

Hackathon  
→ Custom Acceleration of Large Language Model Primitives

Studio Version 9.4.2

Release Date March 2025  
  
Author: Tadej Murovič, [tadej.murovic@codasip.com](mailto:tadej.murovic@codasip.com)

# **BACKGROUND**

## 1. What are LLMs (Large Language Models)?

Large Language Models (LLMs) are AI models (specifically neural networks) trained on massive amounts of text to learn the statistical patterns of language. They can perform tasks such as answering questions, translating languages, summarizing text, and more. LLMs are trained on huge text datasets (e.g., Wikipedia, books, websites where they capture grammar, semantics, facts, and patterns of usage, which allows them to generate coherent, human-like text or solve language-related tasks.

## 2. A Brief History Leading to Transformers

1. Recurrent Neural Networks (**RNN**s) and **LSTM**s (1990s–2010s):
   * RNNs are networks designed to handle sequential data by “remembering” past inputs through internal hidden states.
   * LSTMs (Long Short-Term Memory) and GRUs (Gated Recurrent Units) were inventions to fix problems with vanilla RNNs (like short-term memory loss—vanishing gradients or exploding gradients).
   * These worked quite well for tasks like language modeling, speech recognition, etc. But training on long sequences still posed challenges (memory, computational bottlenecks, slower training, difficulty capturing truly long-range dependencies).
2. The **Transformer** Era (2017 onward):
   * ***In 2017, Vaswani et al. published “***[***Attention is All You Need***](https://arxiv.org/abs/1706.03762)***,” introducing the Transformer architecture.***
   * This model used attention mechanisms to handle sequences. It dispensed with the recurrent step-by-step approach of RNNs, allowing for parallelization and scaling to large datasets more efficiently.
   * Transformers quickly became the state-of-the-art in many NLP tasks (translation, summarization, QA), leading to the development of massive LLMs such as BERT, GPT, and others.

## 3. High-Level Overview of a Transformer

A classic Transformer has two main components, originally built for machine translation:

1. **Encoder**: Processes an input sequence (e.g., an English sentence).
2. **Decoder**: Generates an output sequence (e.g., a French sentence).

When just language modeling is the goal (like GPT), we often only use a stack of decoders (or a slightly modified structure), focusing on predicting the next word.

### Inside each Transformer layer

1. **Self-Attention**: Computes how each word in a sequence attends to other words in that sequence.
2. **Feed-Forward Network**: A fully connected network applied to each position (word embedding) to transform information further.
3. **Add & Normalize**: Shortcut connections plus layer normalization to keep training stable.

## 4. The Mathematics Behind Transformers

### 4.1 Linear Algebra Primitives

Transformers rely heavily on:

* **Matrix Multiplications**: For transformations of vectors (word embeddings, intermediate representations).
* **Vector Dot Products**: To compute similarity between query-key pairs.
* **Softmax**: To convert raw scores into probabilities.

At a foundational level, everything is sequences of vectors and operations like:

*z=Wx+b*

where W is a matrix, x and z are vectors, and b is a bias vector.

### 4.2 Word Embeddings

What are embeddings?

* We convert words (or tokens) into numerical vectors, typically of dimension d.
* For example, the word “cat” might be transformed into a 300-dimensional vector v\_cat.
* These embeddings are learned so that words used in similar contexts end up with similar vectors.

Why is this useful?

* It turns words from discrete symbols (like “cat,” “dog,” “house”) into vectors that can be processed with linear algebra—an essential step to let neural networks handle textual data.

### 4.3 The Attention Mechanism

In short, the attention mechanism allows the model to dynamically focus on different parts of the input sequence when making predictions. The attention mechanism operates by computing attention weights for each word in the input sequence, which represent the degree of importance of that word with respect to the rest of the words in the text for the prediction task. The attention mechanism is based on three vectors: the query vector, the key vector, and the value vector. The query vector represents the information that the model is looking for, the key vector represents the information in the input sequence, and the value vector represents the information that the model should pay attention to. These concepts can be a little confusing, so let's take a look at an example to get a better understanding. Imagine you're a chef, and you're preparing a recipe that requires several ingredients. To make sure you have all the ingredients, you would have three things:

The **query**: This would be the request for a list of ingredients required for a recipe. It represents the information that you're looking for.

The **keys**: This would be all the ingredients you have in your kitchen. They represent the information that you have available.

The **values**: These would be the ingredients you actually need for the recipe. They represent the information that you should pay attention to.

Each word’s embedding is transformed into three vectors: query key , and value .

Here, is a matrix whose rows might be the embeddings of each word in a sentence, and are learnable parameter matrices.

Similarity (Dot Product) Between Query and Key:

* For a given query q\_i (from the i-th word), you take a dot product with each key k\_j (from every word j in the sentence).
* This dot product measures how much the i-th word should “focus on” or “attend to” the j-th word.

Softmax for Weights:

* Take all the dot products ​ (scaled by sqrt(d) to normalize) and convert them into weights via Softmax:
* ​ is the attention weight from word i to word j.

Weighted Sum of Values:

* Each word’s new representation is the weighted sum of the value vectors , weighted by ​:
* This means the i-th word’s updated representation is built from the “information” (values) of all words in the sentence, weighted by how relevant each word is (the attention weights).

### 4.4 Multi-Head Attention

One attention mechanism might not be enough to capture all nuances (e.g., one head might capture “subject-object relationships,” another might capture “adjectives modifying nouns,” etc.).

* Multi-Head means we run the above attention mechanism multiple times in parallel (with different sets of , then concatenate the results.
* This lets the model learn different types of relationships simultaneously.

### 4.5 Fully Connected (Feed-Forward) Layers

Each word-position’s new representation (the output from multi-head attention) is fed into a small feed-forward network (often 2 fully connected layers).

Self-attention mixes information across words, but we also need a non-linear transformation for each position’s representation. The fully connected layer (applied to each word embedding individually) can learn more complex mappings. Example analogy: You gather relevant info from your entire environment (via attention), then you do personal “thinking” with that info (the feed-forward step).

## 5. Building an LLM with Transformers

1. Stacking Transformer Blocks:
   * We repeat layers of (Multi-Head Self-Attention + Feed-Forward).
   * Each layer refines the representation of each token, capturing deeper linguistic and contextual patterns.
2. Training:
   * We typically optimize parameters to predict the next token (word) given the previous context. This is often called language modeling or causal language modeling.
   * Large datasets and powerful hardware allow models to learn a vast amount of linguistic (and factual) knowledge.
3. Inference:
   * After training, to generate text, the model starts with a prompt (initial text) and iteratively predicts the next token until it forms a complete response or meets a stopping criterion.

## 6. Real-Life Examples of Why These Components Are Useful

1. Why use a Fully Connected Layer?
   * If you only used attention, you would mix information from tokens but never transform it in a non-linear way at each position.
   * A feed-forward network can learn patterns like “if the context says the subject is plural, then transform the representation so that the next word might be a plural verb,” etc.
2. Why use Query, Key, Value in Attention?
   * Queries are the vantage point (the token looking for relevant info).
   * Keys are how we “index” or “label” each token’s content.
   * Values are the actual content that gets blended back into the representation.
   * Analogy: In a library (attention mechanism), your query is your search request, the keys are the titles/tags (indices) in the catalog, and the values are the actual books. You check the catalog to find relevant books (keys) and then pull the content from those books (values).
3. Why is Parallelization Important?
   * RNNs read text word by word, which is slow for very long sequences. Transformers let you process entire sentences (or large chunks) at once, making training much faster and more efficient on GPUs.
4. Why Softmax Over Dot Products?
   * We want weights that sum up to 1, indicating how much attention to give each word. Softmax is a smooth way of converting arbitrary numbers into a probability-like distribution.

## 7. Putting It All Together

* Input: You have a sequence of tokens (words or subwords).
* Embedding: Turn each token into a numerical vector.
* Positional Encoding: Add some positional information (since Transformers have no intrinsic notion of “order” the way RNNs do).
* Attention: Let each token gather relevant information from the entire sequence.
* Feed-Forward: Transform each token’s representation.
* Stack: Repeat the above steps for many layers to capture ever more complex patterns.
* Output: Either classify the sequence, generate a translation, or predict the next word for language modeling.

The result: A large language model that can handle grammar, semantics, long-range context, and generate coherent, context-aware text.