PES UNIVERSITY

Topics in Deep Learning

Dr. Shylaja S S

Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)

Department of Computer Science and Engineering

shylaja.sharath@pes.edu

Ack: Devang Saraogi,
Teaching Assistant

Unit 4: Overview of Latest Deep Learning Models

Introduction to GPT

Ack: Devang Saraogi,
Teaching Assistant

Large Language Models

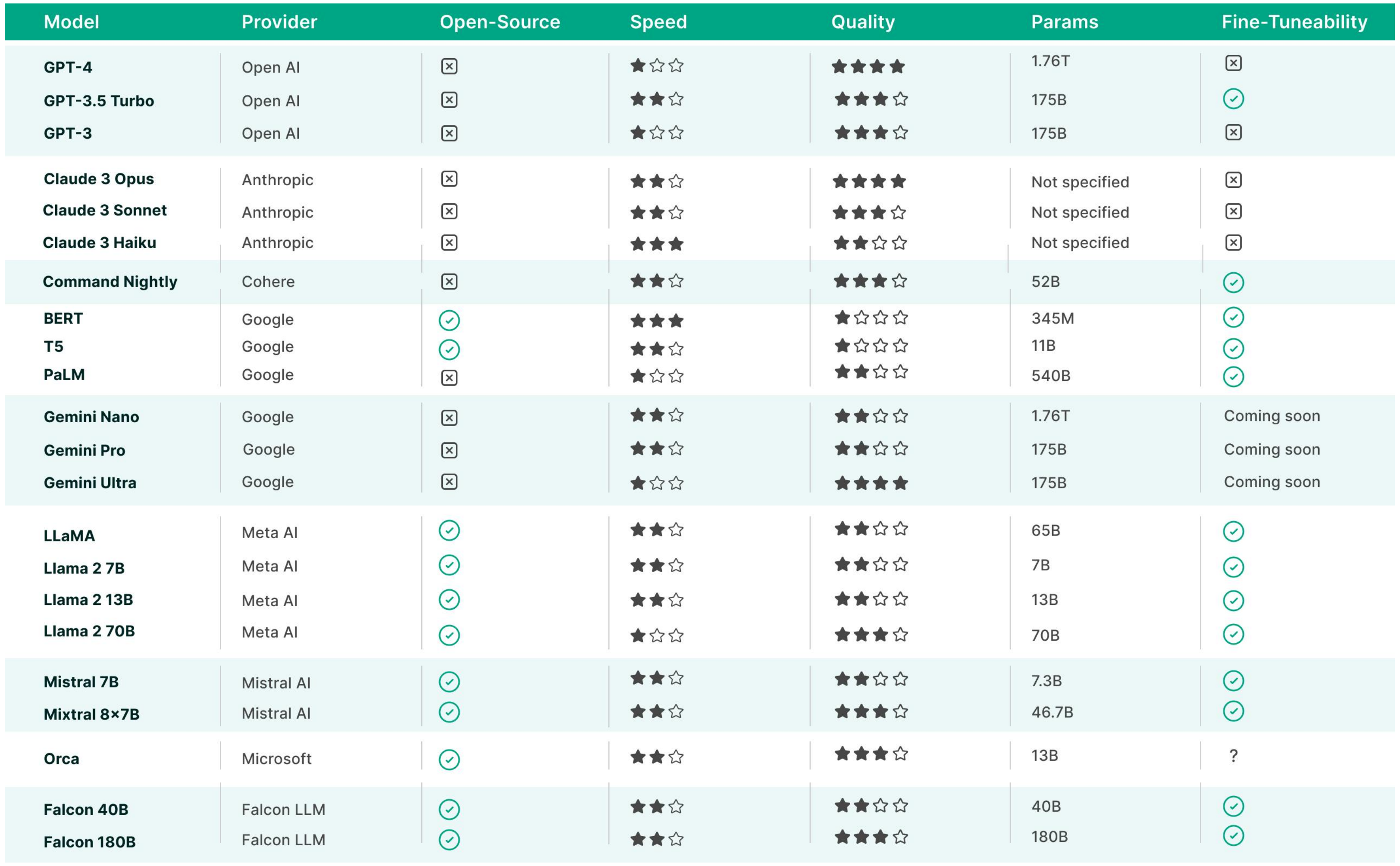
Large Language Models (LLMs)

- Large language models (LLMs) are a category of **foundation** models* trained on immense amounts of data making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks.
- Most popular LLMs, utilize the transformer architecture because of its **abilities to capture long-range dependencies within sequences**, making it well-suited for natural language processing tasks.

Some of the popular LLMs include, GPT-4, GPT-3.5, LLaMA, Gemini, Anthropic's Claude, BERT, Mistral etc. (see next page)

**AI models designed to produce a wide and general variety of outputs. They are capable of a range of tasks and applications, including text, video, image, or audio generation. A singular feature of these models is that they can be standalone systems or used as a 'foundation' for other applications. For example, the LLM called GPT works as the foundation model of ChatGPT.*

Unit 4: Overview of Latest Deep Learning Models – Introduction to GPT



Model	Provider	Open-Source	Speed	Quality	Params	Fine-Tuneability
GPT-4	Open AI	☒	★☆☆	★★★★	1.76T	☒
GPT-3.5 Turbo	Open AI	☒	★★☆	★★★☆	175B	✓
GPT-3	Open AI	☒	★☆☆	★★★☆	175B	☒
Claude 3 Opus	Anthropic	☒	★★☆	★★★★	Not specified	☒
Claude 3 Sonnet	Anthropic	☒	★★☆	★★★☆	Not specified	☒
Claude 3 Haiku	Anthropic	☒	★★★	★★☆☆	Not specified	☒
Command Nightly	Cohere	☒	★★☆	★★★☆	52B	✓
BERT	Google	✓	★★★	★☆☆☆	345M	✓
T5	Google	✓	★★☆	★☆☆☆	11B	✓
PaLM	Google	☒	★☆☆	★★☆☆	540B	✓
Gemini Nano	Google	☒	★★☆	★★☆☆	1.76T	Coming soon
Gemini Pro	Google	☒	★★☆	★★☆☆	175B	Coming soon
Gemini Ultra	Google	☒	★☆☆	★★★★	175B	Coming soon
LLaMA	Meta AI	✓	★★☆	★★☆☆	65B	✓
Llama 2 7B	Meta AI	✓	★★☆	★★☆☆	7B	✓
Llama 2 13B	Meta AI	✓	★★☆	★★☆☆	13B	✓
Llama 2 70B	Meta AI	✓	★☆☆	★★★☆	70B	✓
Mistral 7B	Mistral AI	✓	★★☆	★★☆☆	7.3B	✓
Mixtral 8×7B	Mistral AI	✓	★★☆	★★★☆	46.7B	✓
Orca	Microsoft	✓	★★☆	★★★☆	13B	?
Falcon 40B	Falcon LLM	✓	★★☆	★★☆☆	40B	✓
Falcon 180B	Falcon LLM	✓	★★☆	★★★☆	180B	✓

Transformers as Language Models

Transformers as Language Models

- All the transformer models mentioned earlier have been trained as language models. This means they have been trained on large amounts of raw text in a **self-supervised*** fashion.
- This type of model **develops a statistical understanding of the language** it has been trained on, but it's not very useful for specific practical tasks.
- Because of this, the general pretrained model goes through transfer learning. During this process, **the model is fine-tuned in a supervised way** — that is, using human-annotated labels — on a given task.
- These tasks include predicting the next word in a sentence or predicting a masked word in a sentence.

**Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model. That means that humans are not needed to label the data!*

In the seminal paper Attention is All You Need, the revolutionary Transformer model was introduced.

- A model that uses attention to boost the speed with which these models can be trained.
- The Transformer outperforms the Google Neural Machine Translation model in specific tasks.
- The biggest benefit, however, comes from how The Transformer lends itself to parallelization

Attention Is All You Need

Ashish Vaswani[*] Google Brain avaswani@google.com

Noam Shazeer[*] Google Brain noam@google.com

Niki Parmar[*] Google Research nikip@google.com

Jakob Uszkoreit[*] Google Research usz@google.com

Llion Jones[*] Google Research llion@google.com

Aidan N. Gomez[*] [†] University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser[*] Google Brain lukaszkaiser@google.com

Illia Polosukhin[*] [‡] illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

1 Introduction

Recurrent neural networks, long short-term memory [12] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation [29, 2, 5]. Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures [31, 21, 13].

[*]Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, and efficient inference and visualizations. Lukasz and Aidan spent countless long days designing various parts of and implementing tensor2tensor, replacing our earlier codebase, greatly improving results and massively accelerating our research.

[†]Work performed while at Google Brain.

[‡]Work performed while at Google Research.

Unit 4: Overview of Latest Deep Learning Models

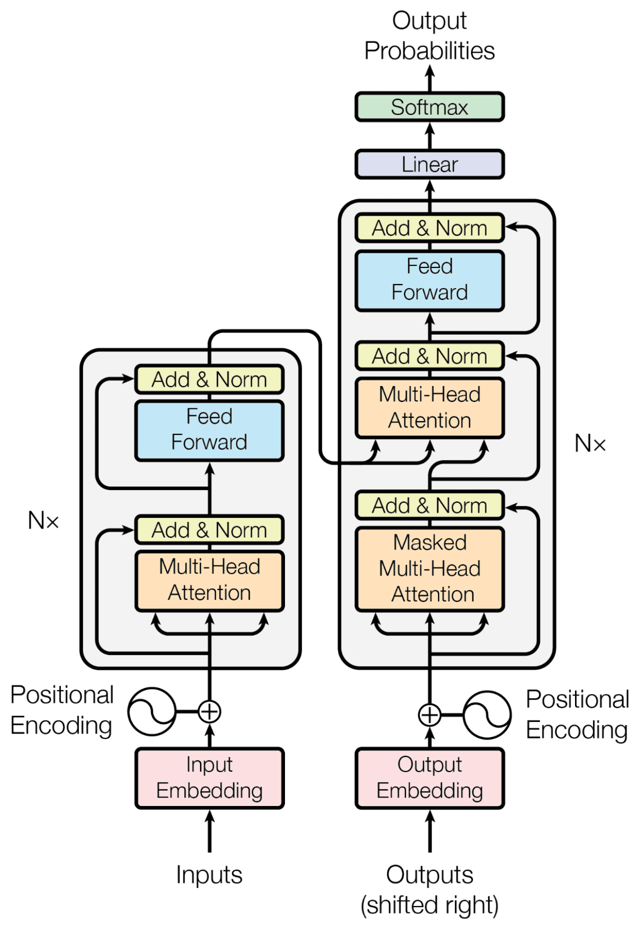
Overview of the Transformer Architecture

Ack: Devang Saraogi,
Teaching Assistant

Transformer Neural Network

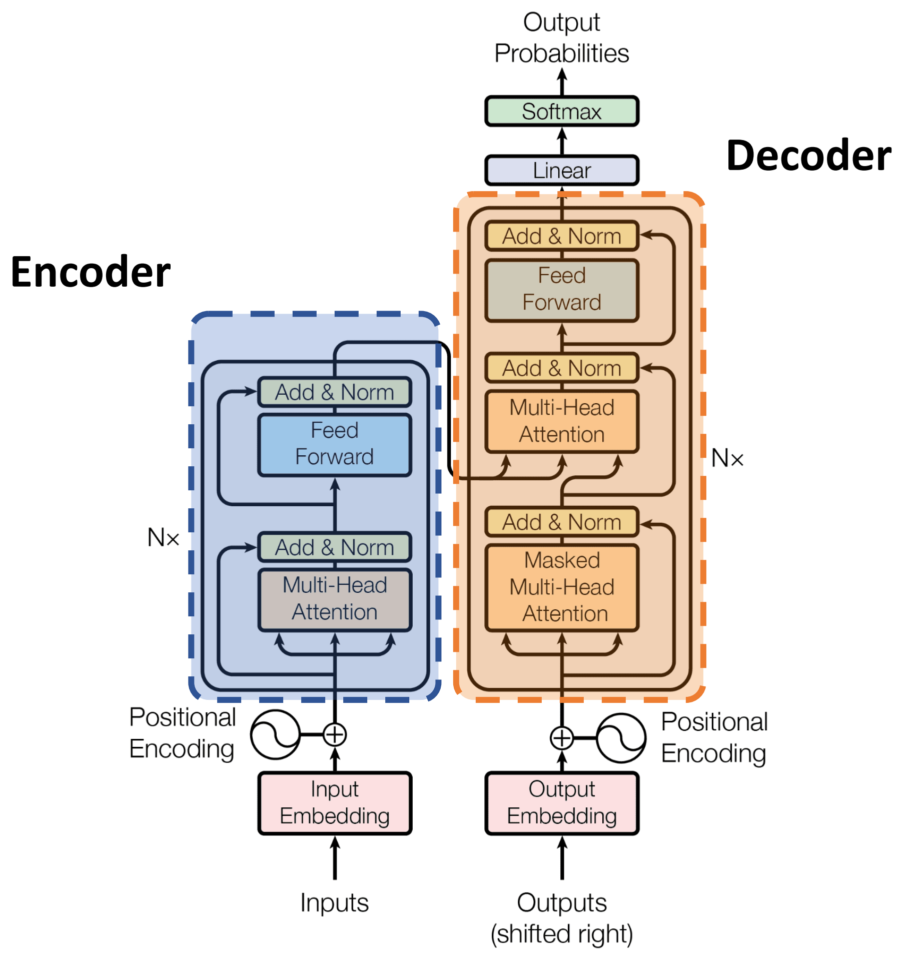
Transformer Architecture

- The Transformer consists of an **encoder** and a **decoder**, each composed of multiple layers.
- Each of these parts are capable of learning **good representations** of language, so good that language models can be built **independently** from each part.
- The key innovation of the Transformer is the use of **self-attention mechanisms** to capture dependencies between different positions in the input sequence. This is made possible by a learned attention score.

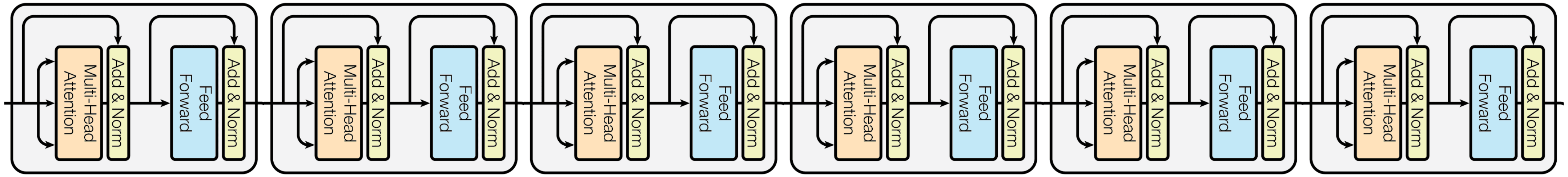


Transformer Neural Network

- A lot of subsequent research work involved either the encoder or the decoder units.
- Massive amounts of training texts were fed to the model to evaluate performance.
- These architectures stacked units as high as it was practically possible.
- **How many units can be stacked?** Turns out that one of the main distinguishing factors between different OpenAI GPT model sizes is the number of stacked decoders.



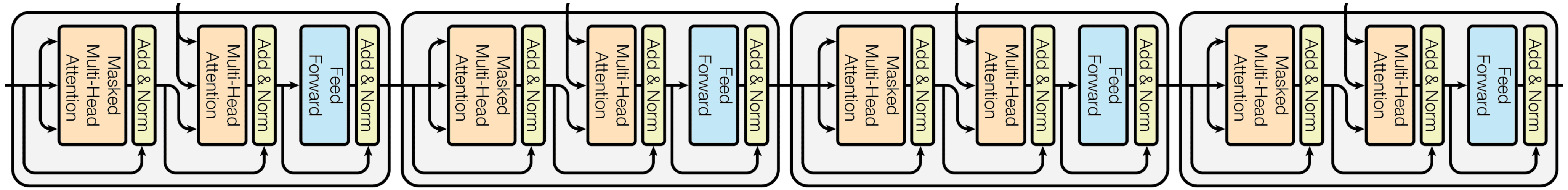
stack the encoder units to get



Bidirectional Encoder Representation Transfer

(BERT)

stack the decoder units to get



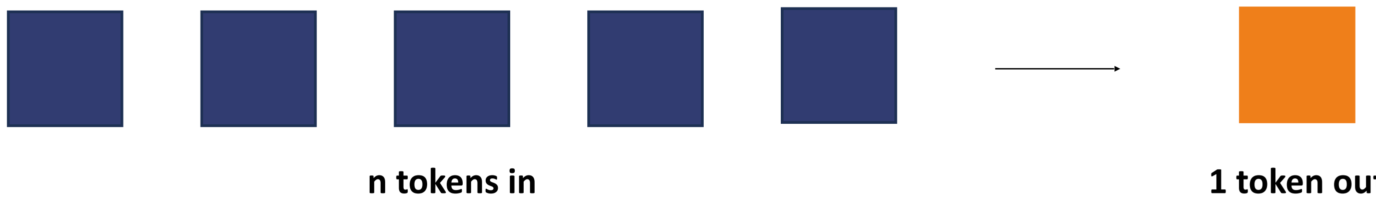
Generative Pretrained Transformer

(GPT)

High Level Look: Input to Output

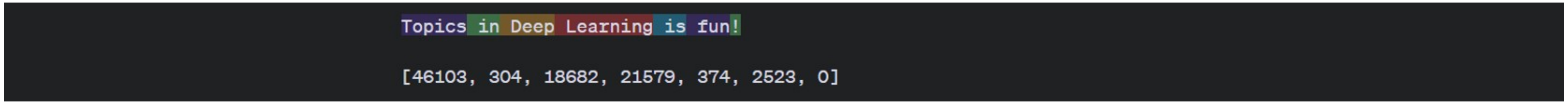
The general idea is to provide a few words as input to kick start the generation and return text that is likely to follow that input.

For example, if the GPT model is given the input “ ”, the model might return as output “ ”.



High Level Look: Input to Output

- Just like other neural networks, this model as well **requires numerical inputs**, so the **input string is first converted into a sequence of numbers**.
- This is achieved by using a **tokenizer**, which breaks the input into **chunks of few letters** and **assigns a number** called a **token** to each unique chunk.
- The tokens are **embedded** and passed to the model which **generates tokens one at a time**.



```
Topics in Deep Learning is fun!

[46103, 304, 18682, 21579, 374, 2523, 0]
```

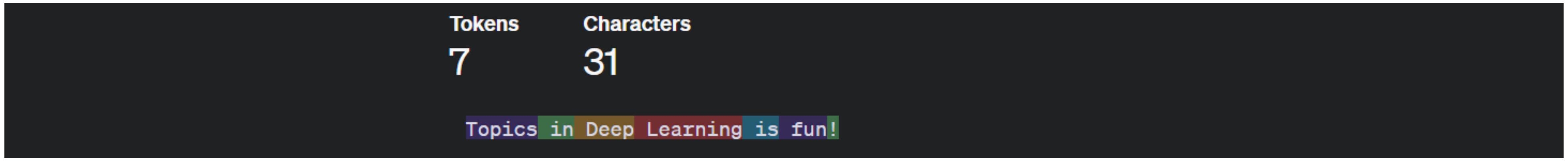
is the 46103rd token, is the 304th token (and so on…) in the tokenizer's vocabulary*. It isn't necessary for a word to be one token. Often complex words are not part of the vocabulary and are perceived by the model as two or more tokens (try the word "transformers").

**This is OpenAI's tokenizer. GPT's vocabulary comprises of 50257 tokens. Try: https://platform.openai.com/tokenizer*

High Level Look: Input to Output

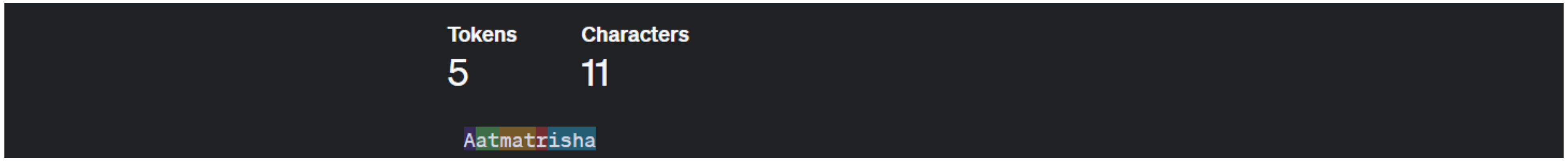
Tokenizer

A token is a chunk of text. Common and short words typically correspond to a single token, such as the words in the image below*.


Tokens 7 Characters 31

```
Topics in Deep Learning is fun!
```

Long and less commonly used words are generally broken up into several tokens. For example the word in the image below is broken up into five tokens.

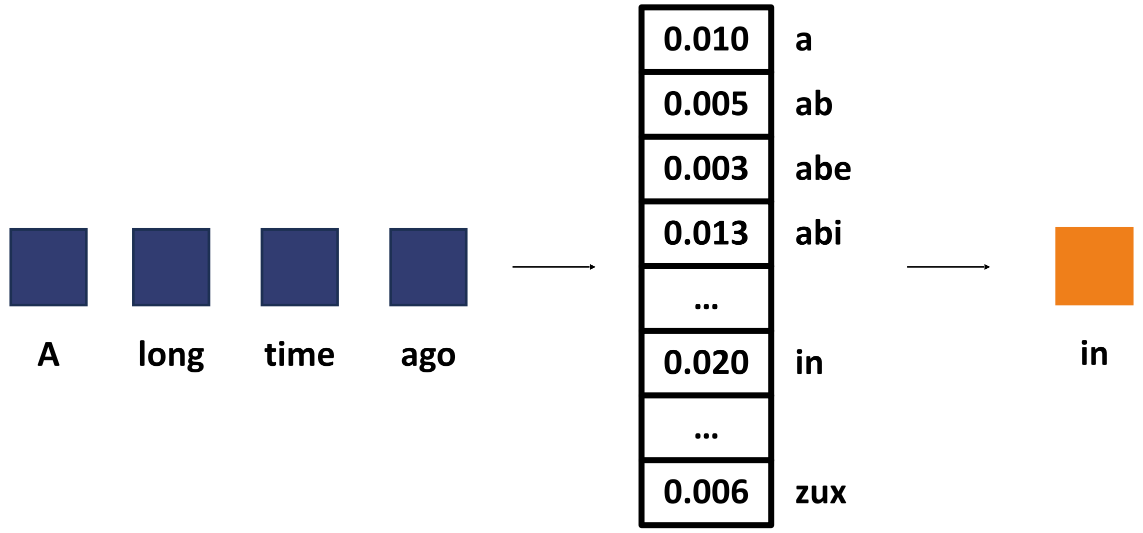

Tokens 5 Characters 11

```
Aatmatrisha
```

**Go to OpenAI’s Tokenizer page, enter some text, and see how it gets split up into tokens. Try: https://platform.openai.com/tokenizer*

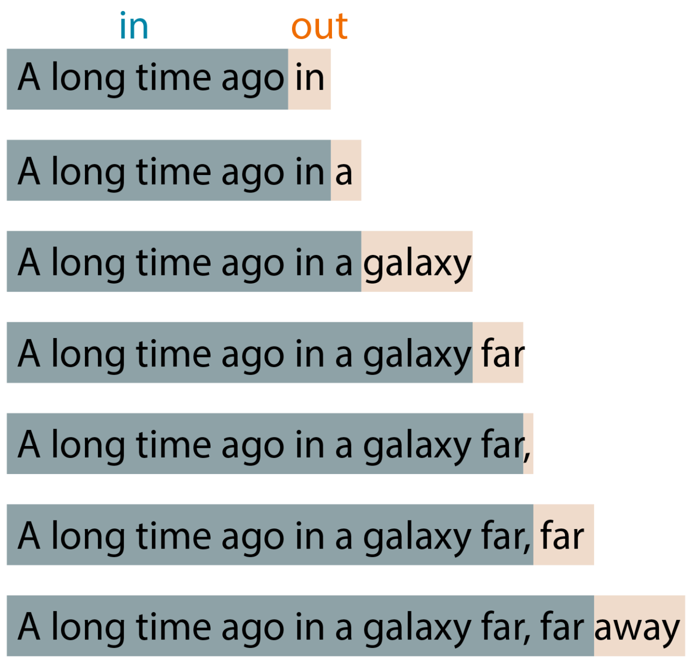
High Level Look: Input to Output

- The generation phase returns **a probability distribution**. The distribution contains a probability value for each possible token, representing how likely it is for that token to come next in the sentence.
- The newly generated token is **appended to the input sequence** and passed to the model for the next token.



High Level Look: Input to Output

- The generation phase returns **a probability distribution**. The distribution contains a probability value for each possible token, representing how likely it is for that token to come next in the sentence.
- The newly generated token is **appended to the input sequence** and passed to the model for the next token.


in
out
A long time ago in
A long time ago in a
A long time ago in a galaxy
A long time ago in a galaxy far
A long time ago in a galaxy far,
A long time ago in a galaxy far, far
A long time ago in a galaxy far, far away

High Level Look: Input to Output

Recap

- Just like other neural networks, this model as well **requires numerical inputs**, so the **input string is first converted into a sequence of numbers**.
- This is achieved by using a **tokenizer**, which breaks the input into **chunks of few letters** and **assigns a number** called a **token** to each unique chunk.
- The tokens are **embedded** and passed to the model which **generates tokens one at a time**.
- The generation phase returns **a probability distribution**. The distribution contains a probability value for each possible token, representing how likely it is for that token to come next in the sentence.
- The newly generated token is **appended to the input sequence** and passed to the model for the next token.

Unit 4: Overview of Latest Deep Learning Models

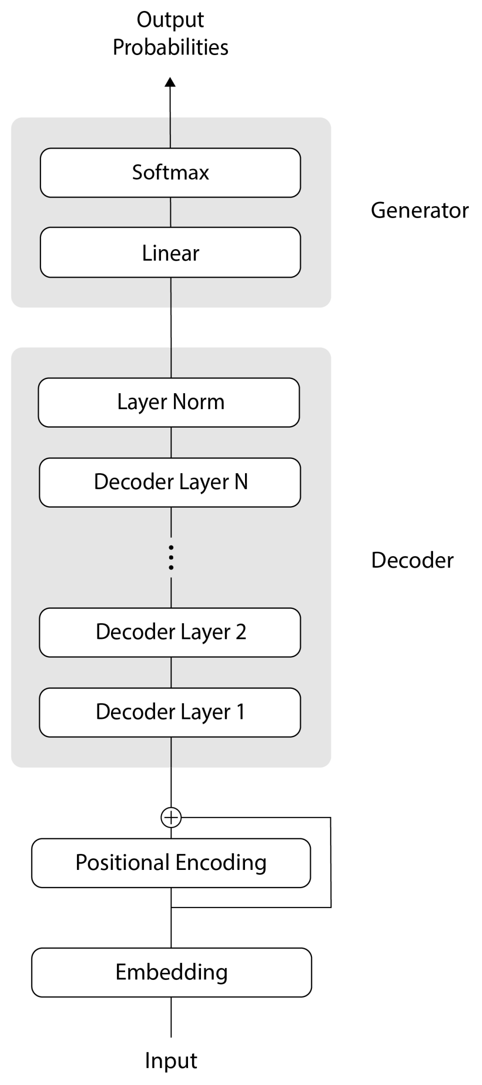
Transformer Architecture

Ack: Devang Saraogi,
Teaching Assistant

Transformer Architecture

High Level Look

- Initially, the input tokens undergo a couple of encoding steps: they're encoded using an **Embedding layer**, followed by a **Positional Encoding layer**, and then the **two encodings are added together**.
- Next, the encoded inputs go through a **sequence of N decoding steps**, followed by a **Normalization layer**.
- Finally, the decoded data is sent through a **Linear layer** and a **Softmax**, ending up with a probability distribution that can be used to select the next token.

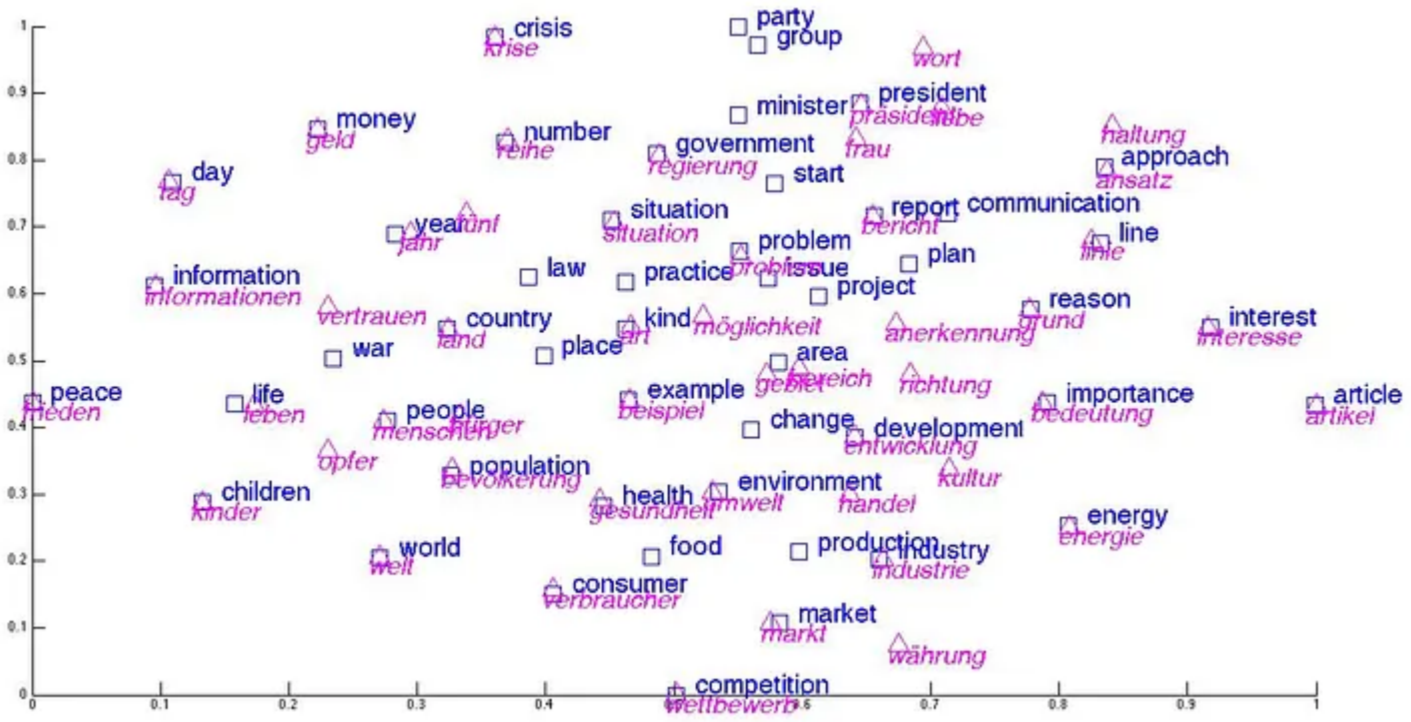


Embedding Layer

The Embedding layer turns each token in the input sequence into a vector. **The purpose of using an embedding instead of the original token is to ensure that a similar mathematical vector representation for tokens that are semantically similar is maintained.**

For example, let’s consider the words “**she**” and “**her**”.

- these words are **semantically similar**, as they both refer to a woman or girl, but the corresponding tokens can be completely different (for example, when using OpenAI’s tokenizer, “she” corresponds to token 7091, and “her” corresponds to token 372).
- embeddings for these two tokens will start out being very different from one another as well, because the weights of the embedding layer are **initialized randomly** and learned during training but if the two words frequently appear nearby in the training data, eventually the embedding representations **will converge to be similar.**



A plot of word embeddings in English and German. The semantic equivalence of words has been understood by the context they have been used in, so similar meanings are **co-located**.

Positional Encoding

Since the Transformer Architecture does not use any recurrent or convolutional layers, it needs a way to explicitly represent the position of each element in the sequence.

To address this issue, **positional encoding is added to the input embeddings of the sequence before being fed into the network.**

- The positional encoding is a **fixed vector that is added to the input embeddings** for each position in the sequence, so that the network can distinguish between different positions.
- The positional encoding can be **pre-computed** or can be **learned during training** with each choice having its advantages and disadvantages.

Positional Encoding

The most commonly used **precomputing** positional encoding scheme in Transformers is based on sinusoidal functions (sin and cosine). Specifically, the positional encoding for the ith position in the sequence and the jth dimension of the embedding is given by:

$$PE(i, j) = sin\left(\frac{i}{10000^{\frac{2j}{d}}}\right)$$

where, d is the dimension of the embedding. The positional encoding is added to the input embedding before being fed into the network, so that the final embedding for the ith position is:

$$X(i) = E(i) + PE(i)$$

where $E(i)$ is the original input embedding for the ith position, and $PE(i)$ is the corresponding positional encoding vector.

Positional Encoding

The main **advantage of precomputing the values for the positional encoding (rather than using a trainable embedding)** is that the model ends up with fewer parameters to train. **This reduction in parameters leads to improved training performance**, which is tremendously important when working with large language models.

On the other hand, having a **trainable embedding** introduces **flexibility**. Using a precomputation scheme involving sinusoidal functions, introduces patterns. This effects the performance of the neural network in certain tasks.

Unit 4: Overview of Latest Deep Learning Models

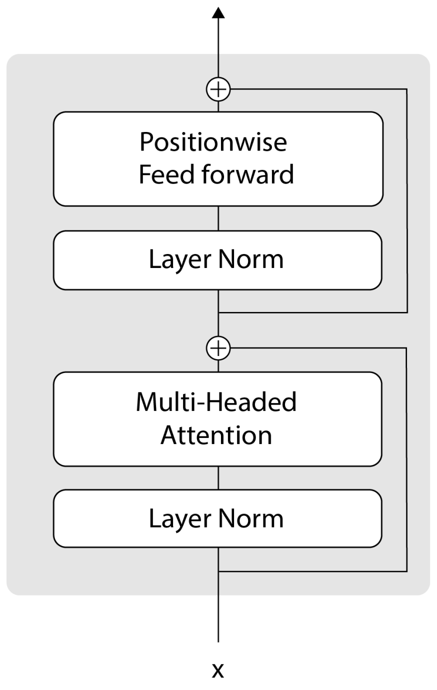
Transformer Architecture (Decoder)

Ack: Devang Saraogi,
Teaching Assistant

Decoder

Decoder layer mainly consists of two steps

- the attention step, is responsible for communication between tokens
- and the feed forward step, is responsible for the computation of the predicted tokens
- the normalization layer ensures each embedding distribution will start at unit normal (centered around zero and with standard deviation of one)
- additionally, there are residual connections surrounding these layers, which enable bypassing these layers; allowing flow of data around these layers **(this helps the model training to converge better)**



Decoder Layer

Masked Self Attention

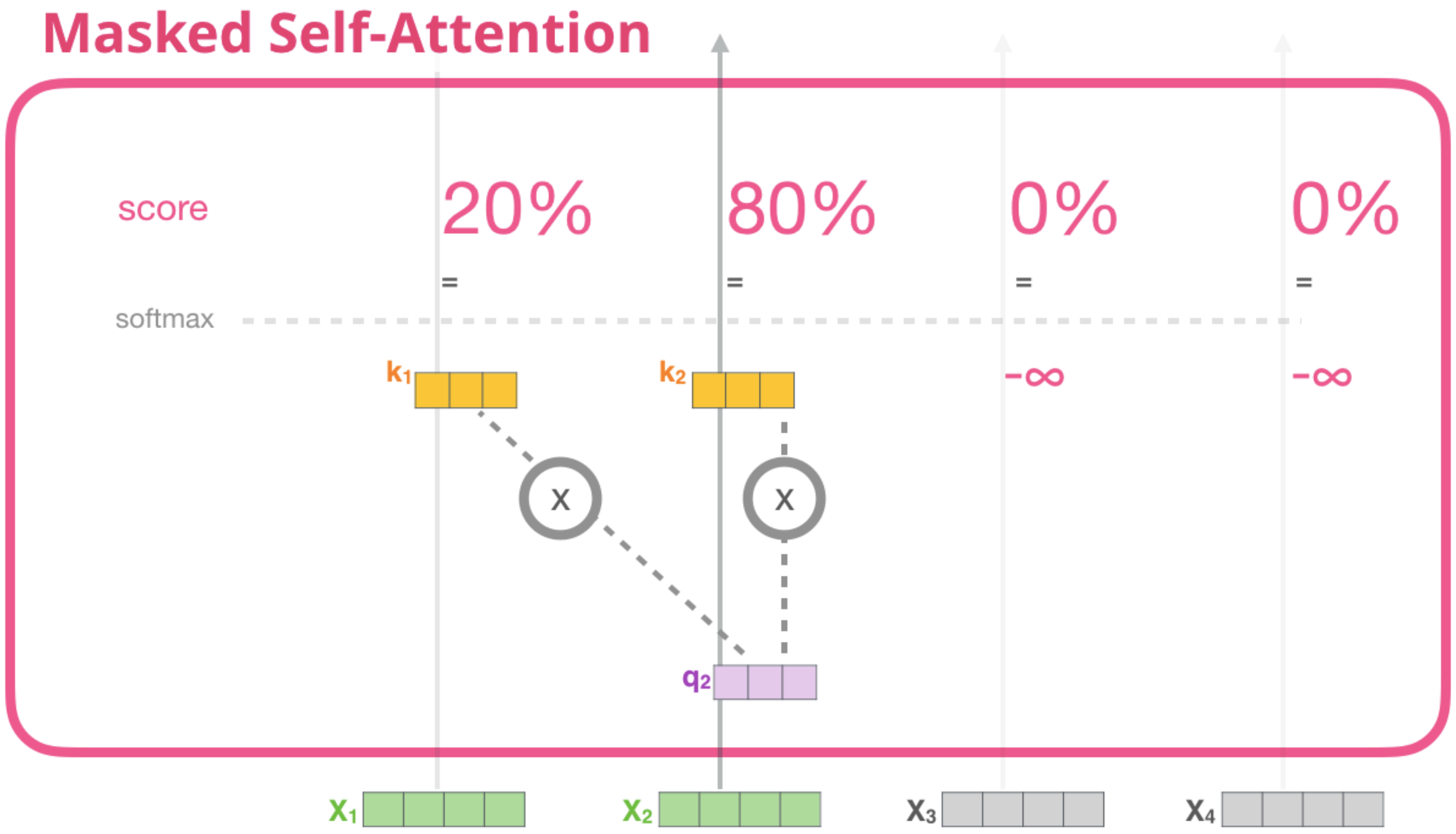
Masked self-attention is identical to self-attention.

As we have seen earlier, self-attention is applied through three main steps:

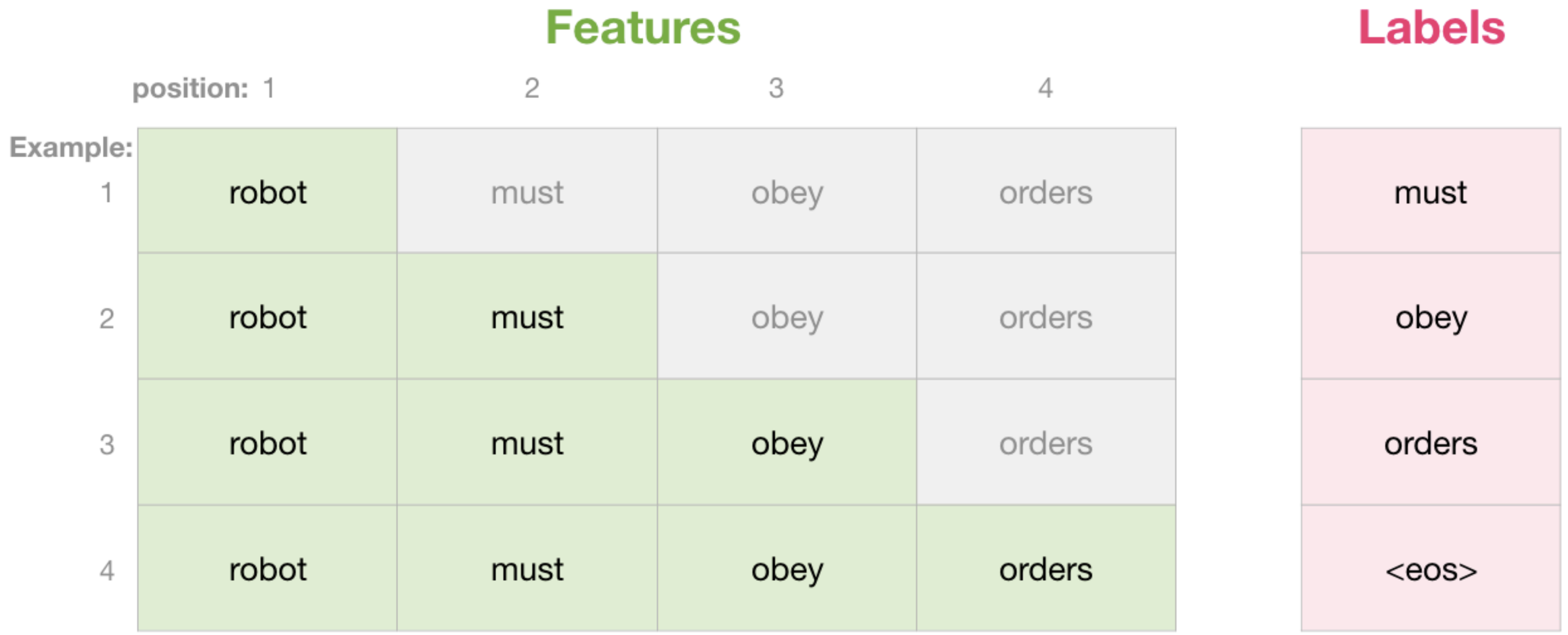
- create the **query**, **key**, and **value** vectors for each token
- for each input token, use its query vector to score against all the other key vectors
- sum up the value vectors after multiplying them by their associated scores

Masked self-attention only differs in scoring step of the self attention mechanism. In the scoring step, the query vector is scored against **all** other key vectors present. **Whereas in the masked self-attention mechanism only the past tokens are considered while scoring. The rest of the tokens are scored as zero.**

Assuming that only two tokens out of the four have been passed through the model and currently the second token is being observed. **In this case, the last two tokens are masked.** So the model interferes in the scoring step. **It basically always scores the future tokens as 0 so the model can't peak to future words.**



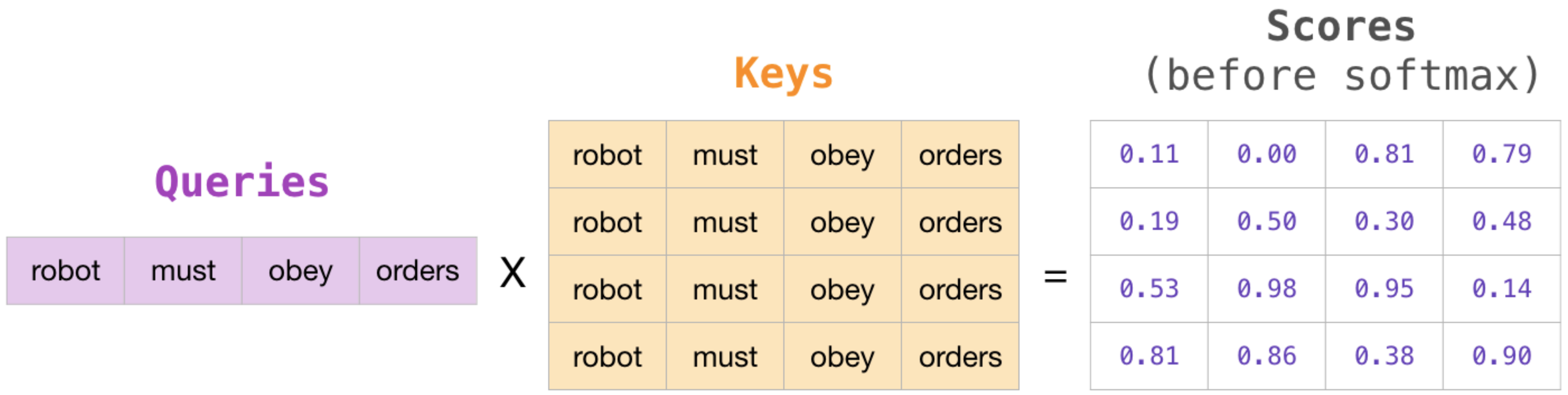
This masking is often implemented as a matrix called an attention mask. Consider a sequence of four words . In a language modeling scenario, this sequence is absorbed in four steps – one per word (assuming for now that every word is a token). As these models work in batches, we can assume a batch size of 4 for this toy model that will process the entire sequence (with its four steps) as one batch.



	Features				Labels
Example: / position:	1	2	3	4	
1	robot	must	obey	orders	must
2	robot	must	obey	orders	obey
3	robot	must	obey	orders	orders
4	robot	must	obey	orders	<eos>

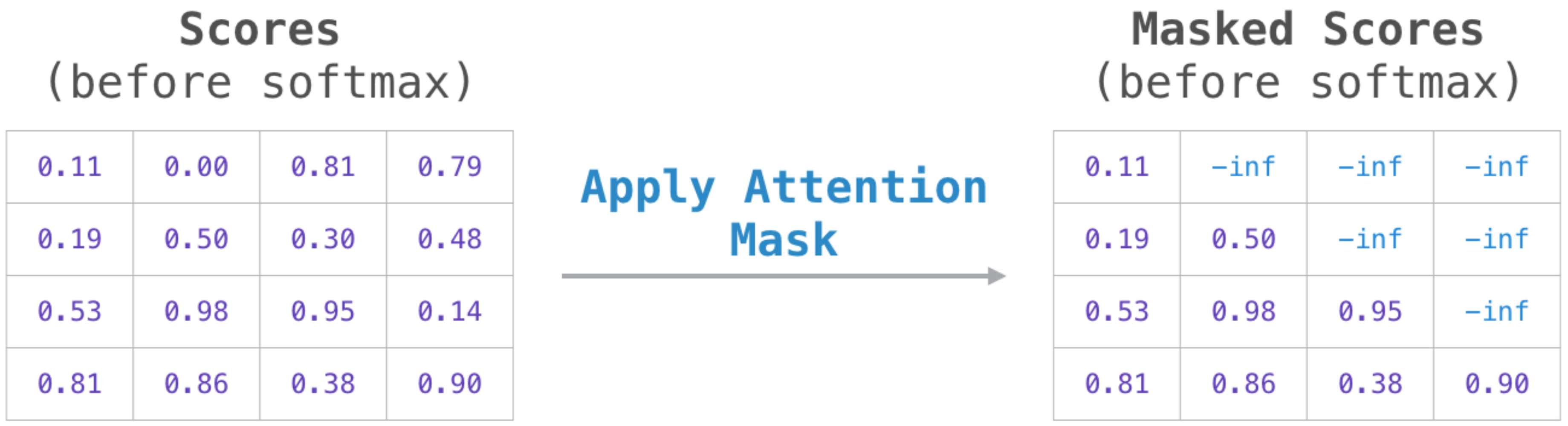
Masked Self Attention

In matrix form, the scores are calculated by multiplying a queries matrix by a keys matrix. Except instead of the word, there would be the query (or key) vector associated with that word in that cell.



Masked Self Attention

After the multiplication, the attention mask triangle is stacked onto the matrix. It set the cells we want to mask to -infinity or a very large negative number.



Scores (before softmax)

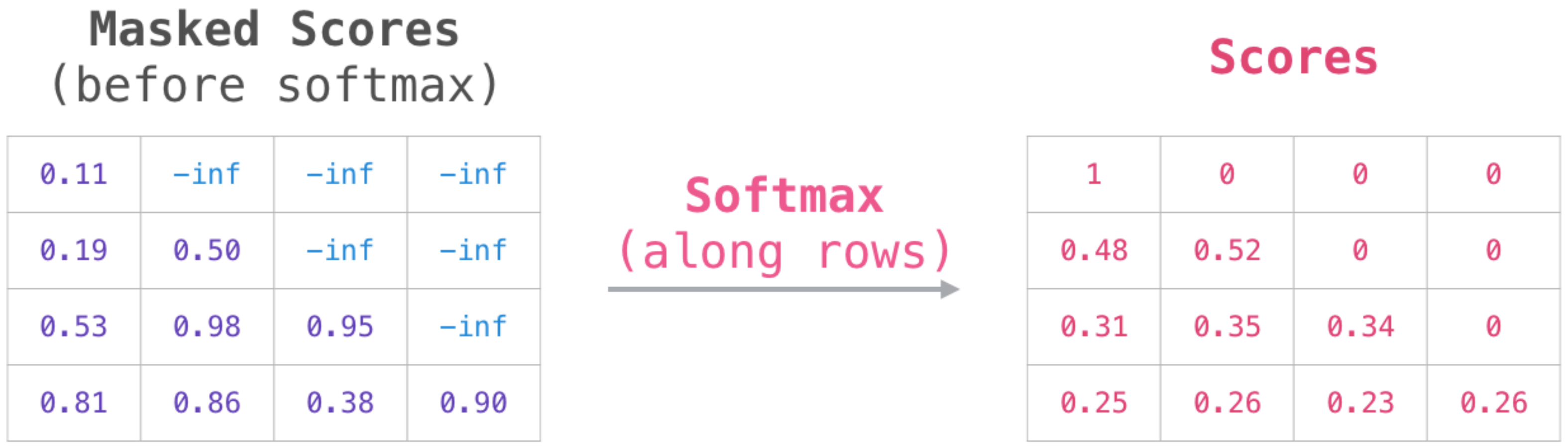
0.11	0.00	0.81	0.79
0.19	0.50	0.30	0.48
0.53	0.98	0.95	0.14
0.81	0.86	0.38	0.90

Masked Scores (before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Masked Self Attention

Then, applying softmax on each row produces the actual scores that are used for self-attention.



Masked Scores (before softmax)

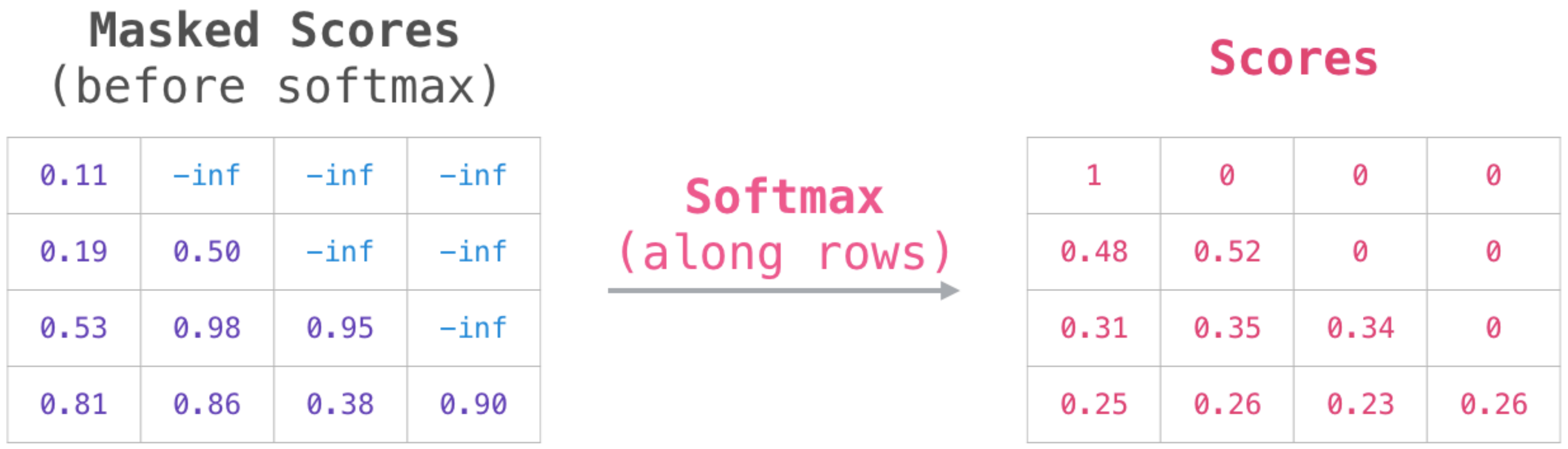
0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

Softmax (along rows)

Scores

1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

because of the attention mask, the tokens which were marked as negative infinity are now zero.



Masked Scores
(before softmax)

0.11	-inf	-inf	-inf
0.19	0.50	-inf	-inf
0.53	0.98	0.95	-inf
0.81	0.86	0.38	0.90

$\xrightarrow{\text{Softmax (along rows)}}$

Scores

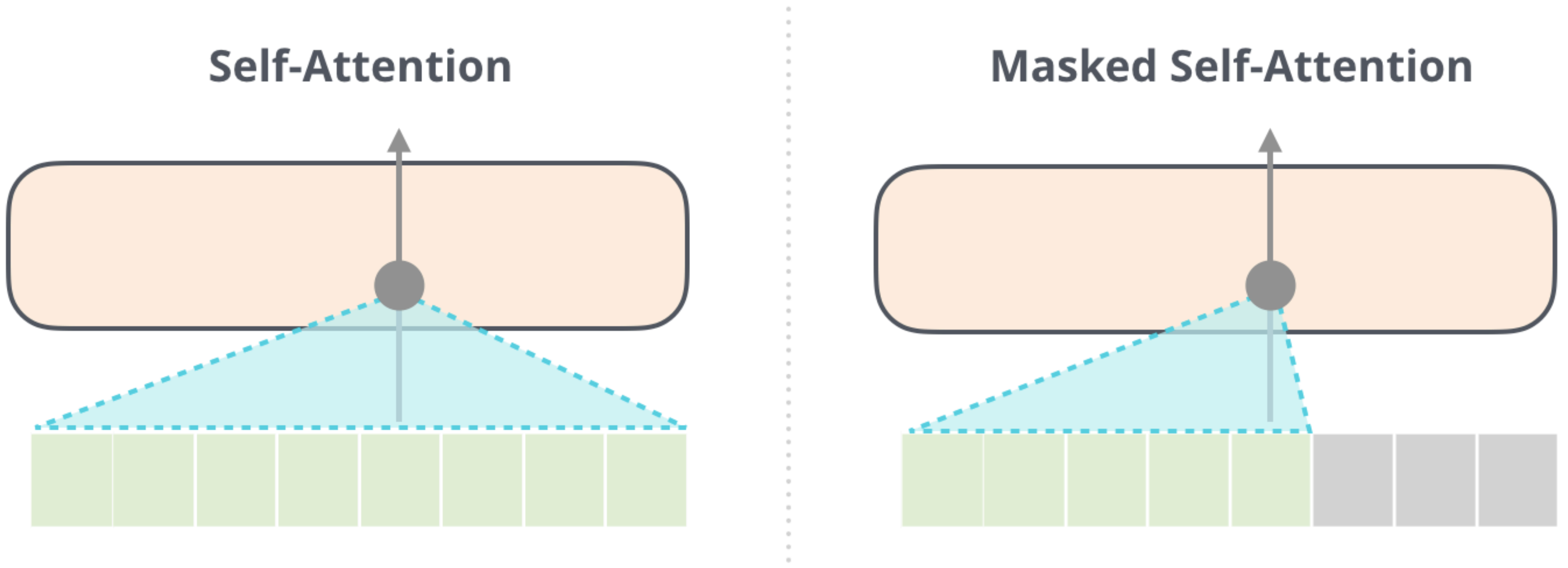
1	0	0	0
0.48	0.52	0	0
0.31	0.35	0.34	0
0.25	0.26	0.23	0.26

what this scores table means is the following:

- when the model processes the first example in the dataset (row #1), there is only one word so 100% of its attention will be on that word
- when the model processes the second example in the dataset (row #2), which contains the words , when it processes the word , 48% of its attention will be on , and 52% of its attention will be on

and so on

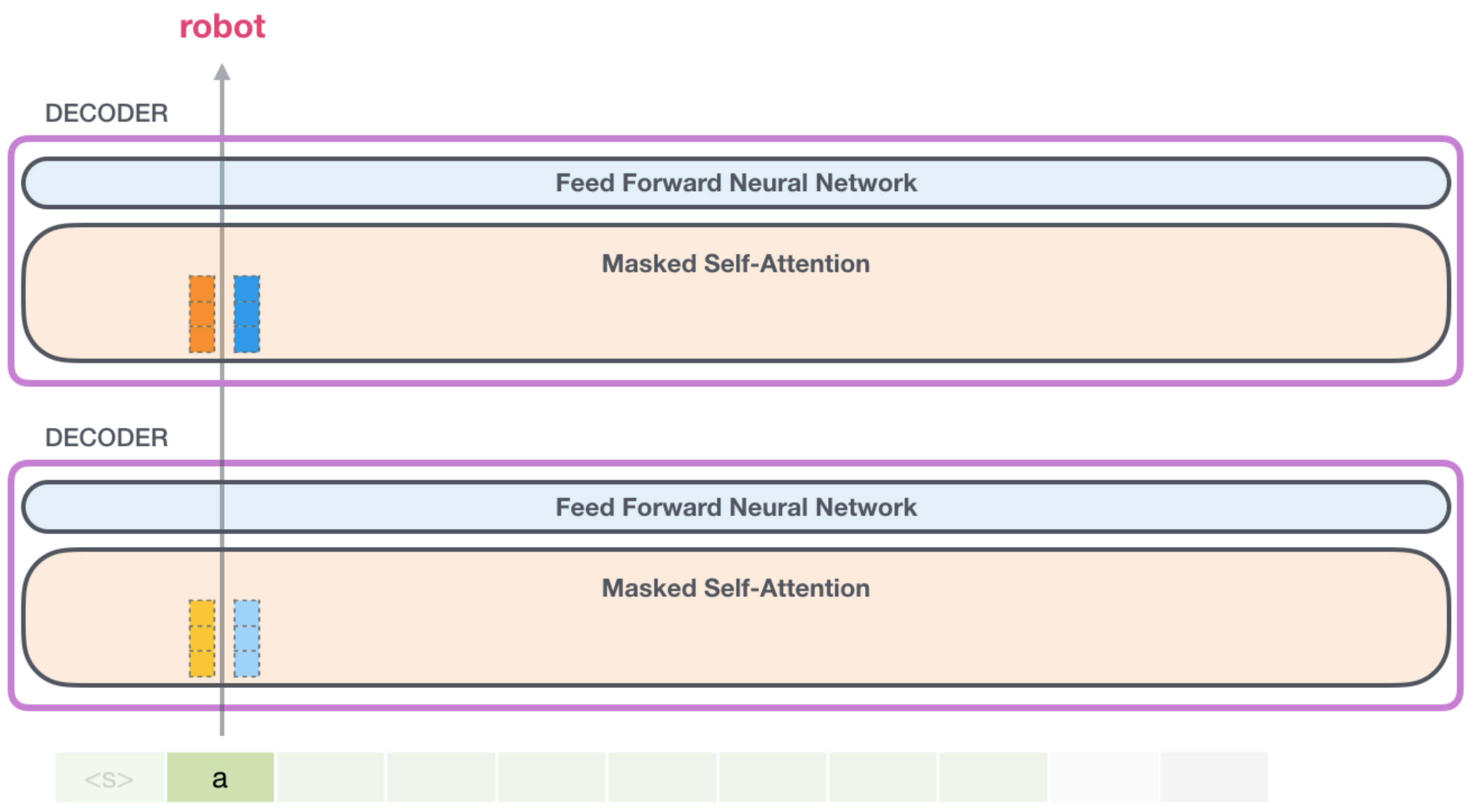
Masked Self Attention



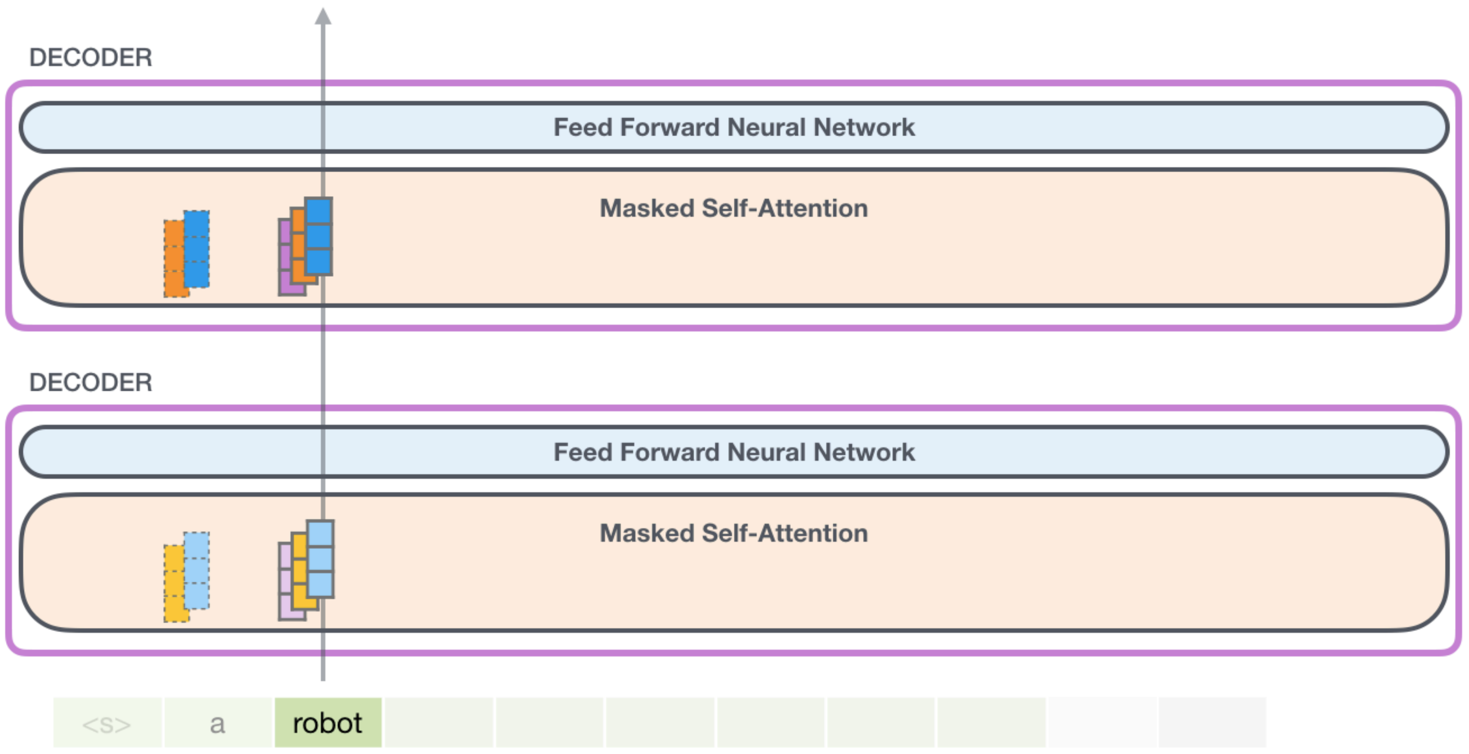
Masking ensures that the model only attends to tokens that have been generated up to the current position. Masking prevents the model from "**cheating**" by looking at future tokens when generating the current token.

Processing One Token At a Time

- Processing the first token in the sequence. The model holds onto the key and value vectors of the token. Every self-attention layer holds on to its respective key and value vectors for that token.



- Now in the next iteration, when the model processes the word , it does not need to generate query, key, and value queries for the a token. **It just reuses the ones it saved from the first iteration**. When the model is only adding one new word after each iteration, it would be inefficient to recalculate self attention along earlier paths for tokens that have already been processed.



PES UNIVERSITY

Unit 4: Overview of Latest Deep Learning Models

Transformer (Multi-headed Masked Attention)

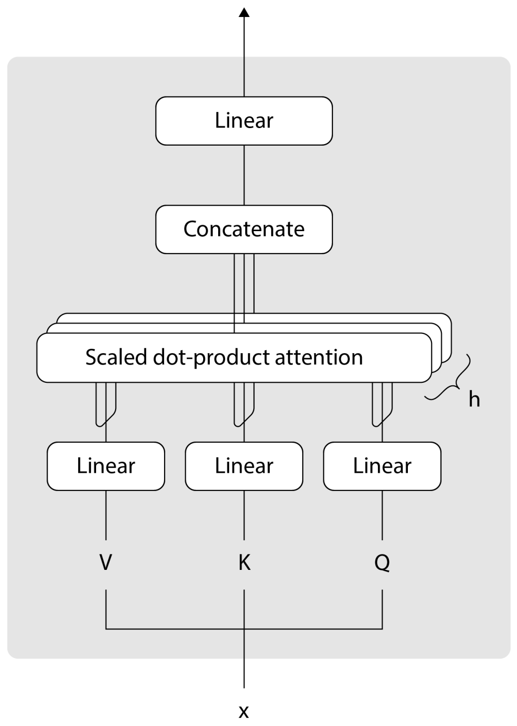
Ack: Devang Saraogi,
Teaching Assistant

Masked Multiheaded Self Attention

As the name implies, the multi-headed attention module processes several instances of attention computations in parallel.

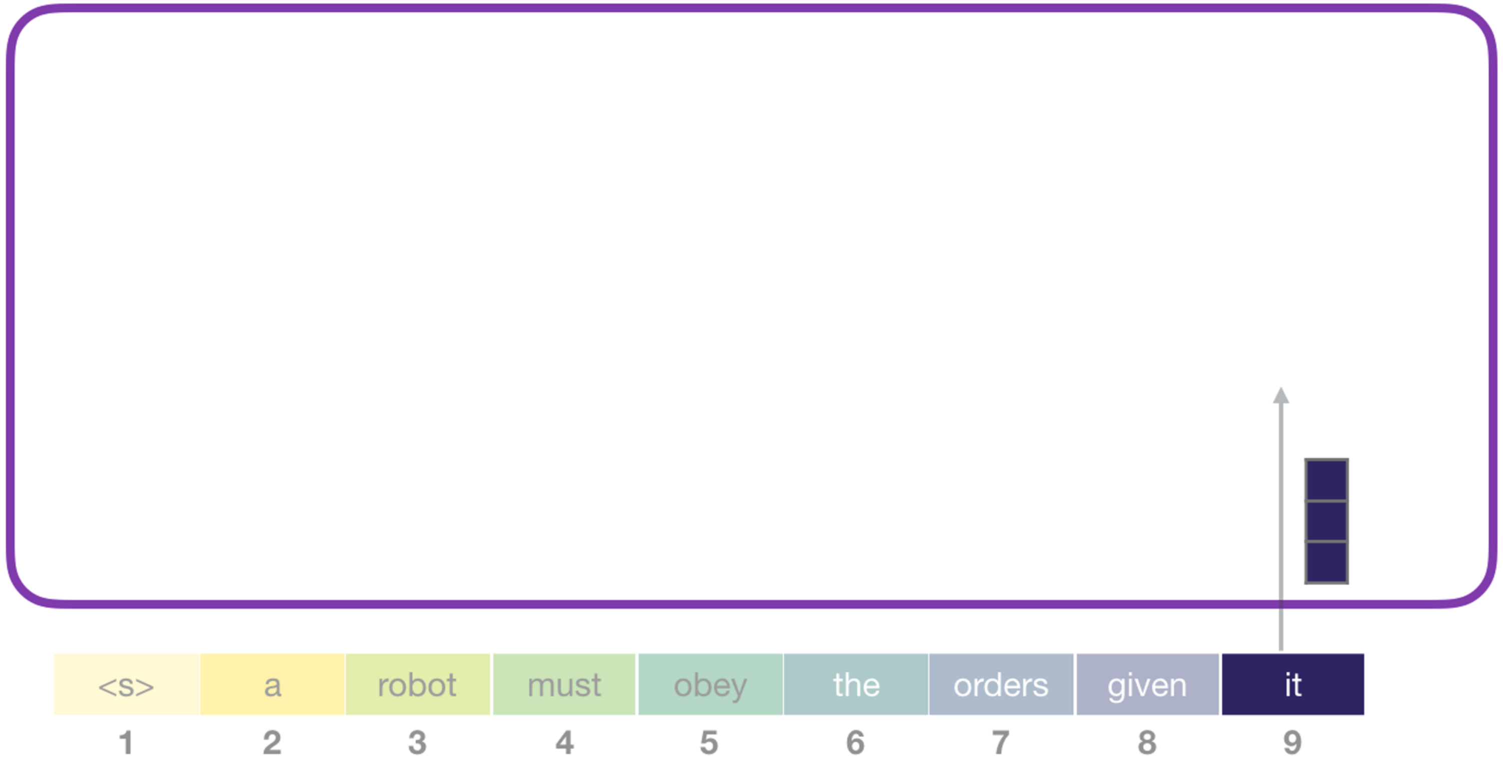
Multi-head attention is capable of **consuming information in different ways** from an input sequence because of the multiple mechanisms (heads). It allows for a more robust way of processing and learning from data.

For example, it can attend to longer-term dependencies versus shorter-term dependencies.



Step 1: Creating Queries, Keys and Values

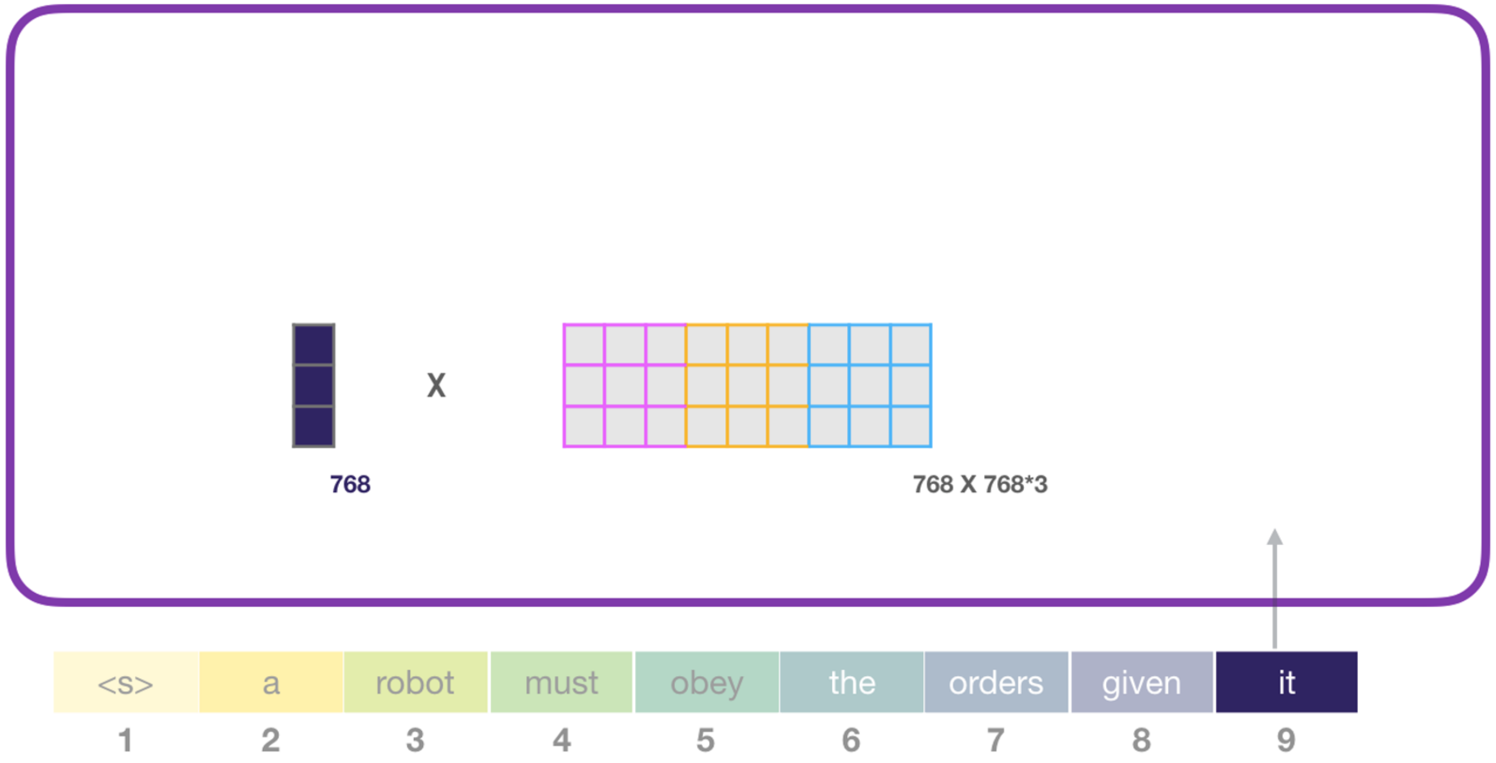
Let’s assume the model is processing the word . In the bottom block (embedding + positional encoding), then its input for that token would be the embedding of **+** the positional encoding for position 9.



Step 1: Creating Queries, Keys and Values

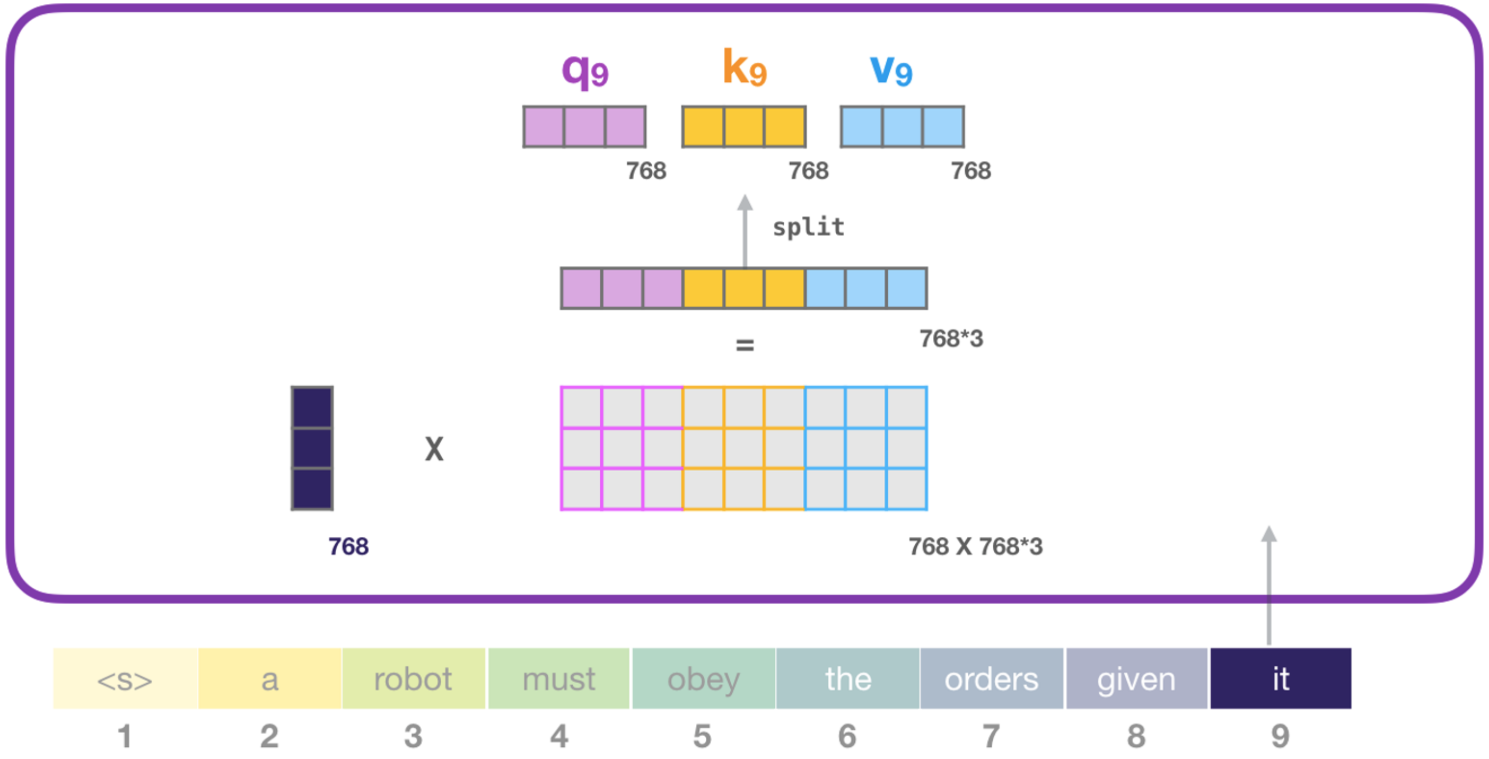
Every block in a transformer has its own weights. The weight matrix is used to create the queries, keys and values.

Self-attention multiplies its input by its weight matrix and adds a bias vector.



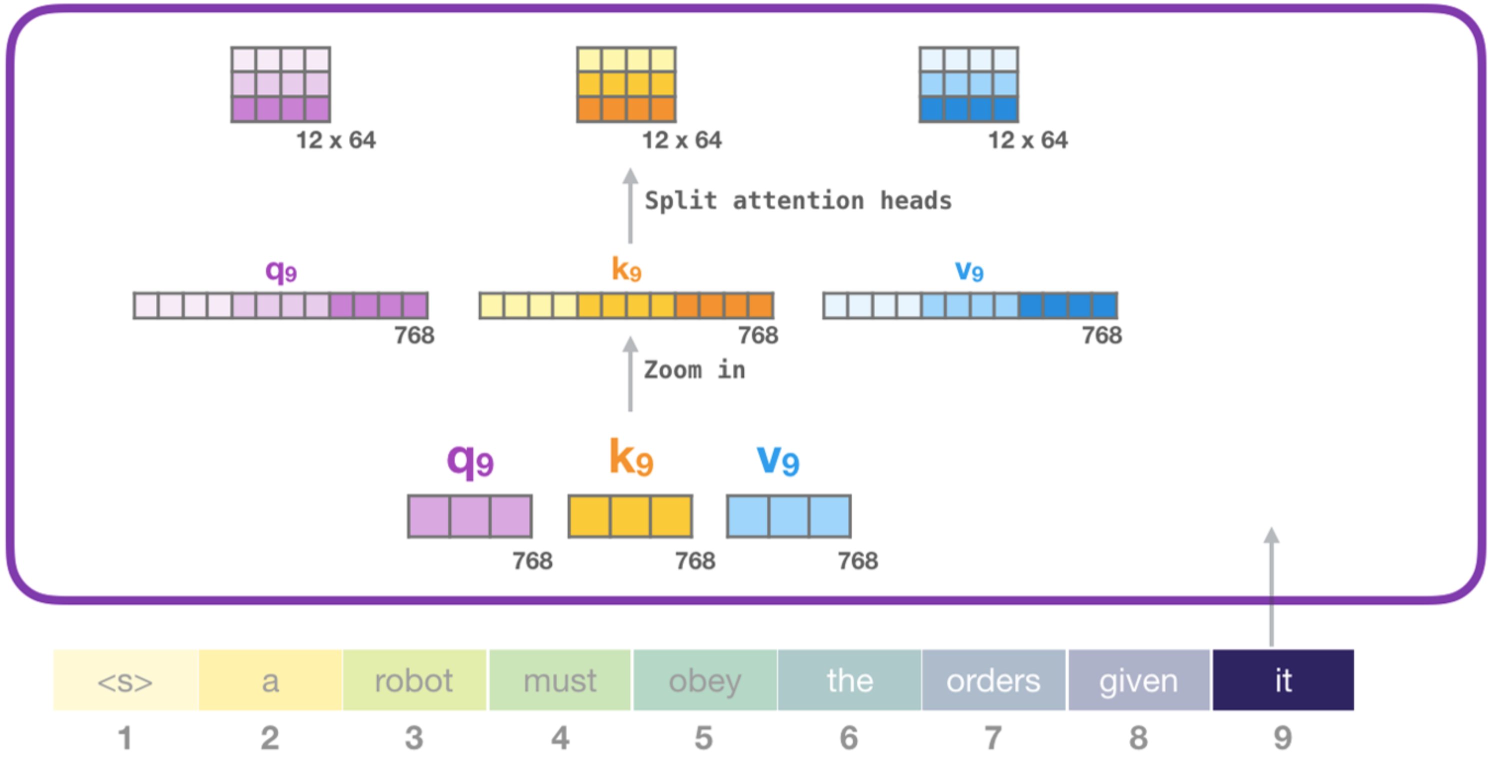
Step 1: Creating Queries, Keys and Values

The multiplication results in a vector that's basically a concatenation of the query, key, and value vectors for the word .



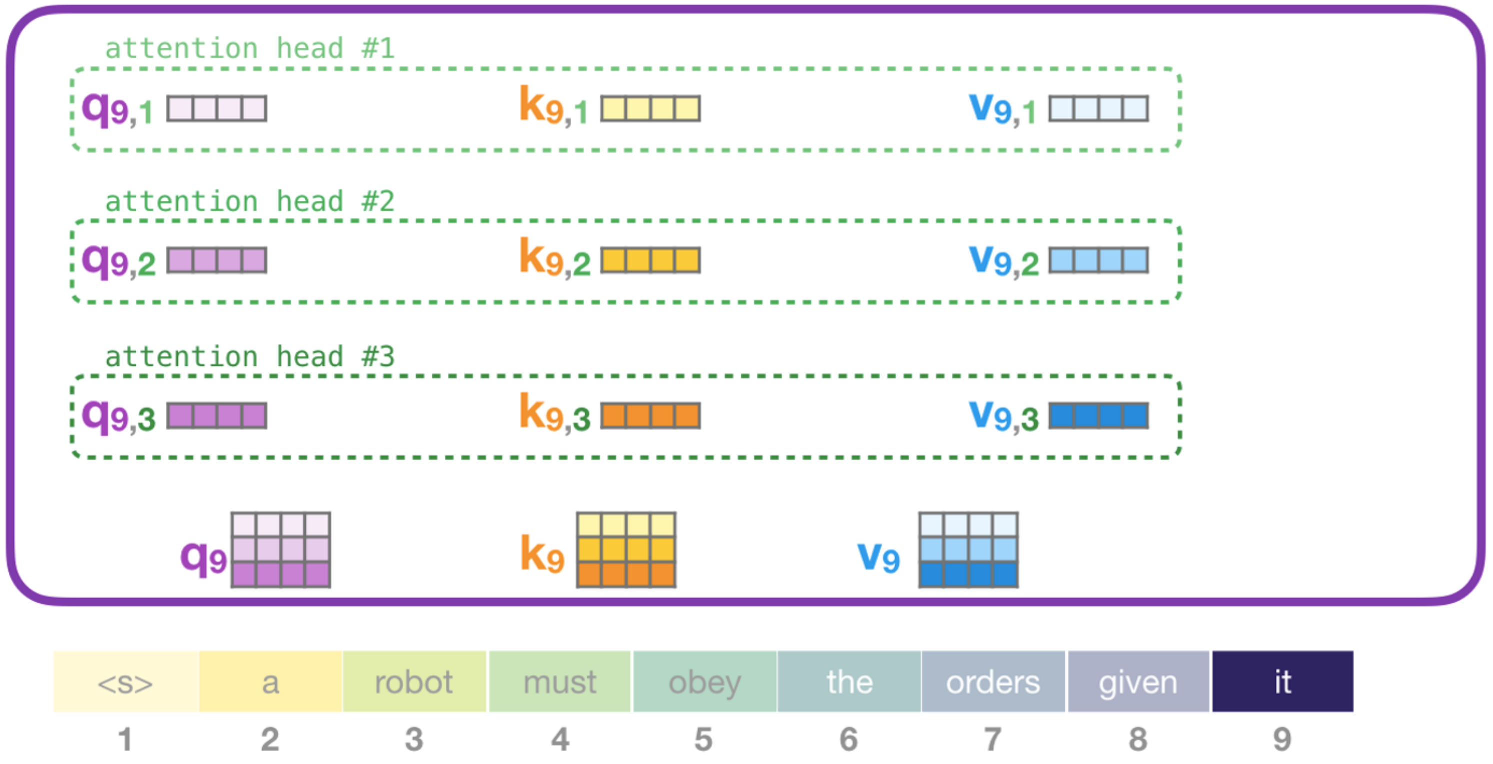
Step 2: Splitting into Attention Heads

Self attention is conducted multiple times on different parts of the **[Q, K, V]** vectors. **“Splitting”** attention heads is simply reshaping the long vector into a matrix.



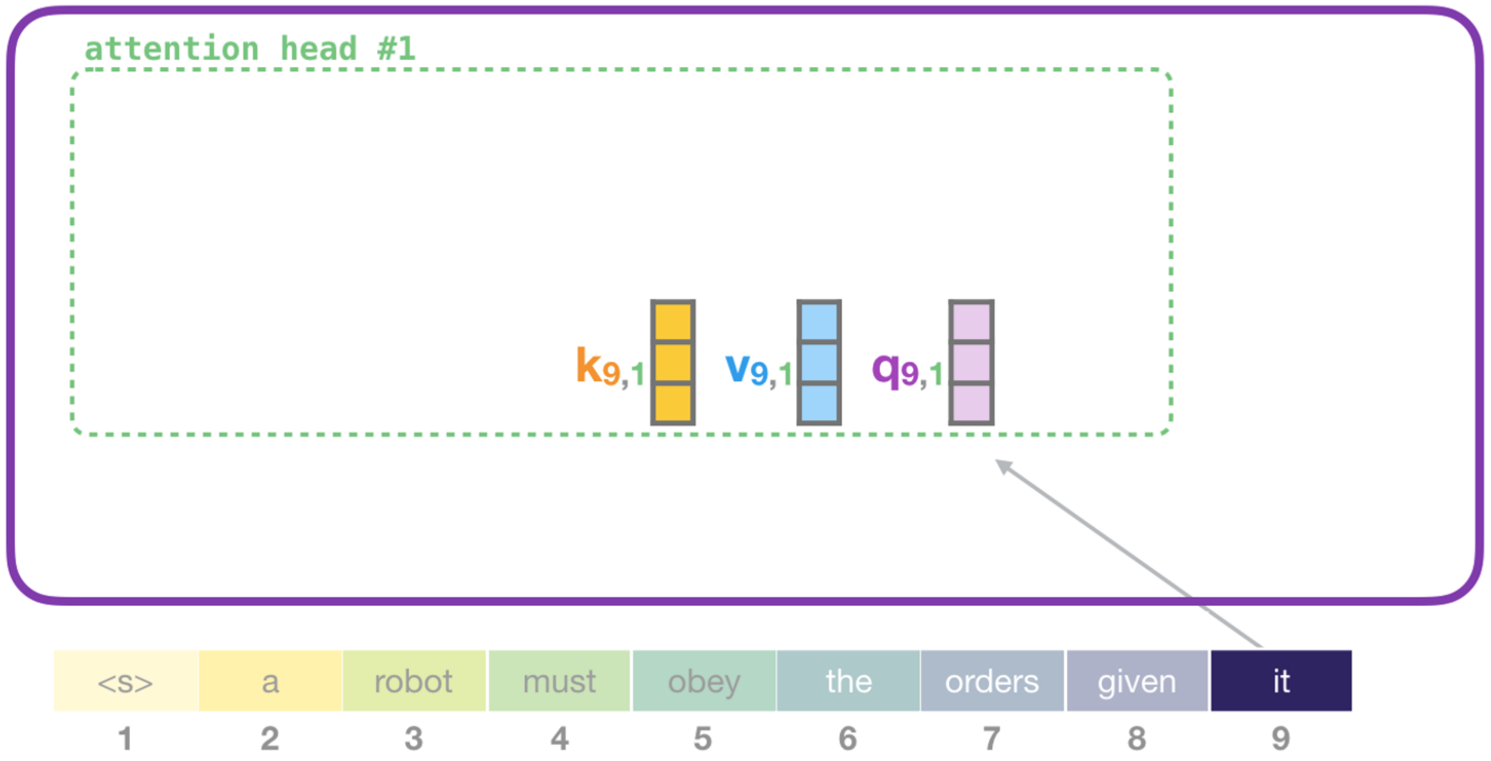
Step 2: Splitting into Attention Heads

Multiple attention heads can be looked at like this (assuming there are three attention heads).



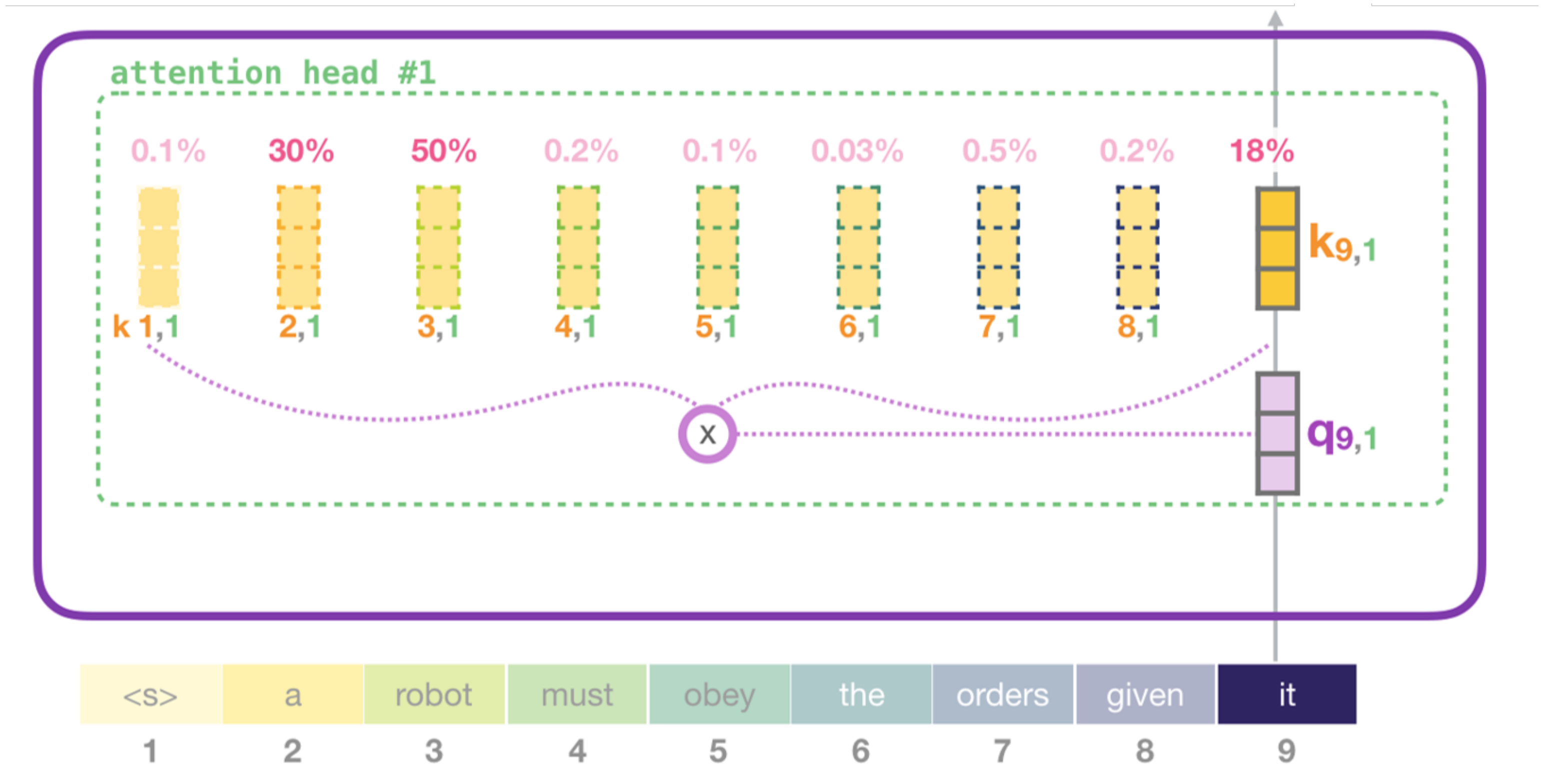
Step 3: Scoring

Considering only one attention head (and that all the others are conducting a similar operation), scoring is performed in each attention head.



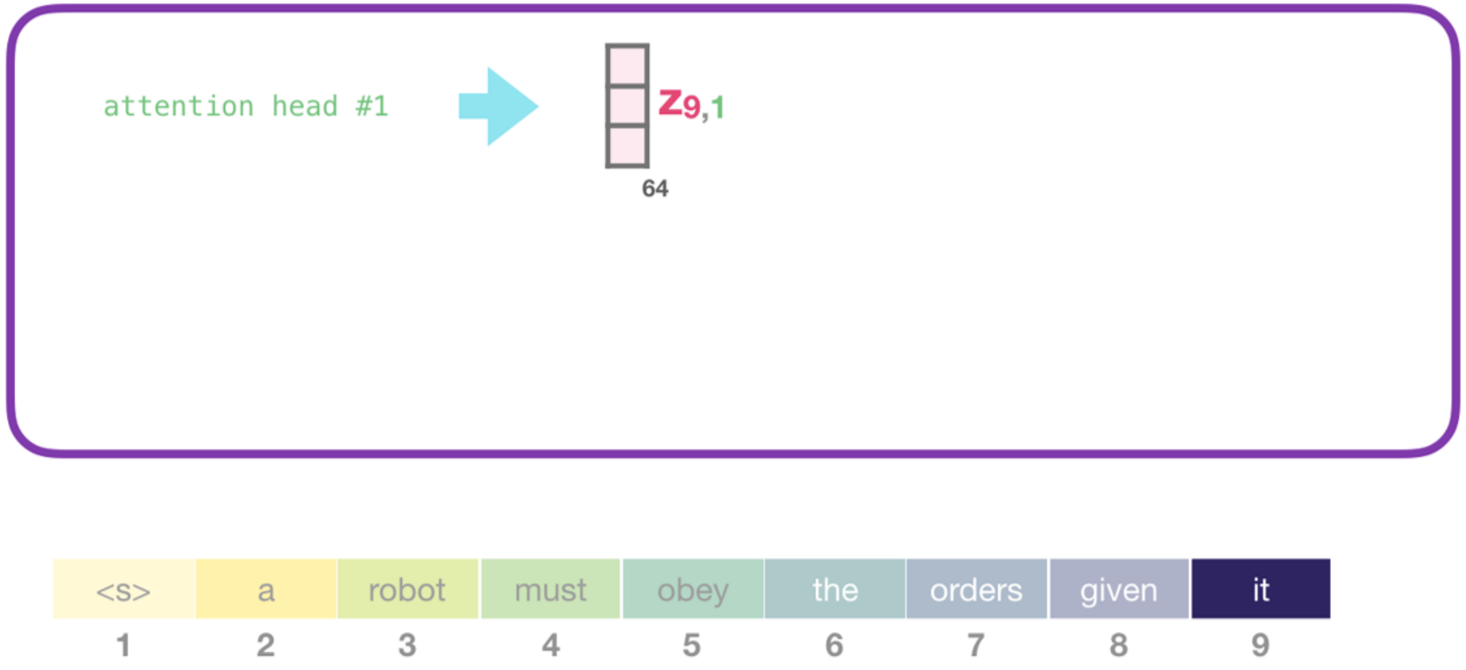
Step 3: Scoring

Query vectors are scored against all other key vectors.



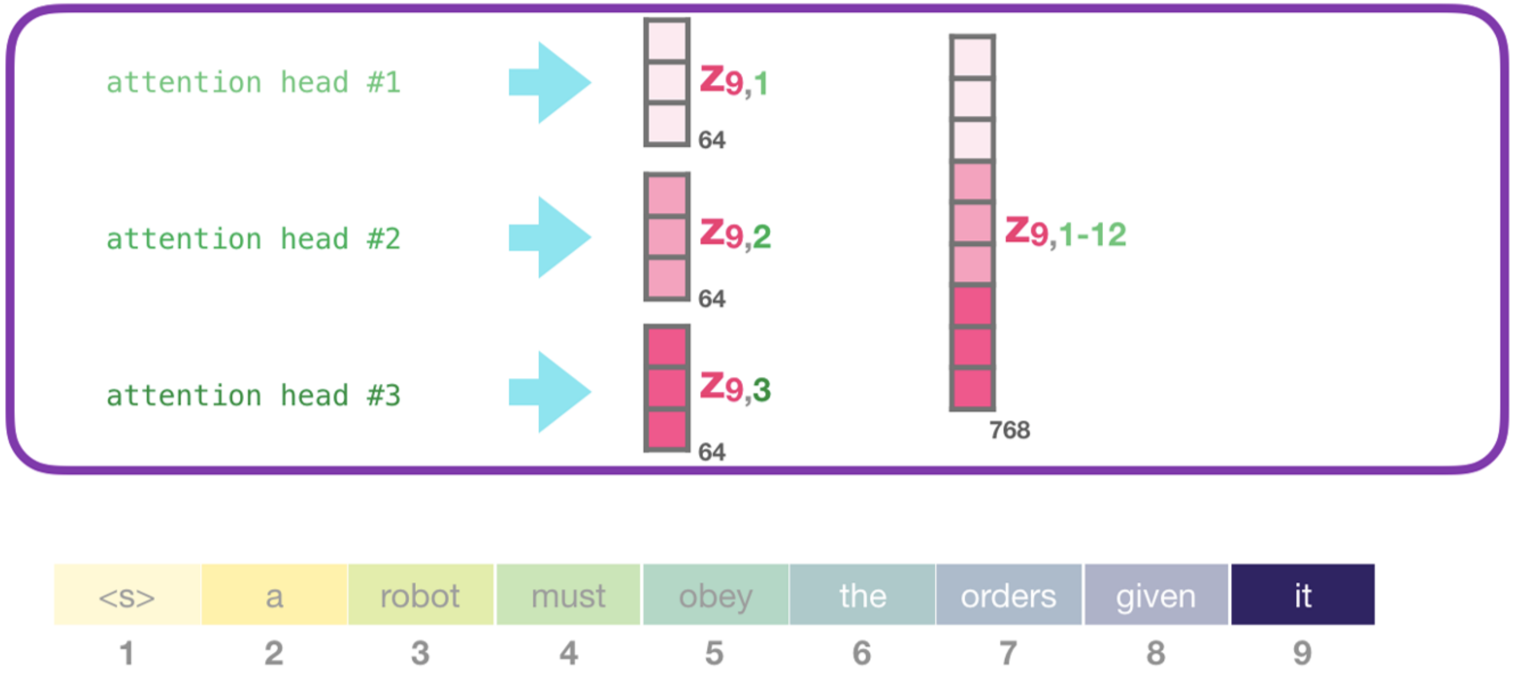
Step 4: Sum

Each value is multiplied with its score and summed up, producing the result of self-attention for attention-head.



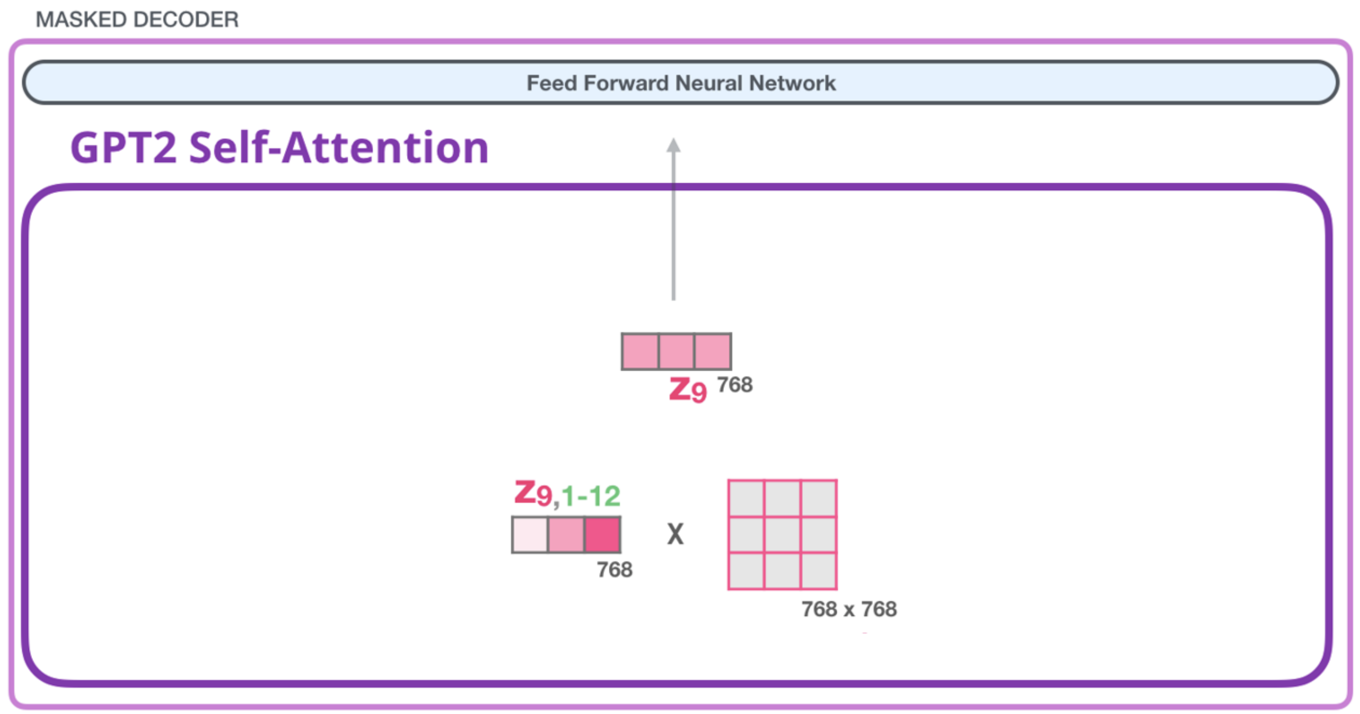
Step 5: Merging Attention Heads

All attention heads are concatenated into one vector. This is not ready for the feed forward layers and needs to be projected into an acceptable vector.



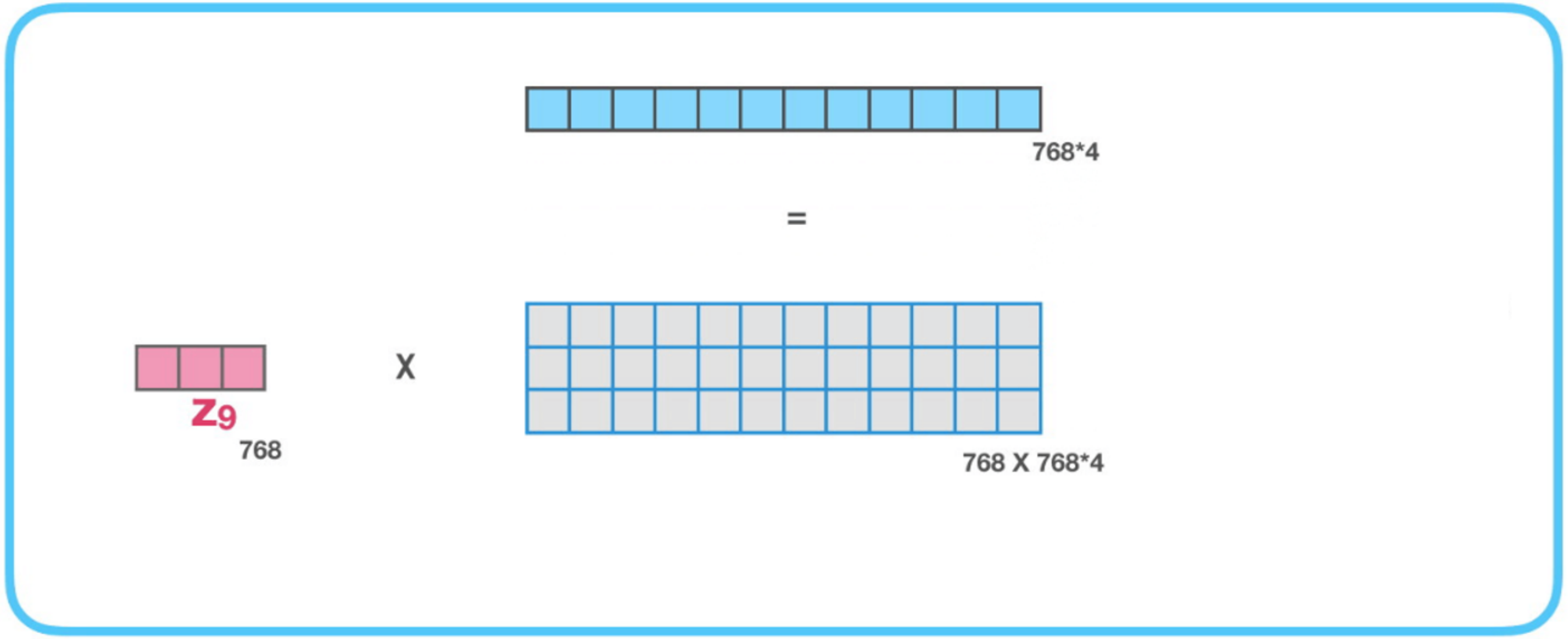
Step 6: Projecting

The model learns how to map concatenated self-attention results into a vector that the feed-forward neural network can deal with. A second large weight matrix projects the results of the attention heads into the output vector of the self-attention sublayer.



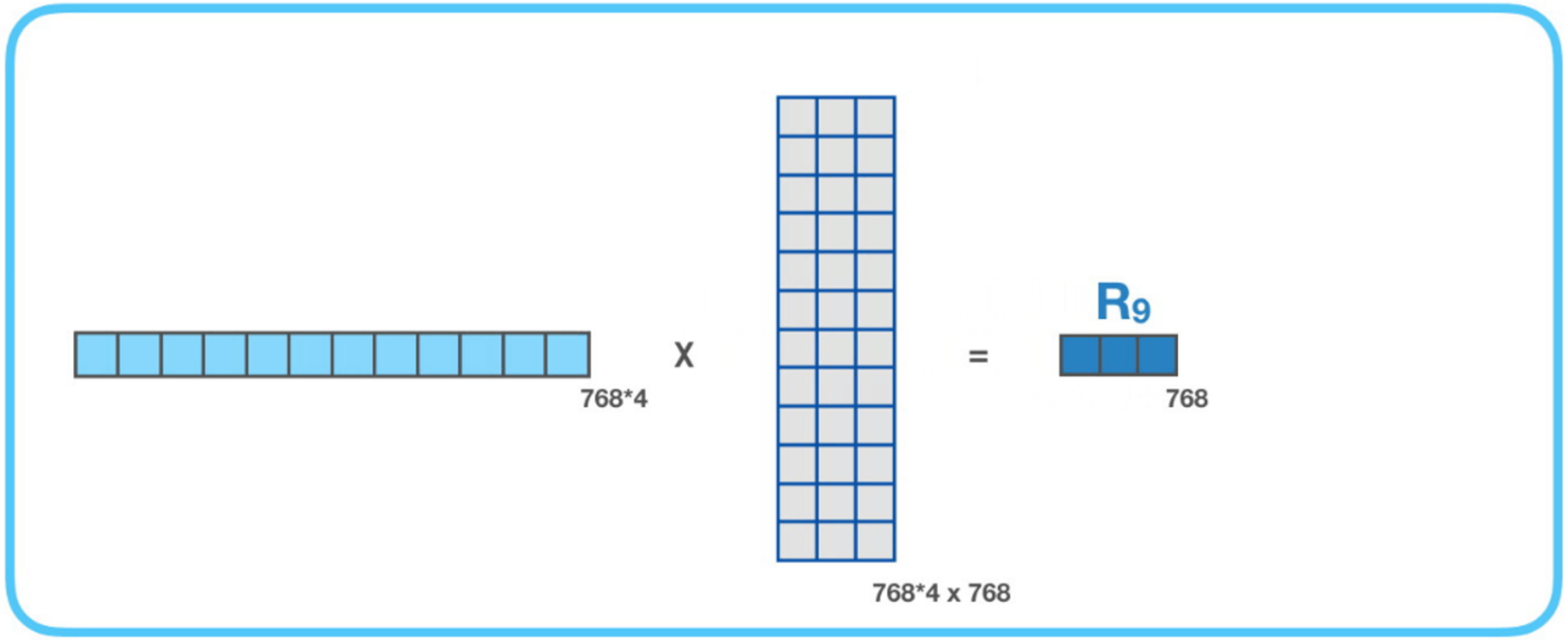
Fully Connected Neural Network

The fully connected neural network is where the block processes its input token after self-attention has included the appropriate context in its representation. It is made up of two layers. **This seems to give transformer models enough representational capacity to handle the tasks that have been thrown at them so far.**



Fully Connected Neural Network

The second layer projects the result from the first layer back into model dimension. **The result of this multiplication is the result of the transformer block for this token.**



Generator

The purpose of the SoftMax is to convert the values in the third tensor dimension into a probability distribution. This tensor of probability distributions is what is returned to the user.

The output contains multiple distributions; one distribution that predicts the token that follows the first input token, another distribution that predicts the token that follows the first and second input tokens, and so on.

The very last probability distribution of each batch of outputs guides the prediction of tokens that follow the entire input sequence.

The Generator is the last piece in the Transformer architecture.

References

- https://bea.stollnitz.com/blog/gpt-transformer/
- https://jalammar.github.io/illustrated-gpt2/
- https://dugas.ch/artificial_curiosity/GPT_architecture.html
- Videos from AI Coffee Break with Letita
- Videos from CodeEmporium

Thank You

Dr. Shylaja S S

Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)

Department of Computer Science and Engineering

shylaja.sharath@pes.edu

Ack: Devang Saraogi,
Teaching Assistant