De-coded: Transformers explained in plain English

No code, maths, or mention of Keys, Queries and Values

Chris Hughes · Follow

Published in Towards Data Science · 15 min read · 4 days ago

Since their introduction in 2017, transformers have emerged as a prominent force in the field of Machine Learning, revolutionizing the capabilities of major translation and autocomplete services.

Recently, the popularity of transformers has soared even higher with the advent of large language models like OpenAI's ChatGPT, GPT-4, and Meta's LLama. These models, which have garnered immense attention and excitement, are all built on the foundation of the transformer architecture. By leveraging the power of transformers, these models have achieved remarkable breakthroughs in natural language understanding and generation; exposing these to the general public.

Despite a lot of good resources which break down how transformers work, I found myself in a position where I understood the how the mechanics worked mathematically but found it difficult to explain how a transformer

works intuitively. After conducting many interviews, speaking to my colleagues, and giving a lightning talk on the subject, it seems that many people share this problem!

In this blog post, I shall aim to provide a high-level explanation of how transformers work without relying on code or mathematics. My goal is to avoid confusing technical jargon and comparisons with previous architectures. Whilst I'll try to keep things as simple as possible, this won't be easy as transformers are quite complex, but I hope it will provide a better intuition of what they do and how they do it.

What is a Transformer?

A transformer is a type of neural network architecture which is well suited for tasks that involve processing sequences as inputs. Perhaps the most common example of a sequence in this context is a sentence, which we can think of as an ordered set of words.

The aim of these models is to create a numerical representation for each element within a sequence; encapsulating essential information about the element and its neighbouring context. The resulting numerical representations can then be passed on to downstream networks, which can leverage this information to perform various tasks, including generation and classification.

By creating such rich representations, these models enable downstream networks to better understand the underlying patterns and relationships within the input sequence, which enhances their ability to generate coherent and contextually relevant outputs.

The key advantage of transformers lies in their ability to handle long-range dependencies within sequences, as well as being highly efficient; capable of processing sequences in parallel. This is particularly useful for tasks such as machine translation, sentiment analysis, and text generation.

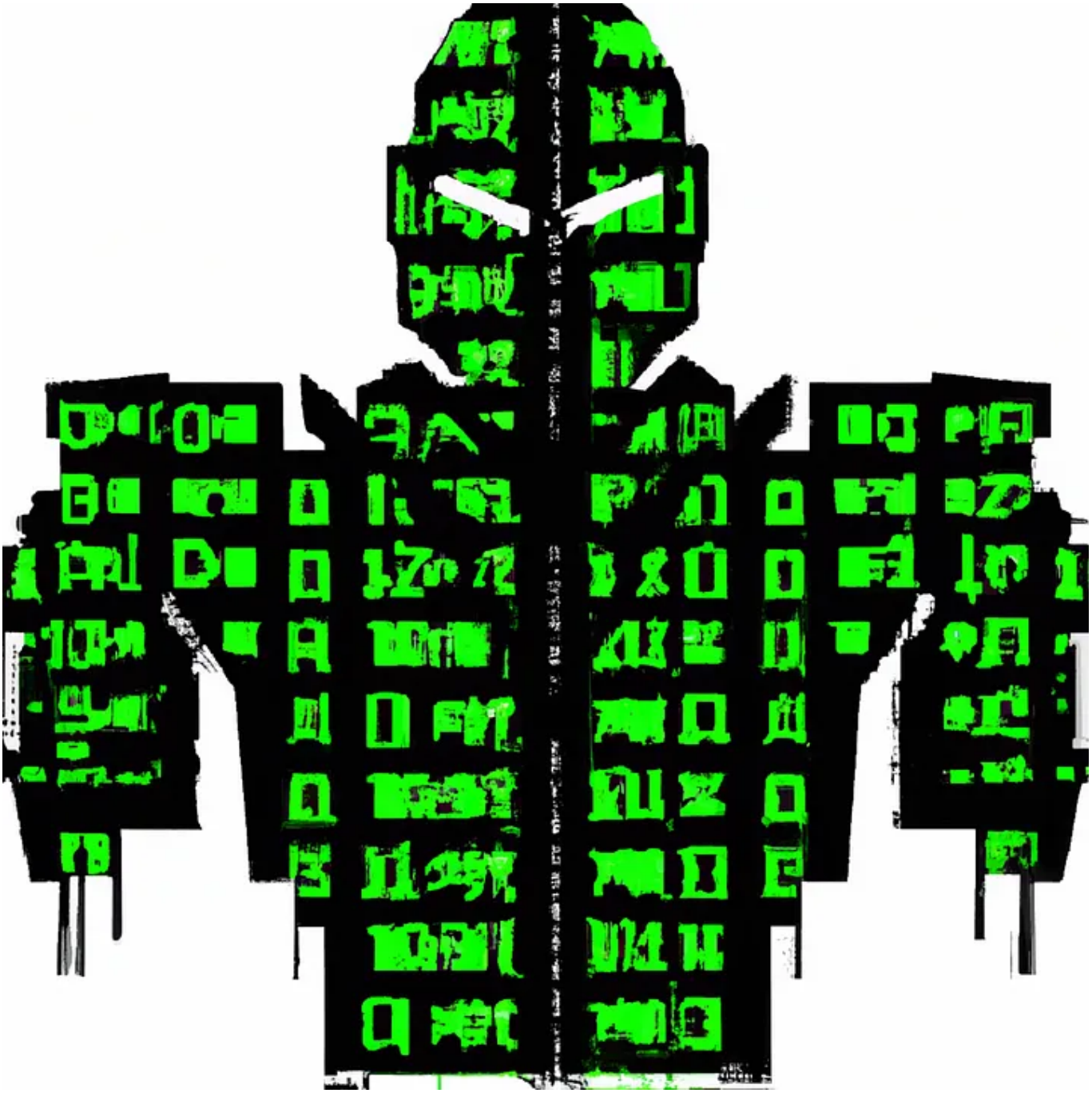


Image generated by the Azure OpenAI Service DALL-E model with the following prompt: “The green and black Matrix code in the shape of Optimus Prime”

What goes into the Transformer?

To feed an input into a transformer, we must first convert it into a sequence of tokens; a set of integers that represent our input.

As transformers were first applied in the NLP domain, let's consider this scenario first. The simplest way to convert a sentence into a series of tokens is to define a *vocabulary* which acts as a lookup table, mapping words to integers; we can reserve a specific number to represent any word which is not contained in this vocabulary, so that we can always assign an integer value.

```
Raw text: A black cat
Vocab: {"A": 0, "black": 1, "cat": 2, "cats": 3}
Tokenized text: [0, 1, 2]
```

In practice, this is a naïve way of encoding text, as words such as *cat* and *cats* are treated as completely different tokens, despite them being singular and plural descriptions of the same animal! To overcome this, different tokenisation strategies — such as byte-pair encoding — have been devised which break words up into smaller chunks before indexing them. Additionally, it is often useful to add special tokens to represent characteristics such as the start and end of a sentence, to provide additional context to the model.

Let's consider the following example, to better understand the tokenization process.

"Hello there, isn't the weather nice today in Drosval?"

Drosval is a name generated by GPT-4 using the following prompt: *"Can you create a fictional place name that sounds like it could belong to David Gemmell's Drenai universe?"*; chosen deliberately as it shouldn't appear in the vocabulary of any trained model.

Using the `bert-base-uncased` tokenizer from the transformers library, this is converted to the following sequence of tokens:

```
[101, 7592, 2045, 1010, 3475, 1521, 1056, 1996, 4633, 3835, 2651, 1999, 2852, 2891, 10175, 1029, 102]
```

The integers that represent each word will change depending on the specific model training and tokenization strategy. Decoding this, we can see the word that each token represents:

```
[CLS] hello there , isn ' t the weather nice today in dr ##os ##val ? [SEP]
```

Interestingly, we can see that this is not the same as our input. Special tokens have been added, our abbreviation has been split into multiple tokens, and our fictional place name is represented by different 'chunks'. As we used the 'uncased' model, we have also lost all capitalization context.

However, whilst we used a sentence for our example, transformers are not limited to text inputs; this architecture has also demonstrated good results on vision tasks. To convert an image into a sequence, the authors of ViT sliced the image into non-overlapping 16x16 pixel patches and concatenated these into a long vector before passing it into the model. If we were using a transformer in a Recommender system, one approach could be to use the item ids of the last *n* items browsed by a user as an input to our network. If

we can create a meaningful representation of input tokens for our domain, we can feed this into a transformer network.

Embedding our tokens

Once we have a sequence of integers which represents our input, we can convert them into embeddings. Embeddings are a way of representing information that can be easily processed by machine learning algorithms; they aim to capture the meaning of the token being encoded in a compressed format, by representing the information as a sequence of numbers. Initially, embeddings are initialised as sequences of random numbers, and meaningful representations are learned during training. However, these embeddings have an inherent limitation: they do not take into account the context in which the token appears. There are two aspects to this.

Depending on the task, when we embed our tokens, we may also wish to preserve the ordering of our tokens; this is especially important in domains such as NLP, or we essentially end up with a *bag of words* approach. To overcome this, we apply *positional encoding* to our embeddings. Whilst there are multiple ways of creating positional embeddings, the main idea is that we have *another* set of embeddings which represent the position of each token in the input sequence, which are combined with our token embeddings.

The other issue is that tokens can have different meanings depending on the tokens that surround it. Consider the following sentences:

It's dark, who turned off the light?

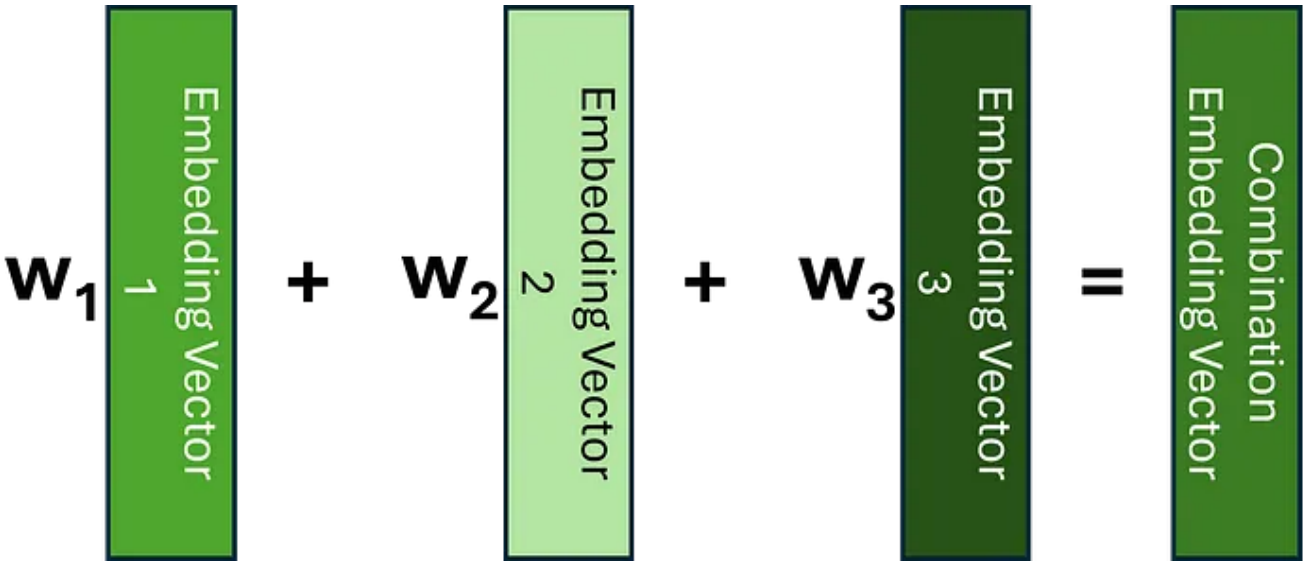
Wow, this parcel is really light!

Here, the word *light* is used in two different contexts, where it has completely different meanings! However, it is likely that — depending on the tokenisation strategy — the embedding will be the same. In a transformer, this is handled by its *attention* mechanism.

Conceptually, what is attention?

Perhaps the most important mechanism used by the transformer architecture is known as *attention*, which enables the network to understand which parts of the input sequence are the most relevant for the given task. For each token in the sequence, the attention mechanism identifies which other tokens are important for understanding the current token in the given context. Before we explore how this is implemented within a transformer, let's start simple and try to understand what the attention mechanism is trying to achieve conceptually, to build our intuition.

One way to understand attention is to think of it as a method which replaces each token embedding with an embedding that includes information about its neighbouring tokens; instead of using the same embedding for every token regardless of its context. If we knew which tokens were relevant to the current token, one way of capturing this context would be to create a weighted average — or, more generally, a linear combination — of these embeddings.



Let's consider a simple example of how this could look for one of the sentences we saw earlier. Before attention is applied, the embeddings in the sequence have no context of their neighbours. Therefore, we can visualise the embedding for the word *light* as the following linear combination.

Token Embedding	It's	Dark	Who	Turned	Off	The	Light
Weight	0	0	0	0	0	0	1

Here, we can see that our weights are just the identity matrix. After applying our attention mechanism, we would like to learn a weight matrix such that we could express our *light* embedding in a way similar to the following.

Token Embedding	It's	Dark	Who	Turned	Off	The	Light
Weight	0	0.3	0	0.1	0.1	0	0.5

This time, larger weights are given to the embeddings that correspond to the most relevant parts of the sequence for our chosen token; which should

ensure that the most important context is captured in the new embedding vector.

Embeddings which contain information about their current context are sometimes known as *contextualised embeddings,* and this is ultimately what we are trying to create.

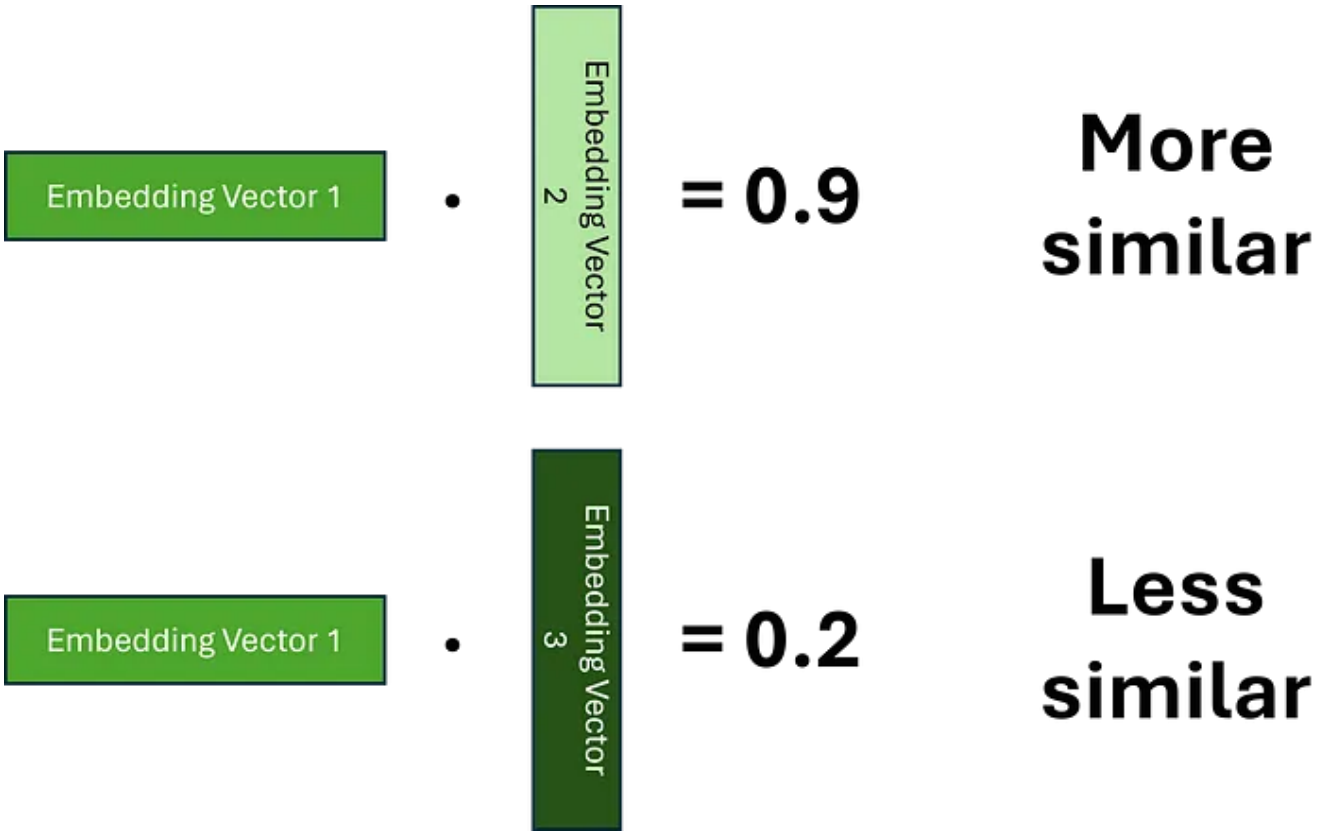
Now that we have a high level understanding of what attention is trying to achieve, let's explore how this is actually implemented in the following section.

How is attention calculated?

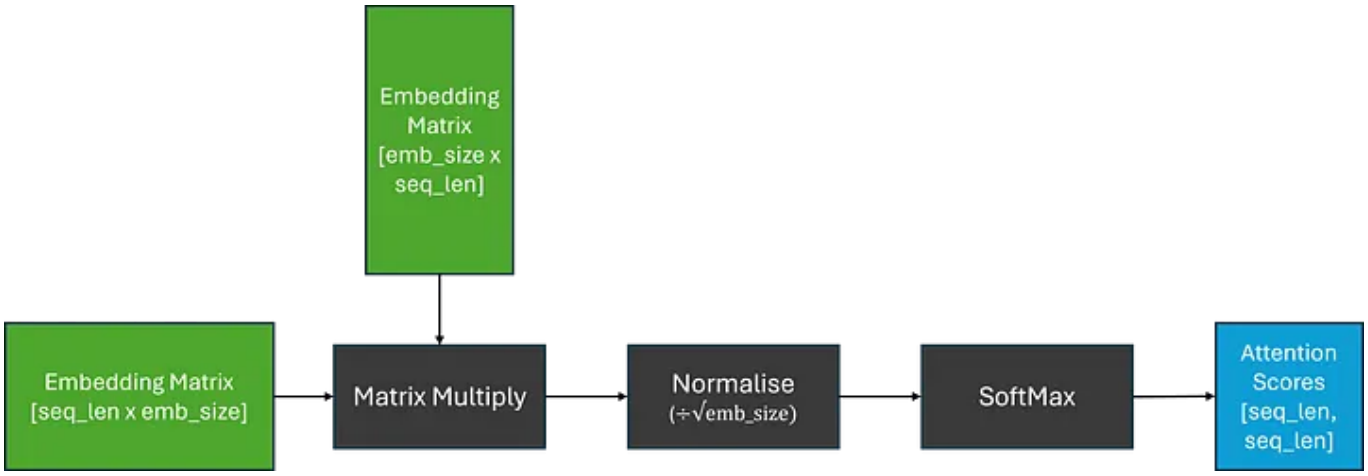
There are multiple types of attention, and the main differences lie in the way that the weights used to perform the linear combination are calculated. Here, we shall consider scaled dot-product attention, as introduced in the original paper, as this is the most common approach. In this section, assume that all of our embeddings have been positionally encoded.

Recalling that our aim is to create contextualised embeddings using linear combinations of our original embeddings, let's start simple and assume that we can encode all of the necessary information needed into our learned embedding vectors, and all we need to calculate are the weights.

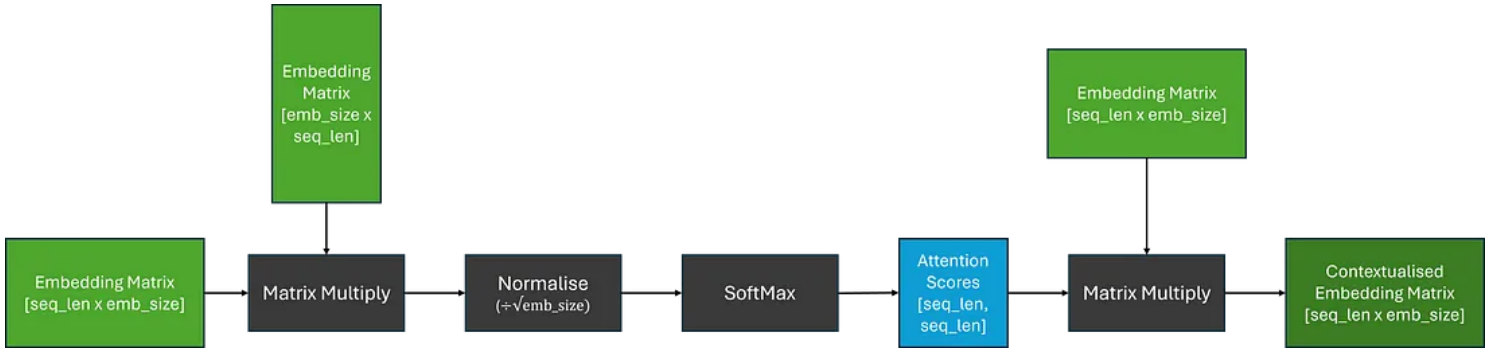
To calculate the weights, we must first determine which tokens are relevant to each other. To achieve this, we need to establish a notion of similarity between two embeddings. One way to represent this similarity is by using the dot product, where we would like to learn embeddings such that higher scores indicate that two words are more similar.



As, for each token, we need to calculate its relevance with every other token in the sequence, we can generalise this to a matrix multiplication, which provides us with our weight matrix; which are often referred to as *attention scores*. To ensure that our weights sum to one, we also apply the SoftMax function. However, as matrix multiplications can produce arbitrarily large numbers, this could result in the SoftMax function returning very small gradients for large attention scores; which may lead to the vanishing gradient problem during training. To counteract this, the attention scores are multiplied by a scaling factor, before applying the SoftMax.



Now, to get our contextualised embedding matrix, we can multiply our attention scores with our original embedding matrix; which is the equivalent of taking linear combinations of our embeddings.

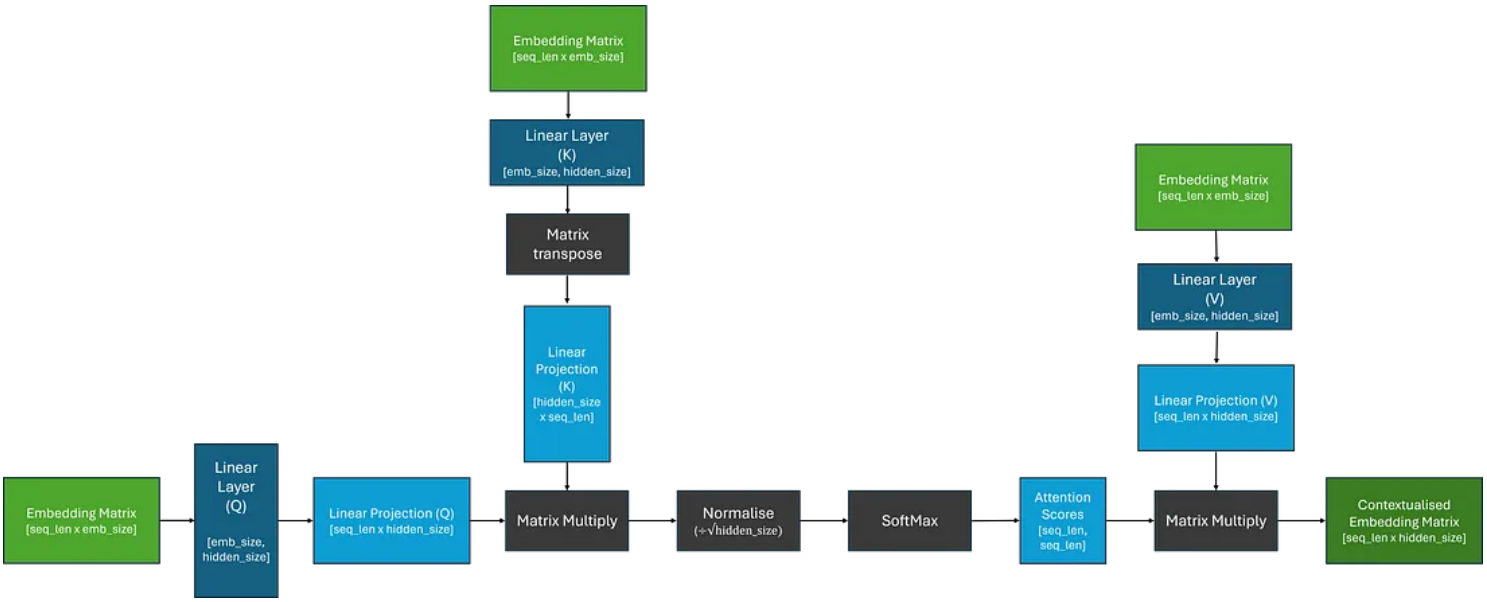


Simplified attention calculation: assuming embeddings are positionally encoded

Whilst it may be possible for a model to learn embeddings which are complex enough to generate attention scores and subsequent contextualised embeddings; we are trying to condense a lot of information into the embedding dimension, which is usually quite small.

Therefore, to make the task slightly easier for the model to learn, let's introduce some more learnable parameters! Instead of using our embedding matrix directly, lets pass this through three, independent linear layers

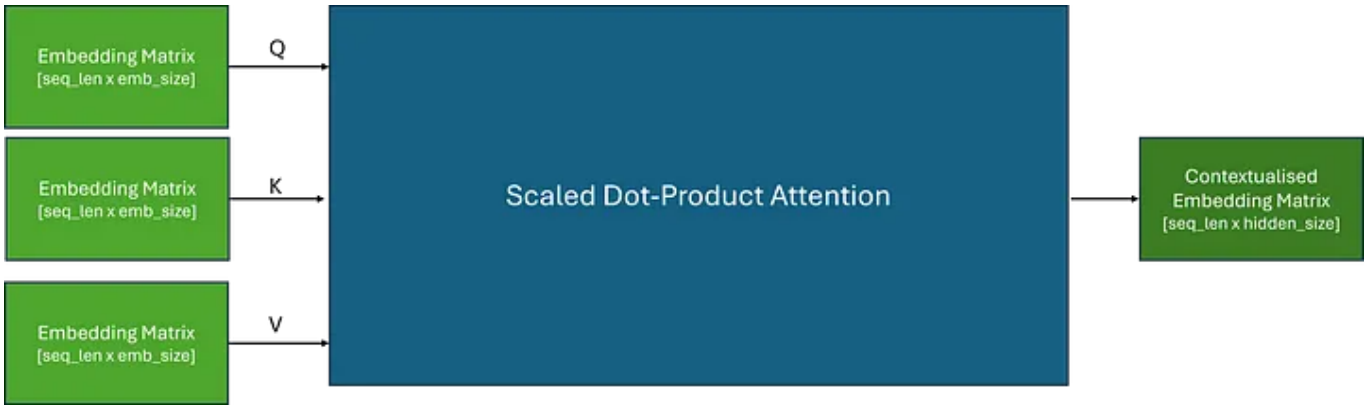
(matrix multiplications); this should enable the model to 'focus' on different parts of the embeddings. This is depicted in the image below:



Scaled dot-product Self Attention: assuming embeddings are positionally encoded

From the image, we can see that the linear projections are labelled Q, K and V. In the original paper, these projections were named *Query, Key and Value*, supposedly taking inspiration from information retrieval. Personally, I never found that this analogy helped my understanding, so I tend not to focus on this; I have followed the terminology here for consistency with the literature, and to make it explicit that these linear layers are distinct.

Now that we understand how this process works, we can think of the attention calculation as a single block with three inputs, which will be passed to Q, K and V.



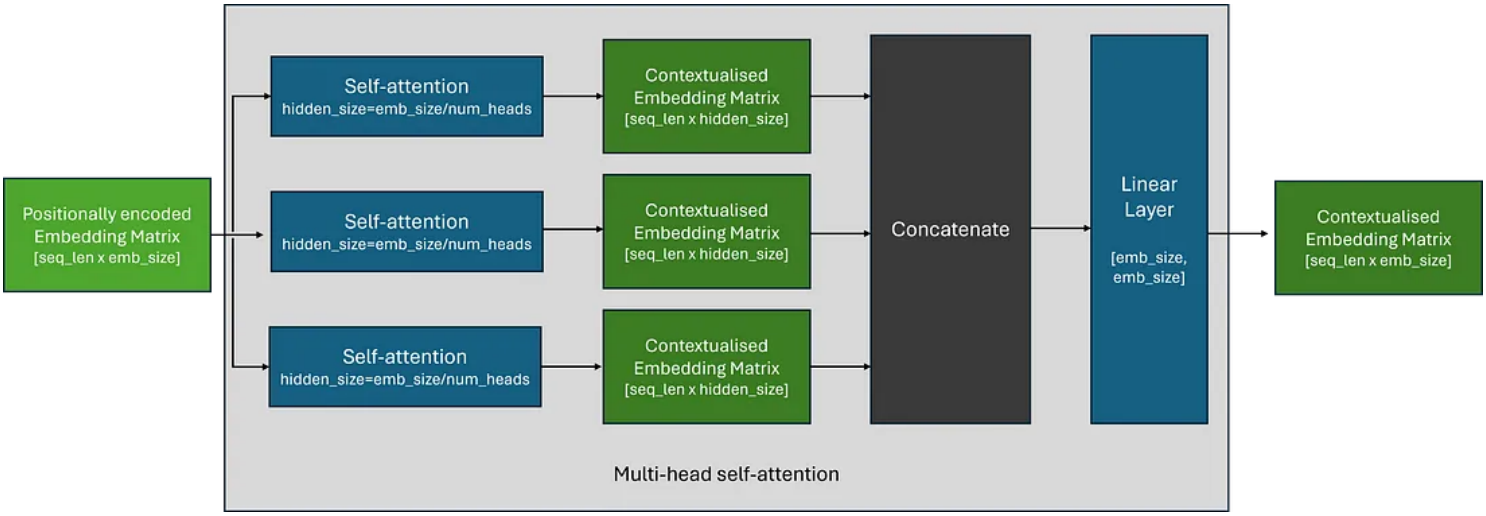
When we pass the same embedding matrix to Q, K and V, this is known as *self-attention*.

What is multi-head attention?

In practice, we often use multiple self-attention blocks in parallel, to enable the transformer to attend to different parts of the input sequence simultaneously– this is known as *multi-head attention*.

The idea behind multi-head attention is quite simple, the outputs of multiple independent self-attention blocks are concatenated together, and then passed through a linear layer. This linear layer enables the model to learn to combine the contextual information from each attention head.

In practice, the hidden dimension size used in each self-attention block is usually chosen to be the original embedding size divided by the number of attention heads; to preserve the shape of the embedding matrix.



What else makes up a Transformer?

Although the paper that introduced the transformer was (now infamously) named *Attention is all you need*, this is slightly confusing, as there are more components to a transformer than just attention!

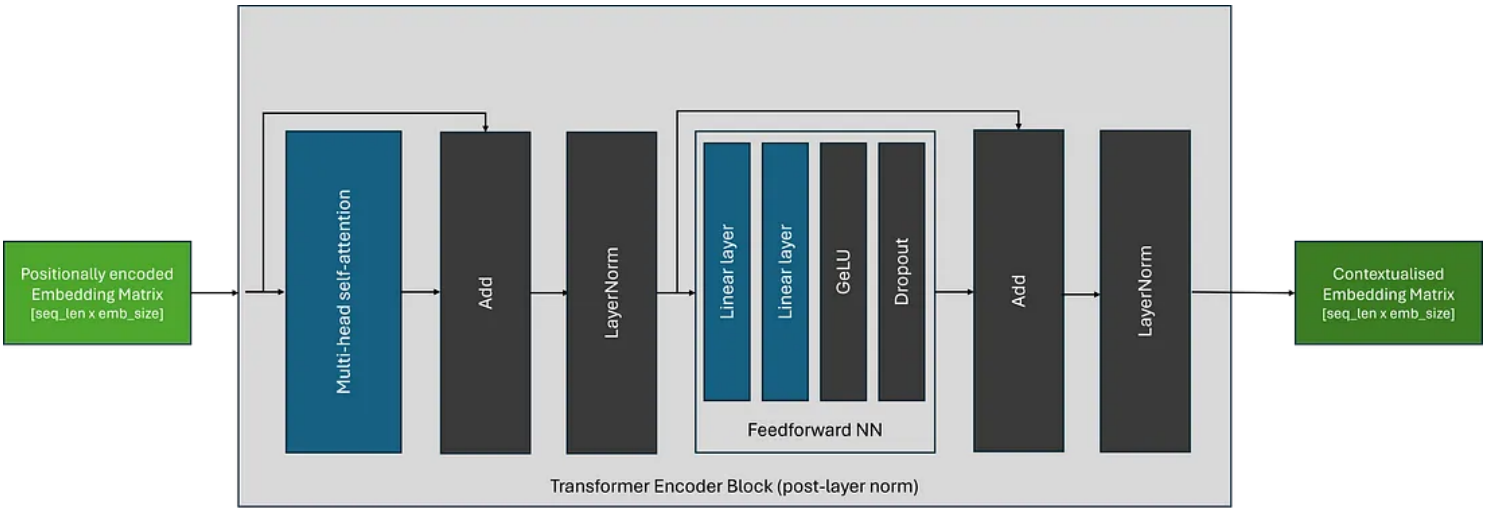
A transformer block also contains the following:

- **Feedforward Neural Network** (FFN): A two-layer neural network which is applied to each token embedding in the batch and sequence independently. The purpose of the FFN block is to introduce additional learnable parameters into the transformer, which are responsible for ensuring that the contextual embeddings are distinct and spread out. The original paper used a GeLU activation function, but the components of the FFN can vary depending on the architecture.
- **Layer Normalisation**: helps stabilize the training of deep neural networks, including transformers. It normalizes the activations for each sequence, preventing them from becoming too large or too small during training; which can lead to gradient-related issues like vanishing or

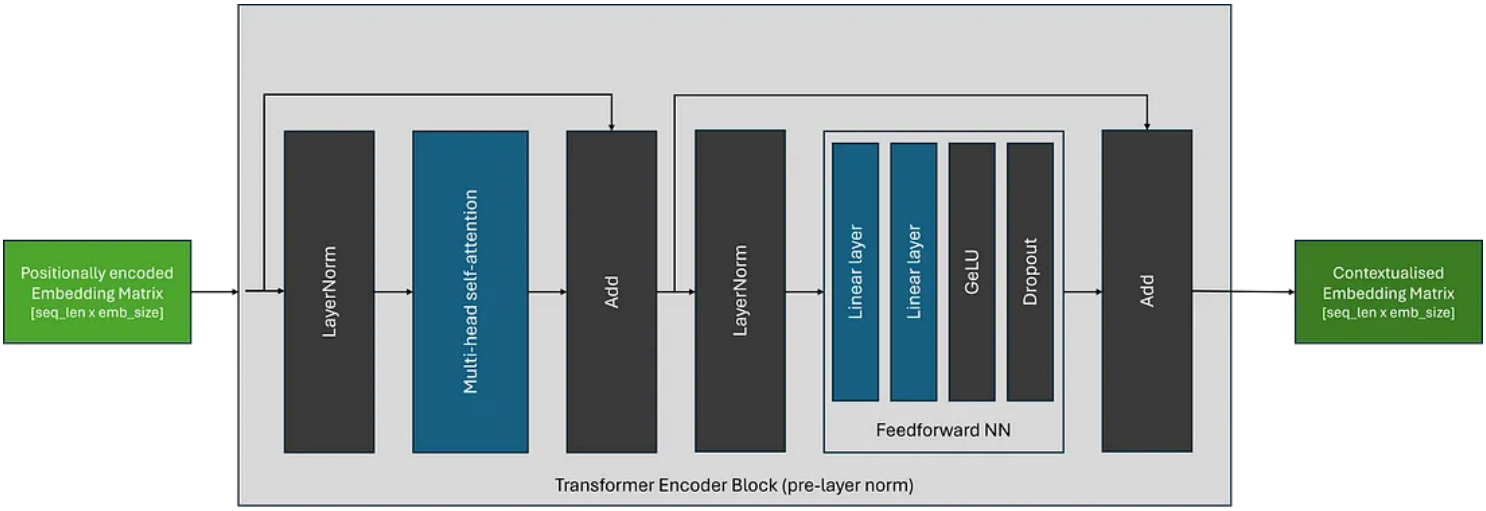
exploding gradients. This stability is crucial for effectively training very deep transformer models.

- **Skip connections:** Like in ResNet architectures, residual connections are used to mitigate the vanishing gradient problem and improve training stability.

Whilst the transformer architecture has remained fairly constant since its introduction, the positioning of the layer normalisation blocks can vary depending on the transformer architecture. The original architecture , now known as *post-layer norm* is presented below:



The most common placement in recent architectures, as seen in the figure below, is *pre-layer norm*, which places the normalisation blocks before the self-attention and FFN blocks, within the skip connections.



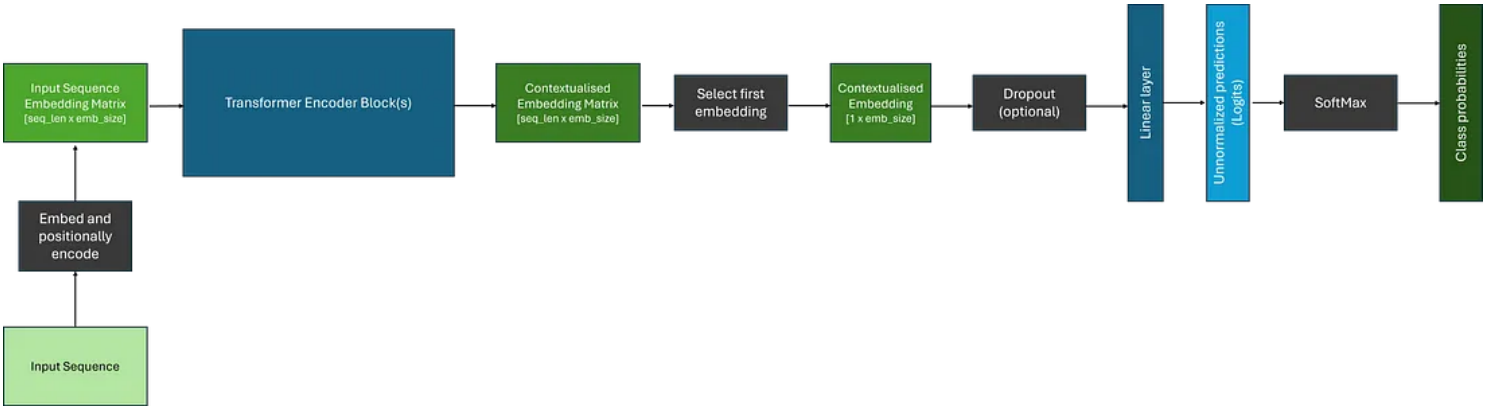
What are the different types of Transformer?

Whilst there are now many different transformer architectures, most can be categorised into three main types.

Encoder Architectures

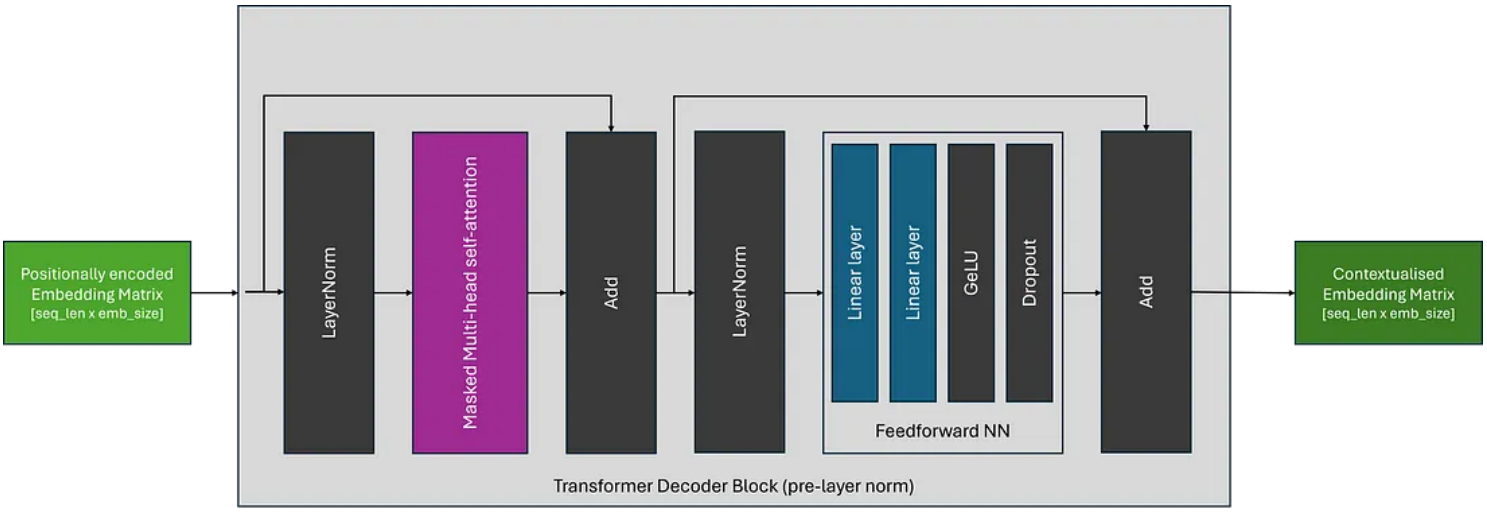
Encoder models aim to produce contextual embeddings that can be used for downstream tasks such as classification or named entity recognition, as the attention mechanism is able to attend over the entire input sequence; this is the type of architecture that has been explored in this article so far. The most popular family of encoder-only transformers are BERT, and its variants.

After passing our data through one or more transformer blocks, we have a complex contextualised embedding matrix, representing an embedding for each token in the sequence. However, to use this for a downstream task such as classification, we only need to make one prediction. Traditionally, the first token is taken, and passed through a classification head; which usually contains Dropout and Linear layers. The output of these layers can be passed through a SoftMax function to convert these into class probabilities. An example of how this could look is depicted below.



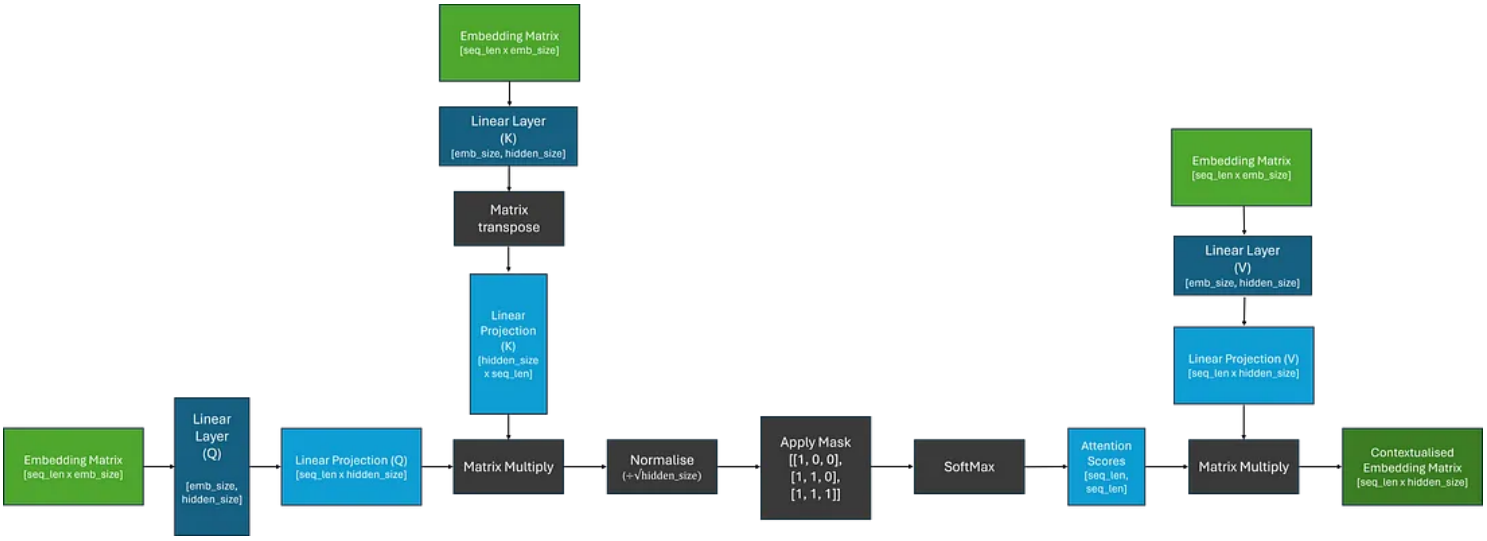
Decoder Architectures

Almost identical to Encoder architectures, the key difference is that Decoder architectures employ a *masked (or causal) self-attention layer*, so that the attention mechanism is only able to attend to the current and previous elements of the input sequence; this means that the contextual embedding produced only considers the previous context. Popular decoder-only models include the GPT family.



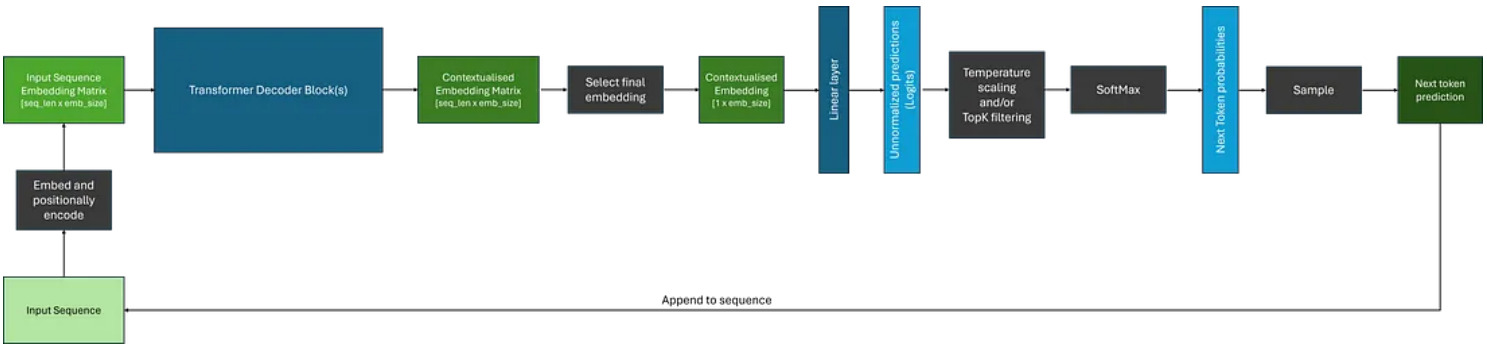
This is usually achieved by masking the attention scores with a binary lower triangular matrix and replacing the non-masked elements with negative infinity; when passed through the following SoftMax operation, this will

ensure that the attention scores for these positions are equal to zero. We can update our previous self-attention figure to include this as below.



Masked self-attention calculation: assuming positionally encoded embeddings

As they can only attend from the current position and backwards, Decoder architectures are usually employed for autoregressive tasks, such as sequence generation. However, when using contextual embeddings to generate sequences, there are some additional considerations when compared to using an Encoder. An example of this is shown below.



We can notice that, whilst the decoder produces a contextual embedding for each token in the input sequence, we usually use the embedding

corresponding to the final token as the input to subsequent layers when generating sequences.

Additionally, after applying the SoftMax function to the logits, if no filtering is applied, we would receive a probability distribution over every token in the model's vocabulary; this can be incredibly large! Often, we wish to reduce the number of potential options using various filtering strategies, some of the most common methods are:

- **Temperature Adjustment:** temperature is a parameter which is applied inside of the SoftMax operation that influences the randomness of the generated text. It determines how creative or focused the model's output is by altering the probability distribution over the output words. A higher temperature flattens the distribution, making outputs more diverse.
- **Top-P Sampling:** this approach filters the number of potential candidates for the next token based on a given probability threshold and redistributes the probability distribution based on candidates above this threshold.
- **Top-K Sampling:** this approach restricts the number of potential candidates to the *K* most likely tokens based on their logit or probability score (depending on implementation)

More details on these methods can be found here.

Once we have altered or reduced our probability distribution over the potential candidate for the next token, we can sample from this to get our prediction — this is just sampling from a multinomial distribution. The predicted token is then appended to the input sequence and fed back into

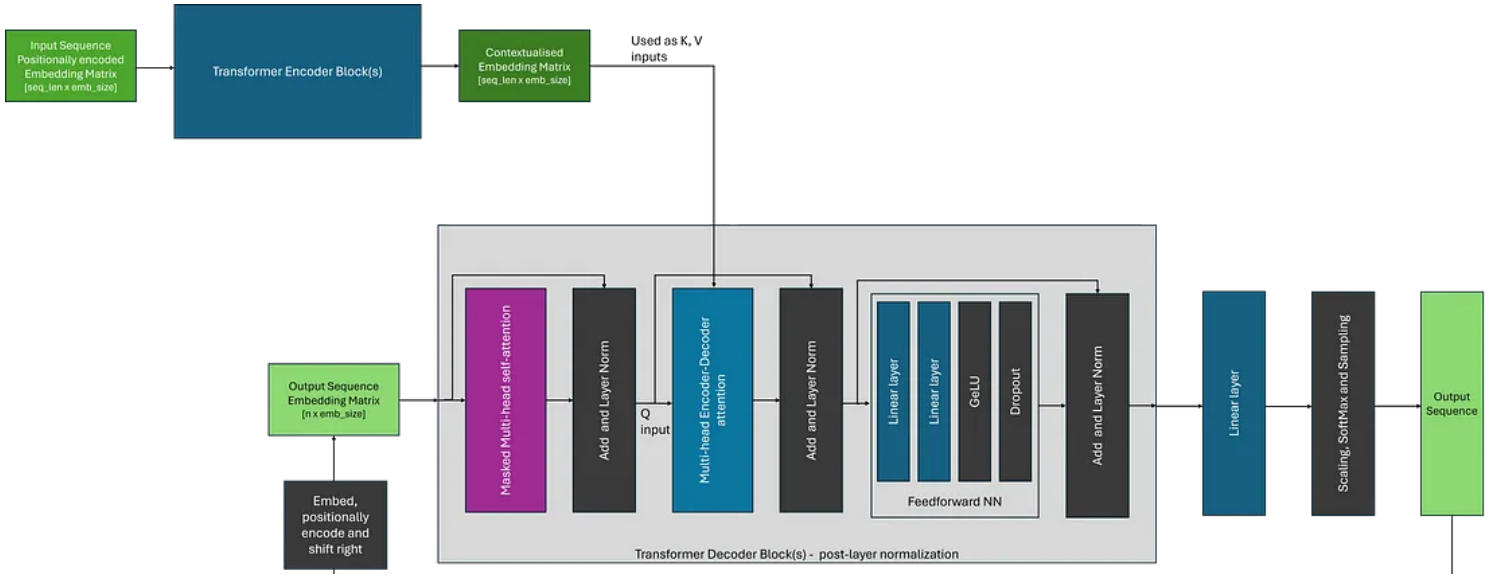
the model, until the desired number of tokens have been generated, or the model produces a *stop* token; a special token to signify the end of a sequence.

Encoder-decoder Architectures

Originally, the transformer was presented as an architecture for machine translation and used both an encoder and decoder to accomplish this goal; using the encoder to create an intermediate representation, before using the decoder to translate to the desired output format. Whilst encoder-decoder transformers have become less common, architectures such as T5 demonstrate how tasks such as question answering, summarisation and classification can be framed as sequence-to-sequence problems and solved using this approach.

The key difference with encoder-decoder architectures is that the decoder uses *encoder-decoder attention*, which uses both the outputs of the encoder (as K and V) and the inputs of the decoder block (as Q) during its attention calculation. This contrasts with self-attention, where the same input embedding matrix is used for all inputs. Aside from this, the overall generation process is very similar to using a decoder only architecture.

We can visualise an encoder-decoder architecture as seen in the figure below. Here, to simplify the figure, I chose to depict the *post-layer norm* variant of the transformer as seen in the original paper; where the layer norm layers are situated after the attention blocks.



Hopefully this has provided an intuition on how transformers work, helped to break down some of the details in a somewhat digestible way and has acted as good starting point into demystifying modern transformer architectures!

Chris Hughes is on LinkedIn

Unless otherwise stated, all images were created by the author.

References

- [1706.03762] Attention Is All You Need (arxiv.org)
- Recent Advances in Google Translate — Google Research Blog
- How GitHub Copilot is getting better at understanding your code — The GitHub Blog

- Introducing ChatGPT (openai.com)
- gpt-4.pdf (openai.com)
- Introducing LLaMA: A foundational, 65-billion-parameter language model (meta.com)
- The Illustrated Transformer — Jay Alammar — Visualizing machine learning one concept at a time. (jalammar.github.io)
- Byte-Pair Encoding tokenization — Hugging Face NLP Course
- Drenai series | David Gemmell Wiki | Fandom
- bert-base-uncased · Hugging Face
- 🤗 Transformers (huggingface.co)
- [2010.11929v2] An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (arxiv.org)
- Getting Started With Embeddings (huggingface.co)
- A Gentle Introduction to the Bag-of-Words Model — MachineLearningMastery.com
- [2104.09864] RoFormer: Enhanced Transformer with Rotary Position Embedding (arxiv.org)
- Scaled Dot-Product Attention Explained | Papers With Code
- Softmax function — Wikipedia
- [1607.06450] Layer Normalization (arxiv.org)
- Vanishing gradient problem — Wikipedia
- [1512.03385] Deep Residual Learning for Image Recognition (arxiv.org)

- [1810.04805] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (arxiv.org)
- [2005.14165] Language Models are Few-Shot Learners (arxiv.org)
- Token selection strategies: Top-K, Top-P, and Temperature (peterchng.com)
- Multinomial distribution — Wikipedia
- [1910.10683] Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (arxiv.org)

Transformers | Deep Learning | Machine Learning | Editors Pick | Data Science

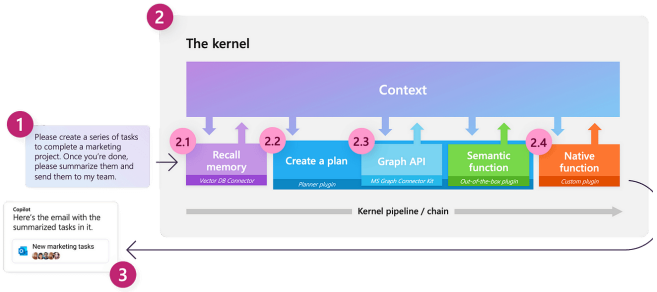
Written by Chris Hughes

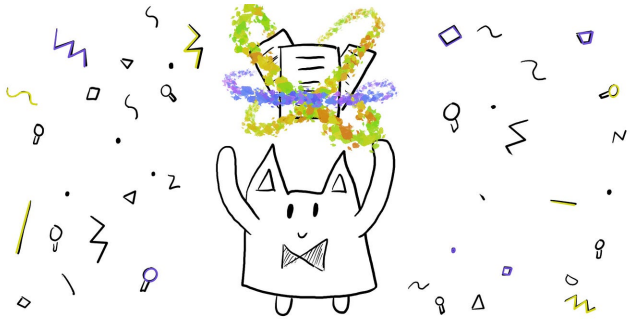
724 Followers · Writer for Towards Data Science

Principal Machine Learning Engineer/Scientist Manager at Microsoft. All opinions are my own.

Follow

More from Chris Hughes and Towards Data Science





Chris Hughes in Towards Data Science

A Pythonista's Intro to Semantic Kernel

In this blog post, I shall demonstrate how to get started with the Semantic Kernel using...

33 min read · Sep 3

153

Adrian H. Raudaschl in Towards Data Science

Forget RAG, the Future is RAG-Fusion

The Next Frontier of Search: Retrieval Augmented Generation meets Reciprocal...

· 10 min read · Oct 6

834 14

Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

10 min read · Sep 17

Chris Hughes in Towards Data Science

Getting Started with PyTorch Image Models (timm): a...

The purpose of this guide is to explore PyTorch Image Models (timm) from a...

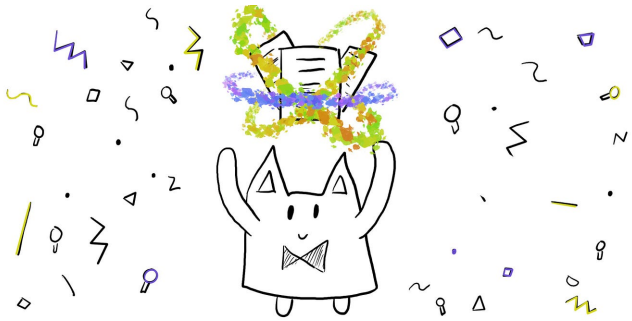
40 min read · Feb 2, 2022

553 11 ••• 918 10 •••

See all from Chris Hughes

See all from Towards Data Science

Recommended from Medium



Adrian H. Raudaschl in Towards Data Science

Forget RAG, the Future is RAG-Fusion

The Next Frontier of Search: Retrieval Augmented Generation meets Reciprocal...

✦ · 10 min read · Oct 6

834 14 •••

Diana Dovgopol in Artificial Corner

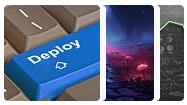
Now You Can Generate AI Images in ChatGPT with DALL-E 3

You don't need to be an expert in creating prompts to generate good images with DAL...

✦ · 8 min read · Oct 7

2.1K 39 •••

Lists



Predictive Modeling w/ Python

20 stories · 483 saves

Practical Guides to Machine Learning

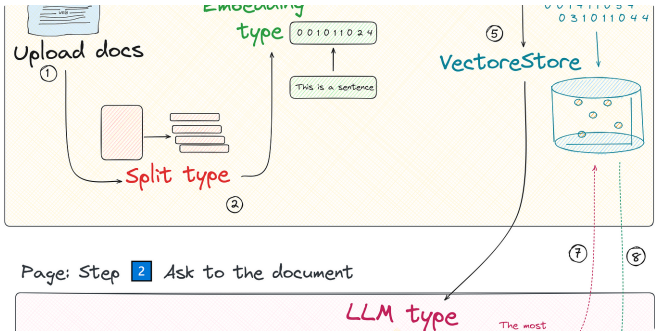
10 stories · 554 saves

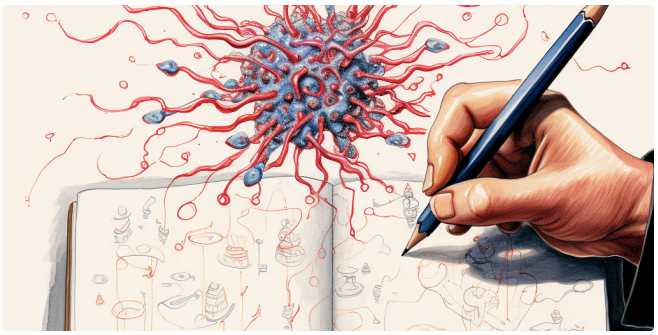
Natural Language Processing

698 stories · 309 saves

New_Reading_List

174 stories · 147 saves





Damian Gil in Towards AI

Talk to your documents as PDFs, txts, and even web pages

Complete guide to creating a web and the intelligence that allows you to ask questions...

13 min read · Aug 24

599 4

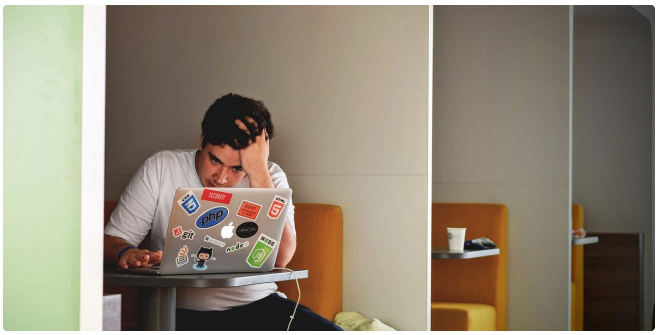
Thomas Smith in The Generator

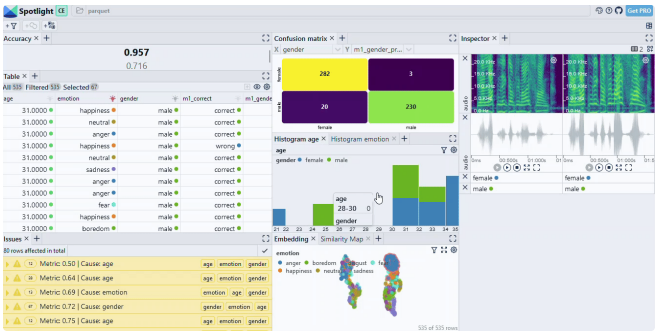
ChatGPT Vision is Changing How I Interact With the Real World

I'm using the system every day—but not how I expected

✦ · 8 min read · Oct 7

2.9K 49





Albers Uzila in Level Up Coding

How to Write Documentation for Your Data Science Projects Using...

Stefan Suwelack in ITNEXT

Hands-On Voice Analytics with Transformers

From logging, typing, and docstring, to beautiful documentation within minutes

· 15 min read · 5 days ago

418 3

Learn how to use open source models for gender detection and sentiment analysis.

7 min read · Sep 27

115 1

See more recommendations