the language model.

*Now*, why is this vocabulary important? Well, it's because the language model can only understand and generate text using the tokens in its vocabulary. So, the vocabulary contains all the words or sub-words that the language model knows.

It's crucial to make sure that the training data used for the tokenizer is representative of the kind of text the model will encounter during both training and actual use. This way, the tokens the model knows are likely to match the ones it encounters in real-world text.

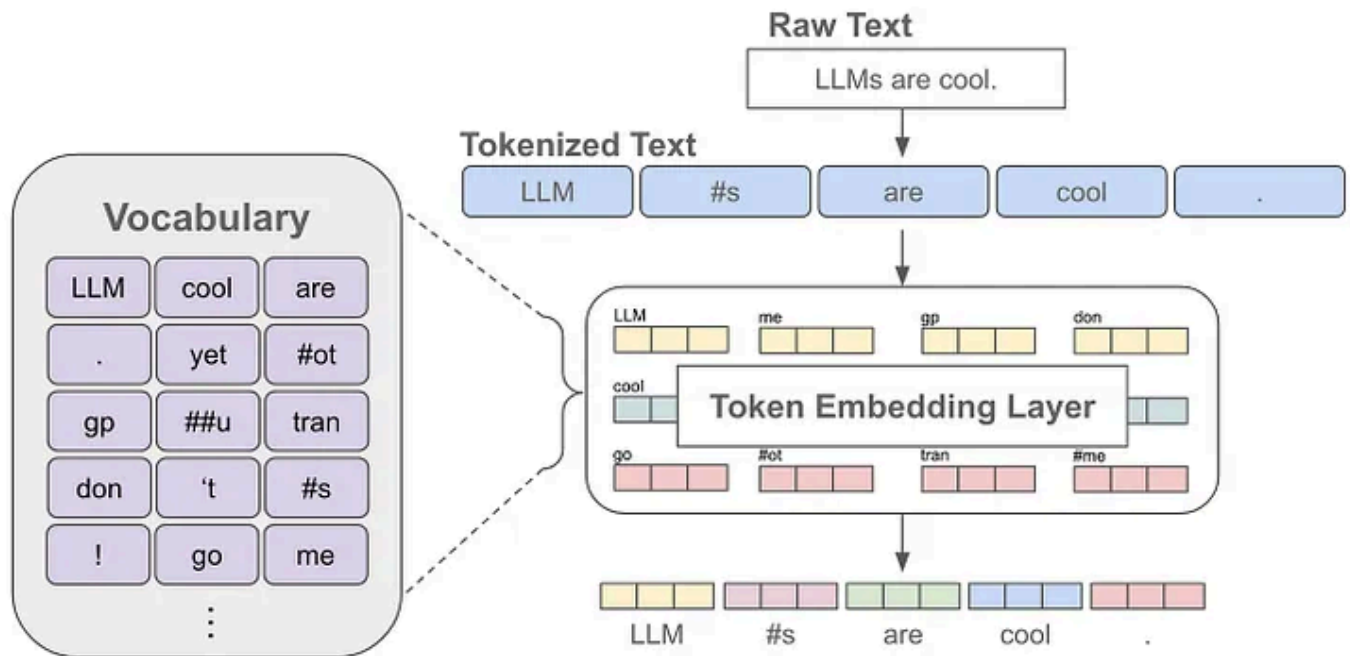
*So, in simple terms, tokens are the building blocks of text, and the vocabulary is like the dictionary that the language model uses to understand and generate text. This understanding of tokens and vocabularies is crucial as we explore the next token prediction and how it drives language models.*

## Tokenization techniques

Numerous different tokenization techniques exist; see [here](#) for an overview. For details on training and using popular tokenizers for LLMs, see [this article](#) that details the byte pair encoding (BPE) tokenizer — *the most commonly used tokenizer for LLMs*. Another tokenization technique that has become recently popular is [byte-level BPE \(BBPE\)](#), which relies upon bytes (instead of textual characters) as the basic unit of tokenization.

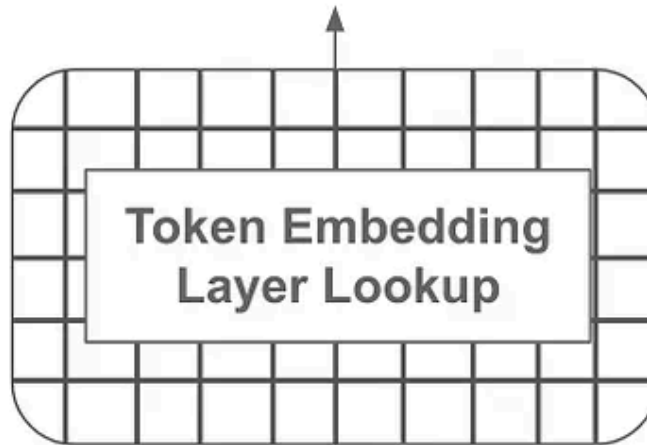
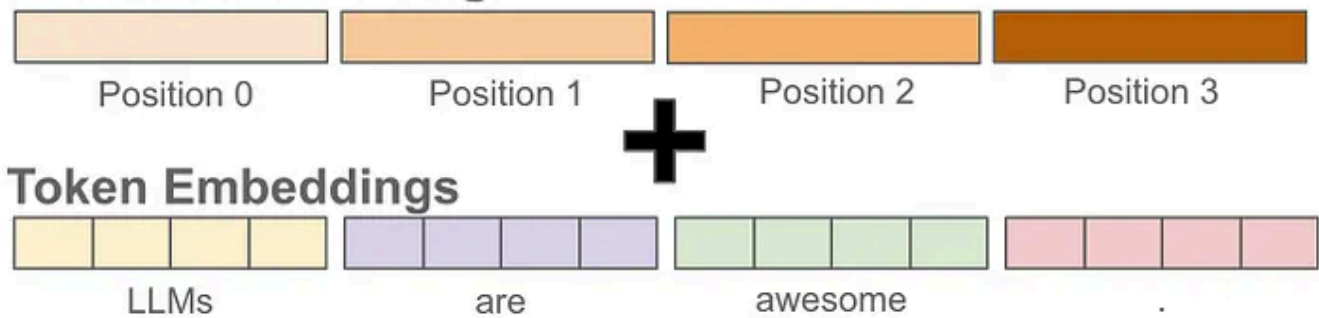
## Token embeddings

Once we have tokenized our text, we look up the embedding for each token within an embedding layer that is stored as part of the language model's parameters. After this, the sequence of textual tokens constructed from our input becomes a sequence of token embedding vectors; see below.



There is one final step required to construct the input that is actually passed to our decoder-only transformer architecture — *we need to add positional embeddings*. Positional embeddings are the same size as token embeddings and treated similarly (i.e., they are stored as part of the language model and trained along with other model parameters). Instead of associating an embedding with each unique token, however, we associate an embedding with each unique position that can exist within a tokenized input; see below for a depiction.

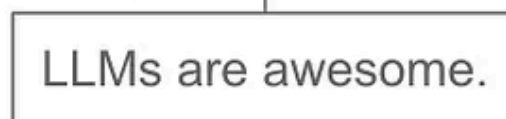
## Position Embeddings



## Tokenized Text

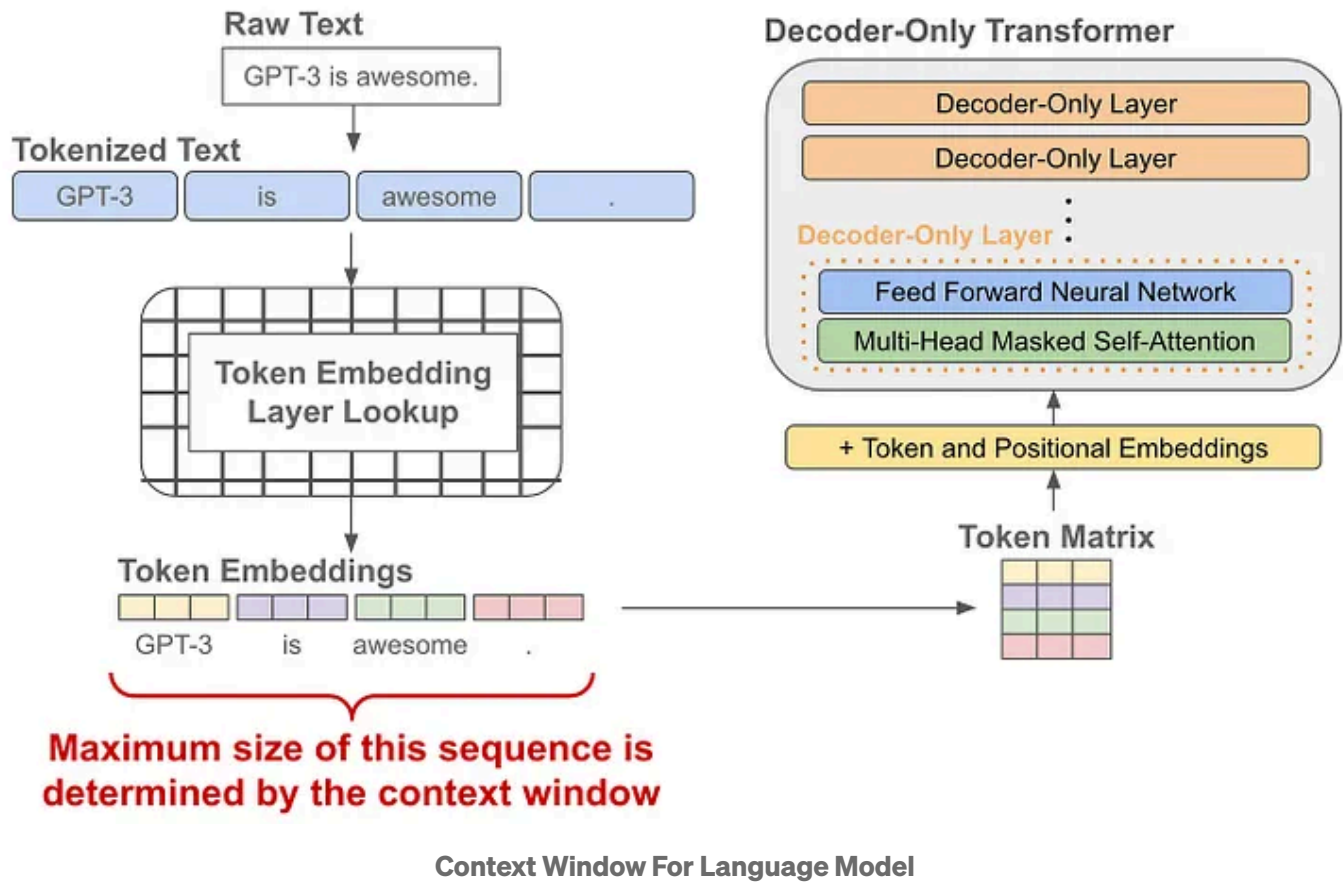


## Raw Text



Position Embeddings With In a Language Model

We add these embeddings to the token embeddings at the corresponding position. Such additive positional embeddings are necessary because the self-attention operation does not have any way of representing the position of each token. By adding positional embeddings, we allow the self-attention layers within the transformer to use the position of each token as a relevant feature during the learning process. Recent research has explored novel techniques for injecting positional information into self-attention, resulting in techniques like RoPE.

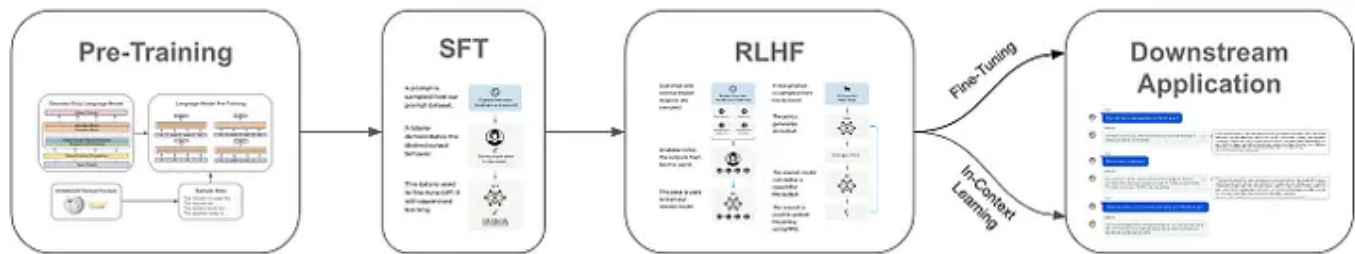


## Context window

Language models are pre-trained with token sequences of a particular size, which is referred to as the size of the context window or the context length. This size — *typically somewhere in the range of 1K to 8K tokens* (though some models are much larger!) — is (usually) selected based on hardware and memory constraints. Given that we only learn positional embeddings for input of this length, the context window limits the amount of input data that an LLM can process. However, recent techniques like ALiBi [7] have been developed to enable extrapolation to inputs longer than those seen during training.

## Language Model Pretraining

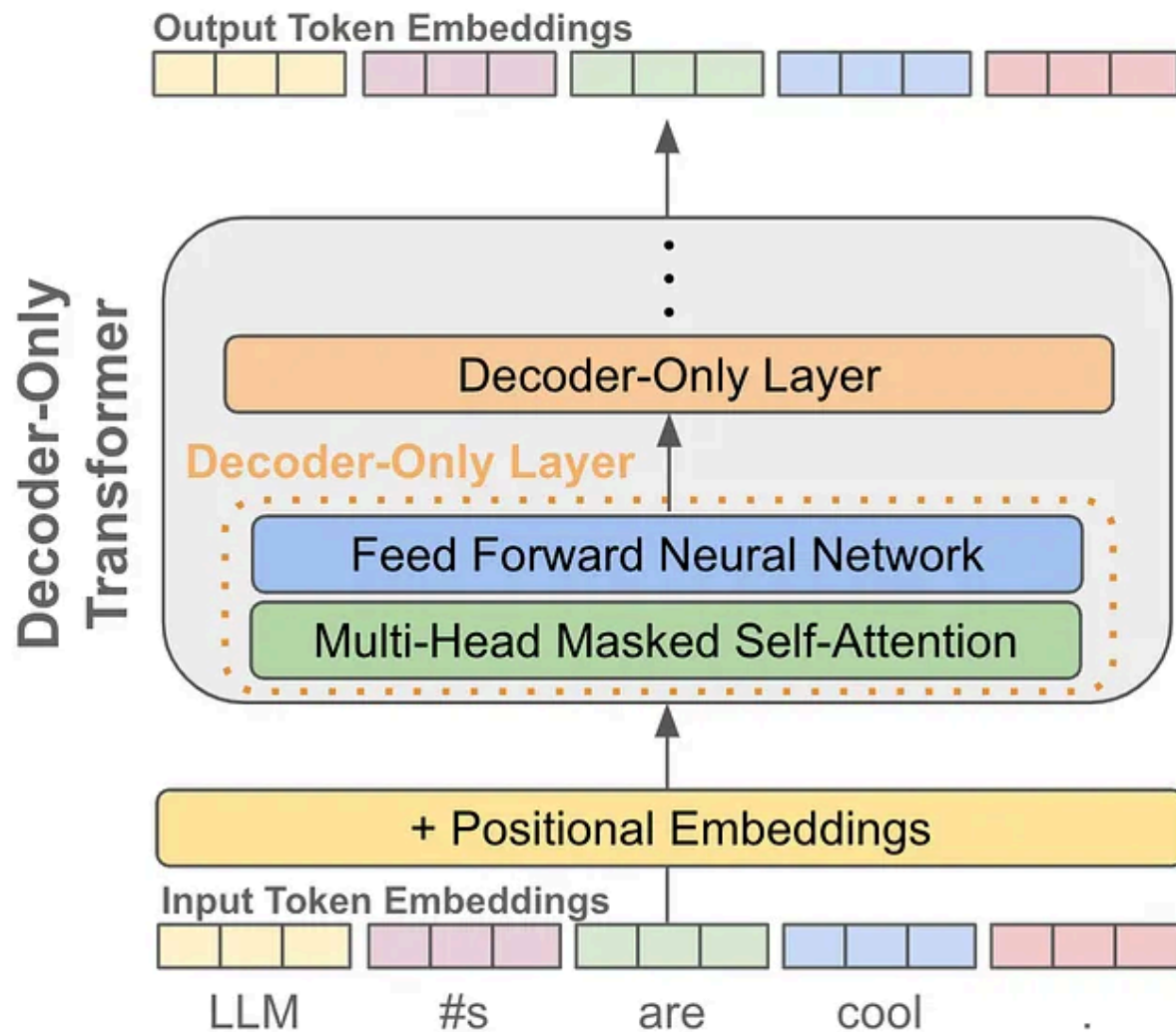




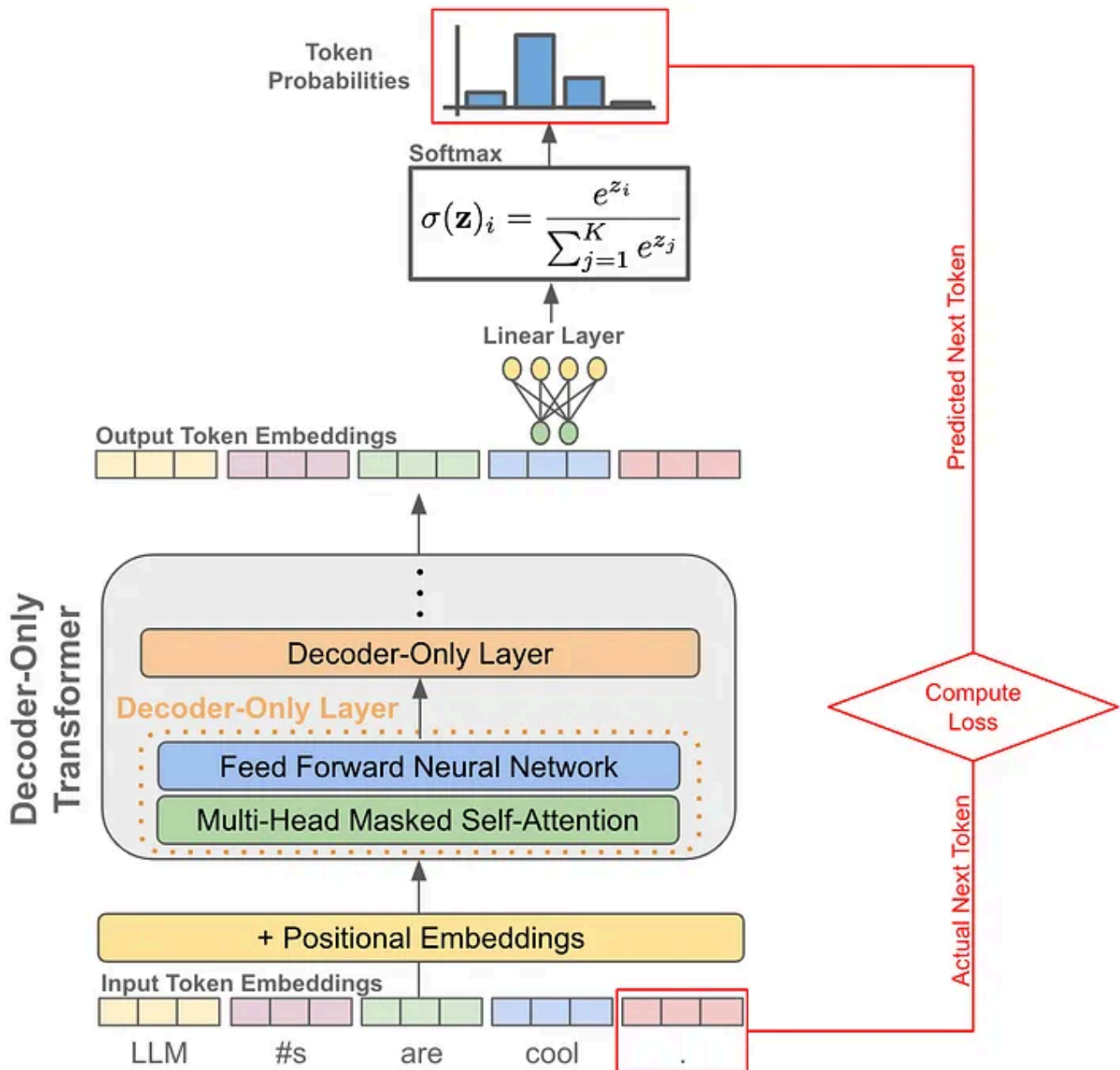
Language models are trained in several steps, as shown above. The first (and most computationally expensive) step is pretraining, which we will focus on within this overview. During pretraining, we get a large corpus of unlabeled text and train the model by *i)* sampling some text from the dataset and *ii)* training the model to predict the next word. This is a self-supervised objective due to the fact that no labels are required. Rather, the ground truth next token is already present within the corpus itself — *the source of supervision is implicit*. Such a training objective is referred to as the next token prediction or the standard language modeling objective.

## Predicting the next token

After we have our token embeddings (with position embeddings), we pass these vectors into a decoder-only transformer, which produces a corresponding output vector for each token embedding; see below.



Given an output vector for each token, we can perform the next token prediction by *i)* taking the output vector for a token and *ii)* using this to predict the token that comes next in the sequence. See below for an illustration.



As we can see above, the next token is predicted by passing a token's output vector as input to a linear layer, which outputs a vector with the same size as our vocabulary. After a softmax transformation is applied, a probability distribution over the token vocabulary is formed, and we can either *i*) sample the next token from this distribution during inference or *ii*) train the model to maximize the probability of the correct next token during pretraining.

Ah, nah it's not over yet.....!

| *“For further reading follow the link to get an in-depth overview of 2nd parts of it.”*

## Understanding the Next Word Prediction: Concept & Code 2nd

*Coming.....*

**Thanks!**



**Written by Akash Kesrwani**

55 followers · 13 following

Follow

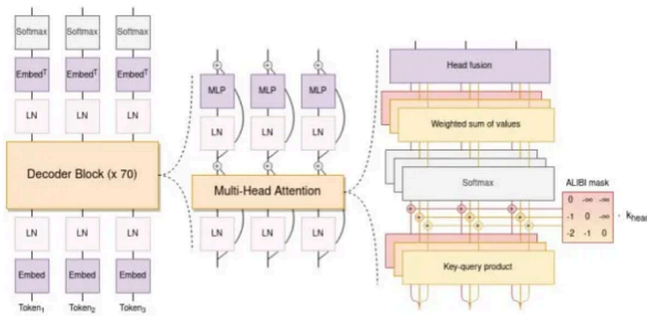
## Responses (2)



Scott Lai

What are your thoughts?

## More from Akash Kesrwni

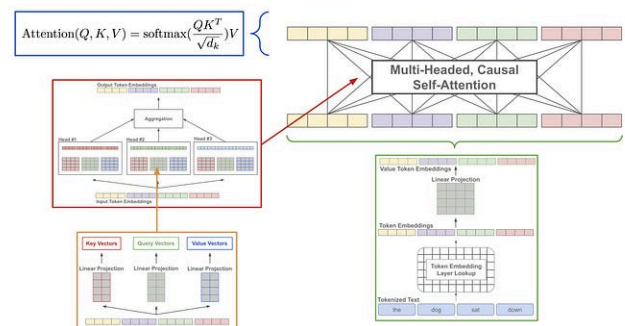


Akash Kesrwni

### What are LLM(Large Language Model)?

Large language models (LLMs) are powerful machine-learning models that can...

Jul 3, 2023 🖱 137 💬 2



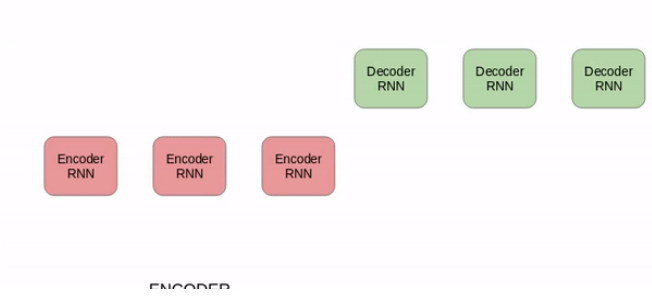
Akash Kesrwni


### Multi-Head Self Attention: Short Understanding

Each “block” of a large language model (LLM) is comprised of self-attention and a feed-...

Sep 8, 2023 🖱 126 💬 2





 Akash Kesrwani

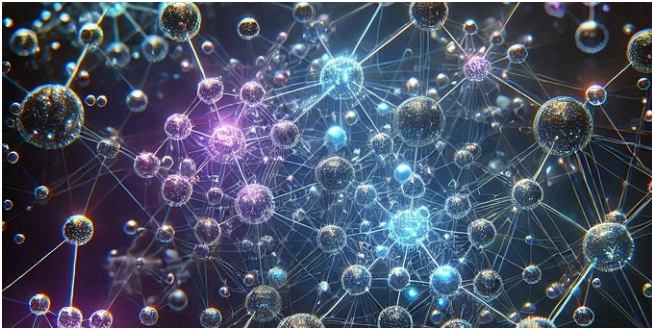
# How do Transformers work in NLP?

Overview

Aug 14, 2023  83



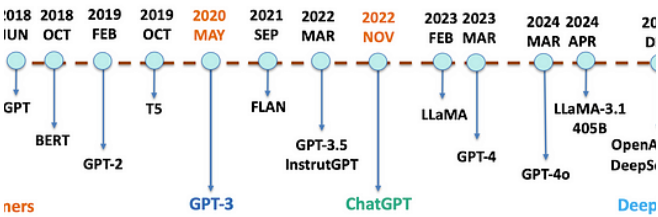
## Recommended from Medium



 Sybrandwildeboer

## How Large Language Models Predict the Next Word (and Why...

## A Brief History of LLMs



 LM Po

## A Brief History of LLMs

From Transformers (2017) to DeepSeek-R1 (2025)

Feb 10 1



...

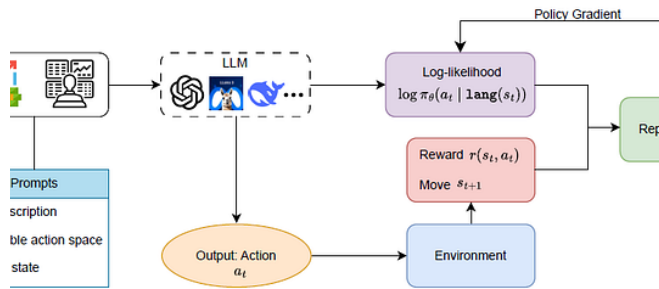


Feb 11 158

2



...



Antonio Velazquez Bustamante

## FLAG-TRADER: Fusing Large Language Models and...

Large language models (LLMs) fine-tuned on multimodal financial data have demonstrate...

Jun 11



...

Luke Jang

## Week 1: Building a GPT-Style LLM from Scratch

Dive into the beginning of building a LLM from scratch.

Apr 22 32



...

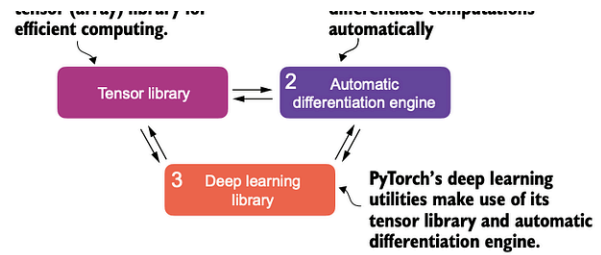


Figure A.1 PyTorch's three main components include a tensor library as a fundamental building block for computing, automatic differentiation for model optimization, and deep learning utility functions, making it easier to





Suvradeep

...