
CUHK
ECON
AI×ECON
Summer School @ CUHK
AUGUST 7-8 2024

Transformer

Almost as if it's intelligent.

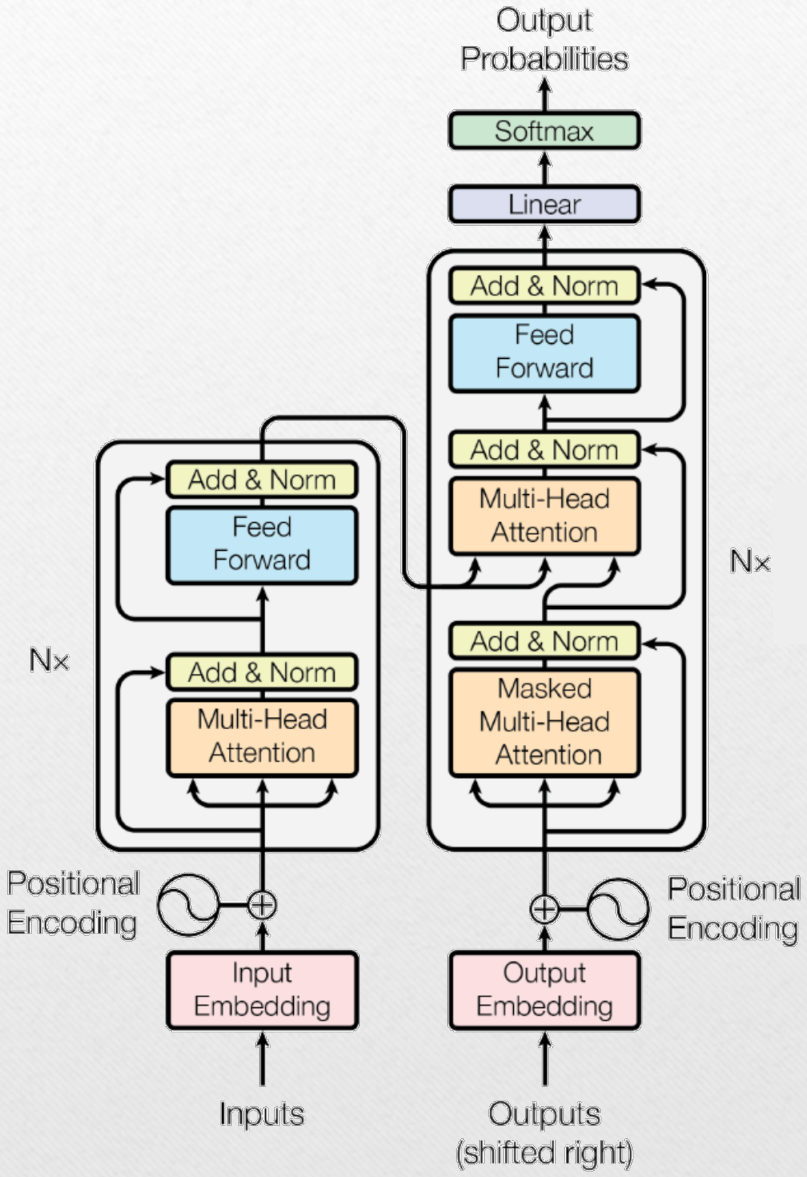
Conversational AI

Image Generation

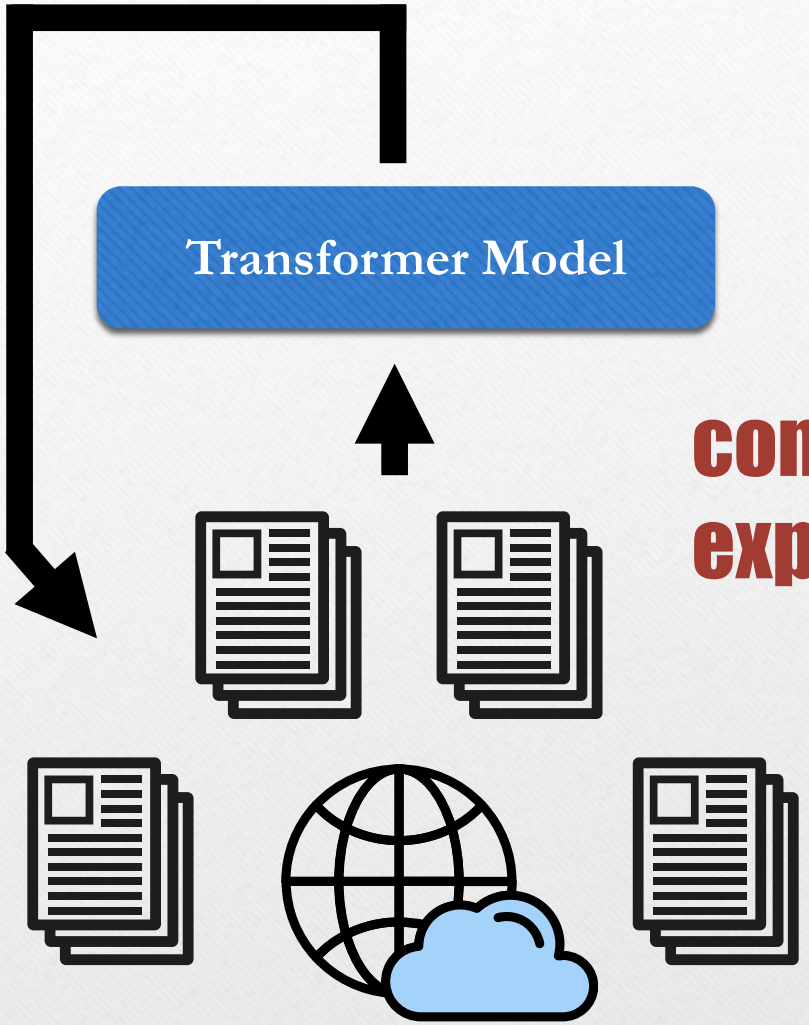
A composite of three DALL-E 3 AI art generations: an oil painting of Hercules fighting a shark, a photo of the queen of the universe, and a marketing photo of "Marshmallow Menace" cereal. *Benj Edwards@Ars Technica*

Transformer

- Transformer-based models such as GPT are the current state of the art in field of natural language processing.
- Original paper: All You Need is Attention (NeurIPS 2017)
- Transformer is the basis of some of the most advanced AI currently in existence, including the Starcraft-playing AlphaStar, text-generating GPT-4 and image-generating Stable Diffusion.

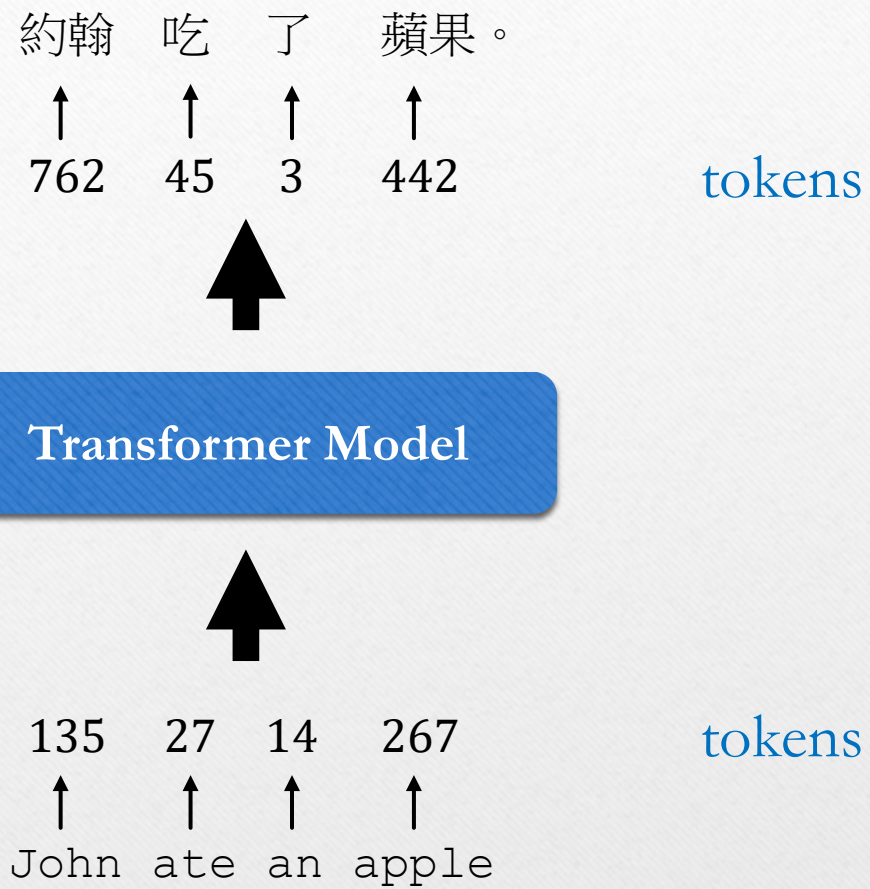


Unsupervised:
No manual labelling



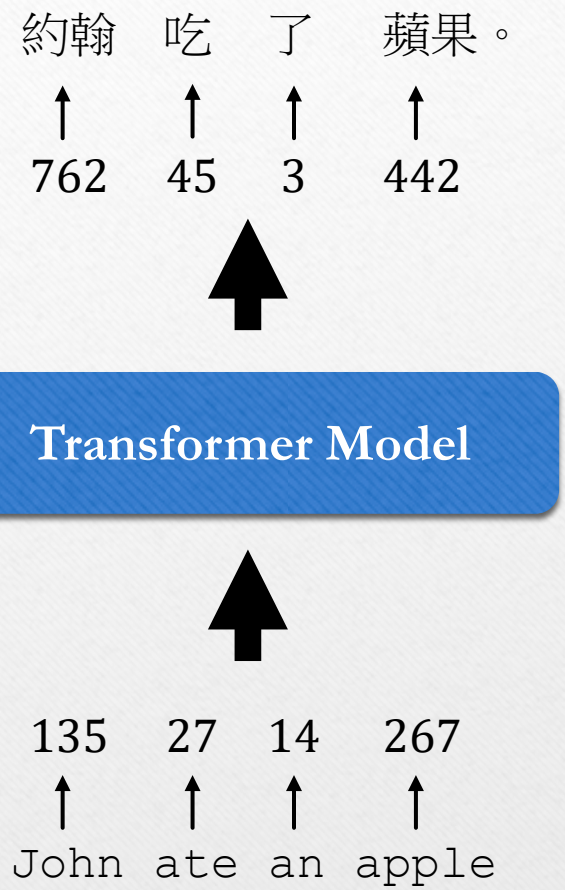
computationally expensive!

Stage 1: Pre-training



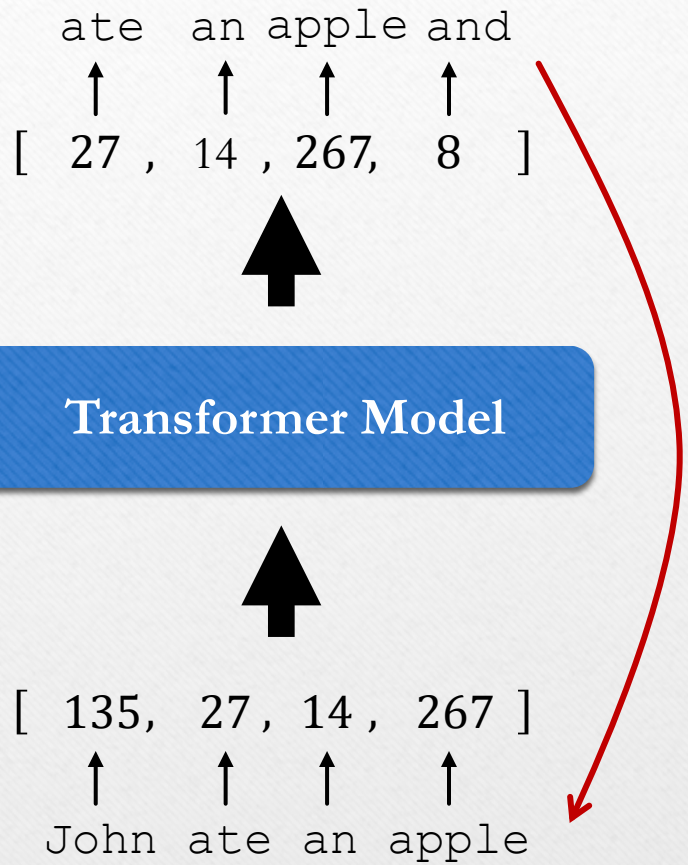
Input and Output

Transformer is a **sequence-to-sequence** model, meaning that the input and output are both vectors.



Output

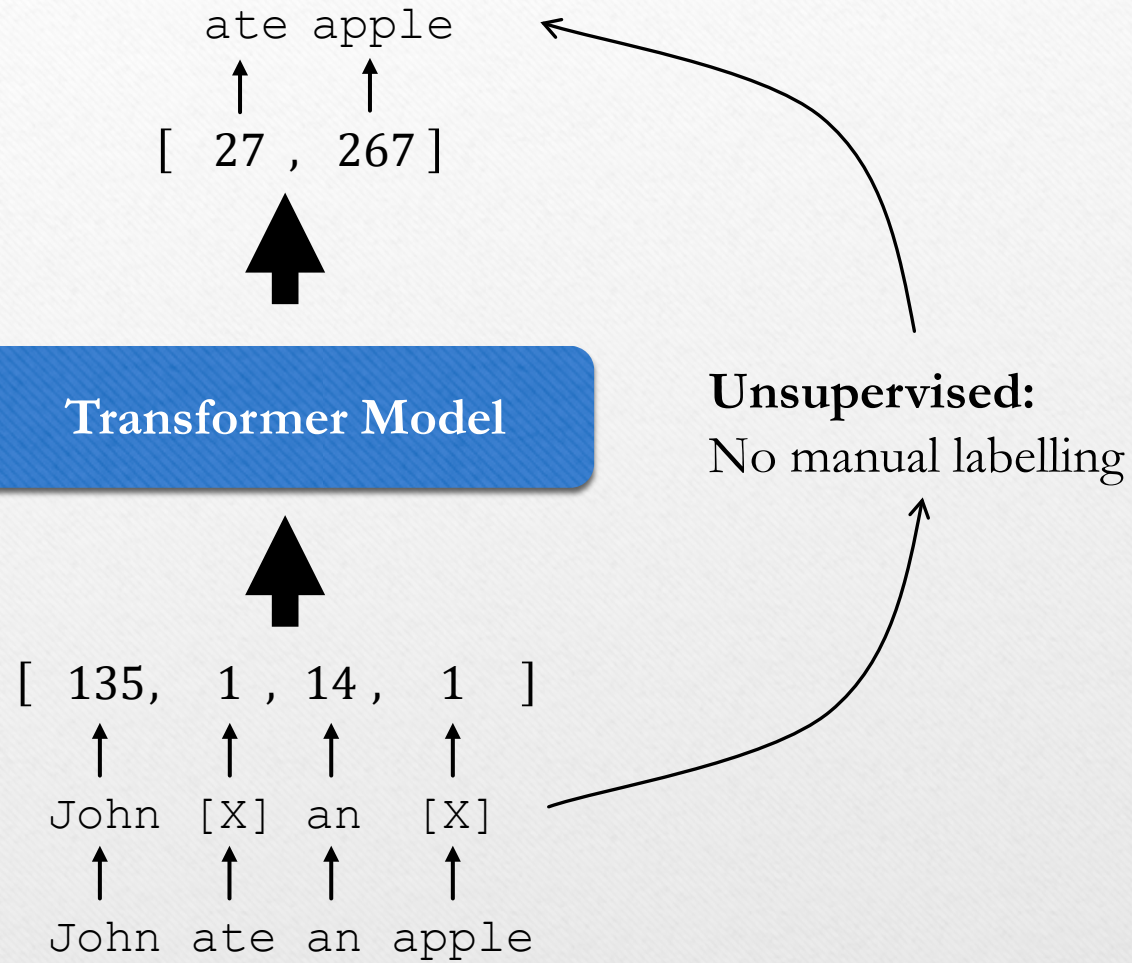
Translation: the model predicts words in another language. This was what the original paper did.



Unsupervised:
No manual labelling

Output

The next word: the model predicts the next word after the input. This is how GPT models are pre-trained.



Fill-in-the-blanks/denoising: the model predicts missing words in a passage from which the input was taken. This is how T5 and BERT models are pre-trained.

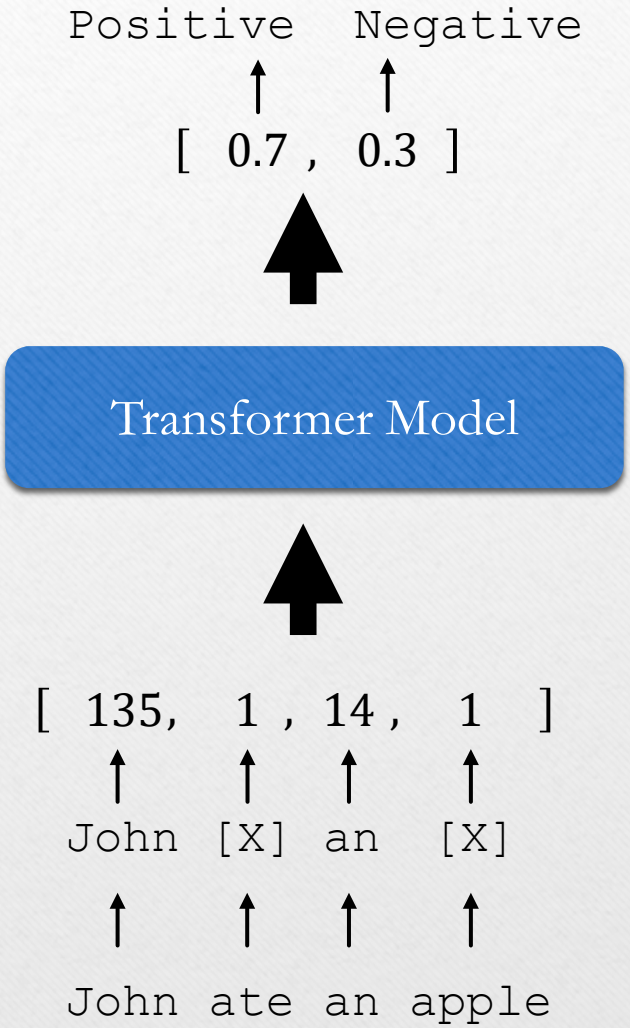
Stage 2: Fine-tuning

The model is then **fine-tuned** to a specific task. This stage can be done with relatively little data and computational power.

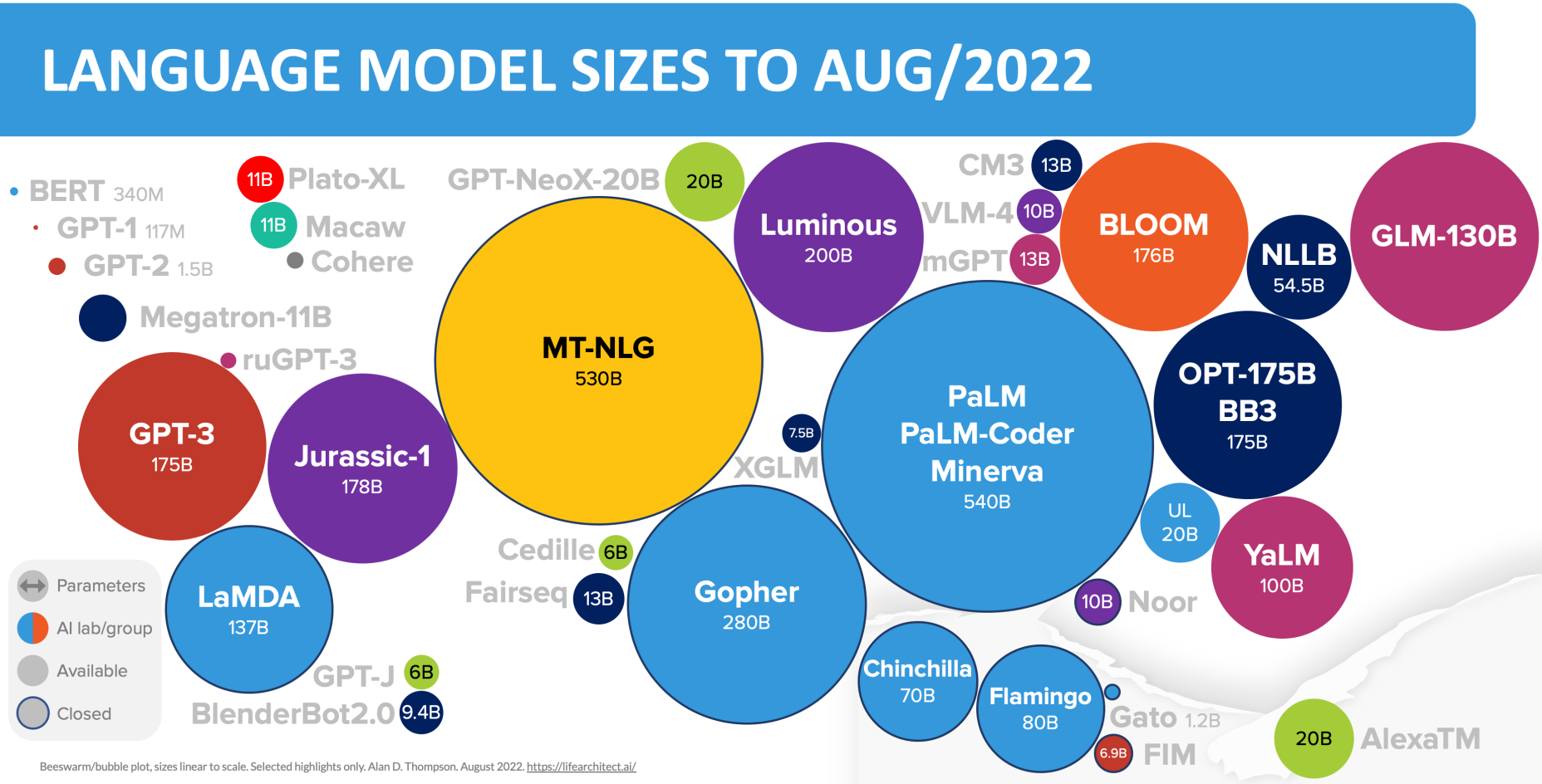
With fine-tuning, the model can adapt to tasks that it was not pre-trained for:

- Instruction following
- Sentiment analysis
- Text classification

Fine-tuning typically only takes a few epochs to achieve good results, because the model already understand language from pre-training.

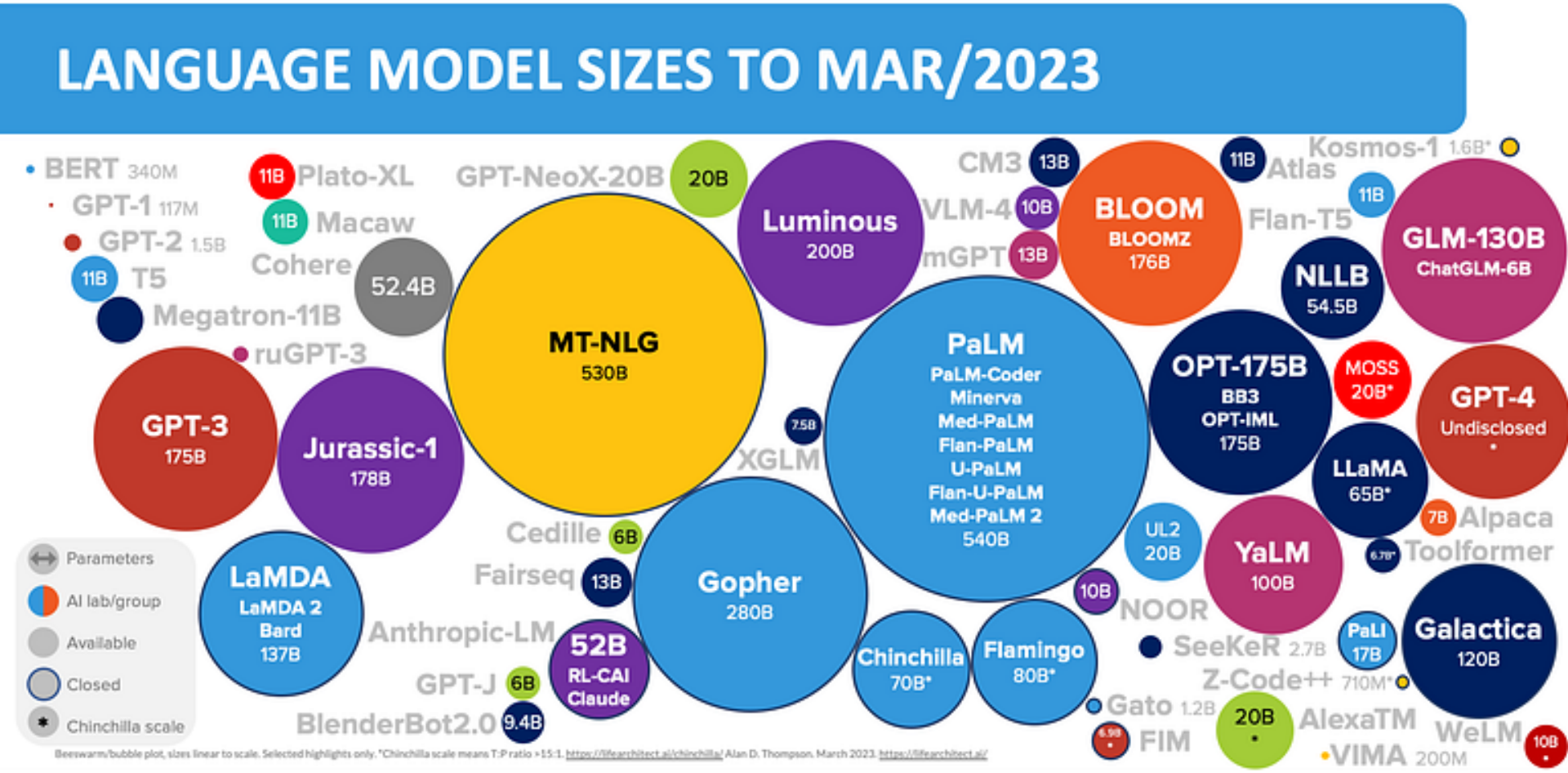


2022 Big



LifeArchitect.ai/models

2023 Big



LifeArchitect.ai/models

Model Categorization

- Transformer-based models, particularly LLMs, are usually categorized by their architecture and the number of parameters they have.
  - E.g. LLaMA 7B means the model uses the LLaMA architecture and has 7 billion parameters.
- Models with more parameters are more powerful, but also more expensive to run.
  - Difference is small for tasks such as text classification, but huge for math complex reasoning.

Meta Llama 3 Instruct model performance

	Meta Llama 3 8B	Gemma 7B - It Measured	Mistral 7B Instruct Measured
MMLU 5-shot	**68.4**	53.3	58.4
GPQA 0-shot	**34.2**	21.4	26.3
HumanEval 0-shot	**62.2**	30.5	36.6
GSM-8K 8-shot, CoT	**79.6**	30.6	39.9
MATH 4-shot, CoT	**30.0**	12.2	11.0

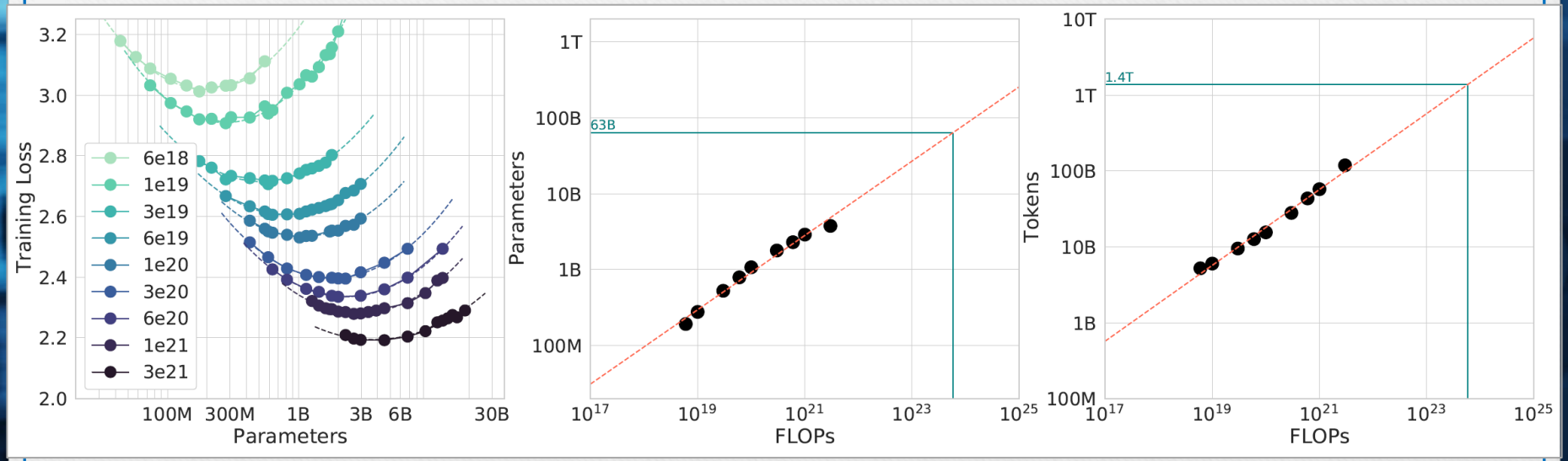
	Meta Llama 3 70B	Gemini Pro 1.5 Published	Claude 3 Sonnet Published
MMLU 5-shot	**82.0**	81.9	79.0
GPQA 0-shot	39.5	**41.5** CoT	38.5 CoT
HumanEval 0-shot	**81.7**	71.9	73.0
GSM-8K 8-shot, CoT	**93.0**	91.7 11-shot	92.3 0-shot
MATH 4-shot, CoT	50.4	**58.5** Minerva prompt	40.5

Source: Meta

Why Bigger Model?

Large language models are likely undertrained. In other words, the amount of data and computational power spent on training is insufficient relative to model size.

…but we have basically run out of data, so training larger models for longer it is.



Hoffman et al (2022) a.k.a. "The Chinchilla Paper"

How to Use Transformer-based models?

- Training large Transformer-based models from scratch is very expensive, so in practice you will need to use pretrained models.
- **Closed-source models:** OpenAI and Anthropic provide an API to their models, with different payment tiers. These models are most suitable if your task can be framed as questions. Inaccessible in Hong Kong without VPN.
- **Open-weight models:** Pretrained models available for download. The easiest service to use are Hugging Face and GPT4All, with the most popular models being Meta's LLaMa, MosaicML's MPT, TII's Falcon and BERT. BERT is particularly suitable for fine-tuning for non-conversational task such as text classification, due to its relatively small size and encoder-only architecture.

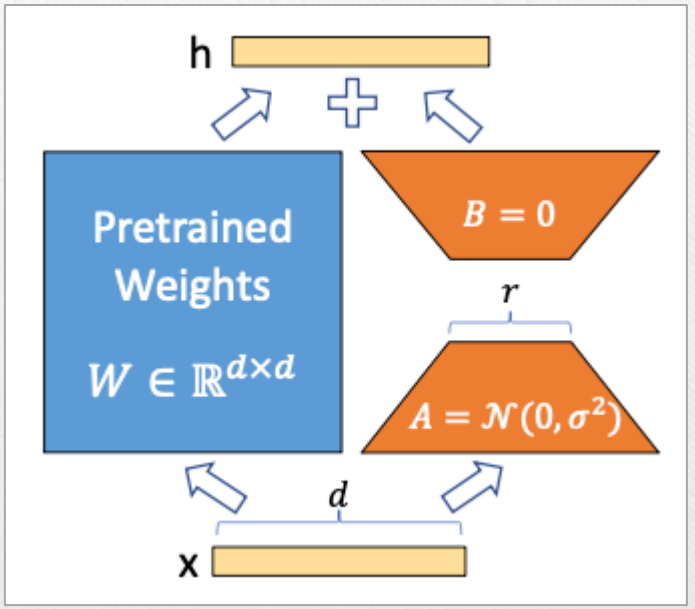
Memory Requirement for Open-Weight Models

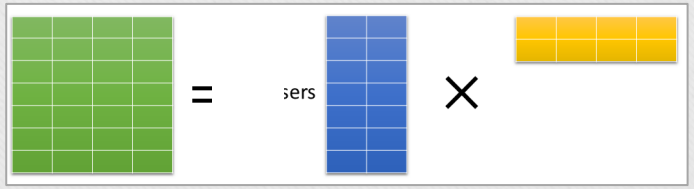
- Models are generally saved in 16-bit numeric format, which means each parameter takes 2 bytes of memory.
  - LLaMA 7B would take approximately 14GB of memory.
- If the model requires more memory than your system—particularly your GPU—can provide, you can lower the numeric precision. This is called **quantization** and will result in lower model performance.
  - LLaMA 7B takes 7GB of memory under 8-bit and 3.5GB under 4-bit.
- Note that the above is just the memory requirement of loading the model. Training the model could require multiples of that, due to the need to save the gradient for every weight and the activation for every neuron for back propagation.

LoRA: Low-Rank Adapter

LoRA deals with the memory problem by only training a low-rank matrix that is then added to the original LLM. The original LLM can be loaded in lower precision and without tracking gradients.

Pre-trained LoRA addons to popular open-weight models are commonly available.





How Does Transformer Work?

Not necessary if you just
want to use pre-trained models!

Why Transformer?

```
John ate an apple. It was delicious.
John ate an apple. He likes it.
```

- Consider the following above passages.

Self-Attention

John ate an apple. **It** was delicious.

John ate an apple. **He** likes it.

- The most important idea of a Transformer is **self-attention**.
- When the model reads a particular word, the output depends on *all* words, weighed by how important they are given the word in concern.
- We will now go through what a **self-attention layer** does.

```
John ate an apple. It was delicious.
```

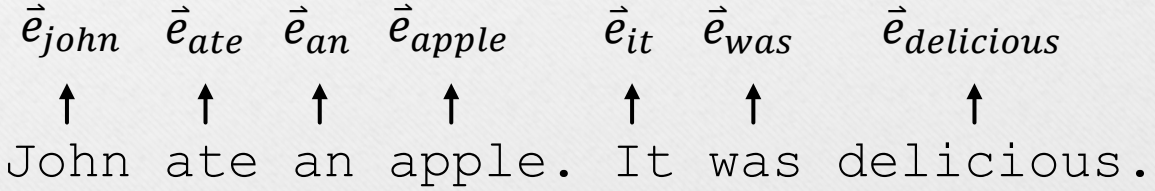
This is one input sample.

```
[start] John ate an apple. It was delicious. [end]
```

During preprocessing, special characters are added to the input and target text phrases. These characters denote the start, the end and other contextual information and are treated just like another any other word.

```
[start] John ate an apple. It was delicious. [end]
```

For space reasons I will omit them in latter slides.



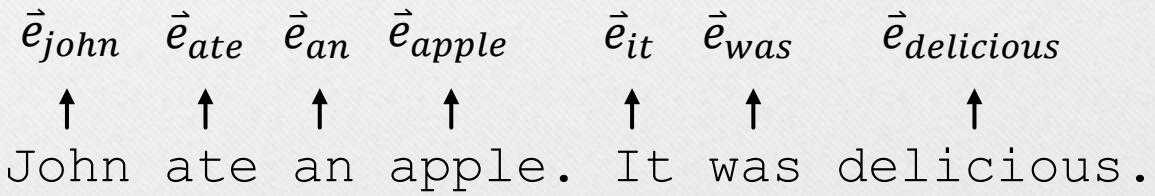
As in all NLP neural network models, we start with assigning each word an embedding—a vector learnt through training to represent the nature of each word.

e.g. Noun Verb Male

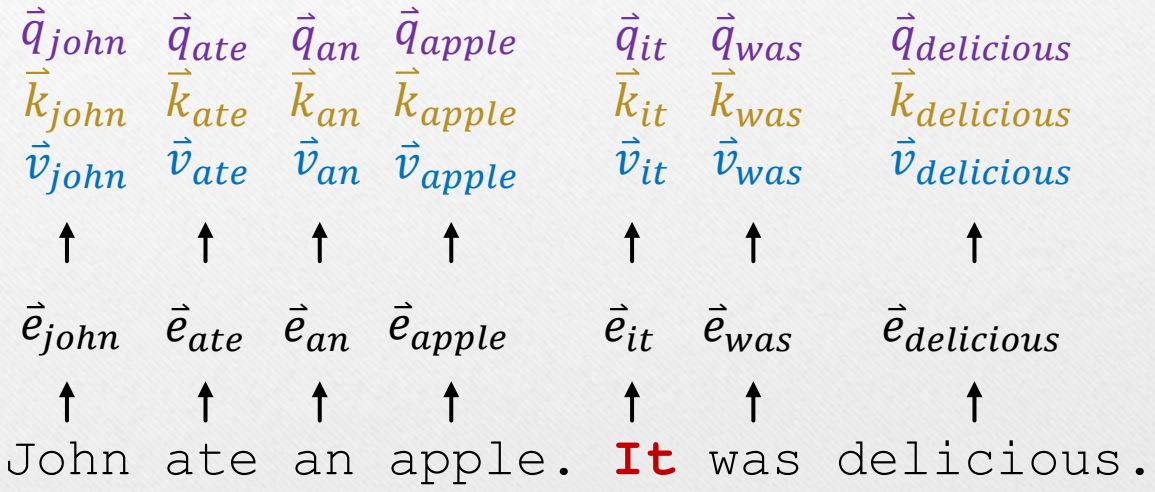
$$\vec{e}_{john} = [1 \quad 0 \quad 1 \quad ...]$$

$$\vec{e}_{ate} = [0 \quad 1 \quad 0 \quad ...]$$

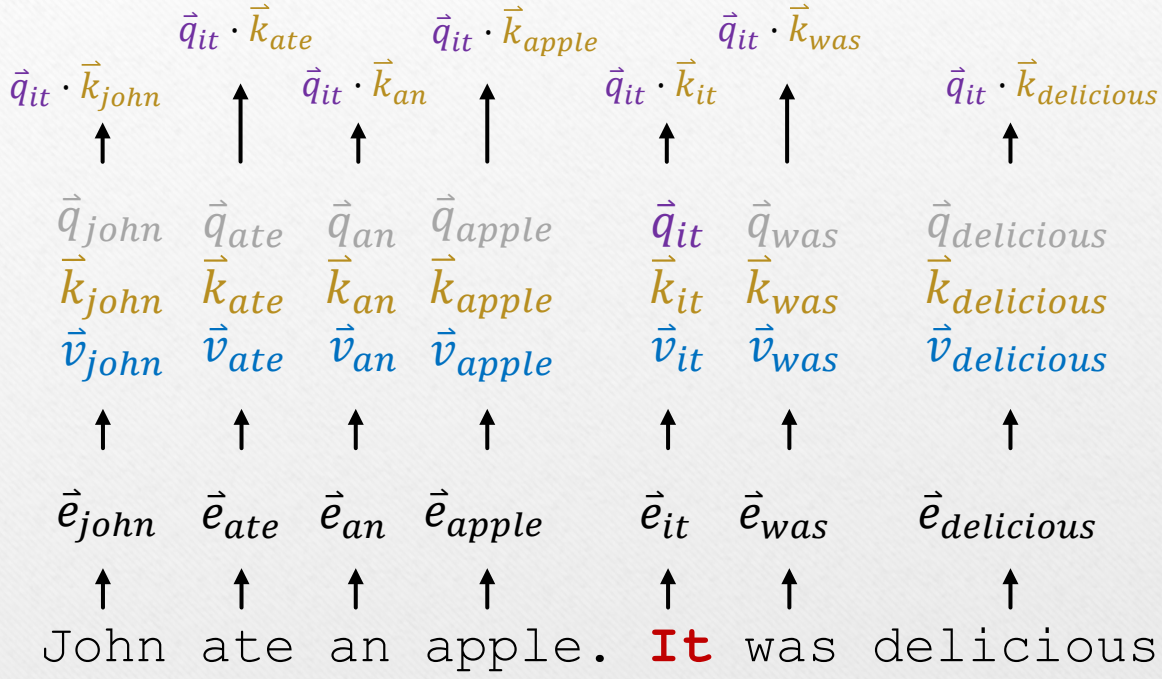
$$\vec{e}_{he} = [0 \quad 0 \quad 1 \quad ...]$$



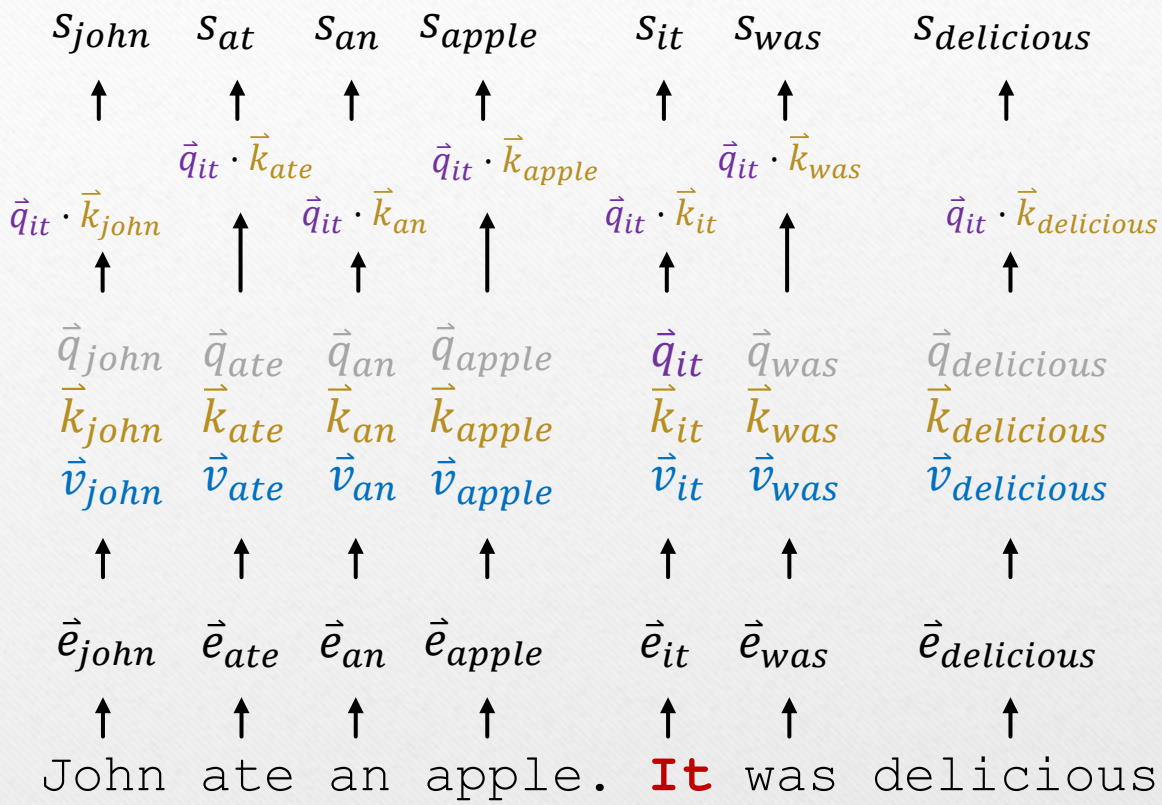
As in all NLP neural network models, we start with assigning each word an embedding—a vector learnt through training to represent the nature of each word.



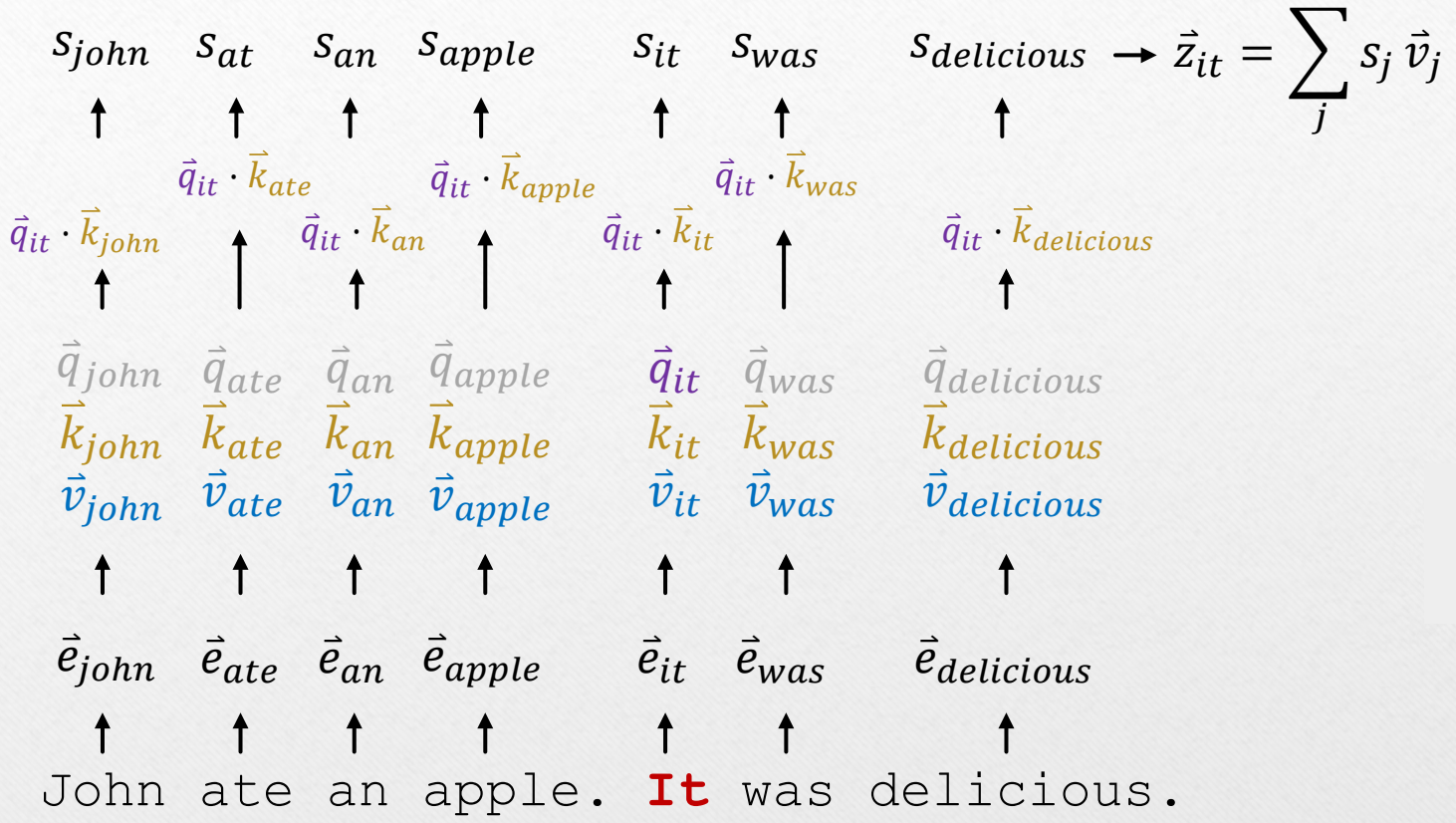
The attention mechanism requires three vectors as inputs: **query**, **key** and **value**. In the simplest case, they are simply the output from the previous layer.



When we process a word i, we compute a **score** for *all* words by taking the dot product of i's query and each word's key.



A weight is assigned to each word by feeding the score to a softmax function, i.e. multinomial logit.



The output z_i of the attention mechanism for word i is the sum of the multiple of each word's weight with its value.

Why Does Self-Attention Make Sense?

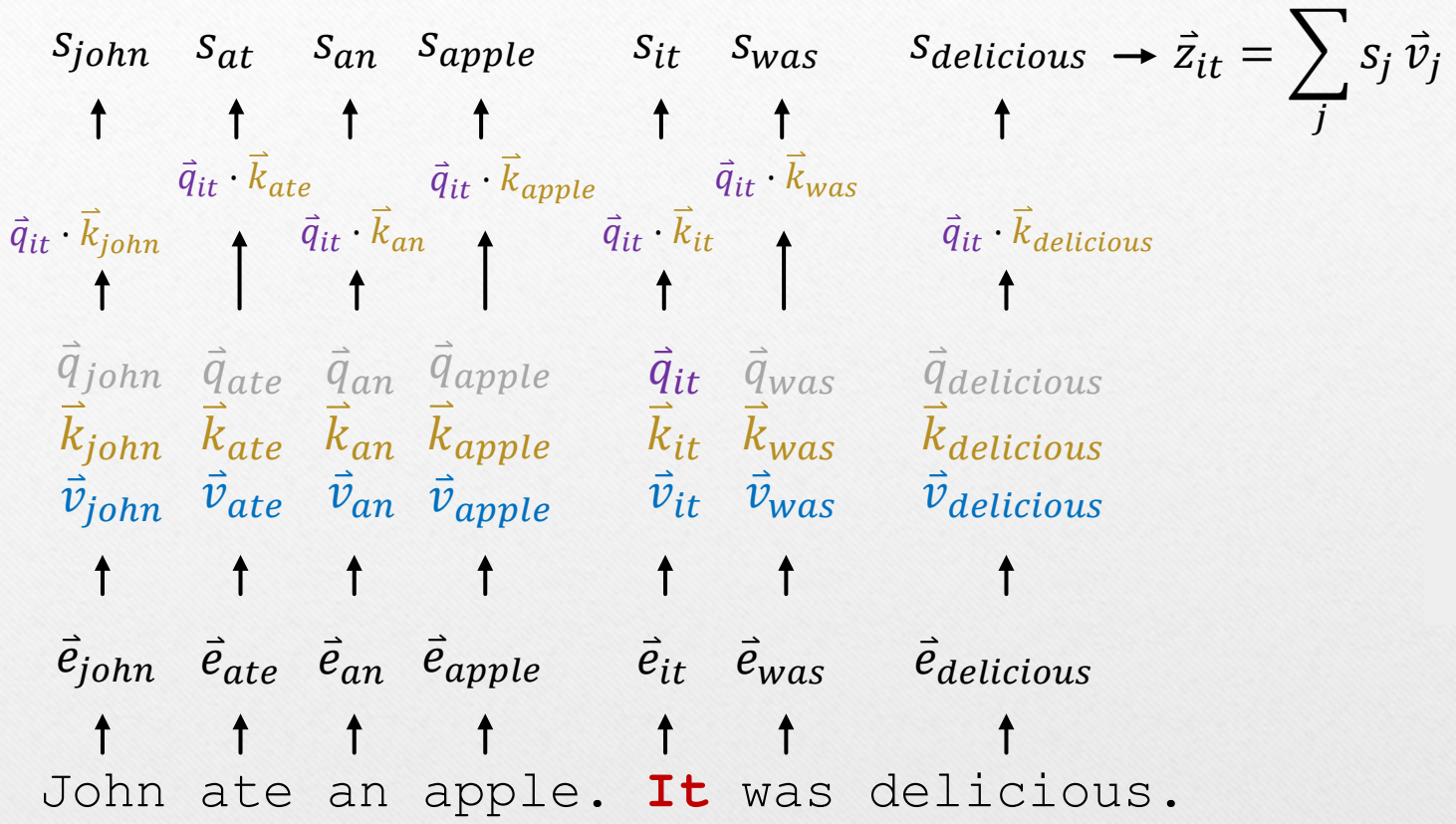
- Reconsider our example:

$$\vec{h}_{john} = [1 \quad 0 \quad 1]$$
$$\vec{h}_{ate} = [0 \quad 1 \quad 0]$$
$$\vec{h}_{he} = [0 \quad 0 \quad 1]$$

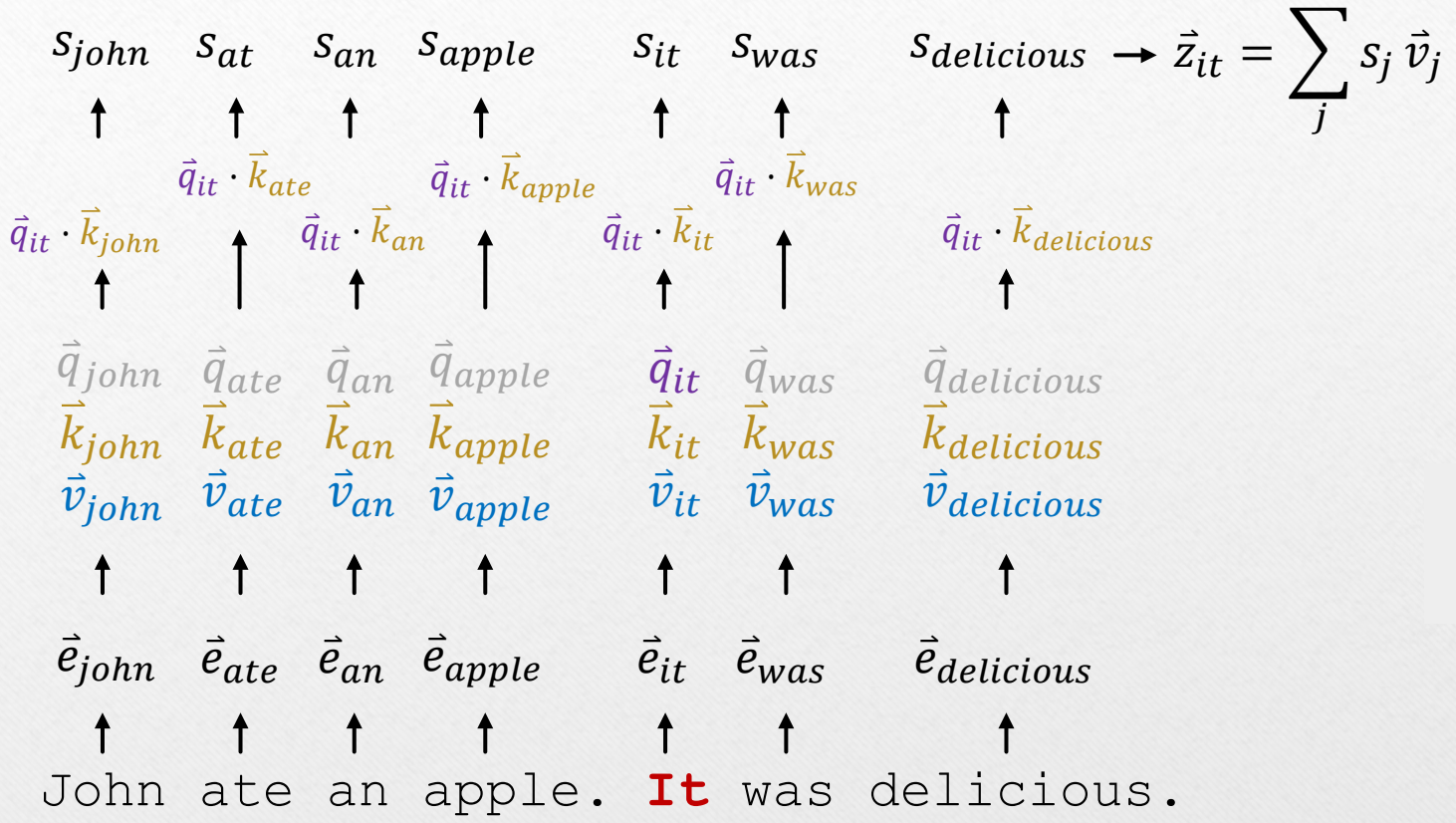
- When processing the word "he", the attention mechanism takes the dot product of the vector representations of "he" and each word:

$$s_{ate} = \vec{h}_{he} \cdot \vec{h}_{ate} = 0$$
$$s_{john} = \vec{h}_{he} \cdot \vec{h}_{john} = 1$$

- Words with similar vector representations get a higher weight. This allows the model to "pay attention" to relevant words.



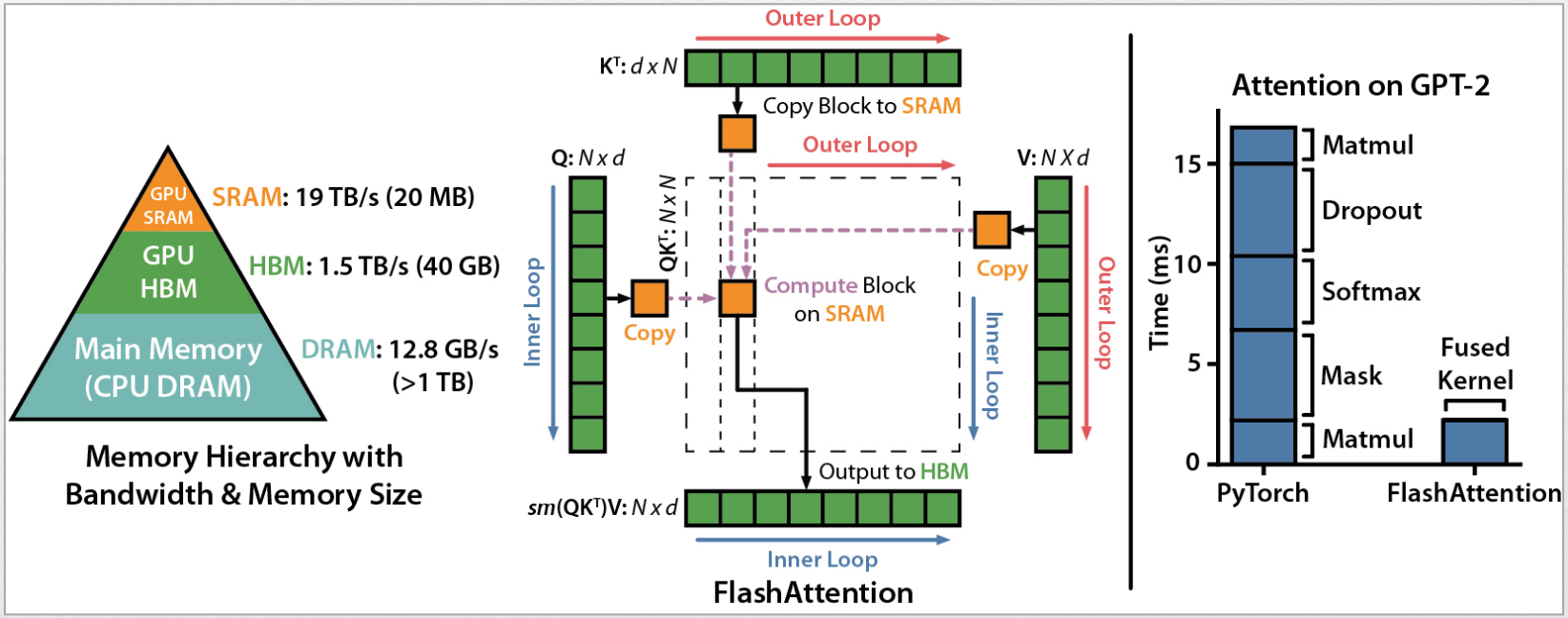
As self-attention is applied to all words, the mechanism is bidirectional—or more accurately, non-directional. It makes no distinction between words in different position.

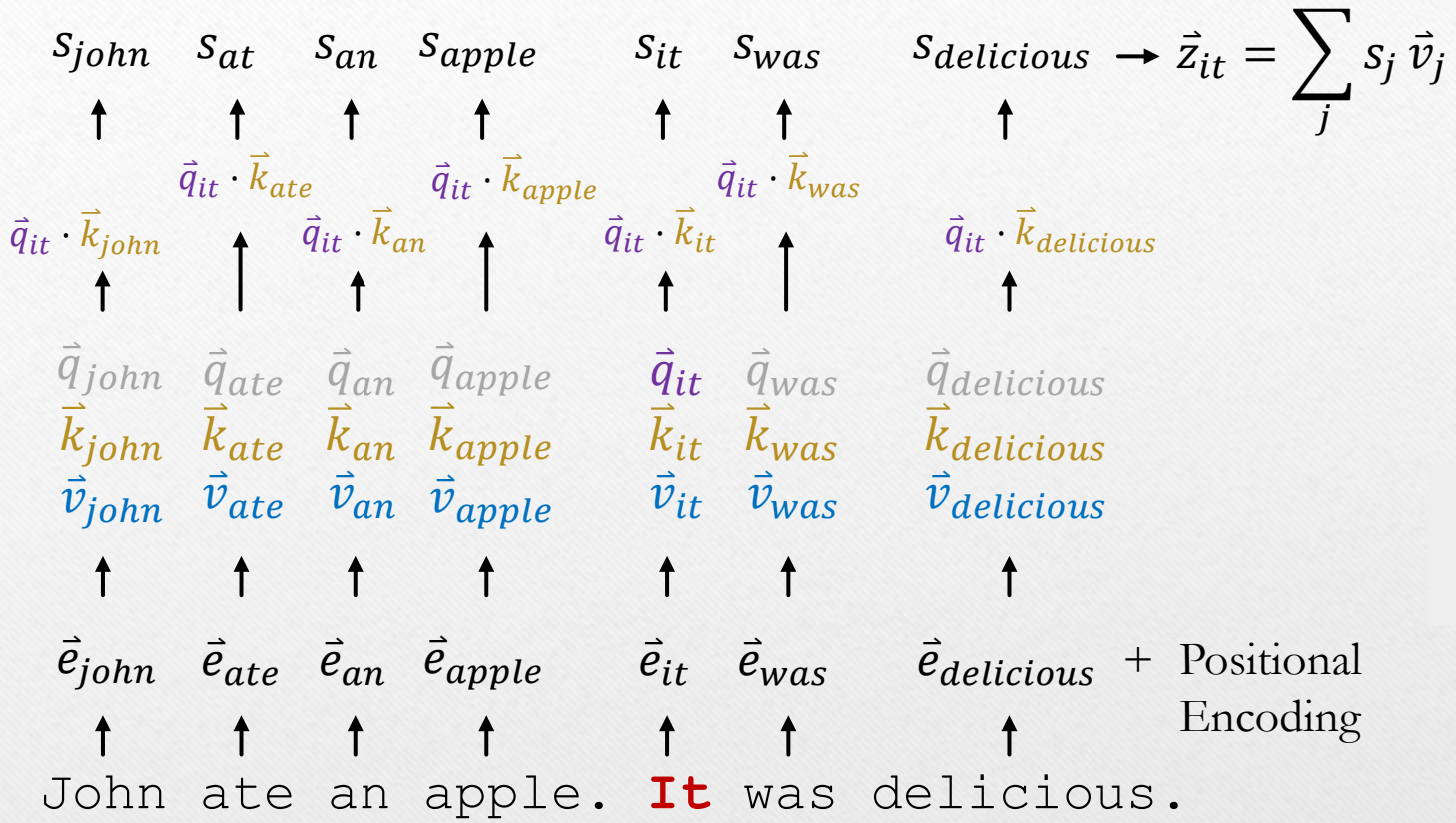


The fact that self-attention is applied to all words means that it takes n^2 operations and memory to process n words. This is costly and is the main reason why older transformer models cannot handle long sequence of words.

There are different ways to make self-attention less costly, including:

- Computing an approximate value instead of an exact value
- Limiting self-attention to nearby words
- Better algorithm to take better advantage of GPU architecture: **Flash Attention**

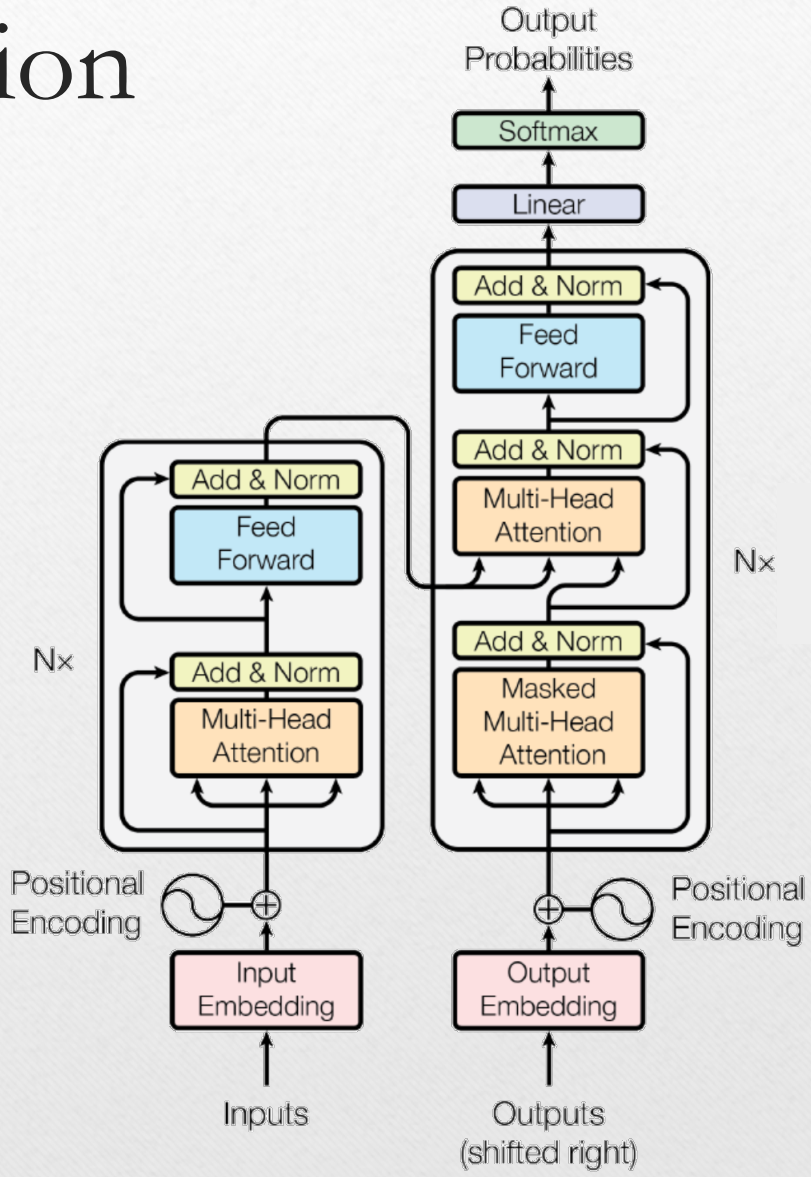


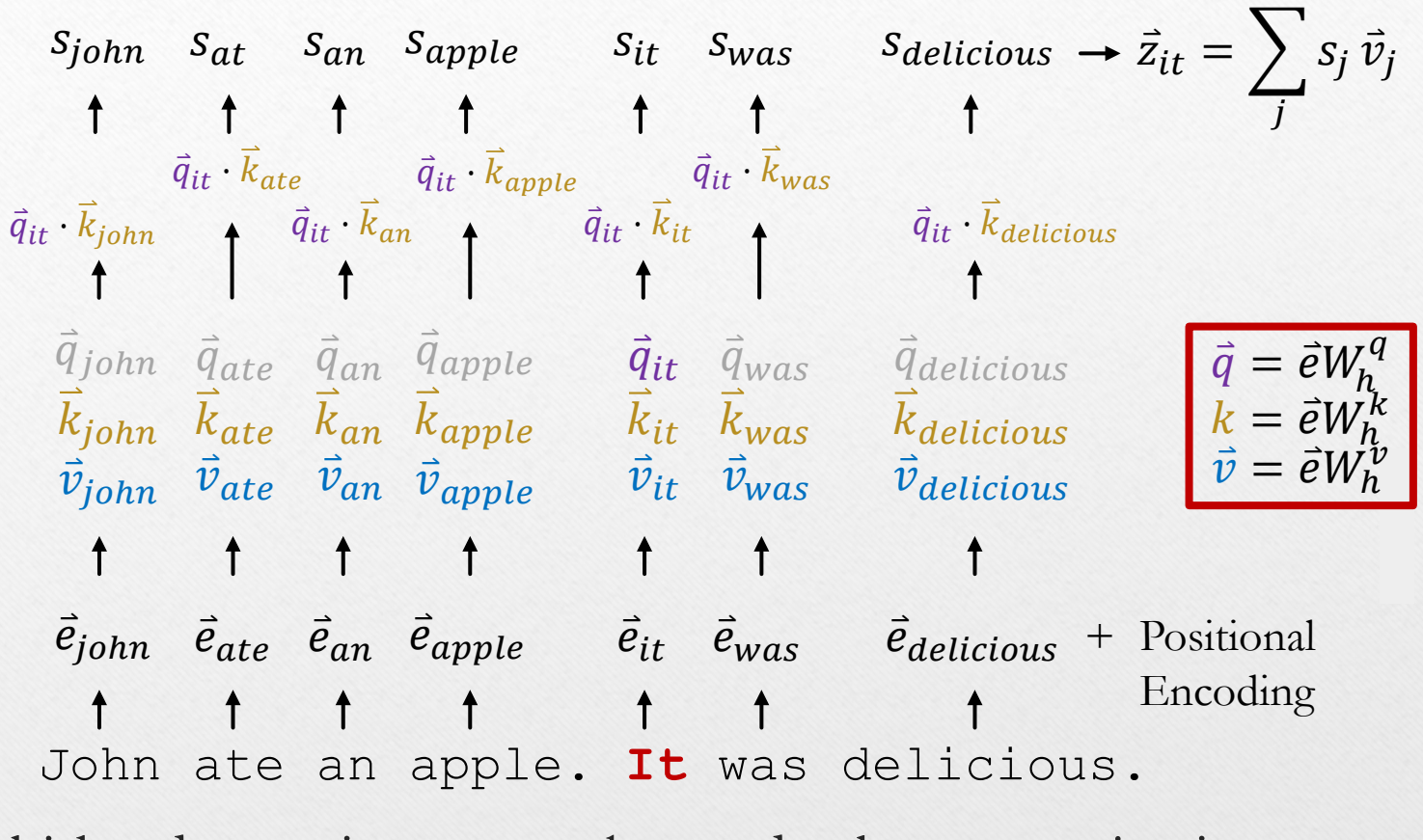


To provide the model with word positions, a **positional encoding** is added to the usual word embedding. The positional encoding can be fixed or learnt.

Multi-Head Attention

- What we just saw is **single-head (self-)attention**.
- **Multi-head attention** simply means we have multiple copies of the process, each with different weights.





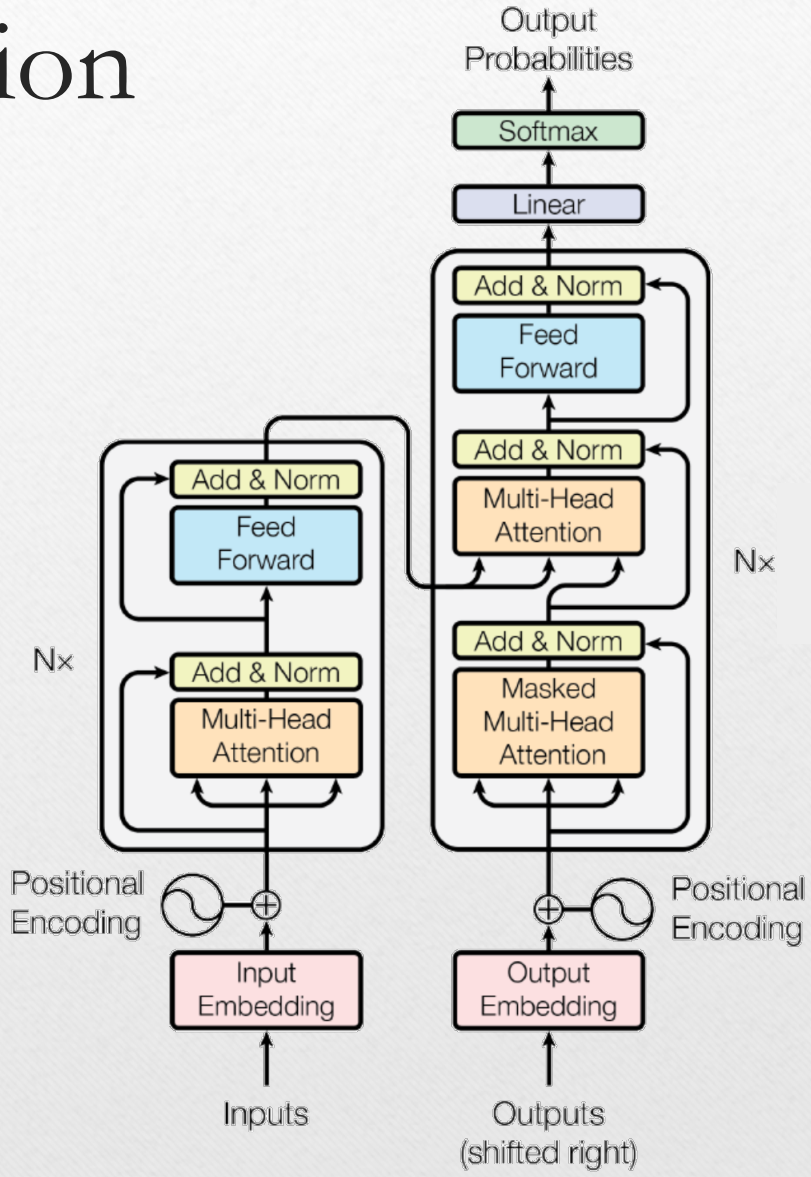
In multi-head attention, query, key and value are projections of the previous layer's output. You can view these as new embeddings capturing specific aspects of a word.

The weights W_h^q, W_h^k and W_h^v are learnt from training.

Multi-Head Attention

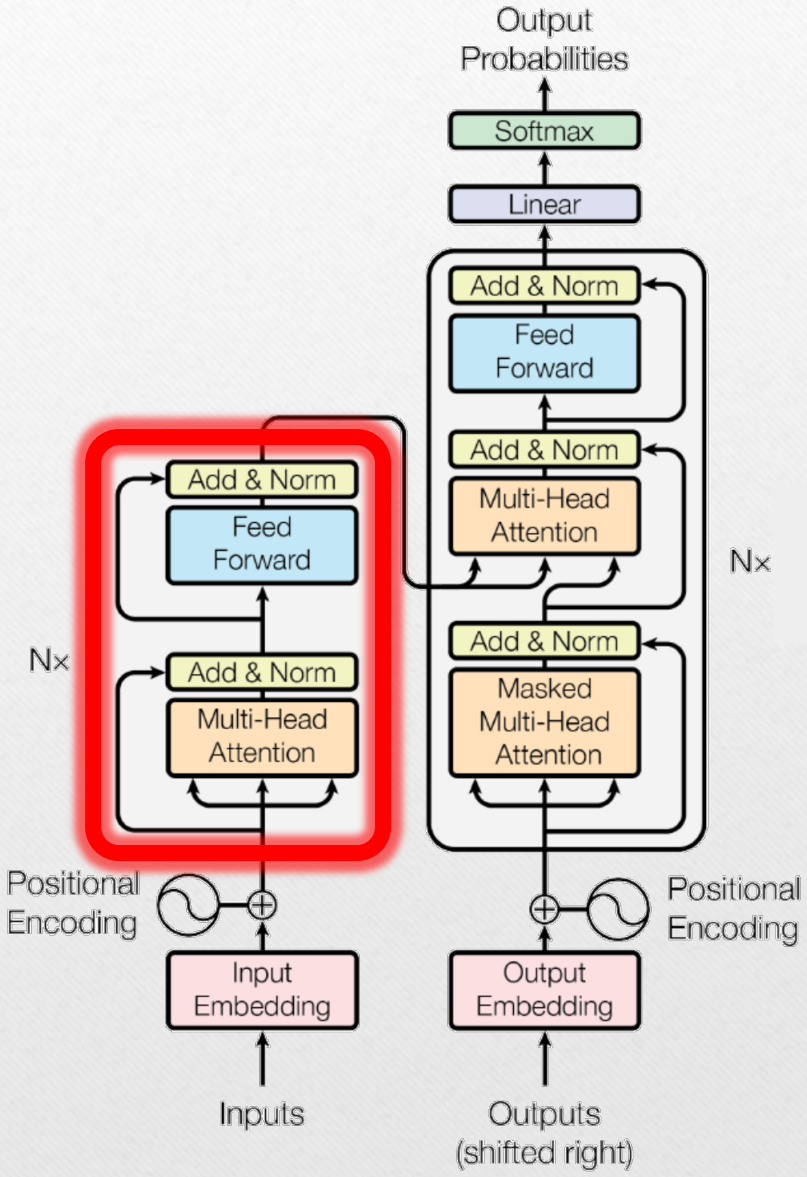
- What we just saw is **single-head (self-)attention**.
- **Multi-head attention** simply means we have multiple copies of the process, each with different weights.
- The weights from the heads are concatenated and multiply to yet another set of weights, generating the final output of the attention layer:

$$\vec{z}_i = [\vec{z}_{1,i}, \ldots, \vec{z}_{h,i}]W_o$$



Encoder

- The output from the attention mechanism is then
  - Added to the output of the previous layer (i.e. residual connection)
  - Normalized
  - Fed to a fully-connected network
  - Another residual connection added and normalized
- The whole set of computation is called a **Transformer Encoder Layer**.
- Multiple encoder layers stacked together form the **Transformer Encoder**.
- The final output is a vector for each word, just like RNN.

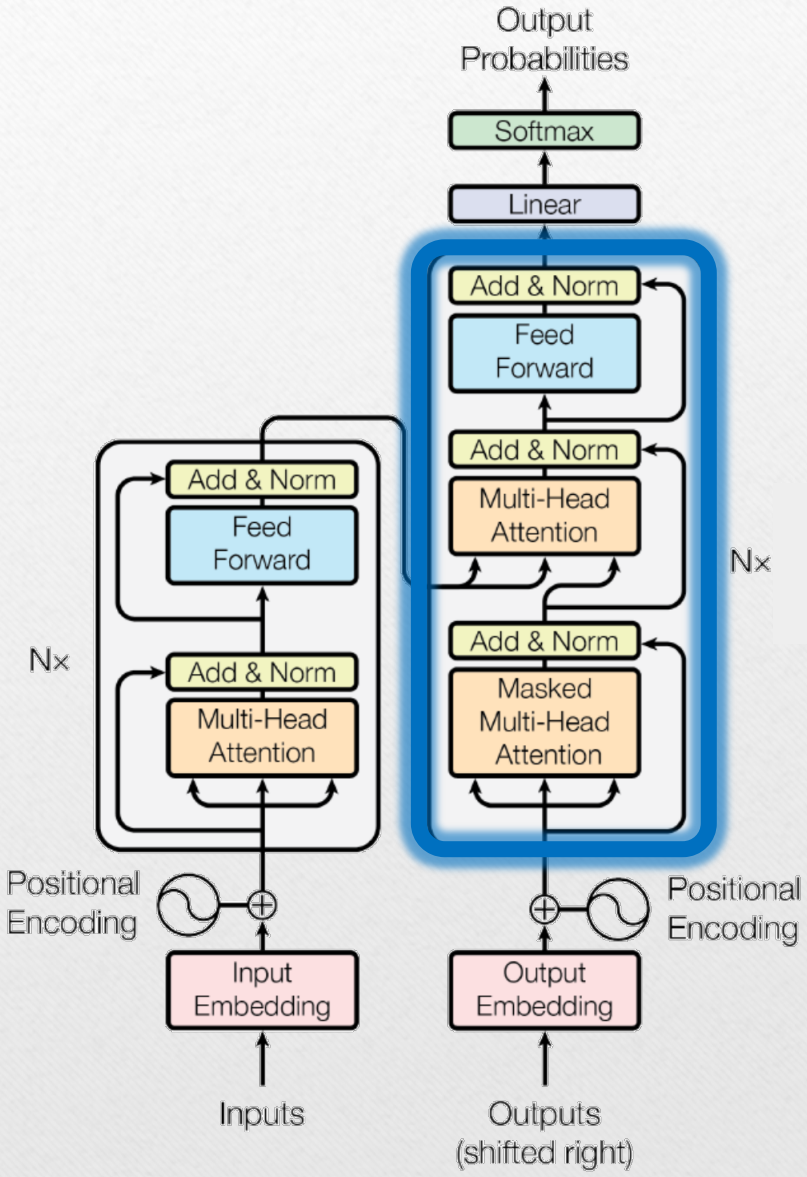


Decoder

The **Transformer Decoder** is responsible for generating a new string of text based on the input.

A **Transformer Decoder Layer** is very similar to the encoder layer, with two crucial differences:

1. The self-attention mechanism is masked. When we process word i, the keys and values of words $> i$ are not used. This is similar to unidirectional RNN.
2. There is a second attention mechanism where the keys and values are the output of the encoder.

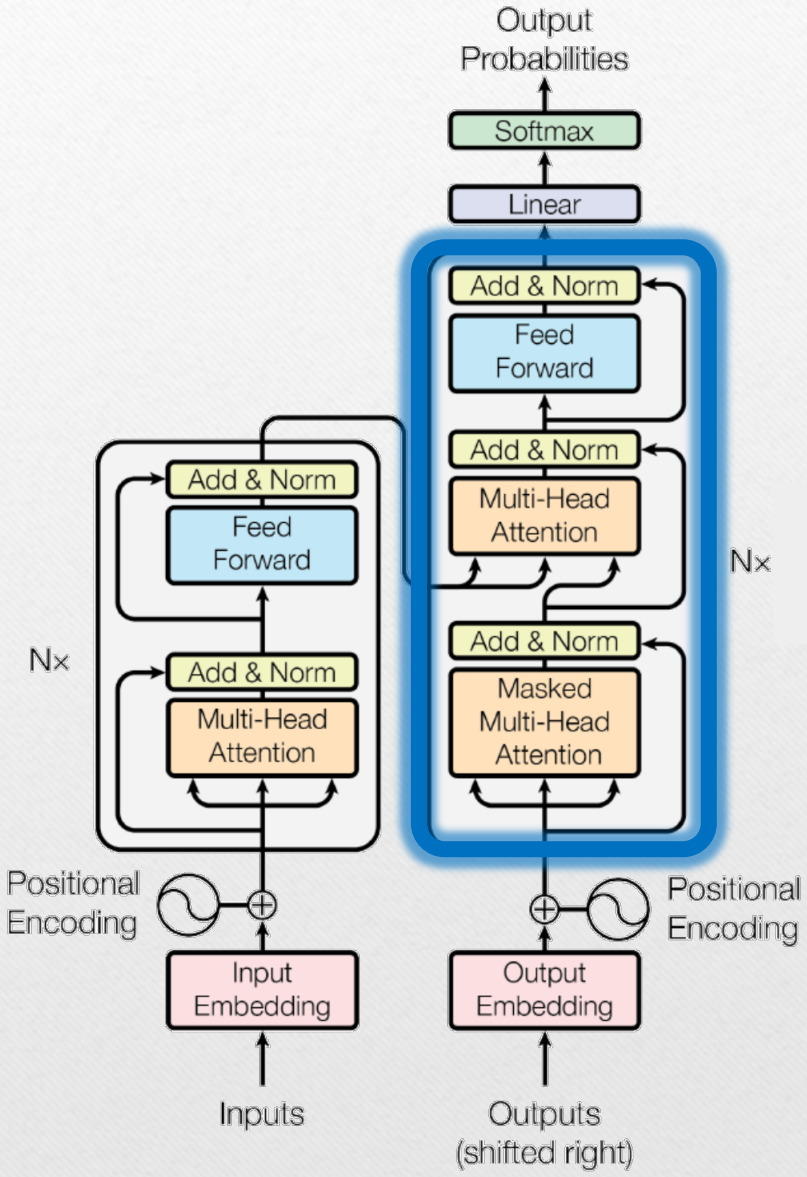


Decoder

The decoder takes a list of words as input. It predicts what the next word should be.

During training, the input to the decoder is simply the desired output shifted right by one word.

- Desired output:
  `John ate an apple [end]`
- Input:
  `[start] John ate an apple`
- When predicting `ate`, the effective input is `[start] John`



Important Families of Transformer Model

There are three major families of Transformer-based models:

- The original Transformer has Transformer Encoder and Decoder. The most important model in this family is Google's T5.
- GPT (**G**enerative **P**re-**T**raining) only uses the Transformer Decoder. Models in this family include OpenAI's GPT 1, 2, 3 and 4, Meta's LLaMA 1, 2 and 3, MosaicML's MPT and TII's Falcon.
- BERT (**B**idirectional **E**ncoder **R**epresentation from **T**ransformers) only uses the Transformer Encoder. Models in this family include BERT, RoBERTa, DistilBERT and ALBERT.

Inference of Decoder-Only Model

The decoder takes a list of words as input. It predicts what the next word should be.

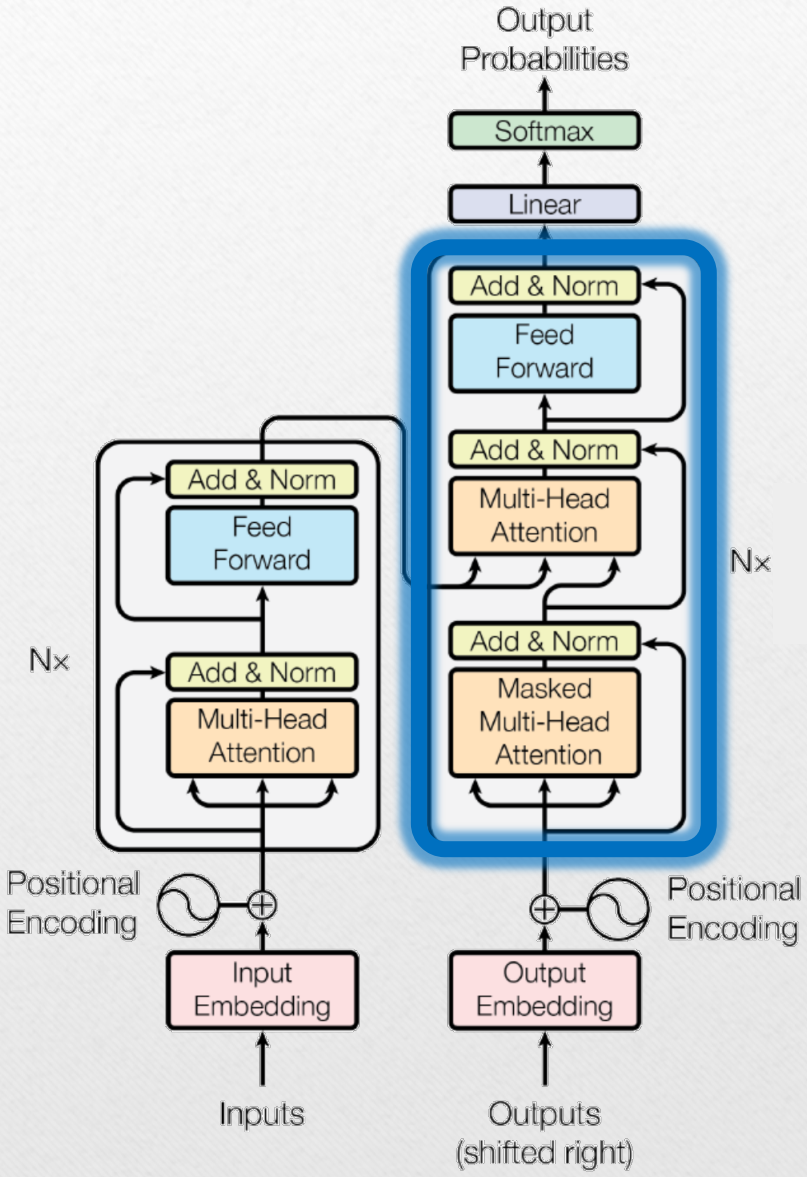
During inference, after the decoder generates the next word, it will be used as part of the input to generate the word after the next.

`[start]` → `?`

Say the model predicts `Paul`, so next we have:

`[start] Paul` → `?`

The way decoder-only models work means they are mostly used for text generation and not other tasks.

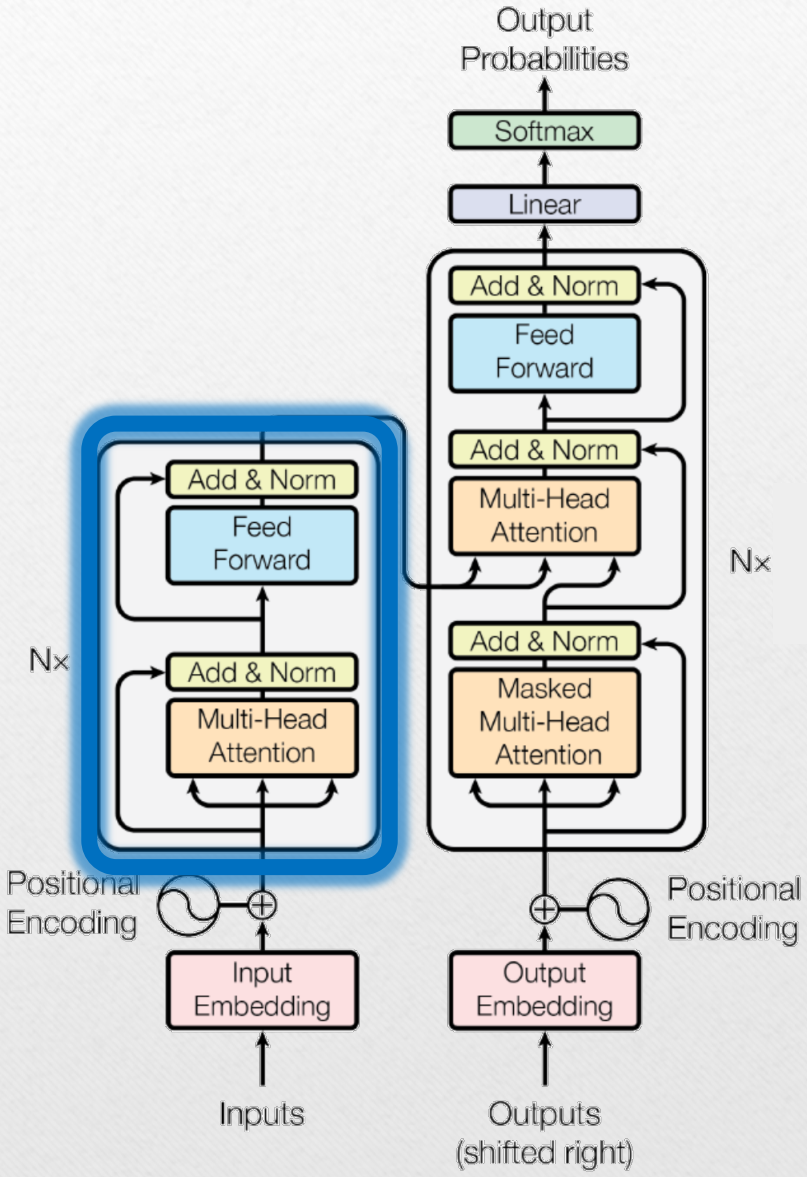


Inference of Encoder-Only Model

When using a trained model for inference, the following happens:

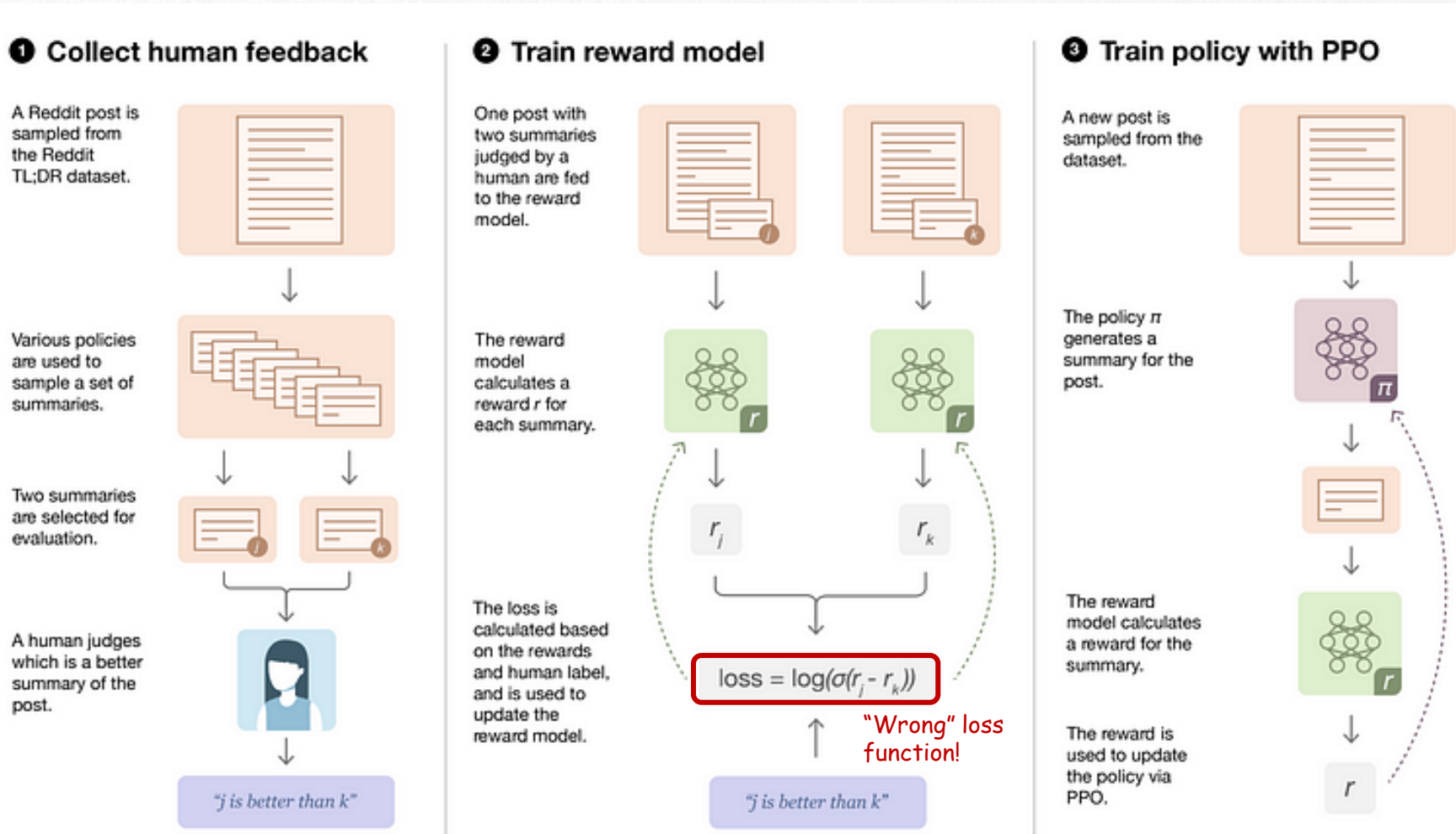
1. Provide the encoder with the input text, e.g. a sentence in English.
2. Encoder generates a vector for each word.

Because the generated vector for each word utilizes information from all words, it is common to use just one vector—e.g. the first word—for downstream tasks such as text classification.



Reinforcement Learning with Human Feedback (RLHF)

- Instruction-following LLMs—i.e. chatbots—are fine-tuned from instruction-answer datasets.
- Such datasets are relatively small—10,000s samples vs billions of samples in the pre-training stage. How to scale up the fine-tuning process?
- **RLHF** trains a reward model, itself a small LLM, based on a relatively small number of samples of human choices over pairs of generated answers
- This reward model can then be used to fine-tune the chatbot over many *generated* samples



Source: Nisan Stiennon et al (2020). "Learning to Summarize from Human Feedback". NeurIPS
Also read: Proximal Policy Optimization

Text Generation Methods

- The decoder takes a list of words as input. It predicts what the next word should be.

  `[start]` → `?`

- The simplest way to do this is to return the most likely word.

$$x_t = \underset{x \in X}{\operatorname{argmax}} P(x|x_{t-1})$$

  This is called **greedy search**.

- Greedy search often produce text that is natural, because the sequence of next most likely words is usually not the sequence that has the highest overall probability. i.e.

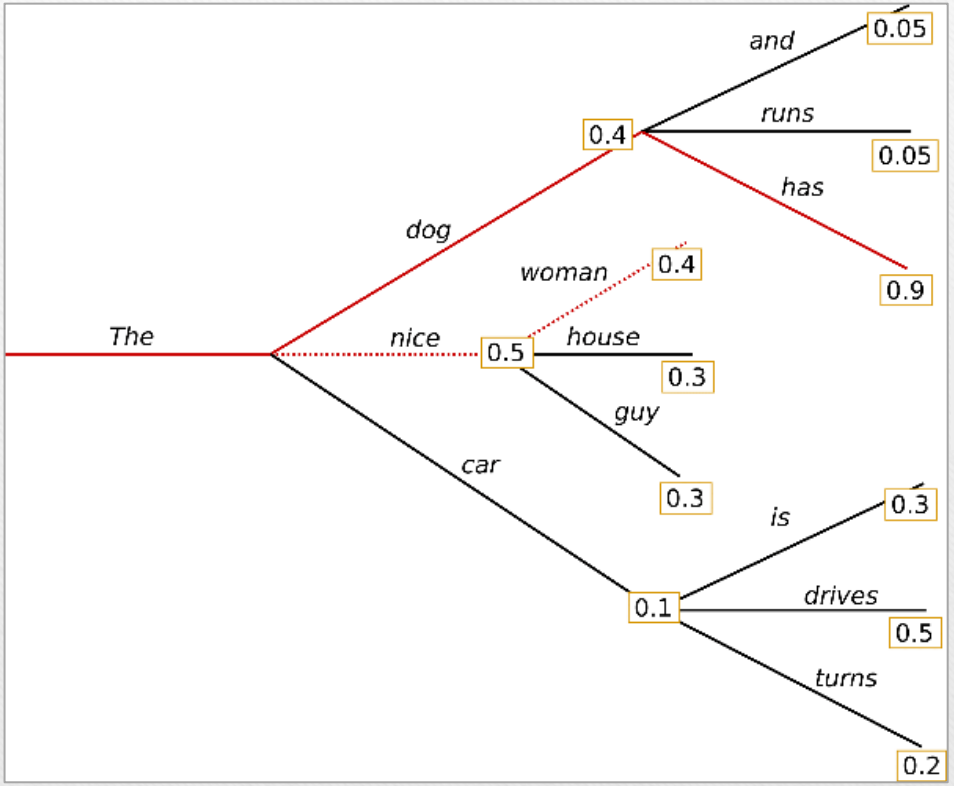
$$x_t = \underset{x \in X}{\operatorname{argmax}} P(x|x_{t-1}) \text{ and } x_{t+1} = \underset{x \in X}{\operatorname{argmax}} P(x|x_t)$$

  does not give you the same answer as

$$x_t, x_{t+1} = \underset{x_t, x_{t+1} \in X}{\operatorname{argmax}} P(x, x_{t+1}|x_{t-1})$$

Text Generation Methods

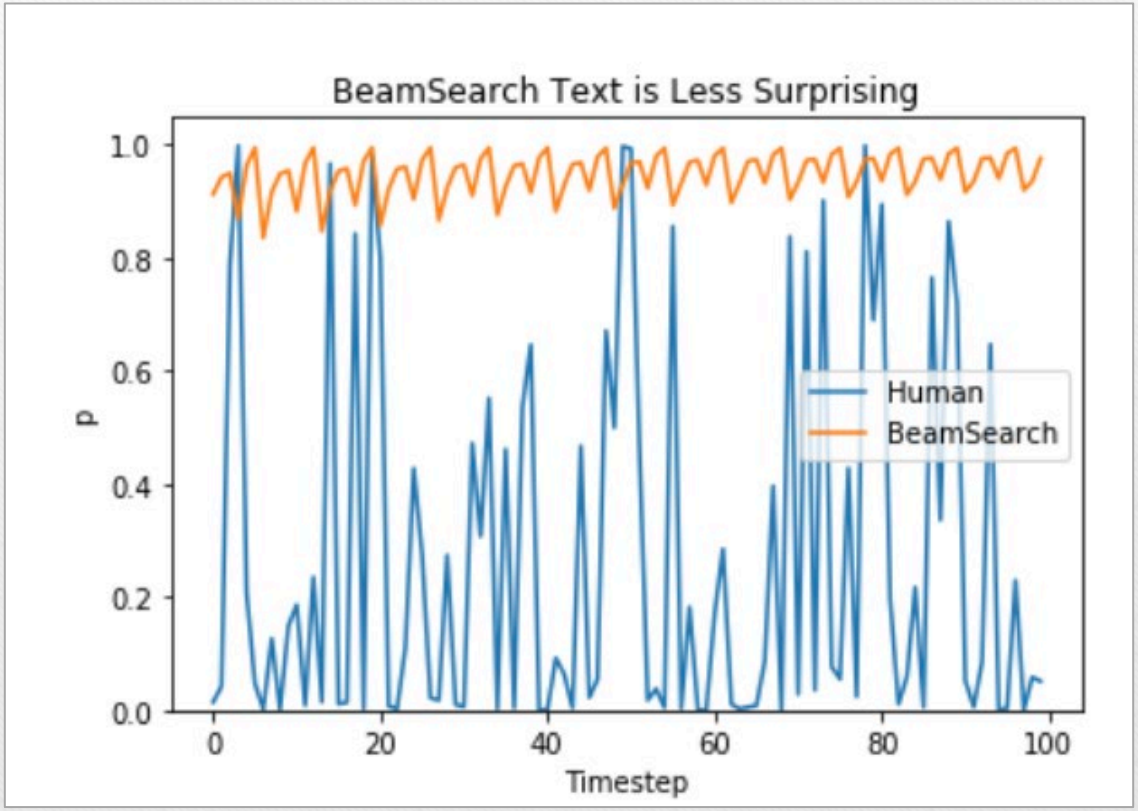
Beam search generates multiple sequences of words and pick the sequence that has the highest overall probability. The number of sequences to keep track of is a hyperparameter.



Source: HuggingFace Blog

Text Generation Methods

But human speech is not based on uttering the next most likely word:



Source: HuggingFace Blog

Text Generation Methods

Regardless of whether we use beam search or not—chatbots most do not due to latency issue—we can also **sample** the next word based on their estimated probability of appearing, instead of always picking the most likely word.

$$\mathrm{P}(x_t = x_i) = P(x_i|x_{t-1})$$

Hyperparameters that control sampling:

- **Temperature**: a lower temperature sharpens the distribution. Temperature = 0 is the same as greedy search.
- **Top-K**: Only sample from the top k-most likely words.
- **Top-P**: Only sample words are within the top cumulative distribution of p
- **Repetition penalty**: lower the probability of sampling words that have already been sampled before.