Illustrated Guide to Transformers-Step by Step Explanation

Michael Phi · Follow

Published in Towards Data Science · 12 min read · May 1, 2020

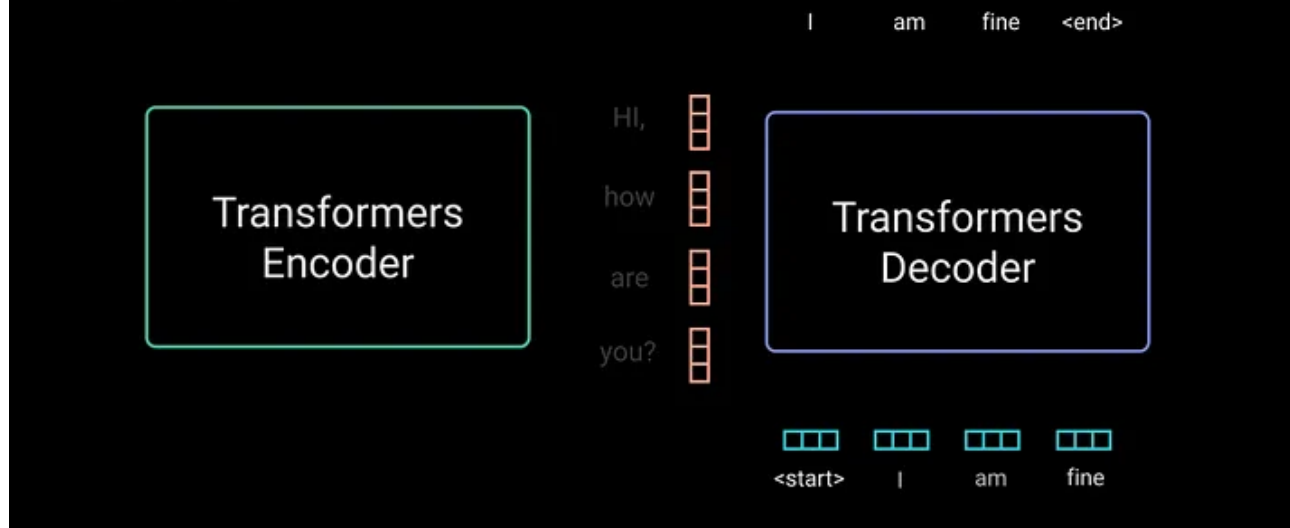Transformers are taking the natural language processing world by storm. These incredible models are breaking multiple NLP records and pushing the state of the art. They are used in many applications like machine language translation, conversational chatbots, and even to power better search engines. Transformers are the rage in deep learning nowadays, but how do they work? Why have they outperform the previous king of sequence problems, like recurrent neural networks, GRU's, and LSTM's? You've probably heard of different famous transformers models like BERT, GPT, and GPT2. In this post, we'll focus on the one paper that started it all, "Attention is all you need".

Check out the link below if you'd like to watch the video version instead.

Illustrated Guide to Transformers Neural Network: A step by step explanati...

Attention Mechanism

To understand transformers we first must understand the attention mechanism. The Attention mechanism enables the transformers to have extremely long term memory. A transformer model can "attend" or "focus" on all previous tokens that have been generated.

Let's walk through an example. Say we want to write a short sci-fi novel with a generative transformer. Using Hugging Face's Write With Transformer application, we can do just that. We'll prime the model with our input, and the model will generate the rest.
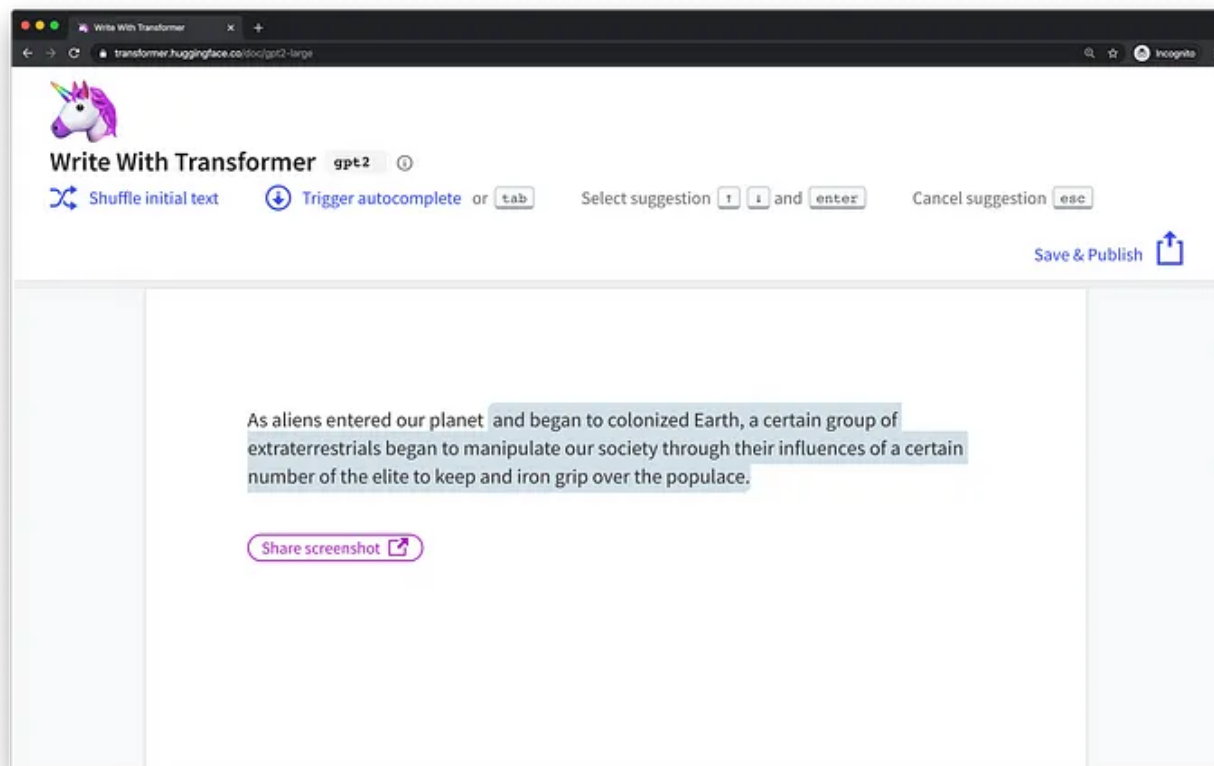
**Our input:** *"As Aliens entered our planet".*

Transformer output: *"and began to colonized Earth, a certain group of extraterrestrials began to manipulate our society through their influences of a certain number of the elite to keep and iron grip over the populace."*

Ok, so the story is a little dark but what's interesting is how the model generated it. As the model generates the text word by word, it can "attend" or "focus" on words that are relevant to the generated word. The ability to know

what words to attend too is all learned during training through backpropagation.

As aliens entered our planet
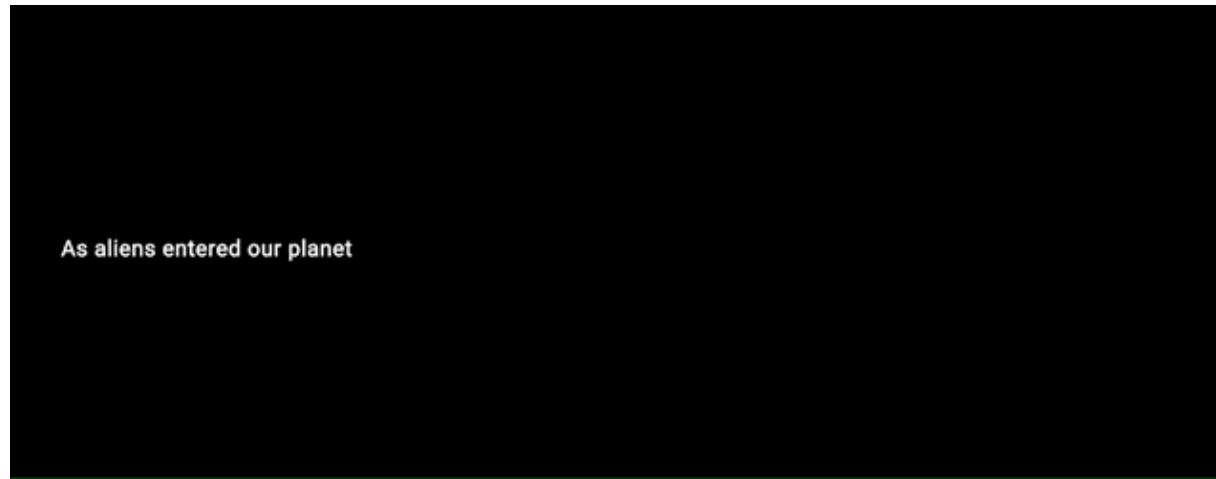
Attention mechanism focusing on different tokens while generating words 1 by 1

Recurrent neural networks (RNN) are also capable of looking at previous inputs too. But the power of the attention mechanism is that it doesn't suffer from short term memory. RNN's have a shorter window to reference from, so when the story gets longer, RNN's can't access words generated earlier in the sequence. This is still true for Gated Recurrent Units (GRU's) and Long-short Term Memory (LSTM's) networks, although they do a bigger capacity to achieve longer-term memory, therefore, having a longer window to reference from. The attention mechanism, in theory, and given enough compute resources, have an infinite window to reference from, therefore being capable of using the entire context of the story while generating the text.
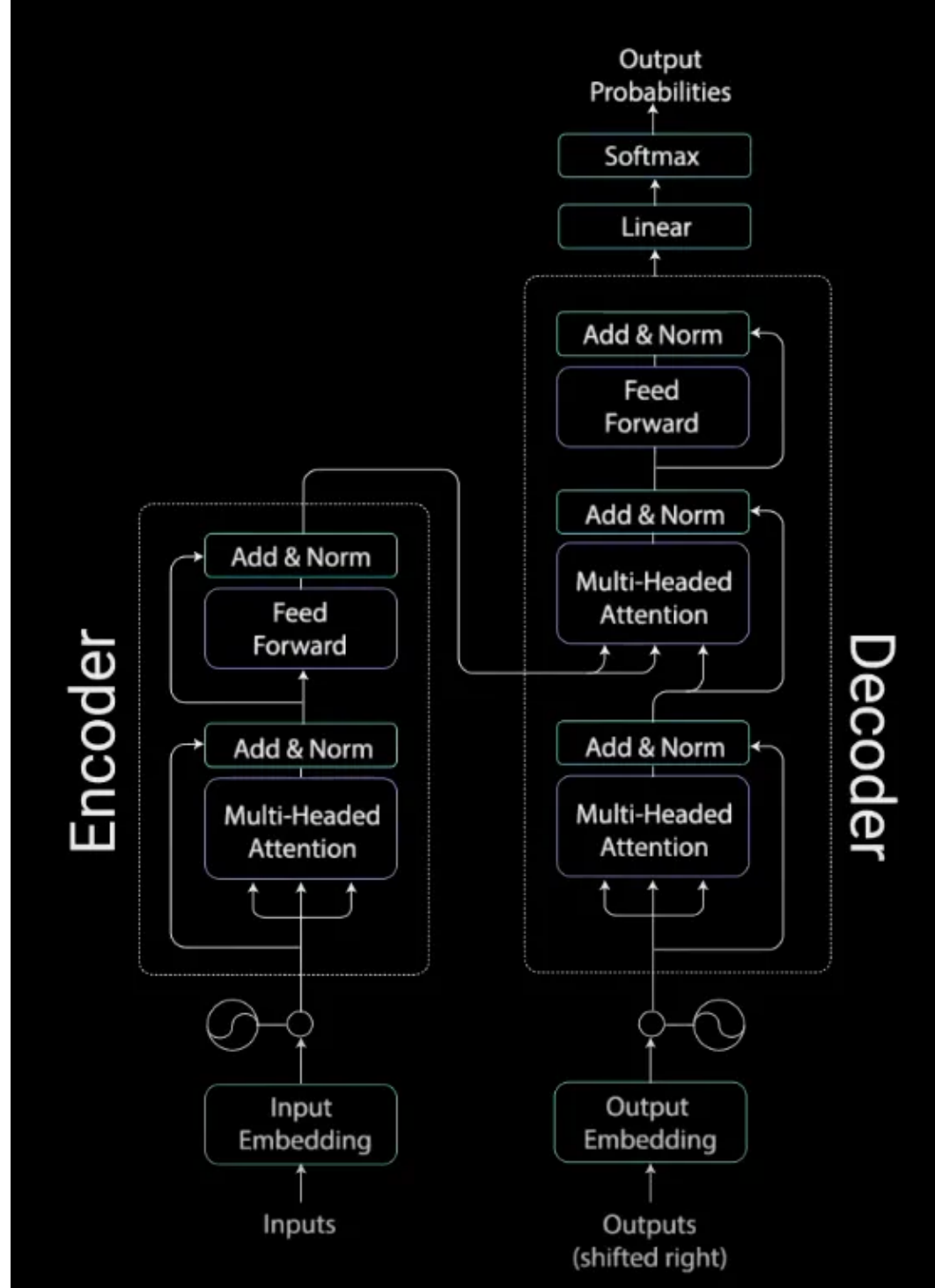
Hypothetical reference window of Attention, RNN's, GRU's & LSTM's

Attention Is All You Need — Step by Step Walkthrough

The attention mechanism's power was demonstrated in the paper "Attention Is All You Need", where the authors introduced a new novel neural network called the Transformers which is an attention-based encoder-decoder type architecture.

Transformer Model

On a high level, the encoder maps an input sequence into an abstract continuous representation that holds all the learned information of that input. The decoder then takes that continuous representation and step by step generates a single output while also being fed the previous output.

Let's walk through an example. The paper applied the Transformer model on a neural machine translation problem. In this post, we'll demonstrate how it'll work for a conversational chatbot.

Our Input: *"Hi how are you"*

Transformer Output: *"I am fine"*

Input Embeddings

The first step is feeding out input into a word embedding layer. A word embedding layer can be thought of as a lookup table to grab a learned vector representation of each word. Neural networks learn through numbers so each word maps to a vector with continuous values to represent that word.
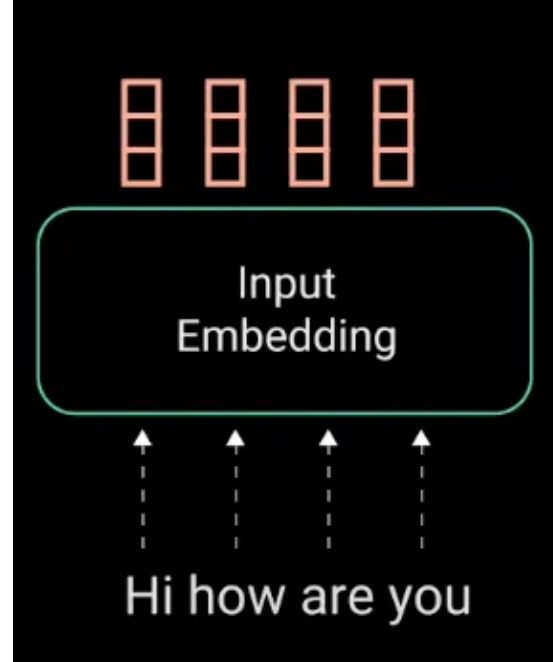
converting Words to Input Embeddings

Positional Encoding

The next step is to inject positional information into the embeddings. Because the transformer encoder has no recurrence like recurrent neural networks, we must add some information about the positions into the input embeddings. This is done using positional encoding. The authors came up with a clever trick using sin and cosine functions.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

We won't go into the mathematical details of positional encoding, but here are the basics. For every odd index on the input vector, create a vector using the cos function. For every even index, create a vector using the sin function. Then add those vectors to their corresponding input embeddings. This successfully gives the network information on the position of each vector. The sin and cosine functions were chosen in tandem because they have linear properties the model can easily learn to attend to.

Encoder Layer

Now we have the encoder layer. The Encoders layers job is to map all input sequences into an abstract continuous representation that holds the learned information for that entire sequence. It contains 2 sub-modules, multi-headed attention, followed by a fully connected network. There are also residual connections around each of the two sublayers followed by a layer normalization.
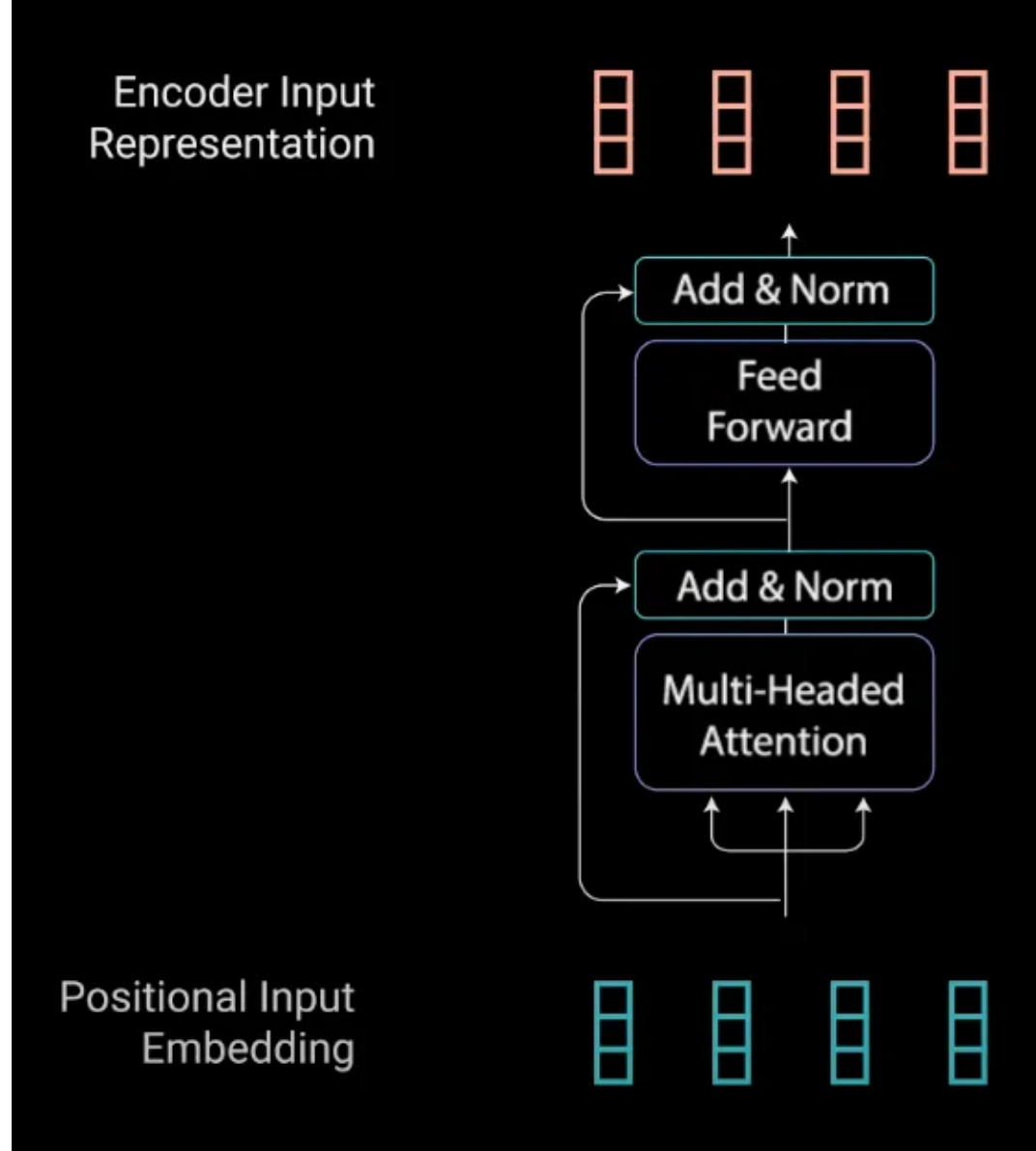
Encoder Layer Sub Modules

To break this down, let's first look at the multi-headed attention module.

Multi-Headed Attention

Multi-headed attention in the encoder applies a specific attention mechanism called self-attention. Self-attention allows the models to associate each word in the input, to other words. So in our example, it's possible that our model can learn to associate the word "you", with "how" and "are". It's also possible that the model learns that words structured in this pattern are typically a question so respond appropriately.
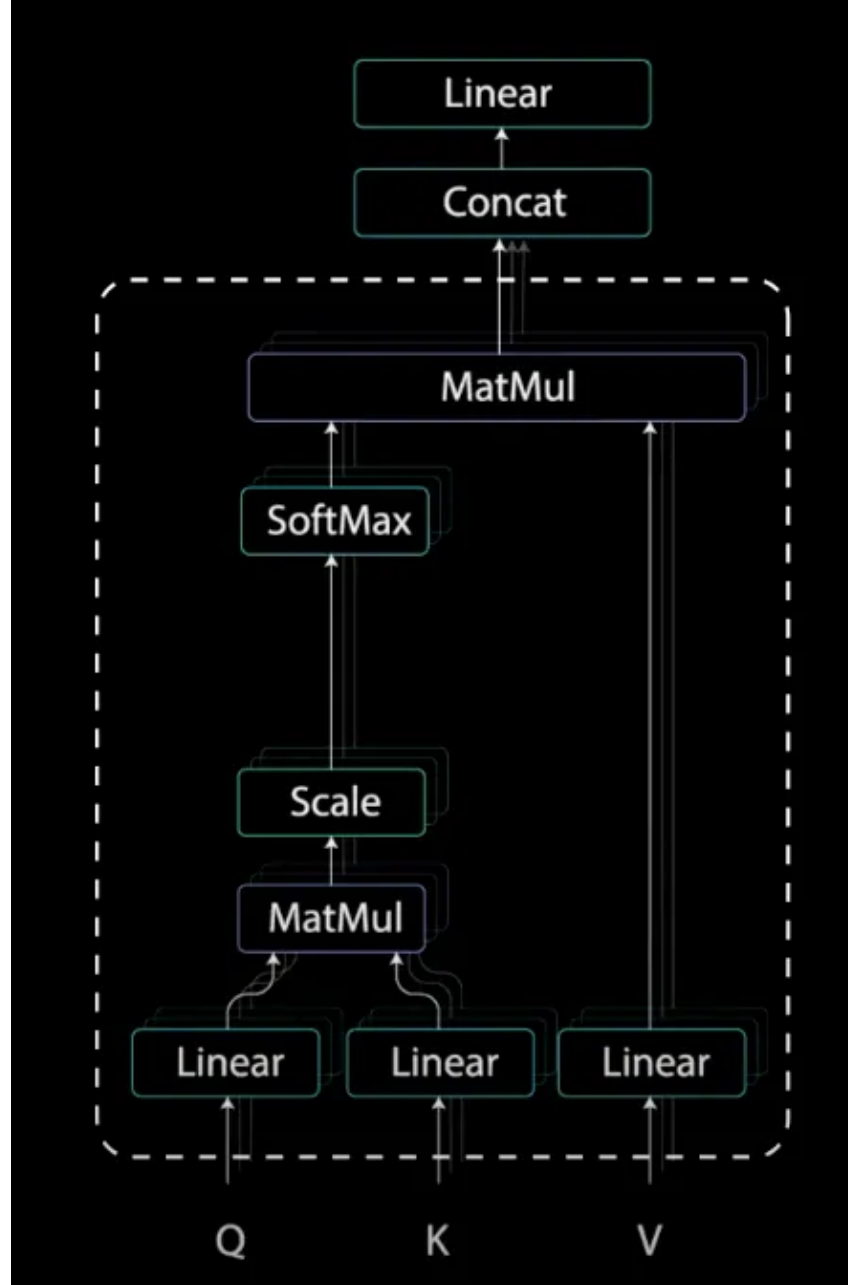
Encoder Self-Attention Operations. Reference this when looking at Illustrations below.

Query, Key, and Value Vectors

To achieve self-attention, we feed the input into 3 distinct fully connected layers to create the query, key, and value vectors.

What are these vectors exactly? I found a good underline explanation on stack exchange stating....

> "The query key and value concept come from retrieval systems. For example, when you type a query to search for some video on Youtube, the search engine will map your **query** against a set of **keys** (video title, description etc.) associated with candidate videos in the database, then present you the best matched videos (**values**).

Dot Product of Query and Key

After feeding the query, key, and value vector through a linear layer, the queries and keys undergo a dot product matrix multiplication to produce a score matrix.
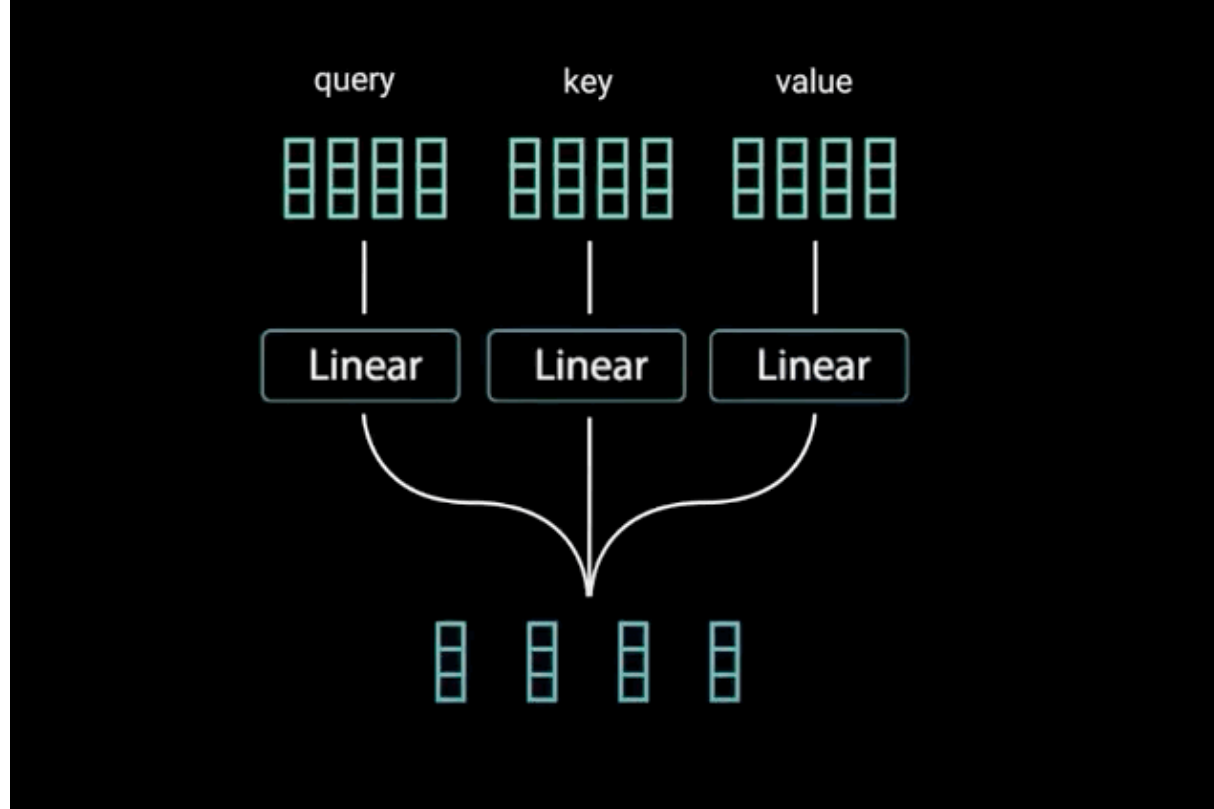
Dot Product multiplication of the query and the key

The score matrix determines how much focus should a word be put on other words. So each word will have a score that corresponds to other words in the time-step. The higher the score the more focus. This is how the queries are mapped to the keys.

Attention scores from the dot product.

Scaling Down the Attention Scores

Then, the scores get scaled down by getting divided by the square root of the dimension of query and key. This is to allow for more stable gradients, as multiplying values can have exploding effects.
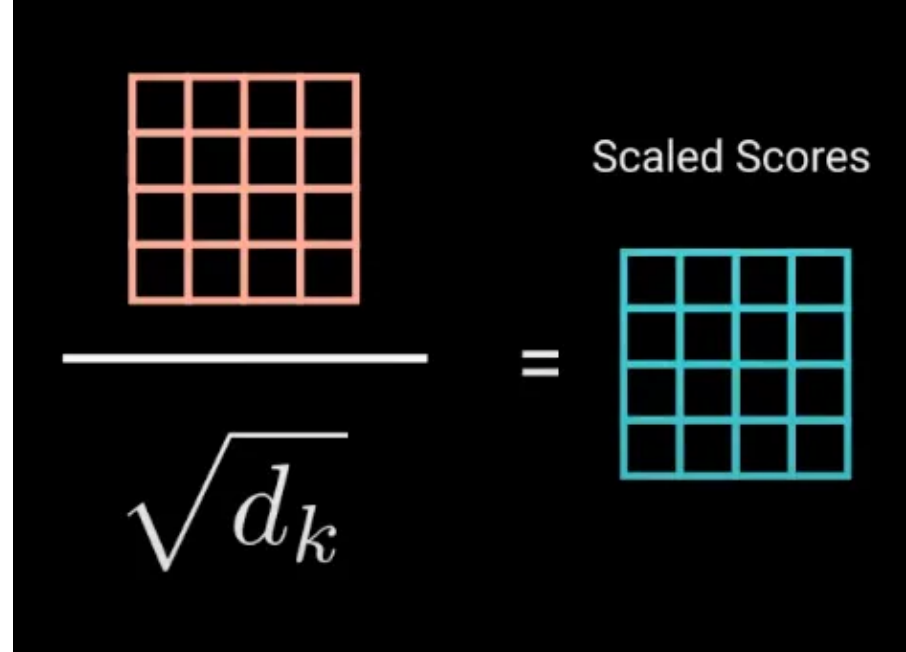
Scaling down the Attention scores

Softmax of the Scaled Scores

Next, you take the softmax of the scaled score to get the attention weights, which gives you probability values between 0 and 1. By doing a softmax the higher scores get heighten, and lower scores are depressed. This allows the model to be more confident about which words to attend too.

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$$

Taking the softmax of the scaled scores to get probability values

Multiply Softmax Output with Value vector

Then you take the attention weights and multiply it by your value vector to get an output vector. The higher softmax scores will keep the value of words the model learns is more important. The lower scores will drown out the irrelevant words. Then you feed the output of that into a linear layer to process.
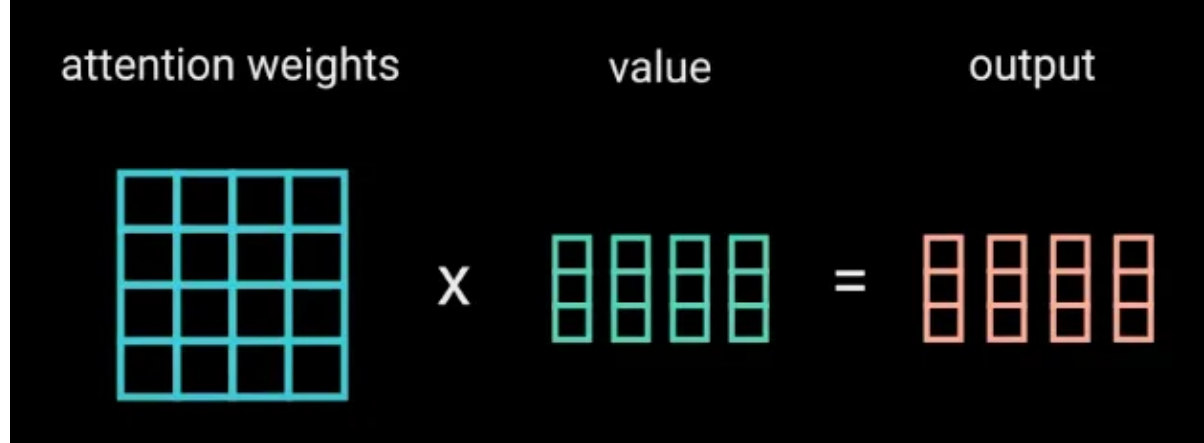
Computing Multi-headed Attention

To make this a multi-headed attention computation, you need to split the query, key, and value into N vectors before applying self-attention. The split vectors then go through the self-attention process individually. Each self-attention process is called a head. Each head produces an output vector that gets concatenated into a single vector before going through the final linear layer. In theory, each head would learn something different therefore giving the encoder model more representation power.
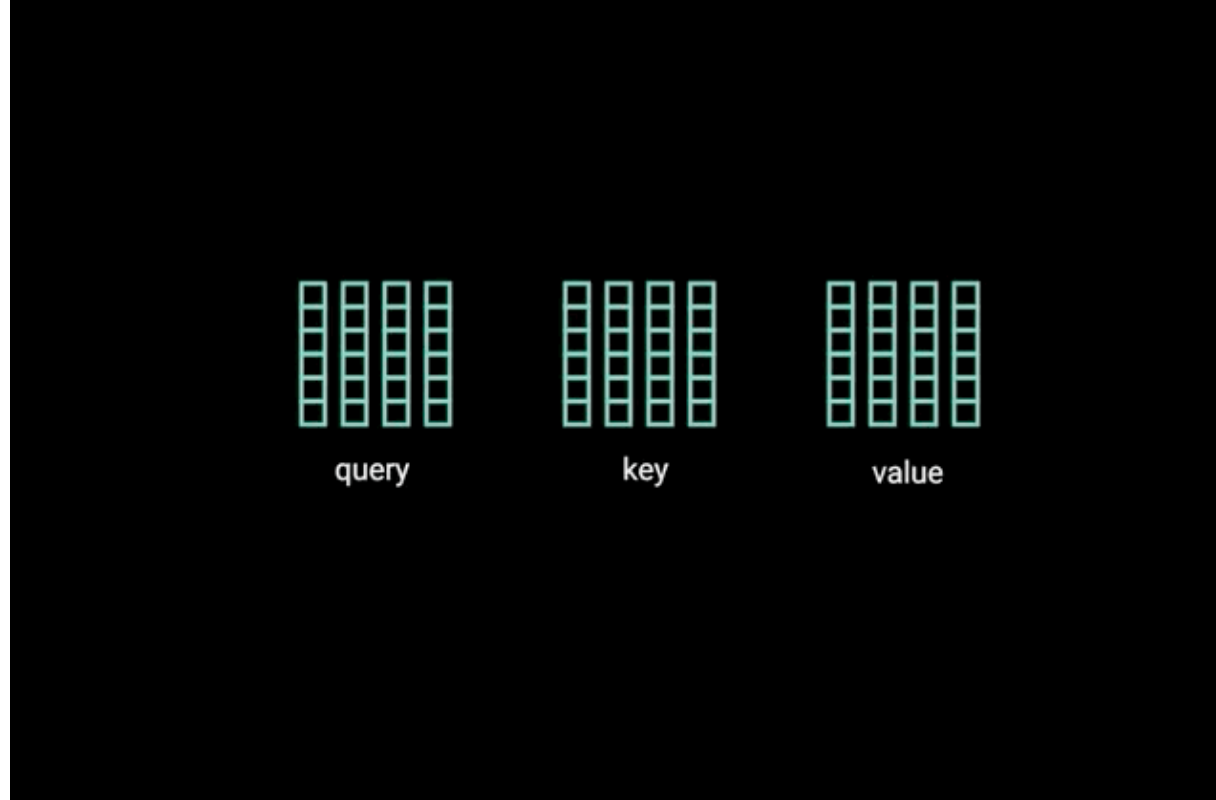
Splitting Q, K, V, N times before applying self-attention

To sum it up, multi-headed attention is a module in the transformer network that computes the attention weights for the input and produces an output vector with encoded information on how each word should attend to all other words in the sequence.
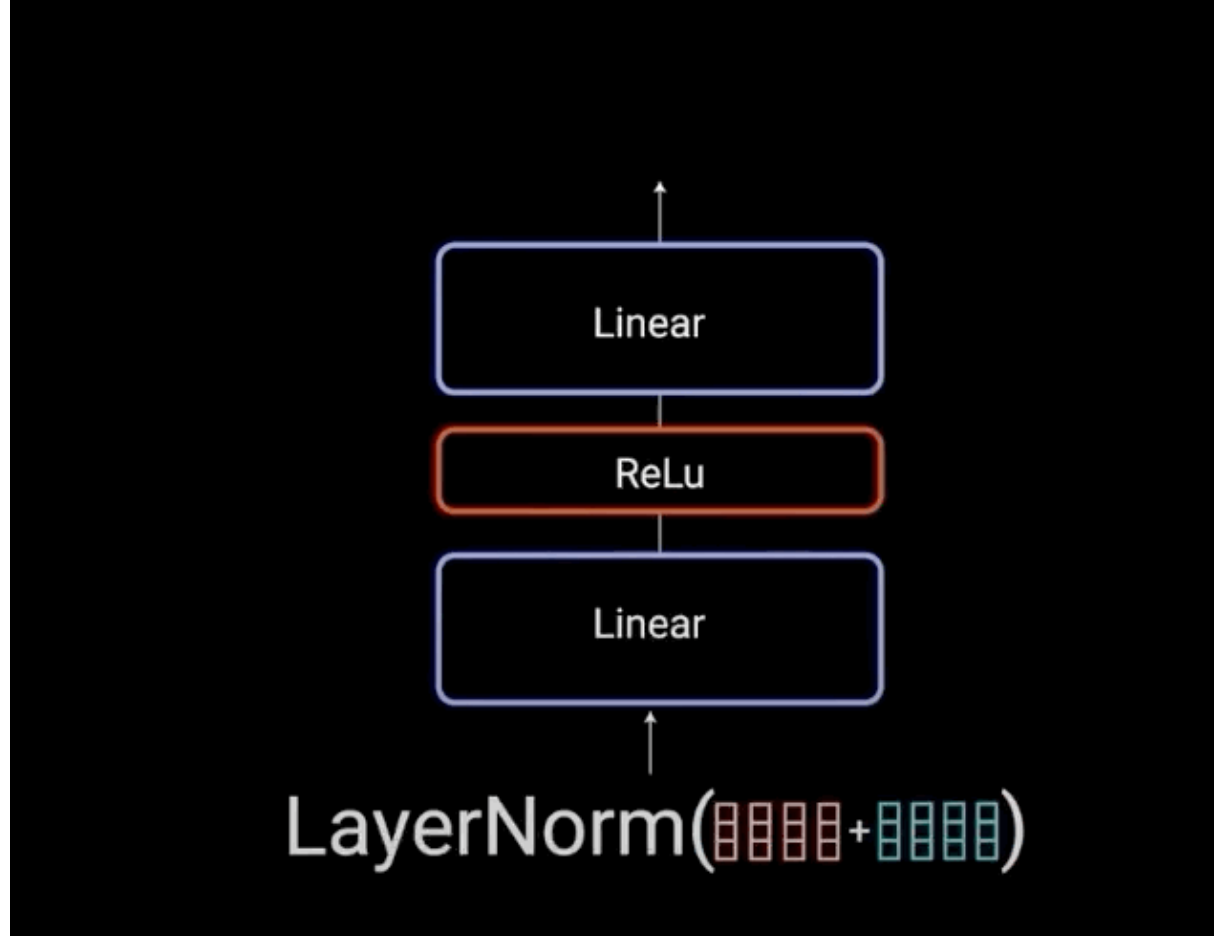
The Residual Connections, Layer Normalization, and Feed Forward Network

The multi-headed attention output vector is added to the original positional input embedding. This is called a residual connection. The output of the residual connection goes through a layer normalization.
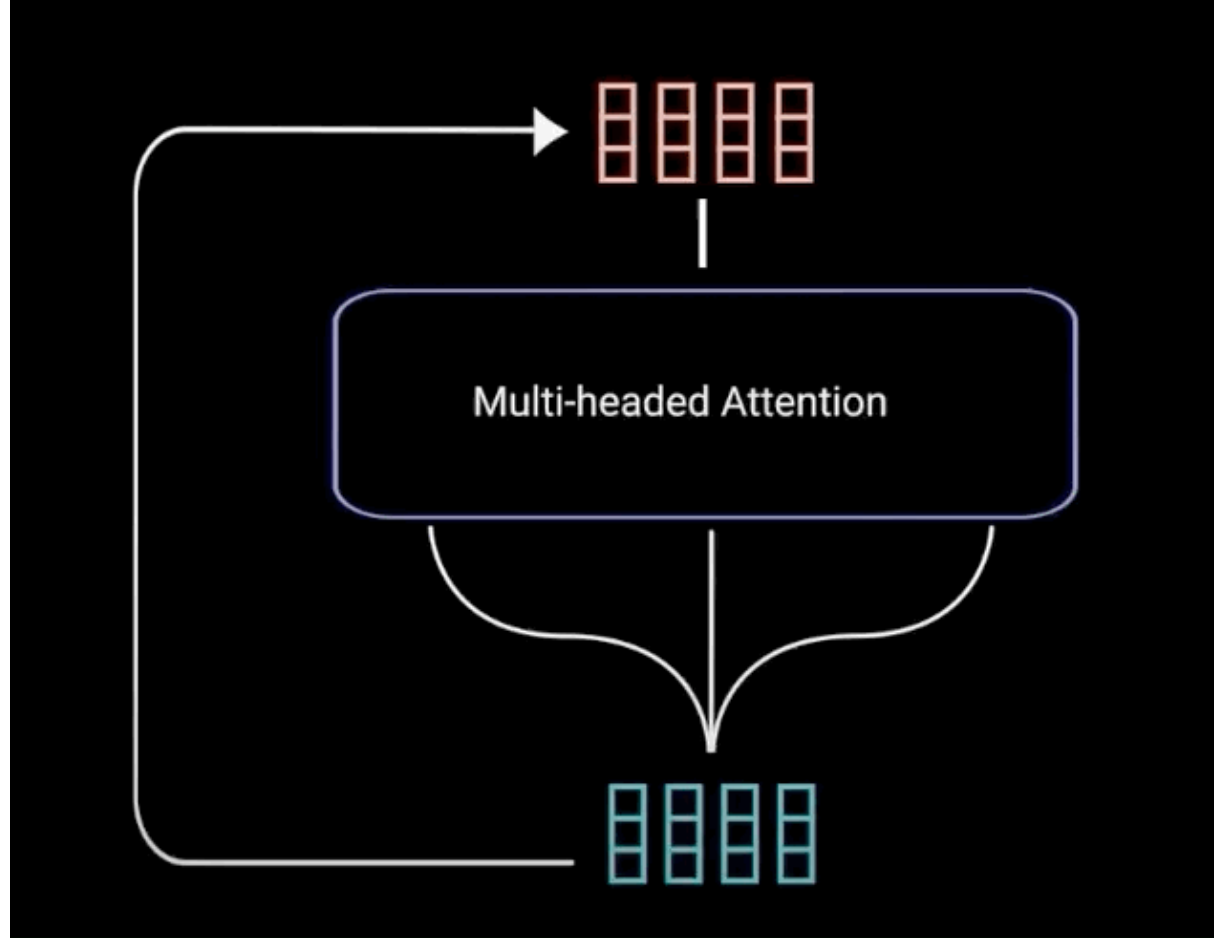
Residual connection of the positional input embedding and the output of Multi-headed Attention

The normalized residual output gets projected through a pointwise feed-forward network for further processing. The pointwise feed-forward network is a couple of linear layers with a ReLU activation in between. The output of that is then again added to the input of the pointwise feed-forward network and further normalized.

Residual connection of the input and output of the point-wise feedforward layer.

The residual connections help the network train, by allowing gradients to flow through the networks directly. The layer normalizations are used to stabilize the network which results in substantially reducing the training time necessary. The pointwise feedforward layer is used to project the attention outputs potentially giving it a richer representation.

Encoder Wrap-up

That wraps up the encoder layer. All of these operations are to encode the input to a continuous representation with attention information. This will help the decoder focus on the appropriate words in the input during the decoding process. You can stack the encoder N times to further encode the information, where each layer has the opportunity to learn different attention representations therefore potentially boosting the predictive power of the transformer network.
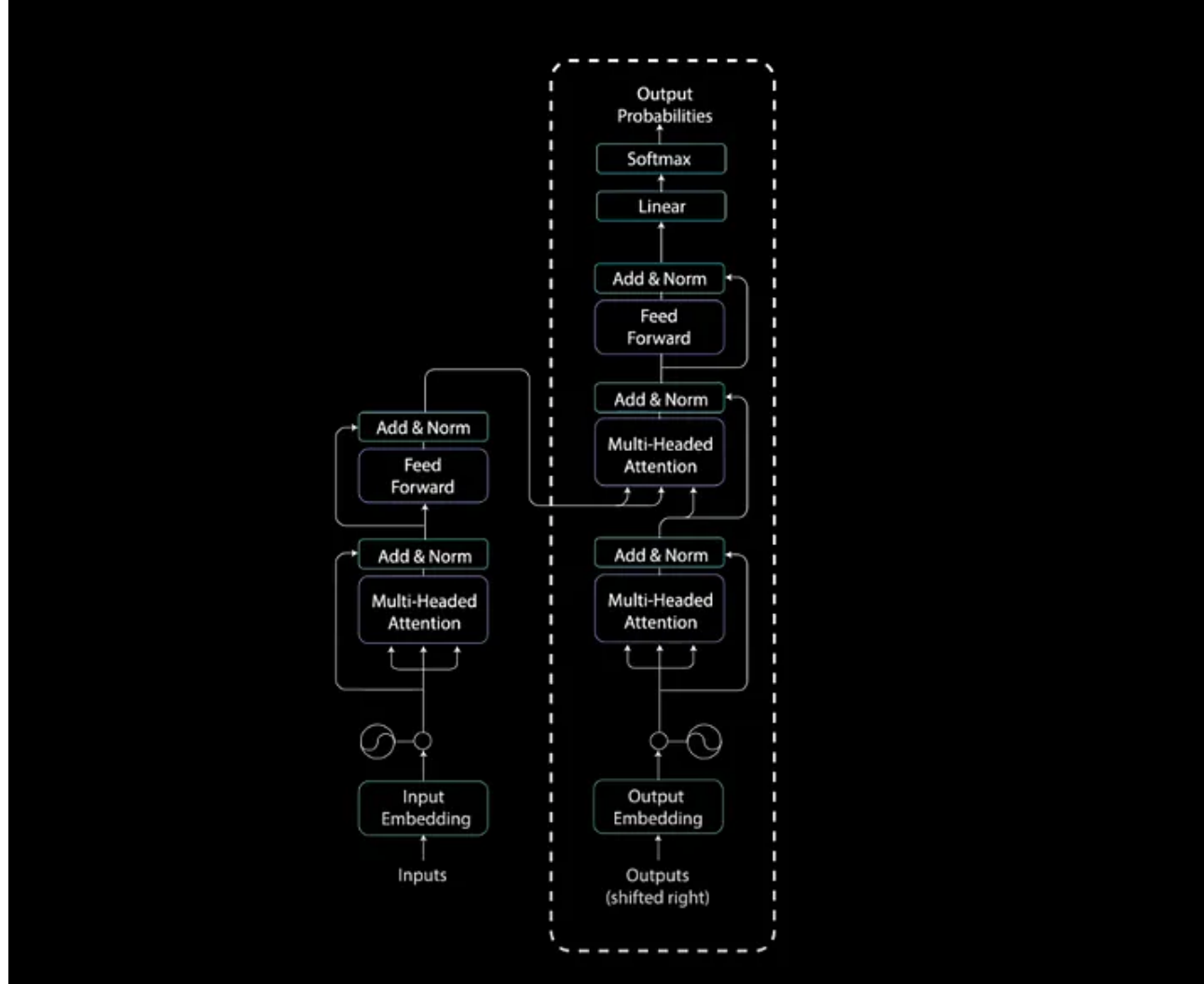
Decoder Layer

The decoder's job is to generate text sequences. The decoder has a similar sub-layer as the encoder. it has two multi-headed attention layers, a pointwise feed-forward layer, and residual connections, and layer normalization after each sub-layer. These sub-layers behave similarly to the layers in the encoder but each multi-headed attention layer has a different job. The decoder is capped off with a linear layer that acts as a classifier, and a softmax to get the word probabilities.
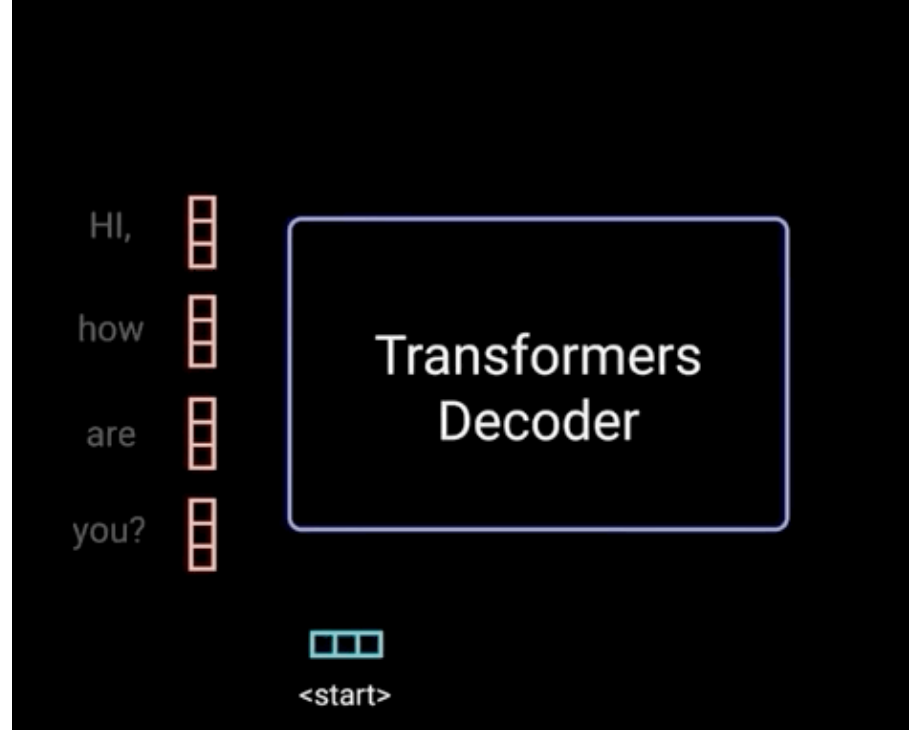
Decoder Layer. Reference This diagram while reading.

The decoder is autoregressive, it begins with a start token, and it takes in a list of previous outputs as inputs, as well as the encoder outputs that contain the attention information from the input. The decoder stops decoding when it generates a token as an output.

The decoder is autoregressive as it generates a token 1 at a time while being fed in the previous outputs.

Let's walk through the decoding steps.

Decoder Input Embeddings & Positional Encoding

The beginning of the decoder is pretty much the same as the encoder. The input goes through an embedding layer and positional encoding layer to get positional embeddings. The positional embeddings get fed into the first multi-head attention layer which computes the attention scores for the decoder's input.

Decoders First Multi-Headed Attention

This multi-headed attention layer operates slightly differently. Since the decoder is autoregressive and generates the sequence word by word, you need to prevent it from conditioning to future tokens. For example, when computing attention scores on the word "am", you should not have access to the word "fine", because that word is a future word that was generated after. The word "am" should only have access to itself and the words before it. This is true for all other words, where they can only attend to previous words.
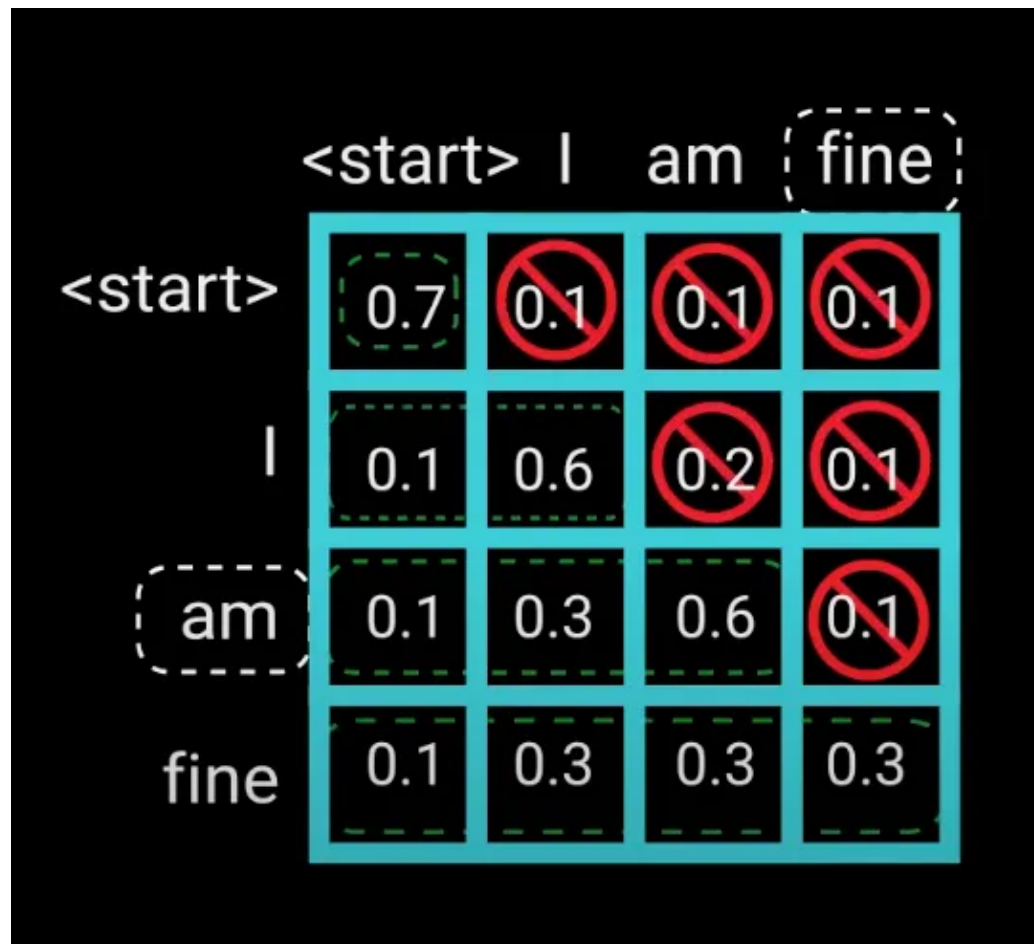

A depiction of Decoder's first Multi-headed Attention scaled attention scores. The word "am", should not any values for the word "fine". This is true for all other words.

We need a method to prevent computing attention scores for future words. This method is called masking. To prevent the decoder from looking at future tokens, you apply a look ahead mask. The mask is added before calculating the softmax, and after scaling the scores. Let's take a look at how this works.

Look-Ahead Mask

The mask is a matrix that's the same size as the attention scores filled with values of 0's and negative infinities. When you add the mask to the scaled attention scores, you get a matrix of the scores, with the top right triangle filled with negativity infinities.
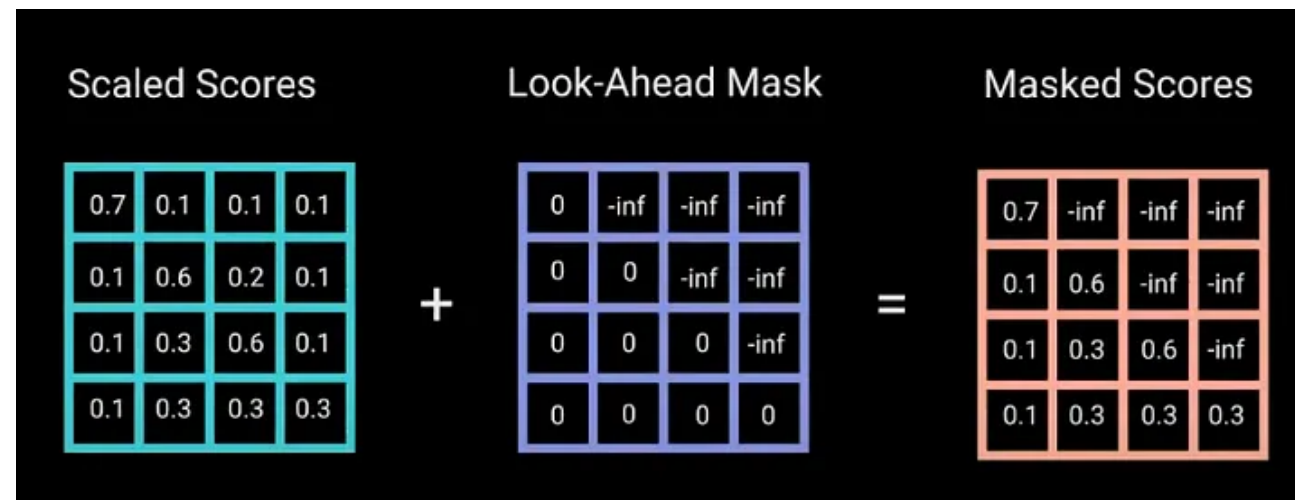

Adding a look-ahead mask to the scaled scores

The reason for the mask is because once you take the softmax of the masked scores, the negative infinities get zeroed out, leaving zero attention scores for future tokens. As you can see in the figure below, the attention scores for

"am", has values for itself and all words before it but is zero for the word "fine". This essentially tells the model to put no focus on those words.
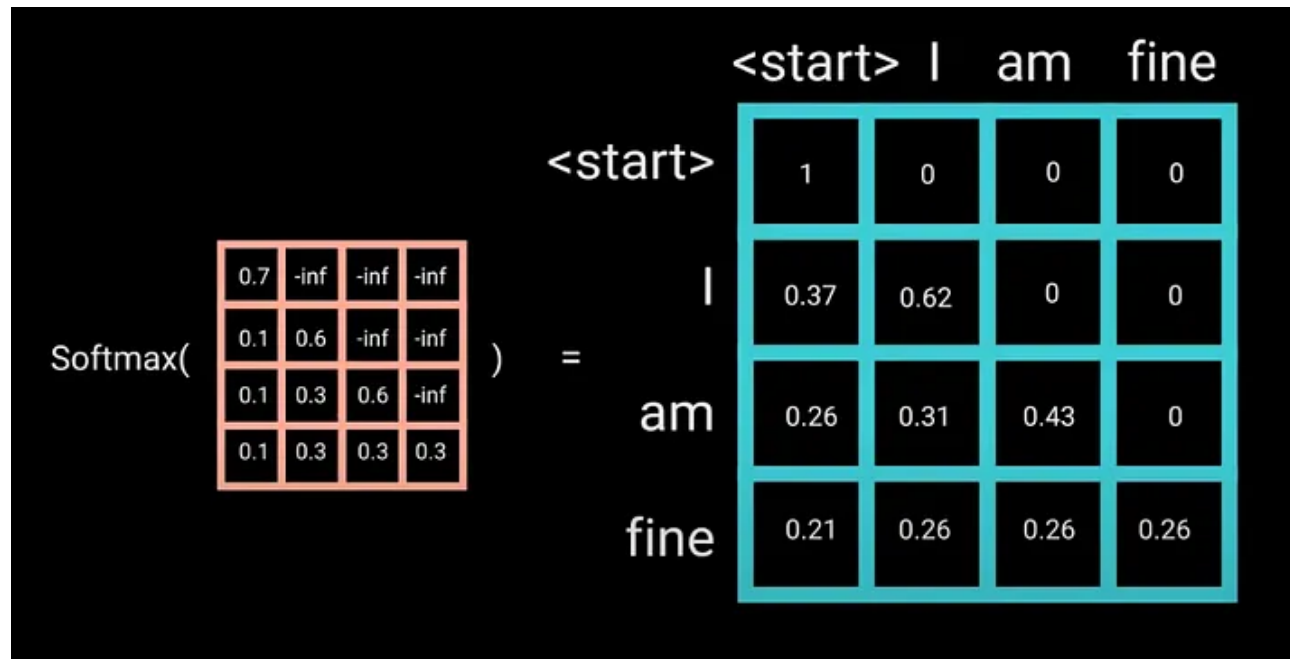
This masking is the only difference in how the attention scores are calculated in the first multi-headed attention layer. This layer still has multiple heads, that the mask is being applied to, before getting concatenated and fed through a linear layer for further processing. The output of the first multi-headed attention is a masked output vector with information on how the model should attend on the decoder's input.
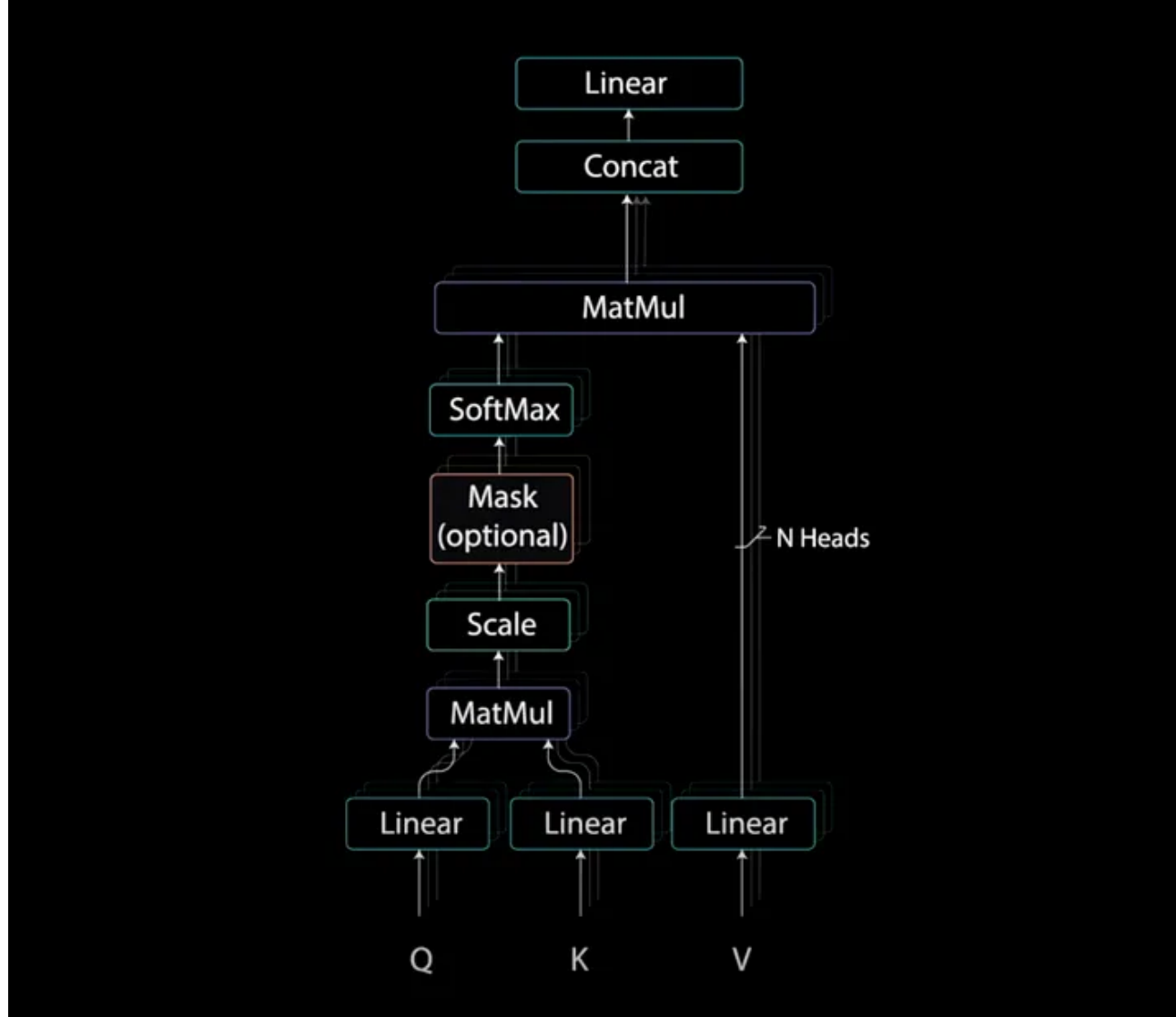
Multi-Headed Attention with Masking

Decoder Second Multi-Headed Attention, and Point-wise Feed Forward Layer

The second multi-headed attention layer. For this layer, the encoder's outputs are the queries and the keys, and the first multi-headed attention

layer outputs are the values. This process matches the encoder's input to the decoder's input, allowing the decoder to decide which encoder input is relevant to put a focus on. The output of the second multi-headed attention goes through a pointwise feedforward layer for further processing.

## Linear Classifier and Final Softmax for Output Probabilities

The output of the final pointwise feedforward layer goes through a final linear layer, that acts as a classifier. The classifier is as big as the number of classes you have. For example, if you have 10,000 classes for 10,000 words, the output of that classier will be of size 10,000. The output of the classifier then gets fed into a softmax layer, which will produce probability scores between 0 and 1. We take the index of the highest probability score, and that equals our predicted word.
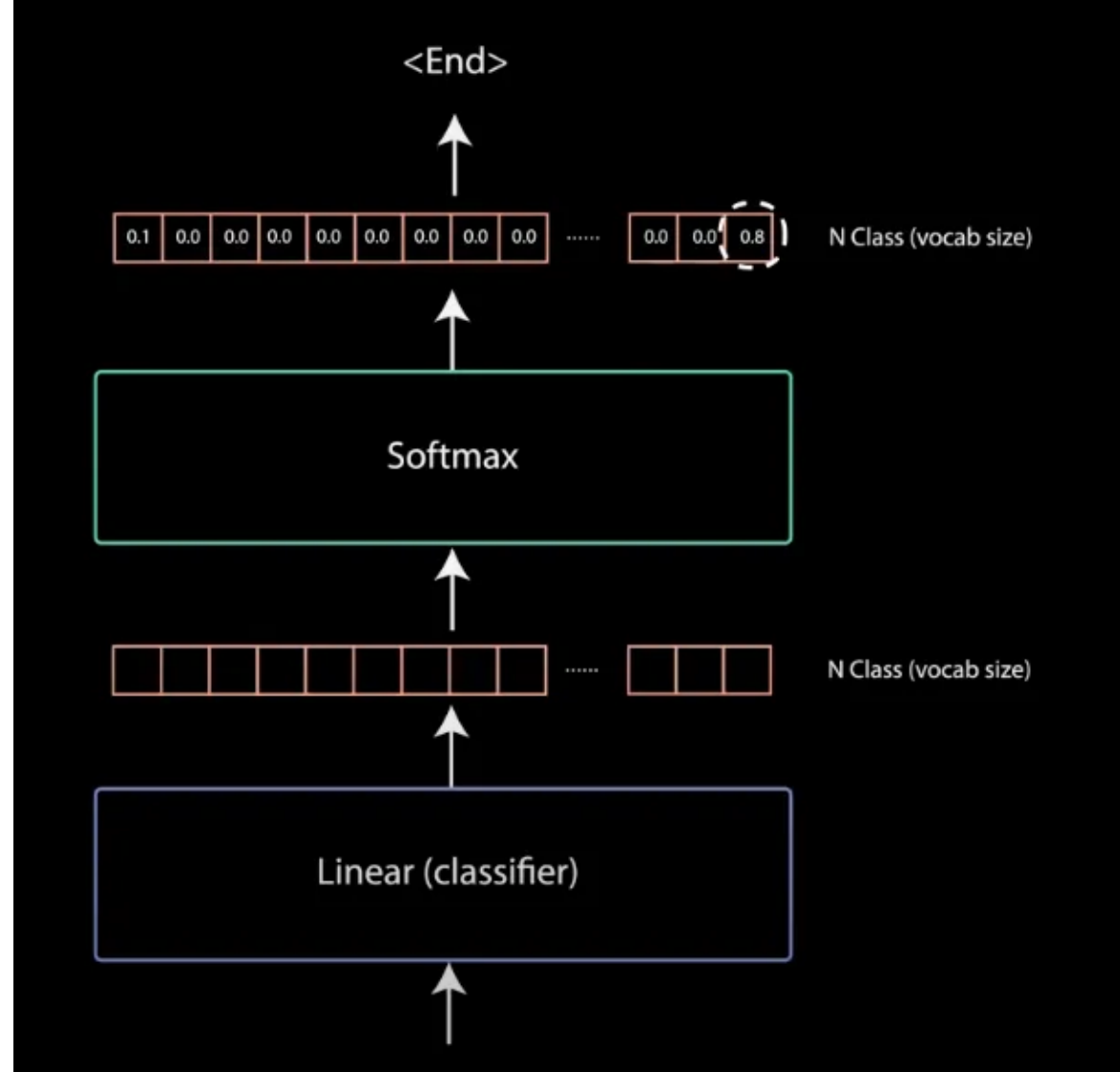
Linear Classifier with Softmax to get the Output Probabilities

The decoder then takes the output, add's it to the list of decoder inputs, and continues decoding again until a token is predicted. For our case, the highest probability prediction is the final class which is assigned to the end token.

The decoder can also be stacked N layers high, each layer taking in inputs from the encoder and the layers before it. By stacking the layers, the model can learn to extract and focus on different combinations of attention from its attention heads, potentially boosting its predictive power.
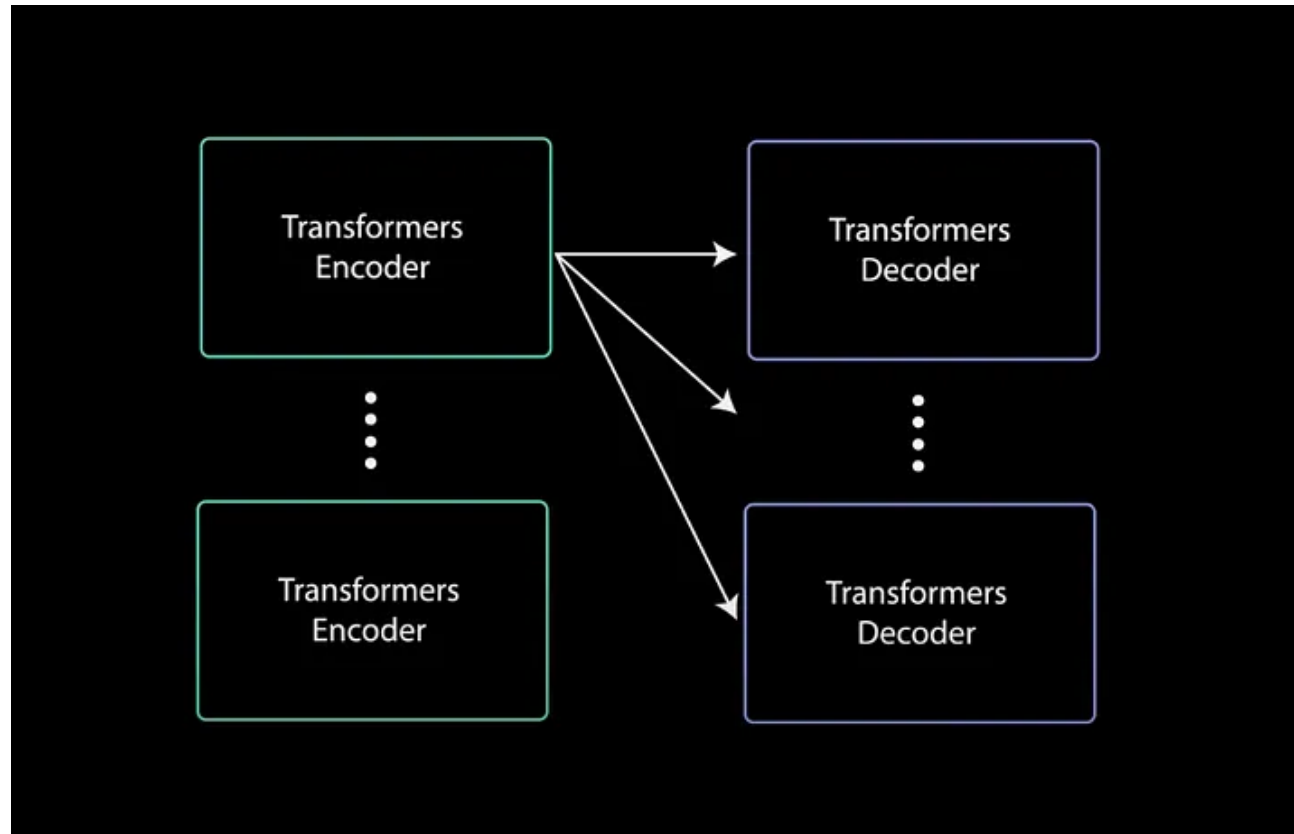
Stacked Encoder and Decoder

And That's It!

And that's it! That's the mechanics of the transformers. Transformers leverage the power of the attention mechanism to make better predictions. Recurrent Neural networks try to achieve similar things, but because they

suffer from short term memory. Transformers can be better especially if you want to encode or generate long sequences. Because of the transformer architecture, the natural language processing industry can achieve unprecedented results.

Check out michaelphi.com for more content like this.

✍🏽 Want more Content? Check out my blog at _https://www.michaelphi.com_

📺 Like to watch project-based videos? Check out my **Youtube**!

🥇 Stay up to date on articles and videos by signing up for my **email newsletter**!

Deep Learning   Machine Learning   Artificial Intelligence   Towards Data Science

Math

Written by Michael Phi

4.6K Followers · Writer for Towards Data Science

Software and Machine Learning Research Engineer. Here to make complex things simple. Follow me to stay on top of A.I. www.michaelphi.com

More from Michael Phi and Towards Data Science
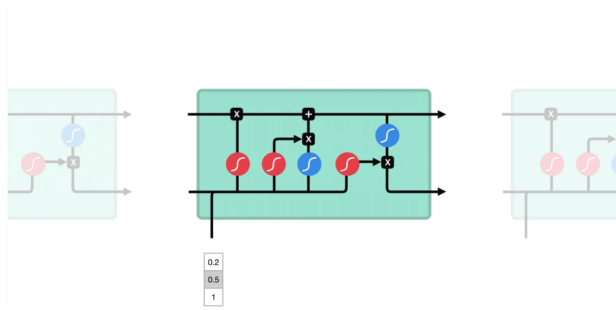
Michael Phi in Towards Data Science

Illustrated Guide to LSTM's and GRU's: A step by step explanation

Hi and welcome to an Illustrated Guide to Long Short-Term Memory (LSTM) and Gated...

10 min read · Sep 24, 2018

Siavash Yasini in Towards Data Science

How to Write Memory-Efficient Classes in Python

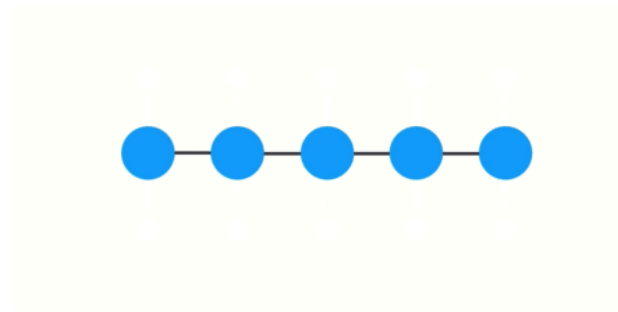Three tricks to prevent your data project from memory overflow

✦ · 7 min read · Jan 13

Natassha Selvaraj in Towards Data Science

ChatGPT for Data Analysis — A Beginner's Guide

An complete tutorial on using ChatGPT for data analysis.

✦ · 12 min read · Dec 23, 2023

Michael Phi in Towards Data Science
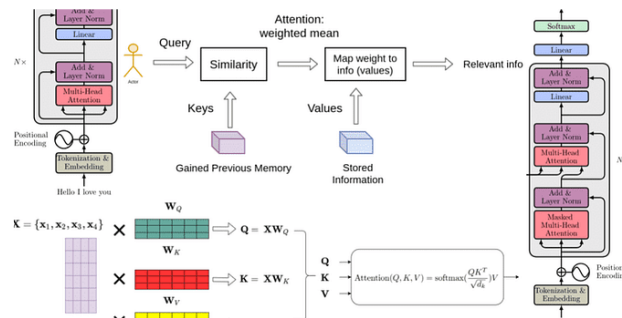
Illustrated Guide to Recurrent Neural Networks

Understanding the Intuition

✦ · 9 min read · Sep 20, 2018

See all from Michael Phi　See all from Towards Data Science
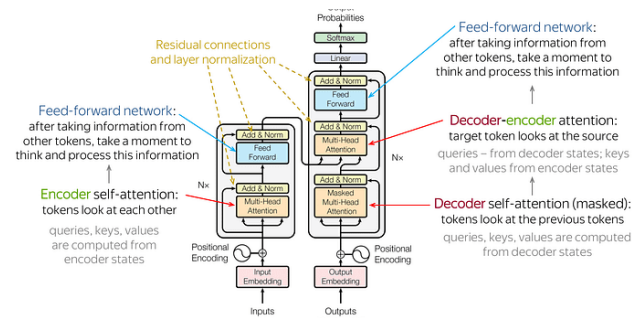
Recommended from Medium

👤 Amanatullah

Transformer Architecture explained

Transformers are a new development in machine learning that have been making a l...
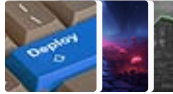
10 min read · Sep 1, 2023

👤 Amirhossein Abaskohi

Navigating Transformers: A Comprehensive Exploration of...
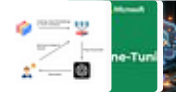
Introduction

9 min read · Aug 16, 2023

Lists

Predictive Modeling w/ Python
20 stories · 828 saves

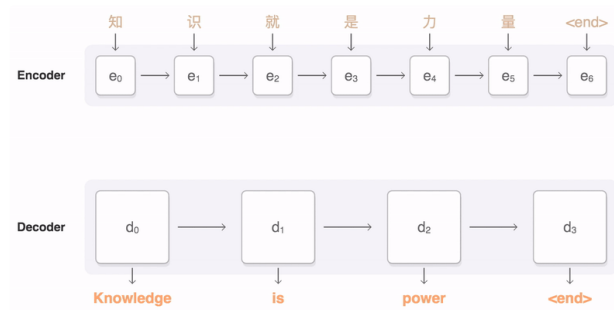**Natural Language Processing**
1118 stories · 591 saves

AI Regulation
6 stories · 287 saves

Practical Guides to Machine Learning
10 stories · 956 saves

Everton Gomede, PhD

Understanding Sequence-to-Sequence (Seq2Seq) Models and...

Introduction

5 min read · Aug 22, 2023

M Rokon

RNN vs. LSTM vs. Transformers: Unraveling the Secrets of...

In the realm of deep learning, sequential data processing is at the heart of many...
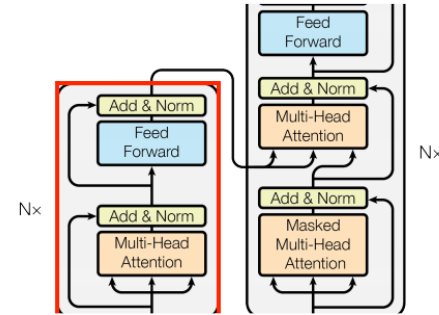
3 min read · Sep 25, 2023

👤 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

👤 Rokas Liuberskis in Python in Plain English

Building a Transformer model with Encoder and Decoder layers in...

In this tutorial, we continue implementing the complete Transformer model in TensorFlow....

✦ · 9 min read · Aug 22, 2023

See more recommendations