The Illustrated GPT-2 (Visualizing Transformer Language Models)

Discussions: Hacker News (64 points, 3 comments), Reddit r/MachineLearning (219 points, 18 comments)

This year, we saw a dazzling application of machine learning. The OpenAI GPT-2 exhibited impressive ability of writing coherent and passionate essays that exceed what we anticipated current language models are able to produce. The GPT-2 wasn't a particularly novel architecture – it's architecture is very similar to the decoder-only transformer. The GPT2 was, however, a very large, transformer-based language model trained on a massive dataset. In this post, we'll look at the architecture that enabled the model to produce its results. We will go into the depths of its self-attention layer. And then we'll look at applications for the decoder-only transformer beyond language modeling.

My goal here is to also supplement my earlier post, The Illustrated Transformer, with more visuals explaining the inner-workings of transformers, and how they've evolved since the original paper. My hope is that this visual language will hopefully make it easier to explain later Transformer-based models as their inner-workings continue to evolve.

Contents

- Part 1: GPT2 And Language Modeling
  - What is a Language Model
  - Transformers for Language Modeling
  - One Difference From BERT
  - The Evolution of The Transformer Block
  - Crash Course in Brain Surgery: Looking Inside GPT-2
  - A Deeper Look Inside
  - End of part #1: The GPT-2, Ladies and Gentlemen
- Part 2: The Illustrated Self-Attention
  - Self-Attention (without masking)

    - 1- Create Query, Key, and Value Vectors
    - 2- Score
    - 3- Sum
    - The Illustrated Masked Self-Attention
    - GPT-2 Masked Self-Attention
    - Beyond Language modeling
    - You've Made it!
- **Part 3: Beyond Language Modeling**
    - Machine Translation
    - Summarization
    - Transfer Learning
    - Music Generation

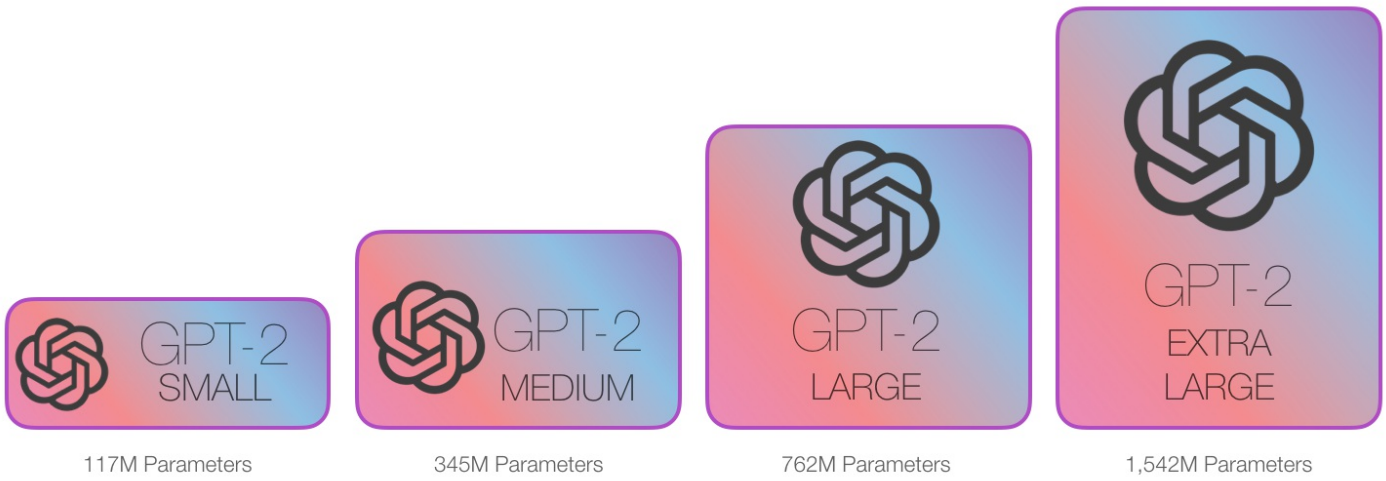
Part #1: GPT2 And Language Modeling

So what exactly is a language model?

What is a Language Model

In The Illustrated Word2vec, we've looked at what a language model is – basically a machine learning model that is able to look at part of a sentence and predict the next word. The most famous language models are smartphone keyboards that suggest the next word based on what you've currently typed.

In this sense, we can say that the GPT-2 is basically the next word prediction feature of a keyboard app, but one that is much larger and more sophisticated than what your phone has. The GPT-2 was trained on a massive 40GB dataset called WebText that the OpenAI researchers crawled from the internet as part of the research effort. To compare in terms of storage size, the keyboard app I use, SwiftKey, takes up 78MBs of space. The smallest variant of the trained GPT-2, takes up 500MBs of storage to store all of its parameters. The largest GPT-2 variant is 13 times the size so it could take up more than 6.5 GBs of storage space.

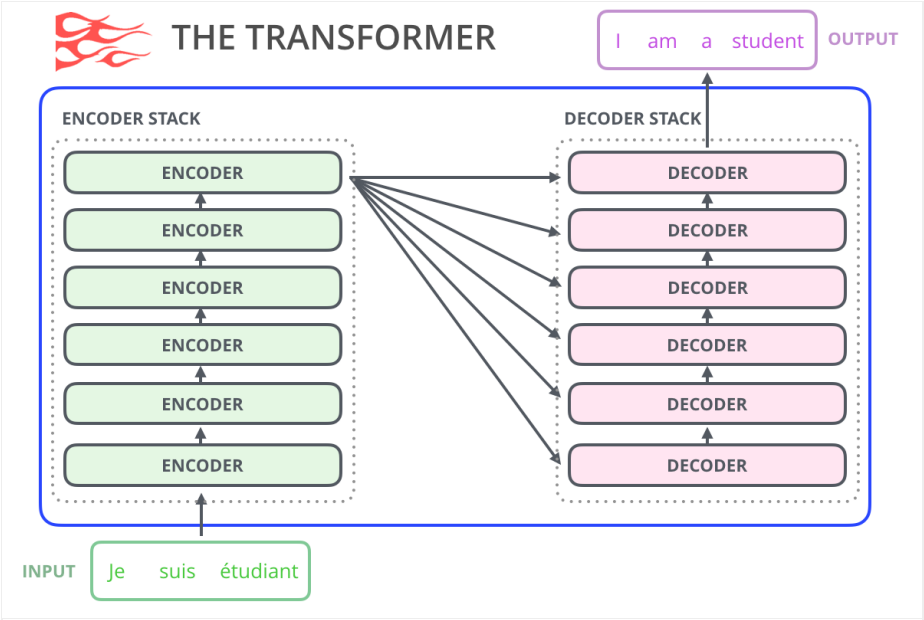


One great way to experiment with GPT-2 is using the AllenAI GPT-2 Explorer. It uses GPT-2 to display ten

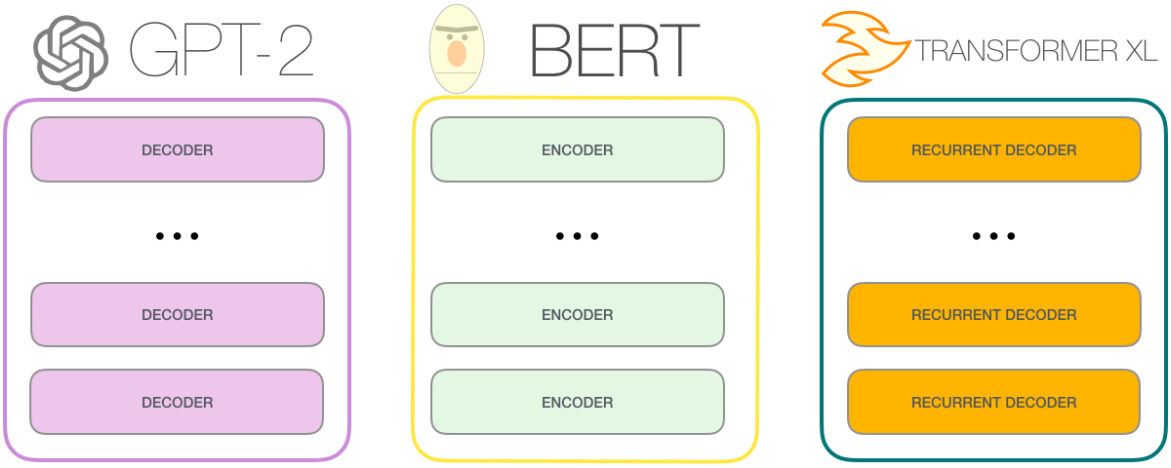
possible predictions for the next word (alongside their probability score). You can select a word then see the next list of predictions to continue writing the passage.

Transformers for Language Modeling

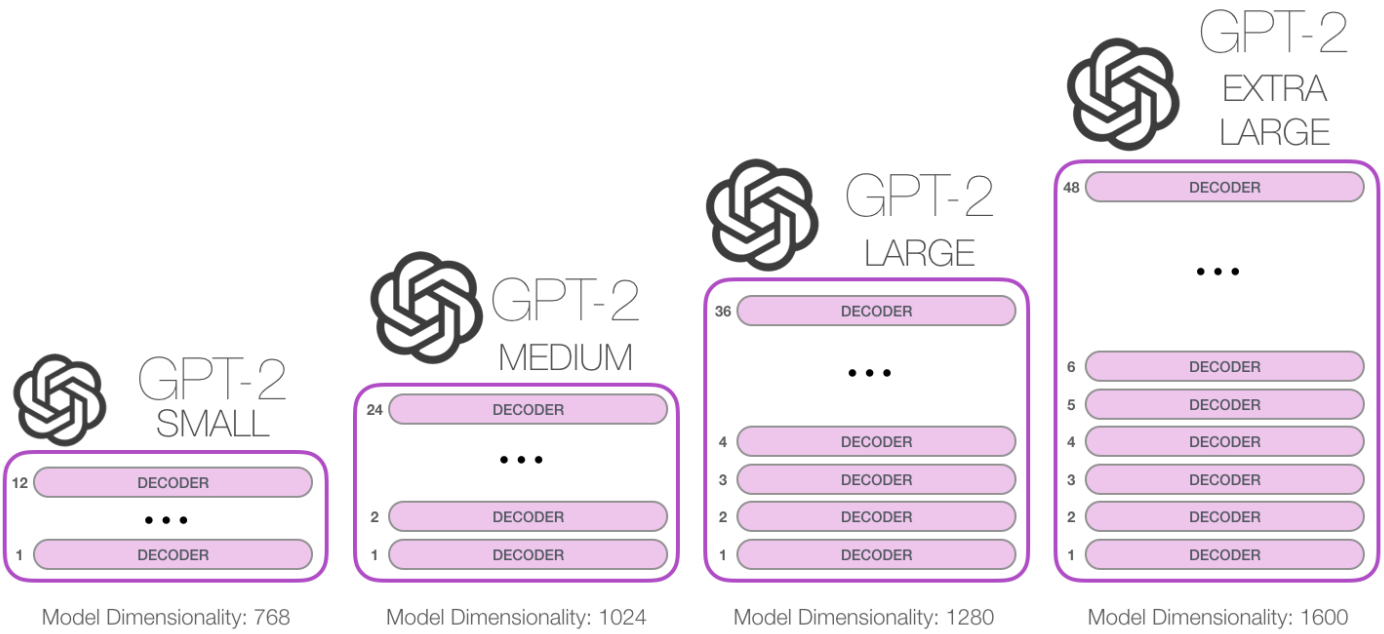
As we've seen in The Illustrated Transformer, the original transformer model is made up of an encoder and decoder – each is a stack of what we can call transformer blocks. That architecture was appropriate because the model tackled machine translation – a problem where encoder-decoder architectures have been successful in the past.



A lot of the subsequent research work saw the architecture shed either the encoder or decoder, and use just one stack of transformer blocks – stacking them up as high as practically possible, feeding them massive amounts of training text, and throwing vast amounts of compute at them (hundreds of thousands of dollars to train some of these language models, likely millions in the case of AlphaStar).



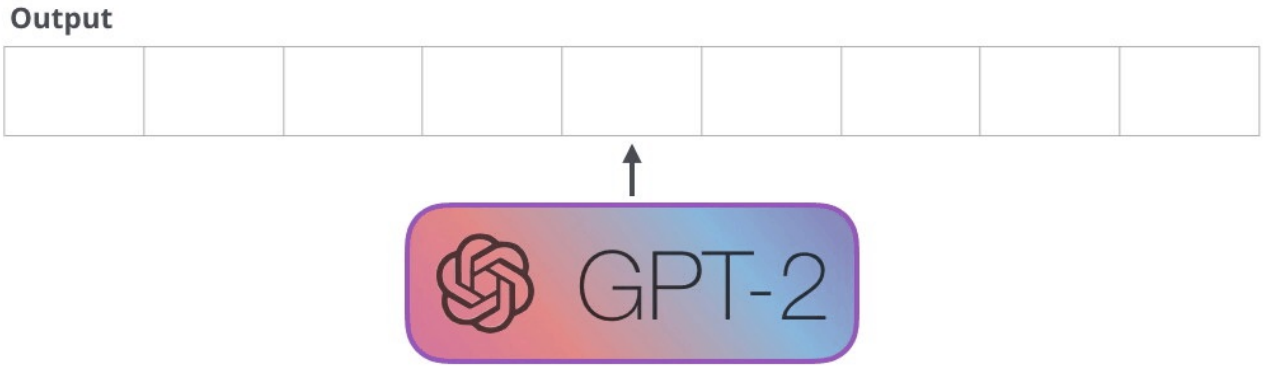
How high can we stack up these blocks? It turns out that's one of the main distinguishing factors between the different GPT2 model sizes:



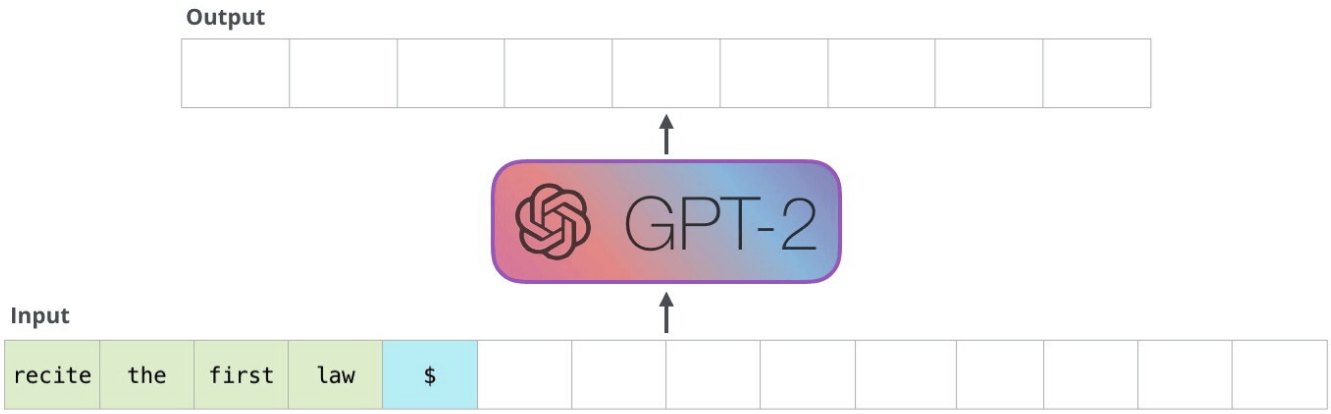
One Difference From BERT

> ***First Law of Robotics***
> *A robot may not injure a human being or, through inaction, allow a human being to come to harm.*

The GPT-2 is built using transformer decoder blocks. BERT, on the other hand, uses transformer encoder blocks. We will examine the difference in a following section. But one key difference between the two is that GPT2, like traditional language models, outputs one token at a time. Let's for example prompt a well-trained GPT-2 to recite the first law of robotics:



The way these models actually work is that after each token is produced, that token is added to the sequence of inputs. And that new sequence becomes the input to the model in its next step. This is an idea called "auto-regression". This is one of the ideas that made RNNs unreasonably effective.



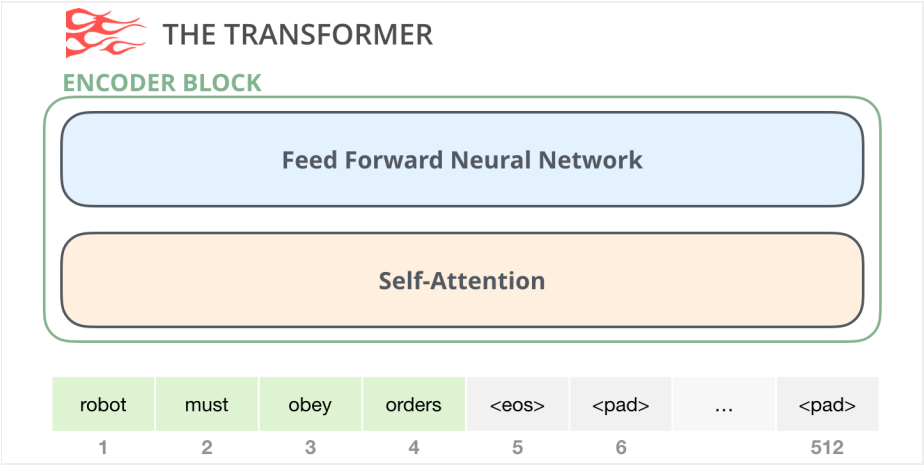
The GPT2, and some later models like TransformerXL and XLNet are auto-regressive in nature. BERT is not. That is a trade off. In losing auto-regression, BERT gained the ability to incorporate the context on both sides of a word to gain better results. XLNet brings back autoregression while finding an alternative way to incorporate the context on both sides.

The Evolution of the Transformer Block

The initial transformer paper introduced two types of transformer blocks:

The Encoder Block

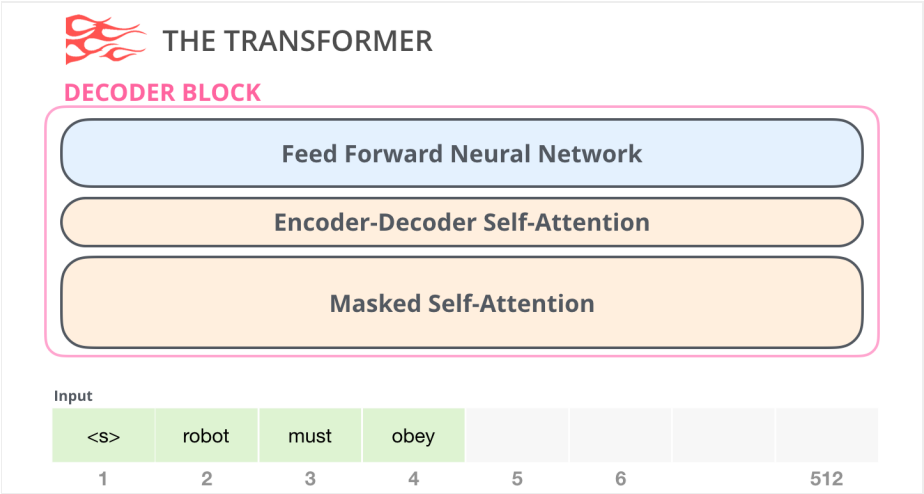
First is the encoder block:



An encoder block from the original transformer paper can take inputs up until a certain max sequence length (e.g. 512 tokens). It's okay if an input sequence is shorter than this limit, we can just pad the rest of the sequence.

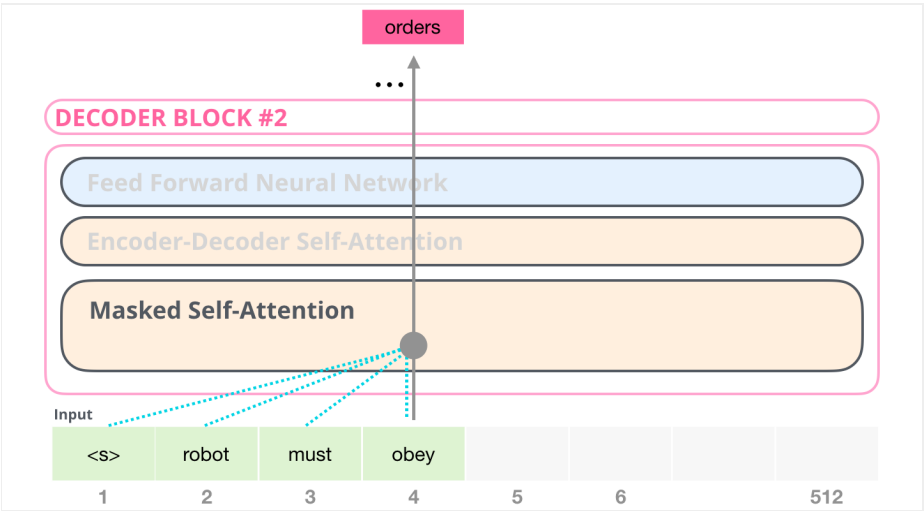
The Decoder Block

Second, there's the decoder block which has a small architectural variation from the encoder block – a layer to allow it to pay attention to specific segments from the encoder:

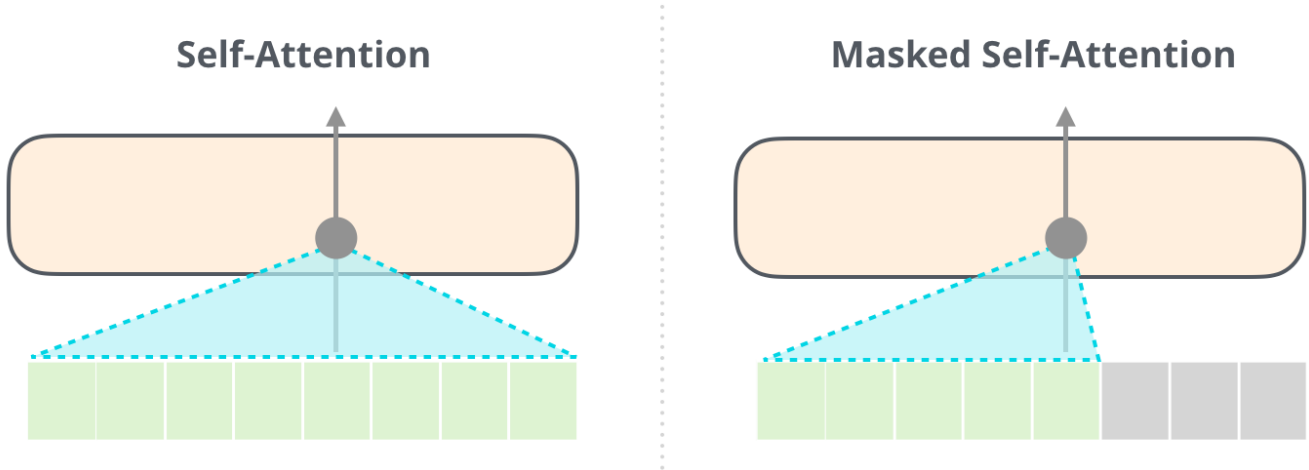


One key difference in the self-attention layer here, is that it masks future tokens – not by changing the word to [mask] like BERT, but by interfering in the self-attention calculation blocking information from tokens that are to the right of the position being calculated.

If, for example, we're to highlight the path of position #4, we can see that it is only allowed to attend to the present and previous tokens:



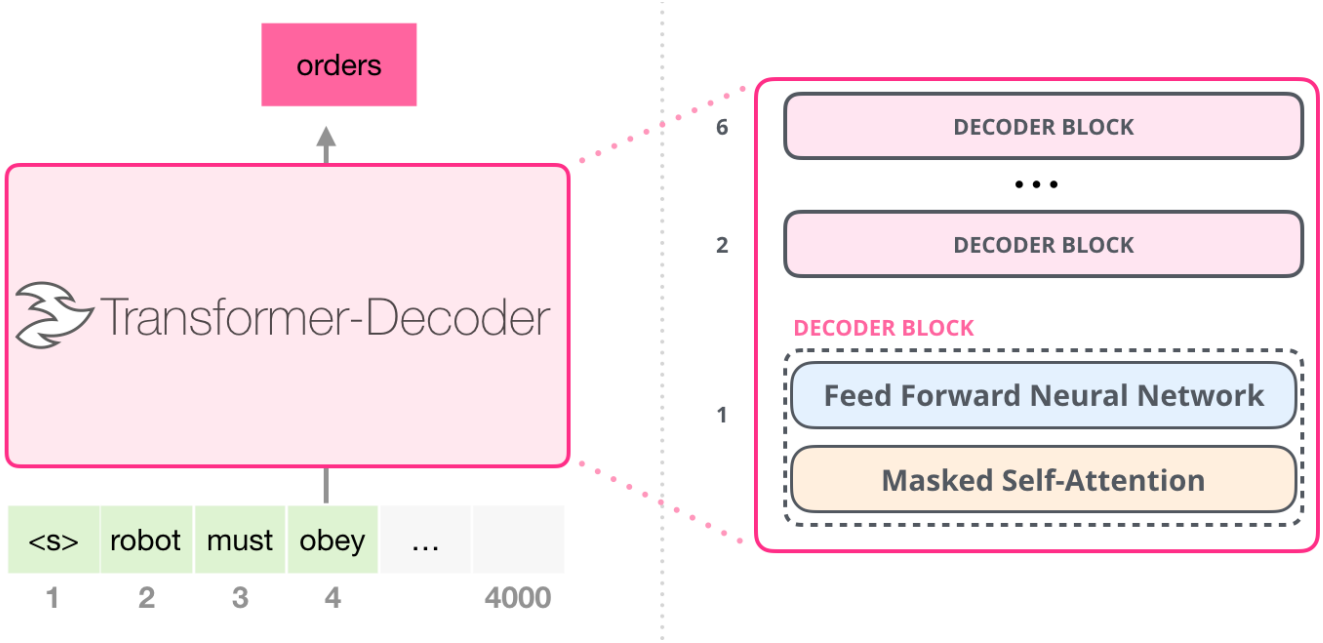
It's important that the distinction between self-attention (what BERT uses) and masked self-attention (what GPT-2 uses) is clear. A normal self-attention block allows a position to peak at tokens to its right. Masked self-attention prevents that from happening:



The Decoder-Only Block

Subsequent to the original paper, Generating Wikipedia by Summarizing Long Sequences proposed another arrangement of the transformer block that is capable of doing language modeling. This model threw away the

Transformer encoder. For that reason, let's call the model the "Transformer-Decoder". This early transformer-based language model was made up of a stack of six transformer decoder blocks:



The decoder blocks are identical. I have expanded the first one so you can see its self-attention layer is the masked variant. Notice that the model now can address up to 4,000 tokens in a certain segment -- a massive upgrade from the 512 in the original transformer.

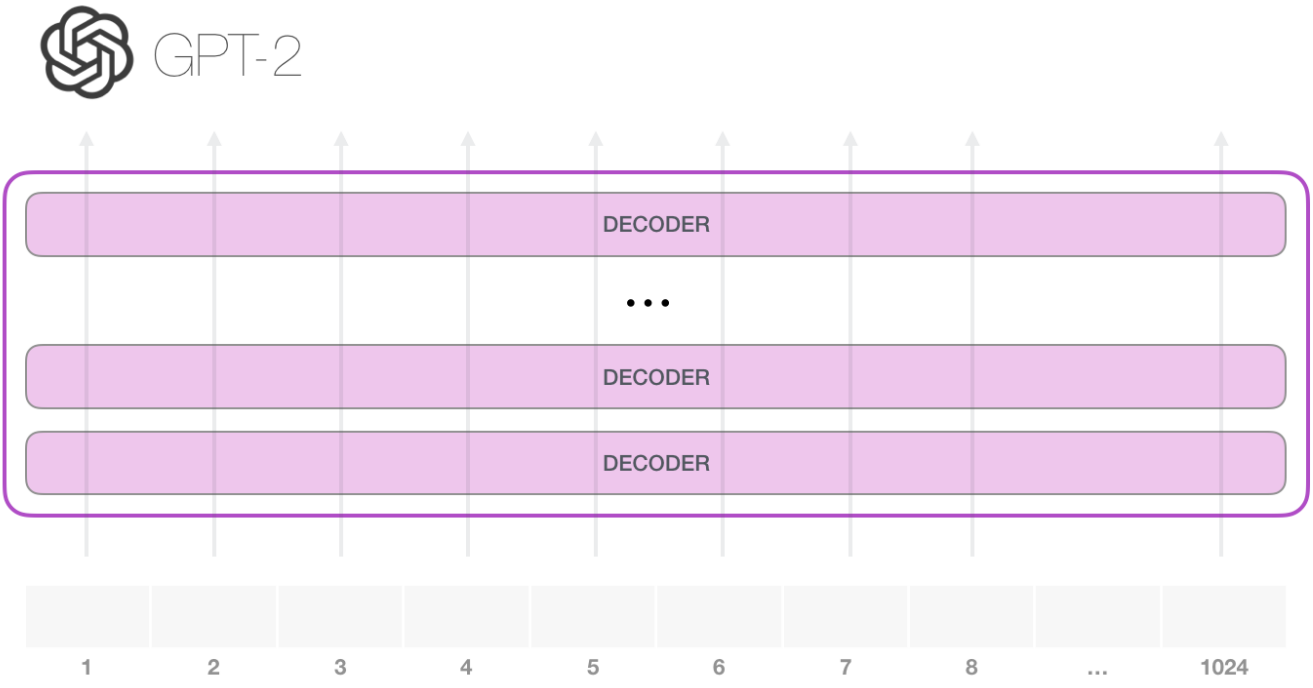
These blocks were very similar to the original decoder blocks, except they did away with that second self-attention layer. A similar architecture was examined in Character-Level Language Modeling with Deeper Self-Attention to create a language model that predicts one letter/character at a time.

The OpenAI GPT-2 model uses these decoder-only blocks.

Crash Course in Brain Surgery: Looking Inside GPT-2

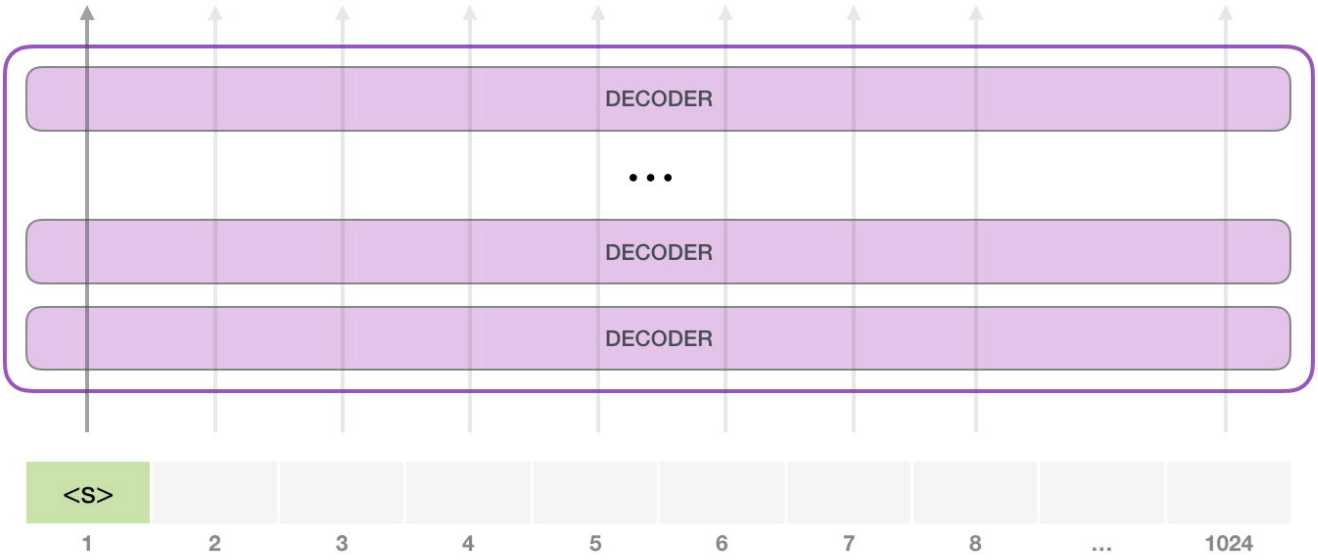
> *Look inside and you will see, The words are cutting deep inside my brain. Thunder burning, quickly burning, Knife of words is driving me insane, insane yeah. ~**Budgie***

Let's lay a trained GPT-2 on our surgery table and look at how it works.



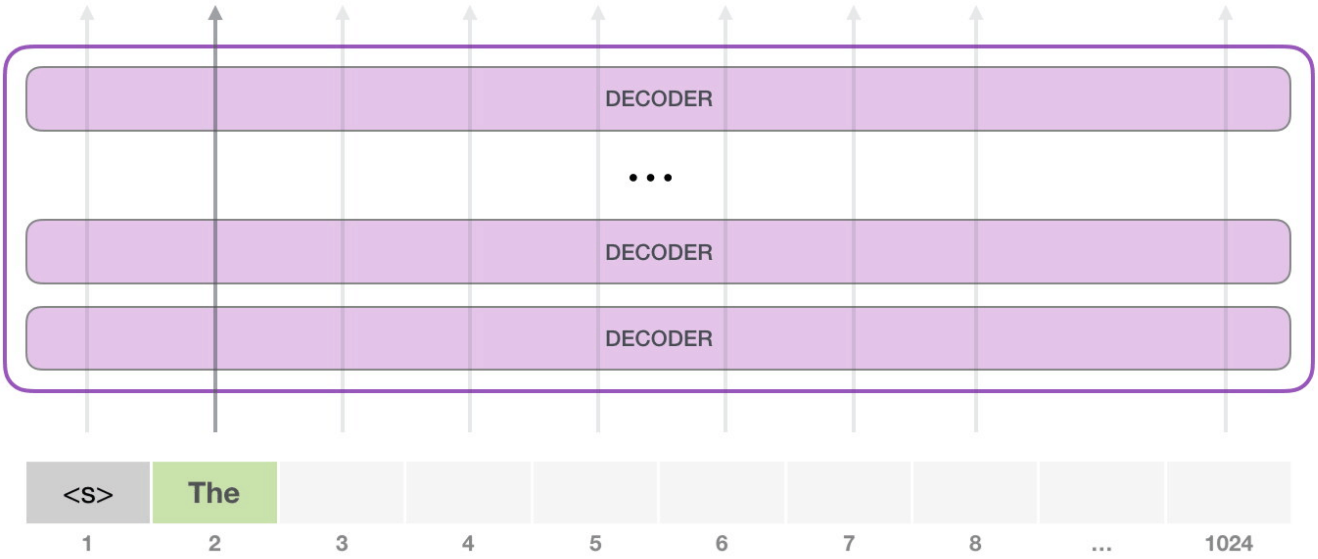
The GPT-2 can process 1024 tokens. Each token flows through all the decoder blocks along its own path.

The simplest way to run a trained GPT-2 is to allow it to ramble on its own (which is technically called *generating unconditional samples*) – alternatively, we can give it a prompt to have it speak about a certain topic (a.k.a generating *interactive conditional samples*). In the rambling case, we can simply hand it the start token and have it start generating words (the trained model uses <|endoftext|> as its start token. Let's call it <s> instead).



The model only has one input token, so that path would be the only active one. The token is processed successively through all the layers, then a vector is produced along that path. That vector can be scored against the model's vocabulary (all the words the model knows, 50,000 words in the case of GPT-2). In this case we selected the token with the highest probability, 'the'. But we can certainly mix things up – you know how if you keep clicking the suggested word in your keyboard app, it sometimes can stuck in repetitive loops where the only way out is if you click the second or third suggested word. The same can happen here. GPT-2 has a parameter called top-k that we can use to have the model consider sampling words other than the top word (which is the case when top-k = 1).

In the next step, we add the output from the first step to our input sequence, and have the model make its next prediction:

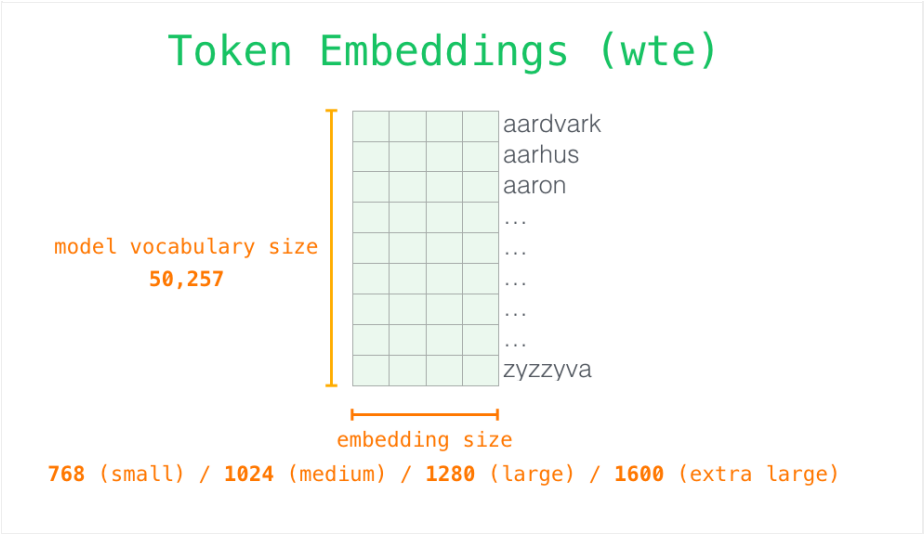


Notice that the second path is the only that's active in this calculation. Each layer of GPT-2 has retained its own interpretation of the first token and will use it in processing the second token (we'll get into more detail about this in the following section about self-attention). GPT-2 does not re-interpret the first token in light of the second token.

A Deeper Look Inside

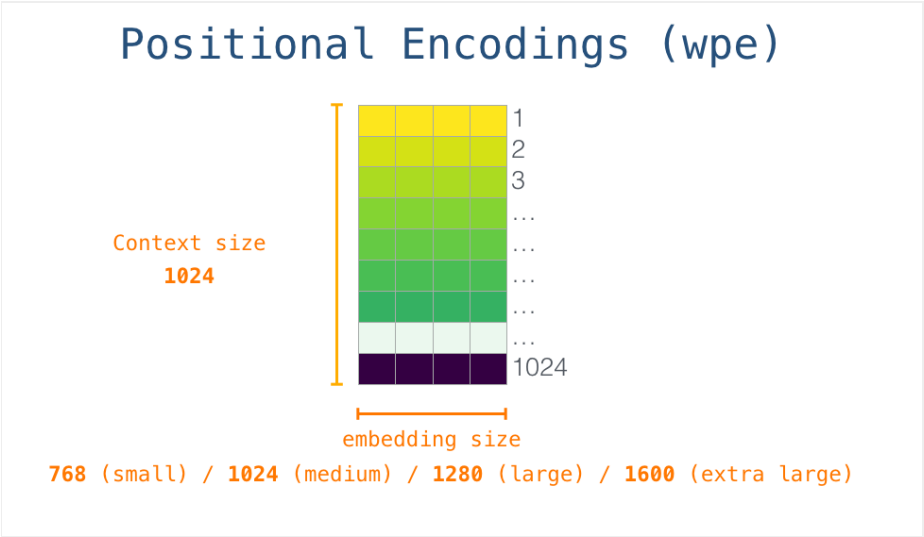
Input Encoding

Let's look at more details to get to know the model more intimately. Let's start from the input. As in other NLP models we've discussed before, the model looks up the embedding of the input word in its embedding matrix – one of the components we get as part of a trained model.



Each row is a word embedding: a list of numbers representing a word and capturing some of its meaning. The size of that list is different in different GPT2 model sizes. The smallest model uses an embedding size of 768 per word/token.

So in the beginning, we look up the embedding of the start token <s> in the embedding matrix. Before handing that to the first block in the model, we need to incorporate positional encoding – a signal that indicates the order of the words in the sequence to the transformer blocks. Part of the trained model is a matrix that contains a positional encoding vector for each of the 1024 positions in the input.

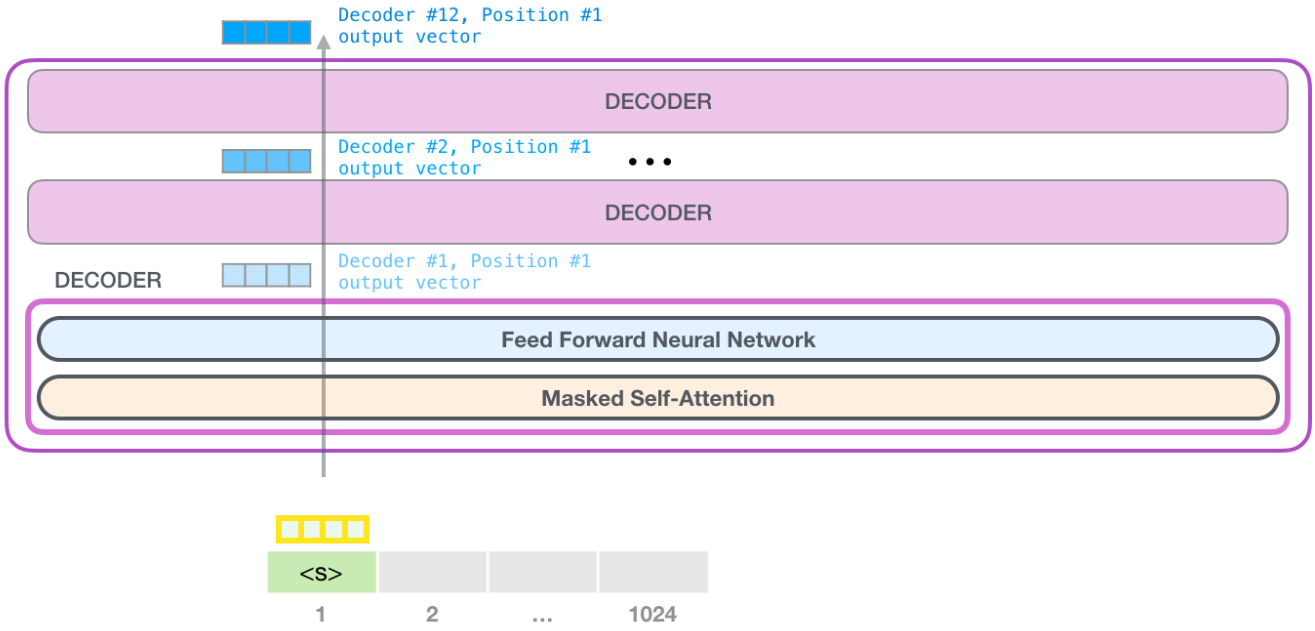


With this, we've covered how input words are processed before being handed to the first transformer block. We also know two of the weight matrices that constitute the trained GPT-2.

Sending a word to the first transformer block means looking up its embedding and adding up the positional encoding vector for position #1.

A journey up the Stack

The first block can now process the token by first passing it through the self-attention process, then passing it through its neural network layer. Once the first transformer block processes the token, it sends its resulting vector up the stack to be processed by the next block. The process is identical in each block, but each block has its own weights in both self-attention and the neural network sublayers.



Self-Attention Recap

Language heavily relies on context. For example, look at the second law:

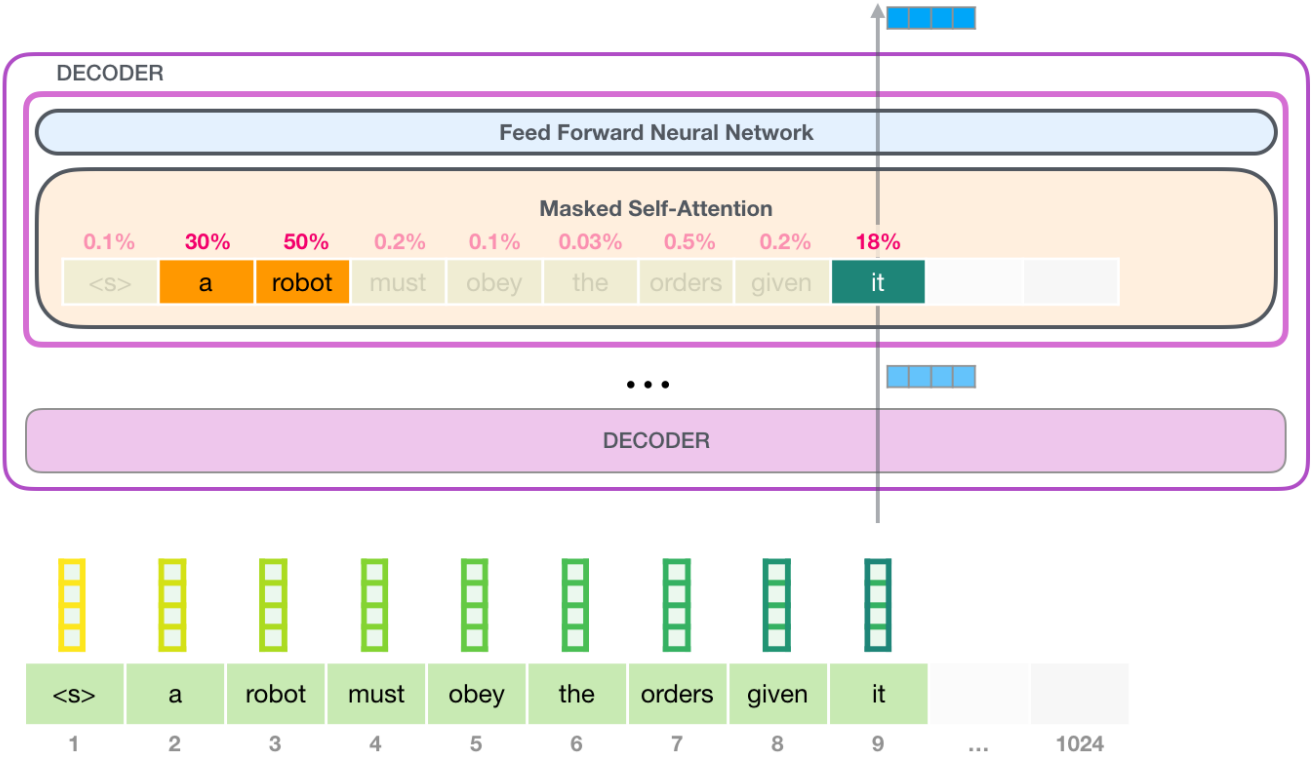
> *Second Law of Robotics*
> *A robot must obey the orders given **it** by human beings except where **such orders** would conflict with the **First Law**.*

I have highlighted three places in the sentence where the words are referring to other words. There is no way to understand or process these words without incorporating the context they are referring to. When a model processes this sentence, it has to be able to know that:

- **it** refers to the robot
- **such orders** refers to the earlier part of the law, namely “the orders given it by human beings”
- **The First Law** refers to the entire First Law

This is what self-attention does. It bakes in the model’s understanding of relevant and associated words that explain the context of a certain word before processing that word (passing it through a neural network). It does that by assigning scores to how relevant each word in the segment is, and adding up their vector representation.

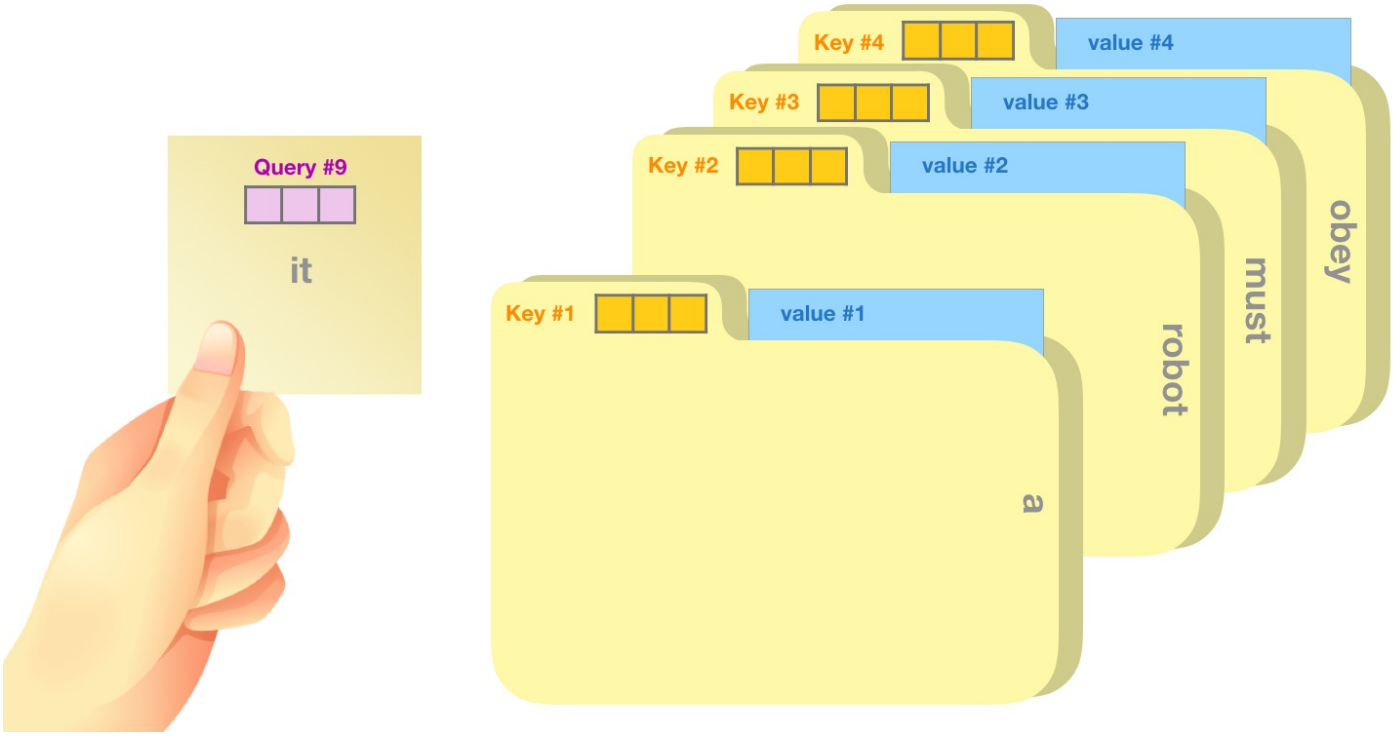
As an example, this self-attention layer in the top block is paying attention to “a robot” when it processes the word “it”. The vector it will pass to its neural network is a sum of the vectors for each of the three words multiplied by their scores.



Self-Attention Process

Self-attention is processed along the path of each token in the segment. The significant components are three vectors:

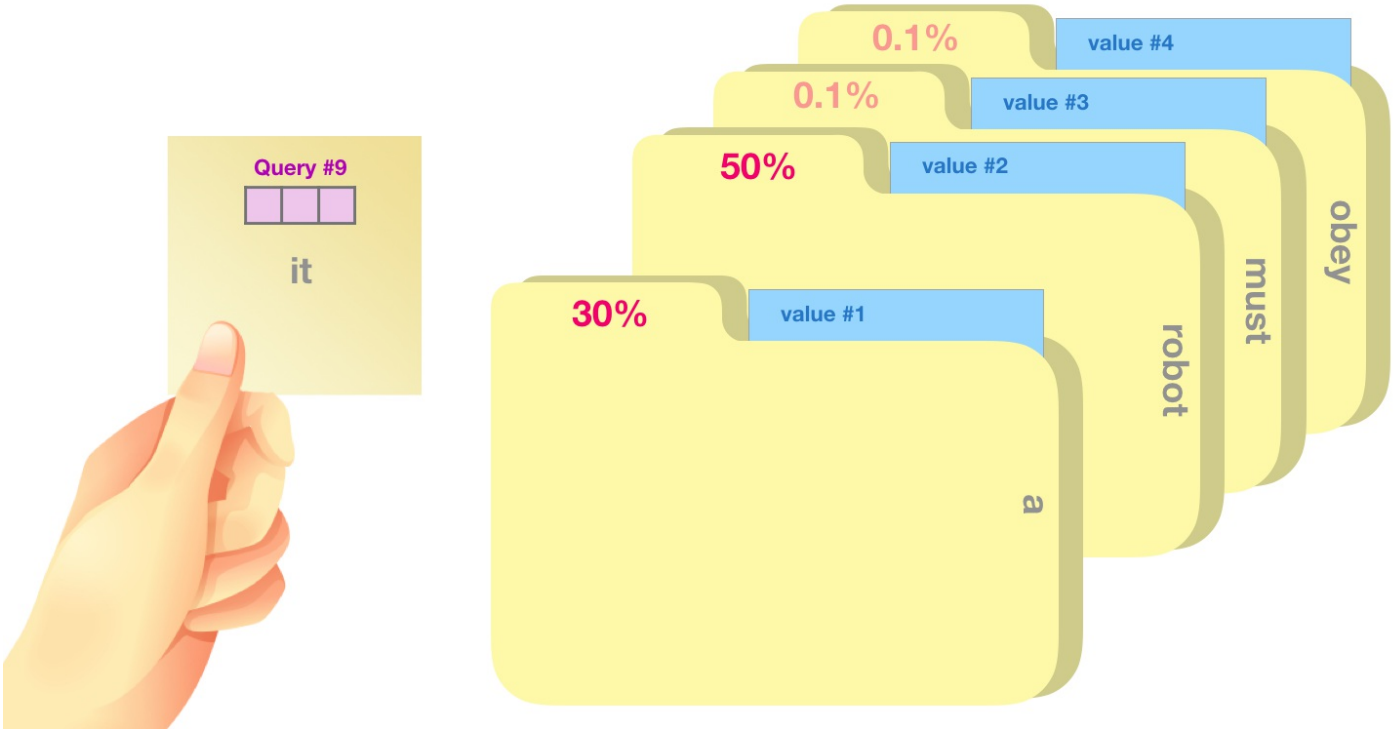
- Query: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we're currently processing.
- Key: Key vectors are like labels for all the words in the segment. They're what we match against in our search for relevant words.
- Value: Value vectors are actual word representations, once we've scored how relevant each word is, these are the values we add up to represent the current word.



A crude analogy is to think of it like searching through a filing cabinet. The query is like a sticky note with the topic

you're researching. The keys are like the labels of the folders inside the cabinet. When you match the tag with a sticky note, we take out the contents of that folder, these contents are the value vector. Except you're not only looking for one value, but a blend of values from a blend of folders.

Multiplying the query vector by each key vector produces a score for each folder (technically: dot product followed by softmax).



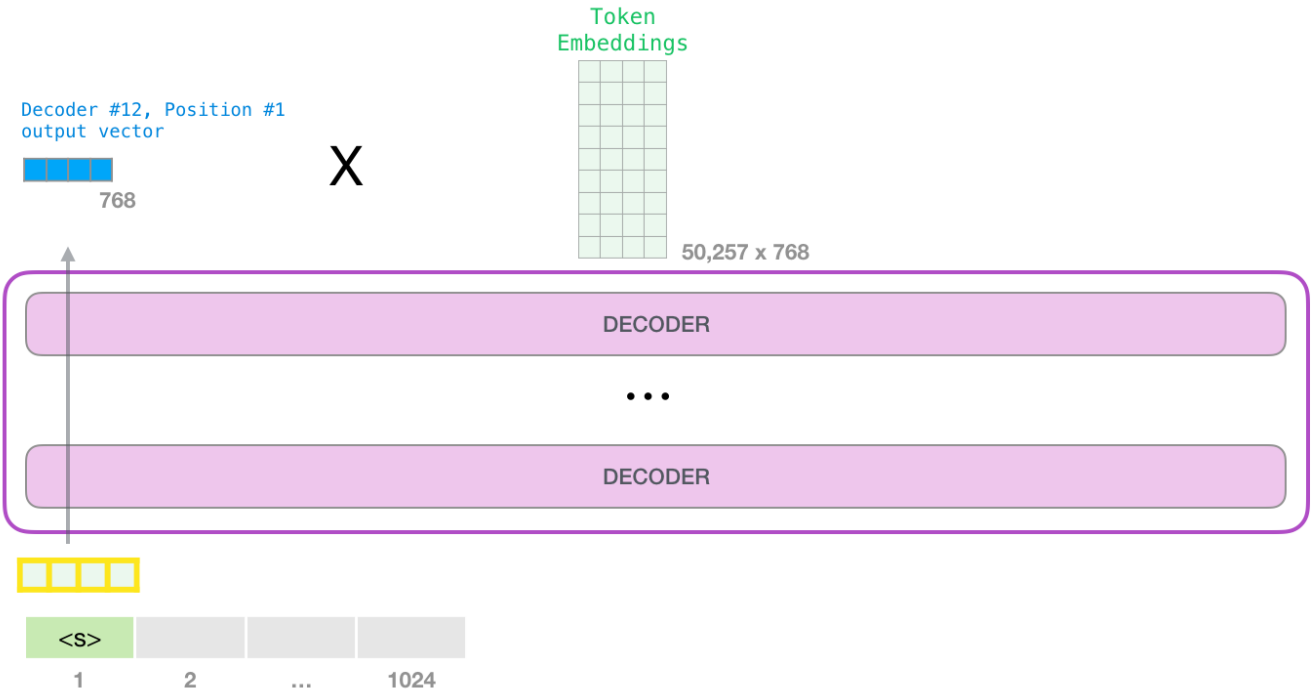
We multiply each value by its score and sum up – resulting in our self-attention outcome.

Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
it		0.19	
		Sum:	

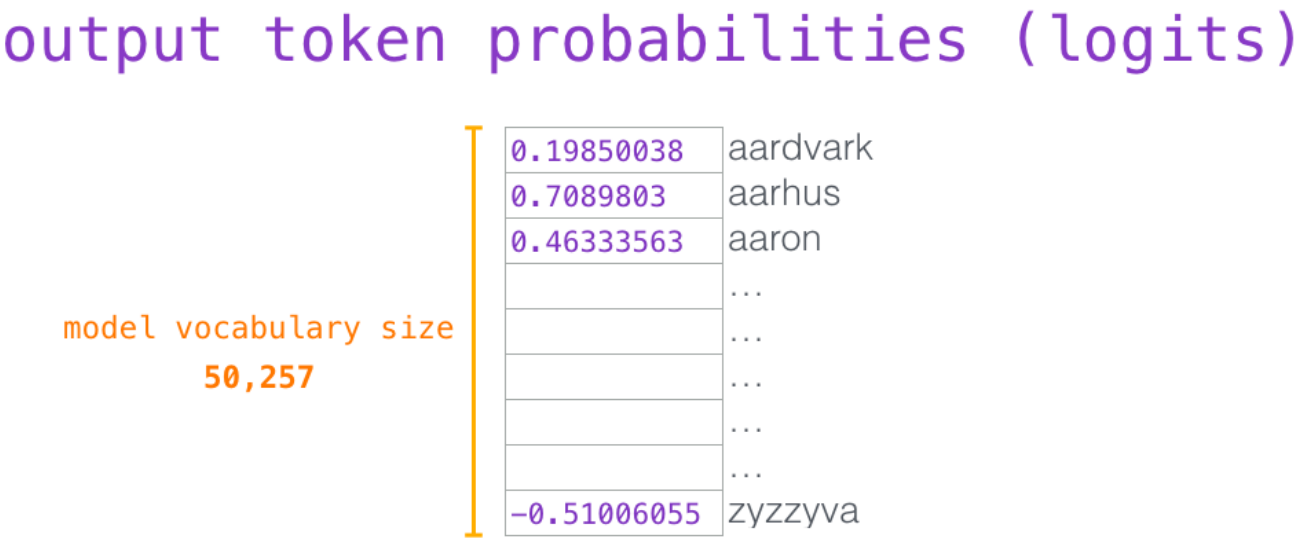
This weighted blend of value vectors results in a vector that paid 50% of its "attention" to the word robot, 30% to the word a, and 19% to the word it. Later in the post, we'll got deeper into self-attention. But first, let's continue our journey up the stack towards the output of the model.

Model Output

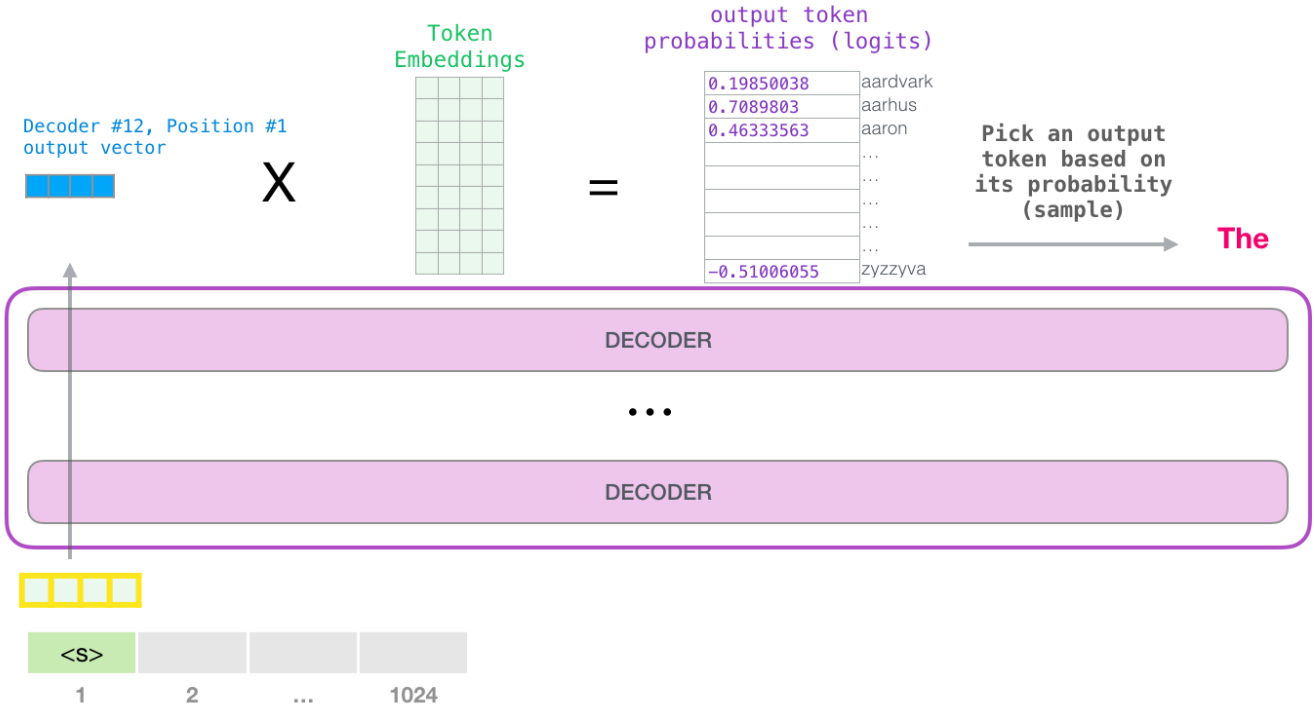
When the top block in the model produces its output vector (the result of its own self-attention followed by its own neural network), the model multiplies that vector by the embedding matrix.



Recall that each row in the embedding matrix corresponds to the embedding of a word in the model's vocabulary. The result of this multiplication is interpreted as a score for each word in the model's vocabulary.



We can simply select the token with the highest score (top_k = 1). But better results are achieved if the model considers other words as well. So a better strategy is to sample a word from the entire list using the score as the probability of selecting that word (so words with a higher score have a higher chance of being selected). A middle ground is setting top_k to 40, and having the model consider the 40 words with the highest scores.



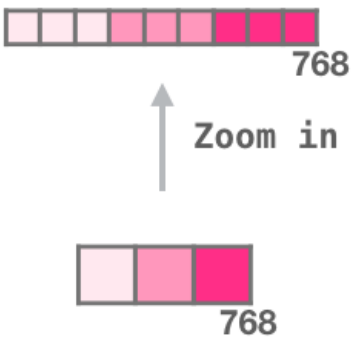
With that, the model has completed an iteration resulting in outputting a single word. The model continues iterating until the entire context is generated (1024 tokens) or until an end-of-sequence token is produced.

End of part #1: The GPT-2, Ladies and Gentlemen

And there we have it. A run down of how the GPT2 works. If you're curious to know exactly what happens inside the self-attention layer, then the following bonus section is for you. I created it to introduce more visual language to describe self-attention in order to make describing later transformer models easier to examine and describe (looking at you, TransformerXL and XLNet).

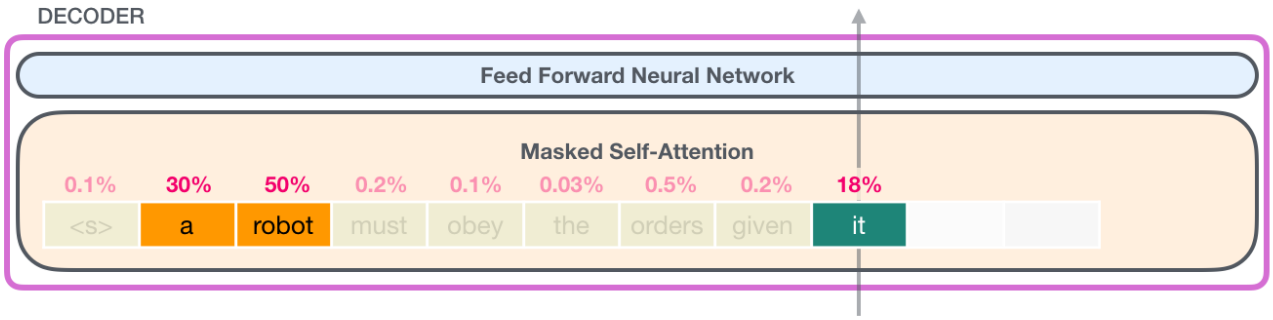
I'd like to note a few oversimplifications in this post:

- I used "words" and "tokens" interchangeably. But in reality, GPT2 uses Byte Pair Encoding to create the tokens in its vocabulary. This means the tokens are usually parts of words.
- The example we showed runs GPT2 in its inference/evaluation mode. That's why it's only processing one word at a time. At training time, the model would be trained against longer sequences of text and processing multiple tokens at once. Also at training time, the model would process larger batch sizes (512) vs. the batch size of one that evaluation uses.
- I took liberties in rotating/transposing vectors to better manage the spaces in the images. At implementation time, one has to be more precise.
- Transformers use a lot of layer normalization, which is pretty important. We've noted a few of these in the Illustrated Transformer, but focused more on self-attentionin this post.
- There are times when I needed to show more boxes to represent a vector. I indicate those as "zooming in". For example:



Part #2: The Illustrated Self-Attention

Earlier in the post we showed this image to showcase self-attention being applied in a layer that is processing the word it:



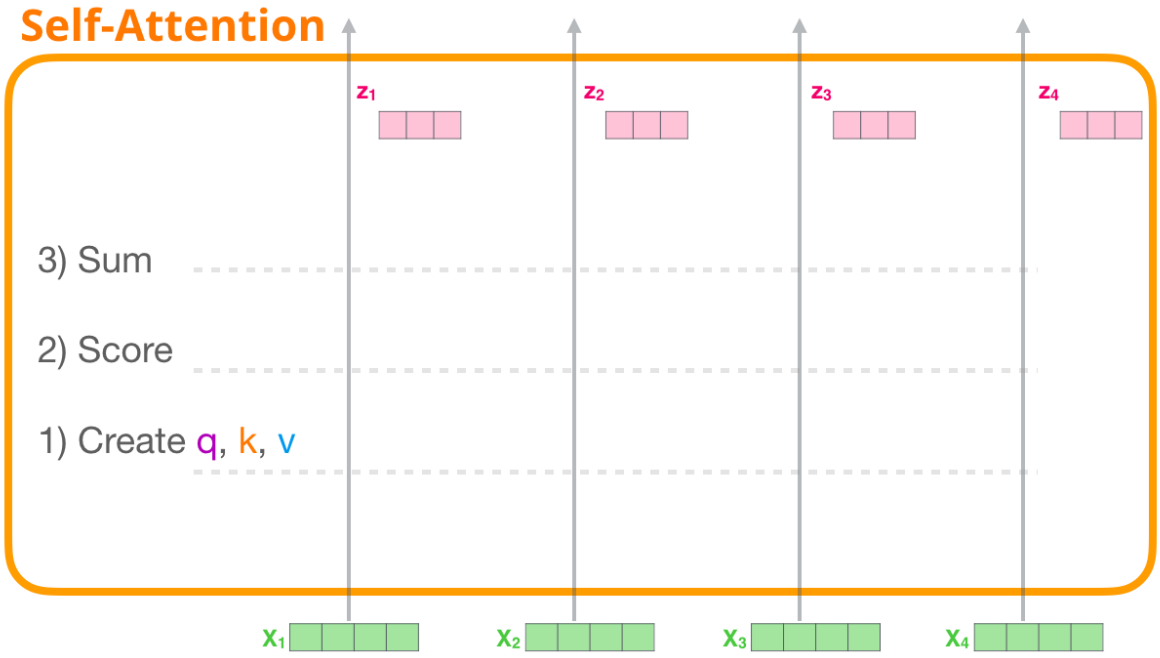
In this section, we'll look at the details of how that is done. Note that we'll look at it in a way to try to make sense of what happens to individual words. That's why we'll be showing many single vectors. The actual implementations are done by multiplying giant matrices together. But I want to focus on the intuition of what happens on a word-level here.

Self-Attention (without masking)

Let's start by looking at the original self-attention as it's calculated in an encoder block. Let's look at a toy transformer block that can only process four tokens at a time.

Self-attention is applied through three main steps:

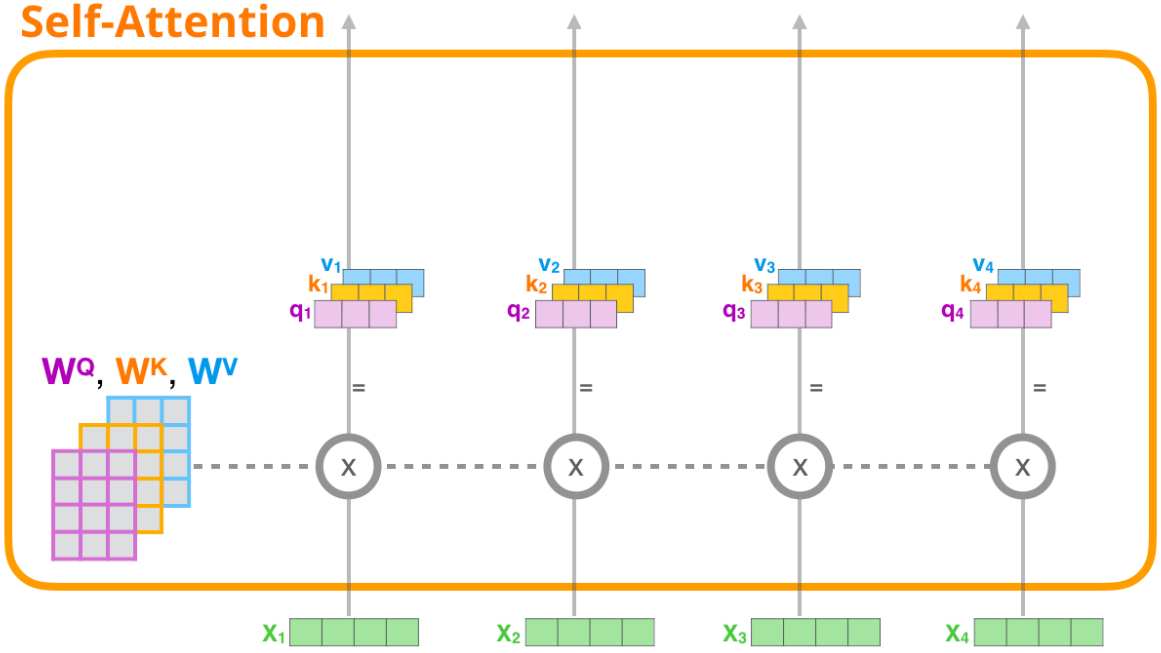
1. Create the Query, Key, and Value vectors for each path.
2. For each input token, use its query vector to score against all the other key vectors
3. Sum up the value vectors after multiplying them by their associated scores.



1- Create Query, Key, and Value Vectors

Let's focus on the first path. We'll take its query, and compare against all the keys. That produces a score for each key. The first step in self-attention is to calculate the three vectors for each token path (let's ignore attention heads for now):

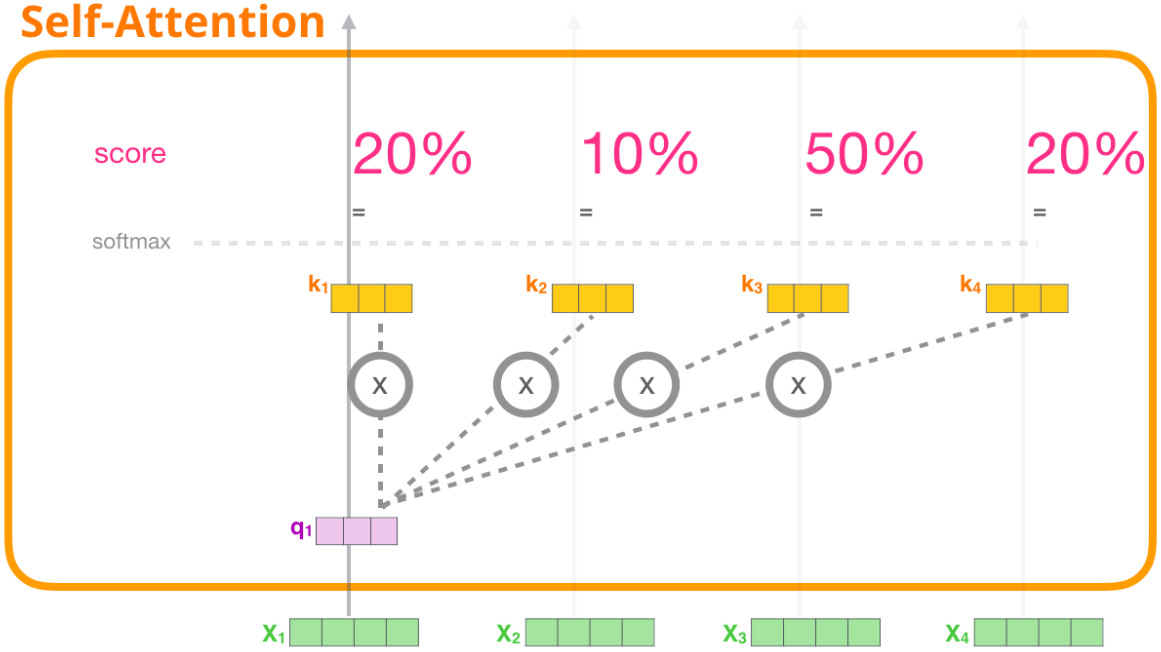
1) For each input token, create a query vector, a key vector, and a value vector by multiplying by weight Matrices W^Q, W^K, W^V



2- Score

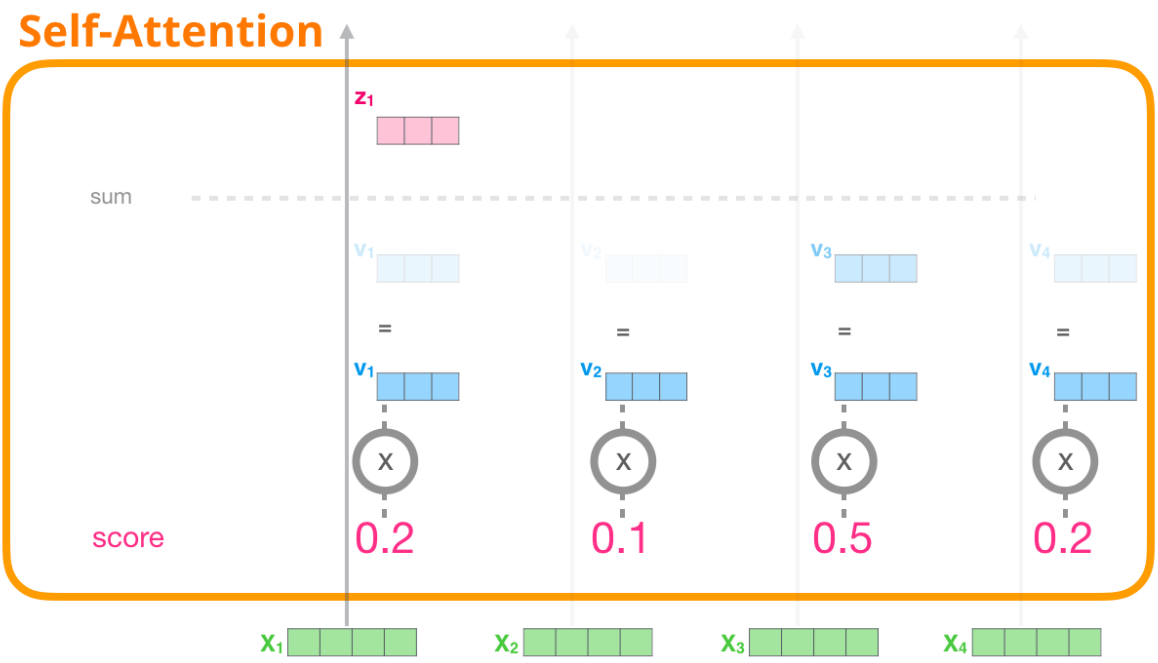
Now that we have the vectors, we use the query and key vectors only for step #2. Since we're focused on the first token, we multiply its query by all the other key vectors resulting in a score for each of the four tokens.

2) Multiply (dot product) the current query vector, by all the key vectors, to get a score of how well they match



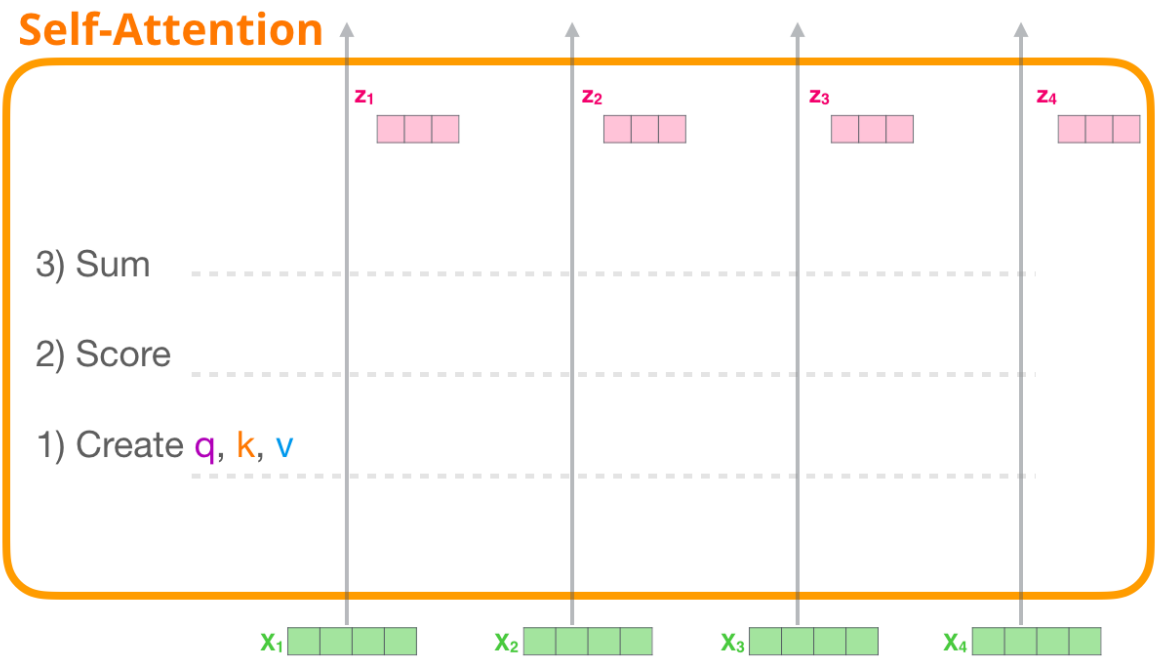
3- Sum

We can now multiply the scores by the value vectors. A value with a high score will constitute a large portion of the resulting vector after we sum them up.



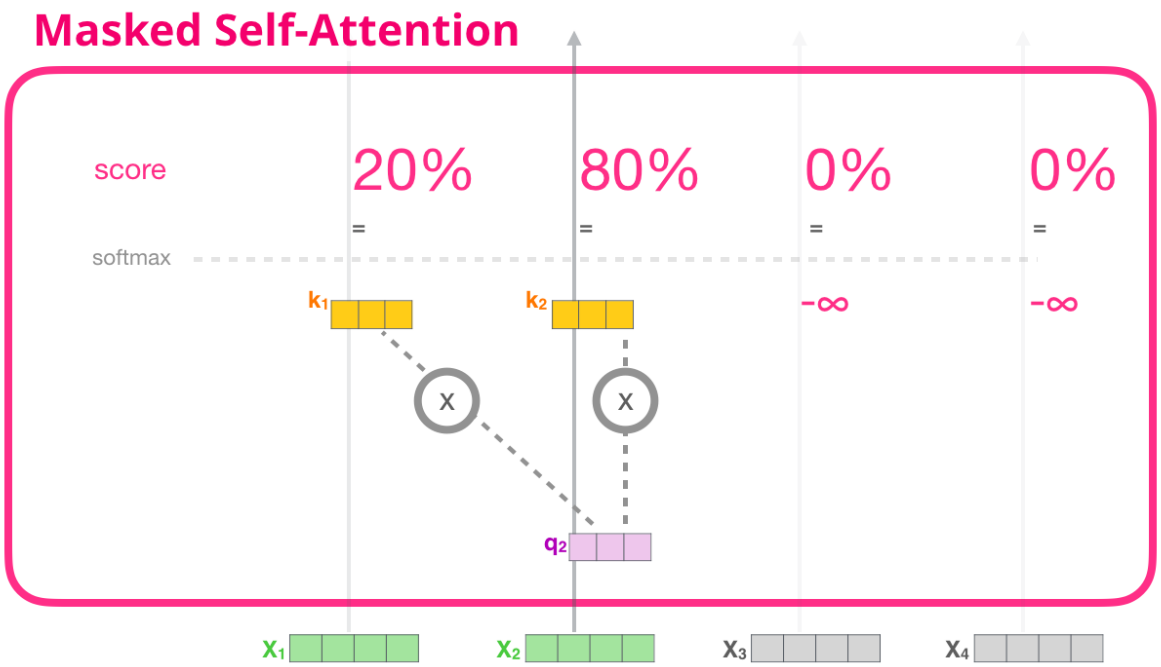
The lower the score, the more transparent we're showing the value vector. That's to indicate how multiplying by a small number dilutes the values of the vector.

If we do the same operation for each path, we end up with a vector representing each token containing the appropriate context of that token. Those are then presented to the next sublayer in the transformer block (the feed-forward neural network):



The Illustrated Masked Self-Attention

Now that we've looked inside a transformer's self-attention step, let's proceed to look at masked self-attention. Masked self-attention is identical to self-attention except when it comes to step #2. Assuming the model only has two tokens as input and we're observing the second token. 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. Think of a sequence of four words ("robot must obey orders", for example). 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
position:	1	2	3	4	
Example: 1	robot	must	obey	orders	must
2	robot	must	obey	orders	obey
3	robot	must	obey	orders	orders
4	robot	must	obey	orders	<eos>

In matrix form, we calculate the scores by multiplying a queries matrix by a keys matrix. Let's visualize it as follows, except instead of the word, there would be the query (or key) vector associated with that word in that cell:

Queries				X	Keys				=	Scores (before softmax)			
robot	must	obey	orders		robot	must	obey	orders		0.11	0.00	0.81	0.79
					robot	must	obey	orders		0.19	0.50	0.30	0.48
					robot	must	obey	orders		0.53	0.98	0.95	0.14
					robot	must	obey	orders		0.81	0.86	0.38	0.90

After the multiplication, we slap on our attention mask triangle. It set the cells we want to mask to -infinity or a very large negative number (e.g. -1 billion in GPT2):

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

Apply Attention Mask

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

Then, applying softmax on each row produces the actual scores we use 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

What this scores table means is the following:

- When the model processes the first example in the dataset (row #1), which contains only one word (“robot”), 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 (“robot must”), when it processes the word “must”, 48% of its attention will be on “robot”, and 52% of its attention will be on “must”.
- And so on

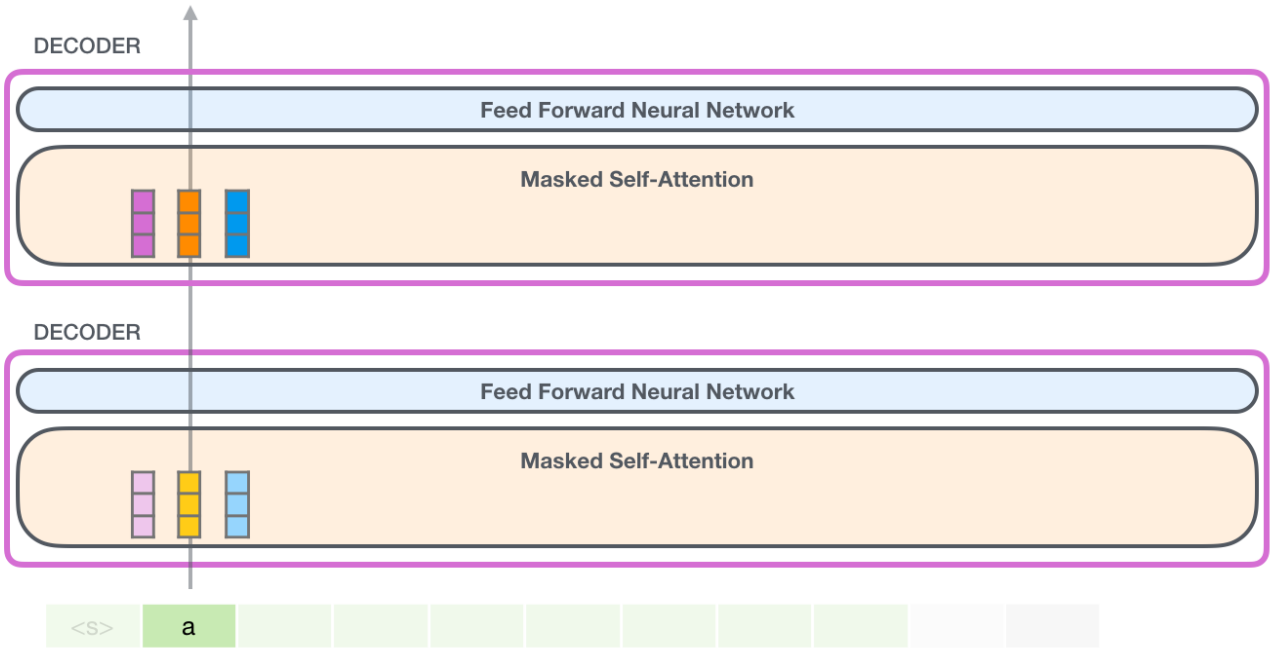
GPT-2 Masked Self-Attention

Let’s get into more detail on GPT-2’s masked attention.

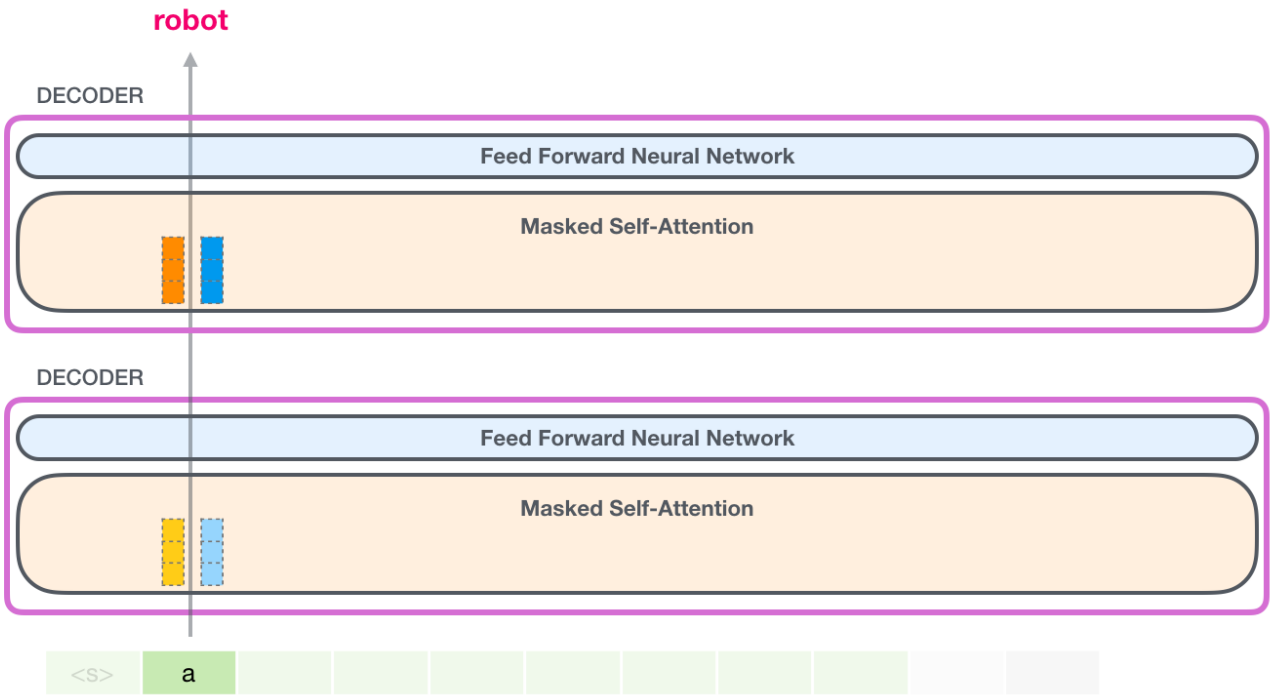
Evaluation Time: Processing One Token at a Time

We can make the GPT-2 operate exactly as masked self-attention works. But during evaluation, when our model is only adding one new word after each iteration, it would be inefficient to recalculate self-attention along earlier paths for tokens which have already been processed.

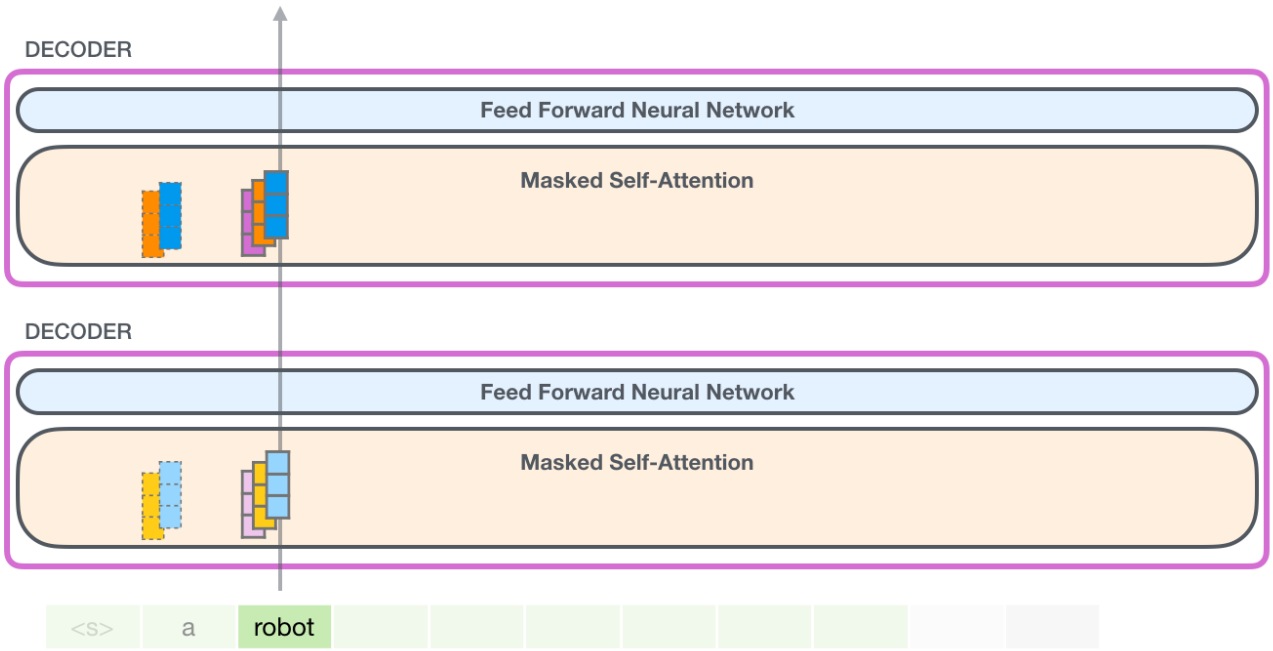
In this case, we process the first token (ignoring `<s>` for now).



GPT-2 holds on to the key and value vectors of the the a token. Every self-attention layer holds on to its respective key and value vectors for that token:

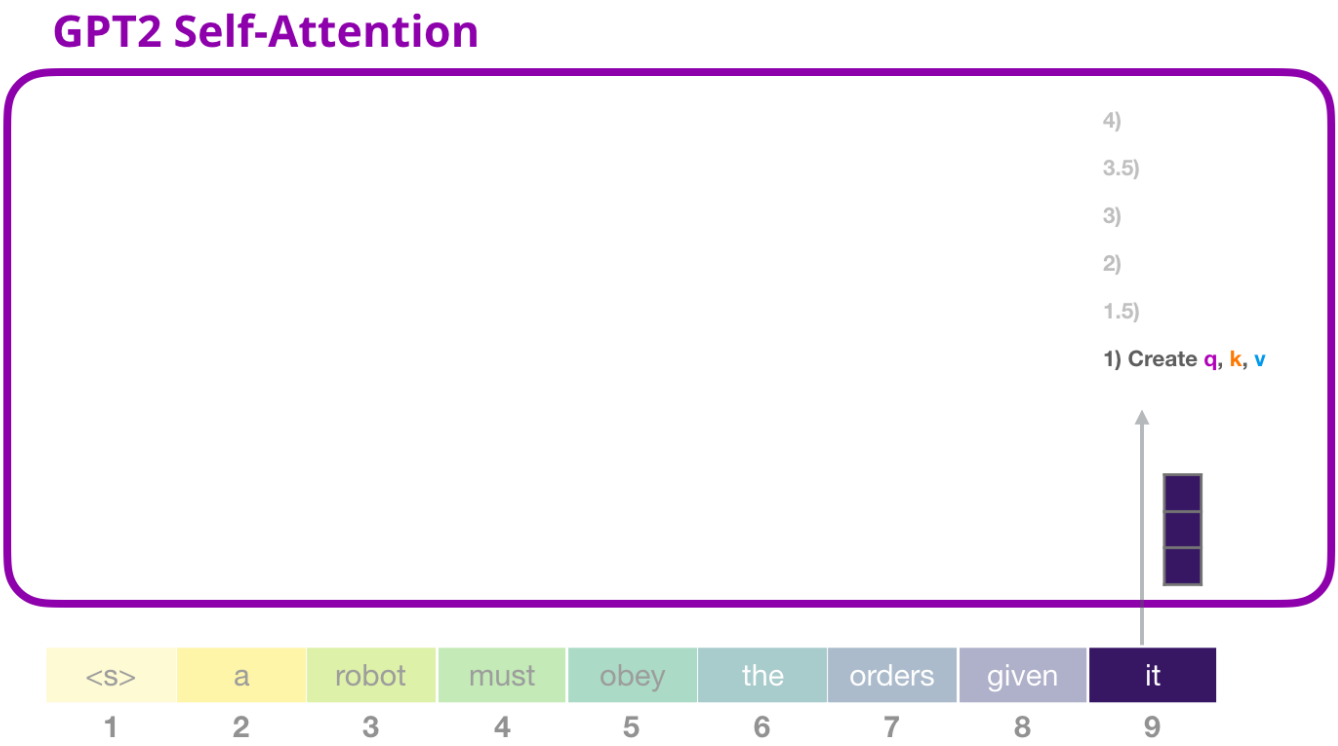


Now in the next iteration, when the model processes the word robot, it does not need to generate query, key, and value queries for the a token. It just reuses the ones it saved from the first iteration:

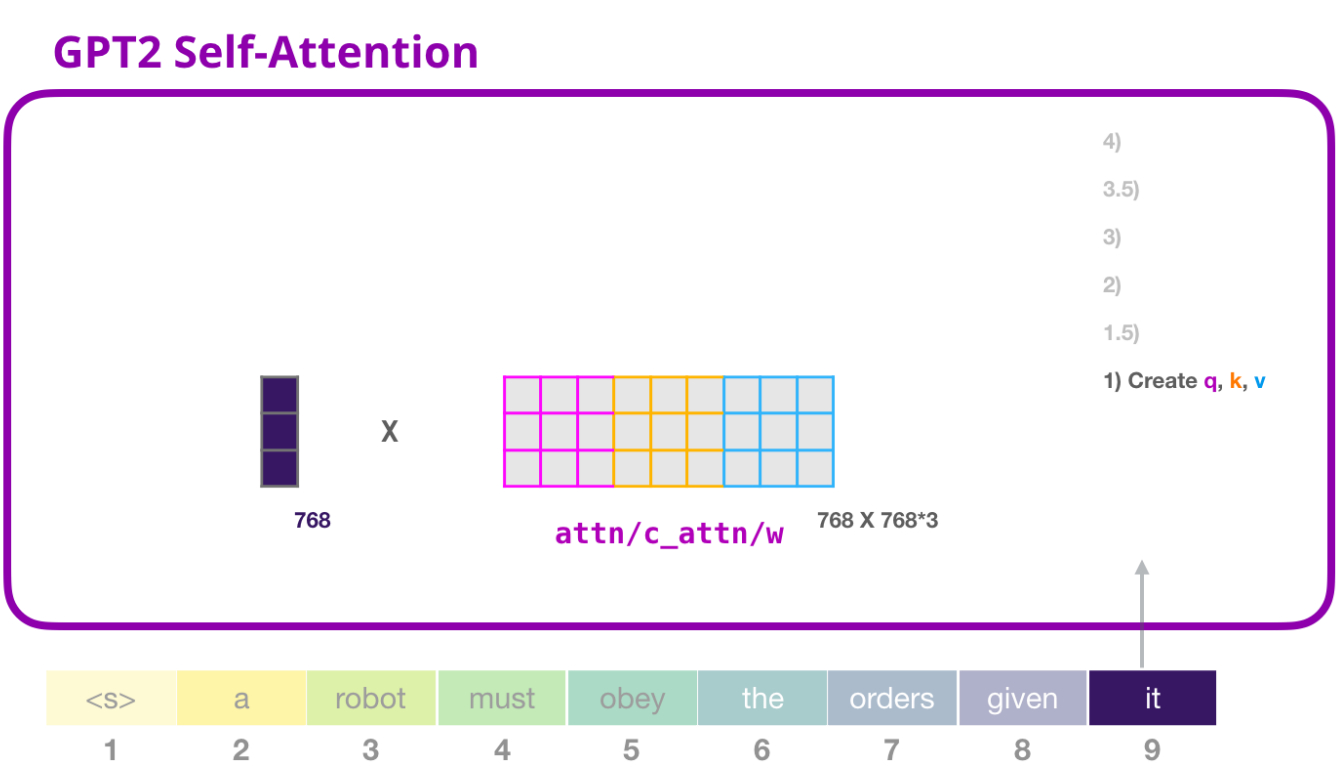


GPT-2 Self-attention: 1- Creating queries, keys, and values

Let's assume the model is processing the word it. If we're talking about the bottom block, then its input for that token would be the embedding of it + the positional encoding for slot #9:

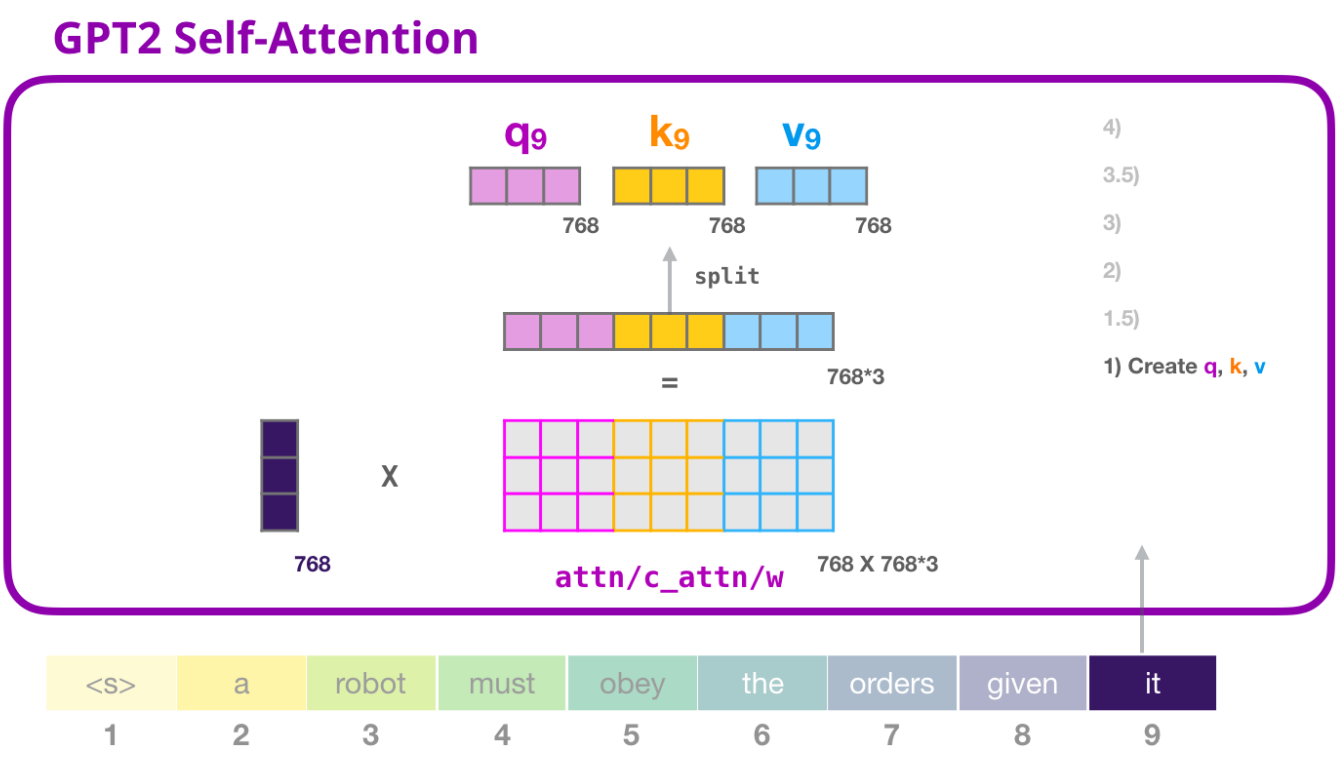


Every block in a transformer has its own weights (broken down later in the post). The first we encounter is the weight matrix that we use to create the queries, keys, and values.



Self-attention multiplies its input by its weight matrix (and adds a bias vector, not illustrated here).

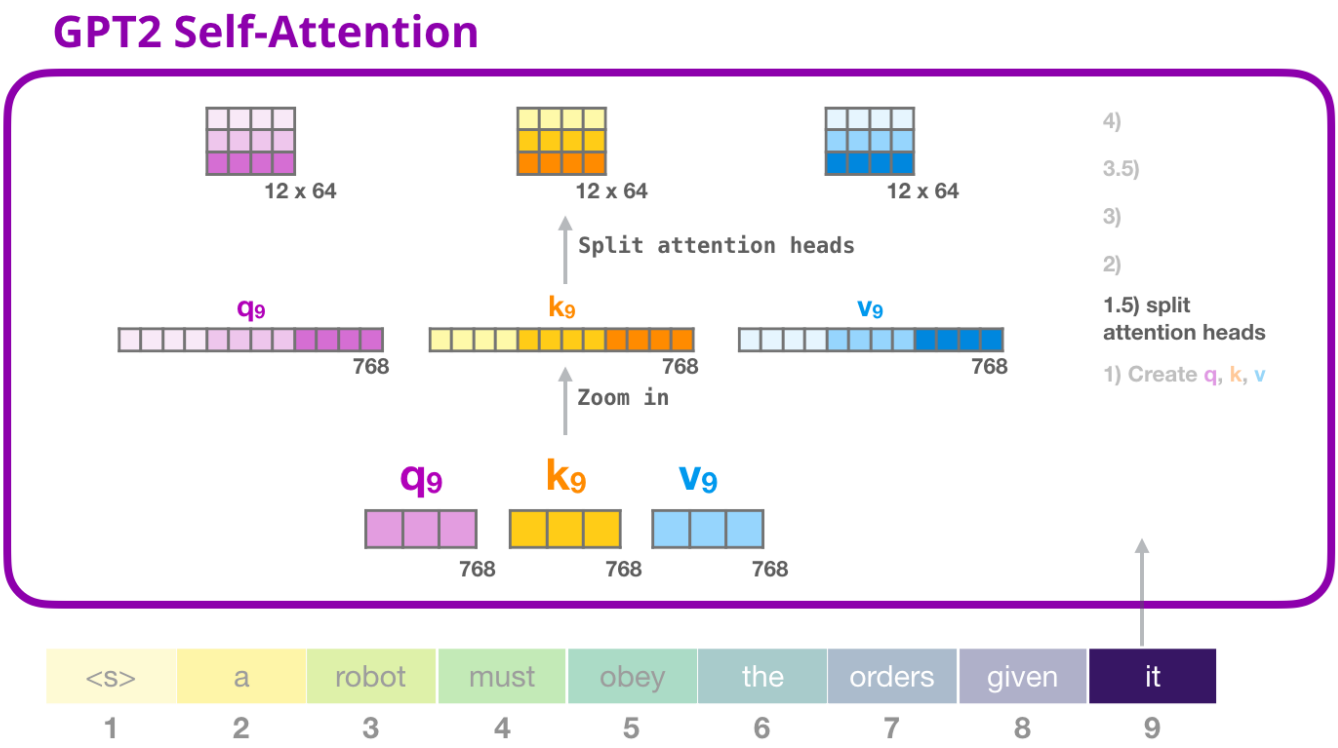
The multiplication results in a vector that's basically a concatenation of the query, key, and value vectors for the word it.



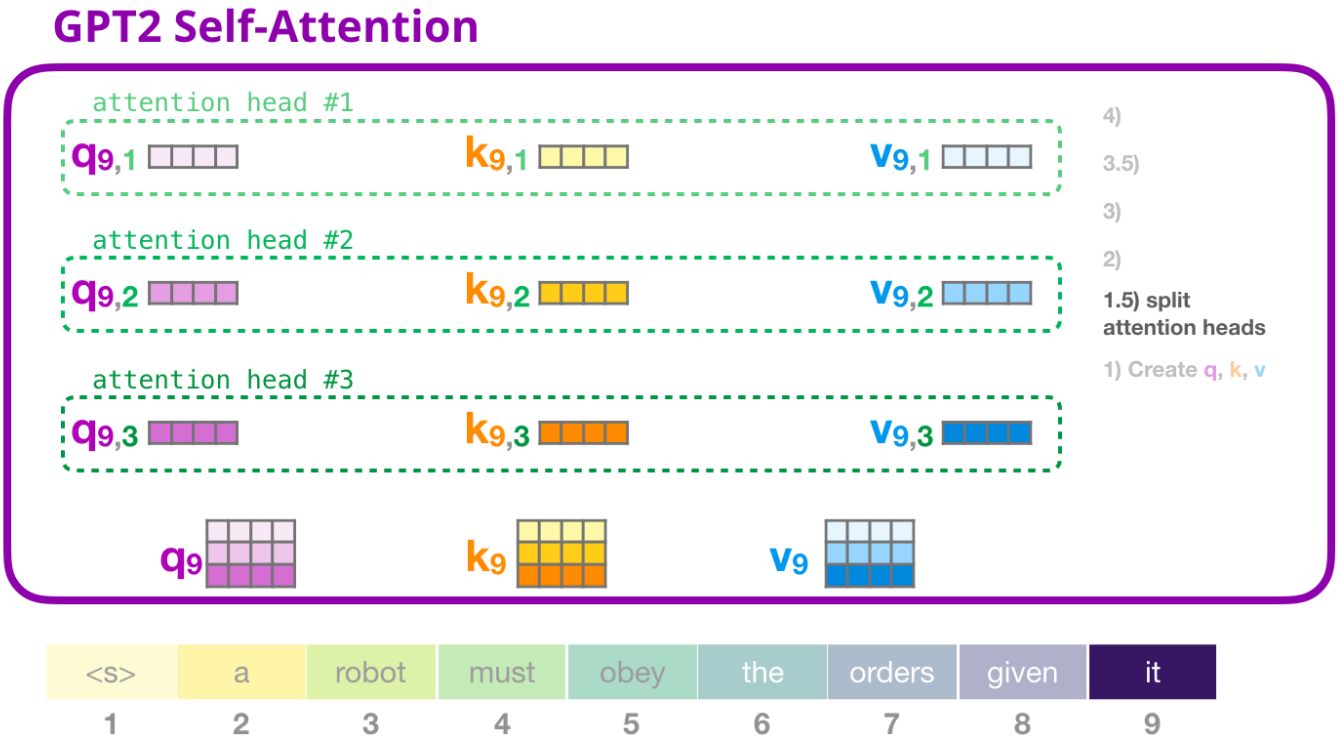
Multiplying the input vector by the attention weights vector (and adding a bias vector aftwards) results in the key, value, and query vectors for this token.

GPT-2 Self-attention: 1.5- Splitting into attention heads

In the previous examples, we dove straight into self-attention ignoring the "multi-head" part. It would be useful to shed some light on that concept now. 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. The small GPT2 has 12 attention heads, so that would be the first dimension of the reshaped matrix:

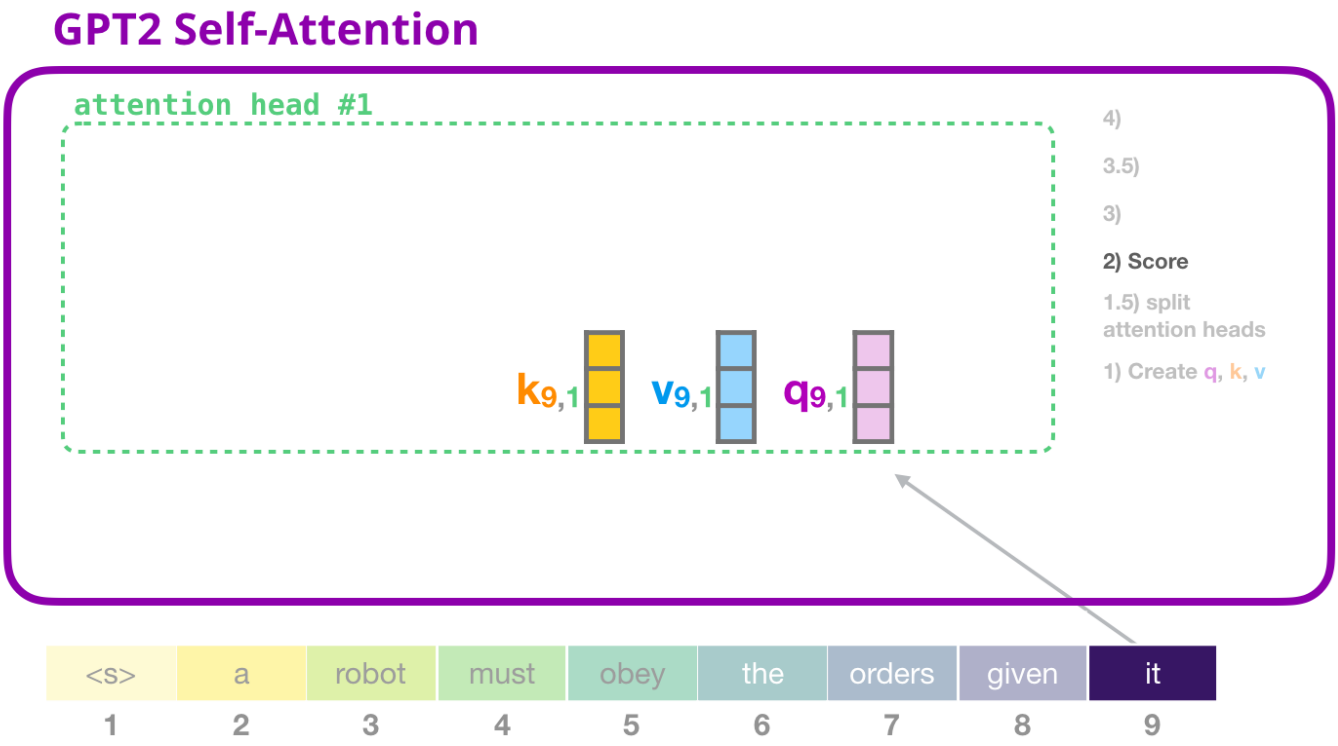


In the previous examples, we've looked at what happens inside one attention head. One way to think of multiple attention-heads is like this (if we're to only visualize three of the twelve attention heads):

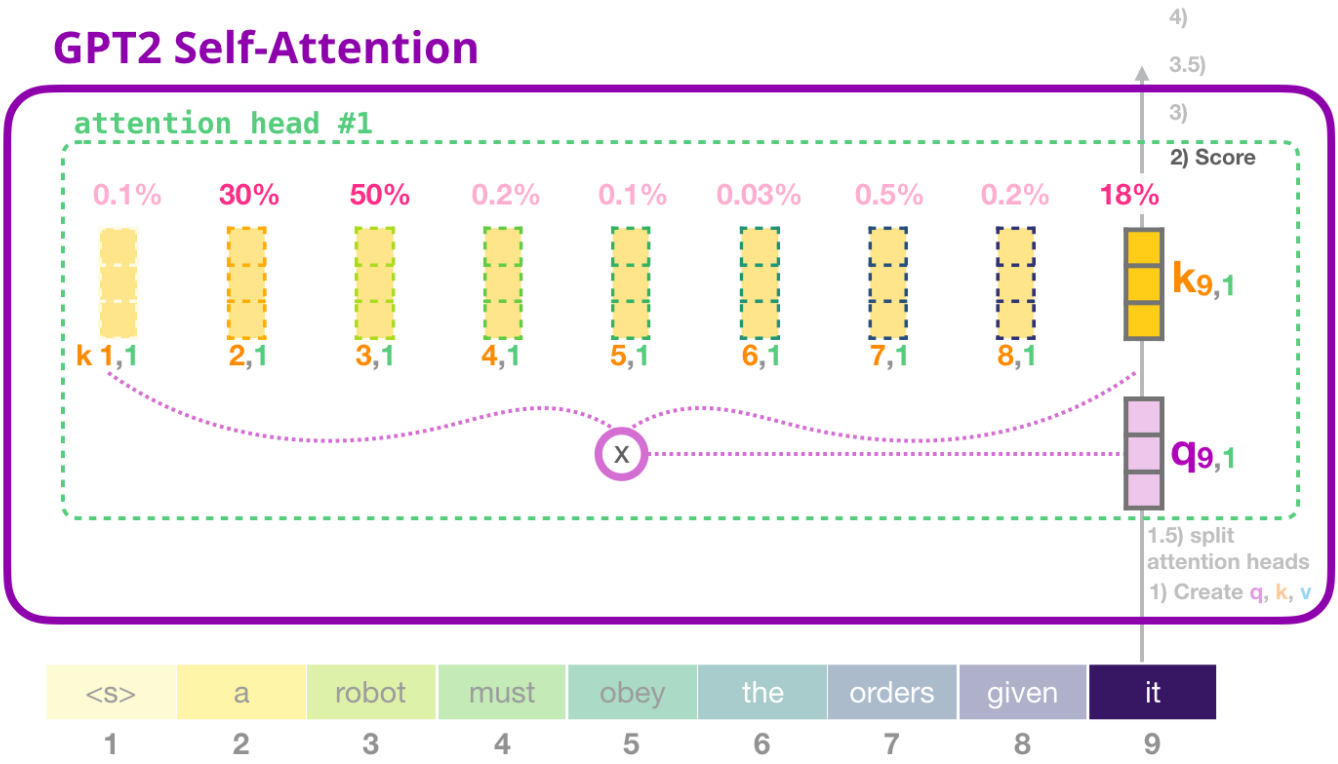


GPT-2 Self-attention: 2- Scoring

We can now proceed to scoring – knowing that we're only looking at one attention head (and that all the others are conducting a similar operation):

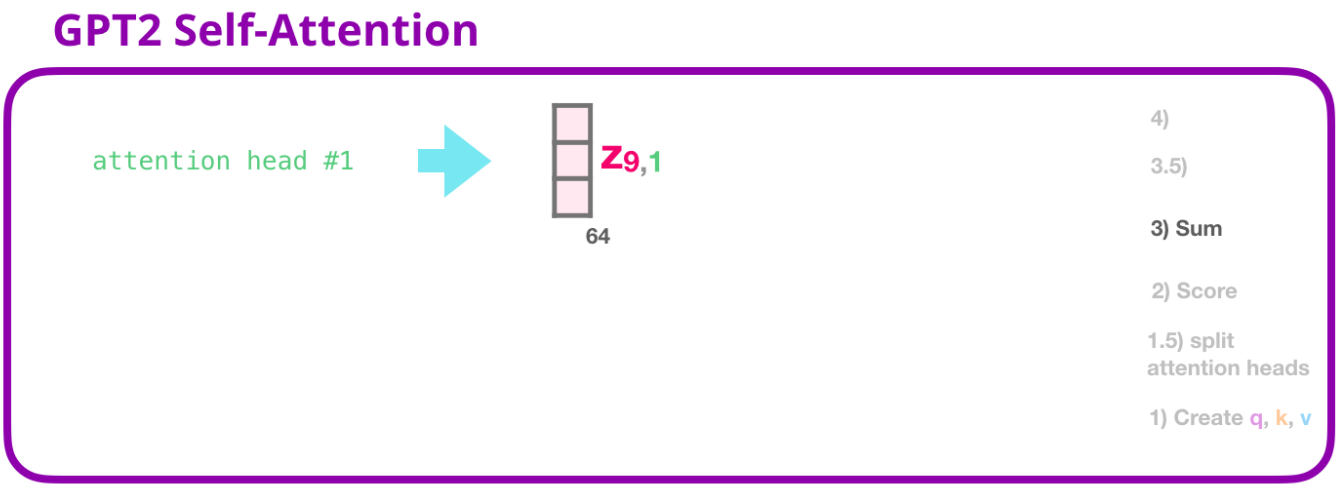


Now the token can get scored against all of keys of the other tokens (that were calculated in attention head #1 in previous iterations):



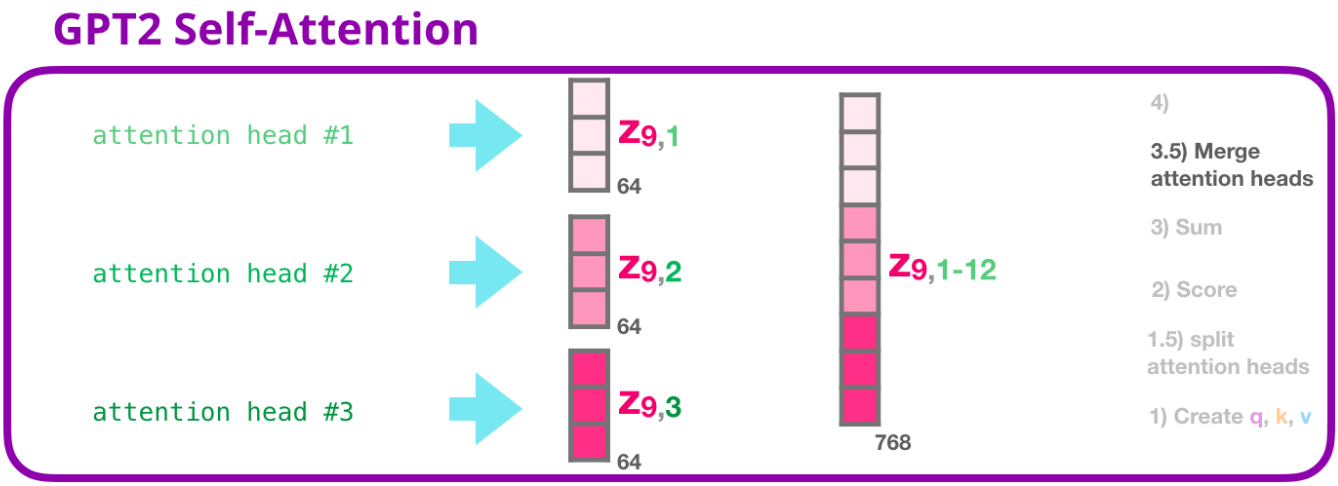
GPT-2 Self-attention: 3- Sum

As we've seen before, we now multiply each value with its score, then sum them up, producing the result of self-attention for attention-head #1:



GPT-2 Self-attention: 3.5- Merge attention heads

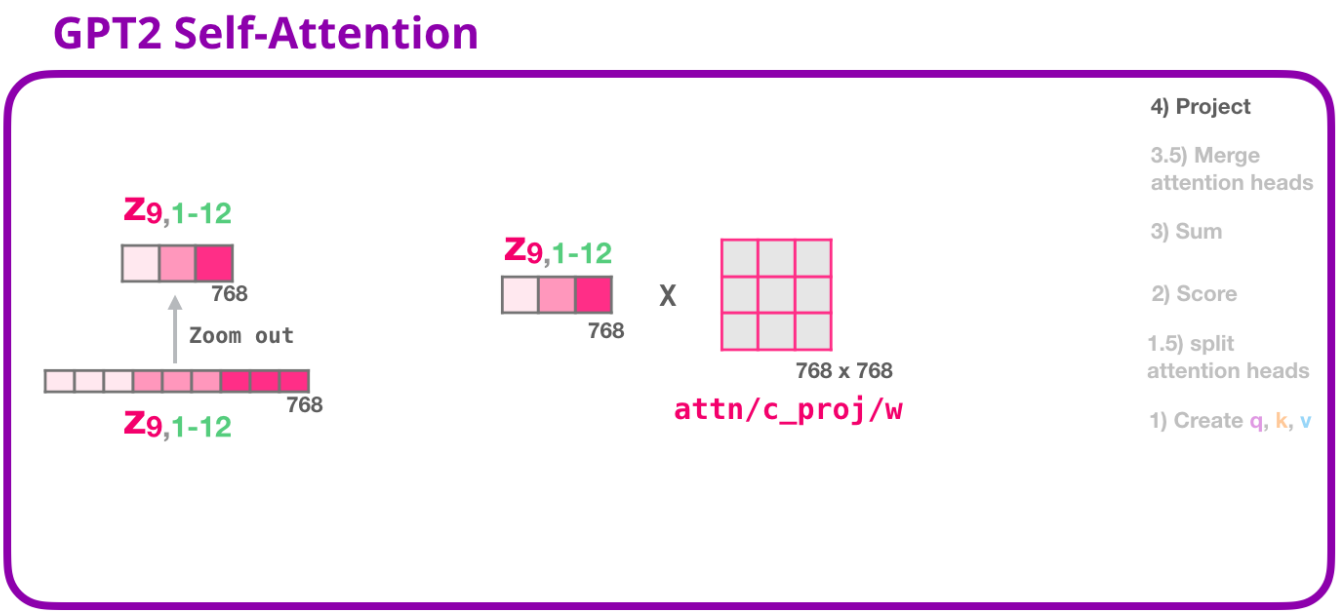
The way we deal with the various attention heads is that we first concatenate them into one vector:



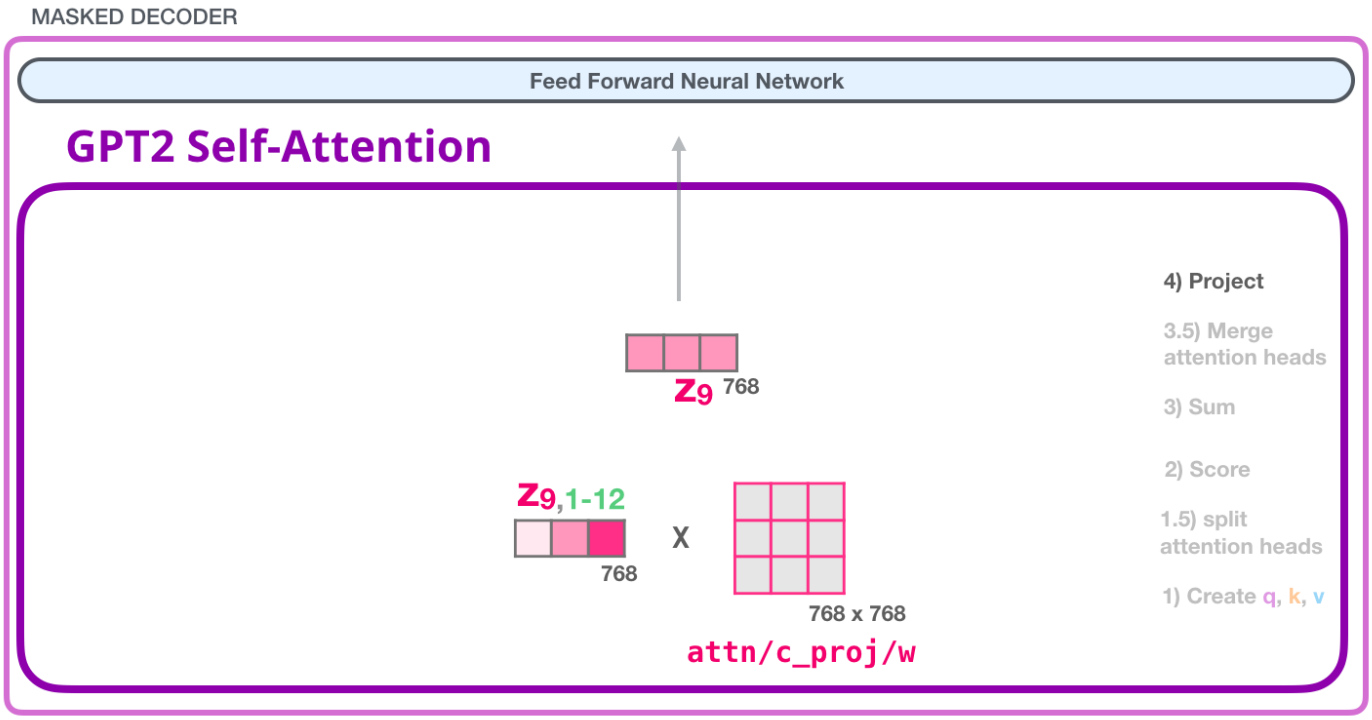
But the vector isn't ready to be sent to the next sublayer just yet. We need to first turn this Frankenstein's-monster of hidden states into a homogenous representation.

GPT-2 Self-attention: 4- Projecting

We'll let the model learn how to best map concatenated self-attention results into a vector that the feed-forward neural network can deal with. Here comes our second large weight matrix that projects the results of the attention heads into the output vector of the self-attention sublayer:

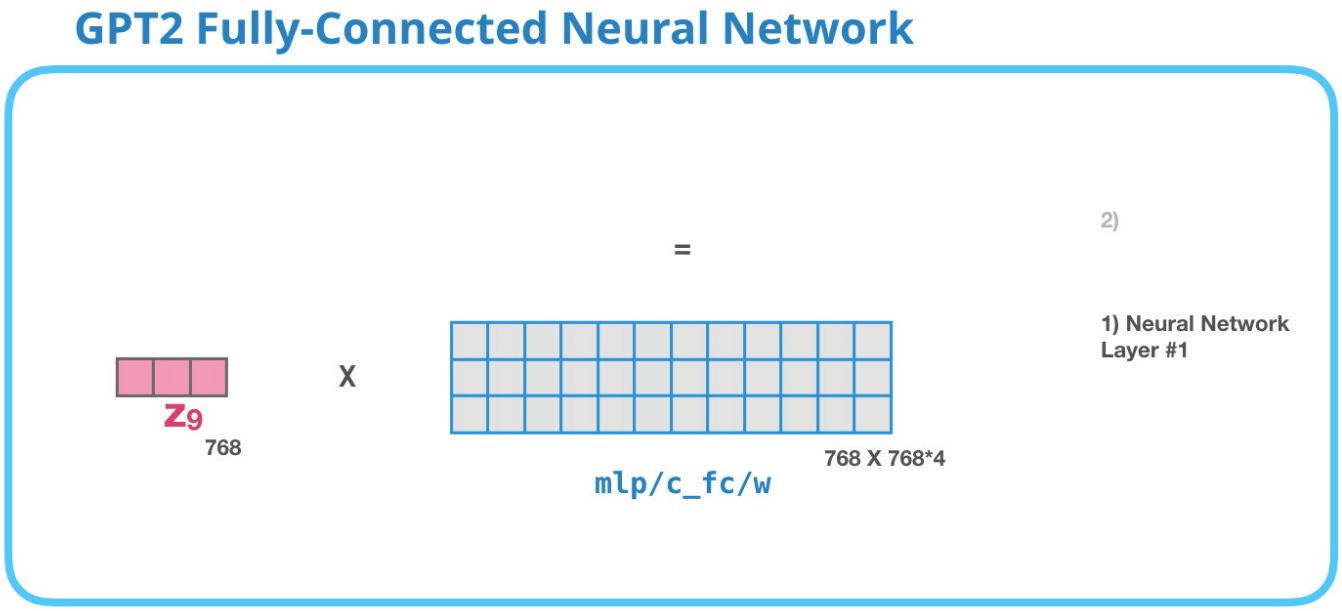


And with this, we have produced the vector we can send along to the next layer:



GPT-2 Fully-Connected Neural Network: Layer #1

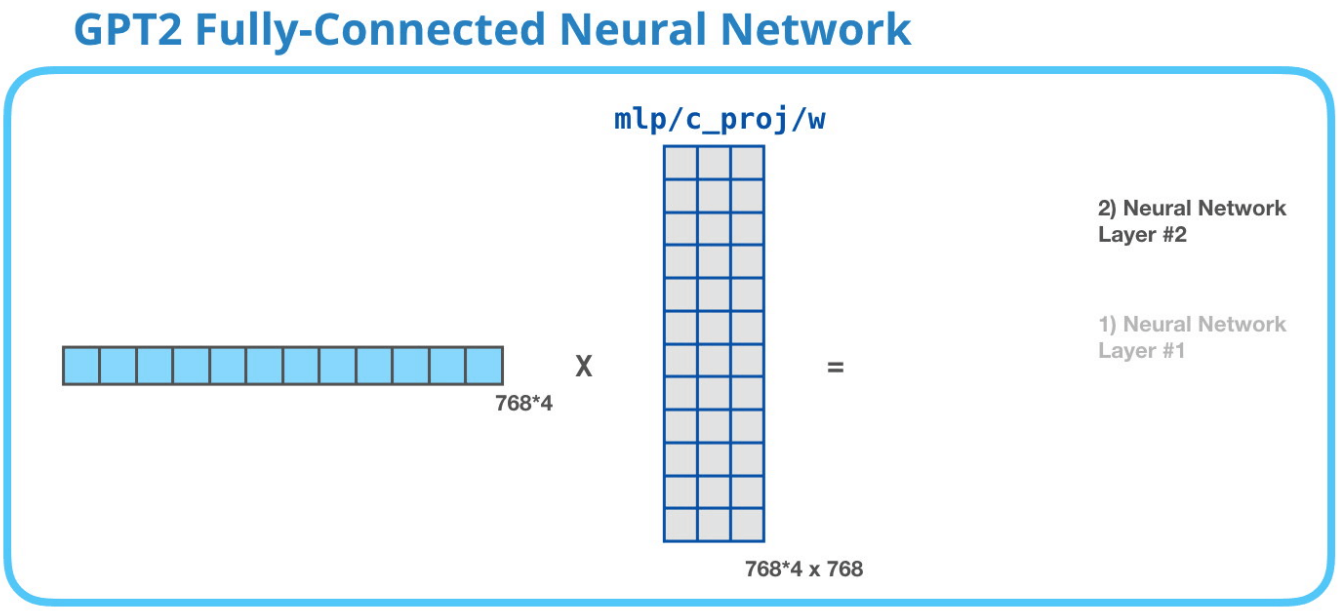
The fully-connected neural network is where the block processes its input token after self-attention has included the appropriate context in its representation. It is made up of two layers. The first layer is four times the size of the model (Since GPT2 small is 768, this network would have 768*4 = 3072 units). Why four times? That's just the size the original transformer rolled with (model dimension was 512 and layer #1 in that model was 2048). This seems to give transformer models enough representational capacity to handle the tasks that have been thrown at them so far.



(Not shown: A bias vector)

GPT-2 Fully-Connected Neural Network: Layer #2 - Projecting to model dimension

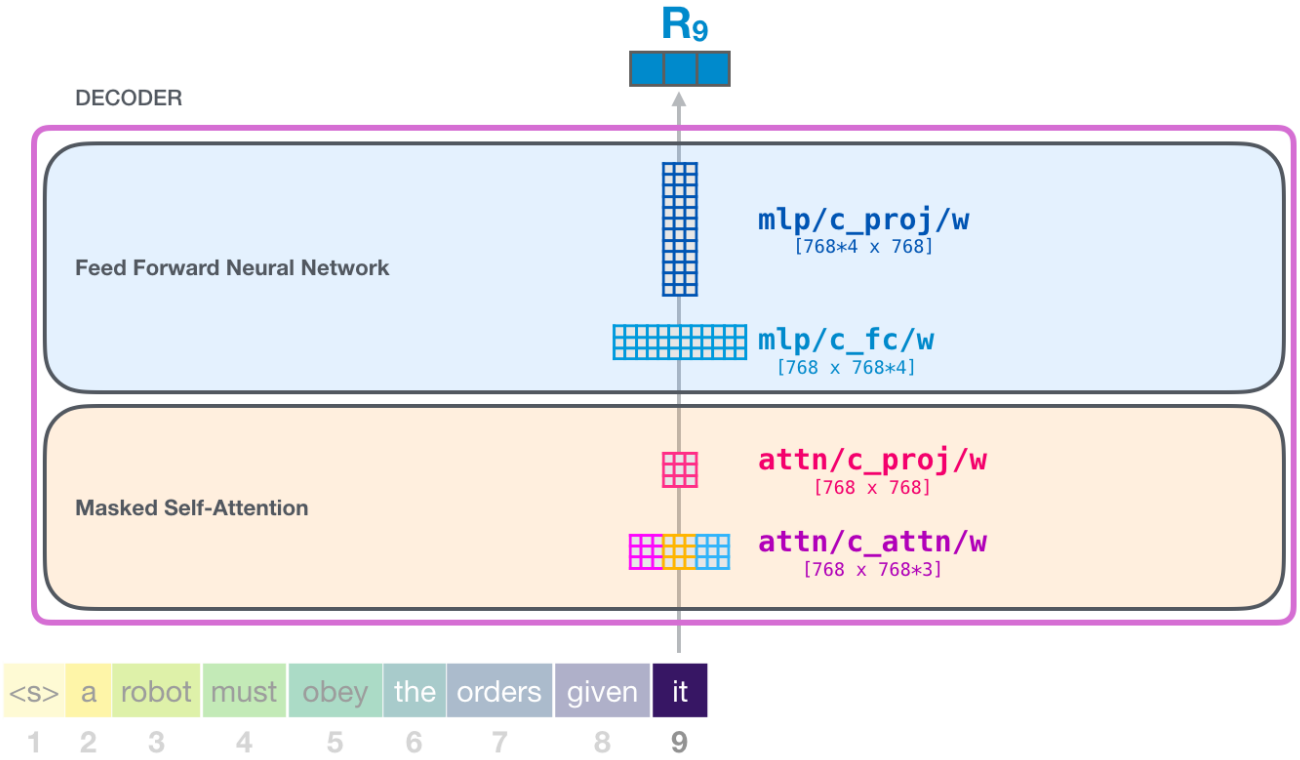
The second layer projects the result from the first layer back into model dimension (768 for the small GPT2). The result of this multiplication is the result of the transformer block for this token.



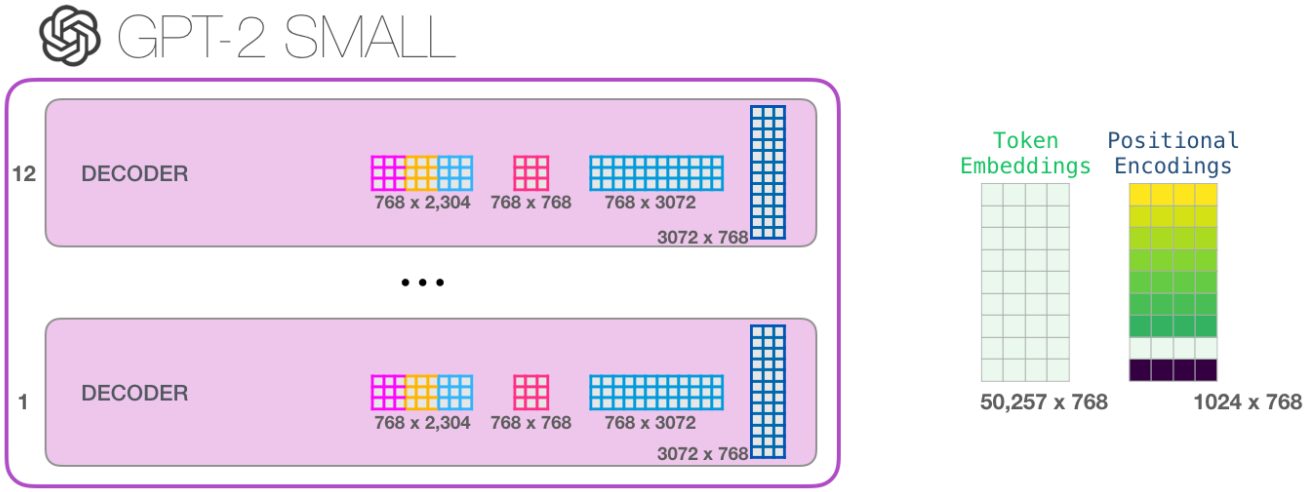
(Not shown: A bias vector)

You've Made It!

That's the most detailed version of the transformer block we'll get into! You now pretty much have the vast majority of the picture of what happens inside of a transformer language model. To recap, our brave input vector encounters these weight matrices:



And each block has its own set of these weights. On the other hand, the model has only one token embedding matrix and one positional encoding matrix:



If you want to see all the parameters of the model, then I have tallied them here:

				Dimensions		Parameters	
Single Transformer Block	Conv1d	**attn/c_attn**	w	768	2,304	1,769,472	
			b		2,304	2,304	
		attn/c_proj	w	768	768	589,824	
			b		768	768	
		mlp/c_fc	w	768	3,072	2,359,296	
			b		768	768	
		mlp/c_proj	w	3,072	768	2,359,296	
			b		768	768	
	Norm	**ln_1**	g		768	768	
			b		768	768	
		ln_2	g		768	768	
			b		768	768	
					Total	7,085,568	per block
					X 12 blocks	85,026,816	In all blocks
		Embeddings		50,257	768	38,597,376	
		Positional Encoding		1,024	768	786,432	
					Grand Total	**124,410,624**	

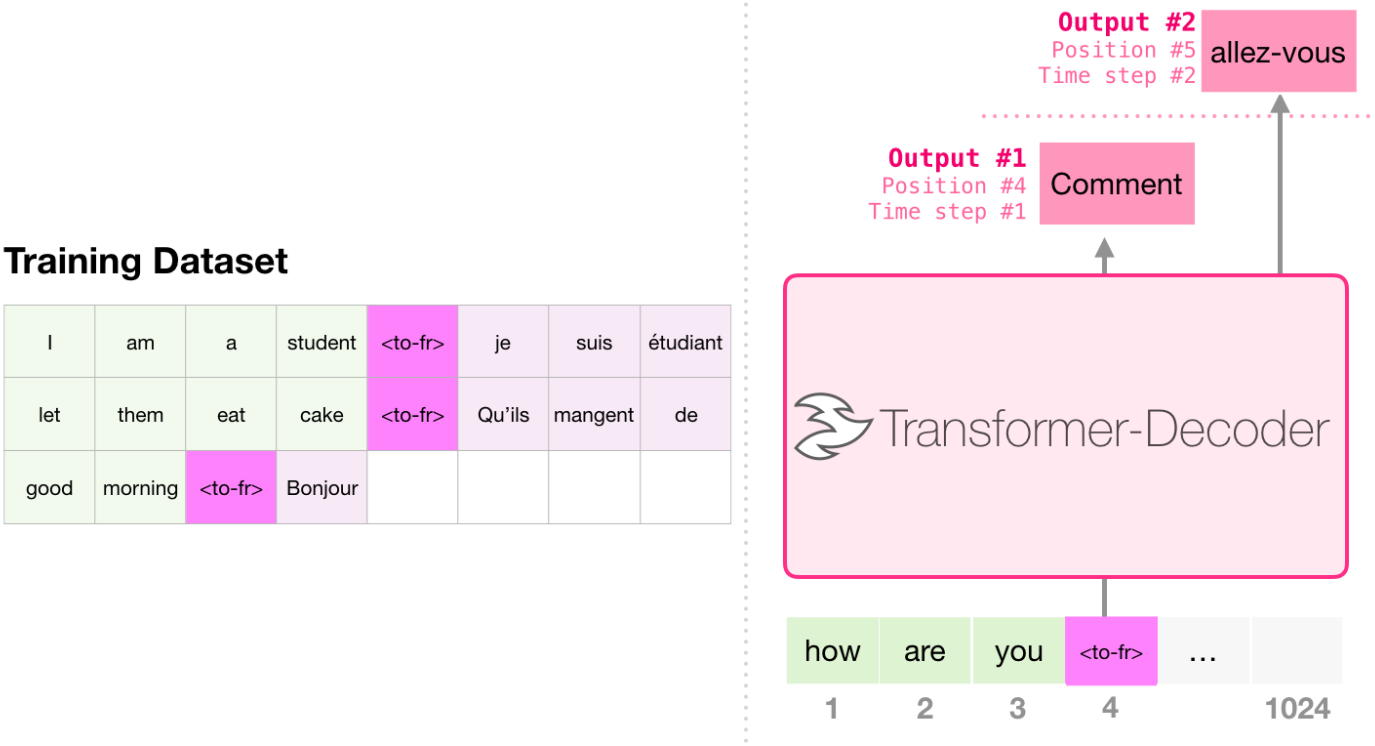
They add up to 124M parameters instead of 117M for some reason. I'm not sure why, but that's how many of them seems to be in the published code (please correct me if I'm wrong).

Part 3: Beyond Language Modeling

The decoder-only transformer keeps showing promise beyond language modeling. There are plenty of applications where it has shown success which can be described by similar visuals as the above. Let's close this post by looking at some of these applications

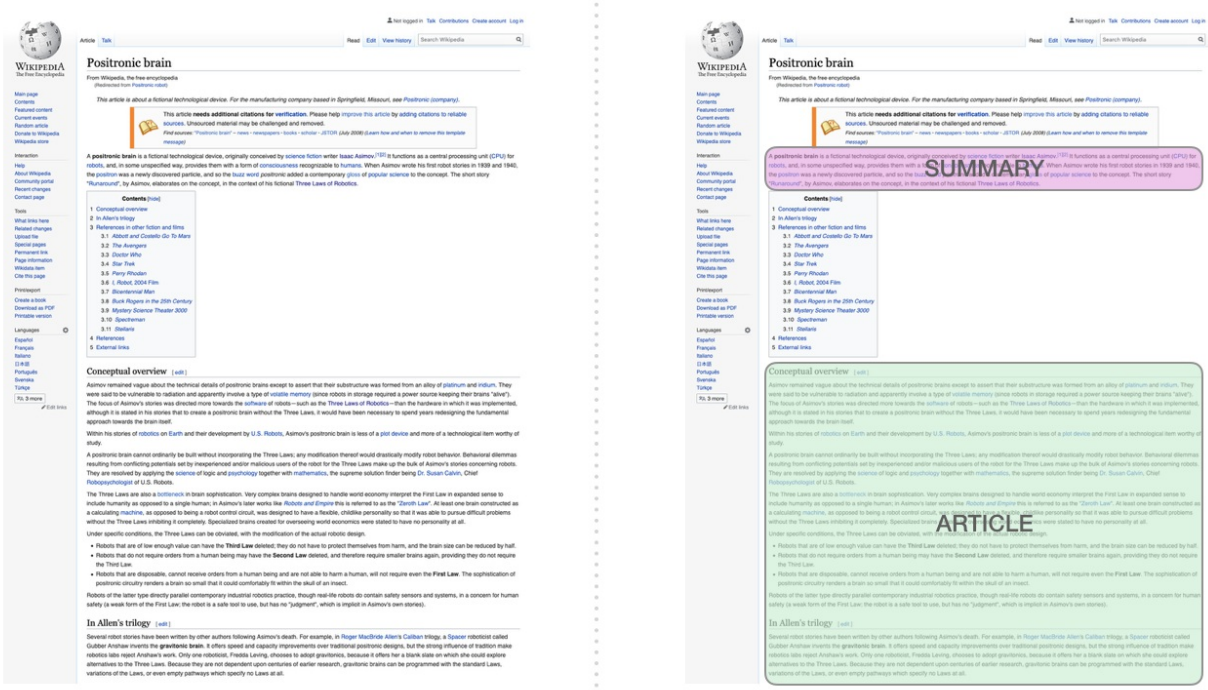
Machine Translation

An encoder is not required to conduct translation. The same task can be addressed by a decoder-only transformer:

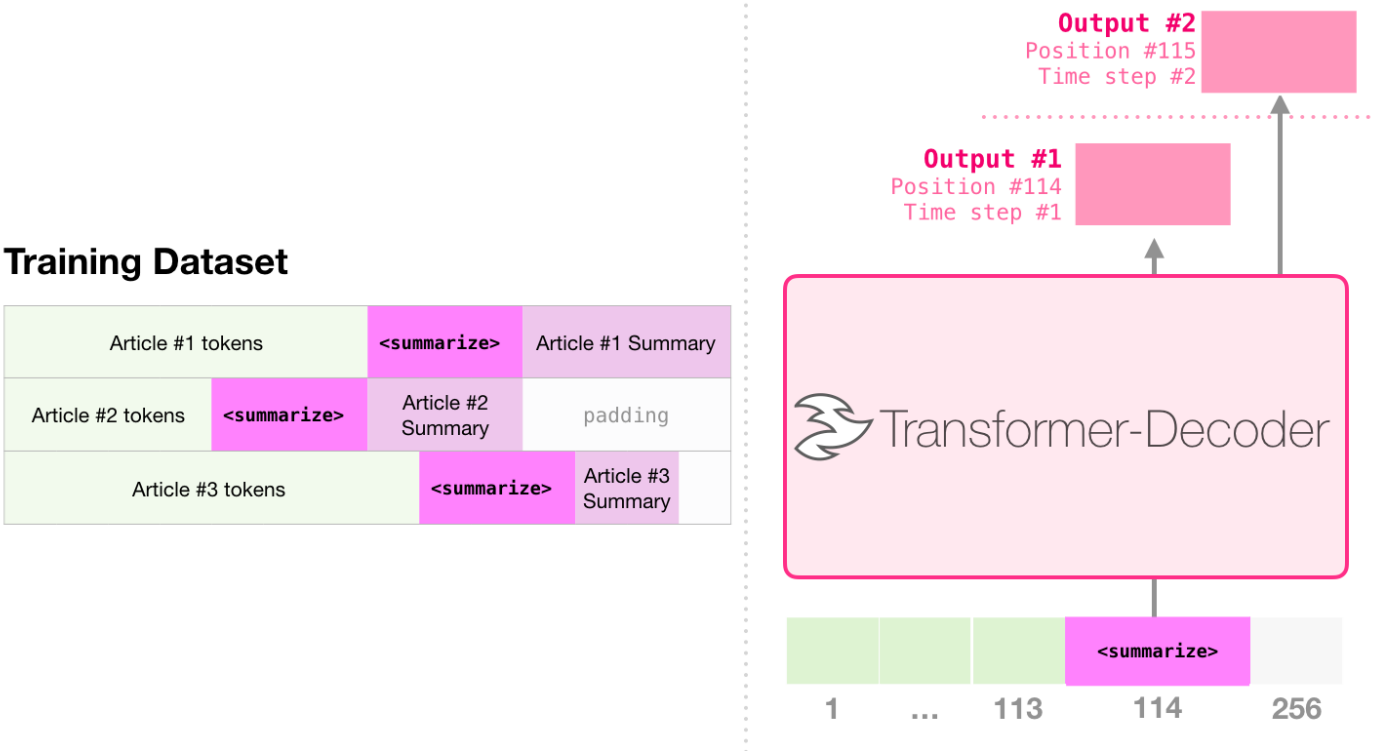


Summarization

This is the task that the first decoder-only transformer was trained on. Namely, it was trained to read a wikipedia article (without the opening section before the table of contents), and to summarize it. The actual opening sections of the articles were used as the labels in the training datasest:



The paper trained the model against wikipedia articles, and thus the trained model was able to summarize articles:



Transfer Learning

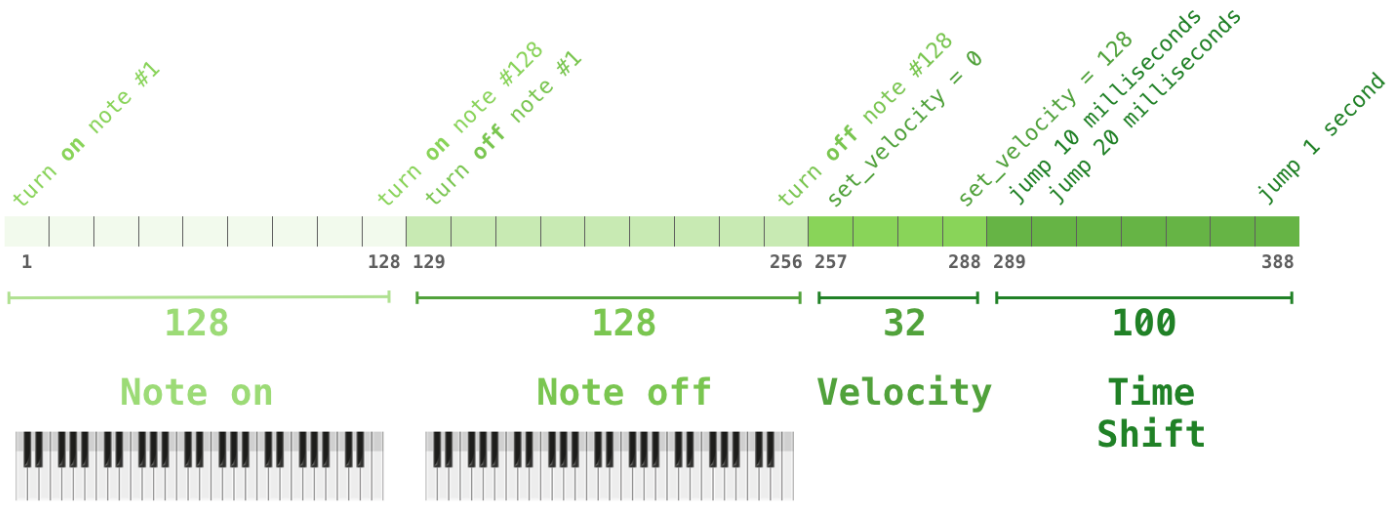
In Sample Efficient Text Summarization Using a Single Pre-Trained Transformer, a decoder-only transformer is first pre-trained on language modeling, then finetuned to do summarization. It turns out to achieve better results than a pre-trained encoder-decoder transformer in limited data settings.

The GPT2 paper also shows results of summarization after pre-training the model on language modeling.

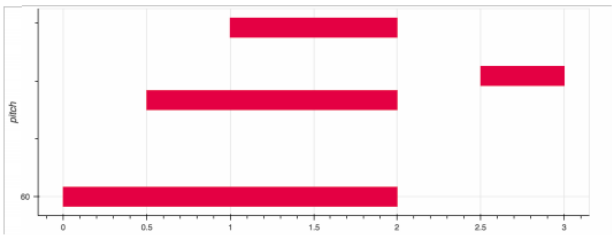
Music Generation

The Music Transformer uses a decoder-only transformer to generate music with expressive timing and dynamics. “Music Modeling” is just like language modeling – just let the model learn music in an unsupervised way, then have it sample outputs (what we called “rambling”, earlier).

You might be curious as to how music is represented in this scenario. Remember that language modeling can be done through vector representations of either characters, words, or tokens that are parts of words. With a musical performance (let’s think about the piano for now), we have to represent the notes, but also velocity – a measure of how hard the piano key is pressed.

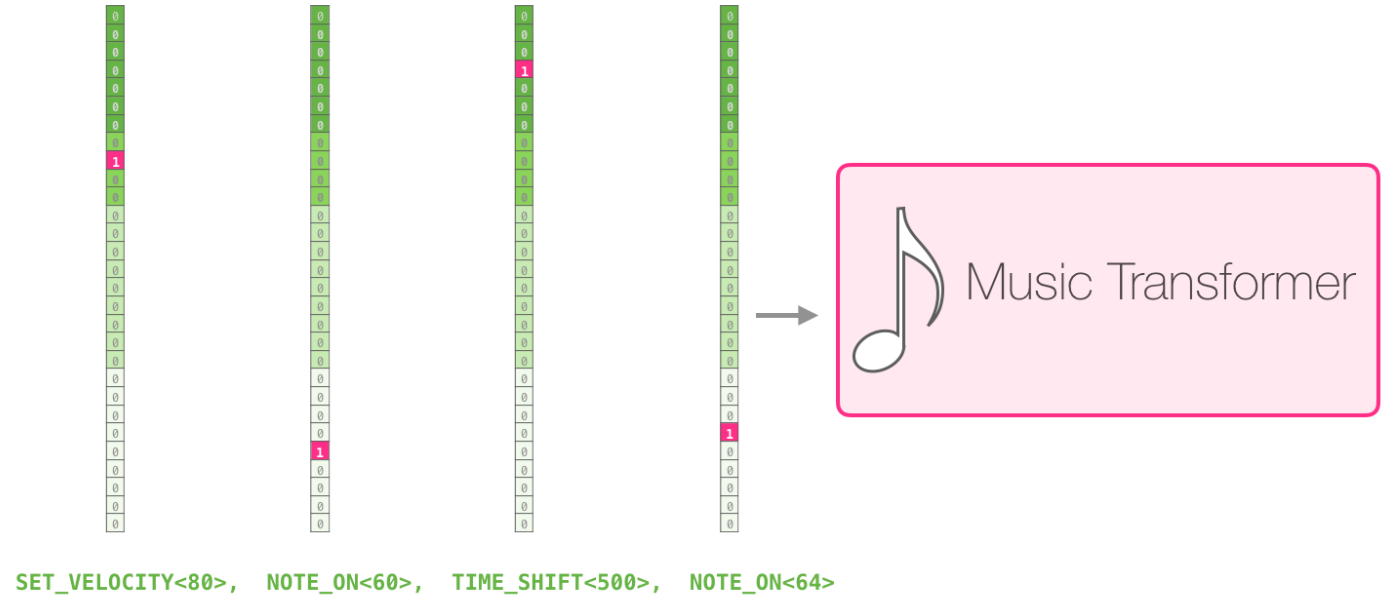


A performance is just a series of these one-hot vectors. A midi file can be converted into such a format. The paper has the following example input sequence:

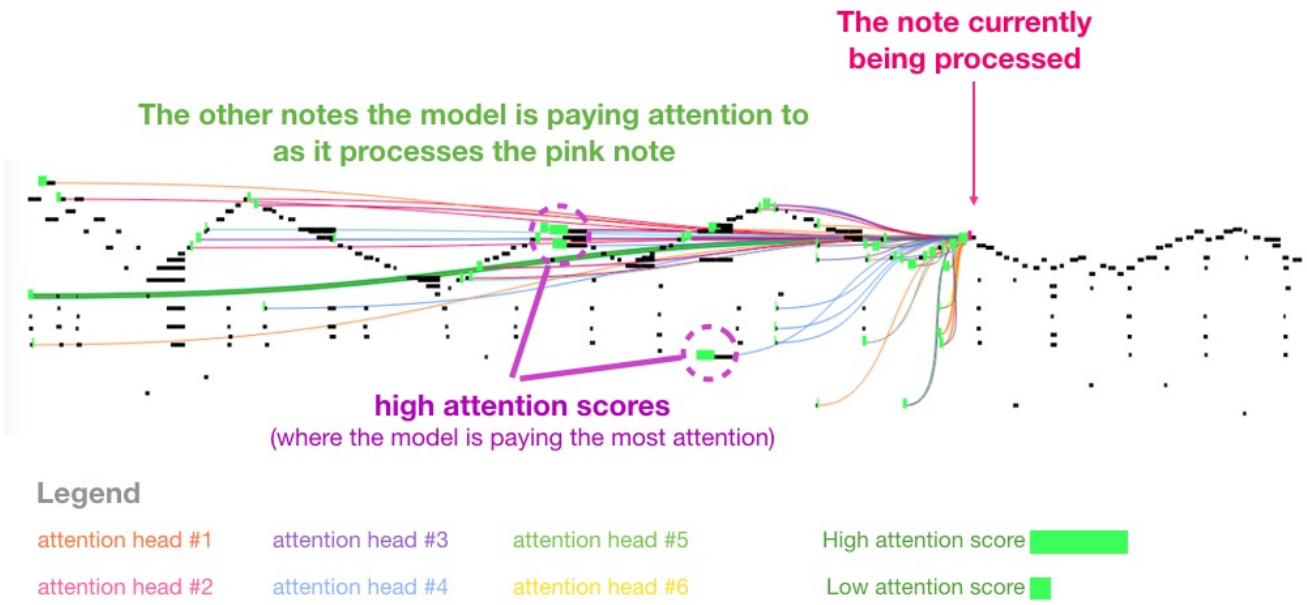


```
SET_VELOCITY<80>, NOTE_ON<60>
TIME_SHIFT<500>, NOTE_ON<64>
TIME_SHIFT<500>, NOTE_ON<67>
TIME_SHIFT<1000>, NOTE_OFF<60>, NOTE_OFF<64>,
NOTE_OFF<67>
TIME_SHIFT<500>, SET_VELOCITY<100>, NOTE_ON<65>
TIME_SHIFT<500>, NOTE_OFF<65>
```

The one-hot vector representation for this input sequence would look like this:



I love a visual in the paper that showcases self-attention in the Music Transformer. I’ve added some annotations to it here:



"Figure 8: This piece has a recurring triangular contour. The query is at one of the latter peaks and it attends to all of the previous high notes on the peak, all the way to beginning of the piece." ... "[The] figure shows a query (the source of all the attention lines) and previous memories being attended to (the notes that are receiving more softmax probabiliy is highlighted in). The coloring of the attention lines correspond to different heads and the width to the weight of the softmax probability."

If you're unclear on this representation of musical notes, check out this video.

Conclusion

This concludes our journey into the GPT2, and our exploration of its parent model, the decoder-only transformer. I hope that you come out of this post with a better understanding of self-attention and more comfort that you understand more of what goes on inside a transformer.

Resources

- The GPT2 Implementation from OpenAI
- Check out the pytorch-transformers library from Hugging Face in addition to GPT2, it implements BERT, Transformer-XL, XLNet and other cutting-edge transformer models.

Acknowledgements

Thanks to Lukasz Kaiser, Mathias Müller, Peter J. Liu, Ryan Sepassi and Mohammad Saleh for feedback on earlier versions of this post.

Comments or corrections? Please tweet me at @jalammar

Written on August 12, 2019

Note: If you translate any of the posts, let me know so I can link your translation to the original post. My email is in the about page.