Transformers

Lecture 5

EN.705.743: ChatGPT from Scratch

Lecture Outline

Lecture 5: Transformers

- Recap (MHA)
- Transformer Layers
- Causal Masking
- Encoders and Decoders
- Decoder-Only Models
- Output Layer

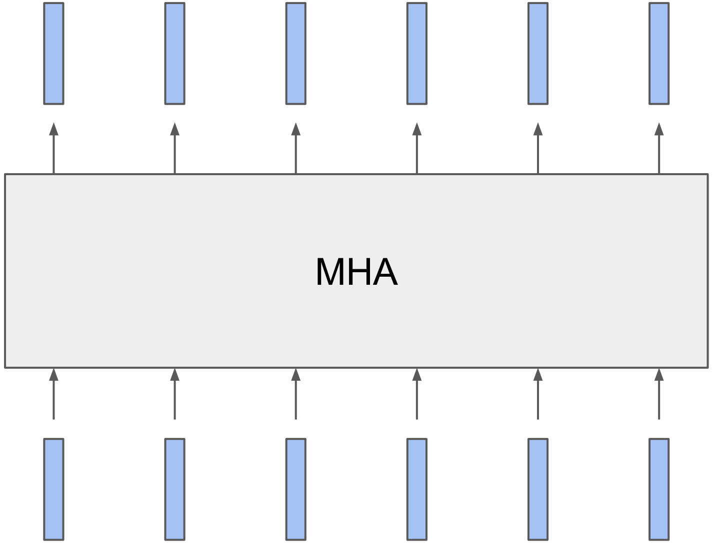
Recap

Recap

Last week, we learned the details of Multihead Attention (MHA)

MHA takes as input a set of vectors, and outputs a set of vectors of the same size.

Y = S vectors of length D



X = S vectors of length D

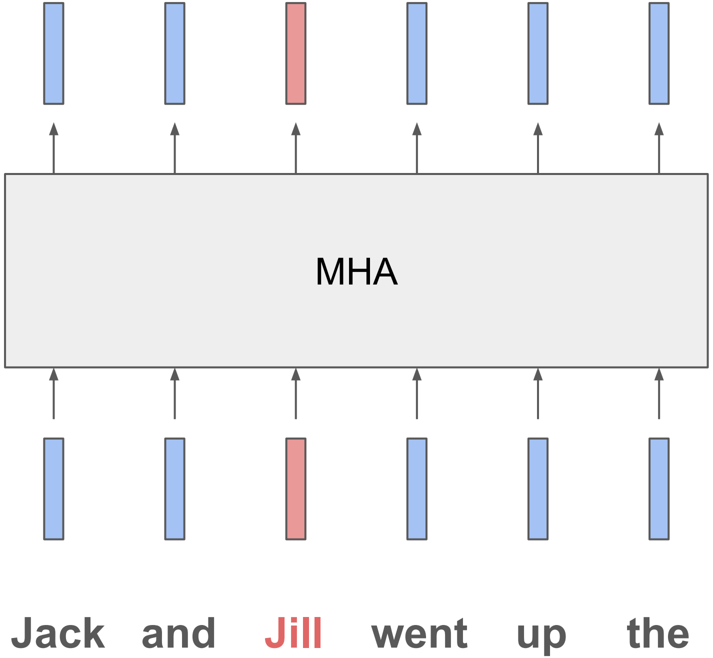
Recap

Last week, we learned the details of Multihead Attention (MHA)

MHA takes as input a set of vectors, and outputs a set of vectors of the same size.

The ith input represents the ith token.

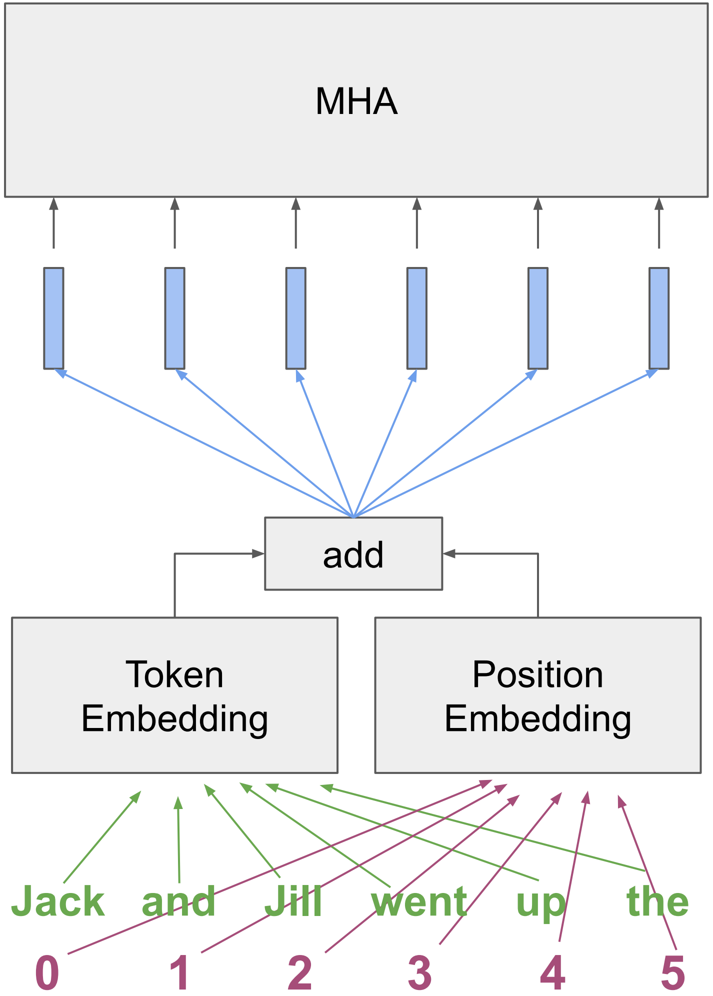
The ith output is a revised representation for that token, taking into account all other tokens in the sequence.



Recap

MHA does not care about the order of the vectors.

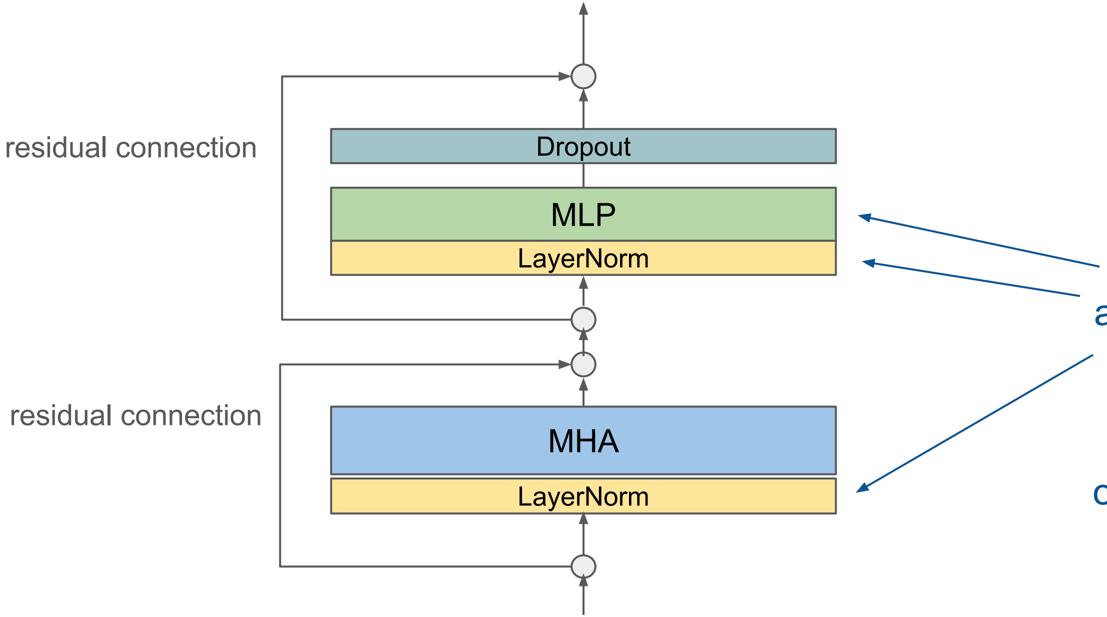
However, when we embed our tokens we include a position embedding that reflects position.



Transformer “Layers”

Transformer Layer

The transformer layer (sometimes called a “transformer block”) is the core building block of an LLM. It consists of MHA, normalizations, an MLP, and a dropout layer:



The normalizations and MLP are applied per vector.

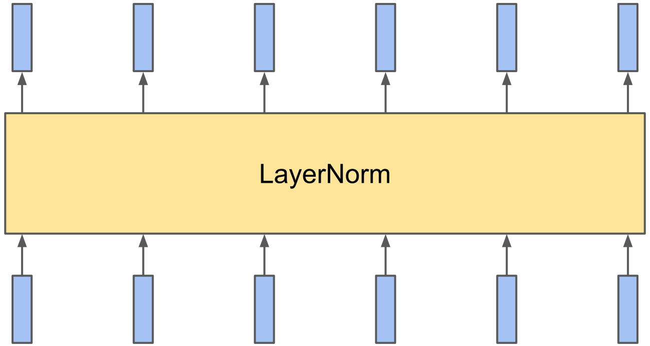
MHA is the only operation on multiple vectors.

LayerNorm

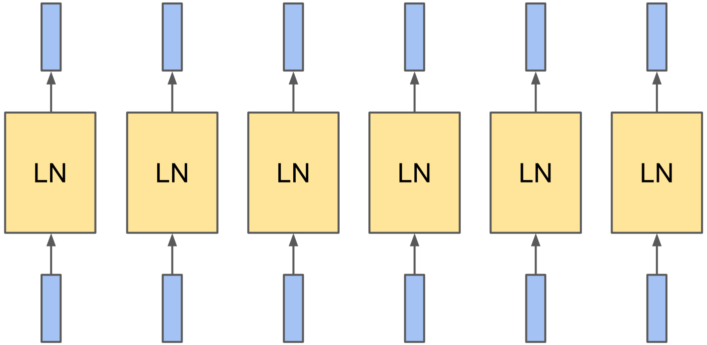
LayerNorm normalizes **each vector separately** by its mean and standard deviation. It also learns a weighting term and bias per feature.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

In this class we will simply use *torch.nn.LayerNorm()*, but it is good to understand what this is doing.

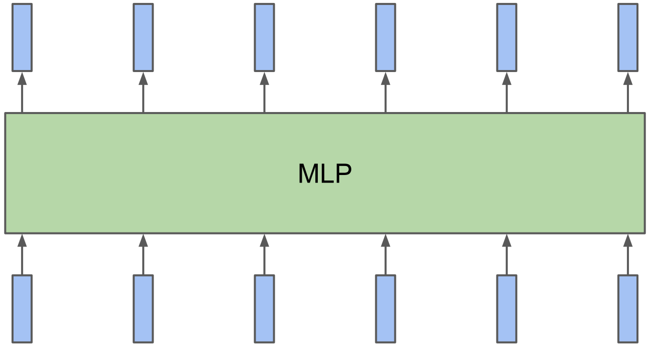


is really the same as:

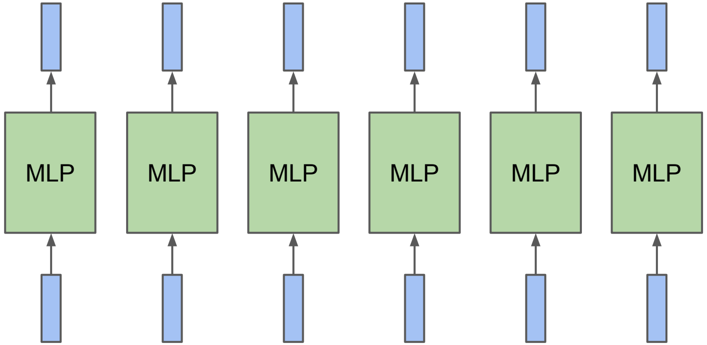


MLP

Like LayerNorm, the MLP is applied to each vector separately.



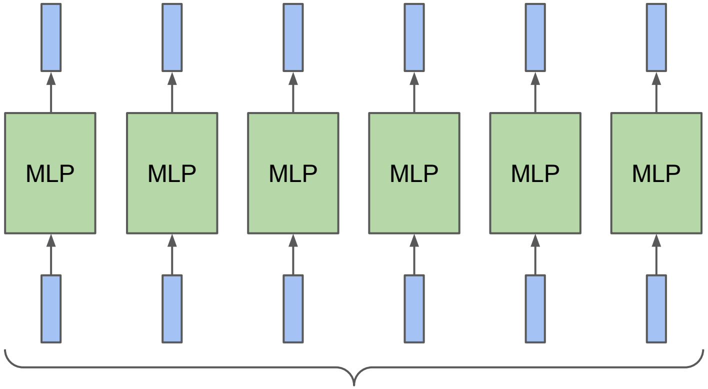
is really the same as:



Parallel MLP Processing

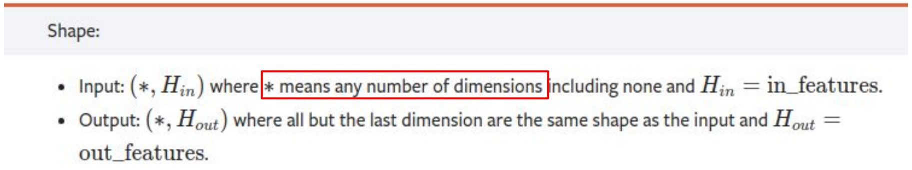
Linear layers in pytorch can be applied to any number of leading dimensions. Normally in an MLP we just have (batch, features), but it could be (batch, …, features) if we want.

So when we run (B, S, D) through an MLP in PyTorch, it will automatically apply to each vector separately.



SxD: Sequence length S, each entry a D-dimensional vector.

A batch of such sequences would be BxSxD.



Shape:

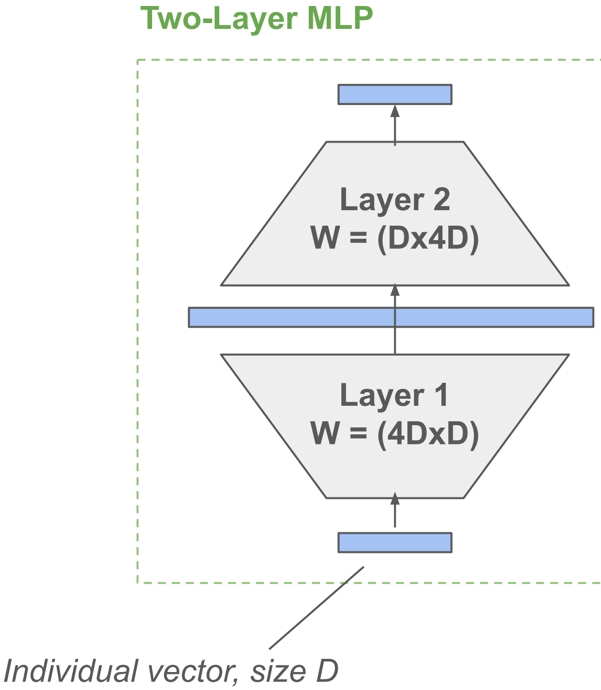
- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

MLP

The MLP is almost always a two-layer MLP, which first projects the vectors to a higher dimension and then projects them back down.

The size of the middle layer is almost always 4*D.

The middle layer uses ReLU (or GeLU, SiLU, etc)



Note

More recent architectures, rather than:

linear(act(linear(x)))

for the MLP, often people use the more exotic:

linear(act(linear(x)) * linear(x))

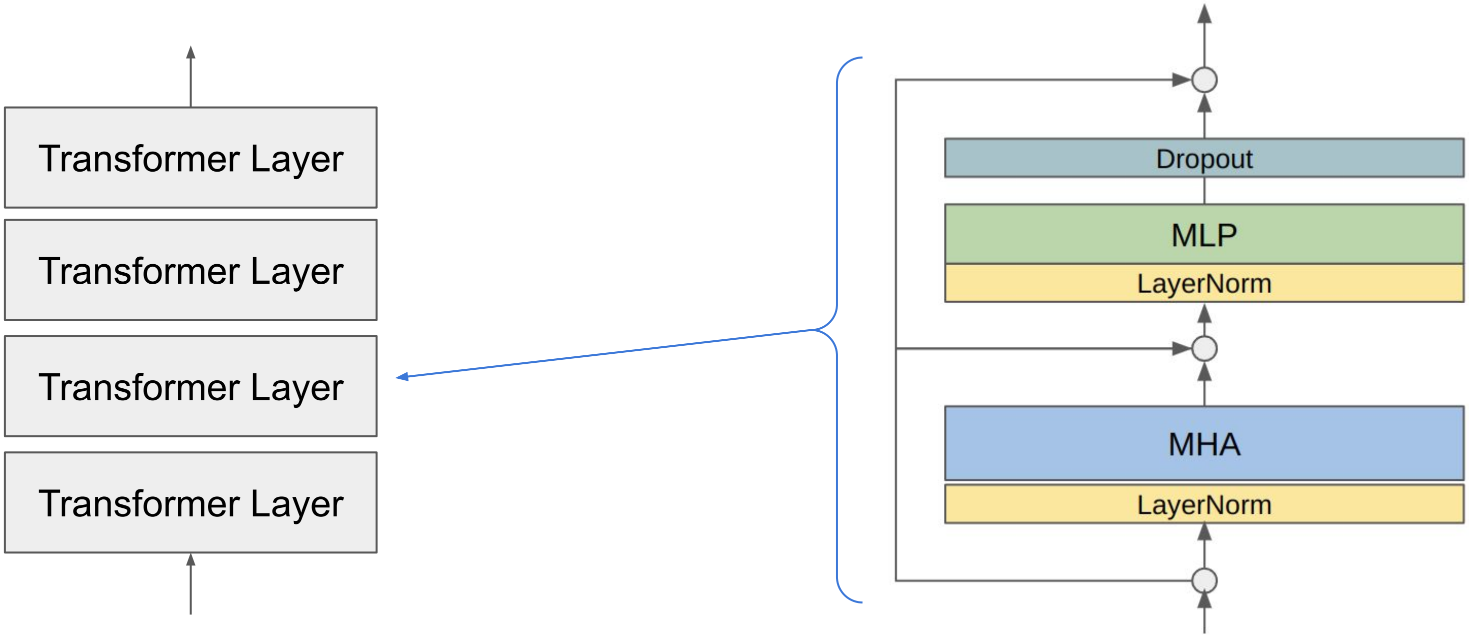
See notes here: https://github.com/meta-llama/llama/issues/245

Don’t worry about this for the homework! Just use linear(act(linear(x))).

Causal Masking

Transformer

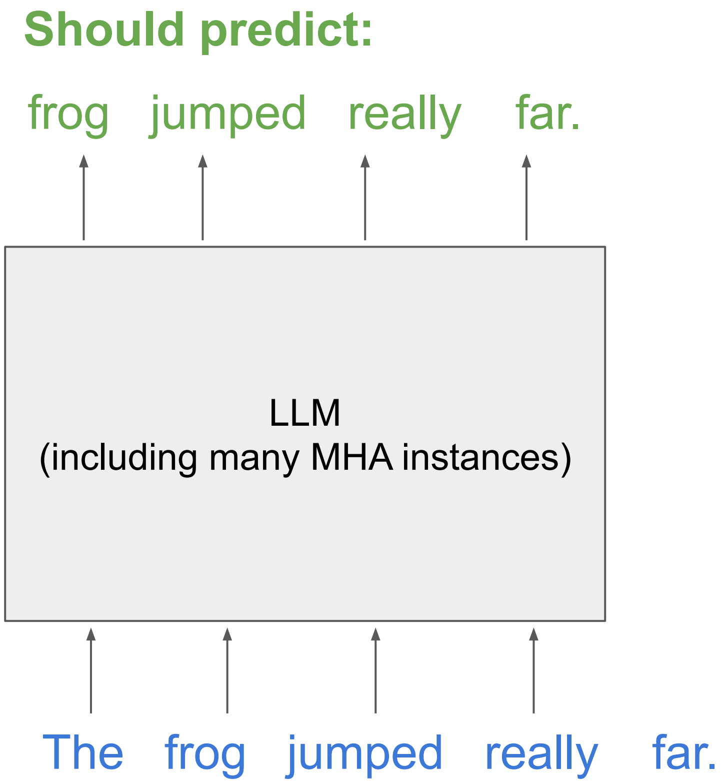
A “transformer” model at a minimum is simply a stack of transformer layers.



Revisiting our training objective

Before we discuss transformer ***models***, there is one detail remaining about transformer ***layers***.

If we “zoom out” and look at our model, we want each output to predict the next token in the sequence.

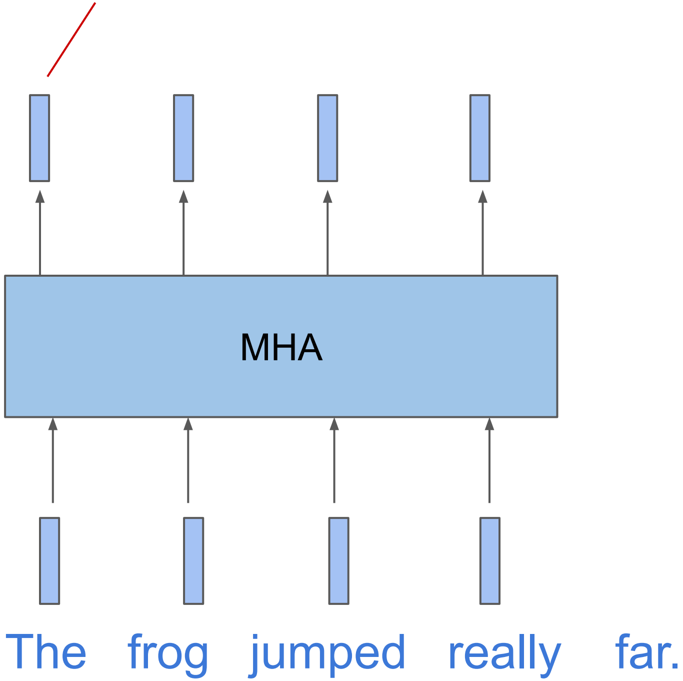


Revisiting our training objective

Before we discuss transformer ***models***, there is one detail remaining about transformer ***layers***.

If we “zoom out” and look at our model, we want each output to predict the next token in the sequence.

However, MHA considers all inputs, so it could simply “look” at the next word.


Computing this vector includes attending to the vector for “frog”... so it’s not that hard to predict “frog”.

Causal Masking

To fix this problem, we apply a “causal mask” to our attention mechanism. Attention scores are reset to zero for future entries.

If we remember the attention equation, the softmax’ed term is a weight applied to V:

How strongly should we weight (“attend to”) the vector represented by each V?

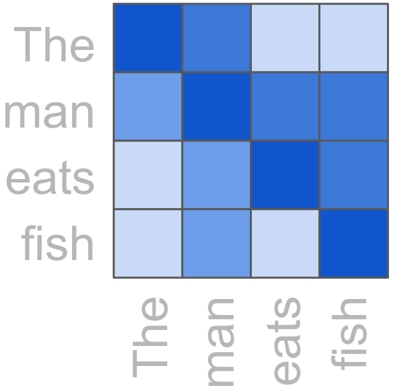
$$softmax(\frac{\overbrace{QK^T}}{\sqrt{D}}) * V$$

Causal Masking

We can visualize these weighting terms as a matrix, which shows us how each input relates to each other input:

$$softmax(\frac{QK^T}{\sqrt{D}}) * V$$

$QK^T =$

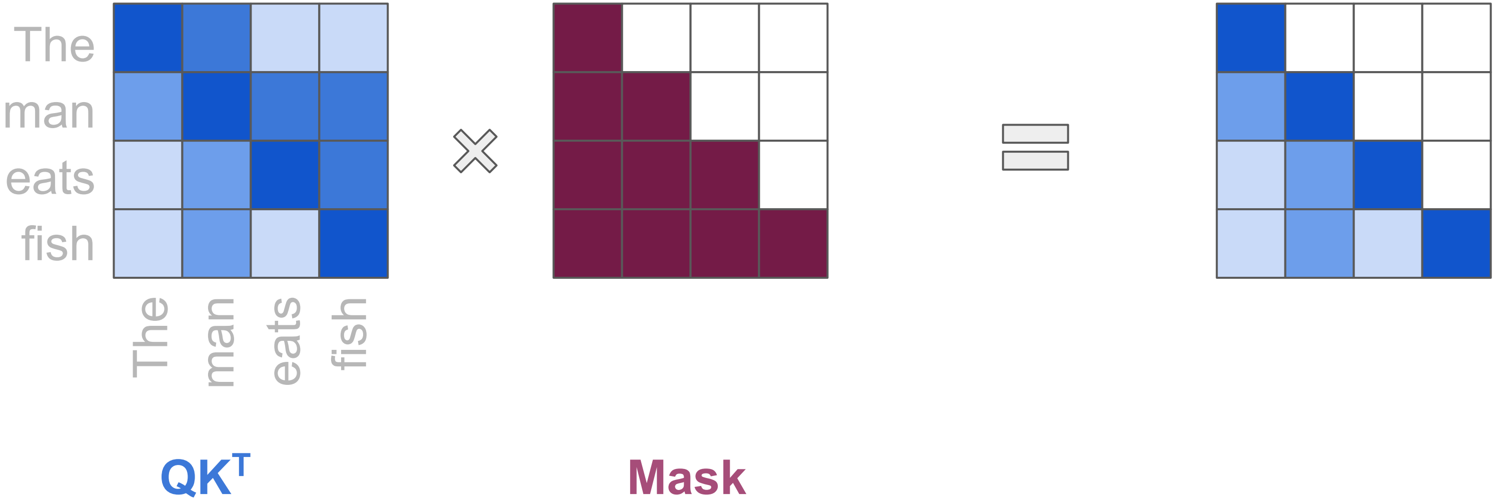


For example, this row might tell us that when understanding “eats”, we pay attention mostly to “man” and “fish”.

(Darker = Higher Value)

Causal Masking

We can prevent attending to future words by masking out relevant values.



Causal Masking

We can prevent attending to future words by masking out relevant values. Our new equation is:

$$Y = softmax(\frac{Mask(QK^T)}{\sqrt{D}}) * V$$

Some notes on this:

- Constructing a mask of this form is actually very easy to do with **torch.tril()**.
- Since we are taking a softmax, we actually assign values to -infinity, not zero (they will become zero after the softmax)
- Transformers with this mask are sometimes called “Causal”, and create “Causal Language Models” (CLM’s for short). All LLMs we discuss in this class are CLMs.

Implementation notes for causal mask

Remember, QK^T is size (H, S, S).

Our mask is size (S, S) for all possible pairs of two sequence elements.

Build the mask as size (S,S) and then repeat it along a new leading dimension to get (H,S,S).

(Or with a batch, this is (B,H,S,S)).

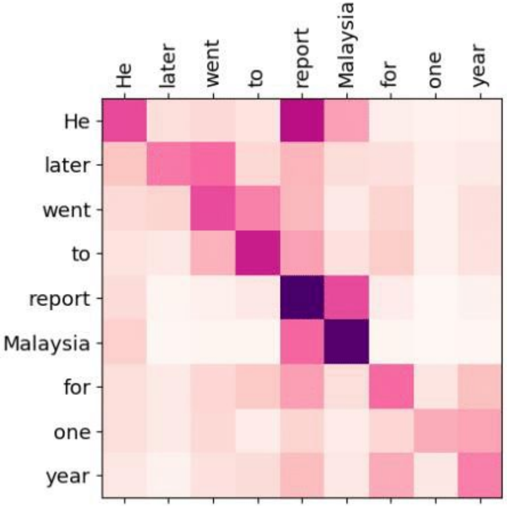
Aside: Visualizing Attention Masks

If you plot an entire SxS attention matrix you can often see patterns. Note they are not symmetric!

Here is an example of an (unmasked) attention matrix:

… attend to these entries.

These entries…



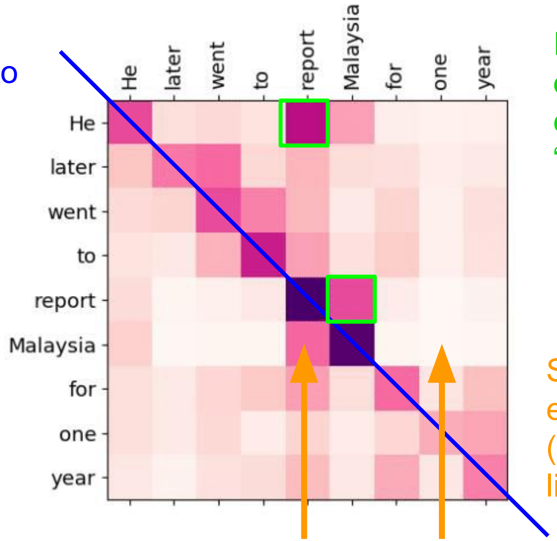
Du & Wang et al, 2022

Aside: Visualizing Attention Masks

If you plot an entire SxS attention matrix you can often see patterns.

Here is an example of an (unmasked) attention matrix:

Almost always see a strong diagonal- entries attending to themselves.



Individual spots show strong entry-entry relationships. In this case, “he” is “reporting”, and he is “reporting” on “Malaysia”.

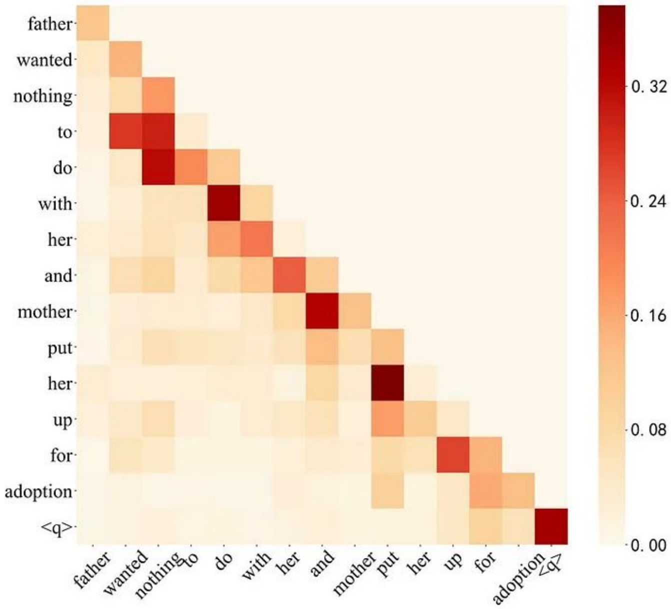
Strong (weak) vertical stripes show entries that are especially important (unimportant). Here the word “one” has little impact on the rest of the sentence.

Du & Wang et al, 2022

Aside: Visualizing Attention Masks

If you plot an entire SxS attention matrix you can often see patterns.

You can of course visualize masked attention as well:



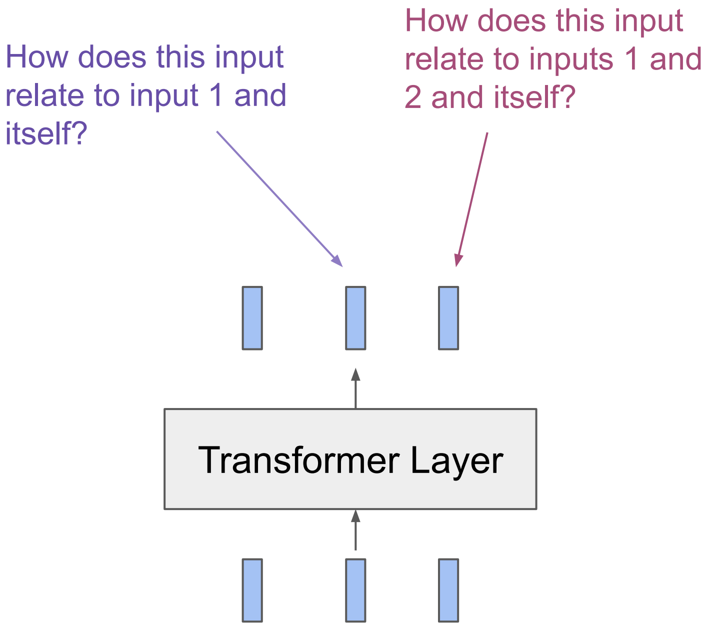
https://medium.com/@AIExplainedML/how-does-the-attention-mechanism-in-gpt-models-work-5f489a59346b

Transformer Models

What do multiple transformer layers do?

To reiterate, a transformer model is simply a stack of transformer layers.

The first layer is easy to understand: it computes how each input token relates to each other (preceding) input token.


How does this input relate to input 1 and itself?
How does this input relate to inputs 1 and 2 and itself?
Transformer Layer

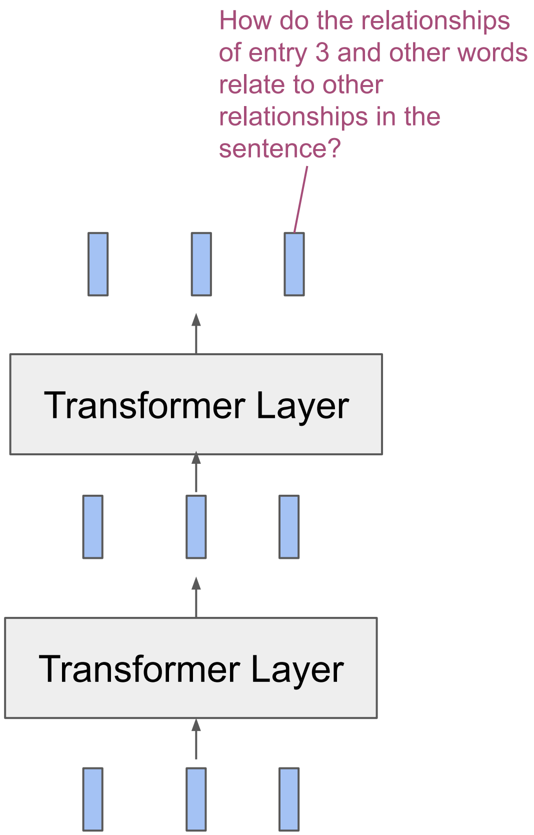
What do multiple transformer layers do?

To reiterate, a transformer model is simply a stack of transformer layers.

The first layer is easy to understand: it computes how each input token relates to each other (preceding) input token.

The second transformer layer is therefore computing something like “relationships between relationships”.

(Actually it does this **per head** in multihead attention)

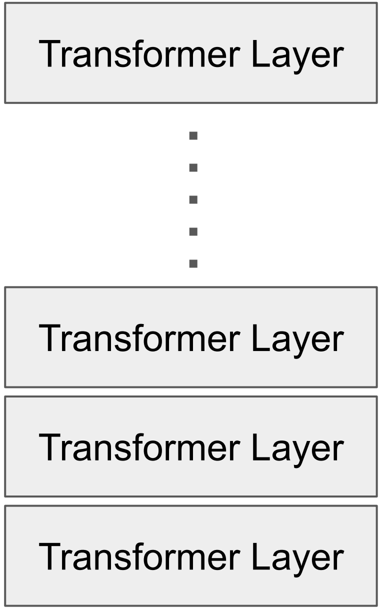

How do the relationships of entry 3 and other words relate to other relationships in the sentence?
Transformer Layer
Transformer Layer

What do multiple transformer layers do?

Many transformer layers together can therefore compute extremely complex relationships among many inputs.

This is useful when the input is long or complicated.

It is especially useful when we want to predict the next word, which may involve many interactions within the preceding text.



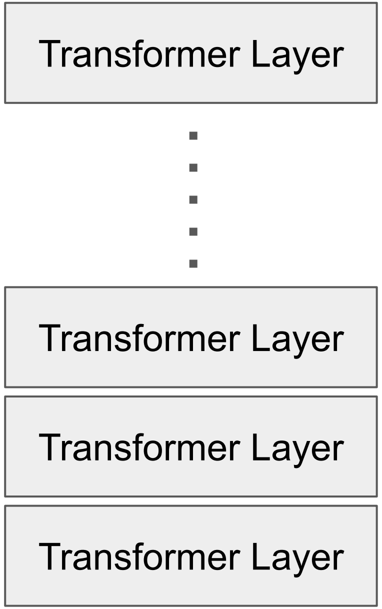
What do multiple transformer layers do?

Remember that in addition to weighting relationships, transformer layers also include a very wide MLP!

This allows them to perform computation, such as computing what the next word should be.

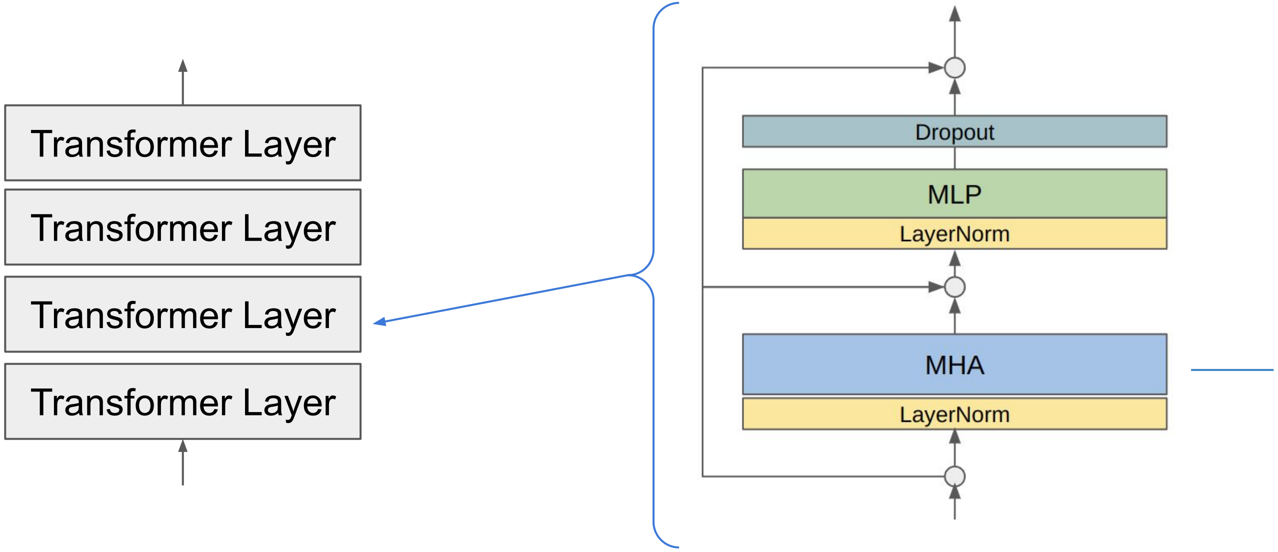
The second transformer layer is more accurately doing something like:

“Computation on relationships between [computations on relationships between individual tokens]”



Transformers are big.

Many LLMs are named according to how many parameters they have. For example “GPT3-175B”, “Mistral 7B”, or “Falcon 40B”. Where do these big parameter counts come from?



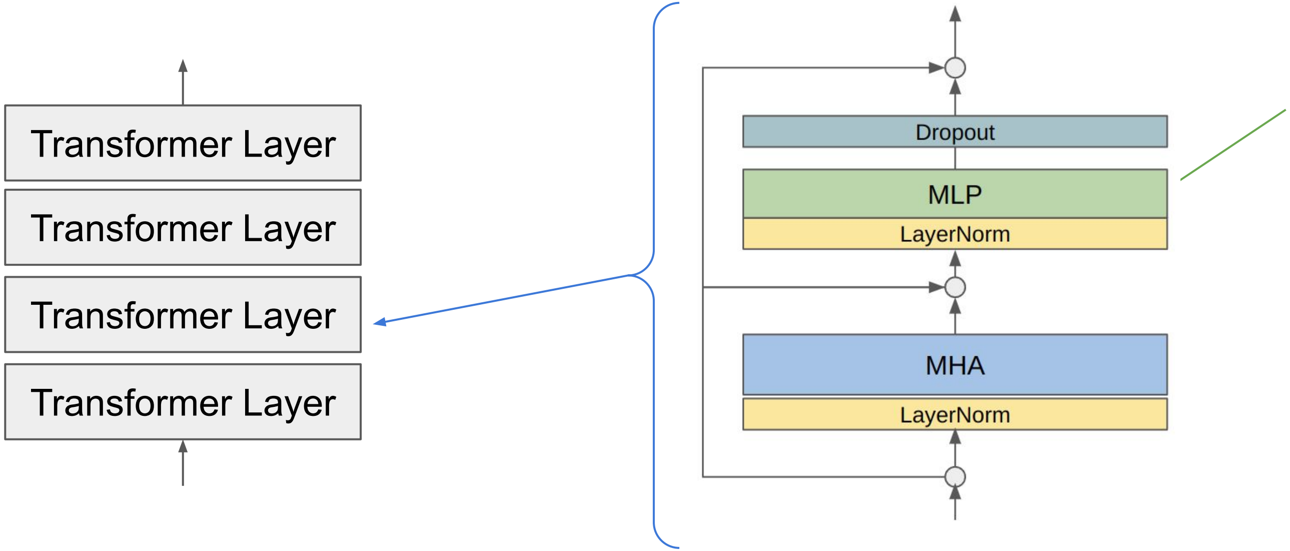
The MHA computes Q, K, and V, each with DxD matrices.

After applying attention in multiple heads, it brings the disparate outputs back together with another DxD matrix.

MHA ~= $4D^2$ parameters.

Transformers are big.

Many LLMs are named according to how many parameters they have. For example “GPT3-175B”, “Mistral 7B”, or “Falcon 40B”. Where do these big parameter counts come from?



The MLP has two layers, each with 4D*D parameters. (One to project vectors to size 4D and one to project back).

MLP ~= 8D^2 parameters

⅔ of the parameters are here!

Transformers are big.

Each transformer layer has roughly 12D^2 parameters, where D is the length of each vector.

D is determined all the way back in our token embeddings, and generally is constant throughout the model.

For a model with L layers, we therefore get:

$$\textbf{Parameters} \sim= 12(L)(D^2)$$

Parameters ~= 12(L)(D^2)

Transformers are big.

D needs to be large enough to capture the nuances of an individual word. Typically D is some large power of 2(ish):

GPT 1 (2018), L=12, D=768	P~=85M	Actual: 117M
GPT 2 (2019), L=48, D=1600	P~=1.47B	Actual: 1.5B
GPT3 (2020), L=96, D=12288	P~=174B	Actual: 175B

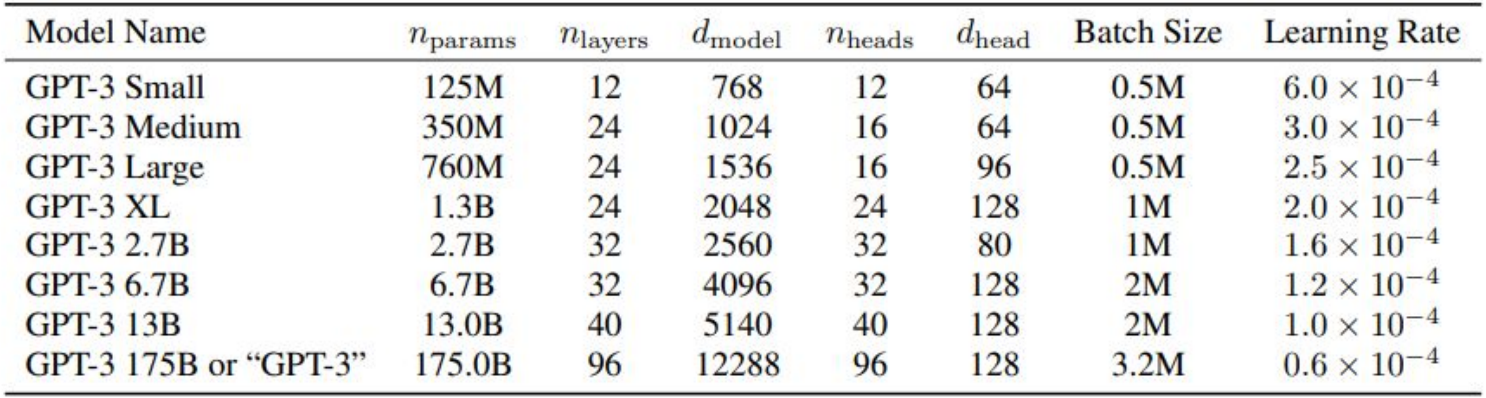
This rule-of-thumb becomes increasingly accurate for large models, which are dominated by the size (and number of) transformer layers.

This is always an underestimate because it doesn't account for embeddings and the output layer. There are more complex equations that can take all of this into account but I find the above to be nice and simple.

Architecture Hyperparameters

As we have just seen, **L**=[number of layers] and **D**=[vector dimension] are the dominant hyperparameters determining model size. These are sometimes **n$_{\text{layers}}$** and **d$_{\text{model}}$**

Additionally, we have **N**=[number of heads in MHA] (aka **n$_{\text{heads}}$**). This is usually a power of 2(ish) number that is similar to the number of layers (12 or 16 for small models, up to 96 or higher for huge models). Finally, we also have the size of the vocabulary.



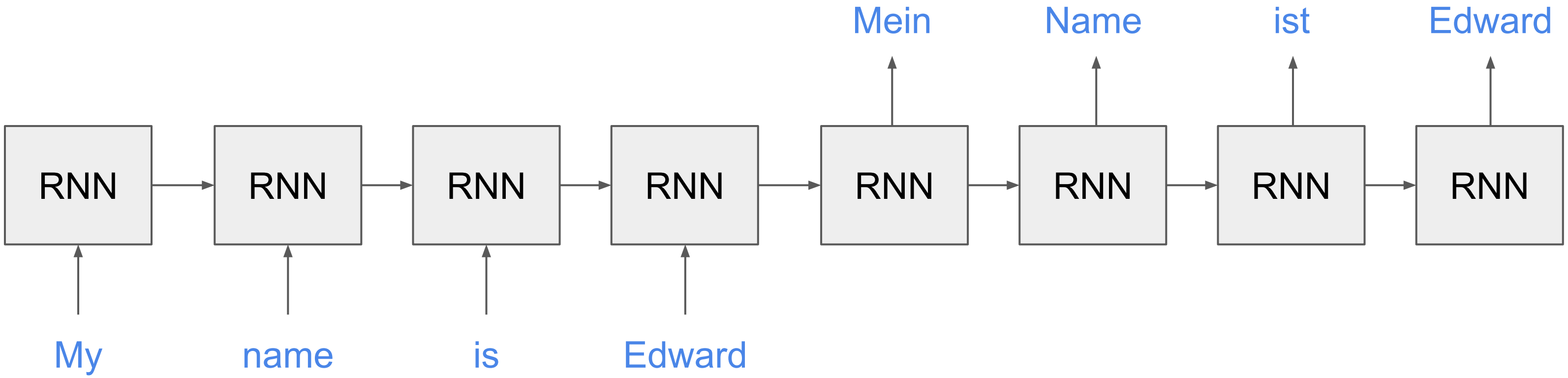
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table of model hyperparameters from GPT3 paper.

Encoders and Decoders

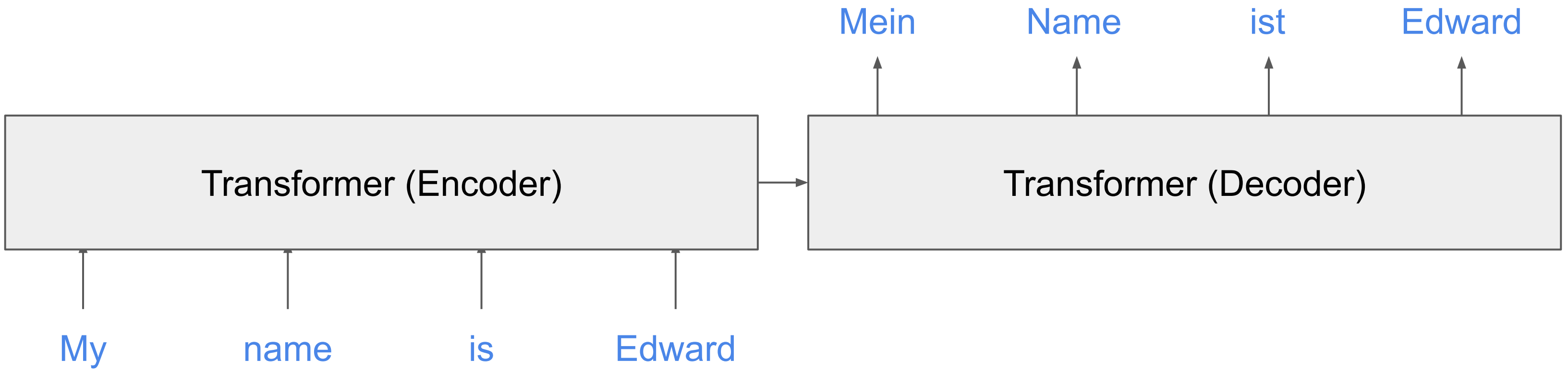
Transformer Origins

Recall the translation RNN example below. The RNN first reads in the sentence on one language, and then outputs the sentence in another language.



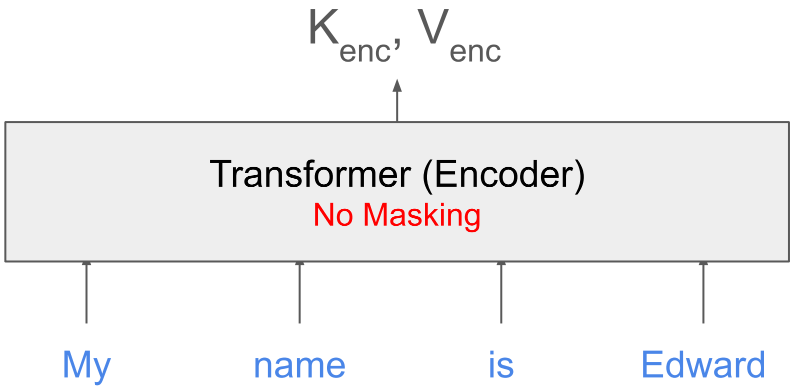
Transformer Origins

Transformers were introduced to fix the problem of long-range dependencies. The entire input is ingested at once, and then the output is produced one token at a time.



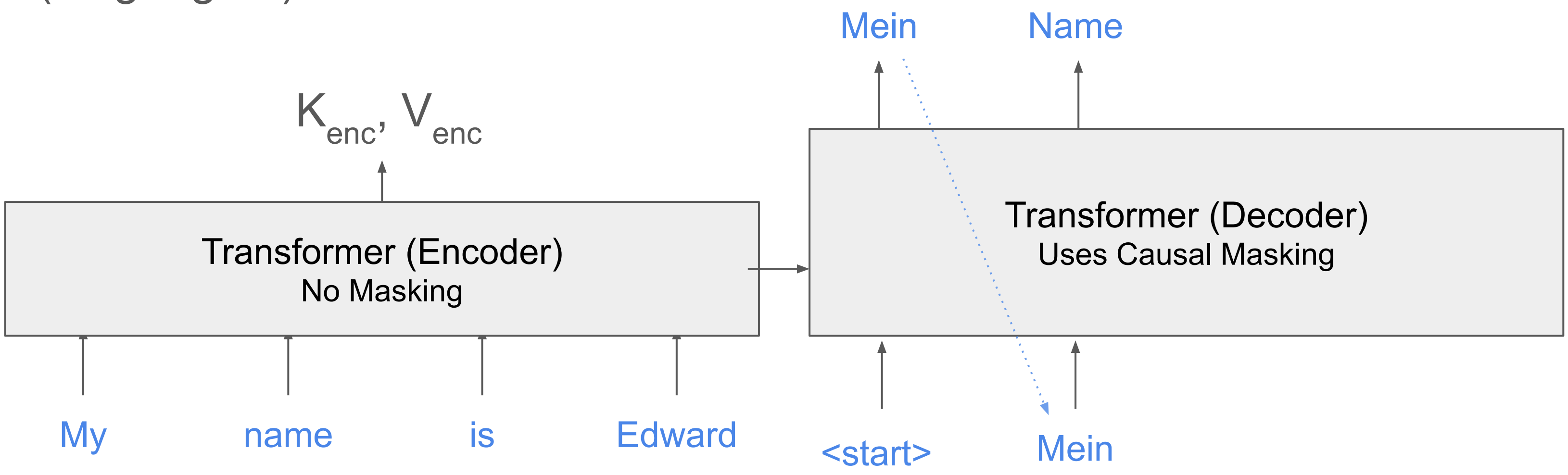
Transformer Encoder/Decoder Walkthrough

First, the entire input (language 1) is taken in by the encoder, which is several stacked transformer layers. There is no causal masking.



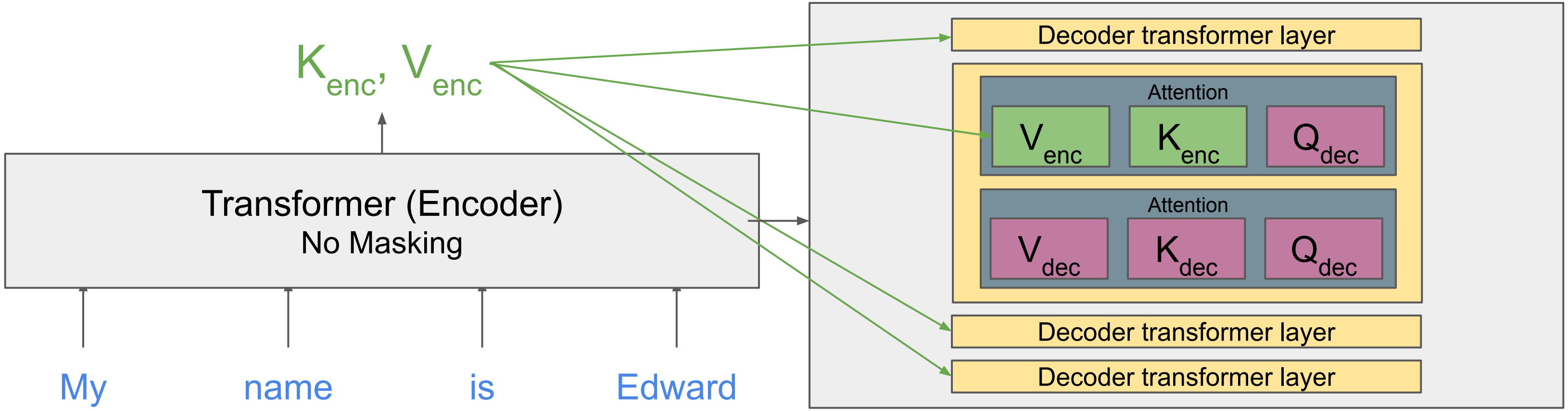
Transformer Encoder/Decoder Walkthrough

A second transformer model is used to generate the translation one token at a time (language 2).



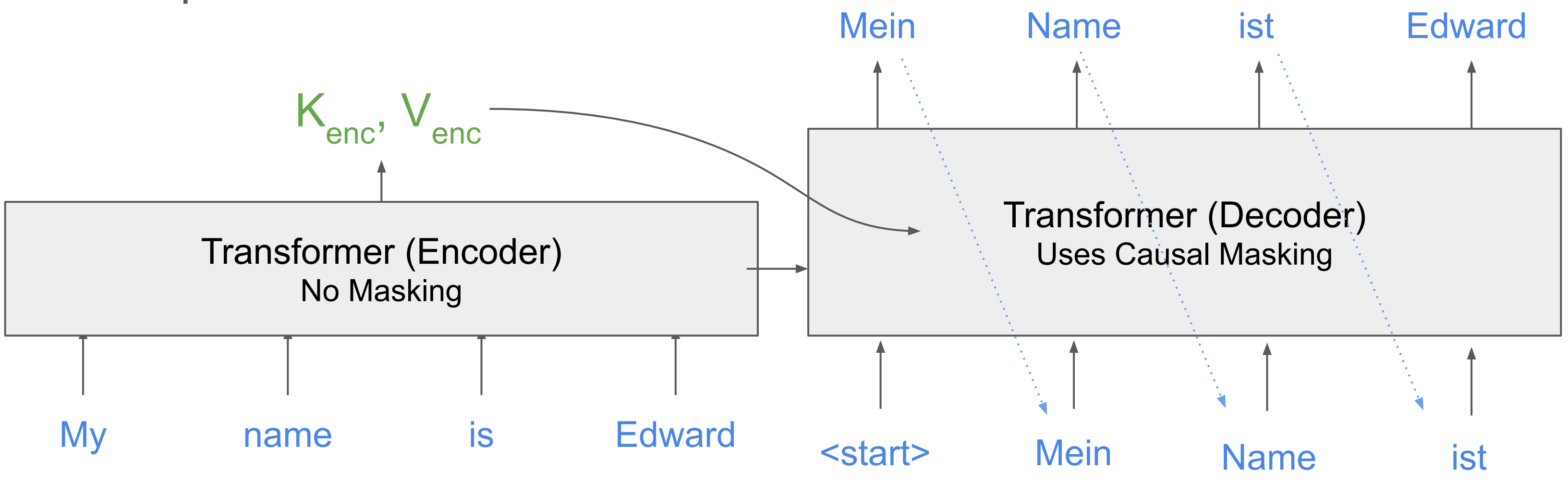
Transformer Encoder/Decoder Walkthrough

The keys and values from the encoder are used in special attention layers in the decoder which: accept Q from the previous layer and K, V from the encoder.



Transformer Encoder/Decoder Walkthrough

Each token generated by the decoder uses the current input as well as the encoder output.

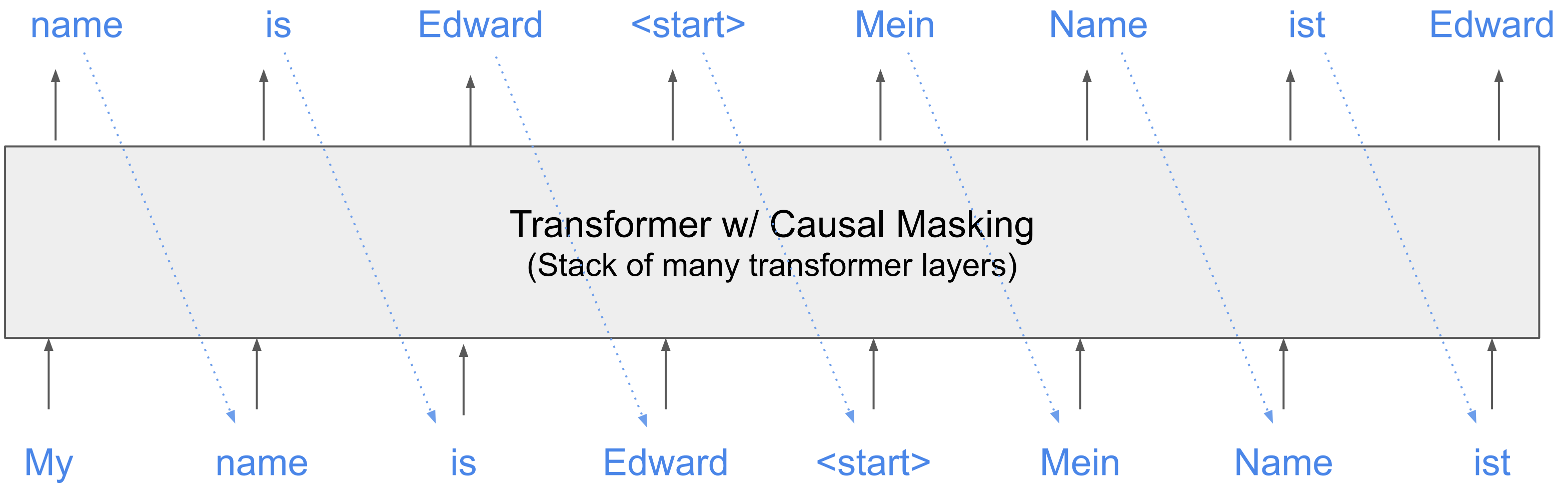


Modern Version

Modern Transformer

It turns out this is overly complicated. You can instead use self-attention for everything, and maybe add special tokens that show boundaries like “start translation” (more on these next week).

Note that this accomplishes the same thing: Each output word is based in the whole input and previous output.



Notes

Using a single transformer model is sometimes called a “decoder-only” model, since it throws out the encoder and just feeds everything through the output portion.

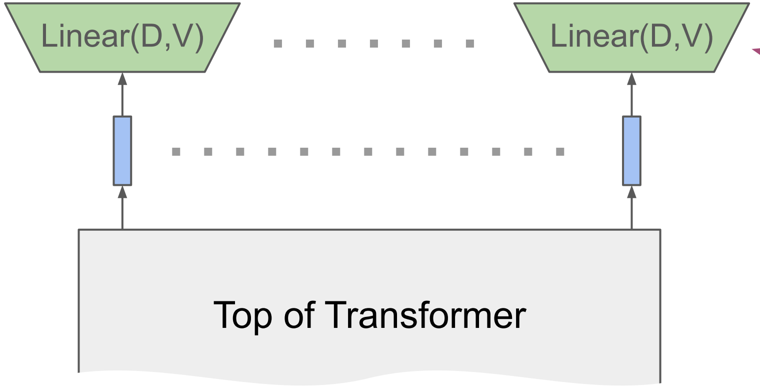
Do not get confused here! Even though it is called “decoder-only”, it does NOT use K, V from some other model. It just uses standard self-attention as we discussed last lecture.

Model Output

Output Layer

The final layer of the network needs to transform vectors of length D into a categorical distribution over the vocabulary.

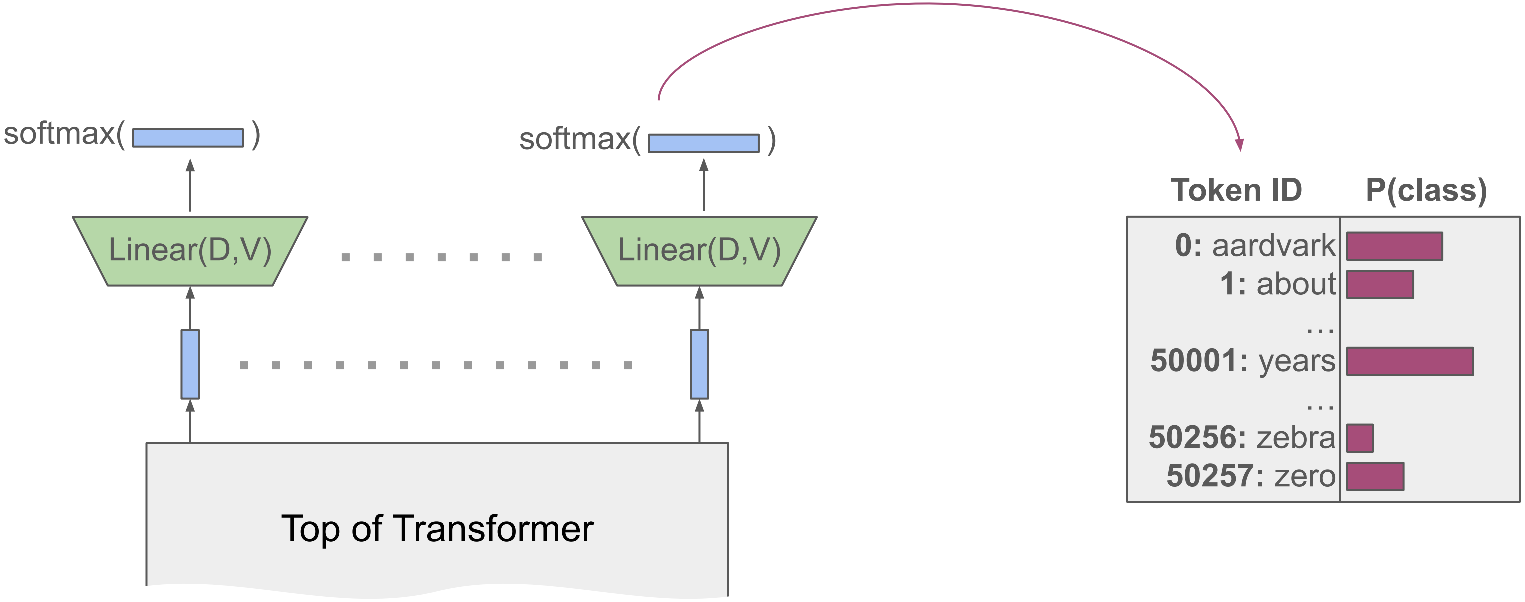
For a vocabulary size V, and tokens of dimension D, we just have a final linear layer on top that projects to the right size.



Just like the MLP in the transformer layer, this is applied to each output vector separately.

Output Layer

Finally, we apply a softmax to each output to give us a categorical distribution over the vocabulary.



(I rotated the vectors only for diagramming purposes.)

Output Layer Size

Like the rest of the transformer, this thing is large.

For GPT2, which has V~=50,000 and D=768, the final output layer has 38.6 million parameters (for one layer!).

For larger models this single layer may have 100s of millions of parameters by itself.

Review Assignments

Homework (will be released Thursday)

Programming:

- Add a causal mask to MHA
- Fill in a class that defines a transformer block including MHA
- Fill in a class that defines a complete LLM (embeddings, several transformer blocks, output head)

Reading:

https://peterbloem.nl/blog/transformers

There will be one short answer about the reading.