Transformers

What we will cover today:

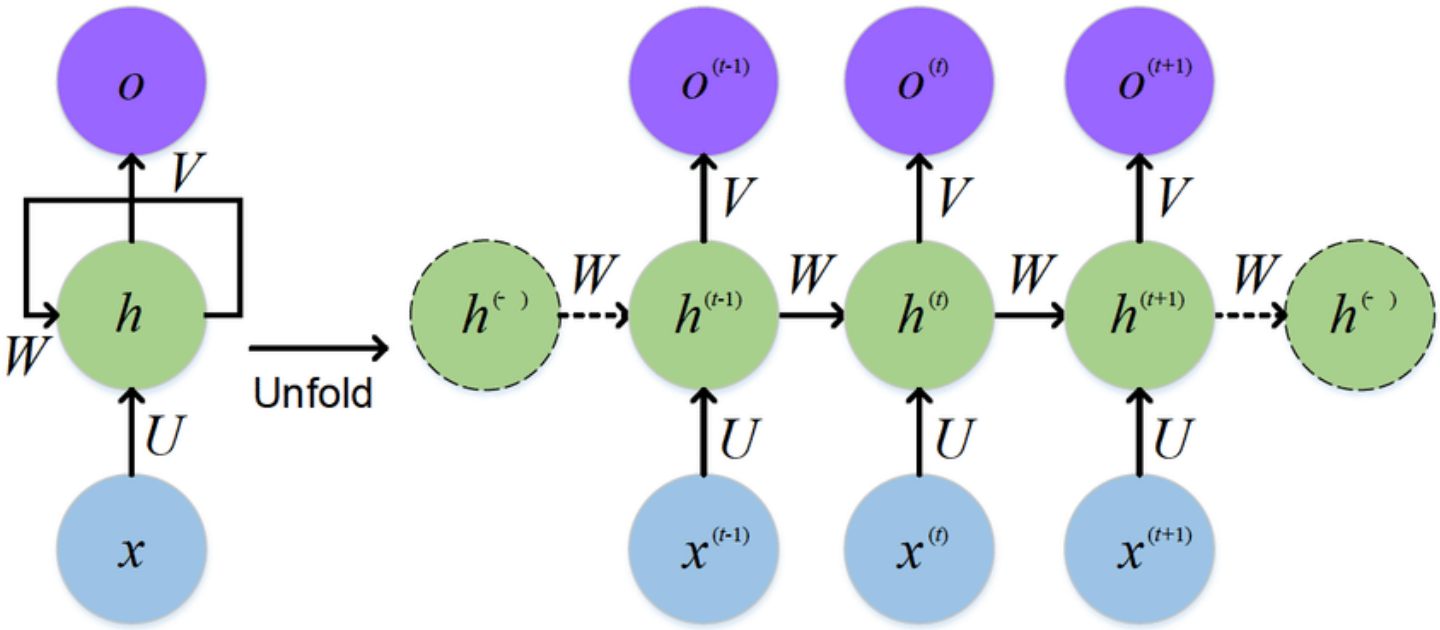
1. Why should you care about Transformers?
2. RNNs: Problems and progress
3. Transformers: Context-aware embeddings
4. Digging Deeper: What we missed
5. The `transformers` Library
6. Going further: GPT, BERT & other models
7. DL Recap & Further reading

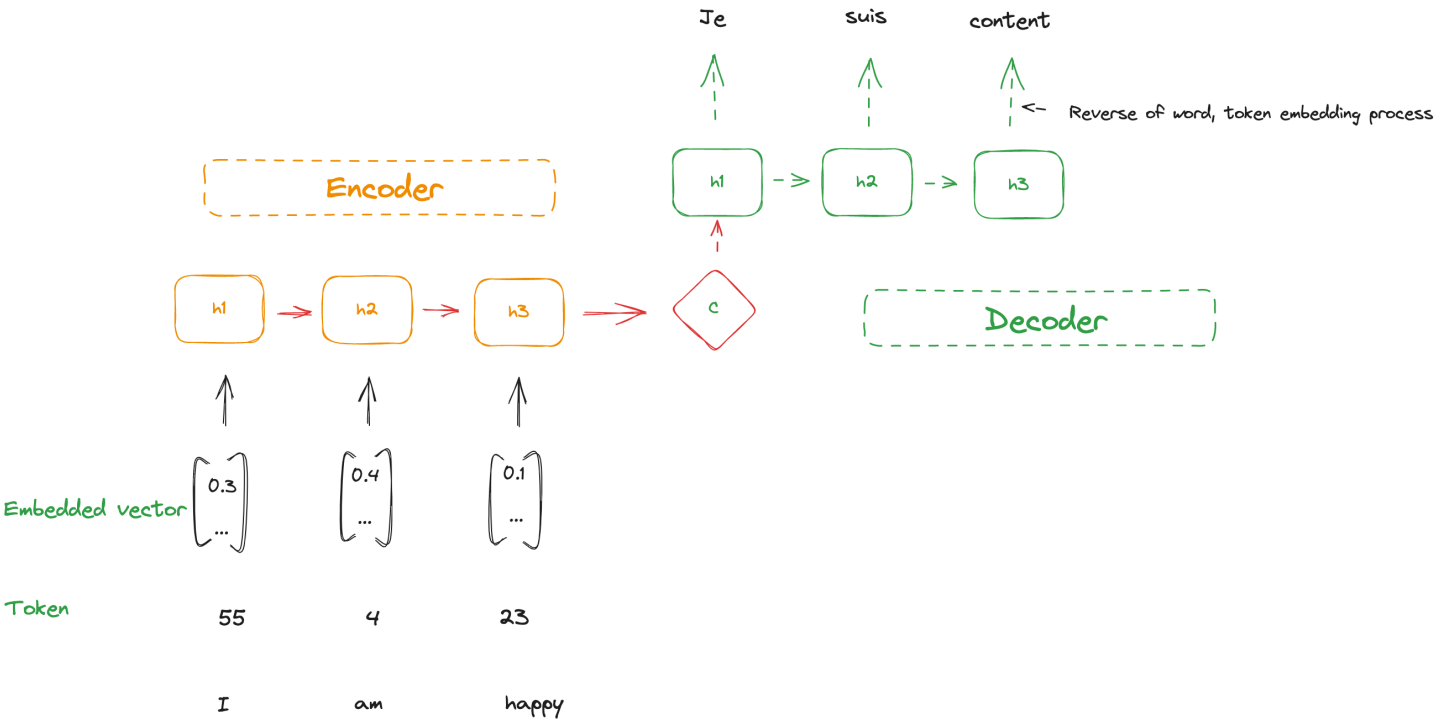
Lecturer starter notebook (https://github.com/lewagon/data-lecture-starters/blob/main/starters/06-DL_05-Transformers.ipynb)

1️⃣ Why should you care about Transformers?

2️⃣ RNNs: Problems and progress

Before we dive into Transformers, let's remind ourselves how a lot of NLP tasks are often tackled by RNNs:

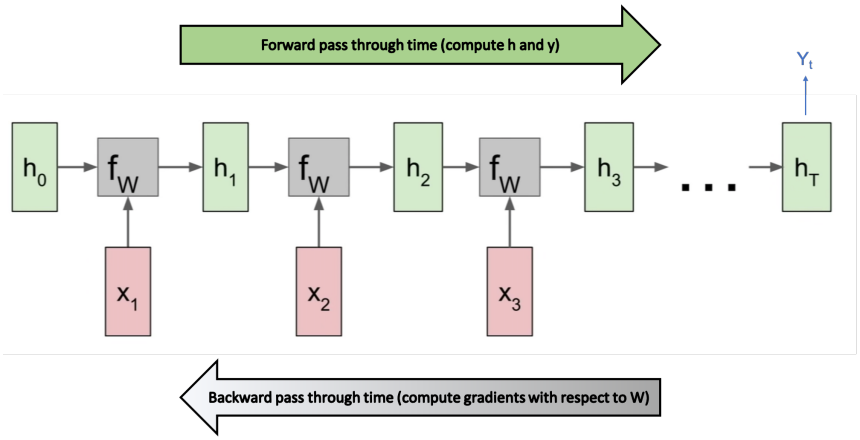

o
V
h
W
U
x
Unfold
h(-)
W
o(t-1)
V
h(t-1)
U
x(t-1)
o(t)
V
h(t)
U
x(t)
o(t+1)
V
h(t+1)
U
x(t+1)
W
h(-)


Je
suis
content
<- Reverse of word, token embedding process
Encoder
h1
h2
h3
c
Decoder
h1
h2
h3
Embedded vector
0.3
...
0.4
...
0.1
...
Token
55
4
23
I
am
happy

What are the key issues we face here?

1. Information bottleneck at interface
2. Vanishing gradient problem
3. We have to compute the entire sequence recursively (makes scaling very hard!)

RNNs suffer from vanishing gradient through time



❗ Backpropagation through time ❗ Within a single layer RNN model.

- During backpropagation, the gradient vanishes to 0 as the time step decreases.
- As a result, simple RNNs are said to have **short memory** (even with variants like LSTM and GRU)

📚 Towards Data Science - Michael Phi - Illustrated Guide to Recurrent Neural Networks (https://medium.com/towards-data-science/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9)

What does this mean for our performance?

A simplification of problems with RNNs:

> Sally adored reading; when she received a book on her birthday she was older

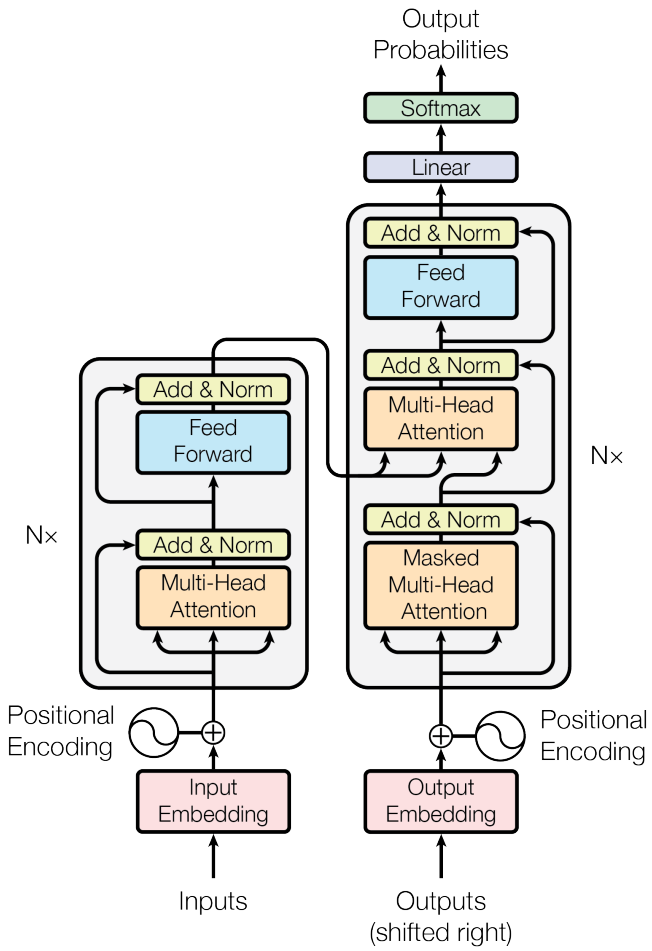
What we'd like:

> Sally adored reading; when she received a book on her birthday she was happy!

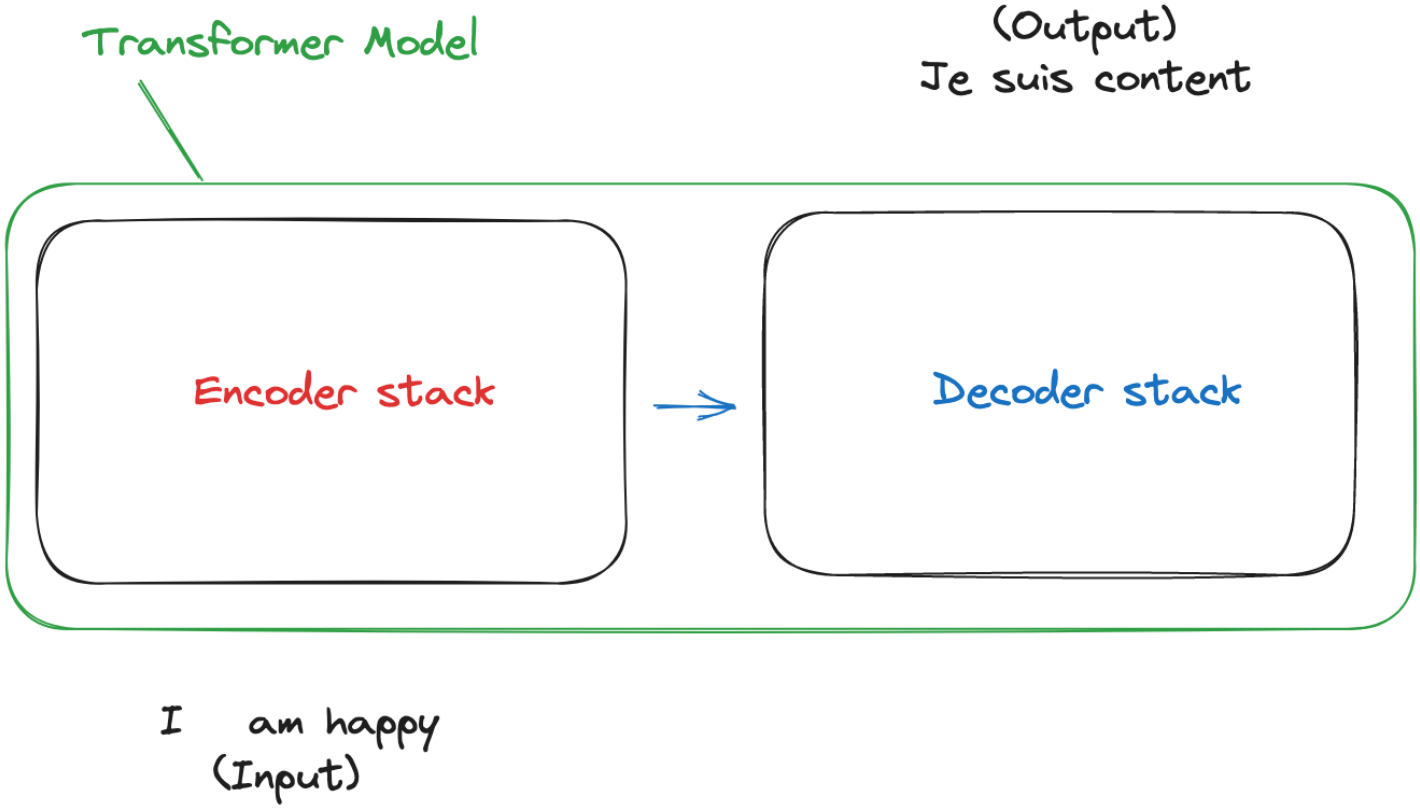
RNNs are likely to miss out on **important context** from earlier in the sentence because of their recency bias 🫠

3️⃣ Transformers

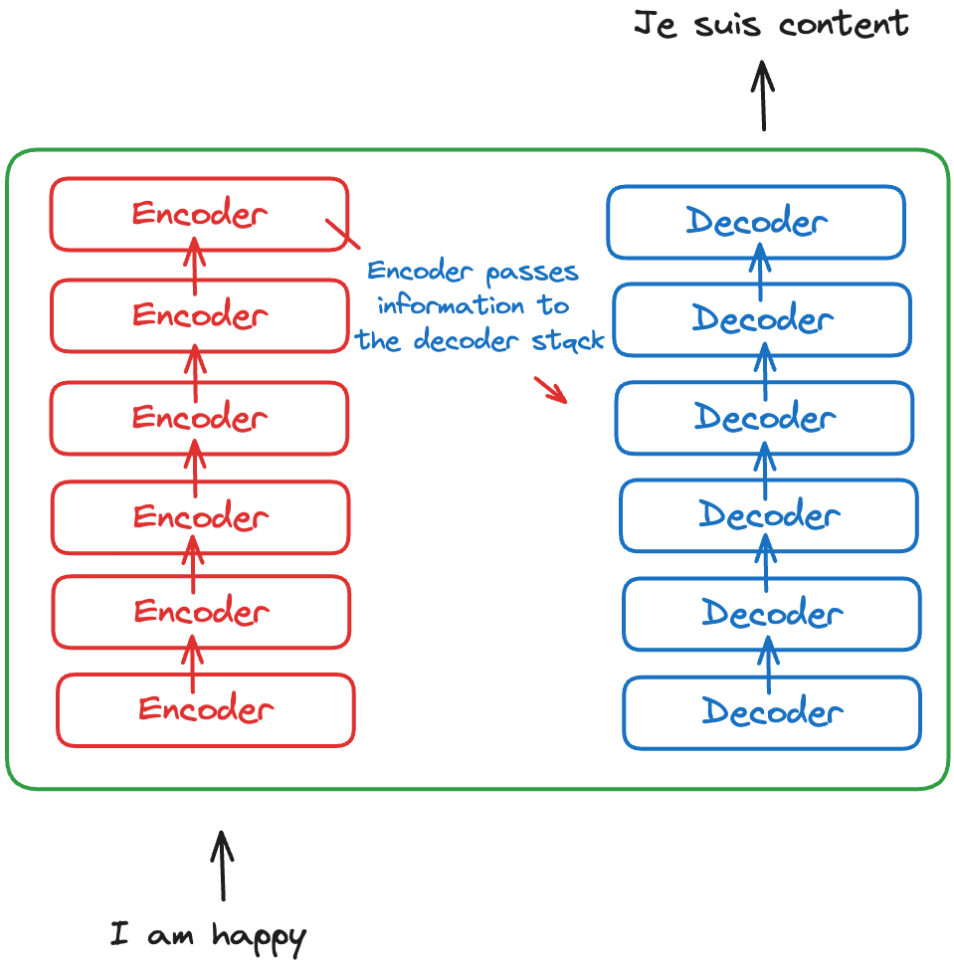
The paper that started it all: [Attention is All You Need (https://arxiv.org/abs/1706.03762)](https://arxiv.org/abs/1706.03762)



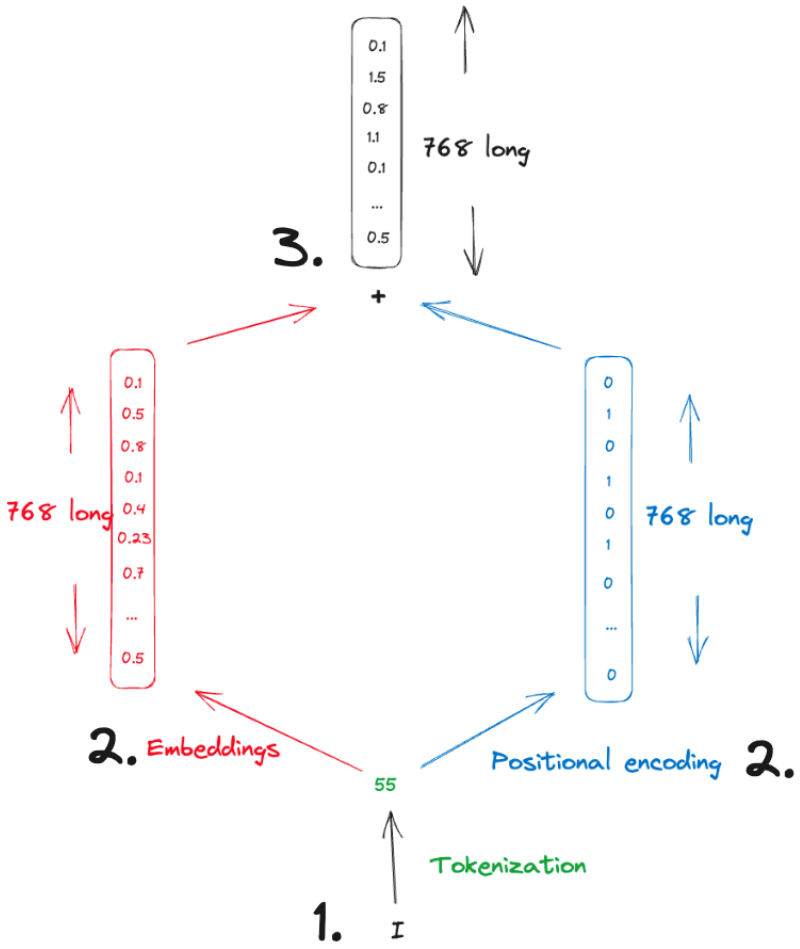
The highest level view:



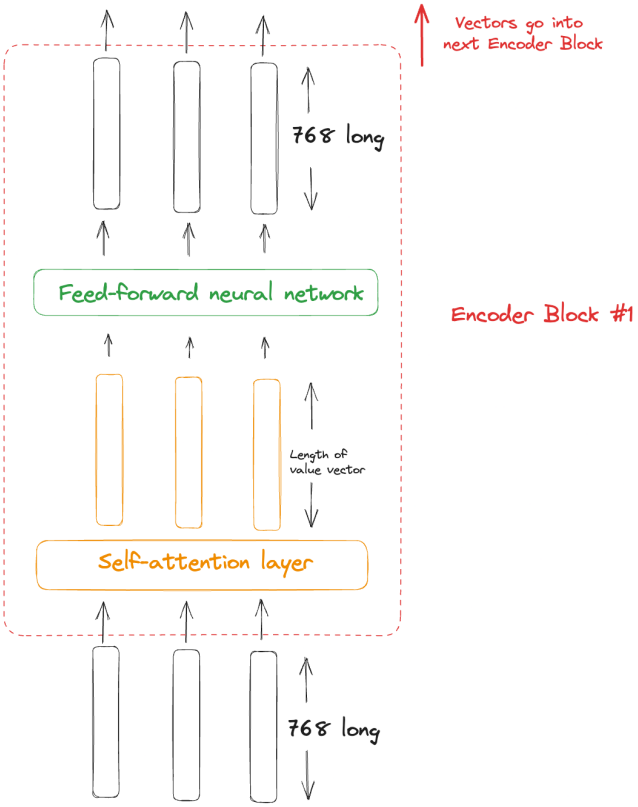
Broken down a bit more:



Before we talk about what's going on inside the encoder layers, let's talk about what's going into it!


0.1
1.5
0.8
1.1
0.1
...
0.5
768 long
3.
+
0.1
0.5
0.8
0.1
0.4
0.23
0.7
...
0.5
768 long
0
1
0
1
0
1
0
...
0
768 long
2. Embeddings
Positional encoding 2.
55
Tokenization
1. I

Zooming in on one of the encoders:



- Usually the **embedding size** is 768
- The **hidden dimension** (the length of the projected Q, K and V vectors) is also 768
- In the example we'll use size 2 for simplicity

❓ What's going on in this strange self-attention layer?

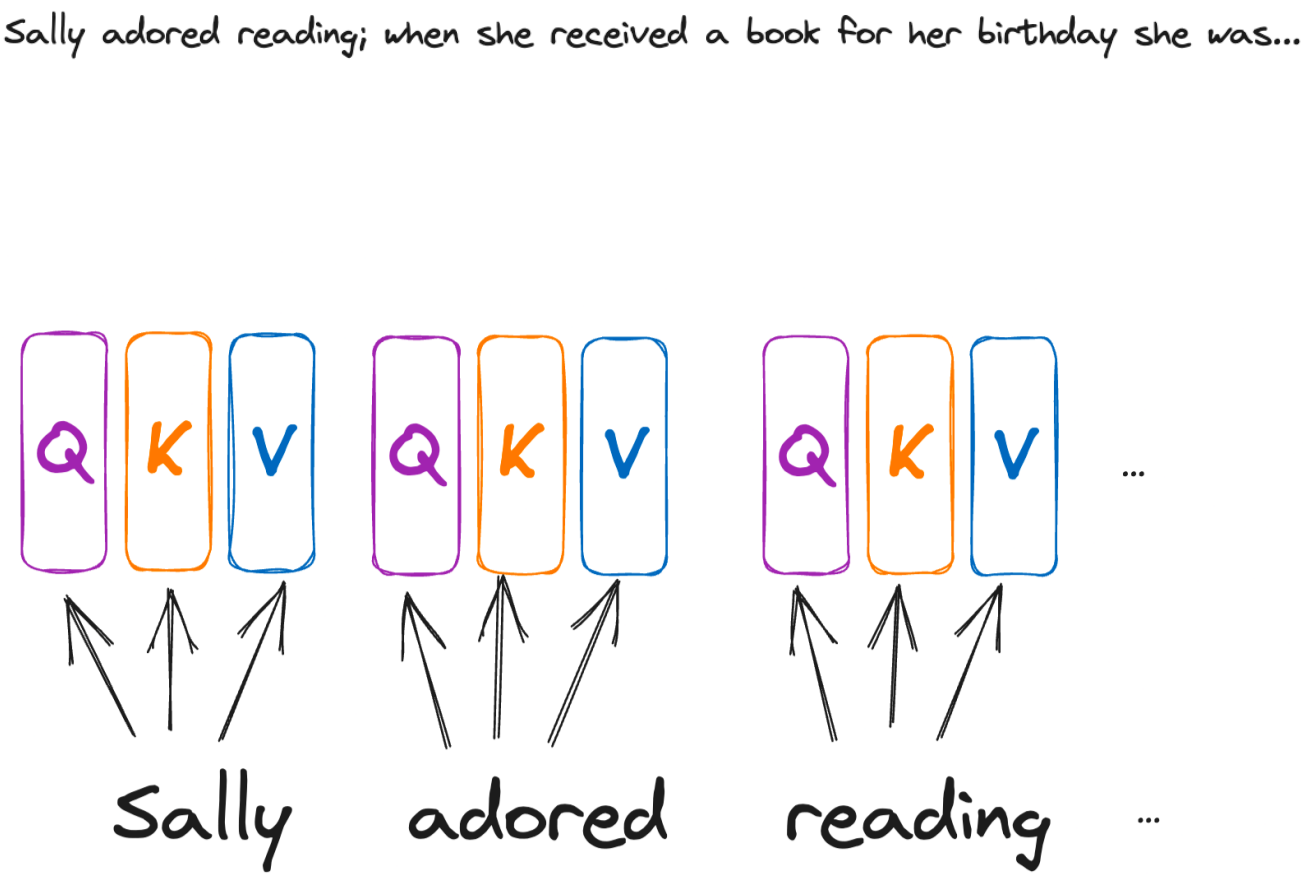
1) Each token (word) embedding gets **projected** ➡️ into 3 further vectors: the **query, key and value vectors**

2) We compute a **scaled dot-product** 🔴 on the query and key vectors to work out how much each word relates to those around it

3) Take these scores and **normalize with softmax** ↴

4) **Multiply by our value vectors** ✖️, sum and pass to our dense neural network.

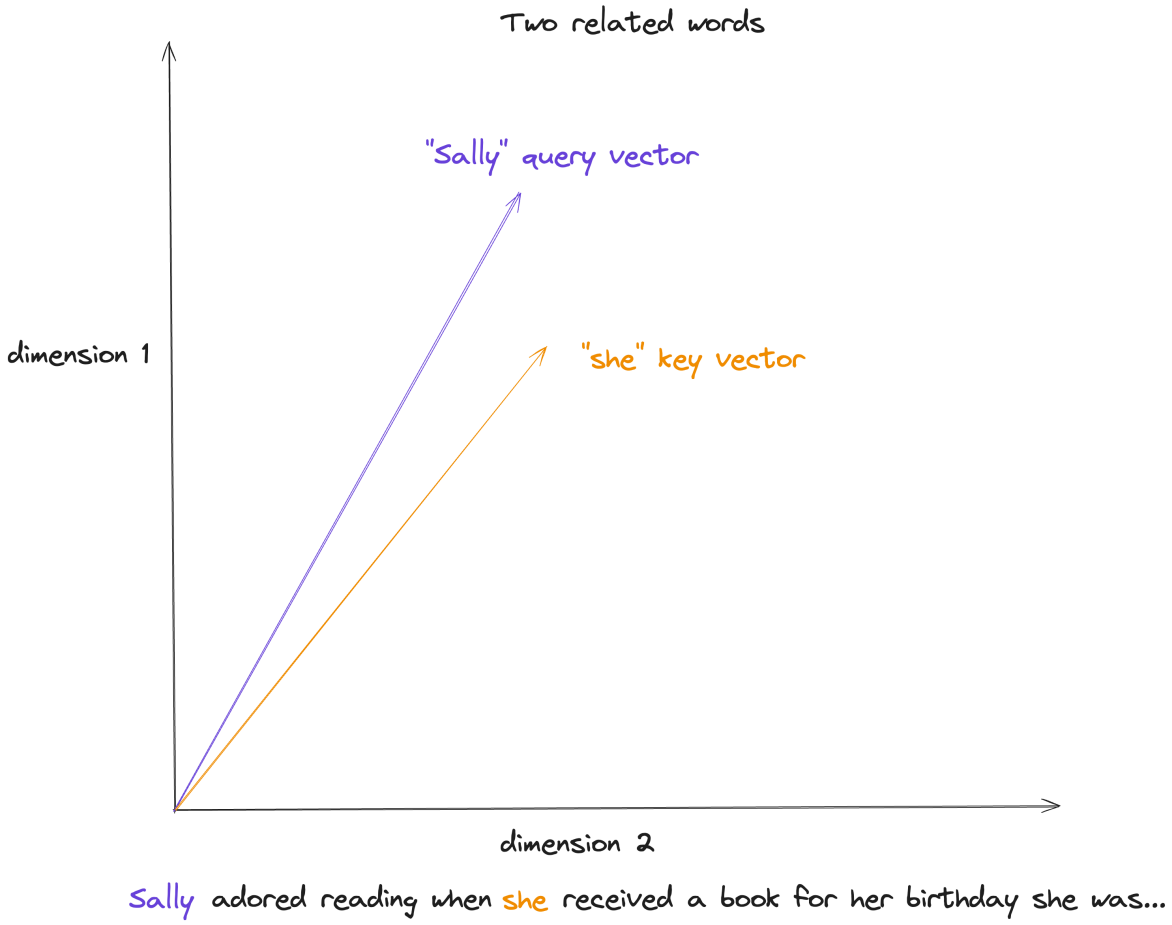
An example sentence



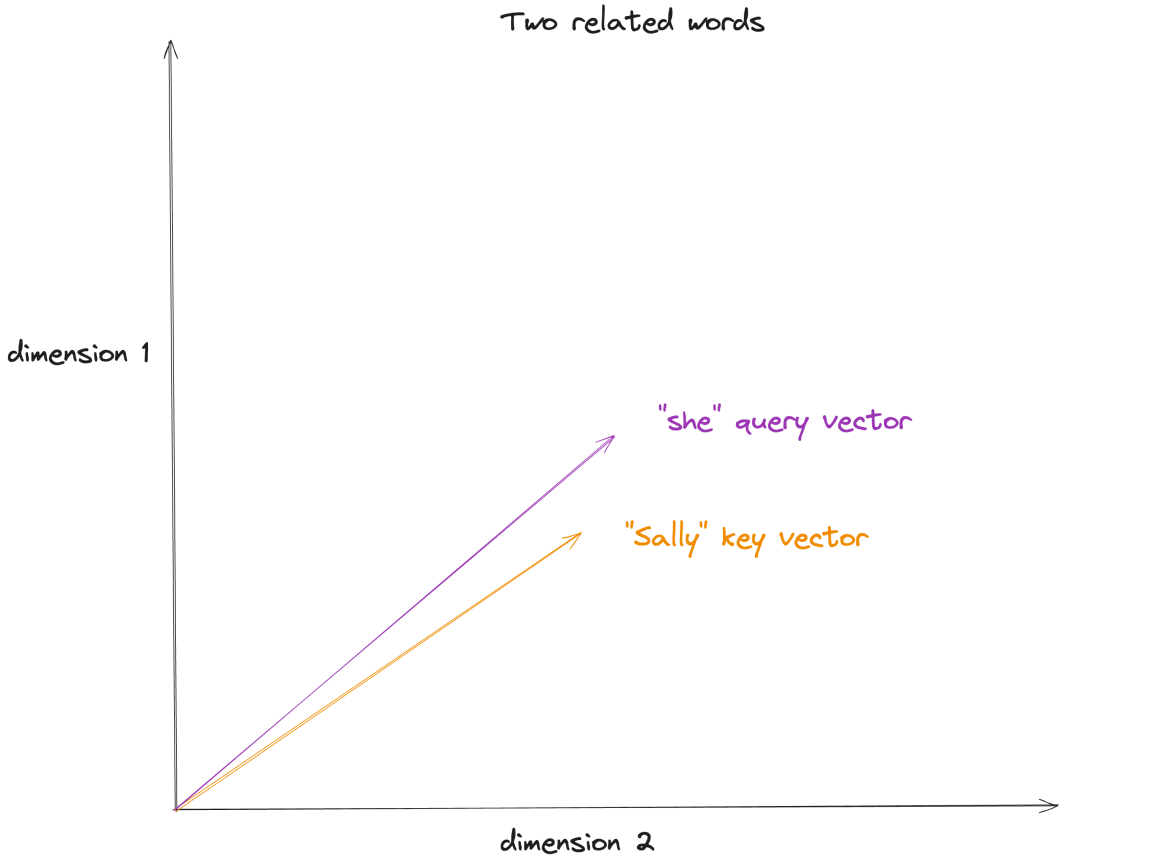
🧠 Each of these three vectors are **learned** as the model sees more data

Three zoomed-in examples:

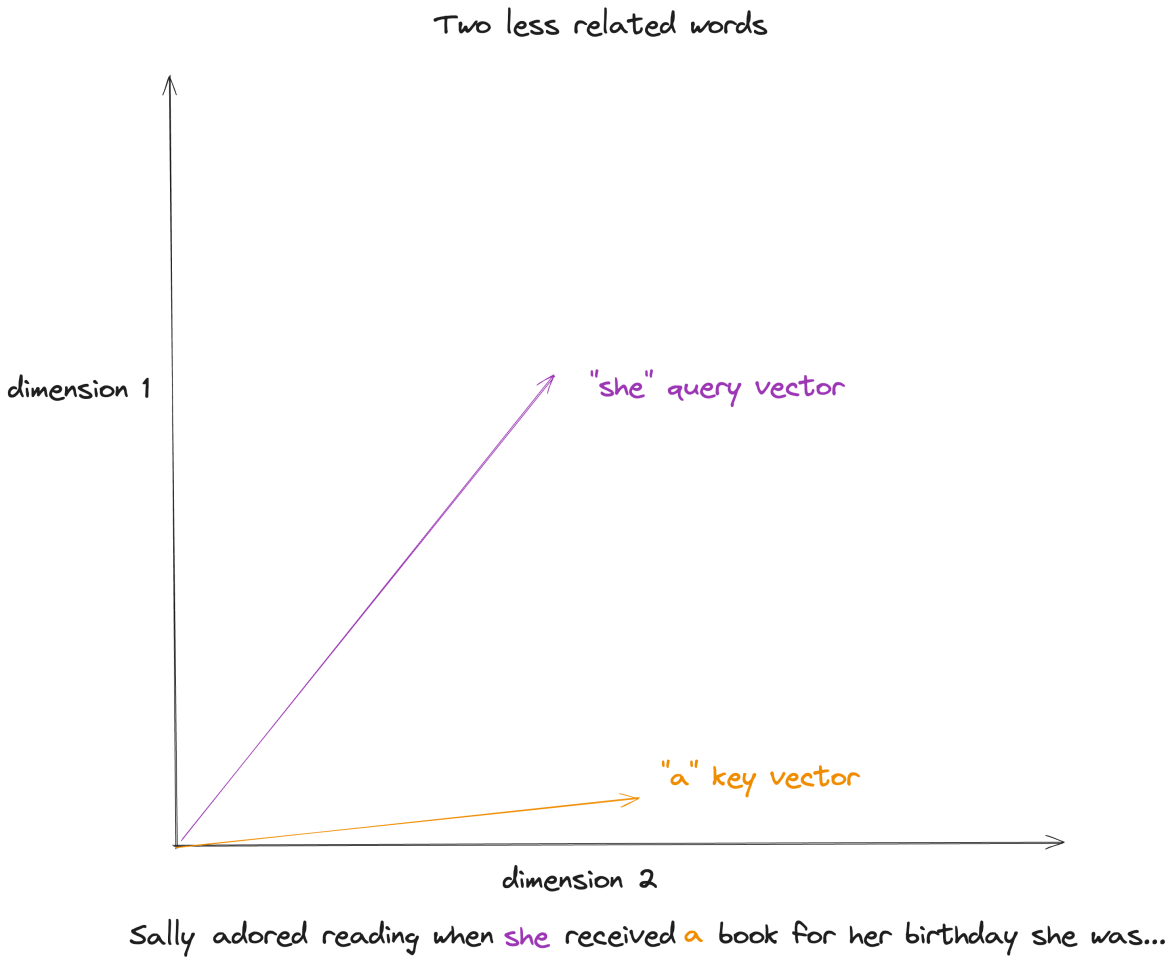
Two words in a sentence that are closely related



The same two words but seen from the other perspective:

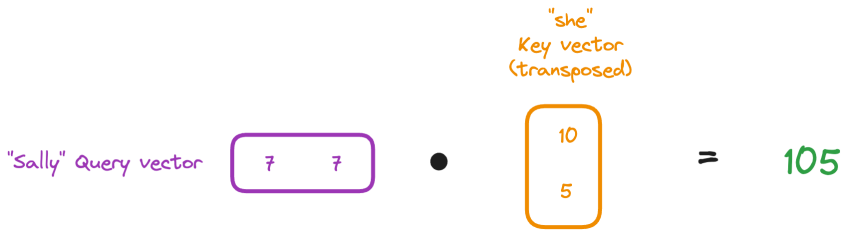


Finally, two words with a weak connection:



Let's look at one dot product

To keep it really simple, we're going to imagine our Q, K and V have only been projected into two dimensions



What happens once we have our dot products?

Dot-product between vectors

	$Sally_K$	$adored_K$	$reading_K$	$when_K$	she_K
$Sally_Q$	112	75	60	12	105

Then we scale:

Scaled dot-product between vectors

We divide by $\sqrt{hidden\_dimension}$ of embedding

In our case the root of 2 ≅ 1.4

	$Sally_K$	$adored_K$	$reading_K$	$when_K$	she_K
$Sally_Q$	80	53	43	8.5	75

Finally we apply softmax:

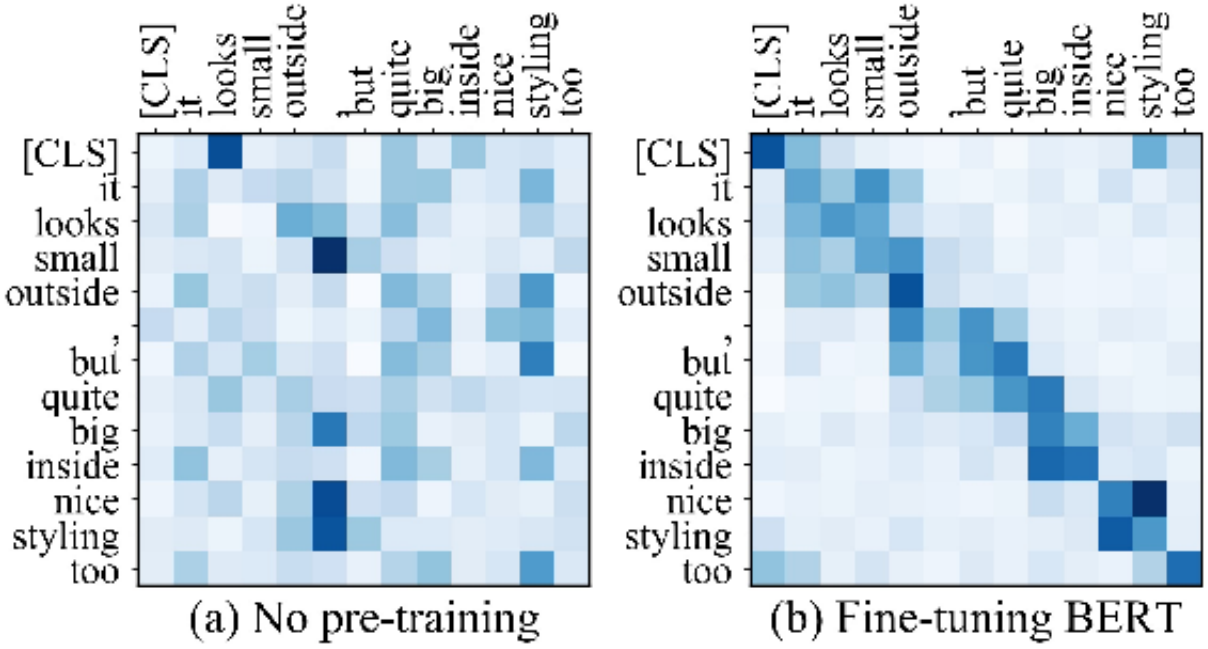
Apply softmax

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

	$Sally_K$	$adored_K$	$reading_K$	$when_K$	she_K
$Sally_Q$	0.48	0.08	0.03	0.01	0.32

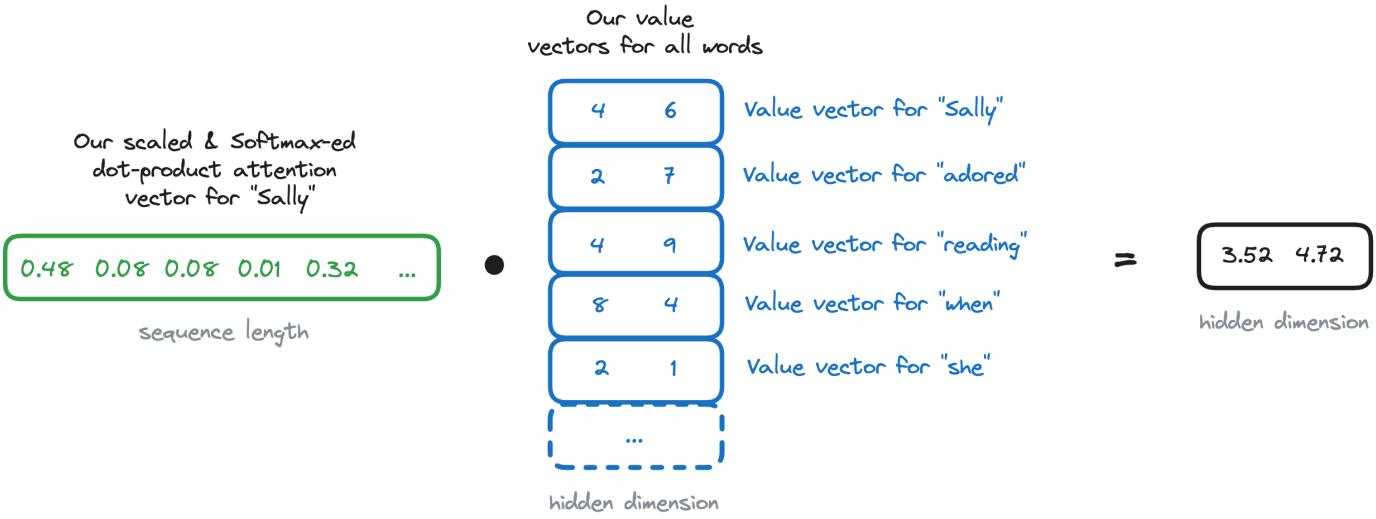
We have to do this for each word in our sentence
You can see how this becomes a matrix operation 💪
We get the scaled dot-product attention for all of our Query and Key vectors:

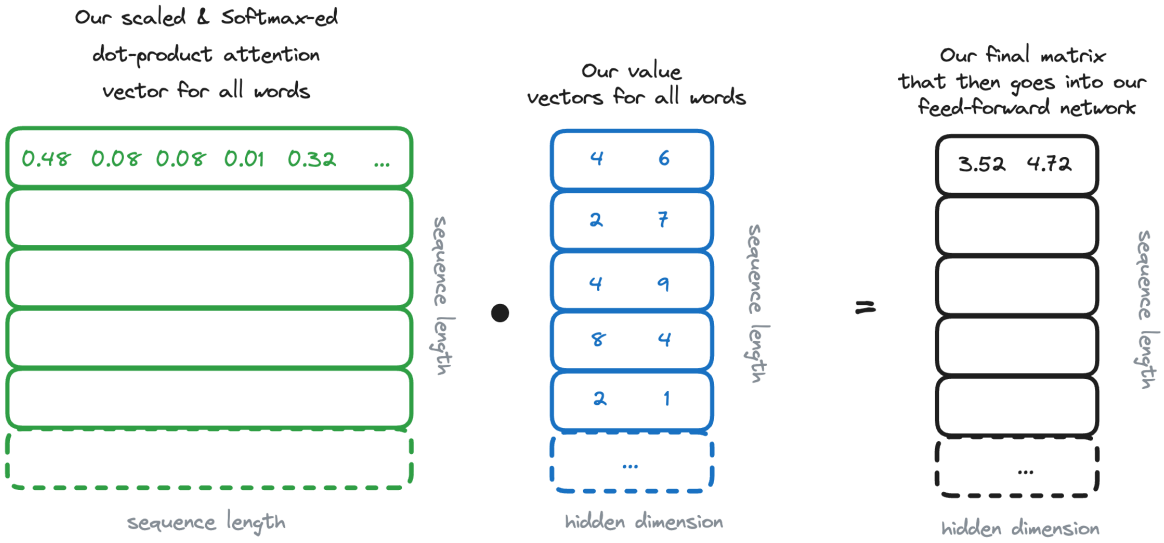
	$Sally_K$	$adored_K$	$reading_K$	$when_K$	she_K	...
$Sally_Q$	0.48	0.08	0.03	0.01	0.32	...
$adored_Q$	0.04	0.03	0.52	0.3	0.01	...
$reading_Q$	0.01	0.02	0.13	0.86	0.01	...
$when_Q$	0.01	0.03	0.02	0.58	0.01	...
she_Q	0.13	0.12	0.02	0.01	0.04	...
...	...	...	...	...	...	



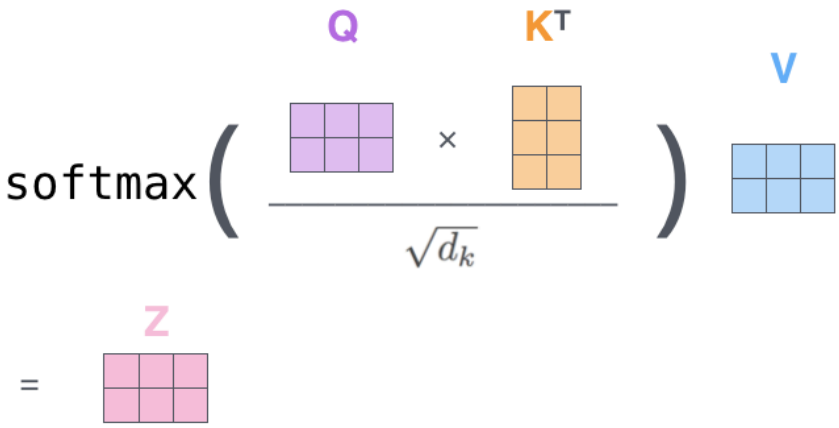
(a) No pre-training (b) Fine-tuning BERT

And then multiply our "similarity score" with all of our Value vectors



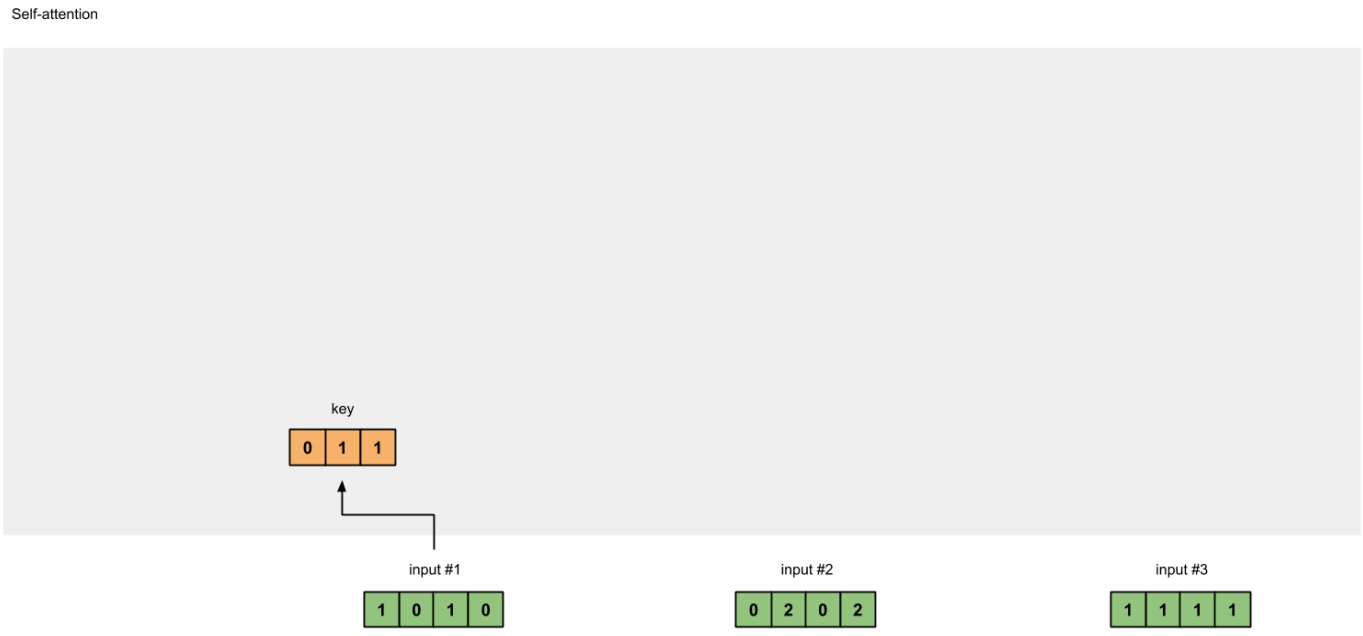


So really the entire thing can be written like this:



We are done with our multiplications
Now we just need to normalize and pass through to the feed-forward neural network
The neural network will output vectors of our original embedding dimension (e.g. 768)

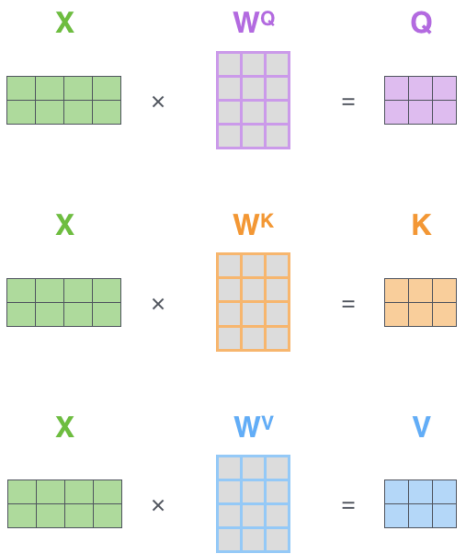
Putting it all together:



Let's check the lecture notebook for some Tensorflow visualizations

🤯 Computing one set of all of these Q, K, V multiplications and processes is what we call "single-headed attention".

When we are doing our initial linear projections (used to create the Q, K and V vectors) we can express these operations as matrices of weights too!



With that in mind, we see the complexity of our model compared to the others:

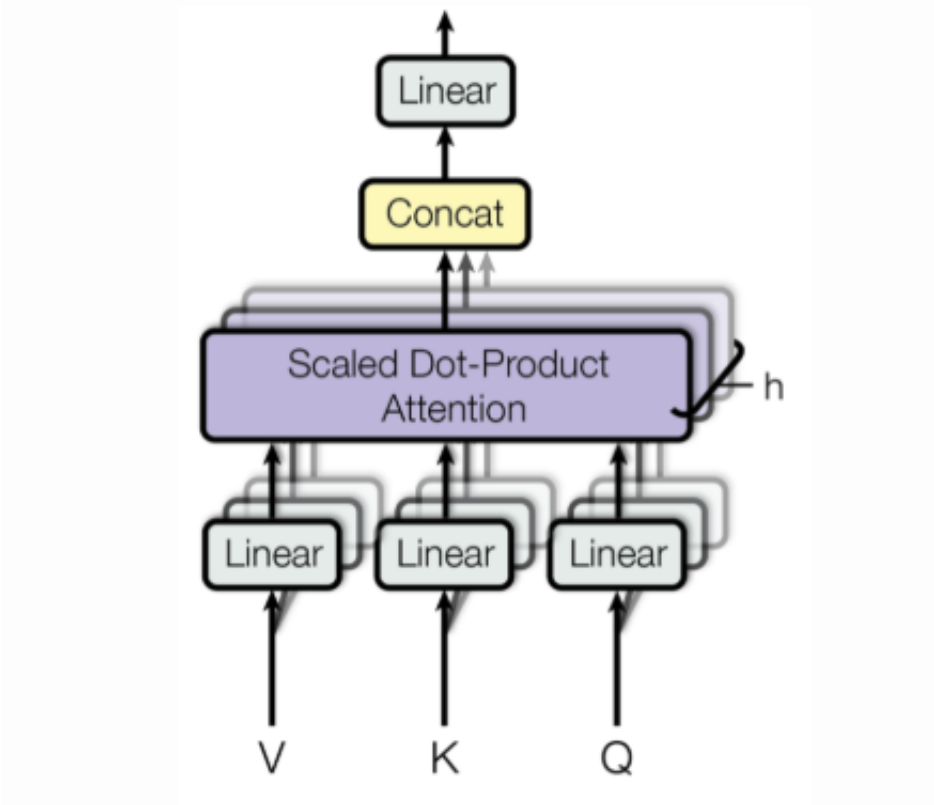
	Ops	Activations
Attention (dot-prod)	$n^2 \cdot d$	$n^2 + n \cdot d$
Attention (additive)	$n^2 \cdot d$	$n^2 \cdot d$
Recurrent	$n \cdot d^2$	$n \cdot d$
Convolutional	$n \cdot d^2$	$n \cdot d$
Multi-Head Attention with linear transformations. For each of the h heads, $d_q = d_k = d_v = d/h$	$n^2 \cdot d + n \cdot d^2$	$n^2 \cdot h + n \cdot d$
Recurrent	$n \cdot d^2$	$n \cdot d$
Convolutional	$n \cdot d^2$	$n \cdot d$

n = sequence length d = depth k = kernel size

Context windows (a.k.a. our max sequence lengths) add a lot of weights, however they have been getting much larger lately (https://www.anthropic.com/index/100k-context-windows)!)

Multi-headed?

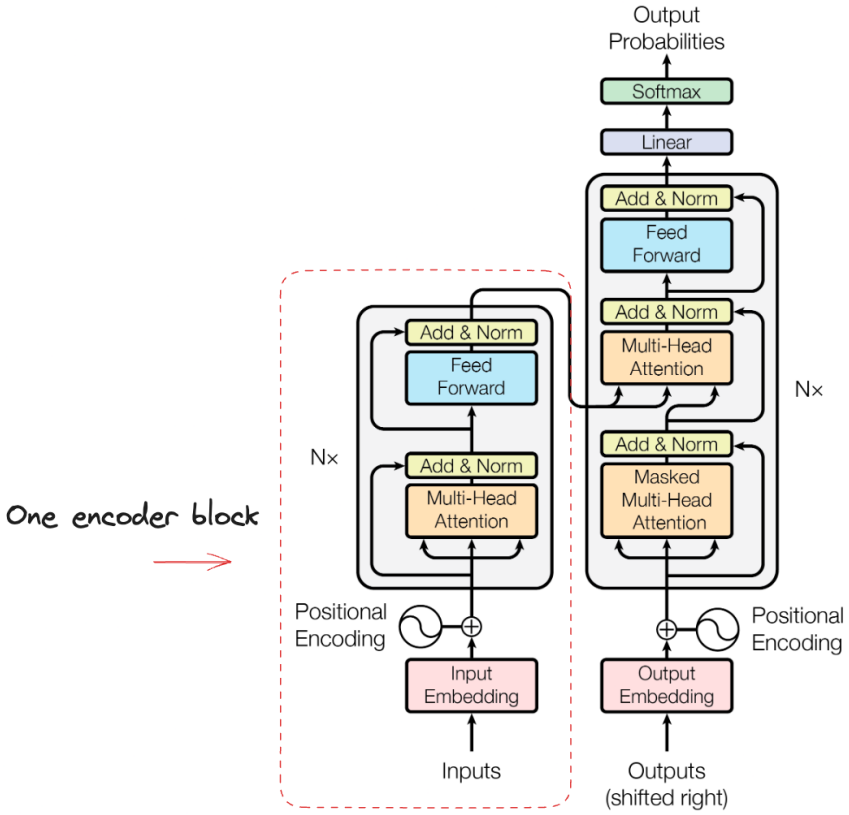
- We can use multiple heads to split up and analyze different parts of our embedding
- Each can focus a different part of the embedding eg. working on semantic vs syntactic features of our sentences 🤯🤯🤯



Let's visualize the attention weights generated by an example, pre-trained model

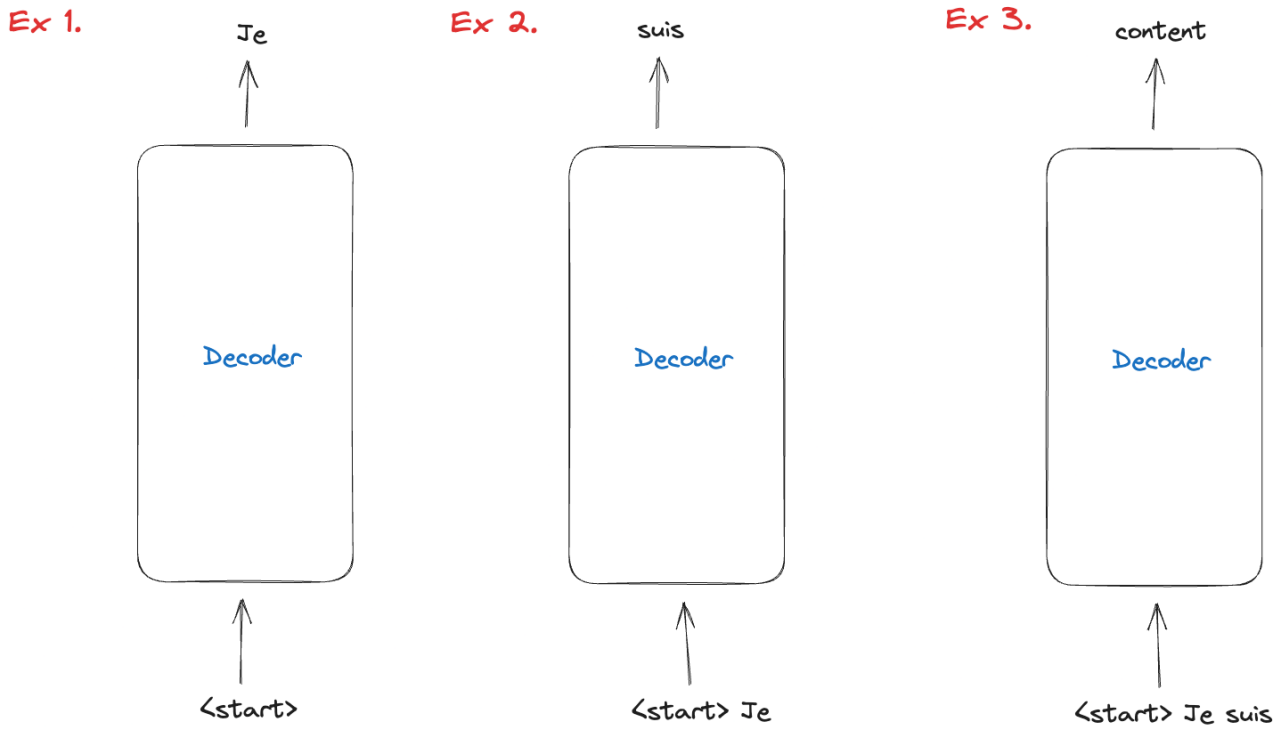
In [ ]:
```
from bertviz import head_view
from transformers import AutoModel, AutoTokenizer

model = AutoModel.from_pretrained("bert-base-uncased", output_attentio
ns = True)

first_sentence = "The lawyer worked on the case"
second_sentence = "I pushed shift to make the letters upper case"

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
viz_input = tokenizer(first_sentence, second_sentence, return_tensors
= "pt")
attention = model(**viz_input).attentions
starter = (viz_input.token_type_ids == 0).sum(dim = 1)
tokens = tokenizer.convert_ids_to_tokens(viz_input.input_ids[0])
head_view(attention, tokens, starter, heads = [8])
```

Let's check in with our original diagram:



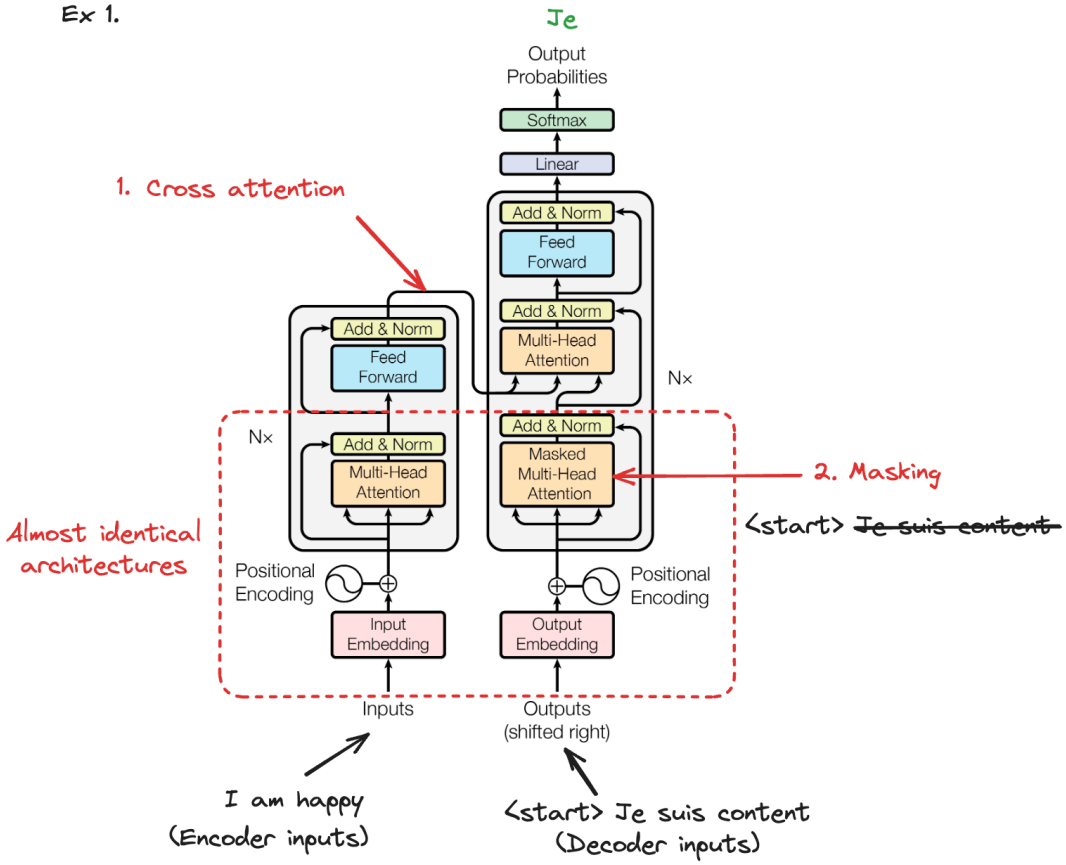
What about the decoder?

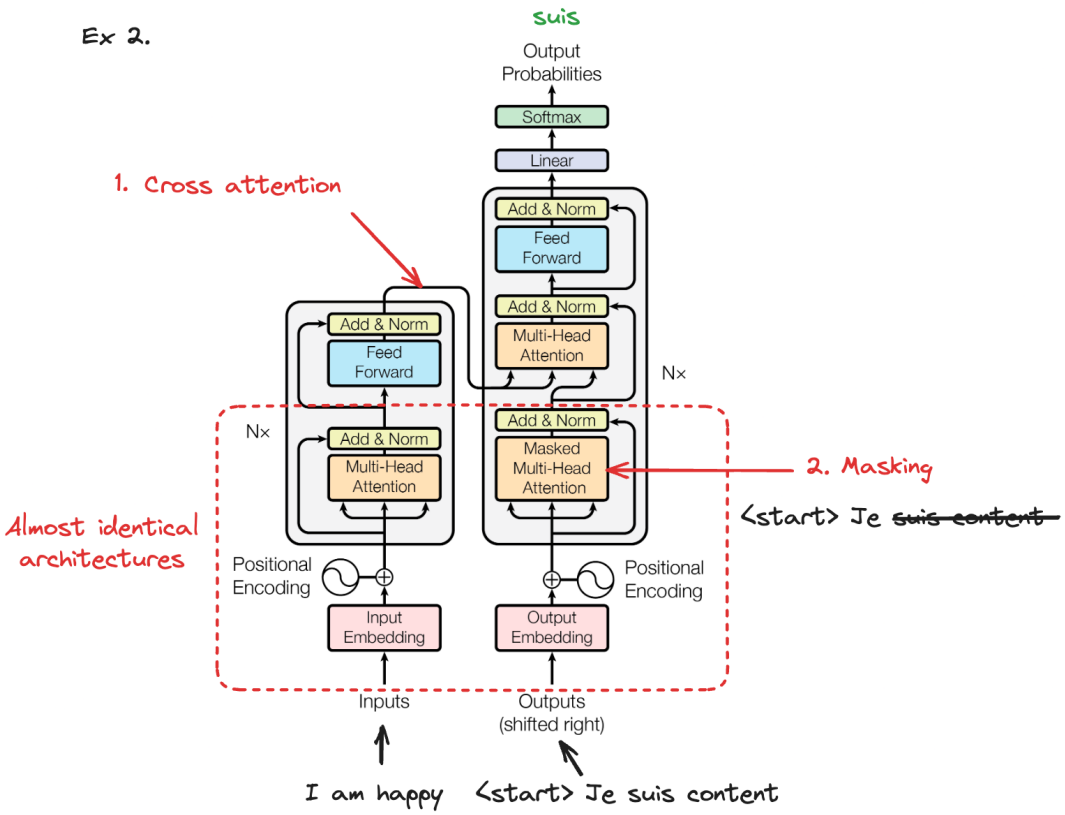
At the highest level view, its job is to choose the **most likely next token**:



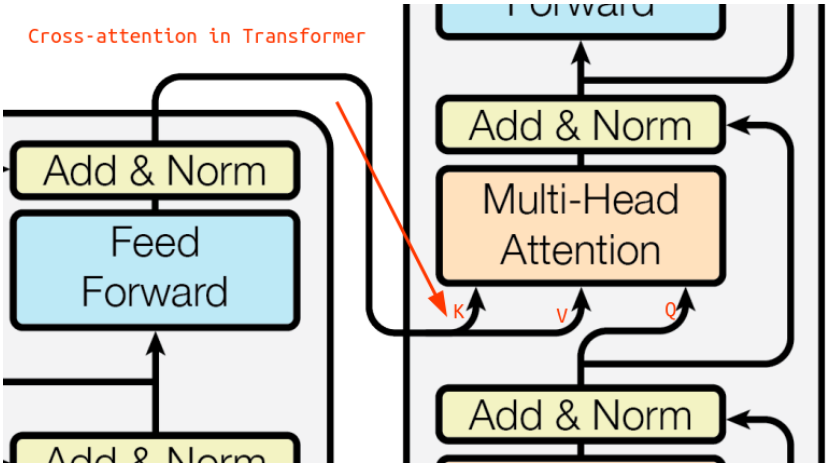
How does it do this? And what happened to all the work the encoder did?

Ex 1.


Je
Output Probabilities
Softmax
Linear
1. Cross attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
2. Masking
<start> Je suis content
Almost identical architectures
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)
I am happy
(Encoder inputs)
<start> Je suis content
(Decoder inputs)



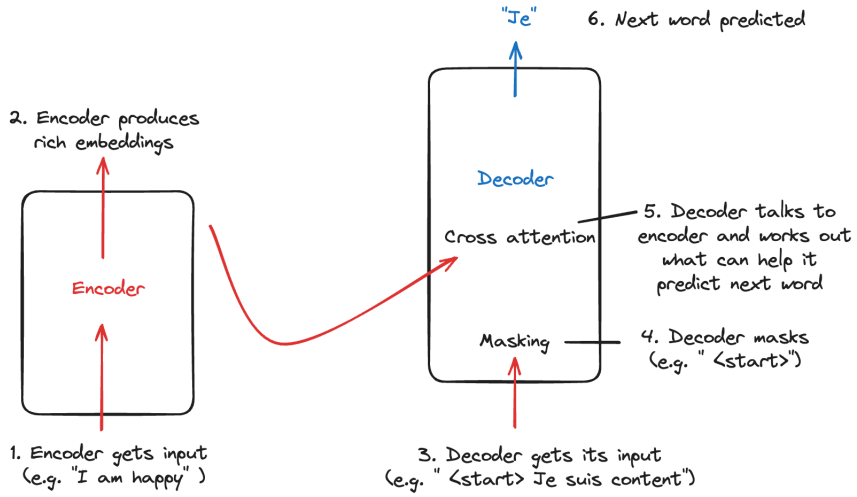
Cross attention



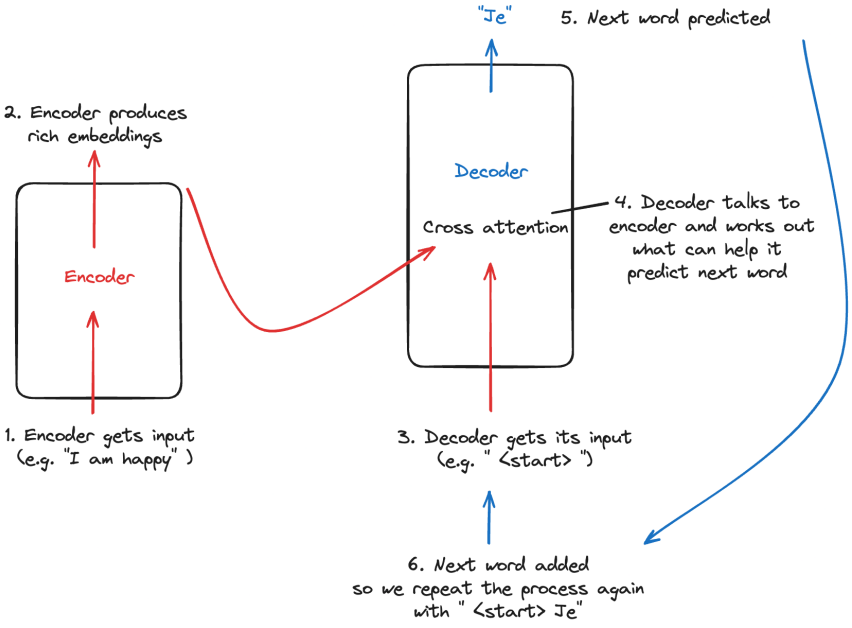
Attention between encoder and decoder (a.k.a. cross-attention):

- **Self-attention** operates within a single sequence and captures the relationships between tokens within that sequence.
- **Cross-attention** operates between two different sequences and captures the relationships between tokens from the source sequence and tokens from the target sequence, allowing the model to generate relevant output based on the information in the source sequence.

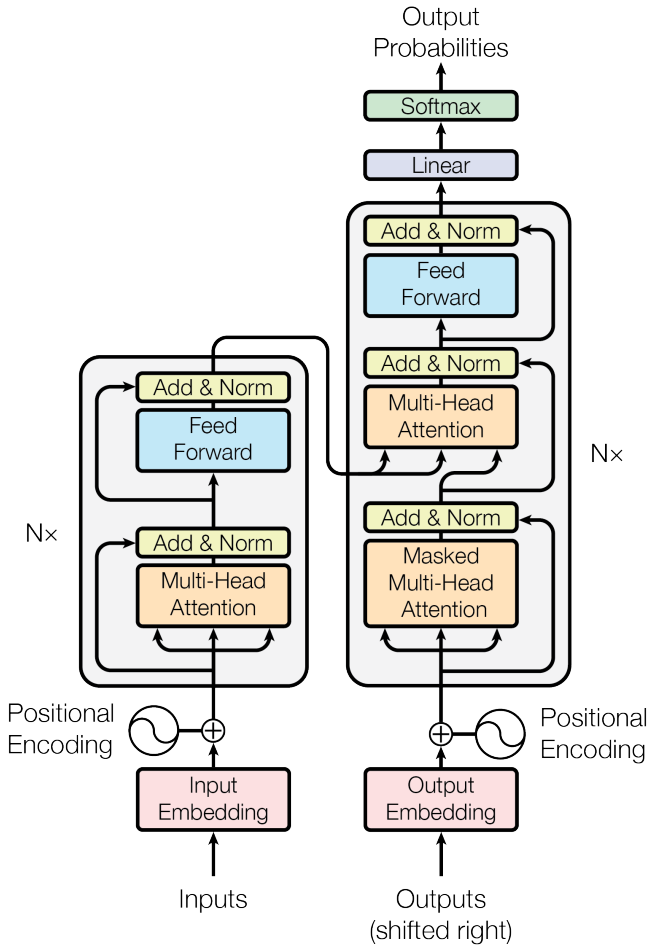
Let's recap the training process at a high level



So what does inference look like?



That's all there is to it!



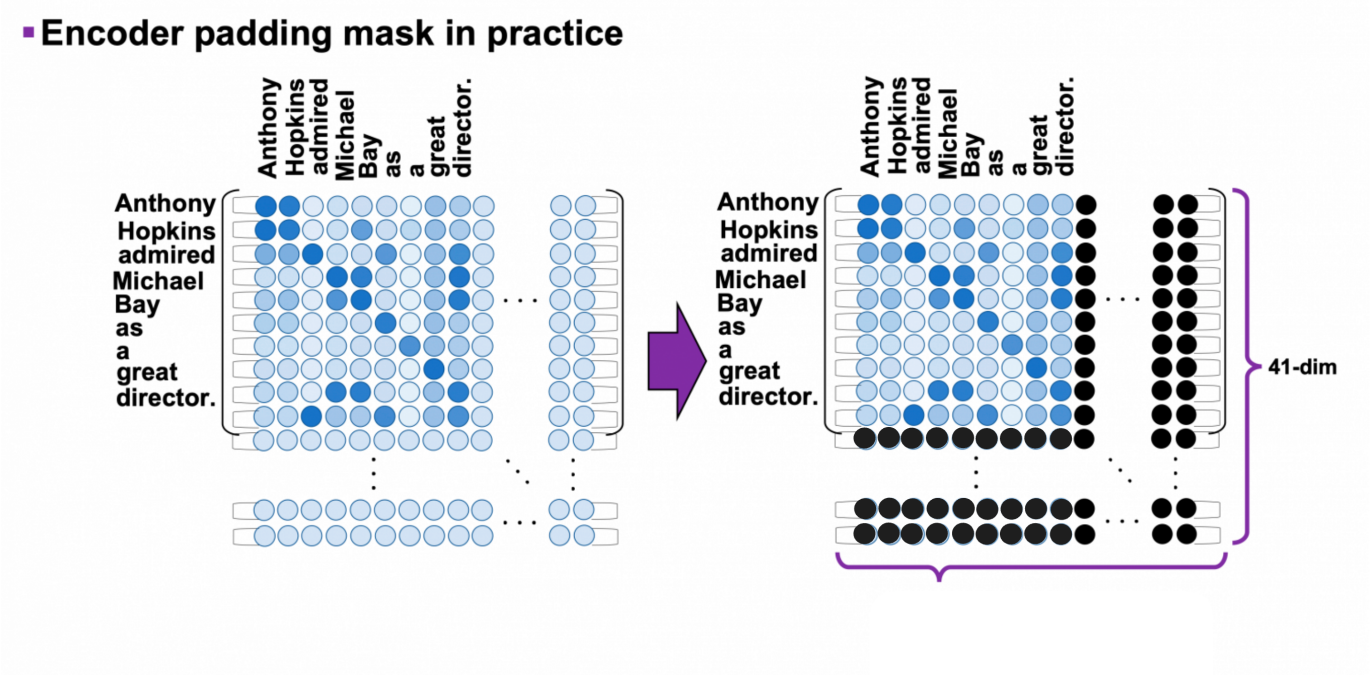
4️⃣ Digging Deeper

What *haven't* we covered yet:

- Self-padding mask
- Skip layers
- Subword tokenization
- Positional encoding

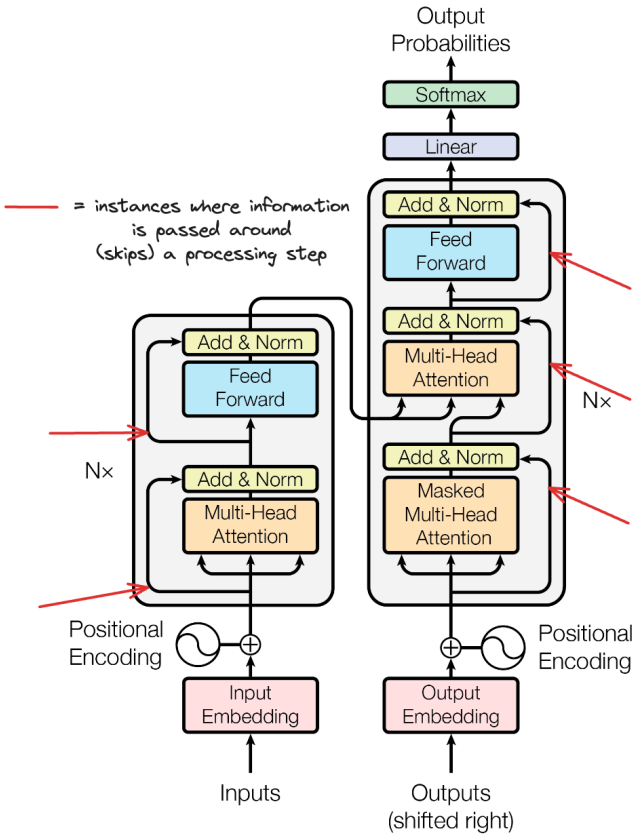
Self-padding mask

- Masks out padding tokens during self-attention, preventing the model from attending to them and ensuring focus only on relevant parts of variable-length input sequences.



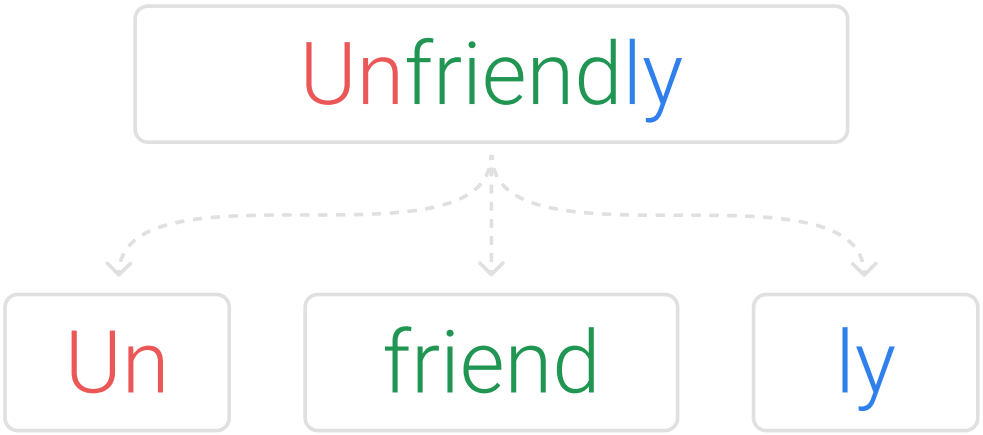
Skip layers

- Information from earlier layers is propagated directly to later layers, aiding in the retention of valuable information during training.
- Faster convergence and improved performance ✅



Subword tokenization

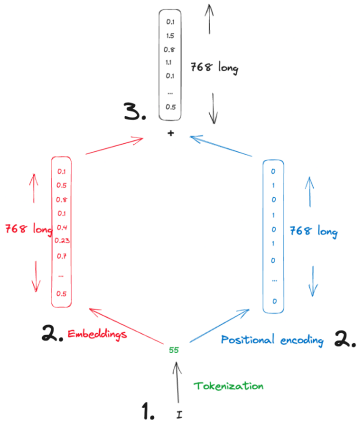
- Enables the model to handle OOV words and capture morphological variations (e.g. "walks" and "walking" aren't viewed as entirely separate tokens but rather composites of shared/ different tokens)
- Makes the language representation more compact, efficient, and generalizable across different word forms



Positional encoding

- Gives the model an idea of each word's position in the sentence
- Can be hard-coded or learned
- Attention is All You Need has a clever hard-coding using varying frequencies to convey positional information

See this video (https://www.youtube.com/watch?v=dichIcUZfOw&ab_channel=HeduAI) for a detailed walkthrough!



5 HuggingFace

Pretty tricky under the hood 🤢 but the `transformers` makes it all super easy in code:

In [ ]:
```
! pip install transformers
```

- Pre-trained Models: Provides a collection of pre-trained state-of-the-art models like BERT, GPT-2, T5, and many more for various NLP tasks.
- Simple: Offers a unified and simple way to find a model, fine-tune it, and deploy it.
- Multilingual: Supports models in multiple languages (although lots are primarily written in `torch` but there are plenty of tricks that enable you to work with PyTorch models)

In [ ]:
```
from transformers import pipeline
pipe = pipeline("translation", model="Helsinki-NLP/opus-mt-en-fr")
```

In [ ]:
```
result = pipe("I am a student and I am studying in London")
result[0]['translation_text']
```

Out[ ]:
```
"Je suis étudiante et j'étudie à Londres."
```

Under the hood of a pipeline

- Models use their own trained **tokenizers** which we load up with the `.from_pretrained() method
- Then we just pass in the creator of the model and model name
- If we want to pass in Tensorflow tensors (which we will), just put TF before our model and pass `from_pt = True`.

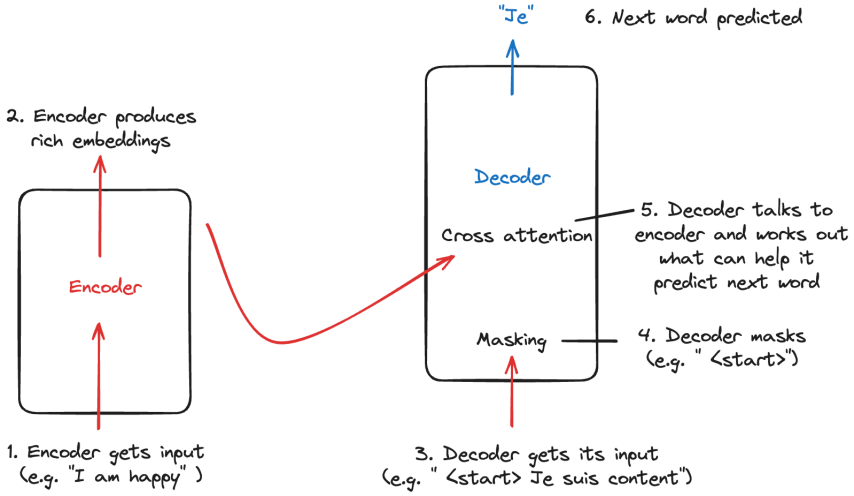
In [ ]:
```
from transformers import AutoTokenizer, TFAutoModelForSeq2SeqLM

tokenizer = AutoTokenizer.from_pretrained("Helsinki-NLP/opus-mt-en-fr")

tokens = tokenizer.encode("This is easy!", return_tensors = "tf")

print(tokens)
```

```
tf.Tensor([[ 160   32 3120  145    0]], shape=(1, 5), dtype=int32)
```

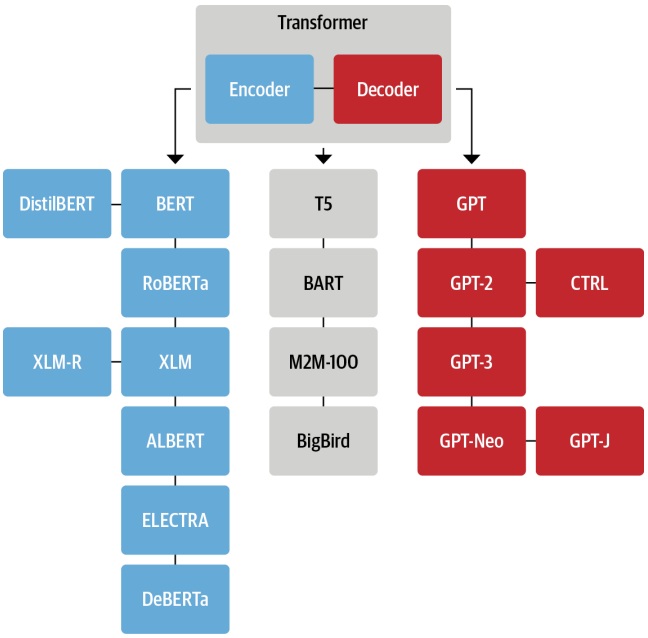
In [ ]:
```
model = TFAutoModelForSeq2SeqLM.from_pretrained("Helsinki-NLP/opus-mt-en-fr", from_pt = True)

output_tokens = model.generate(tokens)

print(output_tokens)

print(tokenizer.decode(output[0]))
```

6 Going further

What are the two parts of our diagram doing?



The Transformer family tree:



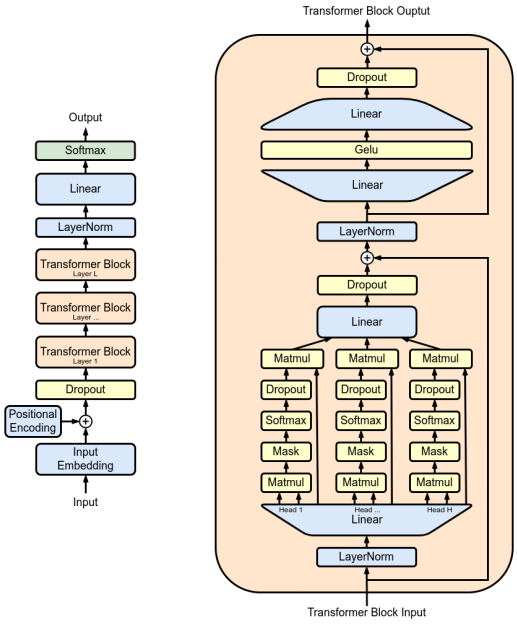
Encoder-only (e.g. BERT): converts an input sequence to a numerical representations. Uses words to the left and right of each word (hence "bidirectional") and is great for things like classification.

Decoder-only (e.g. GPT): takes an input sequence and iteratively predicts the most likely next word (can also be used in a similar manner to encoder-decoder if trained correctly)

Encoder-decoder (e.g. T5 or original "Attention is all you need" paper model): maps one sequence to another

So what makes ChatGPT so great?

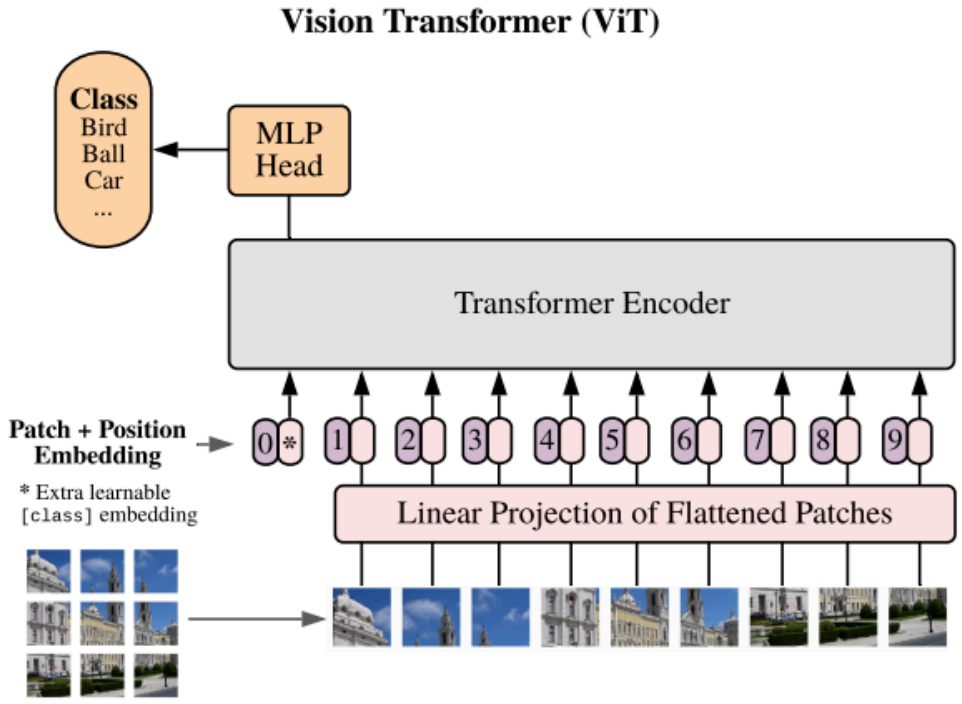
An example of the GPT-2 architecture:



We don't have direct access to ChatGPT's architecure but:

- Huge amounts of data and training
- 175B weights
- RLHF

Vision Transformers & Multi-modal models:



7️⃣ The final question: When to use DL or ML?

If you have standard data

As DL is nothing more than another ML model, you can use either a standard neural network or any other technique: random forest, svm, linear regression, logistic regression, ...

In practice, scikit-learn offers a lot of ML algorithms that you can prototype with and compare very quickly. You should probably start with that. Once done, if you are not satisfied with the result, you can try DL models.

NB: In general, DL will work better if you have a lot of data

If you don't have standard data

In cases where you work with:

- images (and videos)
- temporal data
- text

It is in general better to start with DL architecture. Why?

Because DL has the appropriate operators / functions / tools to work with such data.

For vanilla ML techniques, you would need either to preprocess such data or to tweak your algorithms. In either case, you can't be certain that the operations you do are relevant. It's better to let DL algorithms do their own magic.

Acknowledgements and further reading on Transformers 📚:

The Deep Learning Bible (https://wikidocs.net/book/8027): totally free, continually updated and super clea with accompanying GitHub repos

Intro to Transformers (https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html#transformer_intro): A high level overview of many of the concepts covered here

Jay Alamar's Transformer Illustrated (https://jalammar.github.io/illustrated-transformer/): Fantastic step-by-step visualizations and explanations for multiple Transformer-based models

HuggingFace + Transformers Textbook (https://www.oreilly.com/library/view/natural-language-processing/9781098136789/): If you want to do anything with HuggingFace, this is a fantastic primer.

Accompanying Open Source GH Repo for the above textbook (https://github.com/nlp-with-transformers/notebooks): Totally free access to the examples from the above book!

StatsQuest Video on Self-Attention (https://www.youtube.com/watch?v=zxQyTK8quyY&vl=en): A great step by step breakdown of the process