**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**