**How Transformer LLMs Work**

Introduction

The Transformer Architecture was first introduced in the 2017 paper, Attention is All You Need by Ashish Vaswani et al for machine translation tasks.

The idea it to input an English sentence and have the network output a German sentence.

This same architecture tend to be great at inputting a prompt and outputting a response to that prompt, like a question and the answer to that question.

The original transformer consists of an encoder and a decoder. The encoder preprocesses the entire input English text to extract the context needed to perform the translation. Then the decoder uses the encoder context to generate the German. The encoder + decoder form the basis for the models used in many language models today.

The Encoder model provide rich, context sensitive representations of the input text, and is the basis for the BERT model and most of the embedding models using RAG applications.

The Decoder model performs text generation tasks such as summarizing text, writing code, answering questions, and is the basis for most popular LLMs, such as those from OpenAI, Anthropic.

A diagram of a computer model

AI-generated content may be incorrect.

Course outline:

* Delve into recent developments in LLMs
* Learn about tokenization
* Gain intuition about how transformers work, focusing on the decoder only models.
  + A generative takes in the text prompt, and it generates a text in response by generating one token at a time.
  + How generation process works: The model starts by mapping each input token into an embedding vector that captures the meaning of that token.
  + After that, the model parses these token embeddings through a stack of transformer blocks, where each block is a specific neural network architecture that those designed to learn flexibly from data and also scale well on GPUs.
  + Each transformer block is made up of an attention layer and a feed-forward network. The model then uses the output vectors of the transformer blocks and passes them to the last component, the language modelling head.
  + The language modelling head generates the output token

A diagram of a transformer

AI-generated content may be incorrect.

**Lesson 1: Understanding Language Models: Language as a Bag-of-Words**

Learn about the evolution of how language has been represented numerically.

**Bag-of-words:** An algorithm that represents words as large sparse vectors or arrays of numbers, which simply record the presence of words.

**Word2Vec:** Word representation capture the meaning of words in the context of a few neighbouring words.

**Transformers:** Dense vectors captured the meaning of words in the context of a sentence or a paragraph.

A diagram of a language

AI-generated content may be incorrect.

Language is a tricky concept for computers. Text is unstructured in nature and loses its meaning when represented by zeros and ones or individual characters. As a result, throughout the history of language AI, there has been a large focus on representing language in a structured manner so that it can be more easily used by computers.

From generating text, to creating numerical representations, and classifying textual inputs. These are just a few of the numerous tasks you can do with language AI.

At the start of this language AI field, the focus was mainly on representing language to analyse unstructured data. A first and still very relevant method is by representing language as a bag-of-words.

A diagram of a language

AI-generated content may be incorrect.

**Tokenization:** The process of converting the input text into pieces. Each individual word is called a token. A token can be any even smaller than an entire word.

You can perform the same tokenization process with another document. Now with 2 sets of tokens, you can create something called a vocabulary. Vocabulary contains all unique words found in both input documents. As such, vocabulary would be lesser than the amount of tokens generated, which is referred to as the vocabulary size.

A diagram of a cat

AI-generated content may be incorrect.

“My cat is cute” -> This input has 4 tokens that match with some of the words in the vocabulary. You can then count how many times of a certain token appears in the vocabulary that we already created.

A screenshot of a computer

AI-generated content may be incorrect.

However, you would also need to take note of the vocabulary that do not appear in the input. A sentence not only gives meaning to the words it contains, but also the words it doesn’t.

Bag-of-words: A numerical representation that indicates the count of individual words appearing in the vocabulary. It does nothing more other than this.

The order is important as it allows us to compare different sentences to one another.

In practice, we call this a vector representation -> A list of numerical values that represents the input.

**Lesson 2: Understanding Language Models: (Word) Embeddings**

Bag-of-words has a flaw: It does not consider the semantic nature of text. It considers language to be nothing more than an almost literal bag-of-words, and ignores the semantic nature or meaning of text.

Word2Vec is one of the first successful attempts at capturing the meaning of text in vector embeddings through neural networks.

* To do so, Word2Vec learned semantic representations of words by training on vast amounts of textual data (like the entirety of Wikipedia).
* To generate these semantic representations, Word2Vec leverages neural networks.
* Neural networks: Networks consist of interconnected layers of nodes that process information. Neural networks can have many layers, which each connection has a certain weight depending on the inputs. The weights are often referred to as parameters of the model.

A diagram of a machine

AI-generated content may be incorrect.

* Using these neural networks, Word2Vec generates word embeddings by looking at which other words they tend to appear next to in a given sentence. You start by assigning every word in your vocabulary with a vector embedding.
* Example, 5 values for each word is initialized with random values. Then in every training step, you take pairs of words from training data, and the model attempts to predict whether or not they are likely to be neighbours in a sentence.
* During this training process, Word2Vec learns the relationship between words and distils that information into the embedding.
* If the 2 words tend to have the same neighbour, their embeddings will be closer to one another and vice versa.

A close-up of a sign

AI-generated content may be incorrect.

The resulting embeddings capture the meaning of words.

Example

Assume that you have an embedding for the word “cats”. This embedding has generates values between -1 and 1.

Embeddings attempt to capture meaning by representing the properties of words. For instance, the word “cats” might score low on the properties newborn and human and fruits, while scoring high on the property’s animal and plural.

The number of properties or values an embedding has is called the number of dimensions, and is generally a fixed size.

By doing these for a number of words, you can use these values to get a proxy of the meaning of these words.

The number of dimensions could be quite large, where it is not uncommon to see embeddings with more than a thousand values. However, in reality, you do not actually know what these properties exactly represent as they are learned from complex mathematical calculations.

These properties do allow you to compare embeddings and therefore words with one another. Words with similar meaning would be grouped together, whereas different words are further apart. How similar or dissimilar certain words are, depends on the training data.

Types of Embeddings

There are many types of embedding we can use.

1. Representation Model

When we talk about a model like Word2Vec that converts textual input to embeddings, we refer to it as a representation model as it attempts to represent text as values.

*Token Embedding:* Through tokenization, you can split the sentence up into tokens. Note that this procedure is actually not splitting the input by white spaces. The reason for this is that the models that perform tokenization, also called tokenizers, have a fixed vocabulary. As such, they cannot represent all words that exist, but sometimes have to find combination of words.

You give the representation model these individual tokens, which in turn generates these embeddings, one for each token.

*Word Embeddings*: When you average the embeddings of these tokens, you get a word embedding as it now represents the entire word (vocal + ization).

Similar techniques can be used for entire sentences to create *sentence embeddings*, and the same for longer texts such as documents to create a *document embedding*.

**Lesson 3: Understanding Language Models: Encoding and Decoding Context with Attention**

Word2Vec creates static embeddings. The same embedding is generated for the word “bank”, regardless of the context. “Bank” can both refer to the river bank, and also the financial bank. Its meaning, and therefore its embedding should change depending on the context. Capturing the text context is important to perform some language tasks, such as translation.

Recurrent Neural Networks (RNNs)

A step in encoding this text was achieved through RNNs. These are variants of neural networks that can model sequences as an additional input.

To do so, these RNNs are used for 2 tasks, encoding or representing an input sentence, and decoding or generating an output sentence.

The text is passed through the encoder, which attempts to represent the entire sequence through embeddings. The decoder then uses those embeddings to generate language.

A diagram of a language

Description automatically generated

Each step in this architecture is autoregressive. When generating the next words, this architecture needs to consume all previously generated words.

A diagram of steps and steps

Description automatically generated with medium confidence

Let’s explore this concept of encoding and decoding in a bit more detail.

You again start with the input sentence “I love Llamas” tokenizes into tokens. We can use Word2Vec to create embeddings as the inputs. Although these embeddings are static by itself, the encoder processes the entire sequence in one go and takes into account the context of the embeddings.

The encoding aims to represent the input as well as possible, and generates the context in the form of an embedding. This decoder in turn is in charge of generating language, and does so by leveraging the previously generated context embedding to eventually generate the outputs.

As we explored previously, these output tokens are generated one at a time, which is called autoregressive. This context embedding, however, makes it difficult to deal with longer sentences, since it is merely a single embedding representing the entire input. So the single embedding might fail to capture the entire context of a long and complex sequence.

Attention

In 2014, a solution called attention was introduced that highly improved upon the original architecture. Attention allows the model to focus on parts of the input sequence that are relevant to one another, or attend to each other and amplify their signal.

Attention selectively determines which words are most important in a given sentence.

For example, words with similar meanings (I and Ik in Dutch) have higher attention weights since they are more related. (I and llama) has lower attention weights since they do not relate much to each other in this particular sentence.

A blue and white squares with white text

Description automatically generated

By adding these attention mechanisms to the decoder step, the RNN can generate signals for each input word in the sequence related to the potential outputs.

You can again represent the input using Word2Vec embeddings and pass those to the encoder. Instead of passing only a context embedding to the decoder, the hidden states of all input words are passed to the decoder. A stateful word is an internal vector from a hidden layer of an RNN that contains the information about the previous words. The decoder then uses the attention mechanism to look at the entire sequence. Finally this again generates the language. Due to this attention mechanism, the output tends to be much better since now you look at the entire sequence using embeddings for each token or words instead of the smaller and more limited context embedding. So during generation, the model attends to the most relevant inputs.

The sequential nature of this architecture precludes parallelization during training of the model.

A diagram of a computer language

Description automatically generated

**Lesson 4: Understanding Language Models: Transformers**

In this lesson, you will explore how the technique of attention was further developed and to this day still powers many LLMs.

The true power of attention and what drives the amazing abilities of most LLMs was first explored in the Attention is All You Need paper. This paper introduces the Transformer’s architecture, which is based solely on attention without the RNN. This architecture allows the model to be trained in parallel, which speeds up calculation significantly compared to the RNN based model which precludes parallelization.

**How Transformer Works**

The transformer consists of stacked encoder and decoder blocks. These blocks all have the same attention mechanism that you saw previously, and by stacking these blocks, you amplify the strength of encoder and decoders.

Encoder:

In the encoder, the input (I love llamas) is converted to embeddings, but instead of Word2Vec embeddings, we start with random values.

Then self-attention, which is attention focused on only the input, processes these embeddings and updates them. These updated embeddings contain more contextualized information as a result of the intention mechanism.

They are passed to a feed-forward neural network, which is a similar network that we explored before.

Then, they finally created contextualised token word embeddings. Remember that the encoder is meant for representing text and does a good job of generating embeddings.

A diagram of a computer process

Description automatically generated

Self-attention is an attention mechanism, and instead of processing 2 separate sequences, it processes only 1 sequence (The input by comparing to itself).

Decoder

After the encoder done processing the information, the next step is for decoder. The decoder can take any previously generated words and passed it to the masked self-attention, similar to the encoder to process these embeddings.

Intermediate embeddings are generated and passed to another attention network together with the embeddings of the encoder. Thus processing both what has been generated and what you already have.

This output is passed to a neural network, and finally generates the next word in the sequence.

A diagram of a transformer decoder

Description automatically generated

Masked self-attention is similar to self-attention, but removes all values from the upper diagonal. Therefore, it masks future positions so that any given token can only attend to tokens that came before it. That helps leaking information when generating the output.

Original Transformer Model

The original transformer model is an encoder-decoder architecture that serves translation tasks well, but cannot be used for other tasks like text classification.

**BERT**In 2018, a new architecture called Bidirectional Encoder Representations from Transformers (BERT) was introduced, that could be leveraged for various tasks.

A diagram of a computer process

Description automatically generated

BERT is an encoder only architecture that focusing on representing language and generating contextual word embeddings. These encoder blocks are the same as we saw before (self-attention followed by neural networks). The input contains an additional token, the classification token (CLS).

CLS is used as a representation for the entire input. Often we use this CLS token as the input embedding for fine tuning the model on specific tasks like classification.

To train a BERT-like model, you can use a technique called masked language modelling. You first randomly mask a number of words from your input sentence, and have the model predict these masked words.

A diagram of a transformer

Description automatically generated

By doing so, the model learns the represent language as it attempts to deconstruct these masked words.

Training is typically a 2-step approach. First, you apply masked language modelling on large amounts of data, and this is called pre-training. After which, you can fine-tune your pre-trained model on a number of downstream tasks, including classification.

A diagram of a company

Description automatically generated

**Generative Models**

Generative models in contrast use a different architecture. Assume that again you have another input sequence and randomly initialized embeddings. The input is then passed to the decoder only, as generative models tend to only stack decoders.

One of the first implementations is called Generative Pre-Trained Transformer (GPT) or GPT-1. It uses the deep transformer decoder.

A decoder block uses again masked self-attention, which is then passed to a feed-forward neural network. Note that it does not use any encoders as we explored previously. And finally, the next word is generated.

A diagram of a decodeder

Description automatically generated

These are the 2 flavors that you will see most often: generative models like ChatGPT, and representation models, like embedding models (BERT).

**Context Length**

These models have something in common, called the context length.

You start from an input sequence (Tell me something about llamas.) Now let’s say you already generated some tokens previously (Llamas are).

The original query, together with the previously generated tokens, represent the current context length (The amount of tokens that are currently being processed).

In contrast, a generative LLM like GPT-1 or even a representation model can have a maximum context length, for example, 512. That means that the model can only process 512 tokens at a given time.

A diagram of a diagram

Description automatically generated

Note that this also include the tokens that are being generated as they update the current context length.

These generative models do the LLMs justice. GPT-1 already had more than 100 million parameters. The next version, GPT-2, with over 1 billion parameters, and GPT-3 with already 175 billion parameters. As the number of parameters grew, so did their capabilities. That is why you will often see such large models.

A diagram of a circle with numbers and a point

Description automatically generated

Looking at the year that we called the year of Generative AI, it all started with the well-known ChatGPT model (GPT 3.5). Following the success of ChatGPT, many other proprietary models soon followed. Fortunately, open-source models followed quickly. These are models that have their weights publicly available for us to use. Some of them can even be freely used for commercial purposes.

A graph showing the number of names

Description automatically generated with medium confidence

**Lesson 5: Tokenizers**

In this lesson, we will illustrate what tokens are and how they help transformers do their jobs.

Imagine you have a given input sentence like “Have the bards who”. For language model to process that input text, it will first break down the text into smaller pieces. Each pieces is called a token.

Tokenization: This process of breaking down the text.

Each token is then turned into a numerical representation, also called embeddings. These are vector values that represent the semantic nature of a given text. These embeddings are static, and each embedding is created independent from all other embeddings and tokens. These embeddings are processed by the Large Language Model, and converted into contextualized embeddings. These contextualized embeddings are still one for each input token, but has been processed such that all other tokens are considered. These embeddings can be the output of a model, but also used by the model to then create outputs. In the case of generative models, this can be another token.

A diagram of a diagram

Description automatically generated

**Tokenization Process**

Given input sentence like “Have the bards who”. It is tokenized or encoded into smaller pieces. Tokens can be entire words or pieces of a word.



When these pieces are combined, they form the original words. This process is necessary as tokenizers have a limited number of tokens or vocabulary, so whenever it encounters an unknown word, it can still represented by these sub tokens.

Each token has an associated fixed ID to easily encode and decode the tokens. These are fed to the language model that internally creates the token embeddings. The output of a generative model would then be another token ID, which is decoded to represent an actual token or word.

A diagram of a computer

Description automatically generated

There are many different tokenization levels that we can explore.

***Word tokens:*** When you represent the input as word tokens, the entire sequence is represented by words.

***Sub word tokens:*** Can either be the entire word itself, or sub pieces of the original word. Eg: If the tokenizer does not have a token for the word “bards”, it can still represent it with its tokens for b and ards.

***Character tokens:*** You can split sub word tokens into character tokens, one for each character in the original word, representing the entire input as nothing more than each individual character.

* The smallest representation is that of bytes, which is used to encode a single character of text in a computer. This is done for every single character.
* Note how the symbol needs additional bytes for representation, as that symbol is more complex than just a single-character representation.

In practice, most LLMs have tokenizers that work on a sub word token level. Its vocabulary is flexible and allows to represent most words by either its entire representation or using subtokens.

A screenshot of a computer code

Description automatically generated

**Lesson 6: Architectural Overview**

When we are thinking about transformer LLMs, we know that there’s an input prompt to the model. And then there is a generation where an output text that the model generates.

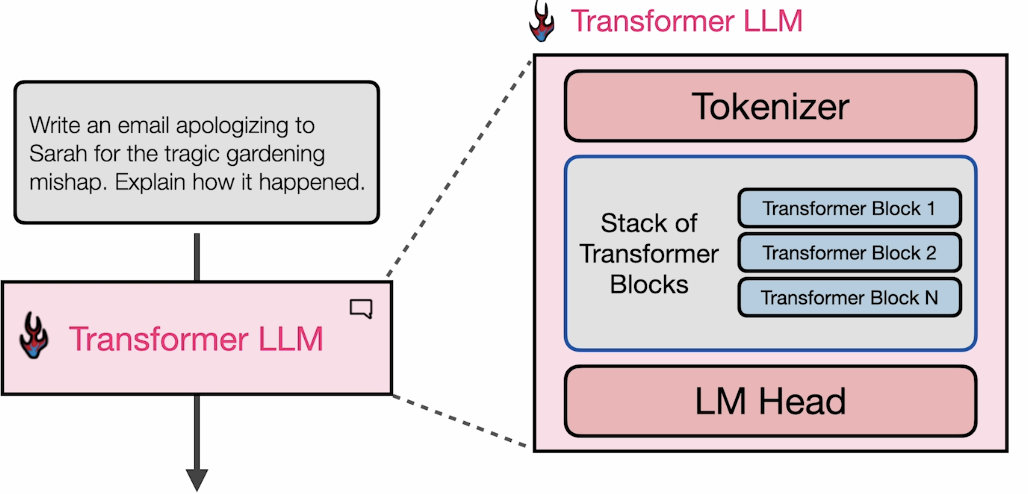
One of the important intuitions to get the force of understanding on how the transformer works is to understand that it generates tokens one by one.

We will break down the generation step so that you can understand the underlying mechanism on how it works.

Transformer is made up of 3 components. One of them is tokenizer which is look into in previous lesson.

And then the output of that tokenizer goes to a stack of transformer blocks. This is where the vast majority of computing is. This is the neural networks, that operates on this and do the magic here.

The output of this stack of transformer blocks goes into a neural network called the Language Modelling Head.



**Language Modelling Head**

At the very end of the processing of a language model, you have all your tokens that you started with that are defined in the tokenizer.

What happens at the end is a kind of scoring or a token probability calculation based on all the processing that the model in the stack of transformers have done, to make sense of the input context and what is requested in the prompt, and what the next token should be in response to that.

And so the result of the language modelling head is this sort of token probability scoring that says, if all the tokens that I know, this is how much percent of probability each of these tokens has. All of these must add up to 100%. The highest probability token may become possibly the output, not necessarily the output.

A screenshot of a computer

Description automatically generated

But you can choose the highest probability token (one method of choosing the next output token). These are called **decoding strategies**.

If you are choosing the top-scoring token, it is a good strategy for a lot of cases. (Temperature = 0, greedy decoding)

Another method is to choose top\_p, that incorporate multiple tokens. So, it might generate the most probable tokens, but it also in some cases with lower probability, it might pick the next highest probability token. It definitely looks at the scoring. It doesn’t always have to pick the top one.

These are sometimes important to generate text that sounds neutral. And it’s sometimes why you generate multiple times using the same prompt, you will get 2 different answers.

**Parallelization**

Another important idea or intuition about transformers which makes transformers perform a lot better than previous methods like RNNs, is that they process all of their input tokens in parallel, and that parallelization makes it time efficient so that we can compute a long context on a lot of different GPUs in similar time. The way to envision this is to think of multiple tracks flowing through this stack of transformer blocks and the number of tracks here is the **context size** of the model.

The generated token in decoder LLM transformers, is the output of the final token in the model.

Generating every other token after the start with the first step that processes the input is a little bit different. Once we generate our first token, we feed that entire prompt with the token that we have generated into the transformer again. **It is a loop**.

And then where you process the inputs one by one as you generate these tokens, the difference is that you can cache these calculations because they are going to be exactly the same. You can cache them to speed up the generation of model. This is called **KV Caching. K means keys, V is values**.

A screenshot of a computer program

Description automatically generated

A diagram of a calculator

Description automatically generated