Attention Is All You Need : A Complete Guide to Transformers

Alejandro Ito Aramendia · Follow

11 min read · Jan 1, 2024

Photo by Renzo Vanden Bussche on Unsplash

Introduction to Transformers

In recent years, the field of artificial intelligence has experienced a paradigm shift with the introduction of transformer based models. Transformers have proven to be pivotal in the development of sharper Natural Language Processing (NLP) models, while also extending its employability in additional areas within machine learning.

The transformer is an architecture that relies on the concept of **attention**, a technique used to provide weights to different parts of an input sequence so that a better understanding of its underlying context is achieved. This allows transformers to perform **machine translation**, **text generation** and many other NLP tasks.

In addition, transformers process inputs in **parallel** making them more efficient and scalable in comparison to traditional sequential models such as RNN and LSTM.

In this article, I will be dissecting the paper, '**Attention Is All You Need**' and its significance in artificial intelligence. All you need will be here, so Attention!

Model Architecture

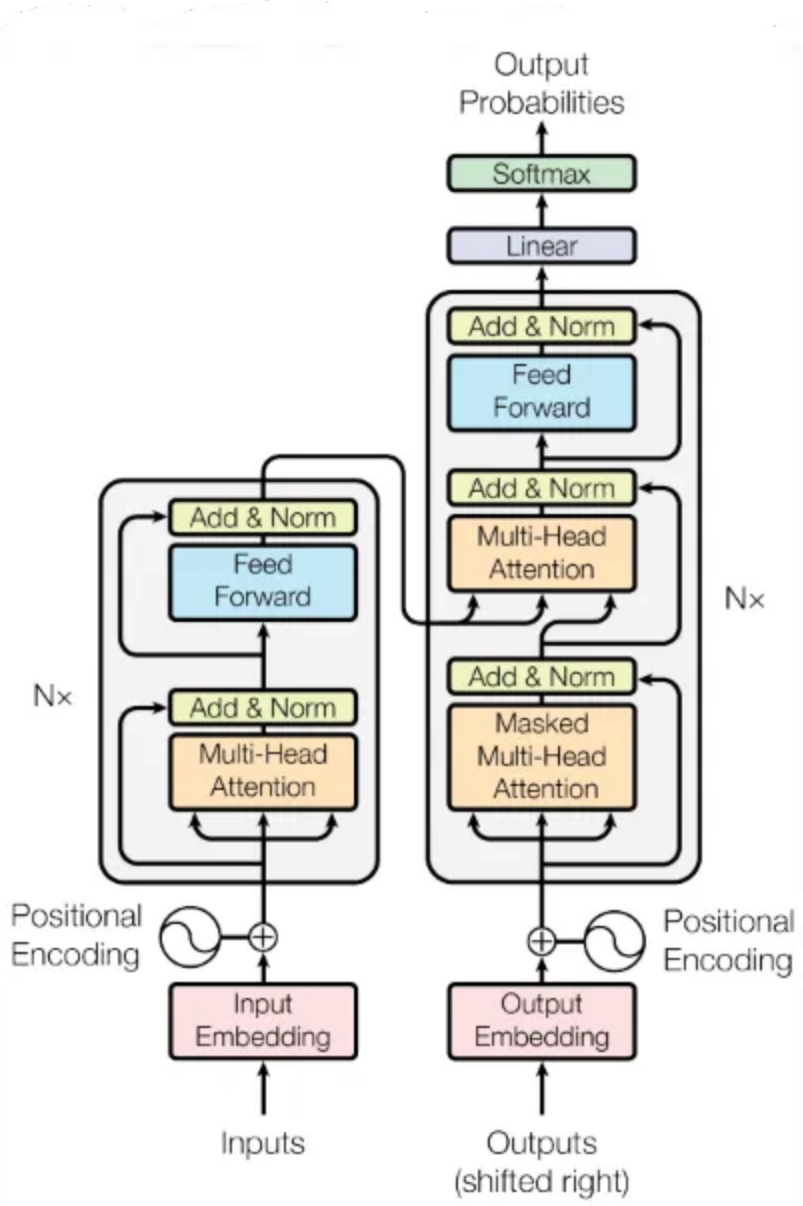
- Encoder
- Decoder
- Attention
- Feed-Forward Networks (FFN)
- Layer Normalisation

- Positional Encoding

The transformer follows an **encoder-decoder** structure, where an input vector **x** is fed into an encoder resulting in a mapped representation **z.** With **z**, the decoder generates an output **y**, one element at a time, **i.e.** first generating y_1, then y_2, all the way until y_m. At each of these steps, the model is **auto-regressive,** meaning that previously generated outputs will always affect future outputs.

The transformer follows an architecture containing stacked **attention layers** followed by **fully connected layers** in both the encoder and decoder. This architecture is illustrated below.

If this seems daunting, do not worry — it will all make sense shortly!



Architectural diagram of a transformer with the **Encoder** and **Decoder** on the **left** and **right**, respectively.

Encoder

The encoder used in this paper is composed of $N = 6$ identical stacked layers. Each layer is divided into **two sub-layers**, the first being a multi-head self-attention layer and the second being a position-wise fully connected feed-forward network layer. The encoder further employs **residual connections** around each sub-layer, which is then followed by layer normalisation.

> ***Residual connections:*** *Adds the input value* ***x*** *to the sub-layer's output* ***f(x)*** *through a separate path, which does not pass through the sub-layer's function. The purpose of this is to directly pass gradients to subsequent layers, thus mitigating vanishing gradients and simplifying the learning of the identity function (if necessary).*

Therefore, the final output of each sub-layer can be described as ***LayerNorm(x + f(x))*** where ***f(x)*** is the function used in the corresponding sub-layer.

Decoder

Like the encoder, the decoder is comprised of ***N = 6*** identical layers. While the encoder only has two sub-layers, the decoder introduces a **third sub-layer** designed to perform multi-head attention over the outputs of the encoder stack. Just like the encoder, residual connections and layer normalisation are still implemented.

In order to make sure that the model is auto-regressive, that is, with only previous positions influencing the present position, the first sub-layer must be modified. Having a **masking mechanism** and the output embeddings offset by one position ensures that the prediction of position ***i,*** solely depends on the known outputs preceding ***i***.

> ***Masking:*** *A triangular matrix with the upper triangle (including the diagonal) filled with* ***-∞*** *and the lower triangle filled with* ***0s****. During attention, masking ensures that positions with values* ***-∞*** *are ignored, i.e. any position to the right. This happens because when the* ***-∞ attention scores*** *pass through the softmax function,* **e-∞ = 0** and so are therefore ignored.

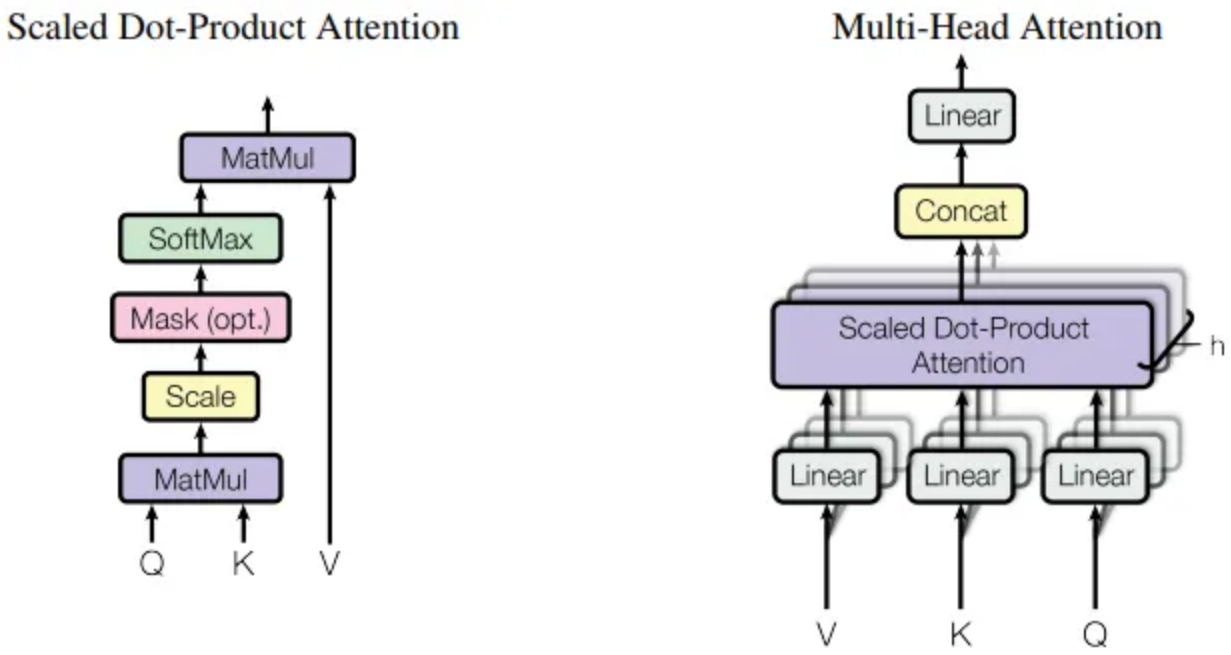
Attention

Now this part is crucial.

The attention mechanism lays the foundation of the transformer model and therefore, will be explained as coherently as possible in the coming section.

Attention is a mechanism used to accurately evaluate how important different sections of an input sequence are to a global context. In this article, I will discuss three attention variants included in the paper.

- Self-Attention
- Multi-Head Attention
- Cross-Attention



The attention architectures used in the transformer model.

Self-Attention

Imagine, you are given the input sequence:

I would like to taste the local food in France.

In this sentence, it is clear that the words: ***taste***, ***local***, ***food*** and ***France*** play significant roles in understanding its underlying context.

Attention operates on three inputs: **Queries** (Q), **Keys** (K) and **Values** (V).

At first glance, these inputs may seem bizarre and confusing, especially given their names. So let me break it down intuitively before I move on.

> *Think of YouTube.*
>
> *First you enter your **Query** in the search bar. Then your **Query** is compared against a set of **Keys** (in this case, video titles, tags and descriptions etc. within the YouTube database). After this, YouTube proceeds to retrieve the videos that best match your **Query**. These video results are referred to as **Values**.*
>
> *There we go, that's all it is.*

In the case of self-attention, the **Query, Key** and **Value** are all derived from the same input. This is because we are only analysing one input vector.

Query, **Key** and **Value** can be calculated through the matrix multiplication of input matrix **X** and a learnt matrix **W**. The weight matrix **W** can be learnt via back propagation on the transformer's loss function.

$$Q = XW_Q \qquad K = XW_K \qquad V = XW_V$$

The derivation of the **Query**, **Key** and **Value** matrices.

In addition, the comparison between **Query** and **Key** can be represented through a **compatibility matrix.** This matrix finds the **similarity** or **compatibility** between two input vectors. Calculating this matrix, therefore, requires a **compatibility function**, which in the case of *'Attention Is All You Need'* is the dot product.

$$Compatibility(Q, K) = Q \cdot K = QK^T =$$

	I	would	like	to	taste	the	local	food	in	France
I	h									
would		h								
like			h							
to				h						
taste					h			h		
the						h				
local							h			h
food					h			h		
in									h	
France							h			h

A compatibility matrix produced from our sentence example with the word combinations likely to score high in compatibility marked in **h**. In reality, every cell would be given a compatibility score, however, these were not marked in order to preclude overwhelming the image.

When dealing with high-dimensionality matrices, the dot product inevitably gets very large. This can be problematic later on given the **softmax function's** sensitivity to large values. During back propagation, a large dot product would produce extremely small gradients (a **vanishing gradient**).

Therefore, in order to resolve this, the **dot product** is scaled by a **division** of $\sqrt{d_k}$ where d_k is the dimensionality of **Query** and **Key**. Now this is where the name **Scaled-Dot Product Attention** comes from.

While the compatibility scores are now somewhat scaled, they are still not quite fully interpretable. In order to obtain a set of weights that accurately evaluate each word in relation to the global context, a **softmax function** is applied.

$$softmax\left(\frac{Q_i K^T}{\sqrt{d_k}}\right) = \frac{e^{\left(\frac{Q_i K^T}{\sqrt{d_k}}\right)}}{\sum_{j=1}^{d_k} e^{\left(\frac{Q_j K^T}{\sqrt{d_k}}\right)}}$$

The objective of the softmax function is to **normalise** the values in the compatibility matrix to values between **0** and **1.** This ensures that the compatibility scores follow a **probability distribution** with the weights of each word summing to **1.** This result is called the **Attention Weight Matrix.**

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) =$$

Attention Weight Matrix

	I	would	like	to	taste	the	local	food	in	France	
I								0.02			
would								0.02			
like								0.05			
to								0.01			
taste								0.2			
the								0.01			
local								0.2			
food	0.02	0.02	0.05	0.01	0.2	0.01	0.2	0.1	0.09	0.3	= 1
in								0.09			
France								0.3			
								= 1			

The resulting attention weight matrix after the softmax function. For clarity purposes, only the '**food**' row and column have been filled. The word combinations that are important to the context are marked in red with their respective weights.

These probability values represent how much attention should be given to each element in the sequence and indicates their importance or contribution to the final context vector.

Finally, there is one more step remaining in the self-attention mechanism, the multiplication of **Value**.

The **Value** matrix is composed of word embedding vectors corresponding to each element of the input sequence. So in essence, the **Value** matrix is a representation of the input sequence.

> ***Word embedding:*** *A word embedding is vector representation of a word or token in a continuous vector space and encapsulates its meaning and context through its position and direction. Words with similar meaning or context hold similar vector representations, 'Queen' and 'King' are good examples.*

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)\mathrm{V}$$

Why do we multiply by **V**? Well, **V** is the matrix representation of the input sequence. Now, the output of the softmax function is a weighting matrix, i.e. telling us how much attention or focus we should pay to each element. Therefore, when multiplying these together, the resulting **attention matrix** conveys the meaning and context of the input sequence.

And there you have it, this is self-attention.

Multi-Head Attention

Initially, performing a single attention operation with $\mathbf{d_m}$**-dimensional Keys**, **Queries** and **Values** could be done, where $\mathbf{d_m}$ is the dimensionality of the model. However, a more effective method exists.

Multi-Head Attention linearly projects the initial **Queries**, **Keys** and **Values h times** to matrices with $\mathbf{d_k}$, $\mathbf{d_k}$ and $\mathbf{d_v}$ dimensionality, respectively. Following this, the standard attention mechanism is applied to the **h** unique **Q**, **K** and **V** projections in **parallel**.

By having each attention head compute its own attention weights and context vector **independently**, the transformer is able focus on different

aspects of the input sequence simultaneously while capturing complex patterns and relationships more effectively.

Subsequently, the **h** resulting outputs are then concatenated and projected via $\mathbf{W_o}$. The purpose of $\mathbf{W_o}$ is to combine the results from each individual attention head into one final output. With a single attention head, such results would not be possible.

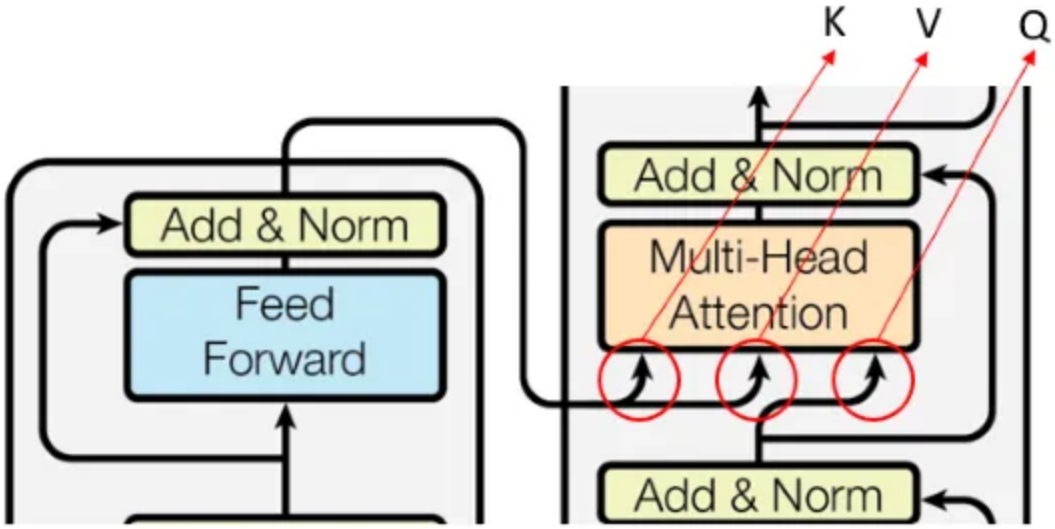
$$MultiHead(Q,K,V) = Concat(head_1, \dots, head_h)W_O$$

$$where\ head_i = Attention(QW_{Q,i}, KW_{K,i}, VW_{V,i})$$

Cross-Attention

In self-attention there is only one input sequence, however, in cross-attention there are **two input sequences**. Simply, that's the only difference.

Cross-attention acts as the bridge between the encoder and decoder. The output of the encoder is used as the **Key** and **Value** in the second multi-head attention layer of the decoder. Furthermore, the **Query** is derived from a separate input sequence.



A magnified image of the encoder-decoder architecture showing the multi-head attention layer that utilises cross-attention.

Take **machine translation** for example, when using a transformer to translate from English to Spanish, we must train the model with two input sequences. The first input being the English sequence and the second input sequence being the Spanish translation . These are both fed into the encoder and decoder stacks, respectively.

By doing this, the model is able to learn different relationships and language patterns between the English sequence and its Spanish counter part. Therefore, when trained, the transformer is able to smoothly translate an English sequence into Spanish.

Feed-Forward Network (FFN)

In addition to the attention sub-layers, both the encoder and decoder implement another sub-layer called the fully connected feed-forward layer. This layer is applied independently and parallelly to each position of the input sequence.

This FFN consists of **two linear transformations** with a **ReLU activation function** placed in between.

> *ReLU: A type of non-linear activation function typically used in neural networks and follows the equation* **ReLU(x) = max(0, x)**.

$$FFN(x) = \max(0, xW_1 + b_1)\, W_2 + b_2$$

The first linear transformation projects the vector into a higher dimensional space allowing for an increased capacity to learn intricate relationships. In this paper, the input was transformed from $\mathbf{d_k = 512}$ to **2048.**

Furthermore, by additionally utilising **ReLU**, the model is capable of learning complex **non-linear** relationships. It is then put through the second linear transformation, which reduces the matrix back to its original dimensionality.

Layer Normalisation

Layer normalisation is an important step that follows after the multi-head attention and feed-forward network sub-layers. Its purpose is to **normalise** and **scale** the outputs of these sub-layers, ensuring a **constant mean** ($\mu = 0$) and **variance** ($\sigma^2 = 1$) throughout the transformer. This increases the transformers stability by preventing any **vanishing** or **exploding gradient** during training.

$$LayerNorm(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

- x represents one of the outputs from the sub-layer.
- μ represents the **mean** of all the $\mathbf{d_v}$ outputs of the sub-layer.
- σ^2 represents the **variance** between all the outputs of the sub-layer.
- ϵ represents an **extremely small** positive number that prevents any division by 0 error.
- γ is a **scaling parameter** that is learnt during training.

- β is a **shifting parameter** that is learnt during training.

Positional Encoding

The order of the input sequence is very important for understanding the meaning and context of a sentence. Take these two sentences for example,

'Alex ate a frog' and **'A frog ate Alex'.**

While using exactly the same words, these two sentences convey vastly different meanings. For the transformer to understand the relative or absolute positions of tokens in a sequence, a '**positional encoding**' must be added to the input embeddings at the bottoms of the encoder and decoder stacks.

In order to differentiate between different positions in the input sequence, the '**positional encoding**' uses **sine** and **cosine** functions of different frequencies.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_m})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_m})$$

Where ***pos*** is position in the sequence and ***i*** is the dimension or index of the positional encoding vector where $\mathbf{0 \le 2i < d_m}$.

Each position in the sequence holds a **unique** positional encoding vector. Realising $\mathbf{2i/d_m \le 1}$, the wavelengths form a geometric sequence from 2π *to* $10{,}000 \cdot 2\pi$. (Since $\lambda = 2\pi/f$).

Furthermore, due to the sinusoidal nature of these functions, the model is able to learn relative positions for any offset k. In other words, PE_{pos+k} can be represented as a linear function of PE_{pos} (think of the rotation matrix).

		Positional Encoding Matrix			
Sequence	Position	$i = 0$	$i = 0$	$i = 1$	$i = 1$
Alex	0	$PE_{00} = \sin(0) = 0$	$PE_{01} = \cos(0) = 1$	$PE_{02} = \sin(0) = 0$	$PE_{03} = \cos(0) = 1$
ate	1	$PE_{10} = \sin(1/1) = 0.84$	$PE_{11} = \cos(1/1) = 0.54$	$PE_{12} = \sin(1/100) = 0.01$	$PE_{13} = \cos(1/100) = 1.0$
a	2	$PE_{20} = \sin(2/1) = 0.91$	$PE_{21} = \cos(2/1) = -0.42$	$PE_{22} = \sin(2/100) = 0.02$	$PE_{23} = \cos(2/100) = 1.0$
frog	3	$PE_{30} = \sin(3/1) = 0.14$	$PE_{31} = \cos(3/1) = -0.99$	$PE_{32} = \sin(3/100) = 0.03$	$PE_{33} = \cos(3/100) = 1.0$

The calculations for the positional encoding vectors of each word in the sequence.

As visible in the table, the positional encoding vectors possess $\mathbf{d_m}$ dimensions. This is necessary as the positional encoding vectors must be able to add element-wise with the **$\mathbf{d_m}$-dimensional** word embedding vectors in order to be later fed into the encoder and decoder stacks.

All Together: The Transformer

Now combine all these previously discussed components and we have a transformer. Great! I hope it made some sense.

And in case not, let's quickly summarise.

The transformer's primary objective is to perform sequence-to-sequence tasks such as machine translation, text summarisation and other NLP tasks.

It does this by capturing complex relationships and long-range dependencies, something previous models struggled to achieve.

The attention mechanism is fundamental to the transformer, by focusing on several areas of a sequence in parallel, it is able to surpass other traditional models such as RNNs in both efficiency and accuracy.

The transformer follows an encoder-decoder structure. At first, word embedding vectors of the input sequence are created. These are then added element-wise to the positional encoding vectors. This resulting matrix is then fed into the encoder and decoder, where it will then under go multi-head self-attention, fully connected feed-forward networks and layer normalisation. Residual connections are additionally implemented in order to help preserve information and make the model more efficient. In order for the encoder to communicate with the decoder, the encoder's output is used as two of the inputs in the second multi-head attention layer of the decoder.

After the encoder-decoder mechanism, the outputs are linearly transformed and fed through a softmax function in order to obtain probabilities. In tasks such as text generation, these probabilities would represent the likelihood of a particular token or word appearing next in the generated output sequence.

This is it for transformers, I hope this article served you well. Please don't hesitate to ask any questions.

Transformers | Machine Learning | Attention | NLP | Data Science

Written by Alejandro Ito Aramendia

Follow

54 Followers

Learning and writing.

More from Alejandro Ito Aramendia

Alejandro Ito Aramendia

Convolutional Neural Networks (CNNs) : A Complete Guide

Everything You Need to Know

8 min read · Jan 1, 2024

121

Alejandro Ito Aramendia

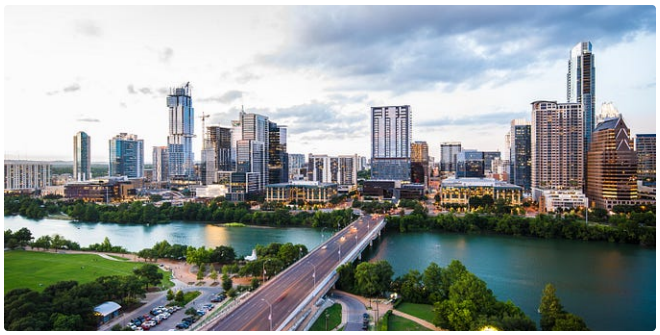
Deformable Convolutional Networks (DCNs) : A Complete...

Everything You Need to Know

8 min read · Feb 1, 2024

127 2

$cg(n)$



Alejandro Ito Aramendia

A Guide to Understanding Big-O, Big-Ω and Big-Θ Notation

Time Complexity is critical when it comes to programming. It is a key decider on whether...

5 min read · Oct 1, 2023

60 1

Alejandro Ito Aramendia

A Guide to Dijkstra's Algorithm | All You Need

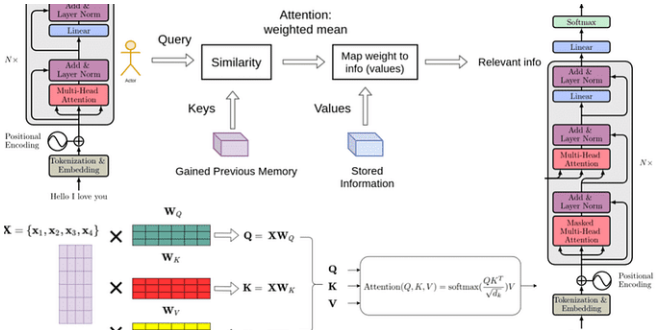
Picture this, you are on holiday in a foreign country and you are lost. The area is...

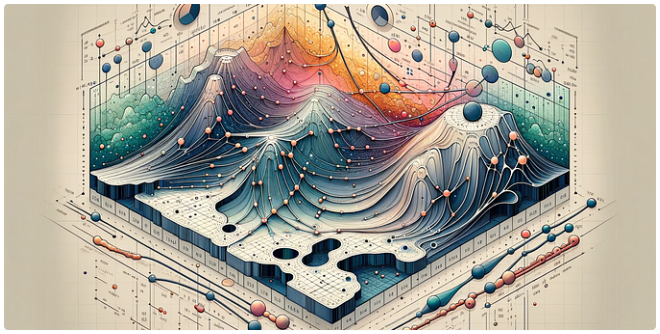
8 min read · Oct 31, 2023

52

See all from Alejandro Ito Aramendia

Recommended from Medium





Amanatullah

Transformer Architecture explained

Transformers are a new development in machine learning that have been making a lo...

10 min read · Sep 1, 2023

308 2

Cristian Leo in Towards Data Science

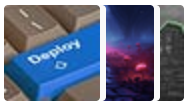
The Math behind Adam Optimizer

Why is Adam the most popular optimizer in Deep Learning? Let's understand it by diving...

16 min read · Jan 31, 2024

1.6K 9

Lists



Predictive Modeling w/ Python

20 stories · 902 saves

Practical Guides to Machine Learning

10 stories · 1057 saves

Natural Language Processing

1196 stories · 667 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 307 saves

Suman Das

Fine Tune Large Language Model (LLM) on a Custom Dataset with...

The field of natural language processing has been revolutionized by large language...

15 min read · Jan 25, 2024

Austin Star... in Artificial Intelligence in Plain Engli...

Reinforcement Learning is Dead. Long Live the Transformer!

Large Language Models are more powerful than you imagine

8 min read · Jan 14, 2024

525 6 1K 32

Hrithick Sen

The Math Behind the Machine: A Deep Dive into the Transformer...

The transformer architecture was introduced in the paper “Attention is All You Need” by...

12 min read · Jan 16, 2024

87 5

Fareed Khan in Level Up Coding

Solving Transformer by Hand: A Step-by-Step Math Example

Performing numerous matrix multiplications to solve the encoder and decoder parts of th...

13 min read · Dec 18, 2023

1.7K 25

See more recommendations