TRANSFORMERS, TUTORIAL

Transformers Explained Visually - Overview of Functionality

Ketan Doshi

Dec 13, 2020 · 11 mins read

Photo by Arseny Togulev on Unsplash

A Gentle Guide to Transformers, how they are used for NLP, and why they are better than RNNs, in Plain English. How Attention helps i...

We've been hearing a lot about Transformers and with good reason. They have taken the world of NLP by storm in the last few years. The Transformer is an architecture that uses Attention to significantly improve the performance of deep learning NLP translation models. It was first introduced in the paper Attention is all you need and was quickly established as the leading architecture for most text data applications.

Since then, numerous projects including Google's BERT and OpenAI's GPT series have built on this foundation and published performance results that handily beat existing state-of-the-art benchmarks.

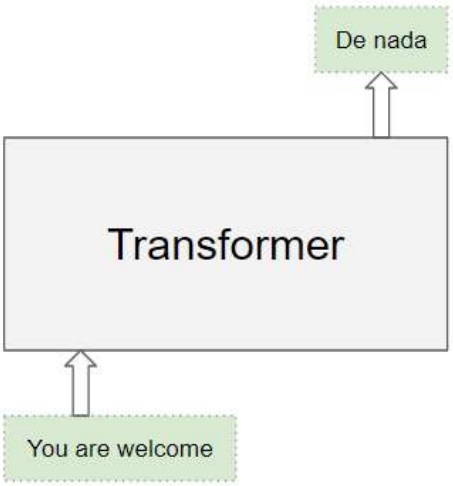
Over a series of articles, I'll go over the basics of Transformers, its architecture, and how it works internally. We will cover the Transformer functionality in a top-down manner. In later articles, we will look under the covers to understand the operation of the system in detail. We will also do a deep dive into the workings of the multi-head attention, which is the heart of the Transformer.

Here's a quick summary of the previous and following articles in the series. My goal throughout will be to understand not just how something works but why it works that way.

1. **Overview of functionality** — this article (*How Transformers are used, and why they are better than* RNNs. *Components of the architecture, and behavior during Training and Inference*)
2. **How it works** (*Internal operation end-to-end. How data flows and what computations are performed, including matrix representations*)
3. **Multi-head Attention** (*Inner workings of the Attention module throughout the Transformer*)
4. Why Attention Boosts Performance (*Not just what Attention does but why it works so well. How does Attention capture the relationships between words in a sentence*)

What is a Transformer

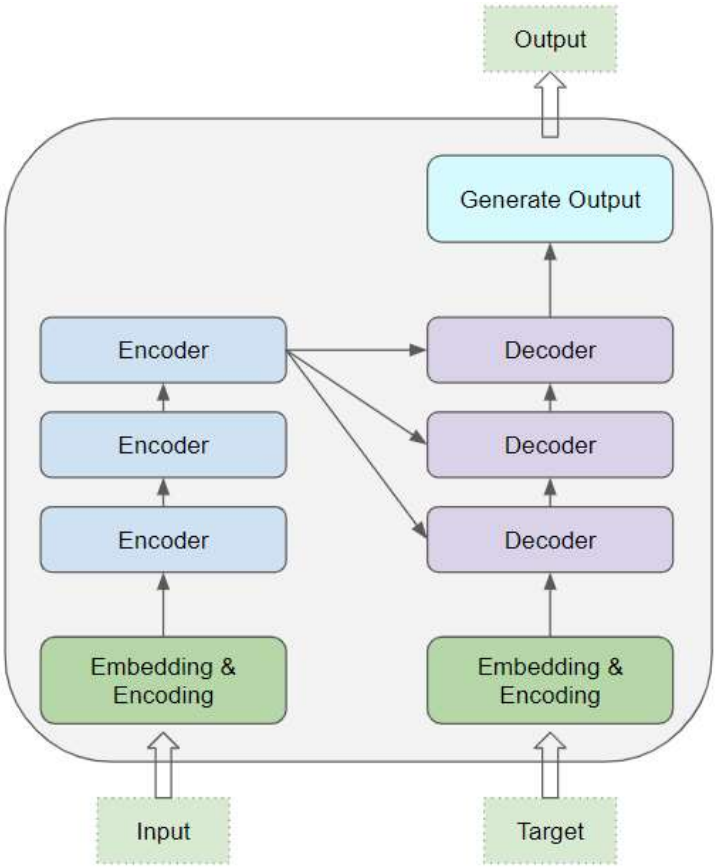
The Transformer architecture excels at handling text data which is inherently sequential. They take a text sequence as input and produce another text sequence as output. eg. to translate an input English sentence to Spanish.



(*Image by Author*)

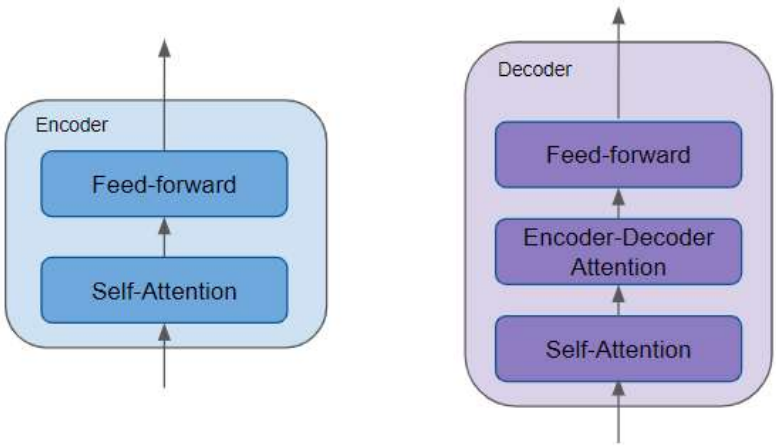
At its core, it contains a stack of Encoder layers and Decoder layers. To avoid confusion we will refer to the individual layer as an Encoder or a Decoder and will use Encoder stack or Decoder stack for a group of Encoder layers.

The Encoder stack and the Decoder stack each have their corresponding Embedding layers for their respective inputs. Finally, there is an Output layer to generate the final output.



(*Image by Author*)

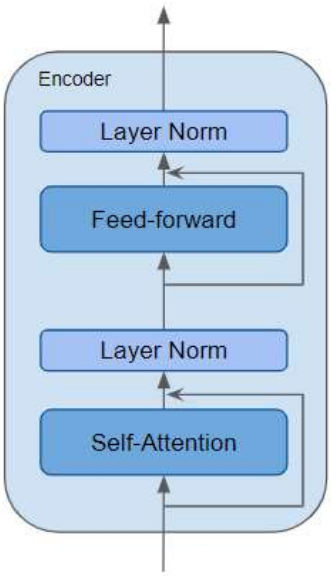
All the Encoders are identical to one another. Similarly, all the Decoders are identical.



(*Image by Author*)

- The Encoder contains the all-important Self-attention layer that computes the relationship between different words in the sequence, as well as a Feed-forward layer.
- The Decoder contains the Self-attention layer and the Feed-forward layer, as well as a second Encoder-Decoder attention layer.
- Each Encoder and Decoder has its own set of weights.

The Encoder is a reusable module that is the defining component of all Transformer architectures. In addition to the above two layers, it also has Residual skip connections around both layers along with two LayerNorm layers.



(*Image by Author*)

There are many variations of the Transformer architecture. Some Transformer architectures have no Decoder at all and rely only on the Encoder.

What does Attention Do?

The key to the Transformer's ground-breaking performance is its use of Attention.

While processing a word, Attention enables the model to focus on other words in the input that are closely related to that word.

eg. 'Ball' is closely related to 'blue' and 'holding'. On the other hand, 'blue' is not related to 'boy'.

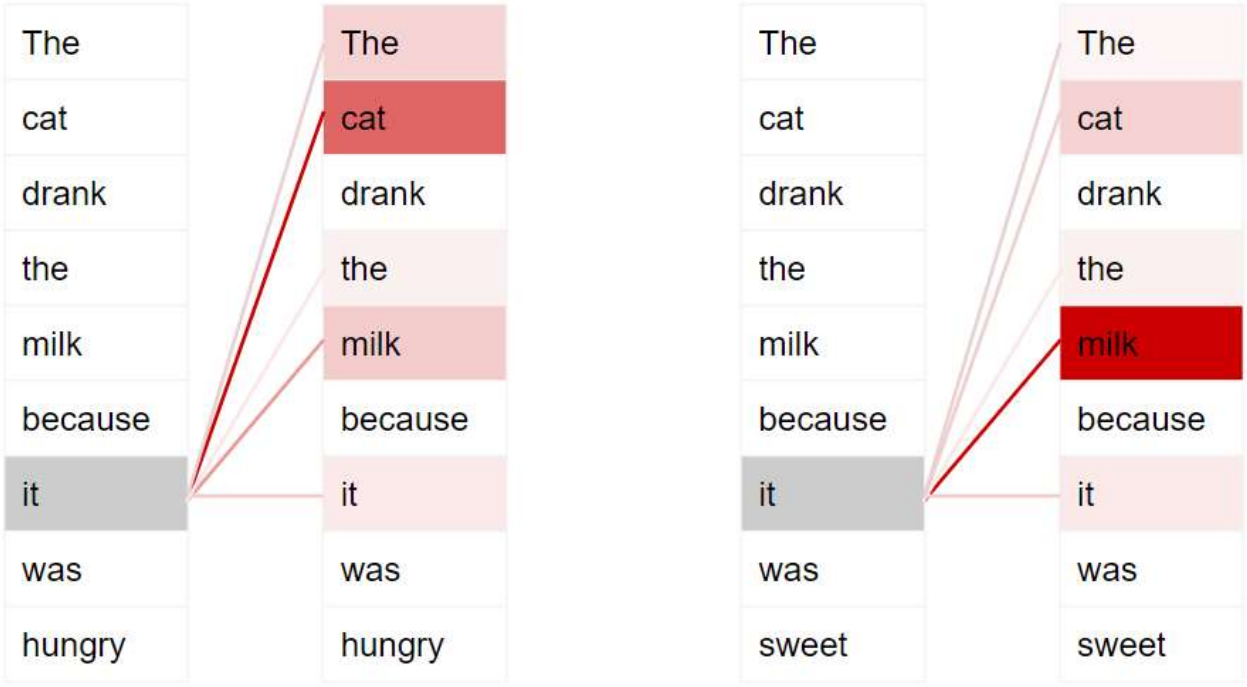
(*Image by Author*)

The Transformer architecture uses self-attention by relating every word in the input sequence to every other word.

eg. Consider two sentences:

- The *cat* drank the milk because **it** was hungry.
- The cat drank the *milk* because **it** was sweet.

In the first sentence, the word 'it' refers to 'cat', while in the second it refers to 'milk. When the model processes the word 'it', self-attention gives the model more information about its meaning so that it can associate 'it' with the correct word.



Dark colors represent higher attention (Image by Author)

To enable it to handle more nuances about the intent and semantics of the sentence, Transformers include multiple attention scores for each word.

eg. While processing the word 'it', the first score highlights 'cat', while the second score highlights 'hungry'. So when it decodes the word 'it', by translating it into a different language, for instance, it will incorporate some aspect of both 'cat' and 'hungry' into the translated word.

(Image by Author)

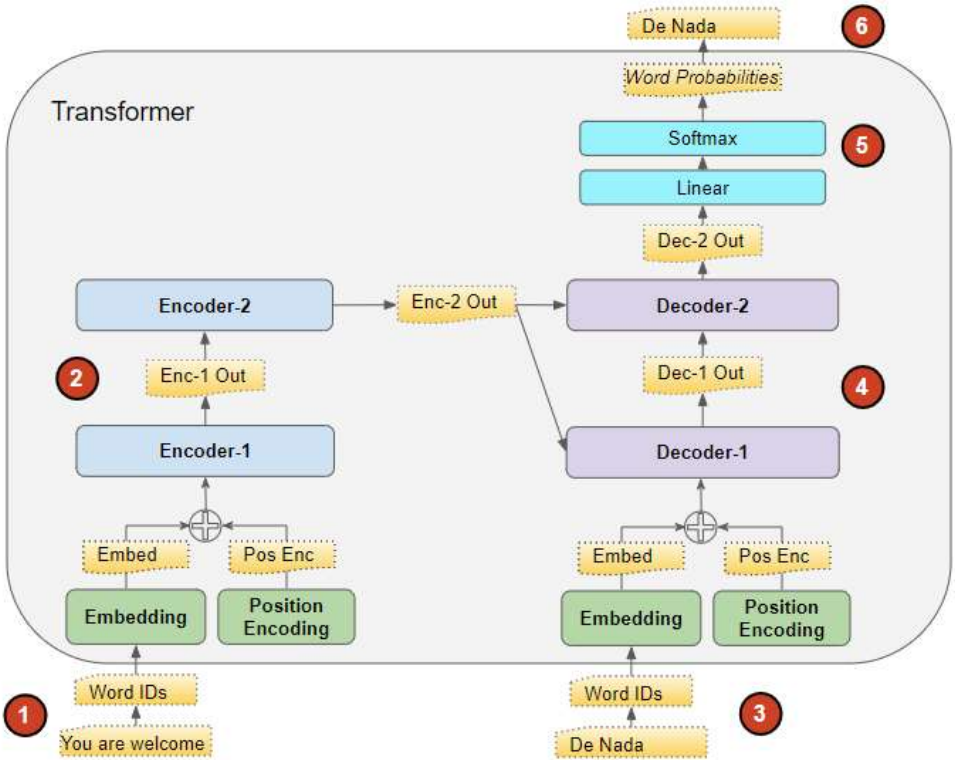
Training the Transformer

The Transformer works slightly differently during Training and while doing Inference.

Let's first look at the flow of data during Training. Training data consists of two parts:

- The source or input sequence (eg. "You are welcome" in English, for a translation problem)
- The destination or target sequence (eg. "De nada" in Spanish)

The Transformer's goal is to learn how to output the target sequence, by using both the input and target sequence.



(Image by Author)

The Transformer processes the data like this:

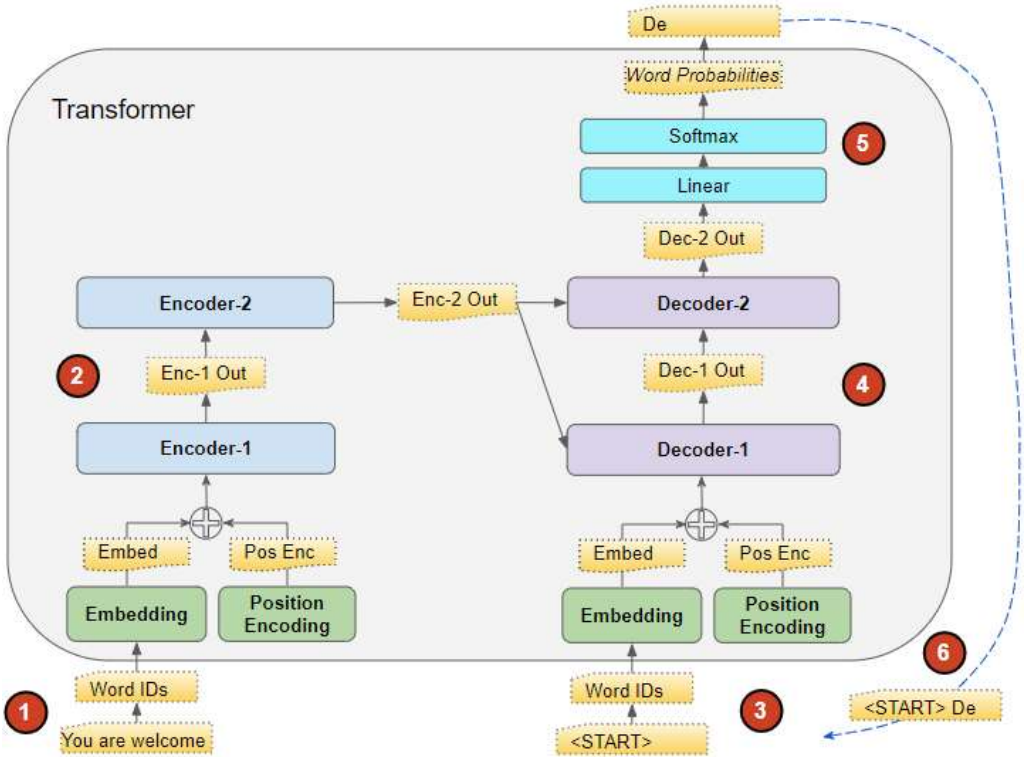
1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
3. The target sequence is prepended with a start-of-sentence token, converted into Embeddings (with Position Encoding), and fed to the Decoder.
4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
5. The Output layer converts it into word probabilities and the final output sequence.
6. The Transformer's Loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients to train the Transformer during back-propagation.

Inference

During Inference, we have only the input sequence and don't have the target sequence to pass as input to the Decoder. The goal of the Transformer is to produce the target sequence from the input sequence alone.

So, like in a Seq2Seq model, we generate the output in a loop and feed the output sequence from the previous timestep to the Decoder in the next timestep until we come across an end-of-sentence token.

The difference from the Seq2Seq model is that, at each timestep, we re-feed the entire output sequence generated thus far, rather than just the last word.



Inference flow, after first timestep (Image by Author)

The flow of data during Inference is:

1. The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
2. The stack of Encoders processes this and produces an encoded representation of the input sequence.
3. Instead of the target sequence, we use an empty sequence with only a start-of-sentence token. This is converted into Embeddings (with Position Encoding) and fed to the Decoder.
4. The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
5. The Output layer converts it into word probabilities and produces an output sequence.
6. We take the last word of the output sequence as the predicted word. That word is now filled into the second position of our Decoder input sequence, which now contains a start-of-sentence token and the first word.

7. Go back to step #1. As before, feed the Encoder input sequence and the new Decoder sequence into the model. Then take the second word of the output and append it to the Decoder sequence. Repeat this until it predicts an end-of-sentence token.

Teacher Forcing

The approach of feeding the target sequence to the Decoder during training is known as Teacher Forcing. Why do we do this and what does that term mean?

During training, we could have used the same approach that is used during inference. In other words, run the Transformer in a loop, take the last word from the output sequence, append it to the Decoder input and feed it to the Decoder for the next iteration. Finally, when the end-of-sentence token is predicted, the Loss function would compare the generated output sequence to the target sequence in order to train the network.

Not only would this looping cause training to take much longer, but it also makes it harder to train the model. The model would have to predict the second word based on a potentially erroneous first predicted word, and so on.

Instead, by feeding the target sequence to the Decoder, we are giving it a hint, so to speak, just like a Teacher would. Even though it predicted an erroneous first word, it can instead use the correct first word to predict the second word so that those errors don't keep compounding.

In addition, the Transformer is able to output all the words in parallel without looping, which greatly speeds up training.

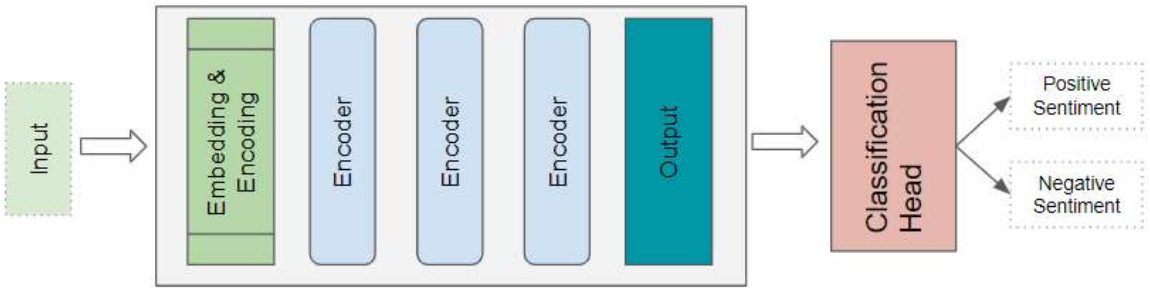
What are Transformers used for?

Transformers are very versatile and are used for most NLP tasks such as language models and text classification. They are frequently used in sequence-to-sequence models for applications such as Machine Translation, Text Summarization, Question-Answering, Named Entity Recognition, and Speech Recognition.

There are different flavors of the Transformer architecture for different problems. The basic Encoder Layer is used as a common building block for these architectures, with different application-specific 'heads' depending on the problem being solved.

Transformer Classification architecture

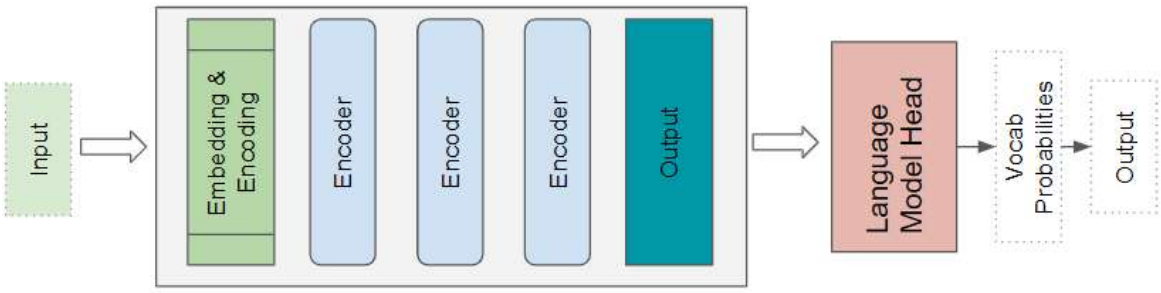
A Sentiment Analysis application, for instance, would take a text document as input. A Classification head takes the Transformer's output and generates predictions of the class labels such as a positive or negative sentiment.



(*Image by Author*)

Transformer Language Model architecture

A Language Model architecture would take the initial part of an input sequence such as a text sentence as input, and generate new text by predicting sentences that would follow. A Language Model head takes the Transformer's output and generates a probability for every word in the vocabulary. The highest probability word becomes the predicted output for the next word in the sentence.



(*Image by Author*)

How are they better than RNNs?

RNNs and their cousins, LSTMs and GRUs, were the de facto architecture for all NLP applications until Transformers came along and dethroned them.

RNN-based sequence-to-sequence models performed well, and when the Attention mechanism was first introduced, it was used to enhance their performance.

However, they had two limitations:

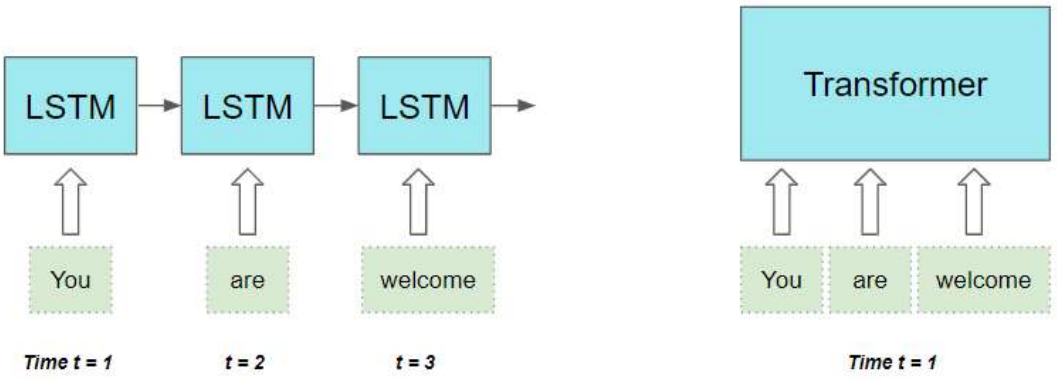
- It was challenging to deal with long-range dependencies between words that were spread far apart in a long sentence.
- They process the input sequence sequentially one word at a time, which means that it cannot do the computation for time-step t until it has completed the computation for time-step $t - 1$. This slows down training and inference.

As an aside, with CNNs, all of the outputs can be computed in parallel, which makes convolutions much faster. However, they also have limitations in dealing with long-range dependencies:

- In a convolutional layer, only parts of the image (or words if applied to text data) that are close enough to fit within the kernel size can interact with each other. For items that are further apart, you need a much deeper network with many layers.

The Transformer architecture addresses both of these limitations. It got rid of RNNs altogether and relied exclusively on the benefits of Attention.

- They process all the words in the sequence in parallel, thus greatly speeding up computation.



(Image by Author)

- The distance between words in the input sequence does not matter. It is equally good at computing dependencies between adjacent words and words that are far apart.

Conclusion

Now that we have a high-level idea of what a Transformer is, we can go deeper into its internal functionality in the next article to understand the details of how it works.

And finally, if you are interested in NLP, you might also enjoy my article on Beam Search, and my other series on Audio Deep Learning and Reinforcement Learning.

State-of-the-Art Techniques

Reinforcement Learning Made Simple (Part 1): Intro to Basic Concepts and Terminology

Let's keep learning!

featured

Ketan Doshi Copyright © 2023.