

Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see
<https://creativecommons.org/licenses/by-sa/2.0/legalcode>



DeepLearning.AI



Generative AI & Large Language Models (LLMs)

**USE CASES,
PROJECT LIFECYCLE, AND
MODEL PRE-TRAINING**

Generative AI & Large Language Model Use Cases & Model Lifecycle

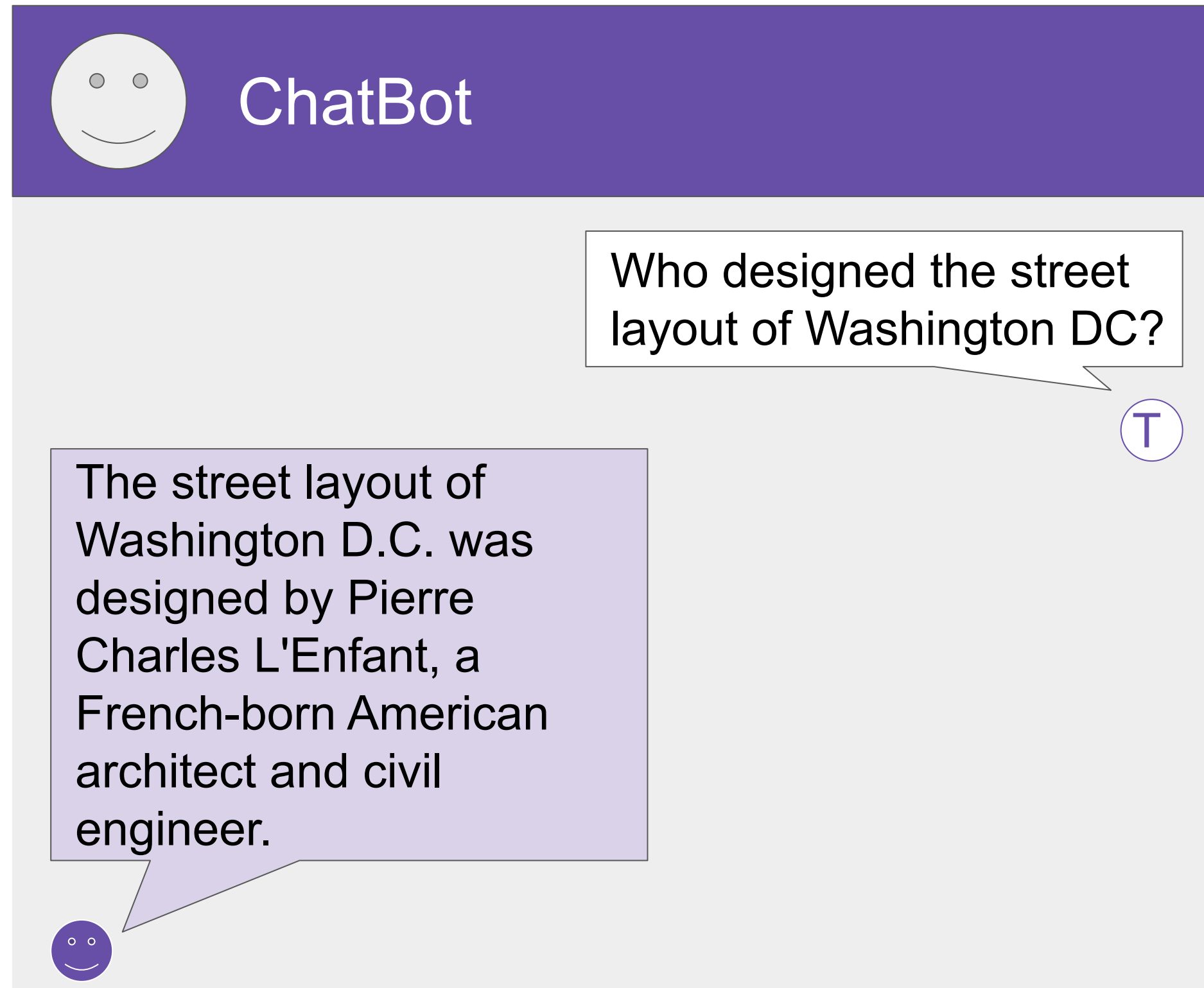


Generative AI & Large Language Models

Below we will see few use cases of Generative AI

Generative AI

Generative AI chat bots
like ChatGPT, Google Bard etc generating responses in the form of TEXT



Generative AI

Generating images from text

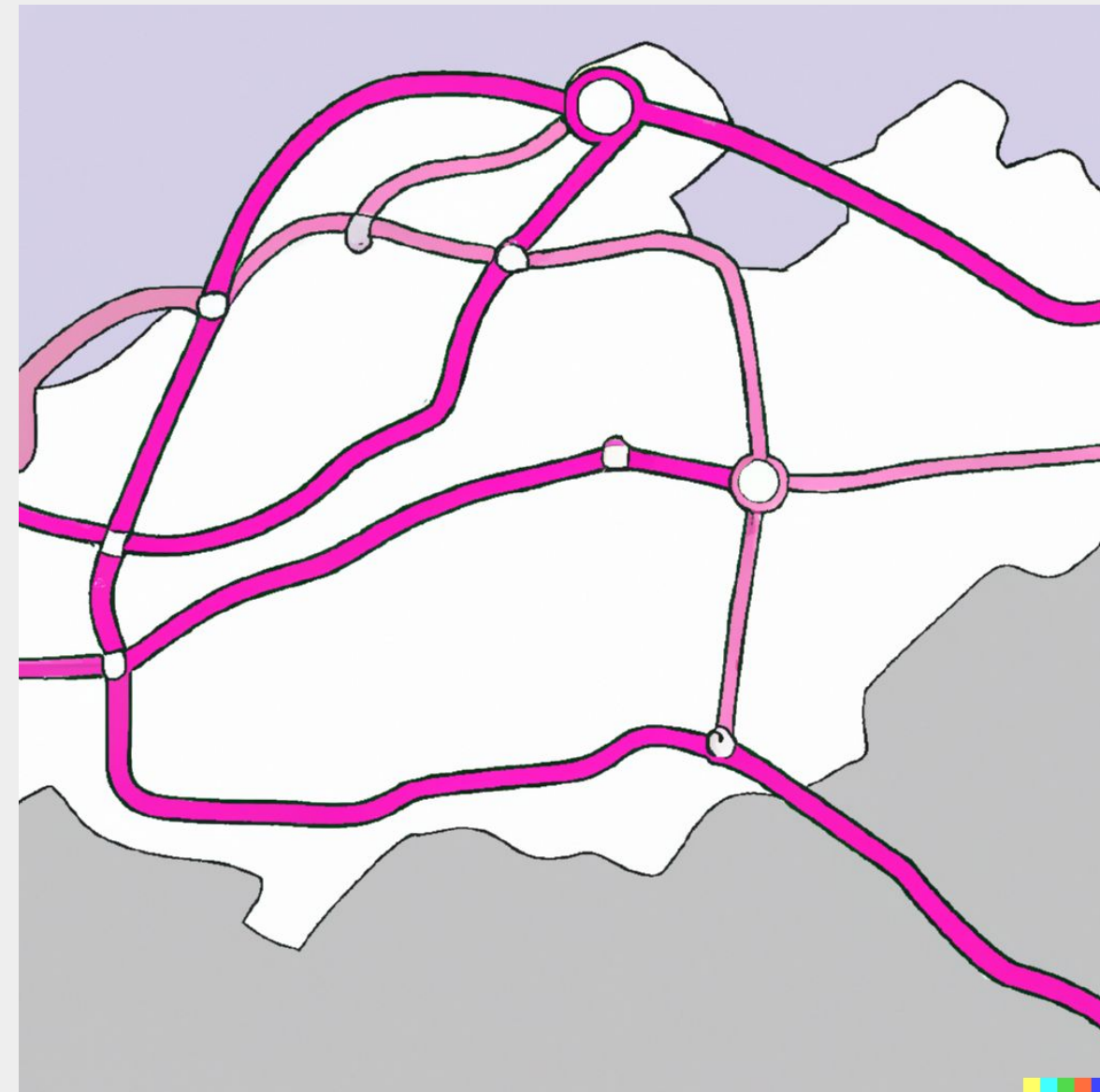
pAIntBox

What do you want to create?

**An imaginary subway map
in a coastal city.**

Image dimensions: by (Max 2048)

Generate



Generative AI

Creating a Plug in to help developers develop a code


Generative AI is the subset of Machine Learning and ML algorithms that underpins Gen AI uses statistical model that finds the pattern in the huge amount of data generated by humans

CodeAId

```
1 def binary_search(arr, x, l, r):_
2     if r >= l:
3         mid = l + (r - l) // 2
4         if arr[mid] == x:
5             return mid
6         elif arr[mid] > x:
7             return binary_search(arr, x, l, mid - 1)
8         else:
9             return binary_search(arr, x, mid + 1, r)
10    else:
11        return -1
```

< 1/2 > Accept

Tab

AI Connected 

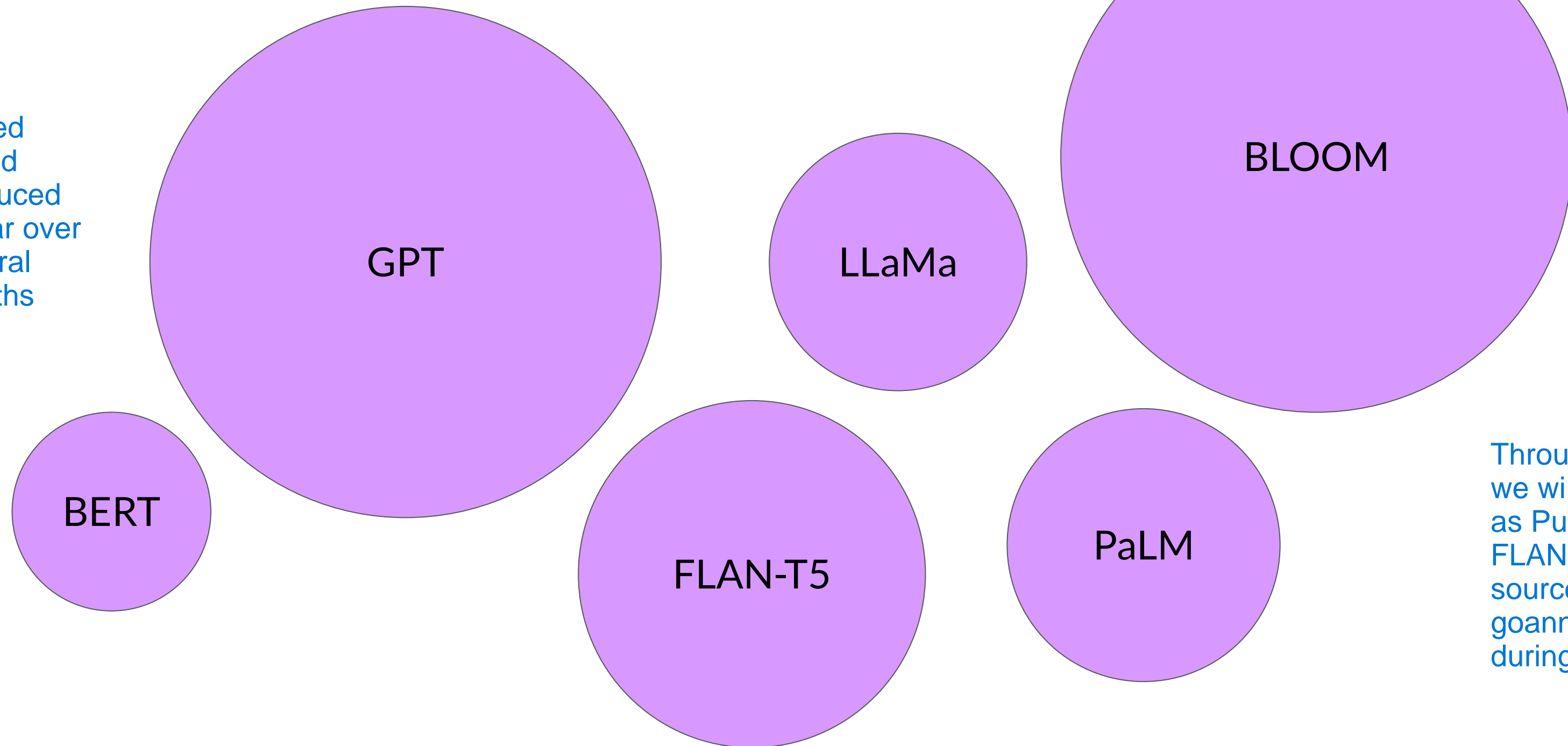
Run security scan

In all the above uses cases discussed we can see that these tools represents the machine capable of creating the content mimicking human ability

Large Language Models

Below are collection of foundation of models, sometimes called base models. These mode are called LLMs. Each of these are having some set of Parameters (in millions, trillions). These Parameters can be seen as the size or memory of model. More Parameter means more memory meaning respective model can perform the more sophisticated tasks.

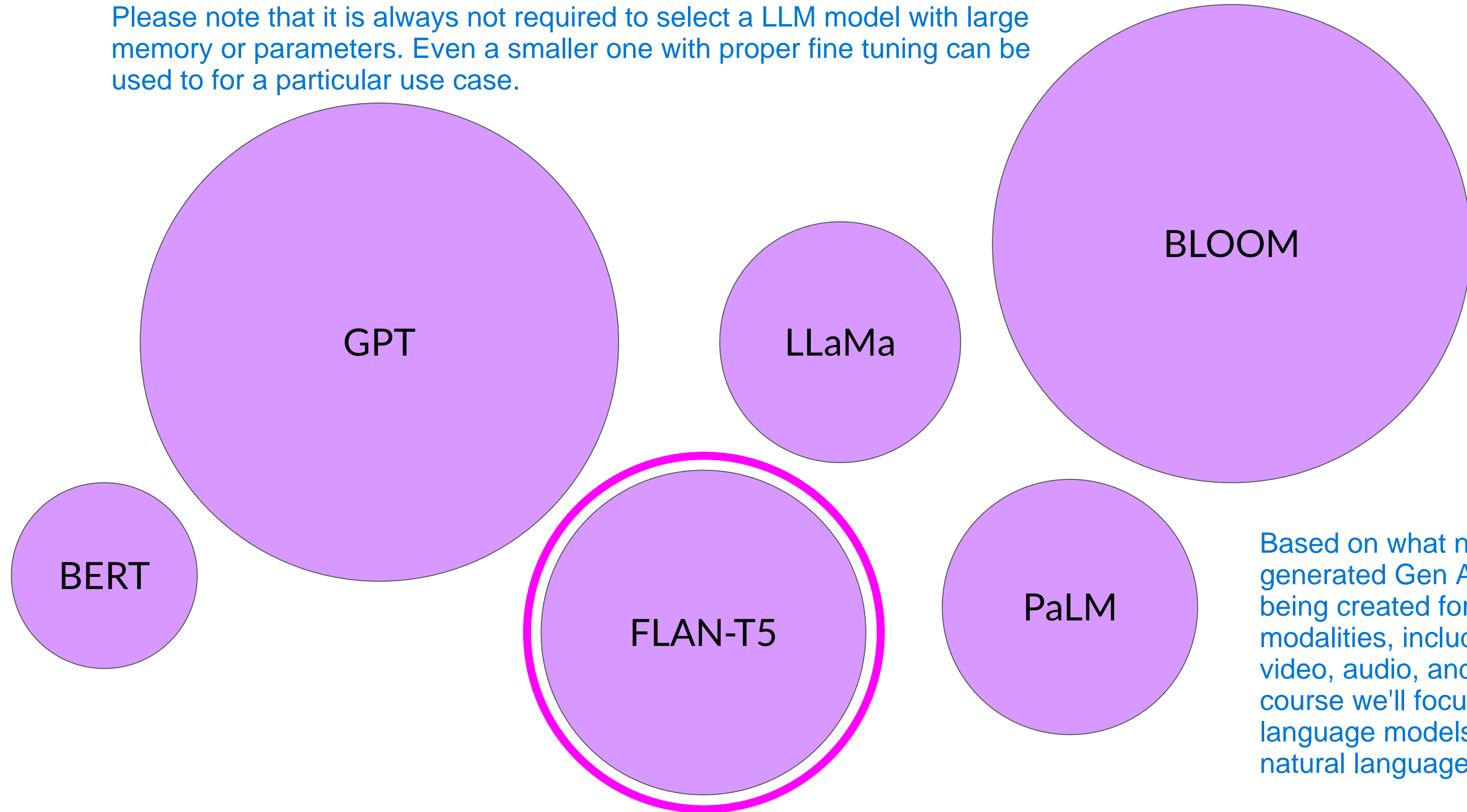
These LLM based model are trained using Data produced by humans so far over a period of several weeks and months



Throughout this course we will represent LLMs as Purple circle and FLAN-T5 is the open source LLM that we goanna make use during practicals.

Large Language Models

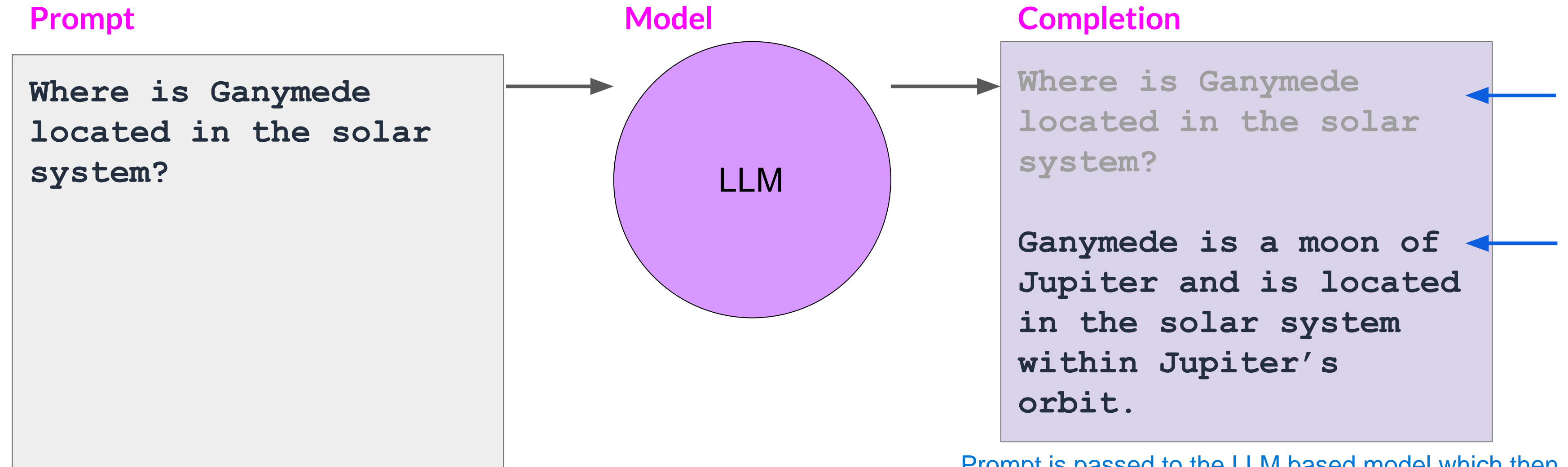
Please note that it is always not required to select a LLM model with large memory or parameters. Even a smaller one with proper fine tuning can be used to for a particular use case.



Based on what needs to be generated Gen AI models are being created for multiple modalities, including images, video, audio, and speech, in this course we'll focus on large language models and their uses in natural language generation.

Prompts and completions

The way we interact with language models is quite different than other machine learning and programming paradigms. In those cases, we will write computer code with formalized syntax to interact with libraries and APIs. In contrast, large language models are able to take natural language or human written instructions called PROMPT and perform tasks much as a human would.



Context window

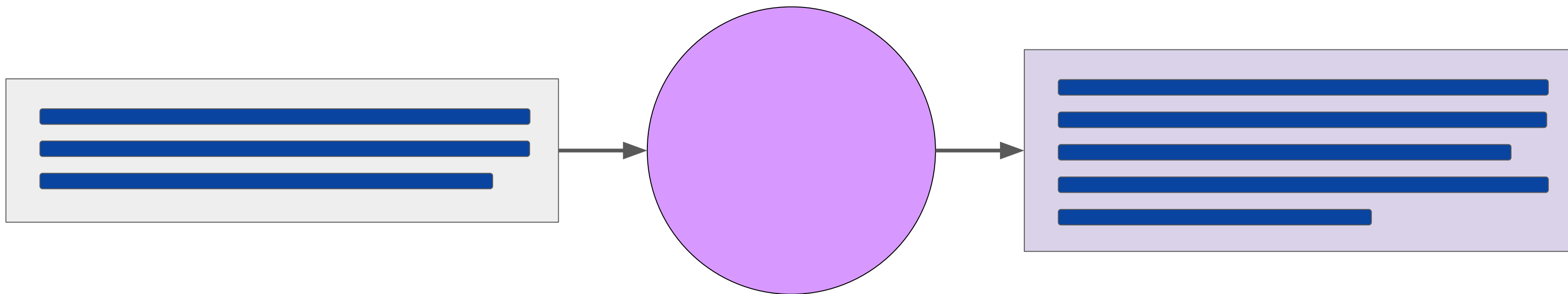
- typically a few 1000 words.

The space or memory that is available to the prompt is called the **CONTEXT WINDOW**, and this is typically large enough for a few thousand words, but differs from model to model.

Prompt is passed to the LLM based model which then produces an Output describing prompt(in text, audio, video or speech) that is called **COMPLETION**.

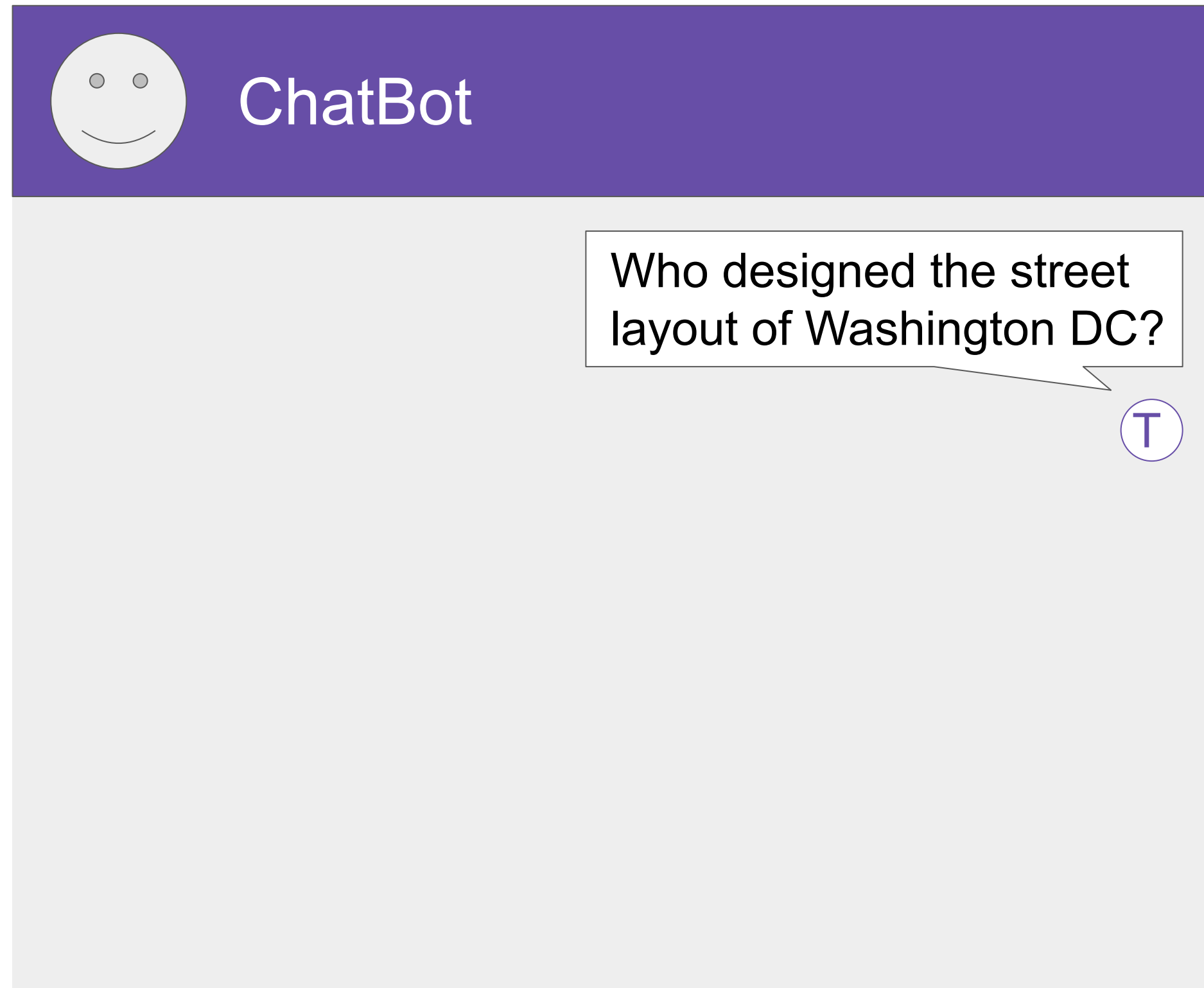
AND this whole process of generating Completion using Model on top of Prompt is known as **INFERENCE** (Inference Statistics)

Prompts and completions

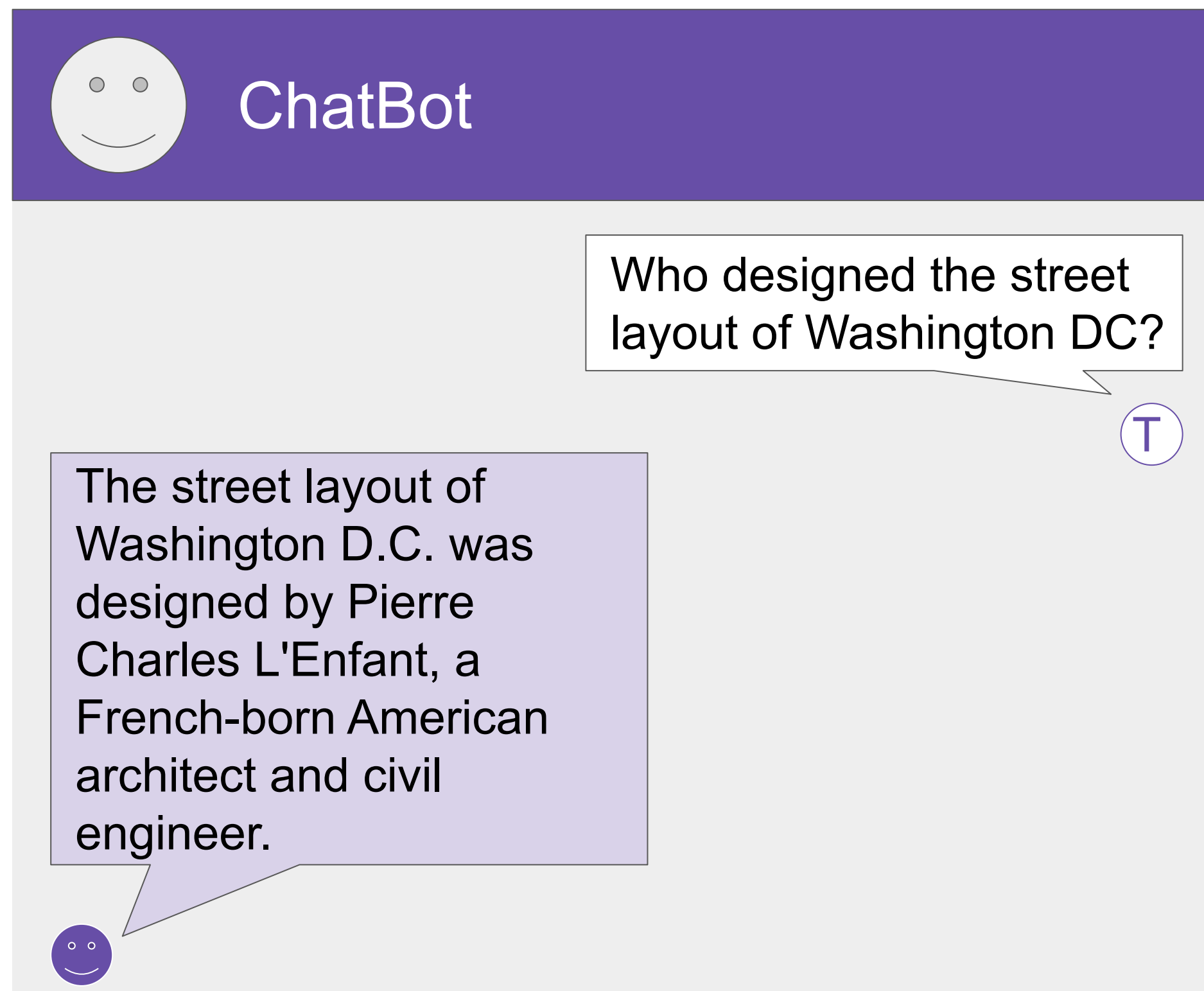


Use cases & tasks

LLM chatbot

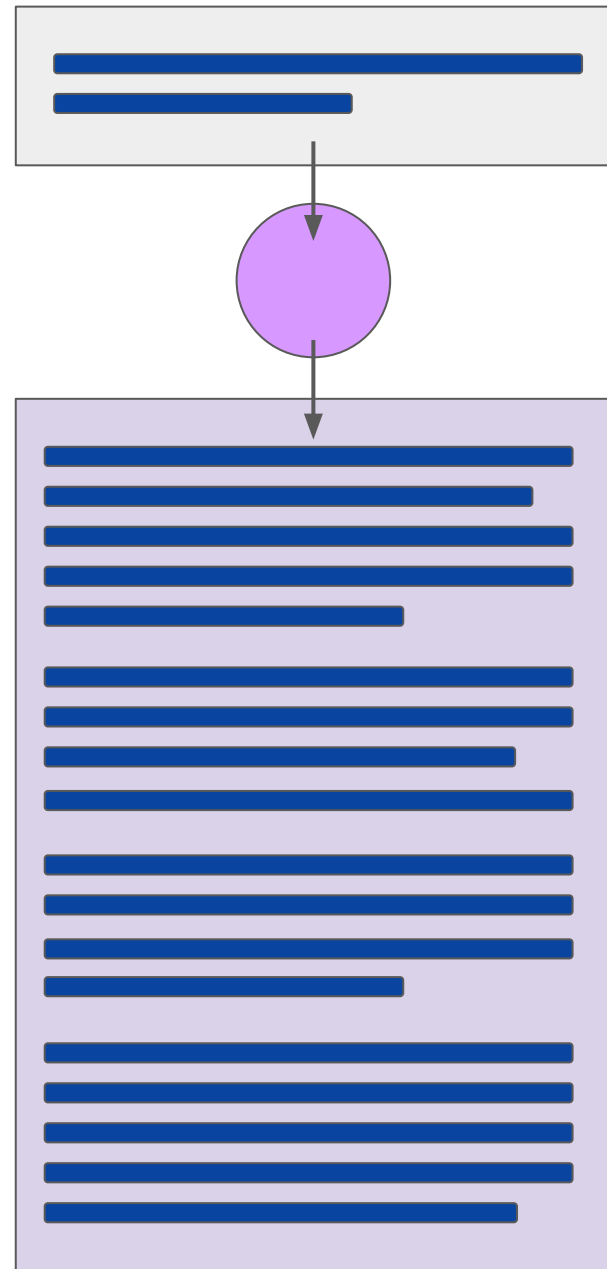


LLM chatbot

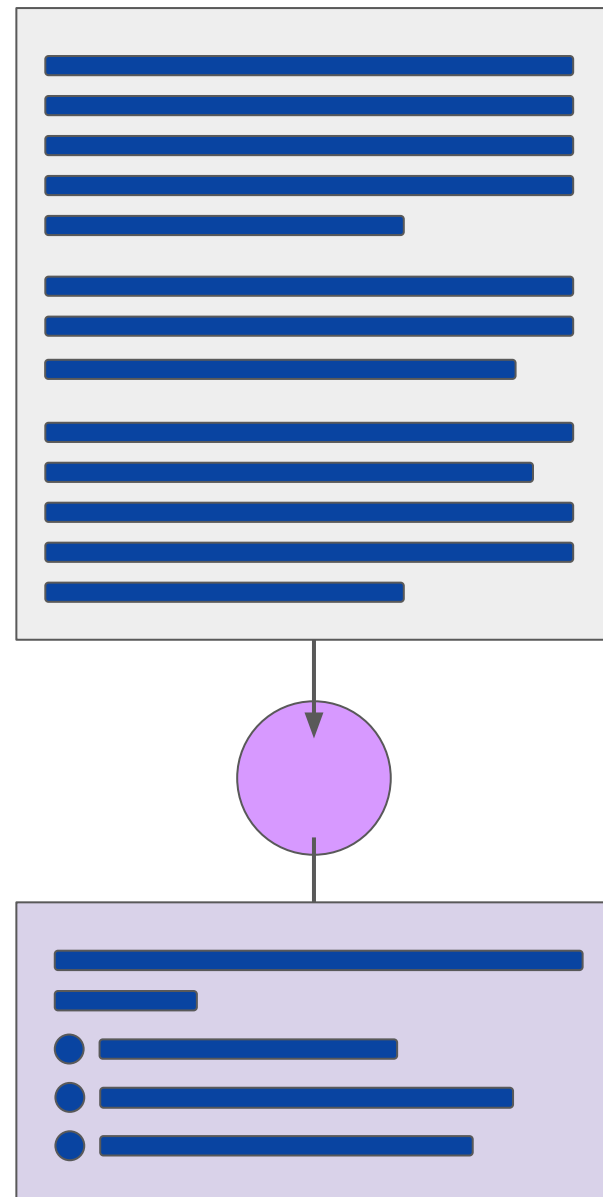


LLM use cases & tasks

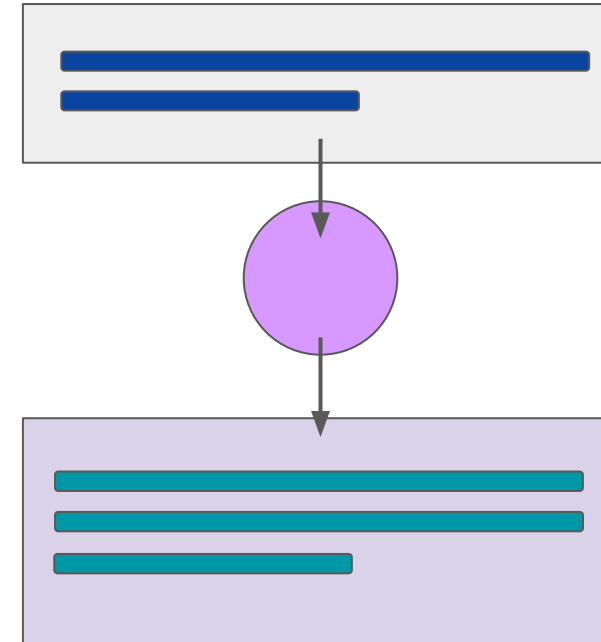
Essay Writing



Summarization

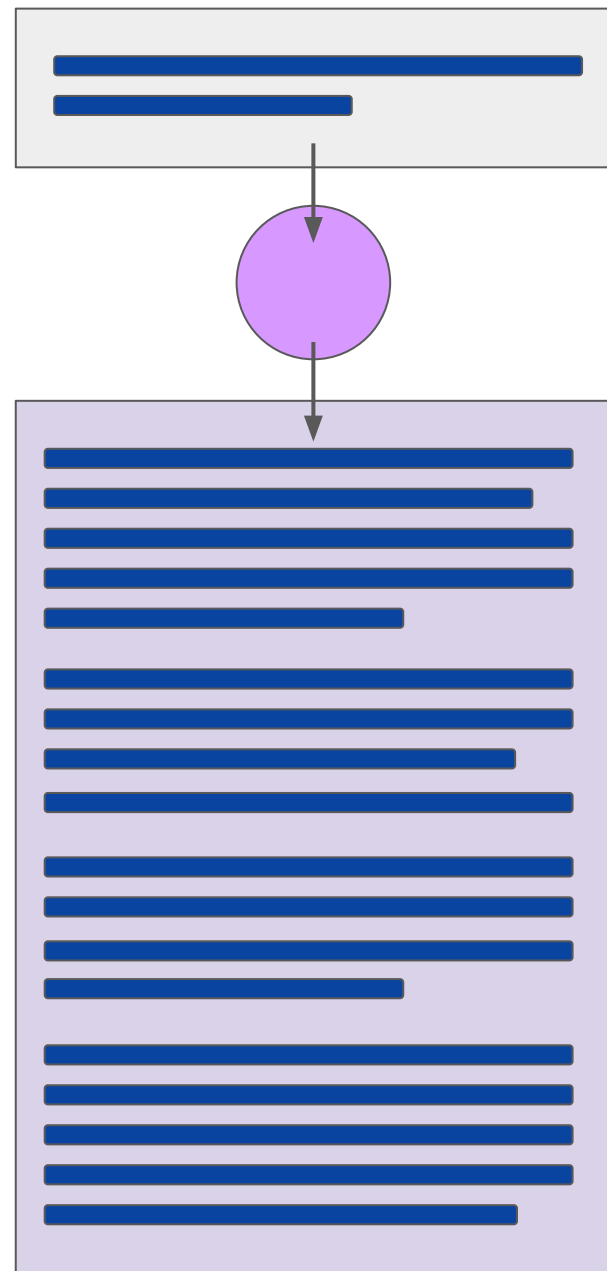


Translation

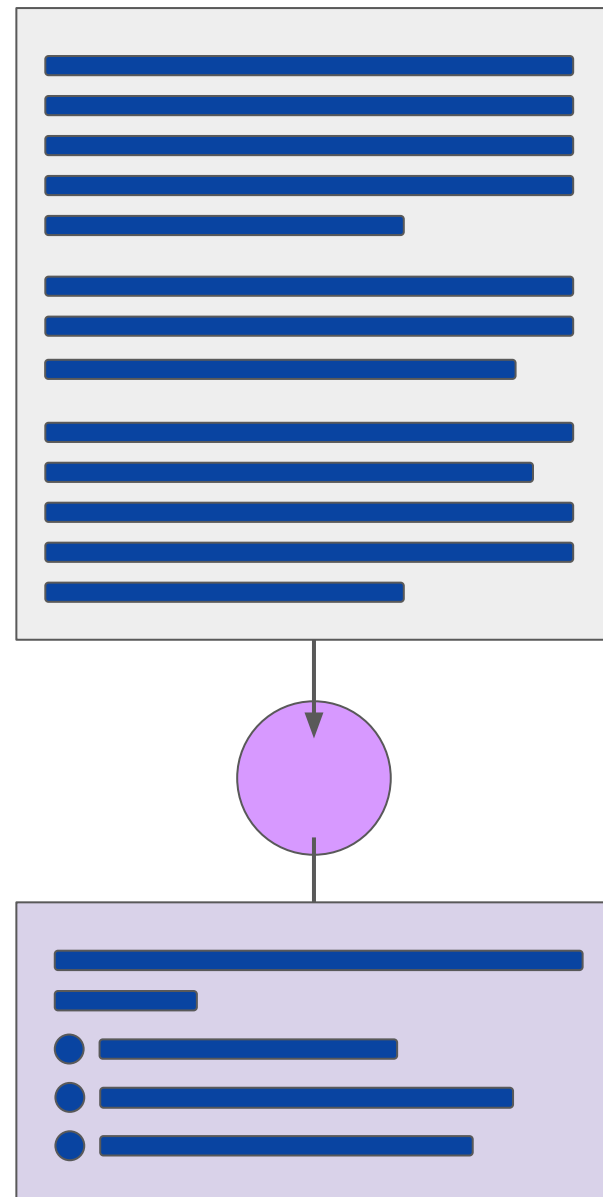


LLM use cases & tasks

Essay Writing

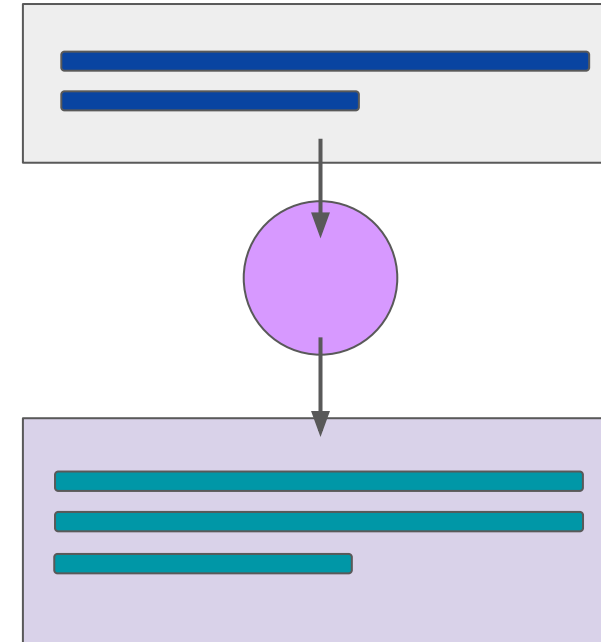


Summarization



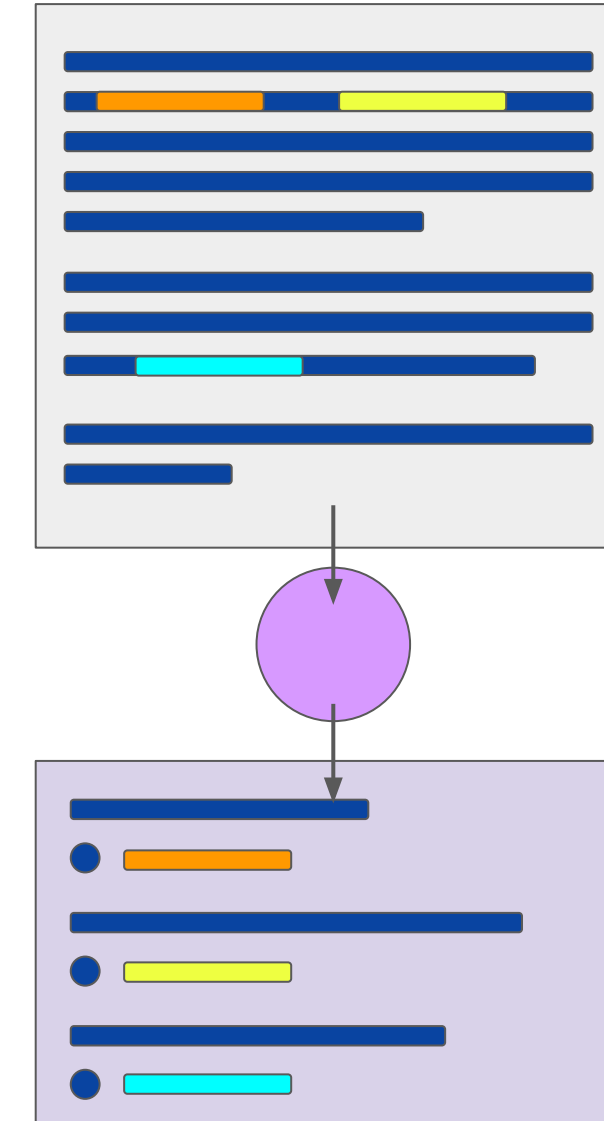
Translation

(E.g: English to Hindi translation)

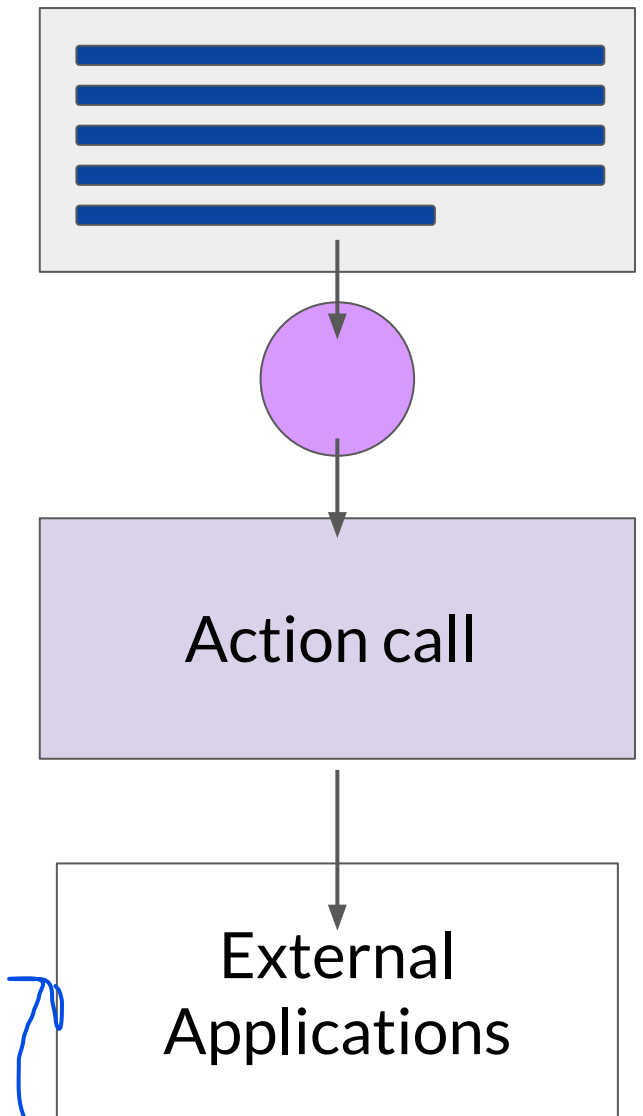


Entity extraction:
That is to identify main keywords
in the sentence. For Eg. Seemashu works with TCS when
passed to model for Entty extraction will give Seemanshu
and TCS as the output.

Information retrieval



Invoke APIs and actions



Open AI plugin features. You can use this ability to provide the model with
information it doesn't know from its pre-training and to enable your model to power
interactions with the real-world. Will learn more in week 3.
For E.g asking ChatGPT when a particular train is going to reach the target
station?

The significance of scale: language understanding

As the scale of foundation models grows from hundreds of millions of parameters to billions, even hundreds of billions, the subjective understanding of language that a model possesses also increases.

This language understanding stored within the parameters of the model is what processes, reasons, and ultimately solves the tasks you give it.

BERT*
110M

BLOOM
176B



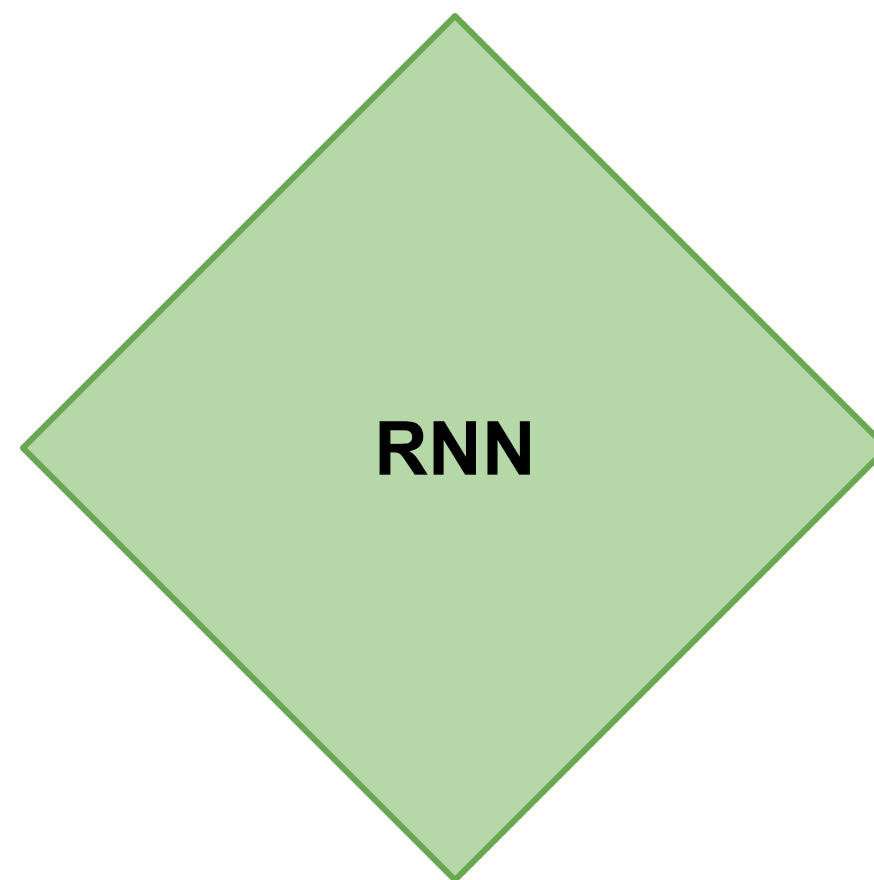
*Bert-base

Transformers are something that powers the LLMs whose architecture and other related things were coined by Google in collaboration with the University of Toronto in 2007 via a Research Paper "Attention is all you need"

How LLMs work - Transformers architecture

Generation algorithms are not new. Earlier we were having RNN that was used as an generative algorithms but due to limited memory and compute it's use cases were limited.

Generating text with RNNs



Generating text with RNNs

Example: Using RNN to generate next word by seeing previous word

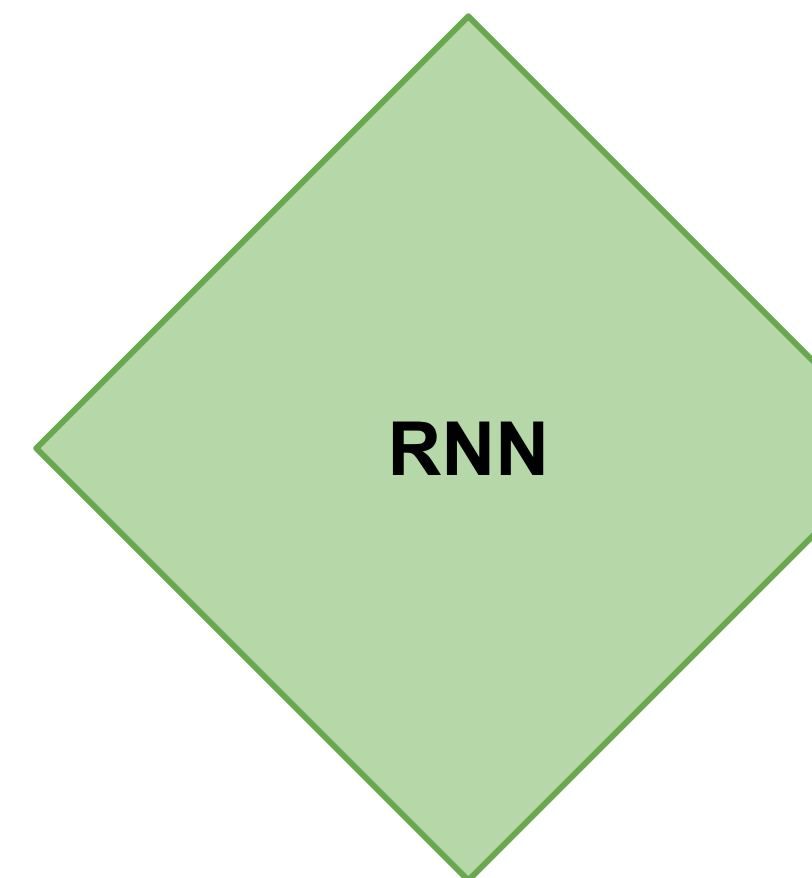
Here , when we give only one word then next word prediction would not be great. So, we can try providing few more words as prefix to predict the next word as an output. This will also, increase the RNN compute and memory resources required



Generating text with RNNs



tea tastes ...



Observe here size of
RNN(Compute and
Memory resource)
increases as we
increase the input size

Generating text with RNNs

?

, my tea tastes ...

RNN

RNN size increases even more

Generating text with RNNs

?

, my tea tastes great.

RNN

As a result RNN predicted great

Generating text with RNNs

The milk is bad, my tea tastes ~~great~~.

RNN

But RNN can generate text or sentence which shows miscellaneous meaning wrt the overall sentence

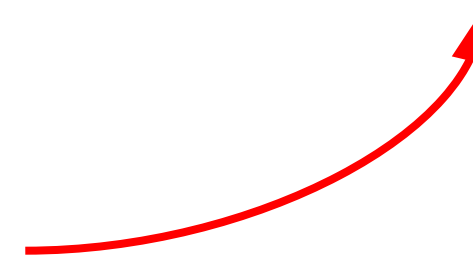
Understanding language can be challenging

Model needs to understand the context of whole sentence or even a document to predict missing words or sentences.

Language is complex where meaning of a word is relative to the context where it is being used. We can understand the same using the below 2 examples:

I took my money to the bank.

River bank?



Homonyms (same word having different meaning) case

Understanding language can be challenging

Syntactic ambiguity case



The teacher's book?

The teacher taught the student with the book.

The student's book?

Transformers

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

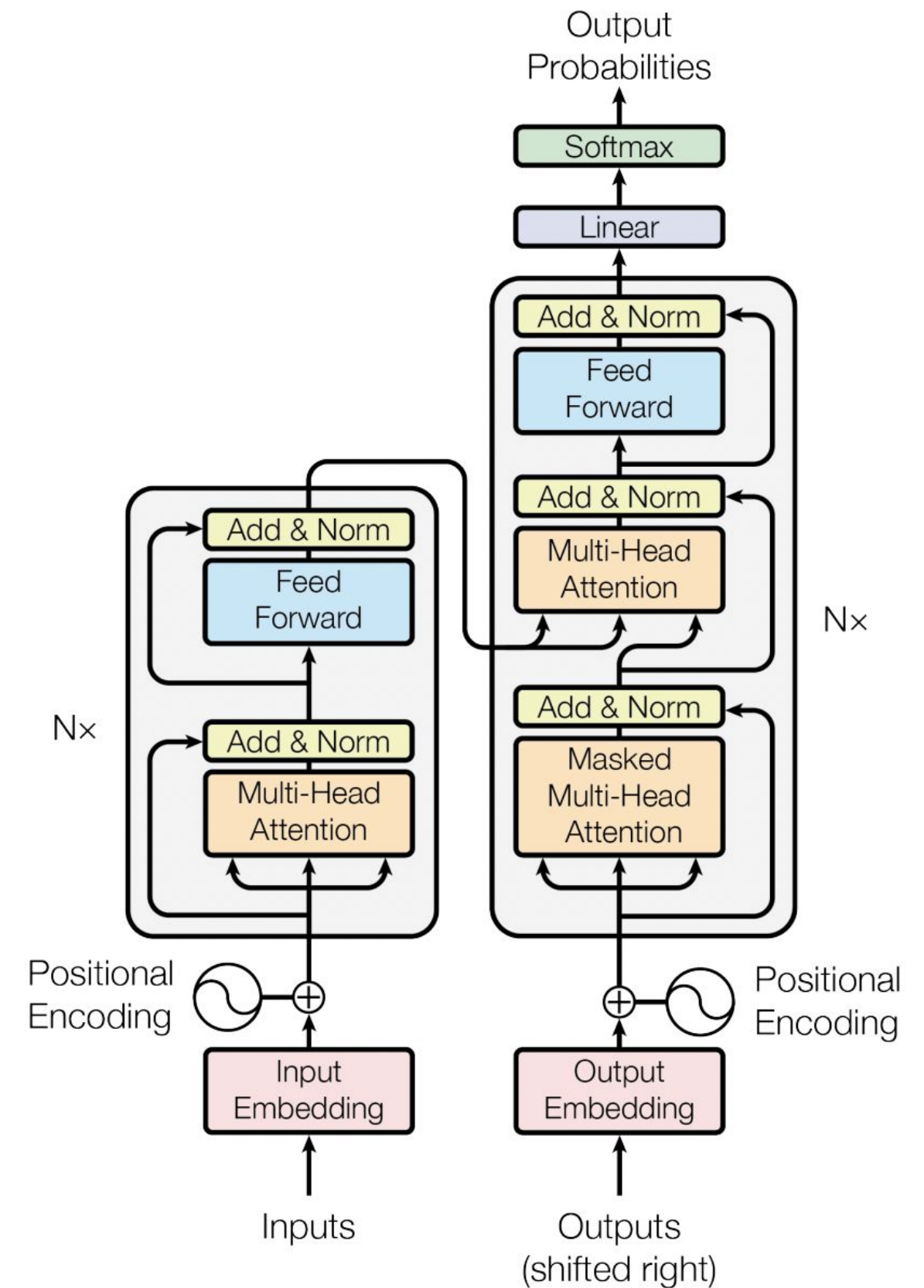
Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaizer@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to



Transformers

In 2017 more Nobel approach came into picture (better than RNN) called Transformers that gave birth to new age of Gen AI that we see today. Following are few advantages of Transformers over previous approaches such as RNN for generation:

- Scale efficiently
- Parallel process
- Attention to input meaning

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com	Noam Shazeer* Google Brain noam@google.com	Niki Parmar* Google Research nikip@google.com	Jakob Uszkoreit* Google Research usz@google.com
---	---	--	--

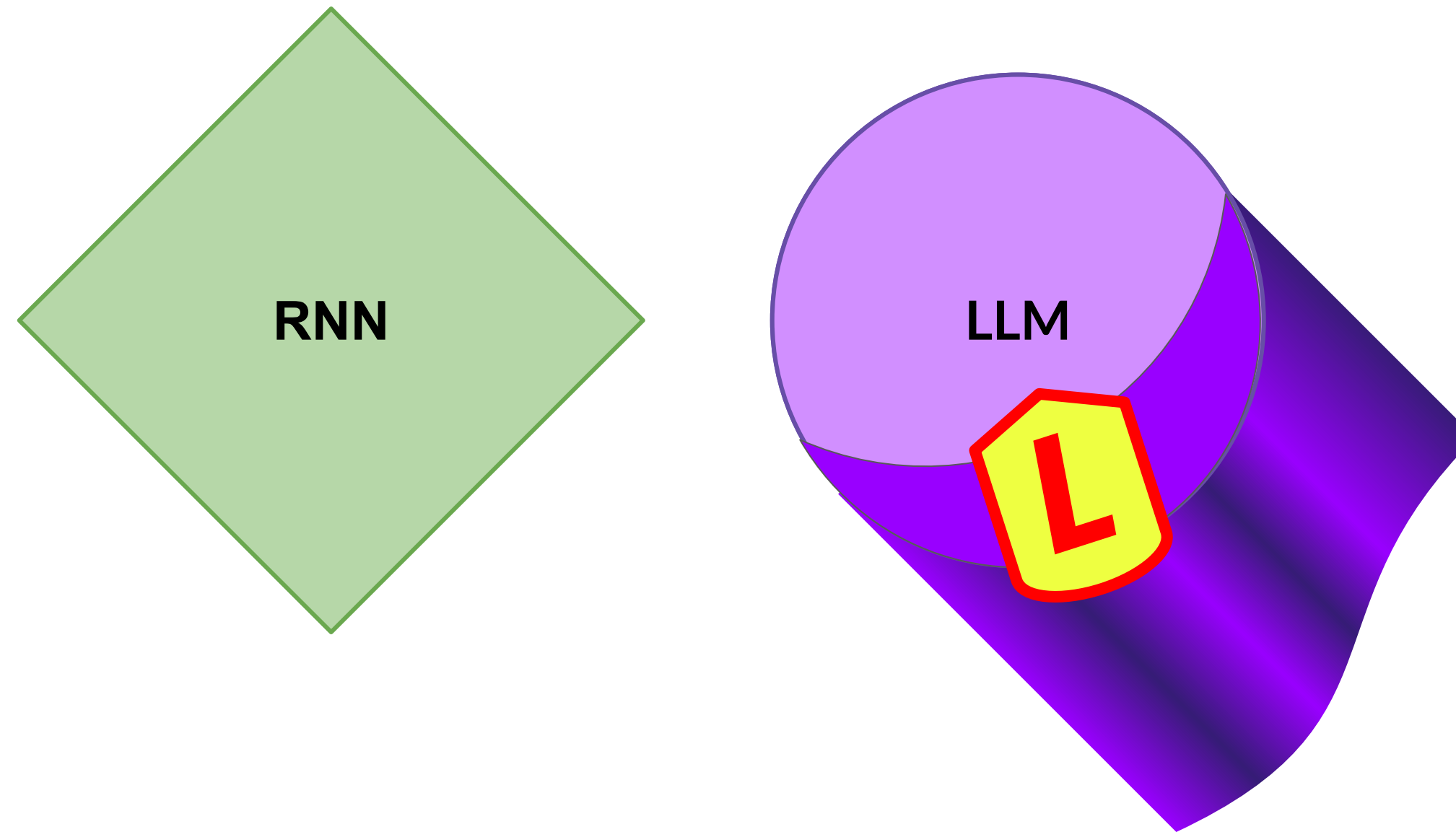
Llion Jones* Google Research llion@google.com	Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu	Łukasz Kaiser* Google Brain lukaszkaizer@google.com
--	---	--

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

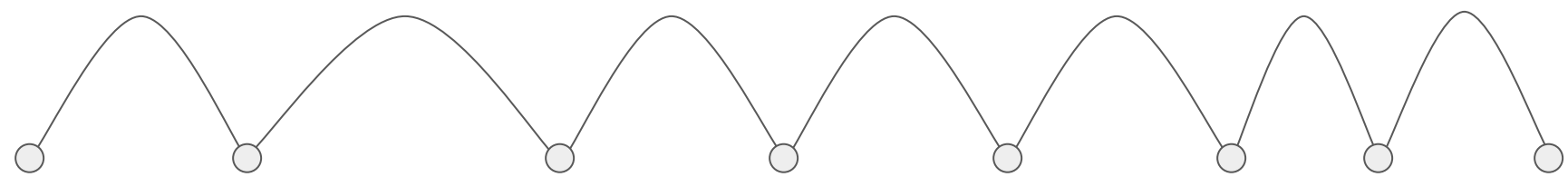
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to

Transformers

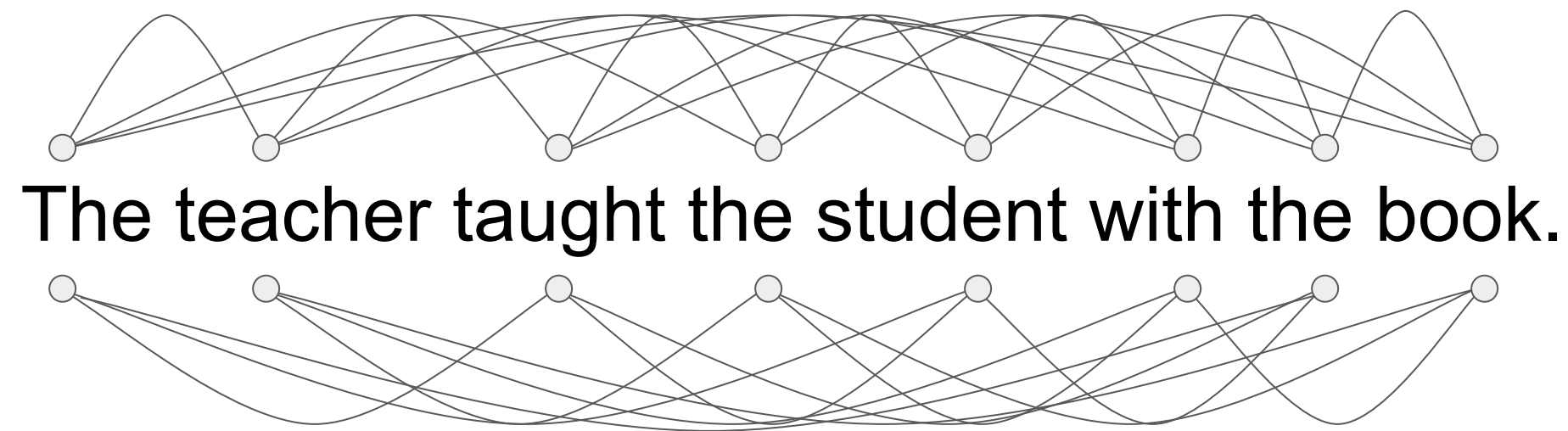


Transformers



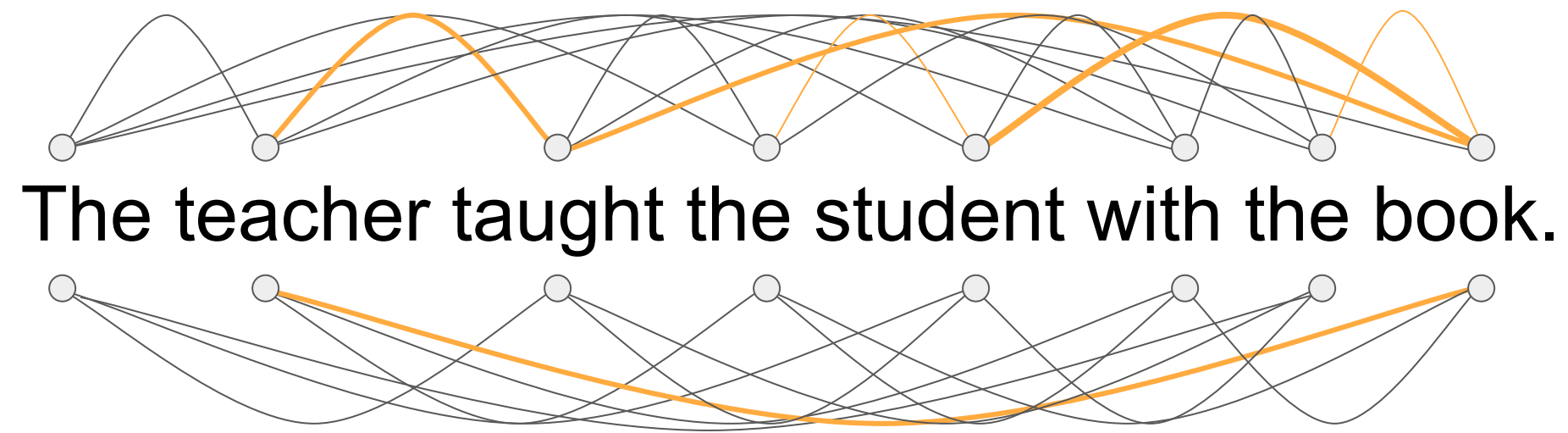
The teacher taught the student with the book.

Transformers



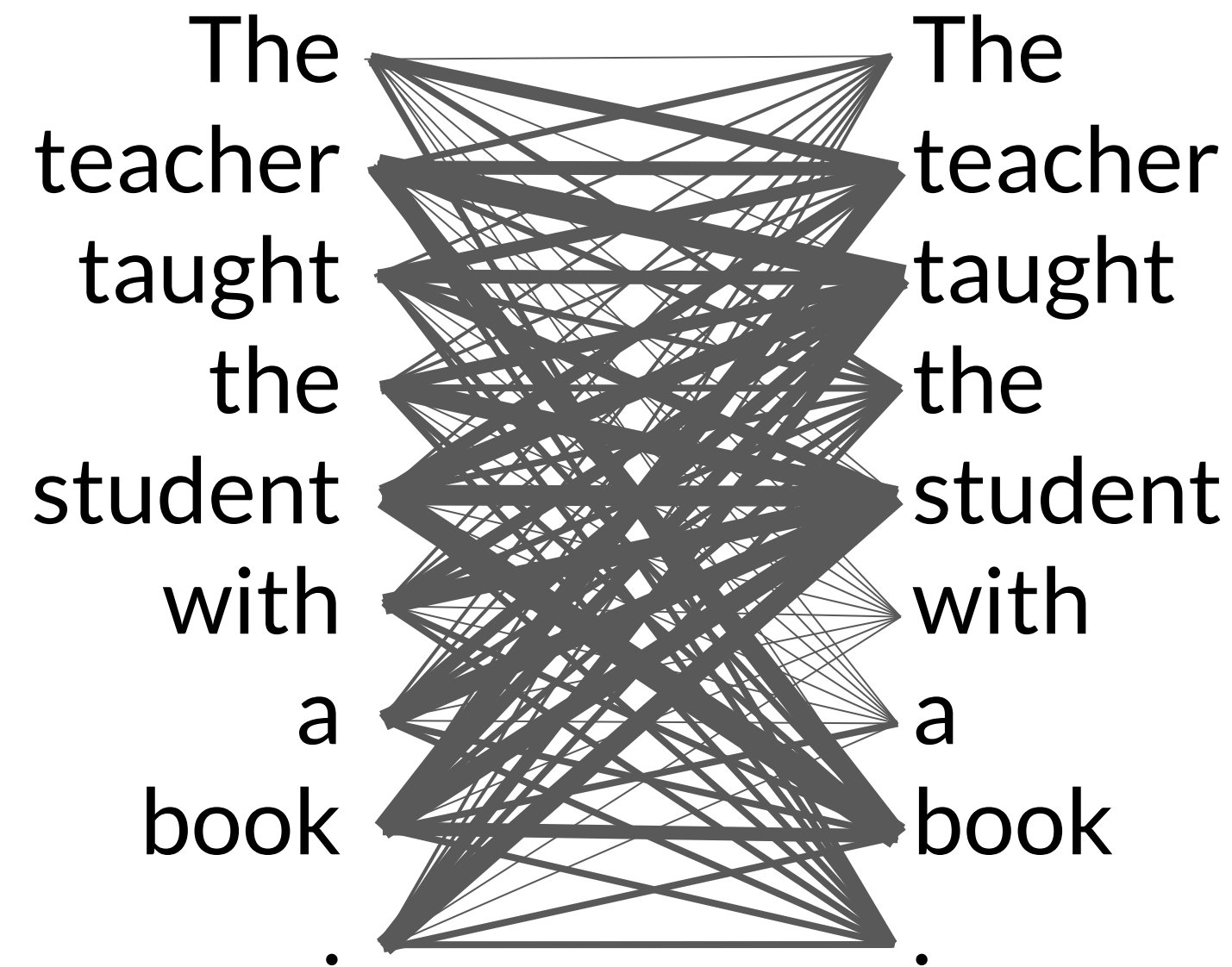
The power of the transformer architecture lies in its ability to learn the relevance and context of all of the words in a sentence. Not just as you see here, to each word next to its neighbor, but to every other word in a sentence. To apply attention weights to those relationships so that the model learns the relevance of each word to each other words no matter where they are in the input. This gives the algorithm the ability to learn who has the book, who could have the book, and if it's even relevant to the wider context of the document. These attention weights are learned during LLM training

Transformers



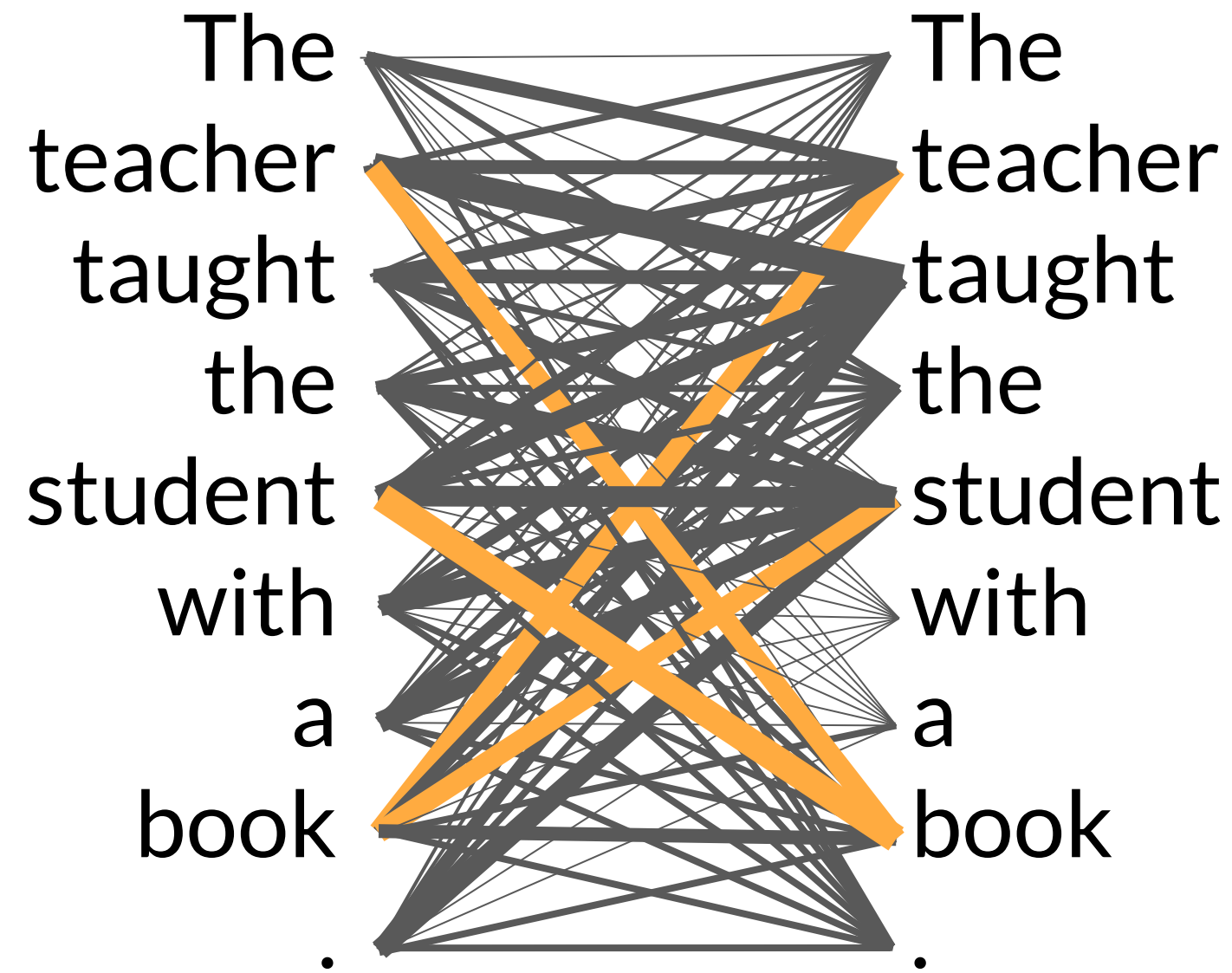
Self-attention

This diagram is called an ATTENTION MAP and can be useful to illustrate the attention weights between each word and every other word.

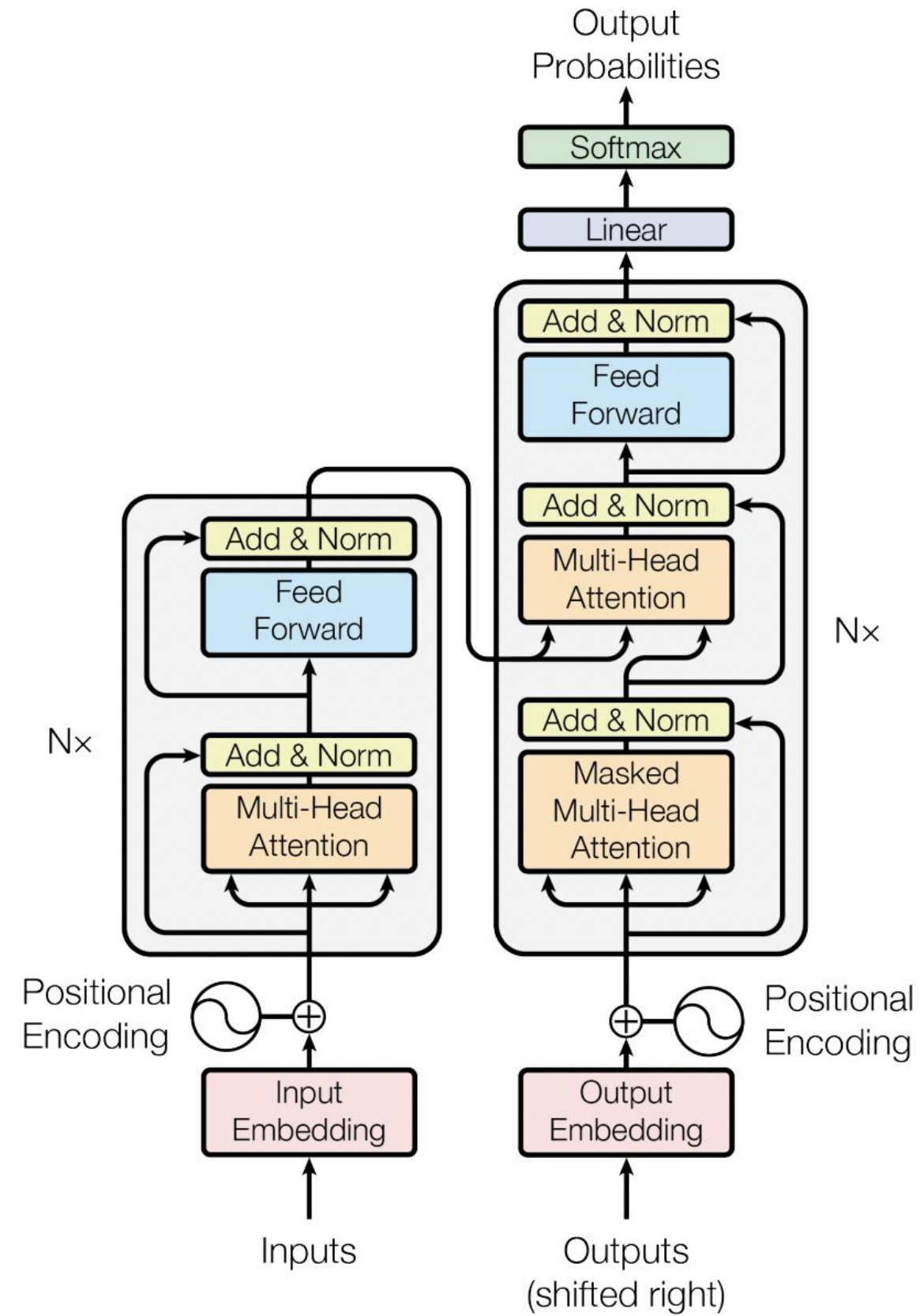


Self-attention

Here for example we can see that the word book is strongly connected with or paying more attention to the word teacher and the word student (Since respective edges are wider). This is called self attention which is the key attribute of the Transformer architecture.

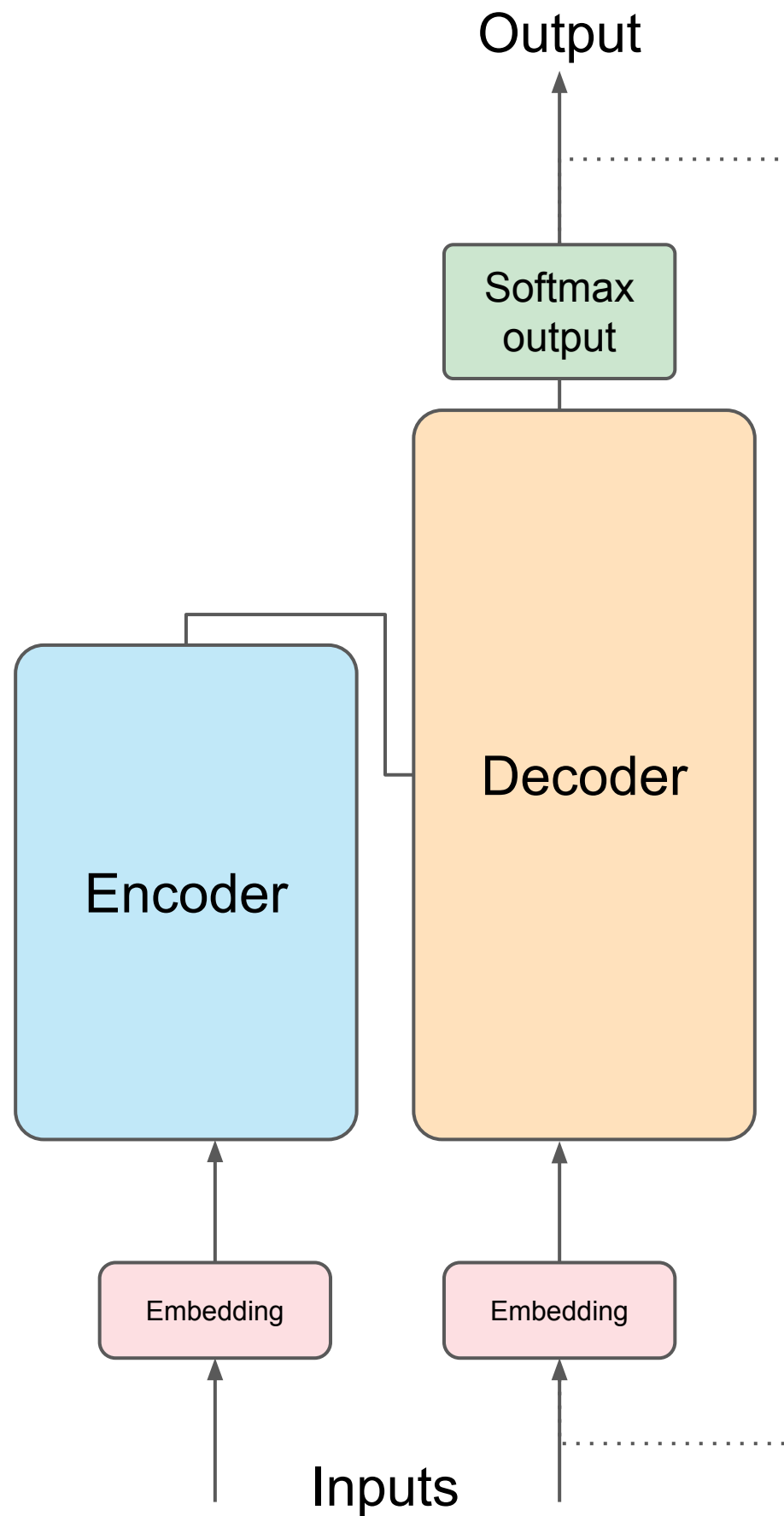


Transformers



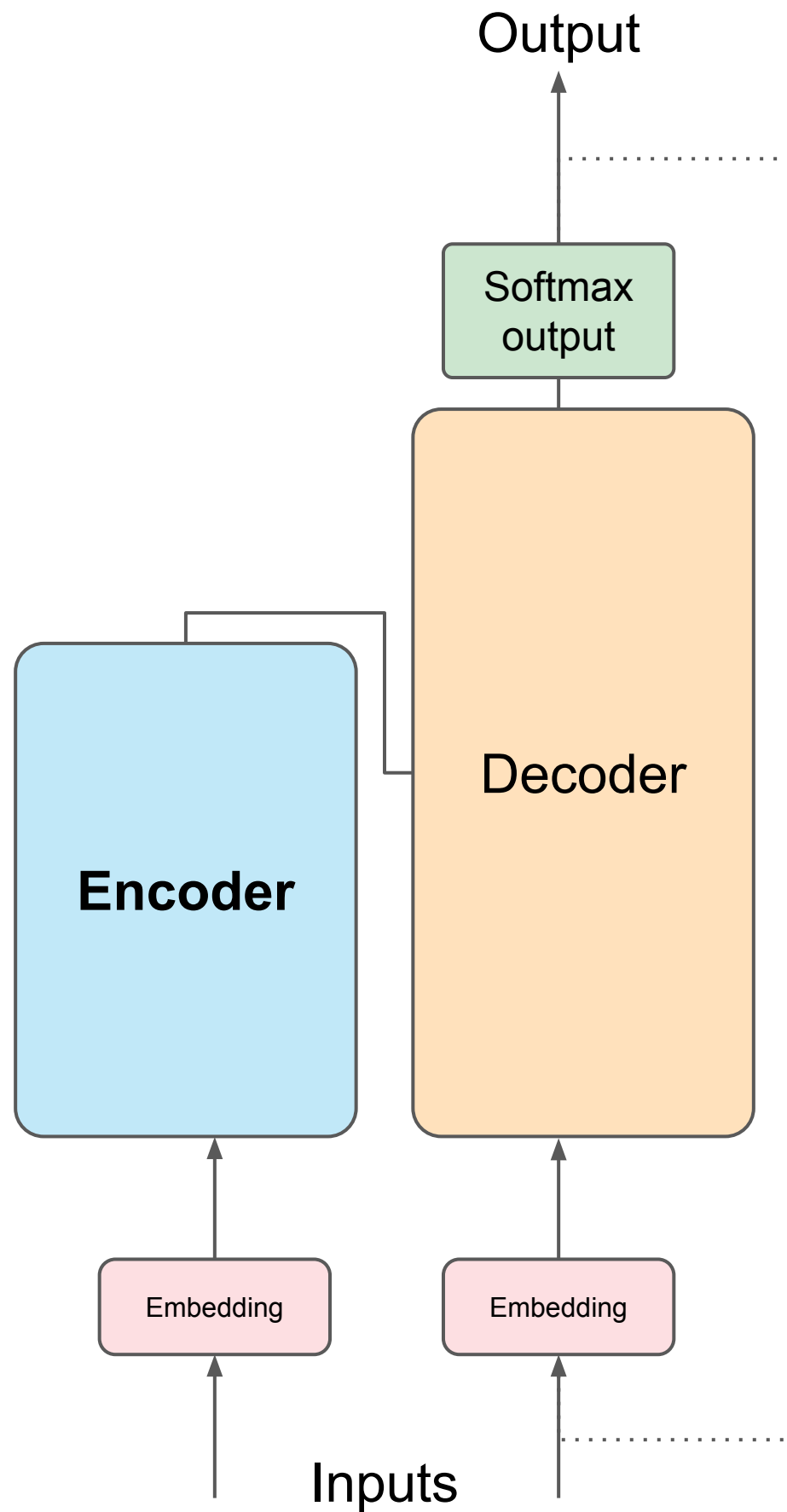
Transformers

Simplified high level architecture of Transformers

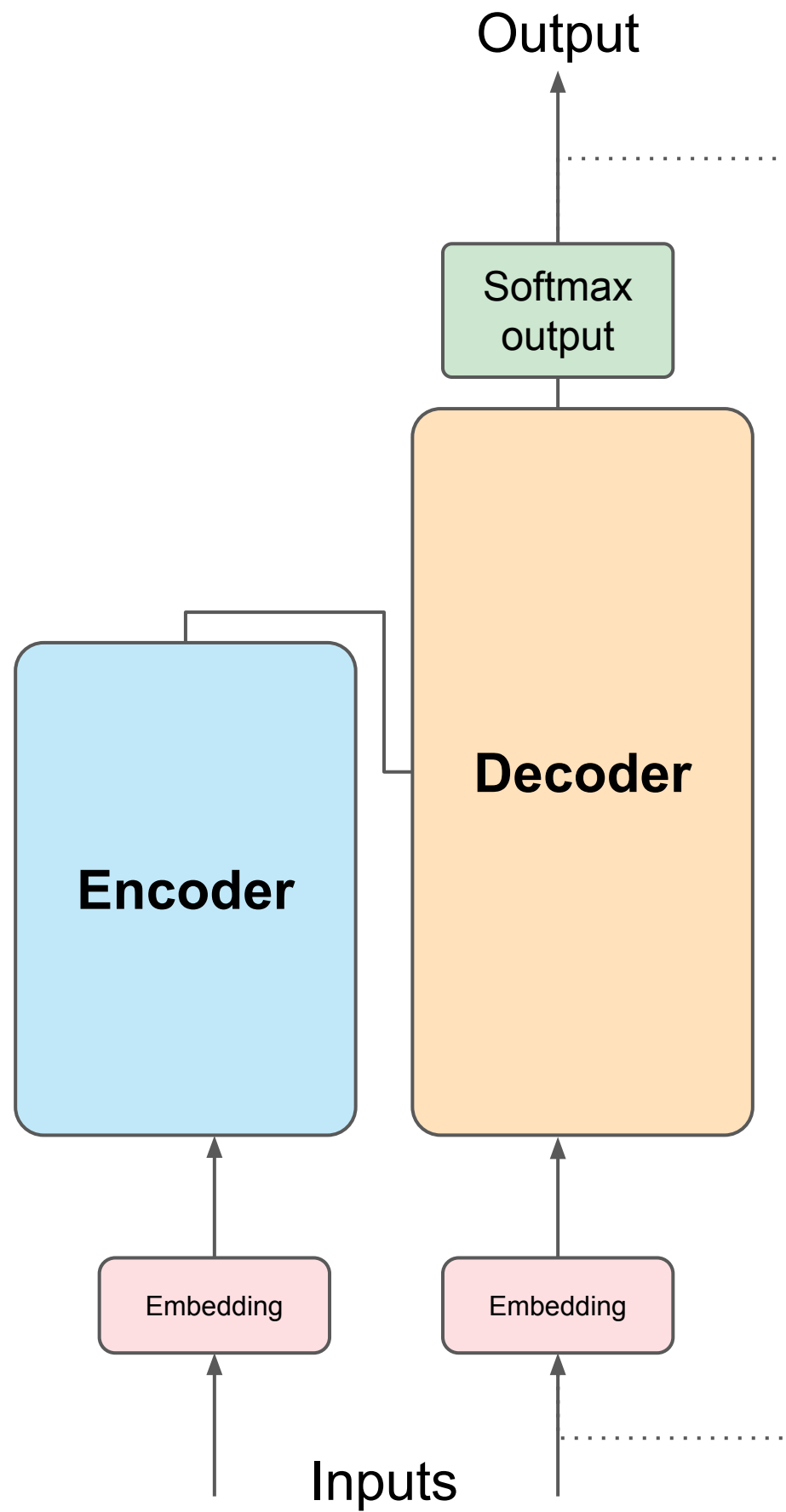


Here Encoder and Decoder work in conjunction with each other where, Input is represented at the bottom whereas Output at the top

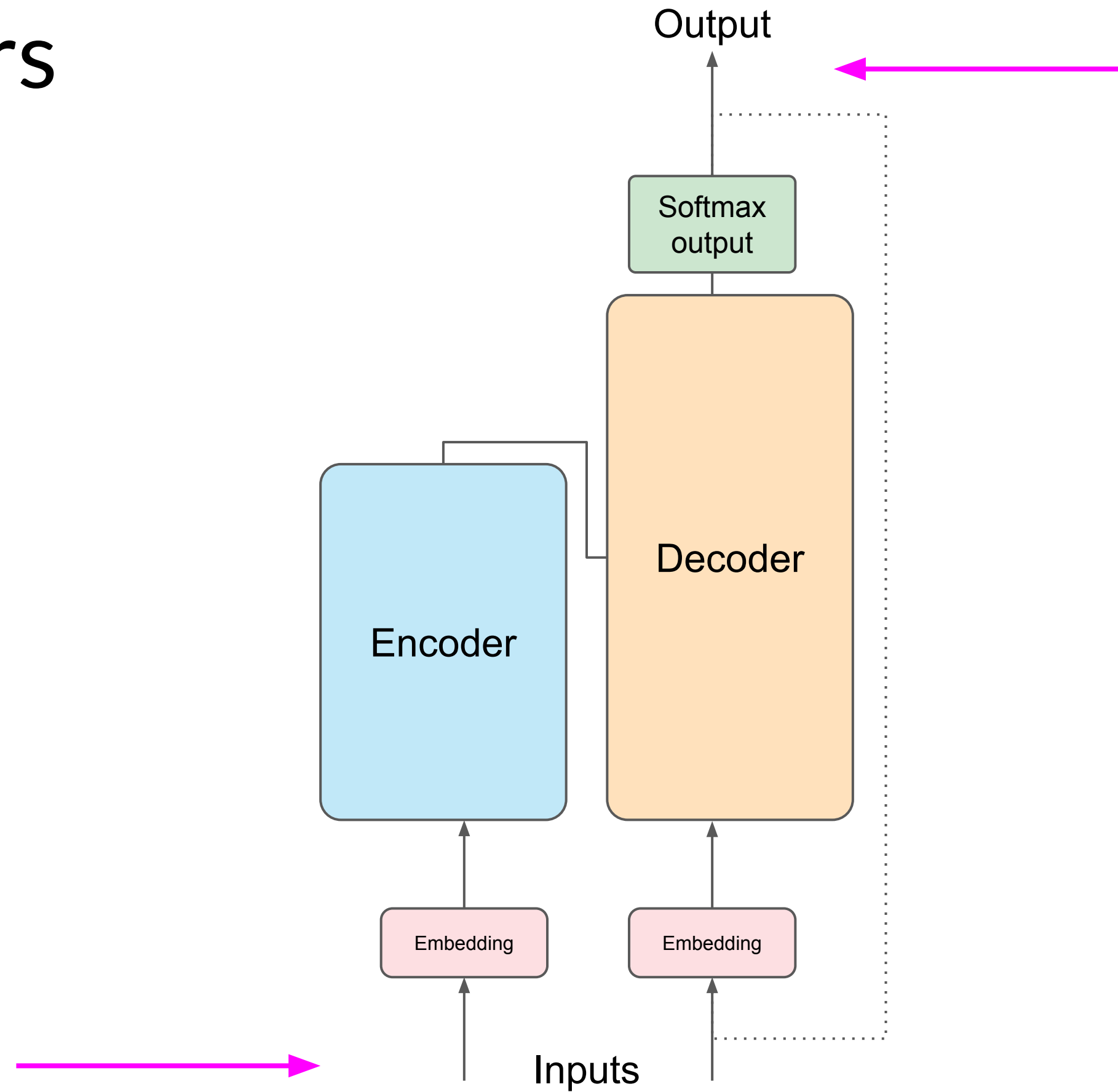
Transformers



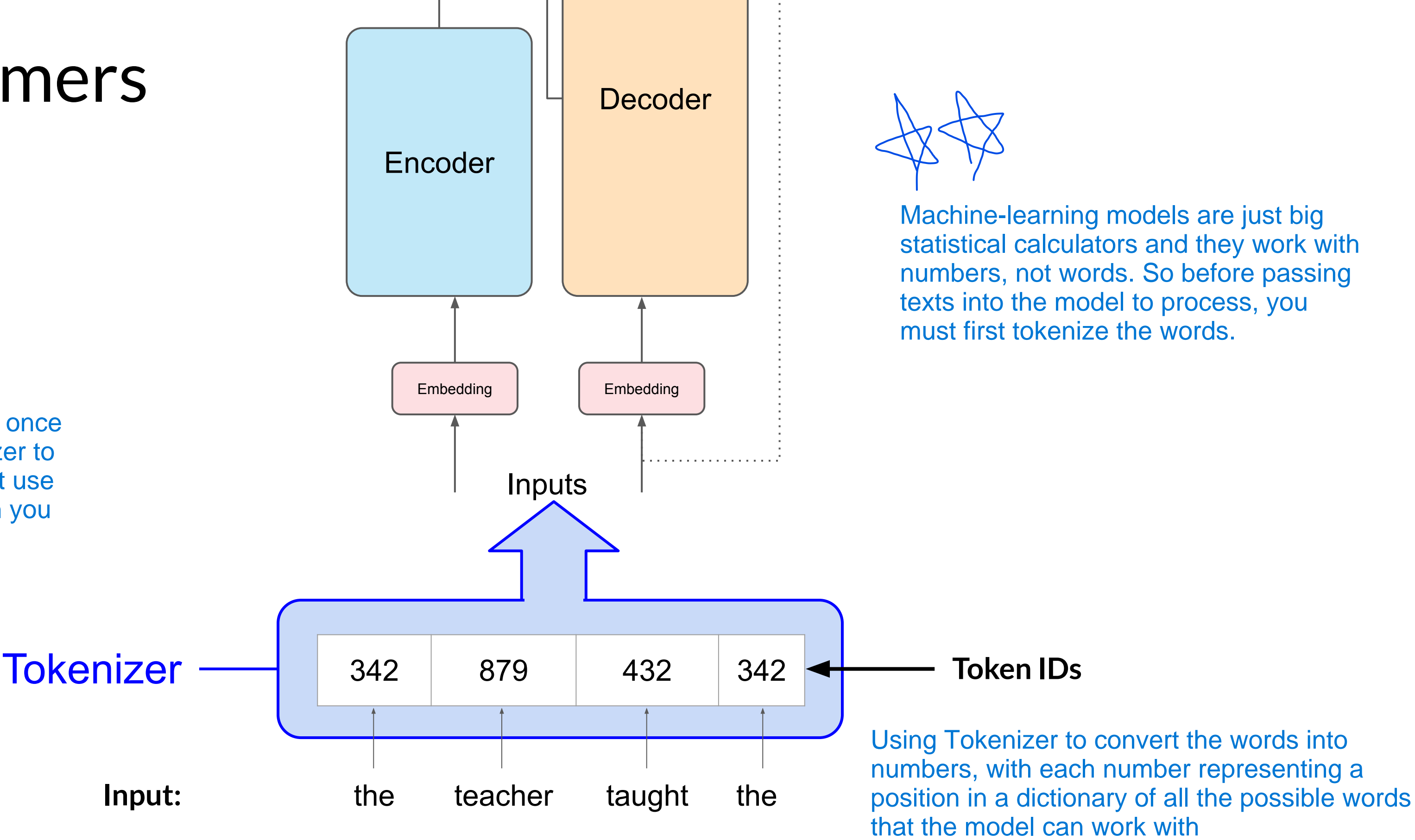
Transformers



Transformers

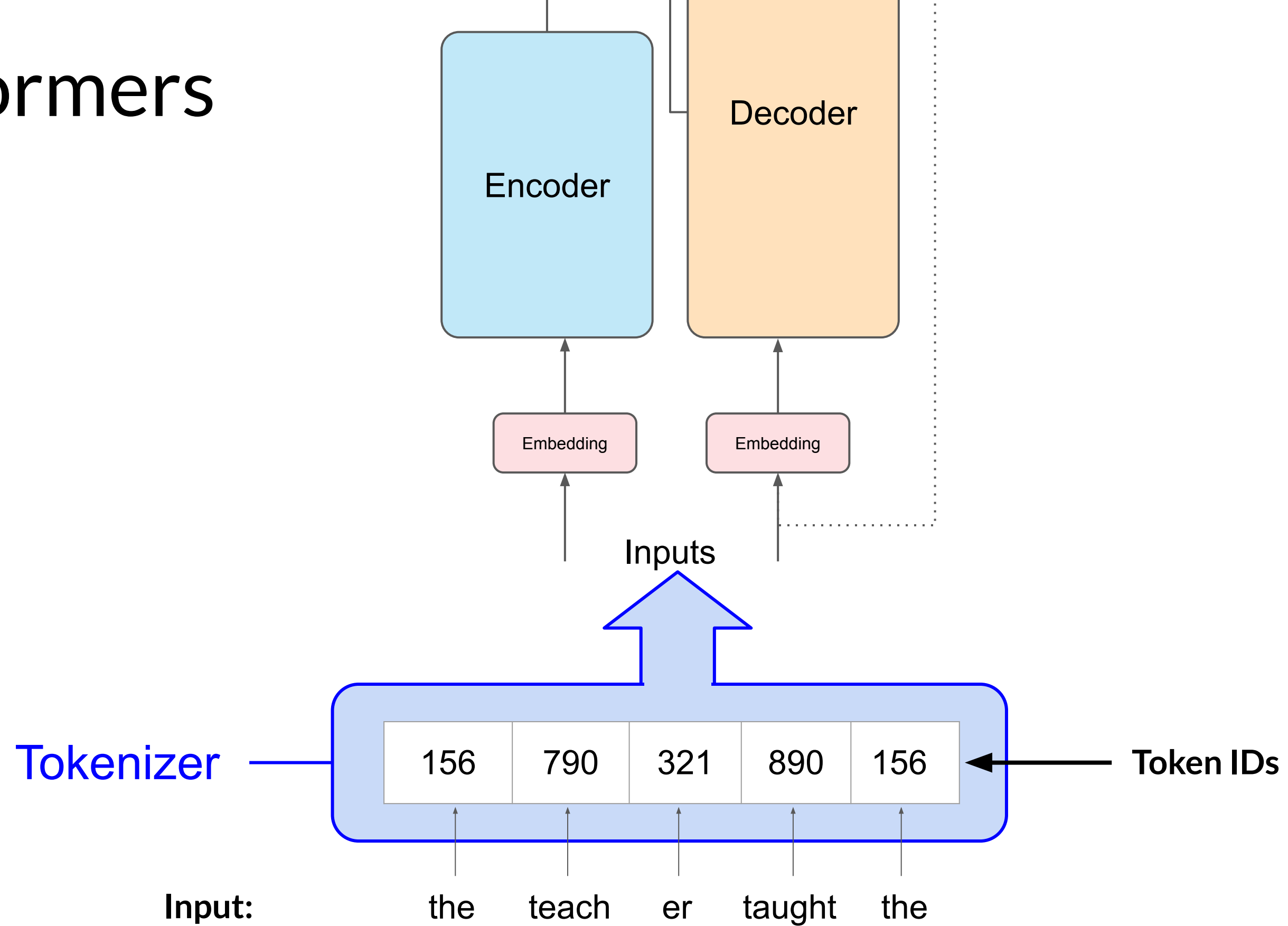


Transformers



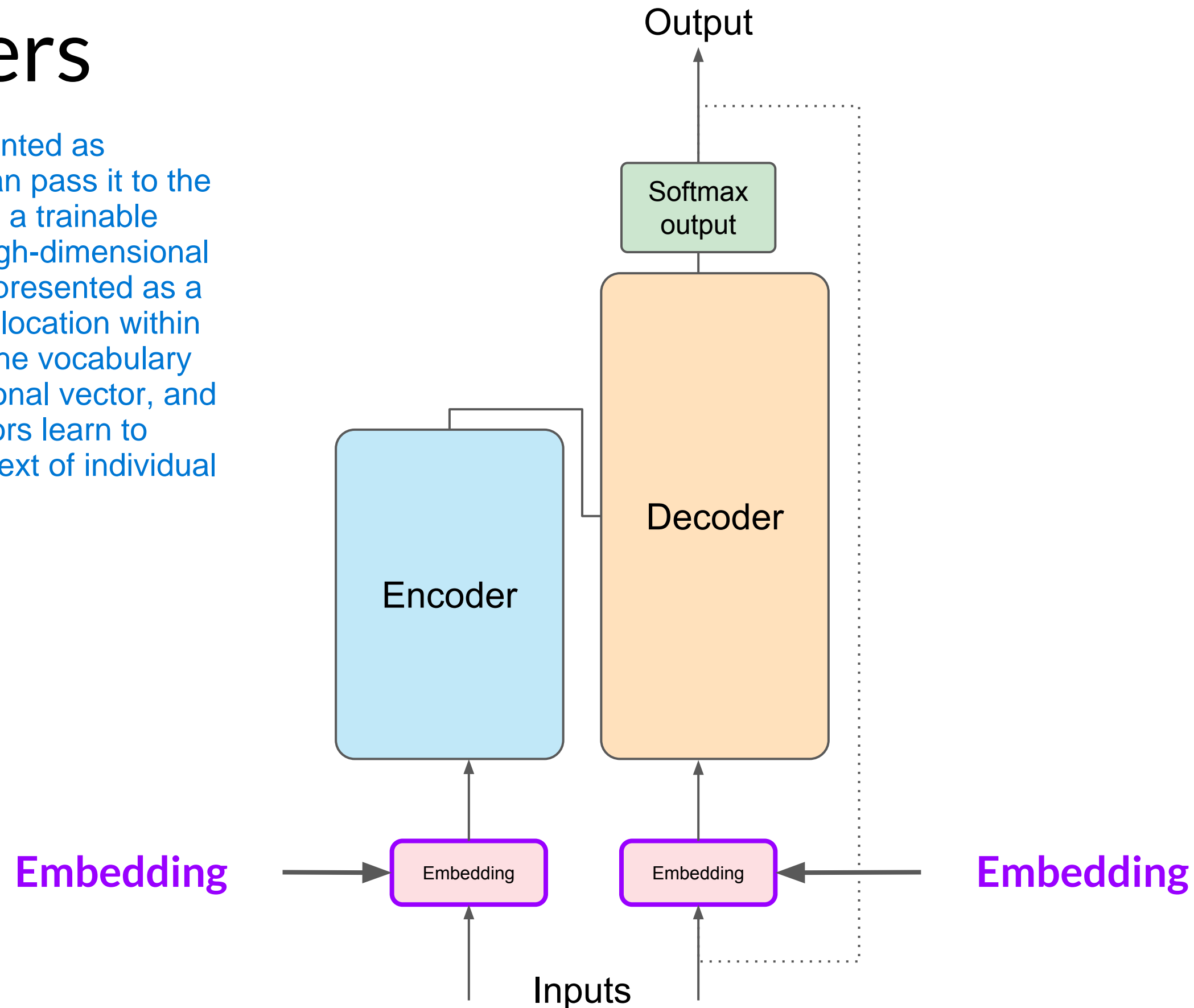
It's important to note that once you've selected a tokenizer to train the model, you must use the same tokenizer when you generate text.

Transformers



Transformers

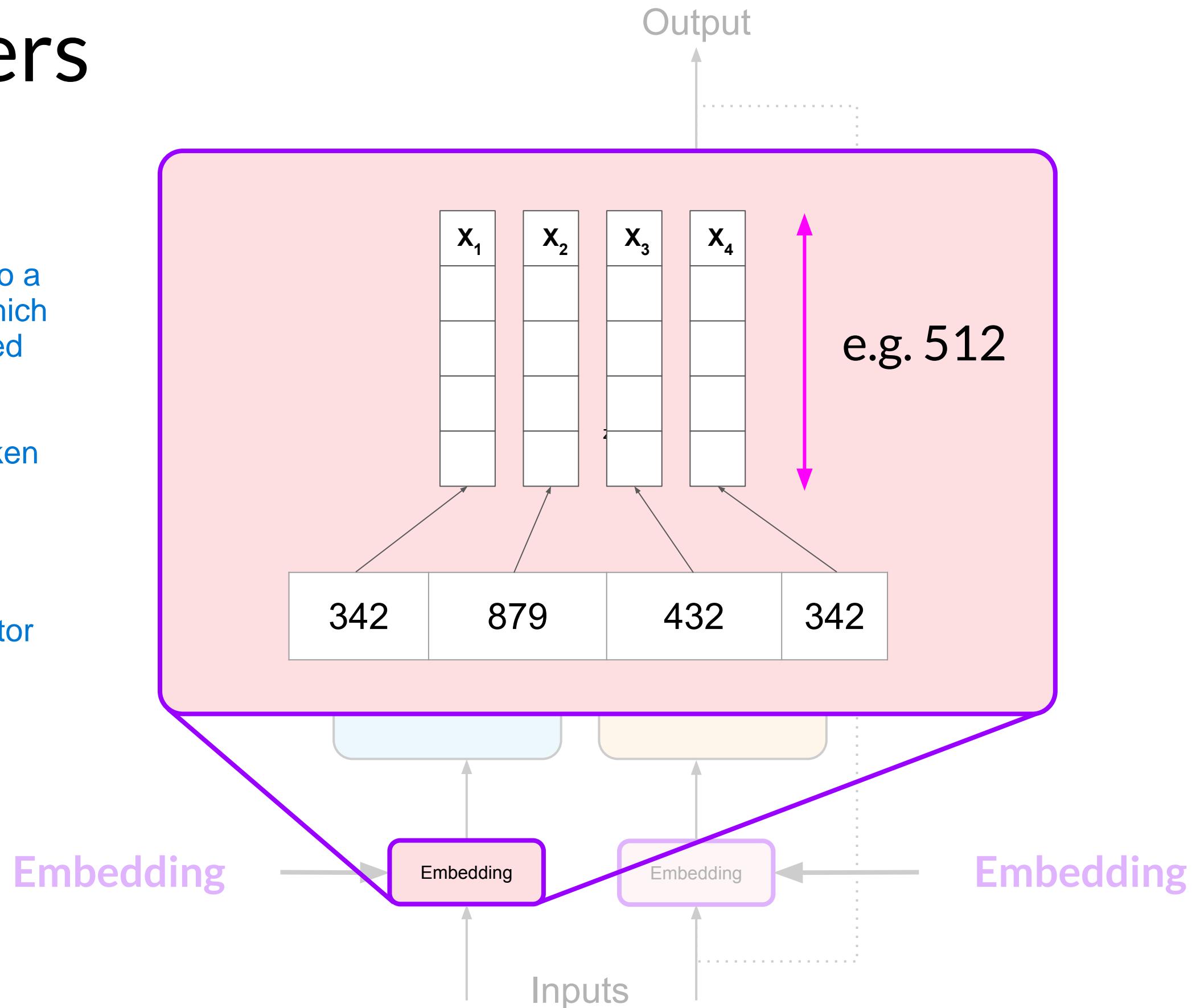
Now that your input is represented as numbers(tokenization), you can pass it to the embedding layer. This layer is a trainable vector embedding space, a high-dimensional space where each token is represented as a vector and occupies a unique location within that space. Each token ID in the vocabulary is matched to a multi-dimensional vector, and the intuition is that these vectors learn to encode the meaning and context of individual tokens in the input sequence.



Transformers

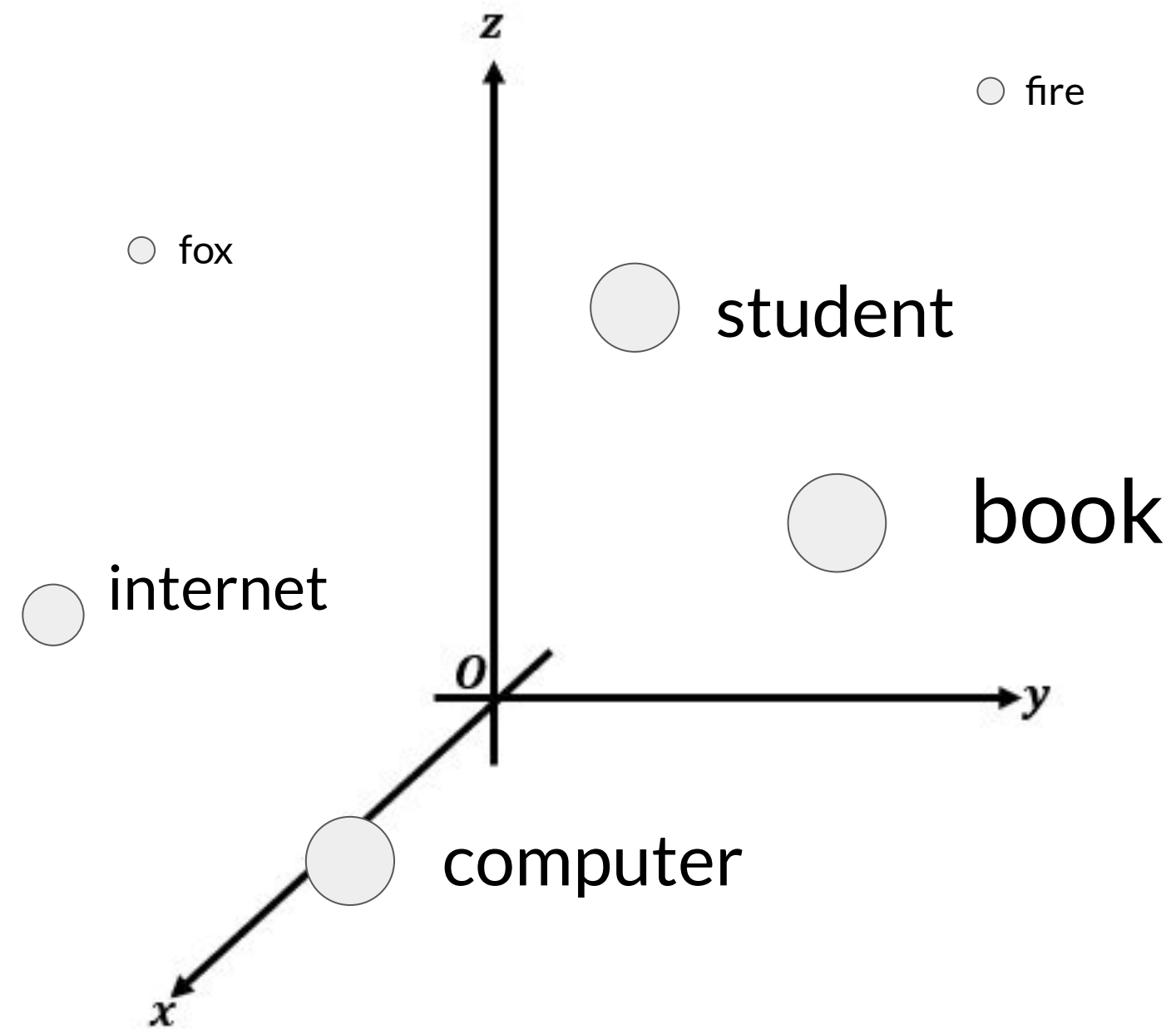
Each word is mapped to a token ID followed by which each token ID is mapped to a vector that will be storing the attention weight of respective token ID/word with all other token ID/word.

In the Transformer research paper the vector size was 512

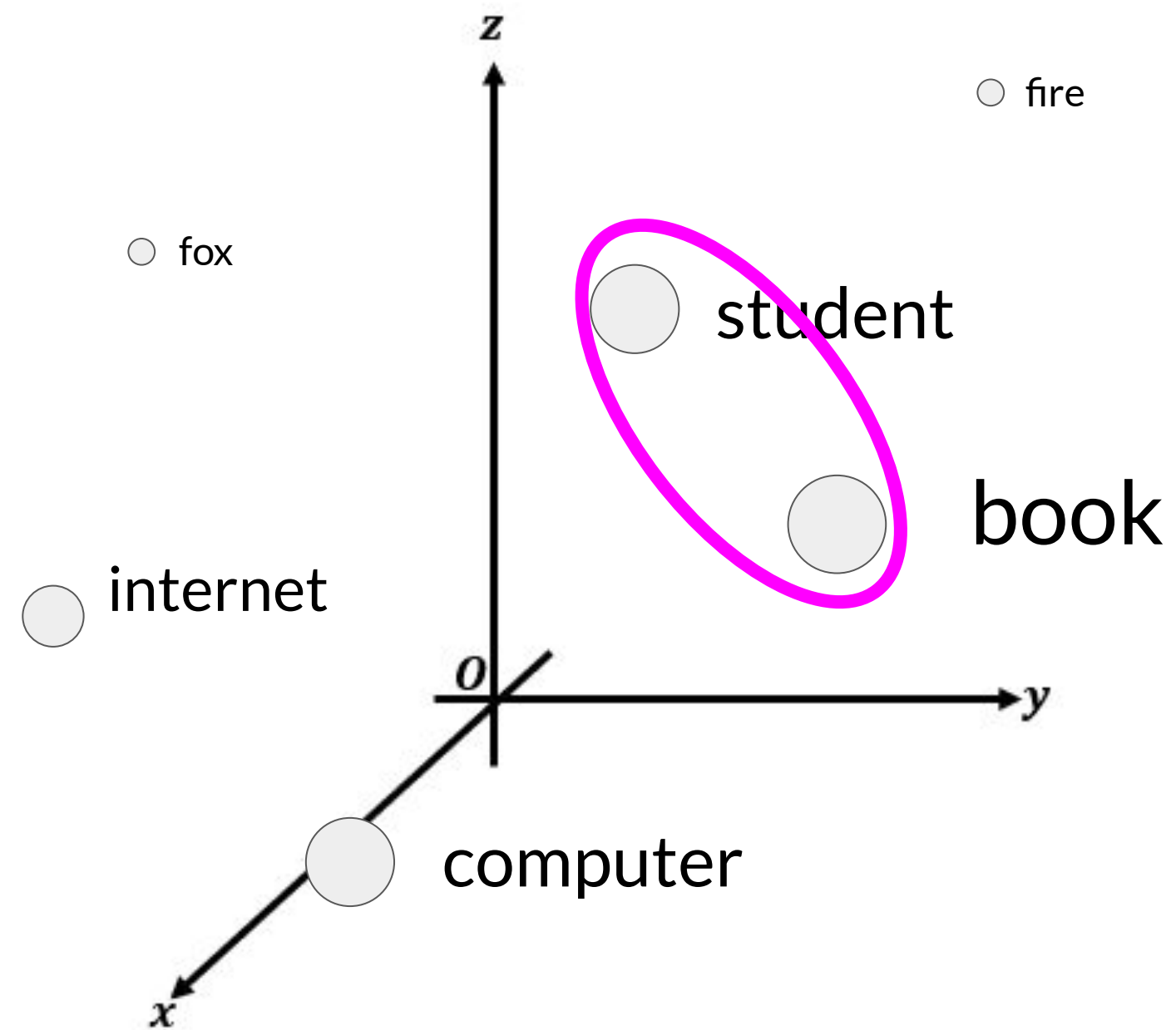


Transformers

For simplicity let's consider the vector of size 3. Now, we can plot the respective words on this 3 D space and observe the relationship b/w different words with each other.

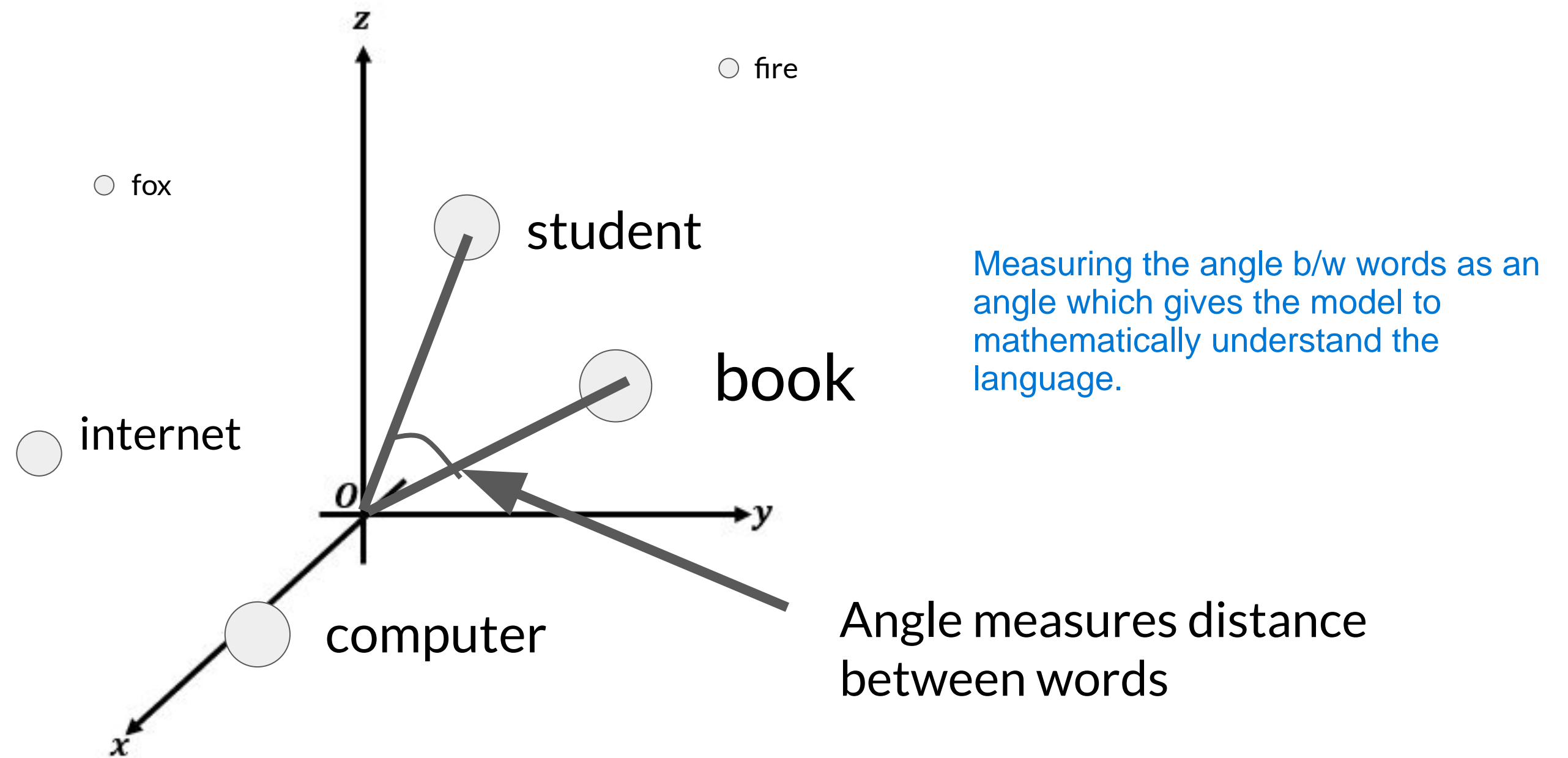


Transformers



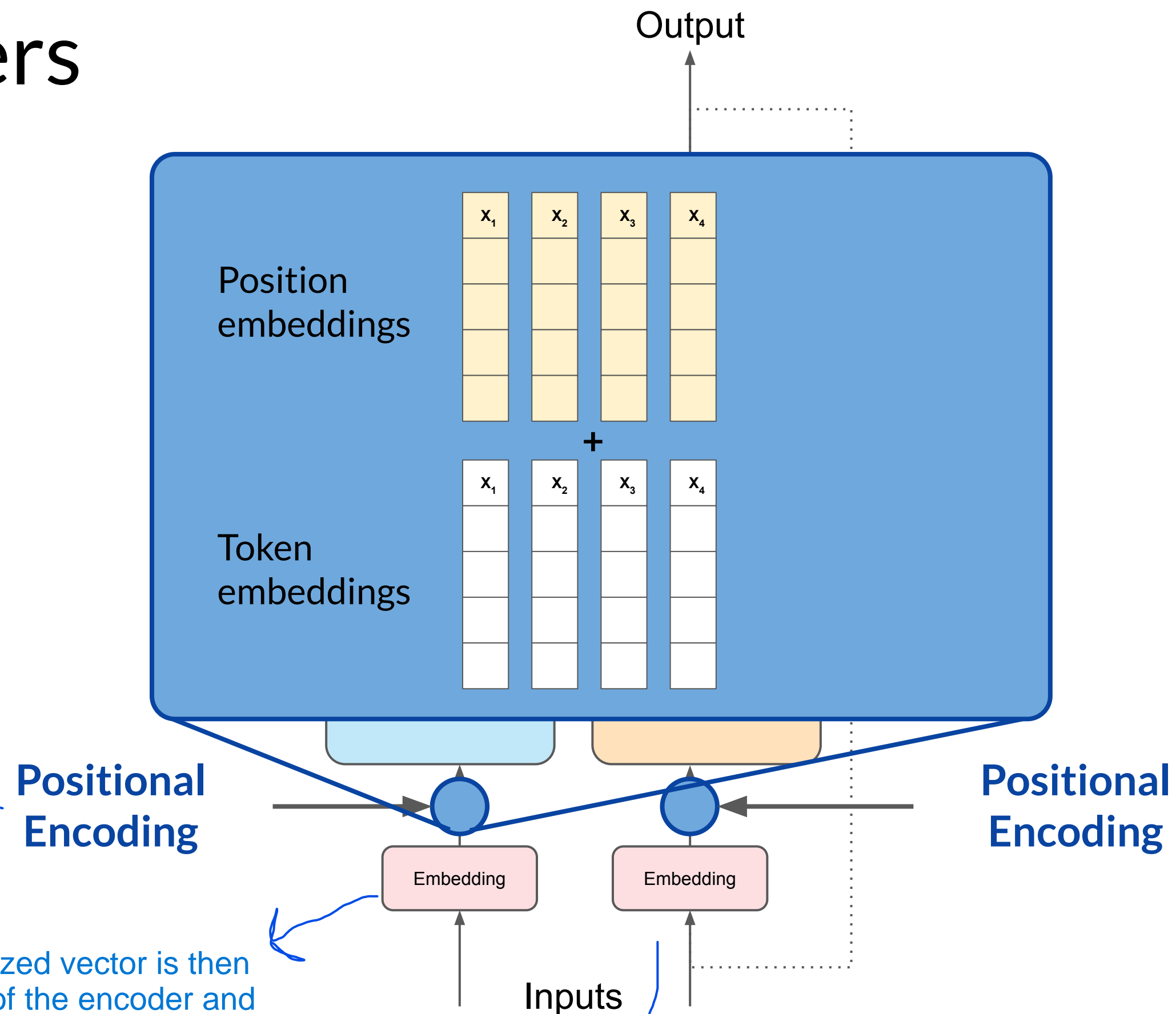
For example how closely words are located with each other in the vector embedding space

Transformers



Transformers

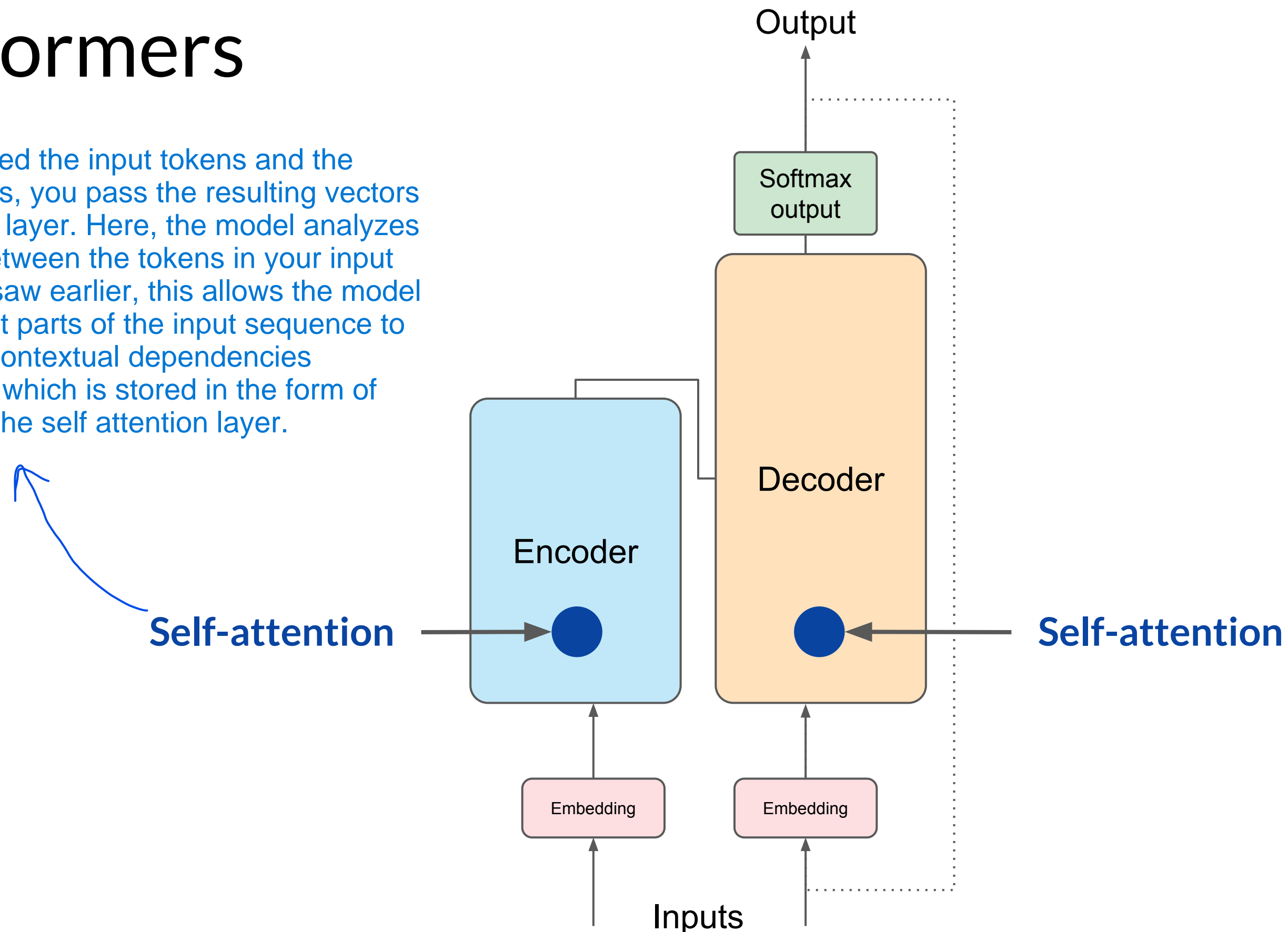
The model processes each of the input tokens in parallel. So by adding the positional encoding, you preserve the information about the word order and don't lose the relevance of the position of the word in the sentence.



The resultant tokenized vector is then placed at the base of the encoder and decoder

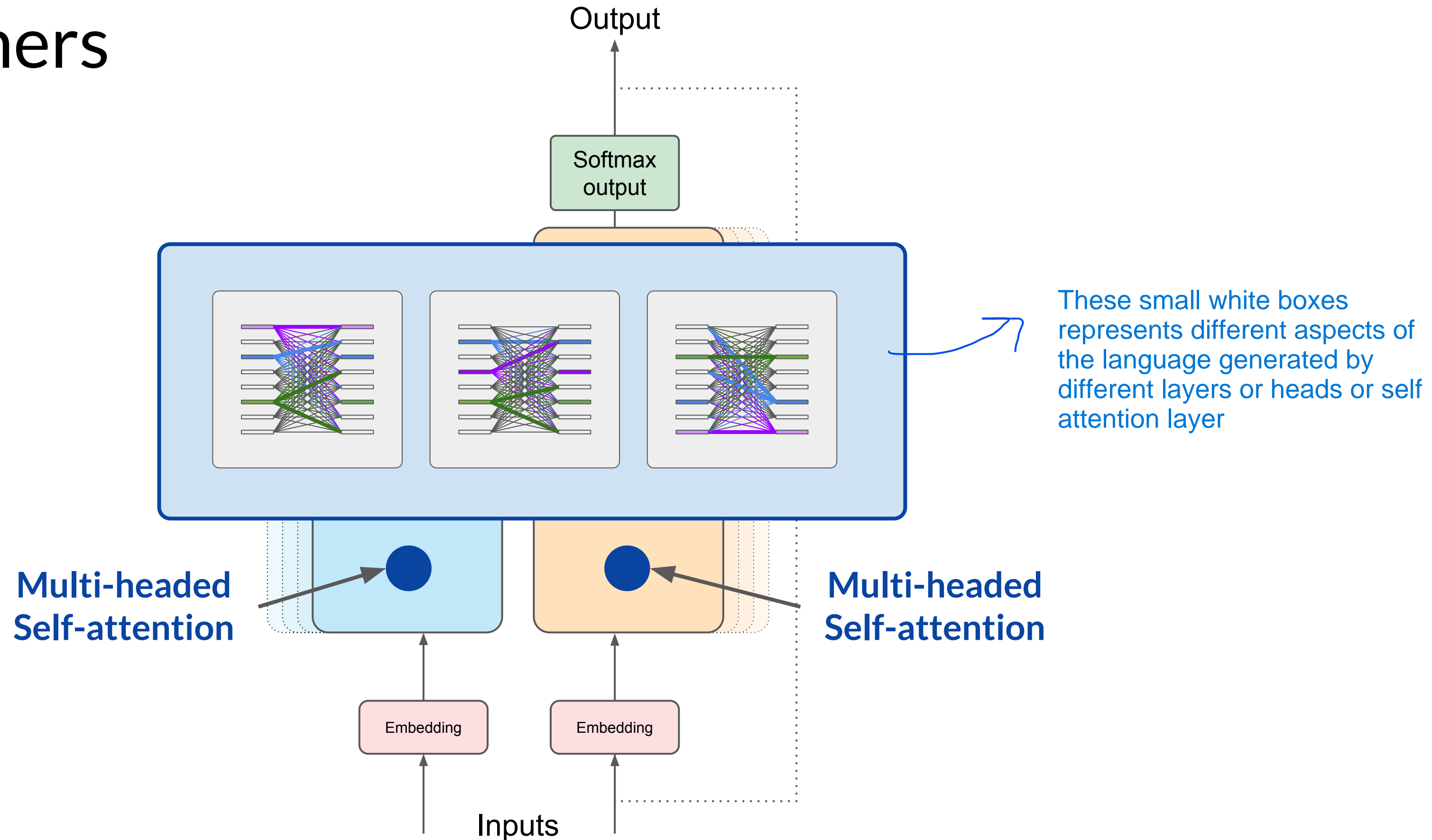
Transformers

Once you've summed the input tokens and the positional encodings, you pass the resulting vectors to the self-attention layer. Here, the model analyzes the relationships between the tokens in your input sequence. As you saw earlier, this allows the model to attend to different parts of the input sequence to better capture the contextual dependencies between the words which is stored in the form of attention weight in the self attention layer.



Transformers

The transformer architecture actually has multi-headed self-attention. This means that multiple sets of self-attention weights or heads are learned in parallel independently of each other. The number of attention heads included in the attention layer varies from model to model, but numbers in the range of 12-100 are common. The intuition here is that each self-attention head will learn a different aspect of language. For example, one head may see the relationship between the people entities in our sentence. Whilst another head may focus on the activity of the sentence. Whilst yet another head may focus on some other properties such as if the words rhyme. It's important to note that you don't dictate ahead of time what aspects of language the attention heads will learn. The weights of each head are randomly initialized and given sufficient training data and time, each will learn different aspects of language.



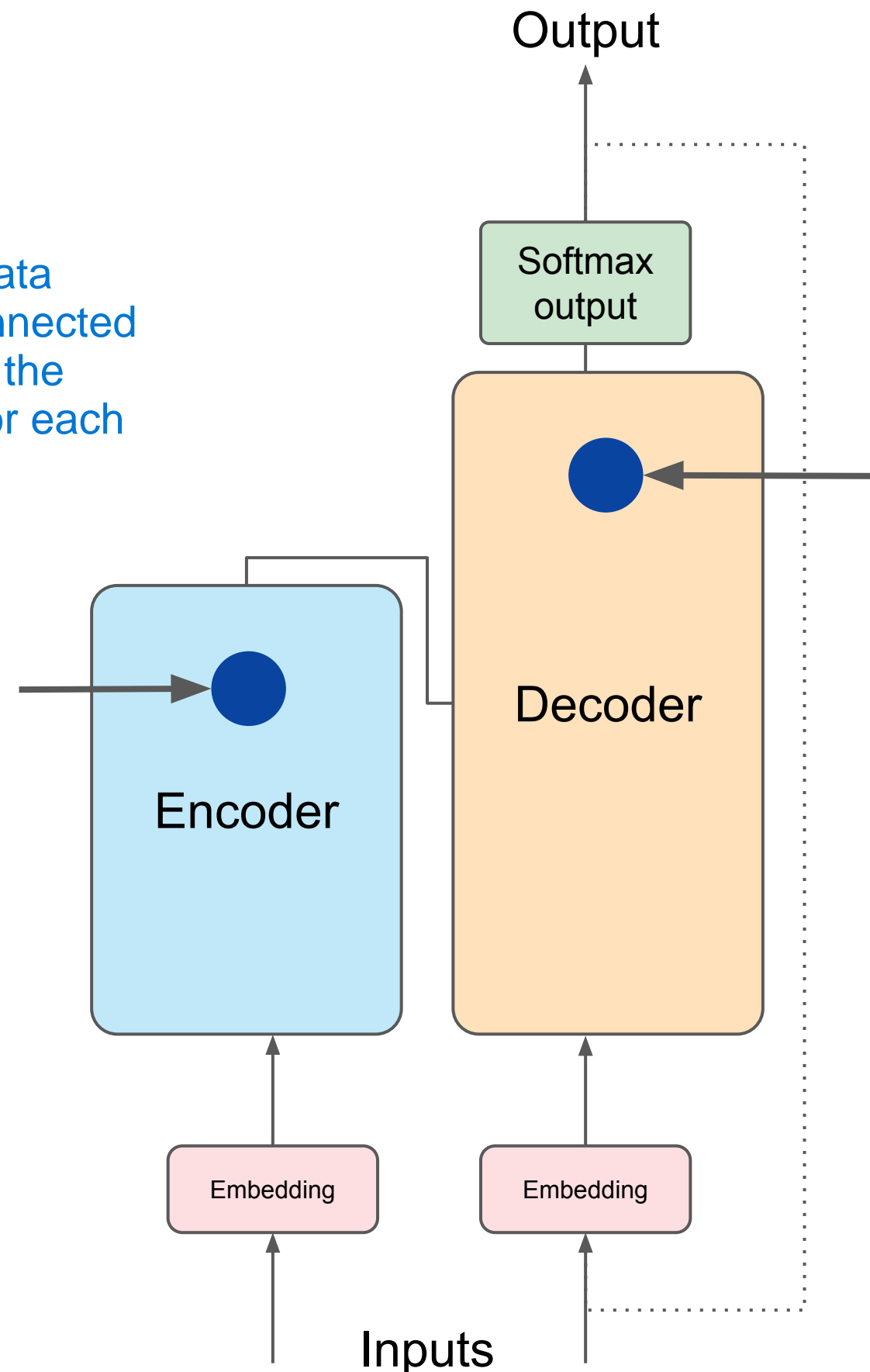
Transformers

Once attention weights are applied to the Input data then the output is processed through the fully connected Feed Forward network whose output represents the vector of logits proportional to probability score for each and every token in the tokenizer dictionary.

What Are Logits?
In machine learning, the term “logits” refers to the raw outputs of a model before they are transformed into probabilities. Specifically, logits are the unnormalized outputs of the last layer of a neural network.

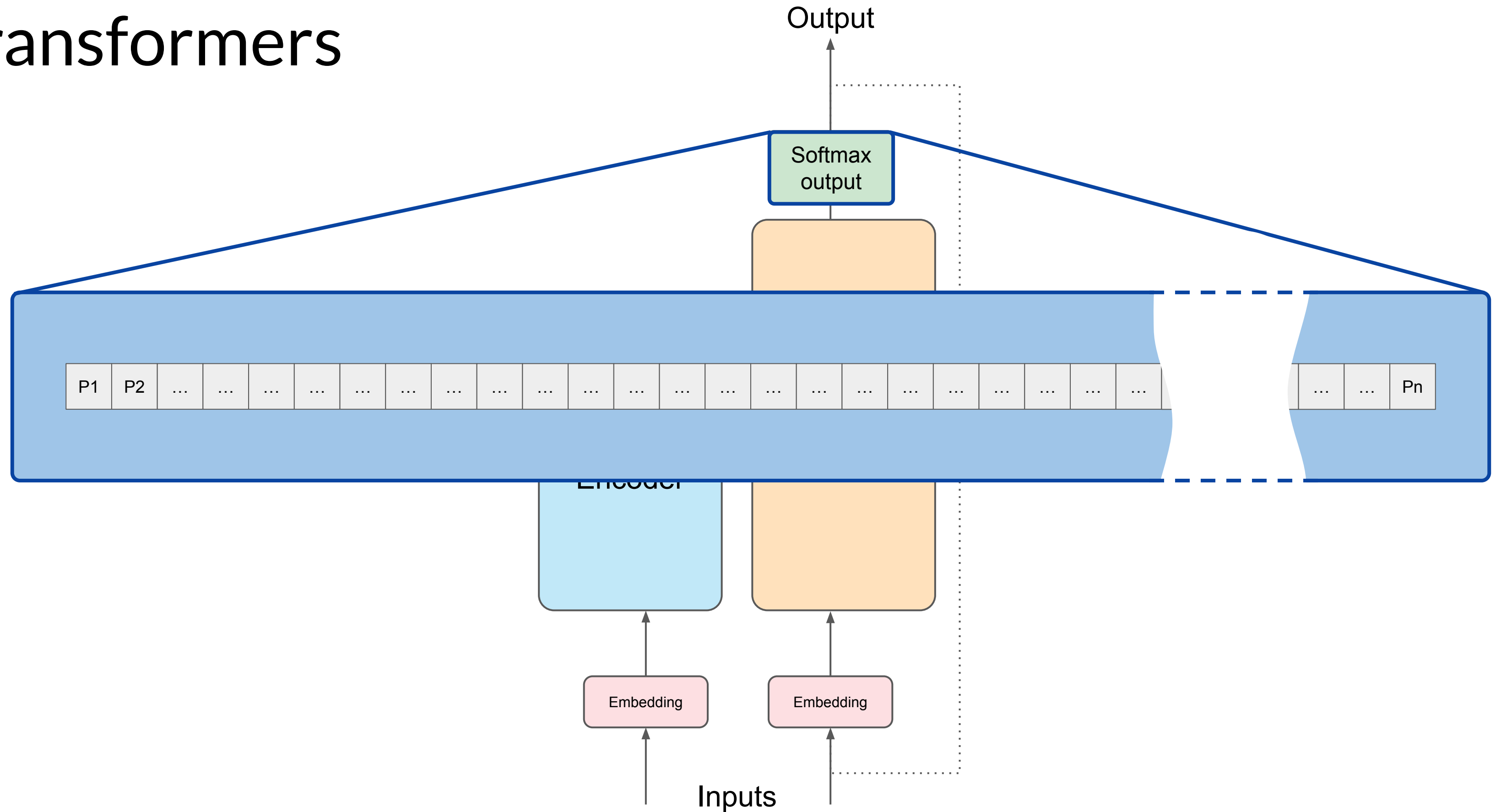
Feed forward network

Feed forward network



At last the unnormalized probability score is passed to SoftMax function that normalizes them into probability score for each word. This output includes probability for every single word in the vocabulary.

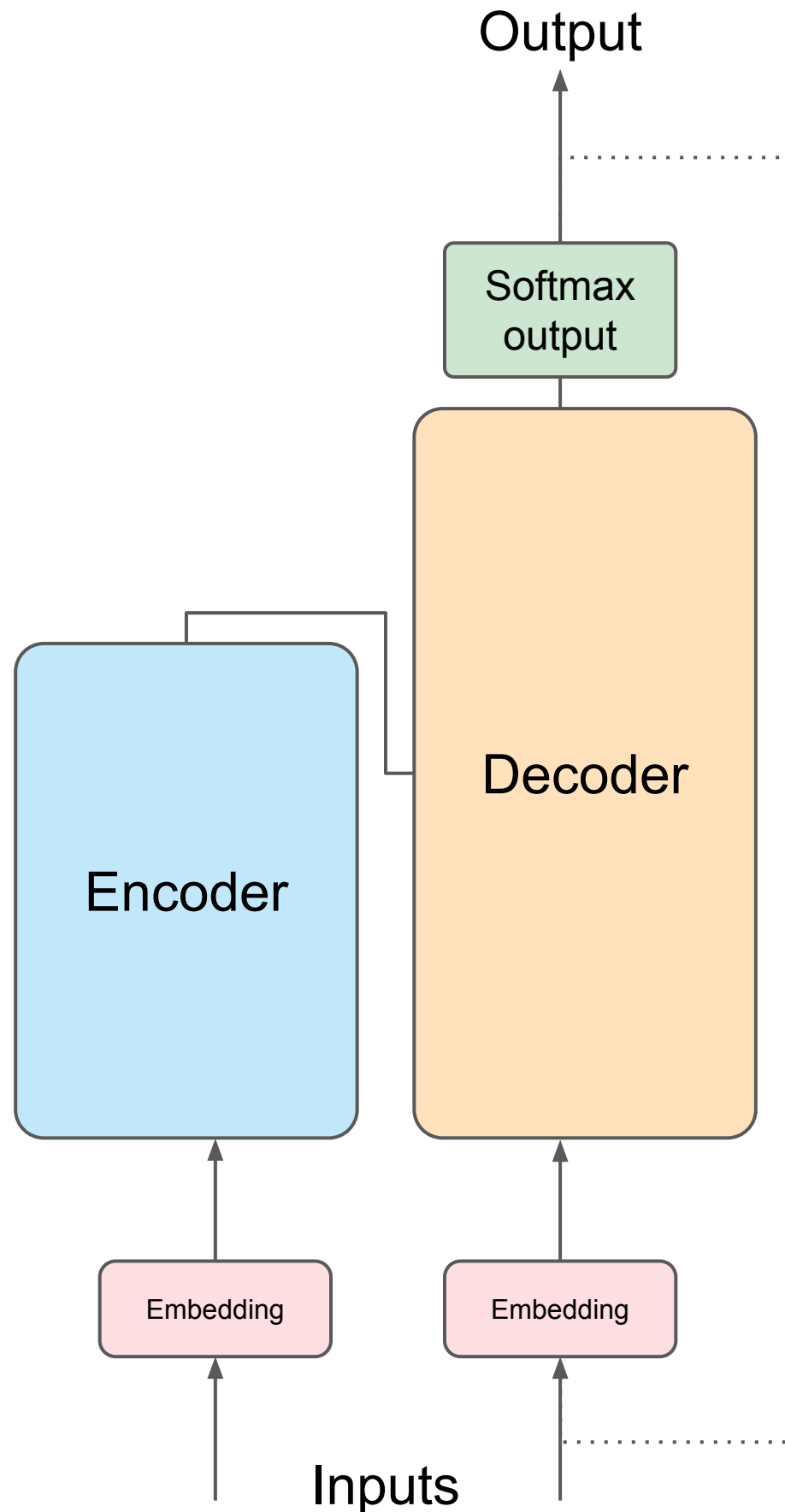
Transformers



Transformers

Translation:
sequence-to-sequence task

J'aime l'apprentissage
automatique

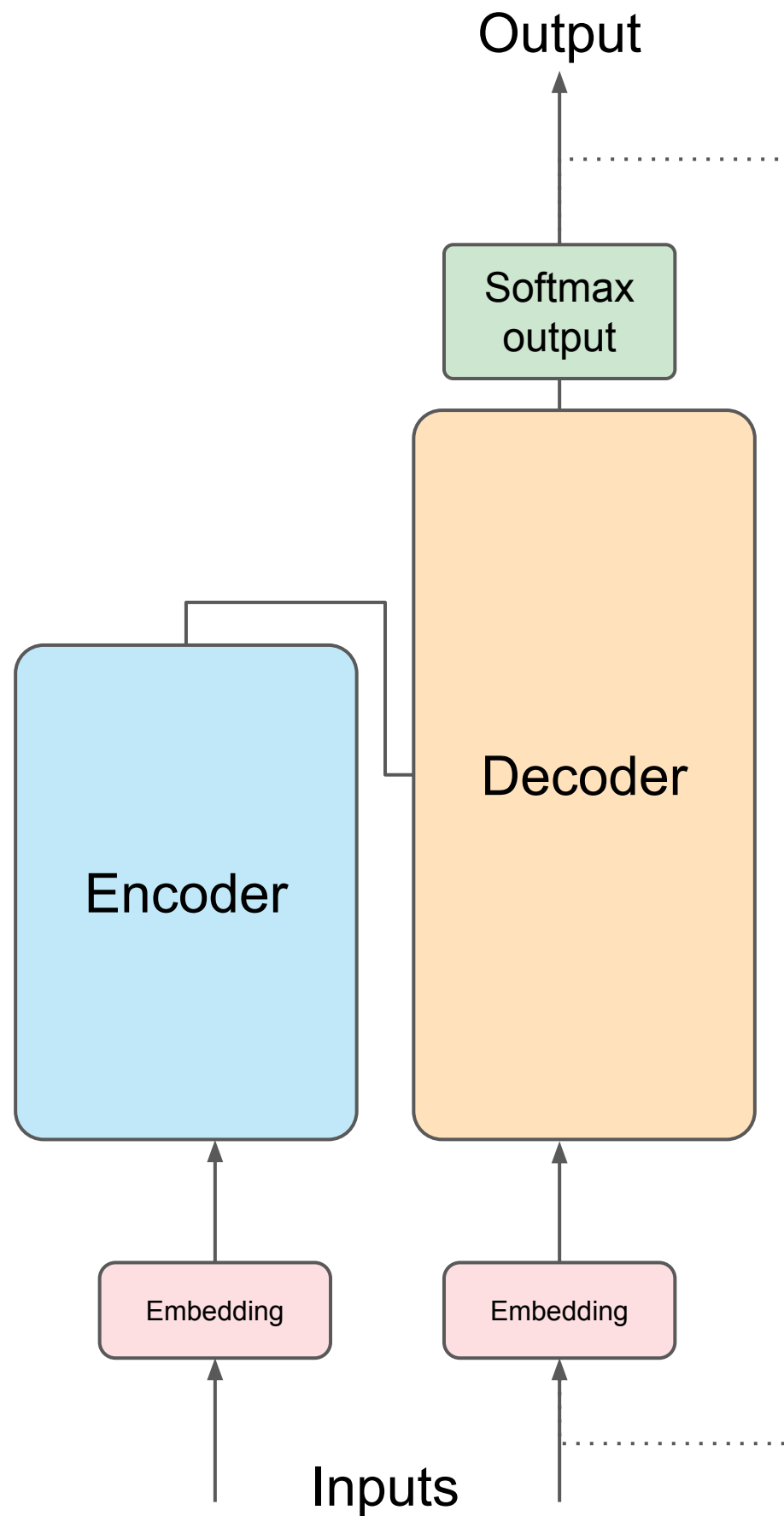
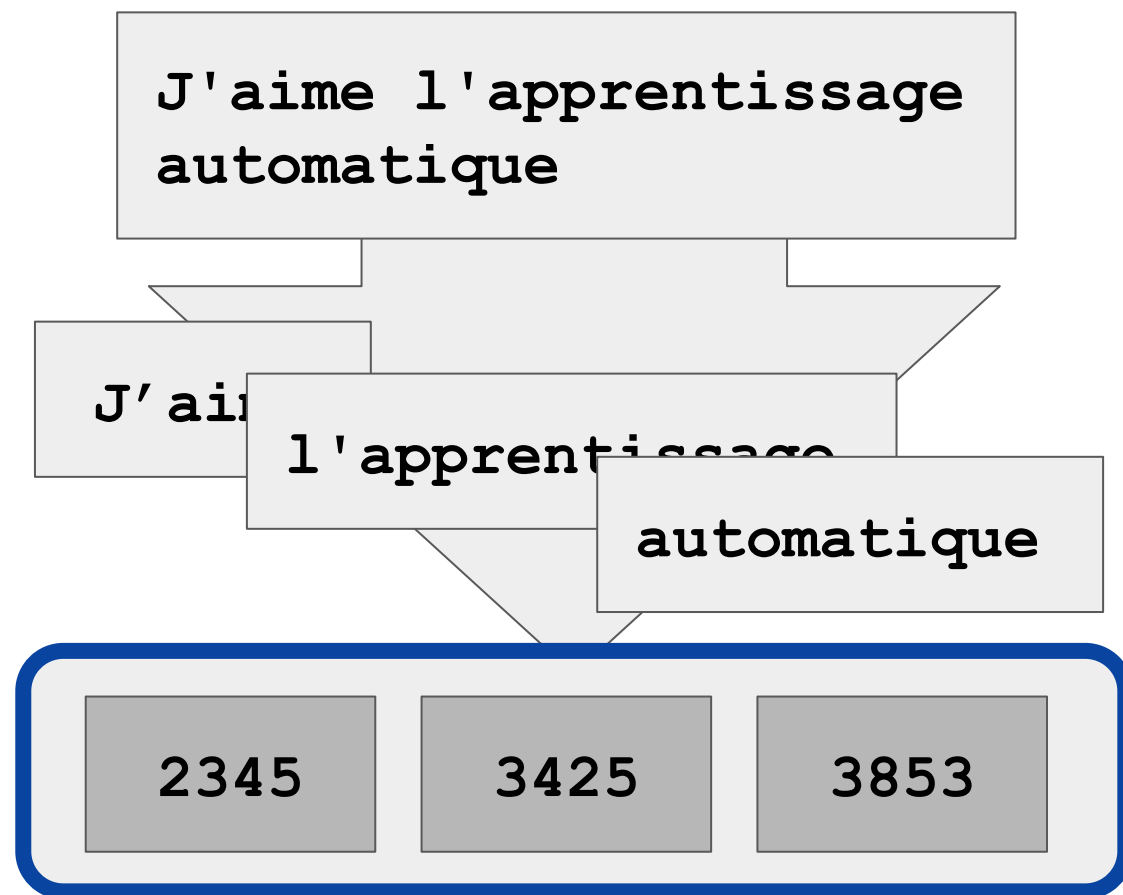


Until now we have seen how major components of transformer looks like, Now we will try to see how the overall prediction process works end to end.

let's understand this with an example where we will be translating Input in French language into English language

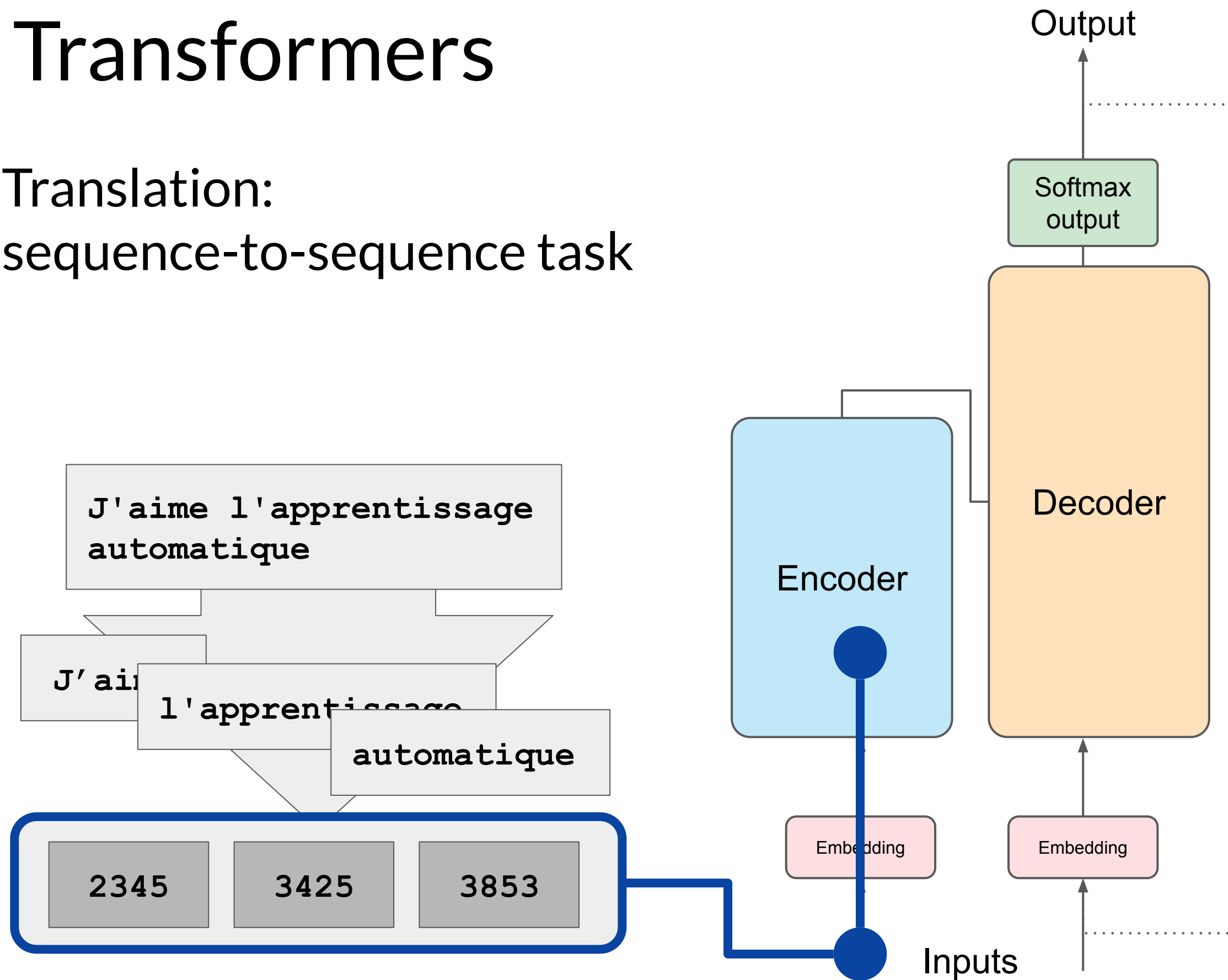
Transformers

Translation:
sequence-to-sequence task



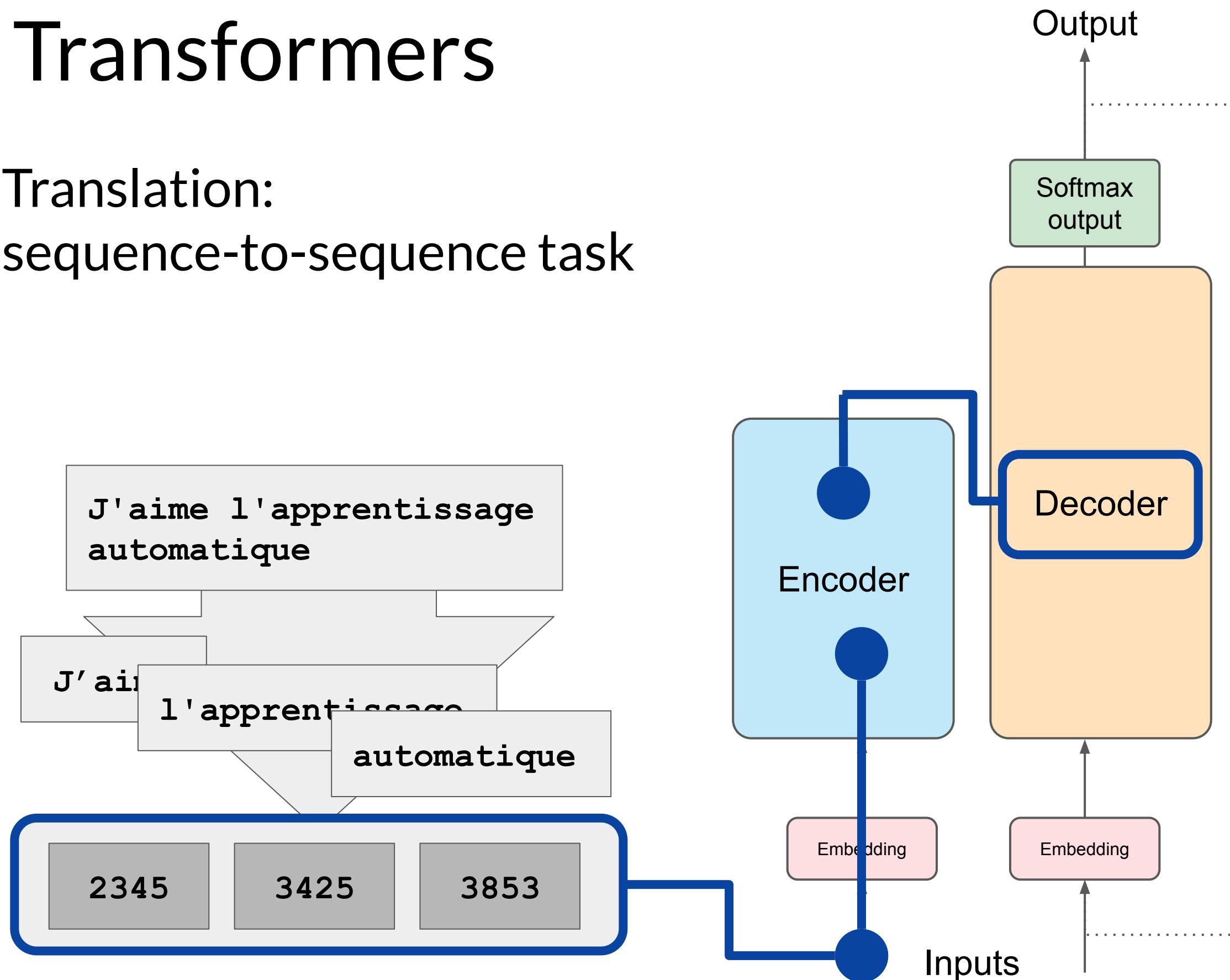
Transformers

Translation:
sequence-to-sequence task



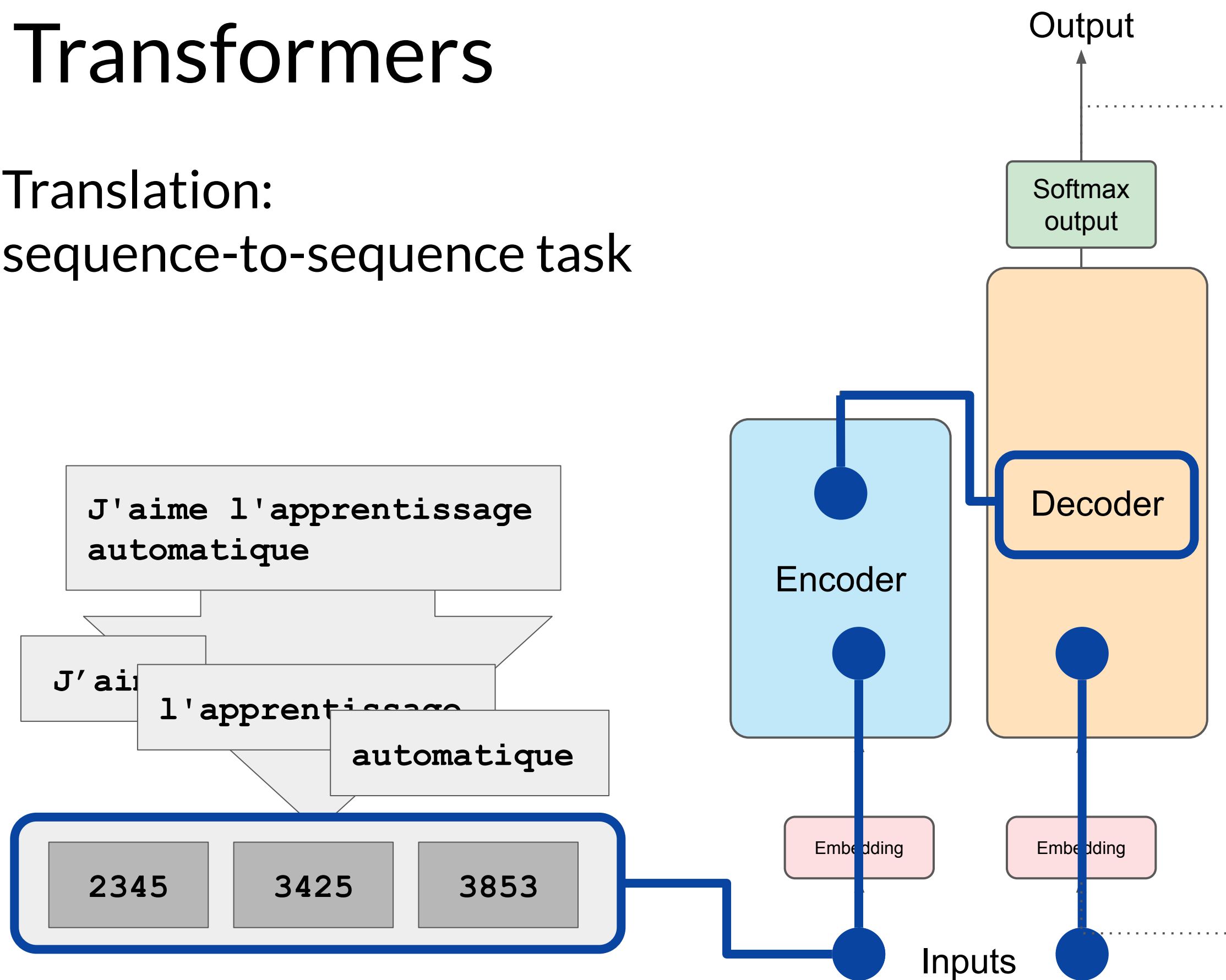
Transformers

Translation:
sequence-to-sequence task



Transformers

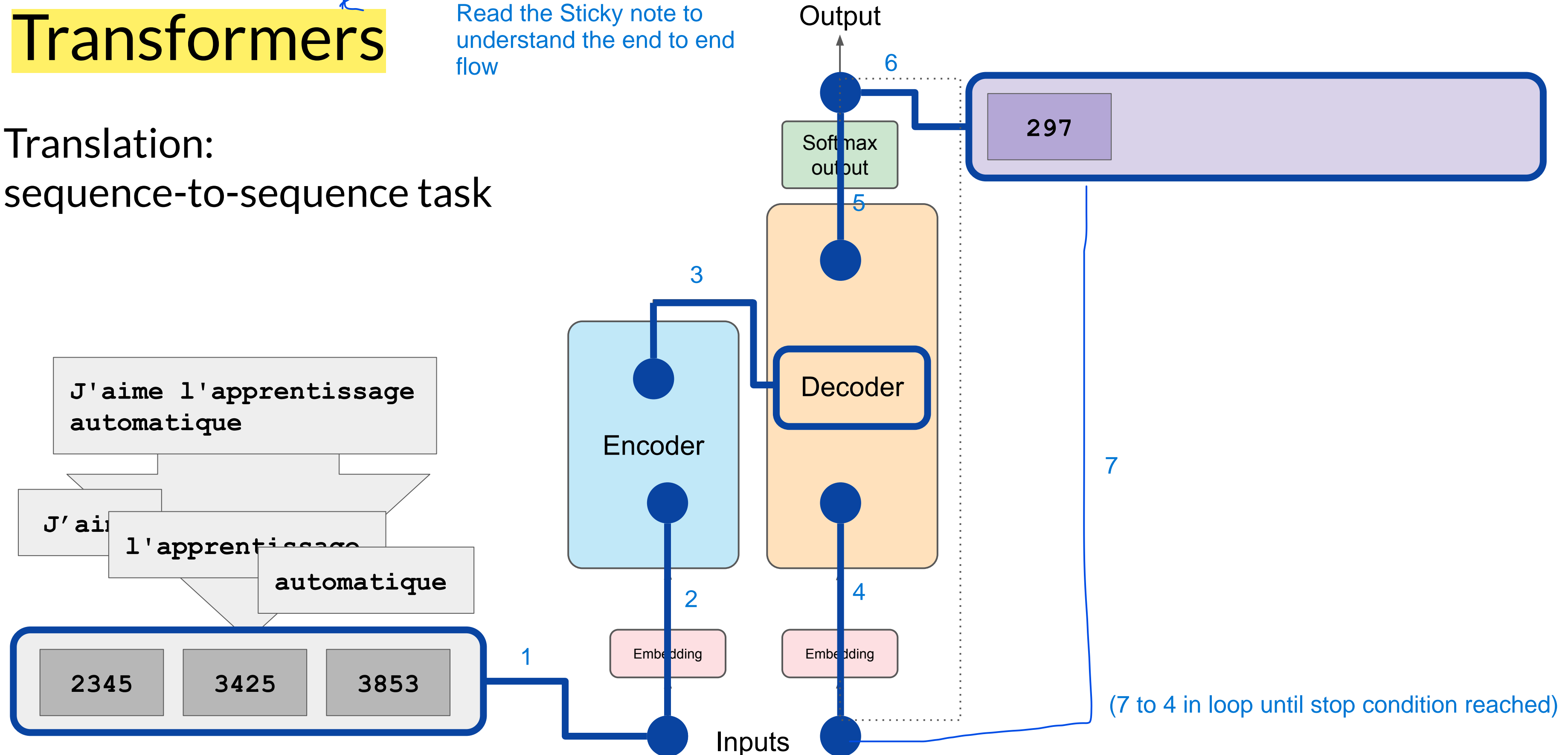
Translation:
sequence-to-sequence task



Transformers

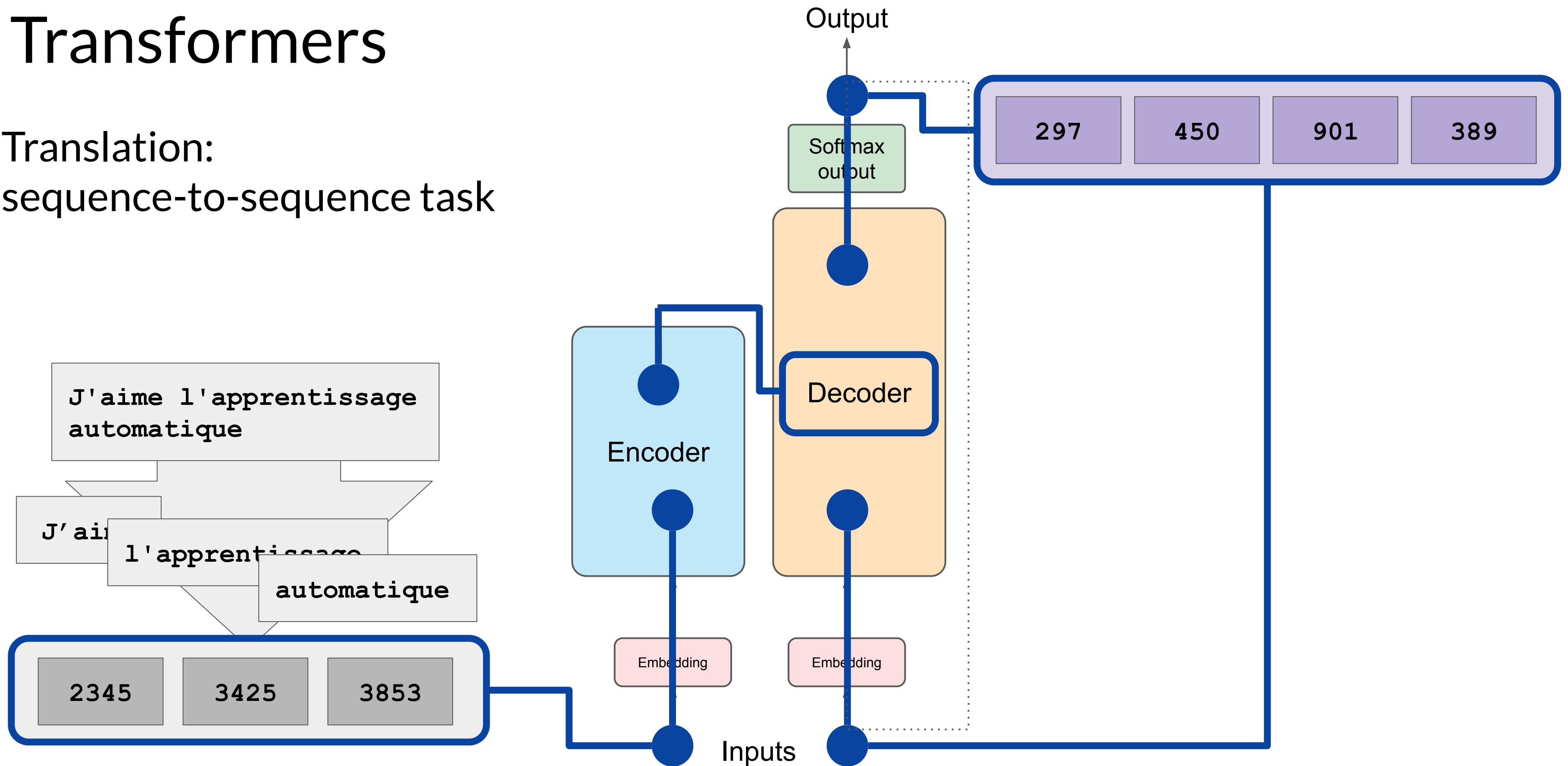
Read the Sticky note to understand the end to end flow

Translation:
sequence-to-sequence task



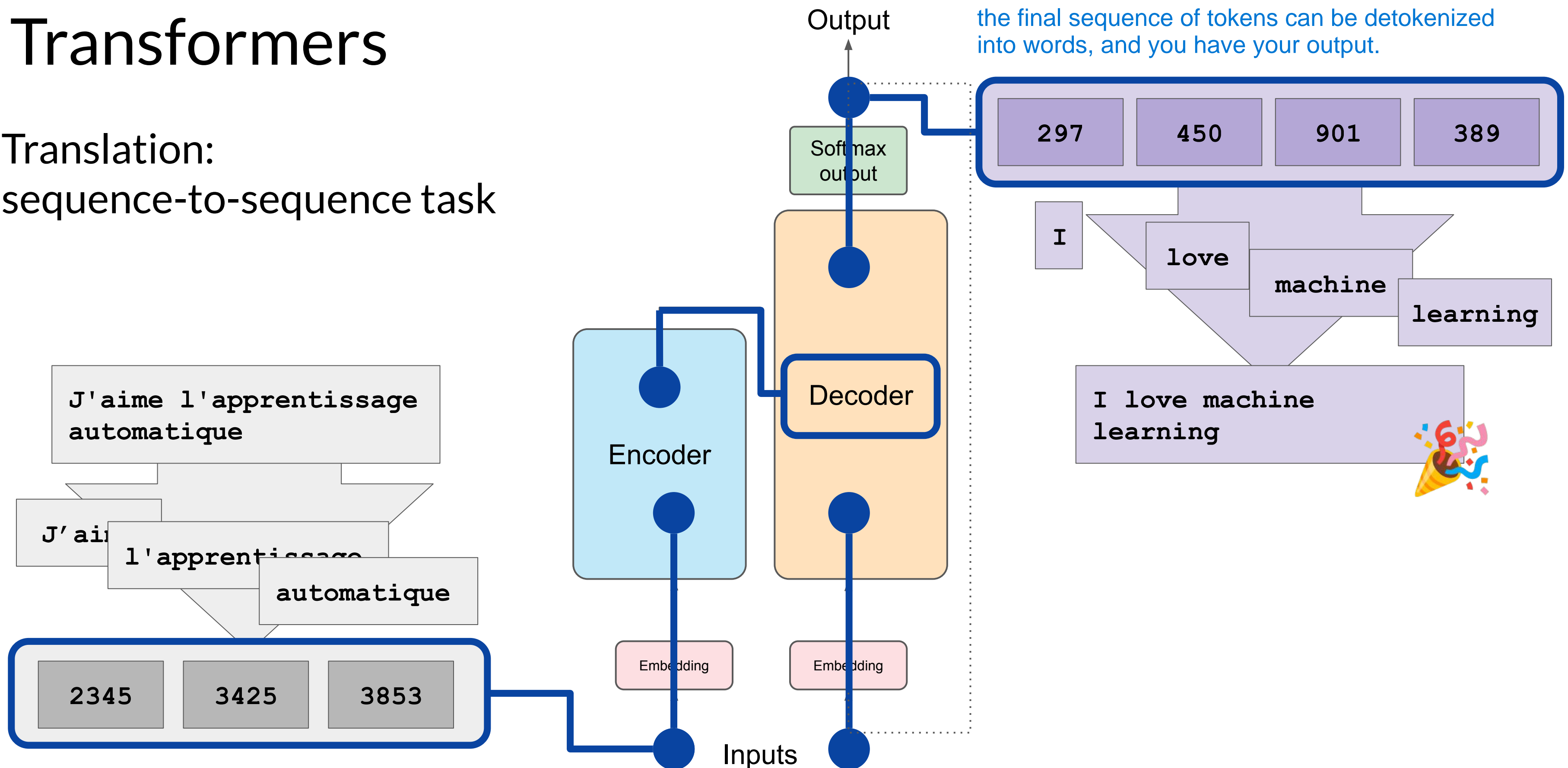
Transformers

Translation:
sequence-to-sequence task

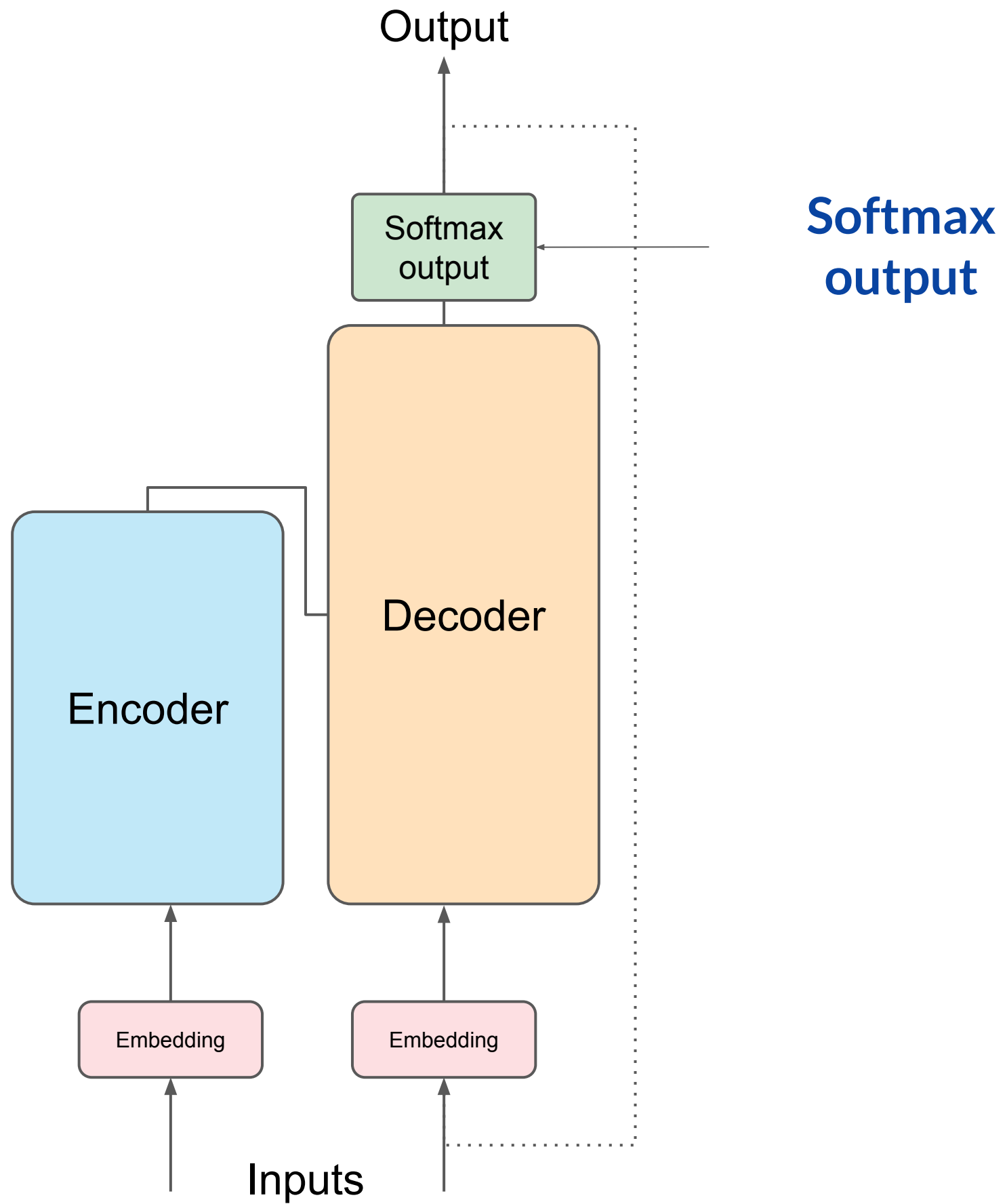


Transformers

Translation:
sequence-to-sequence task



Transformers

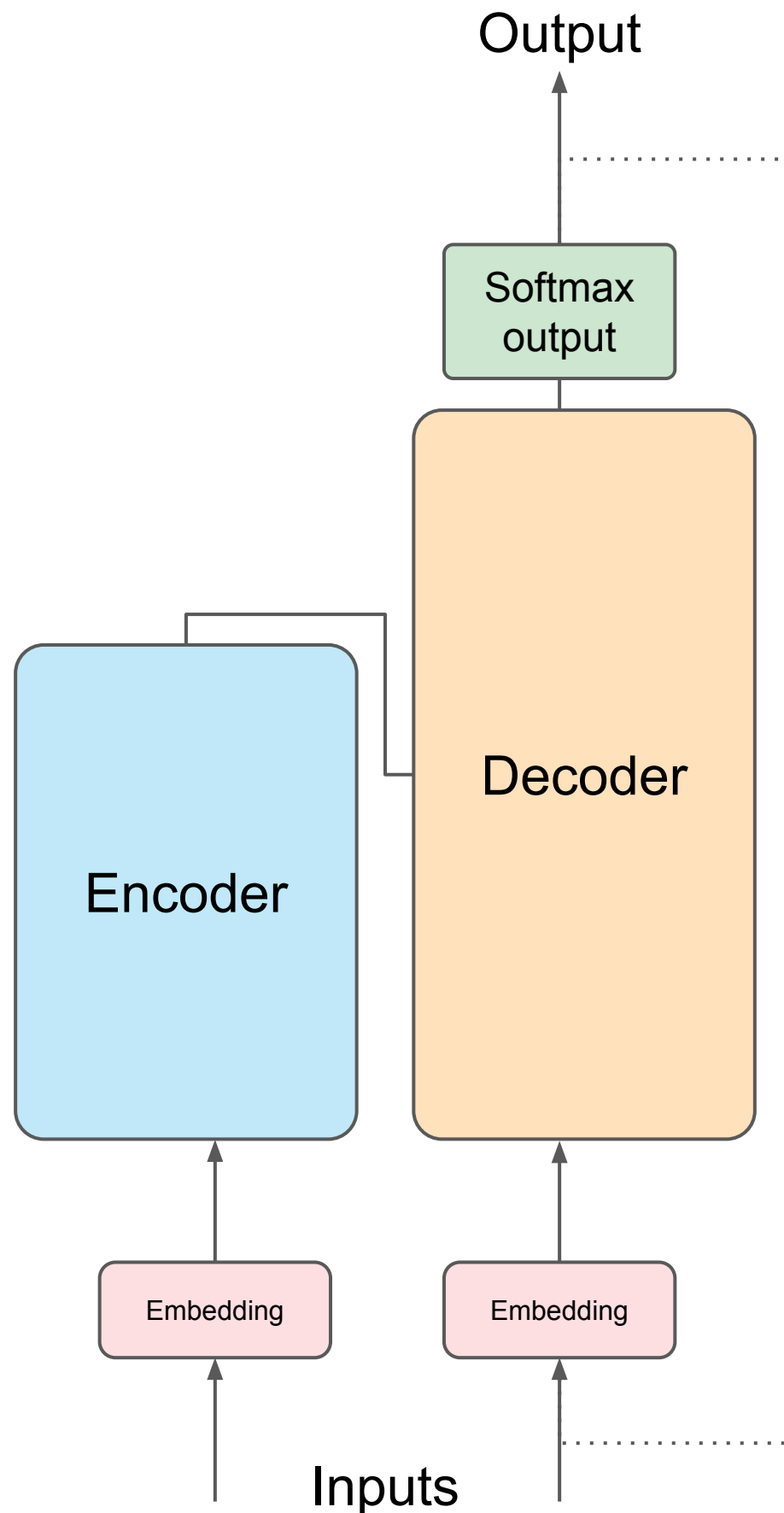


Transformers

TO SUMMARIZE:

Encoder

Encodes inputs (“prompts”) with contextual understanding and produces one vector per input token.



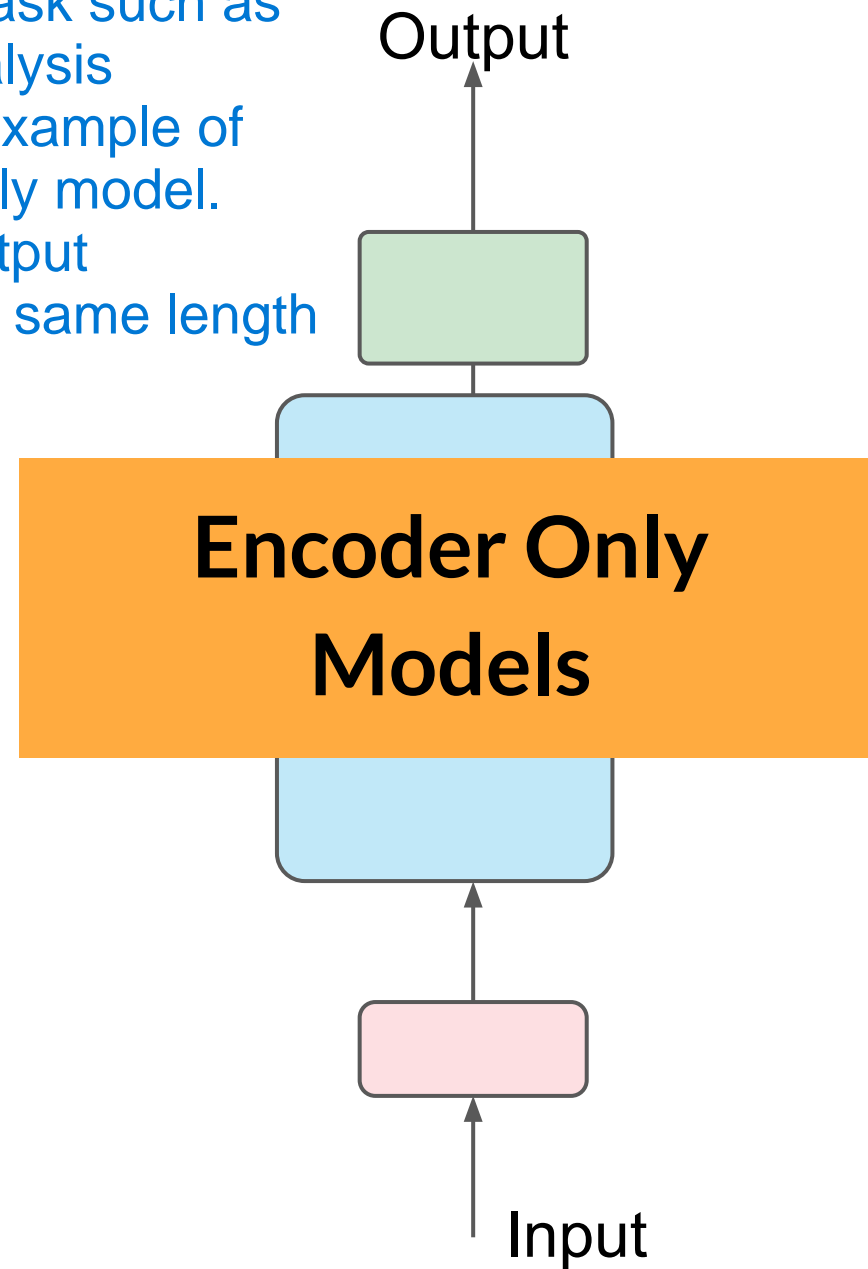
Decoder

Accepts input tokens and generates new tokens.

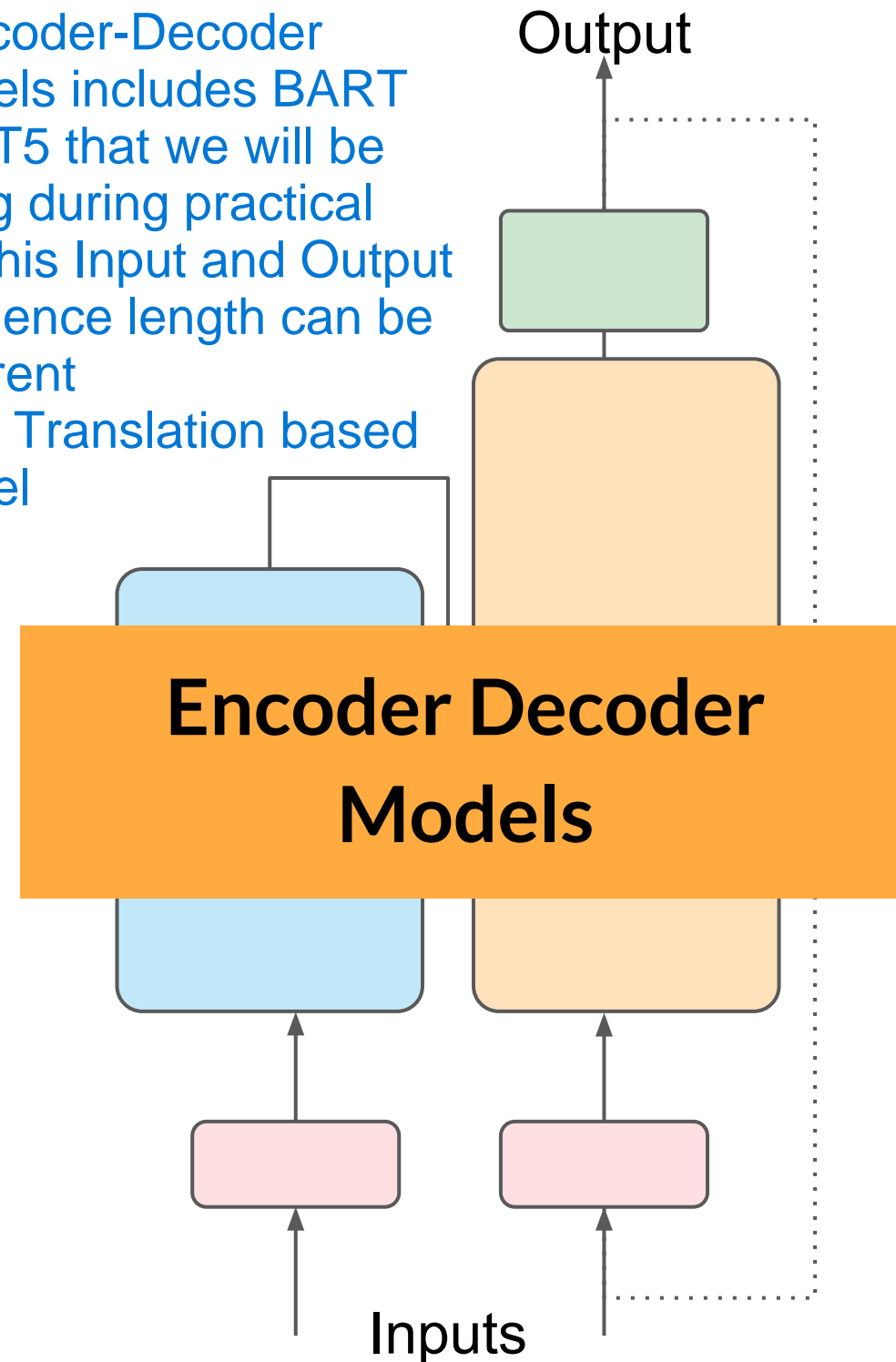
Transformers

In above text translation example we have used both encoder and decoder in conjunction.
We can also split encoder and decoder and use them separately to explore/discover the new variations:

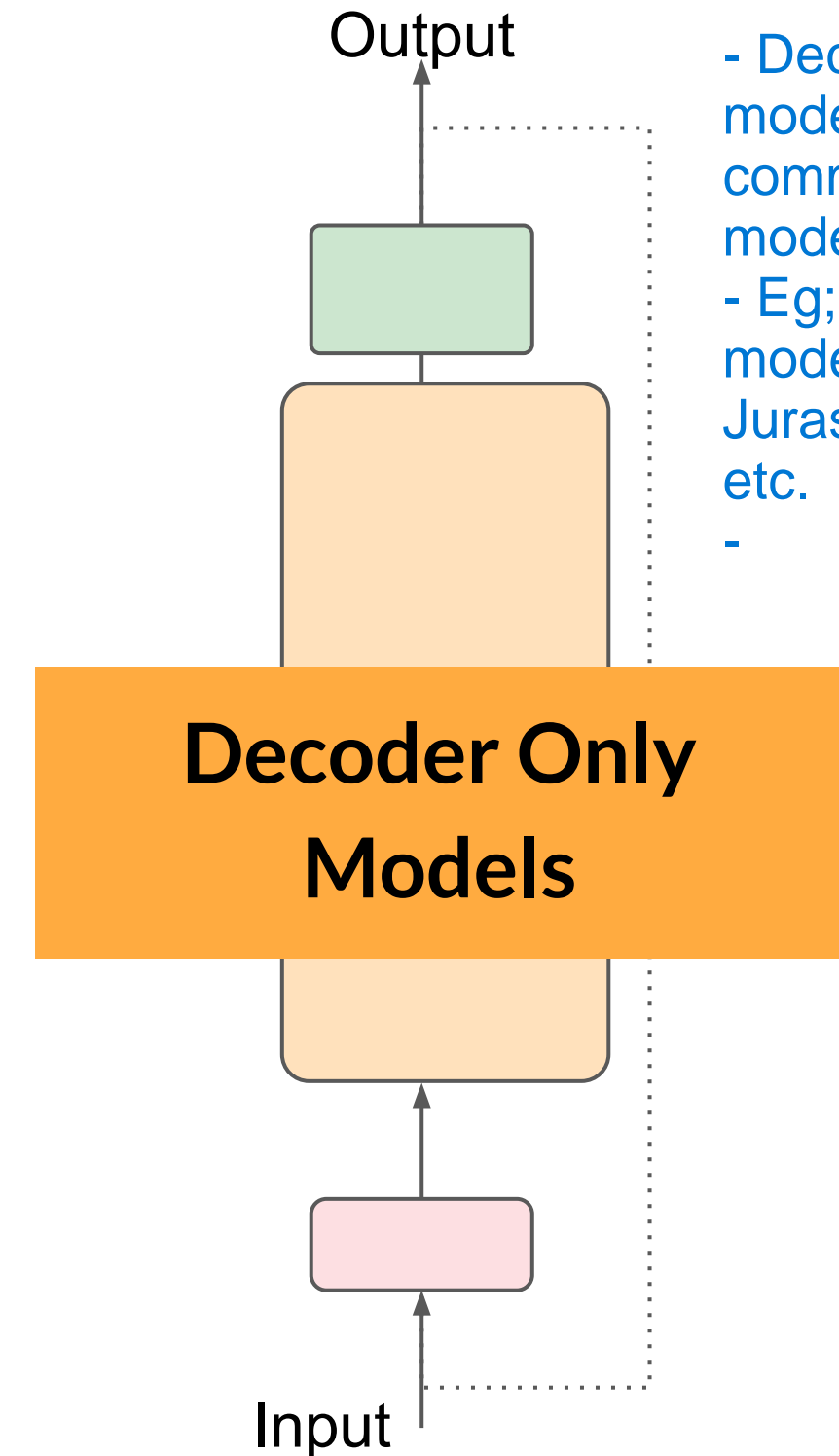
- Encoder only model can be used to perform classification task such as Sentiment Analysis
- BERT is an example of an encoder-only model.
- Input and Output sequence is of same length



- Encoder-Decoder models includes BART and T5 that we will be using during practical
- In this Input and Output sequence length can be different
- Eg; Translation based model



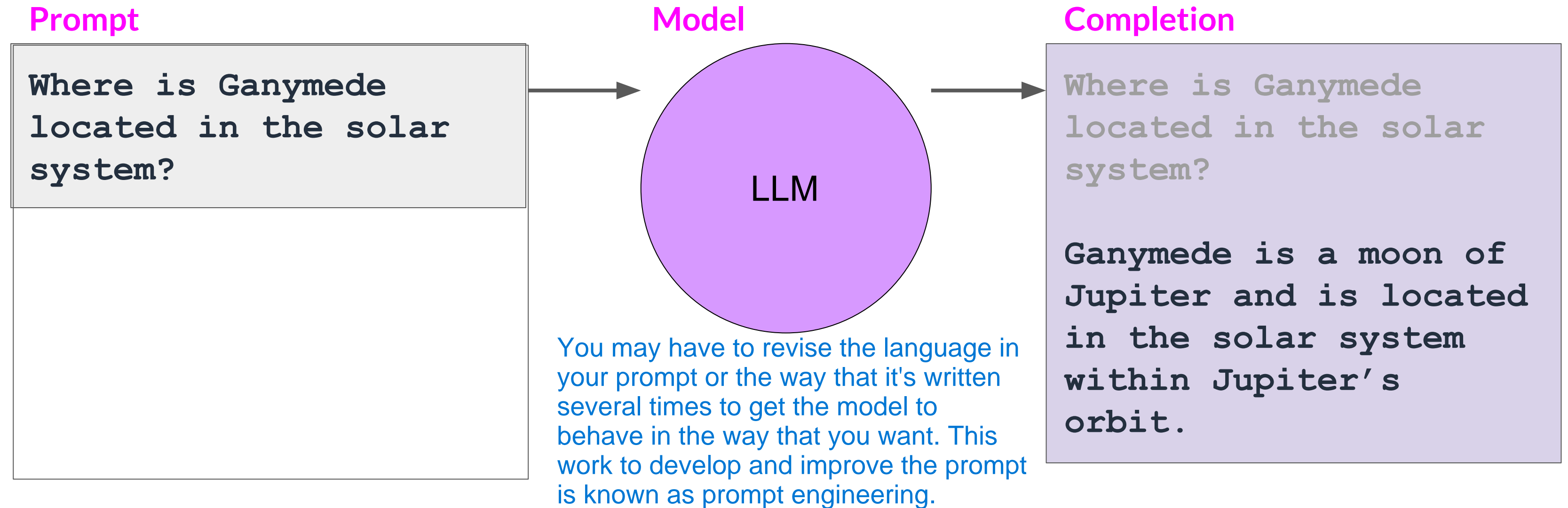
- Decoder only model is the most commonly used model today.
- Eg; GPT family model, BLOOM, Jurassic, LLaMA etc.
-



Prompting and prompt engineering

Prompting and prompt engineering

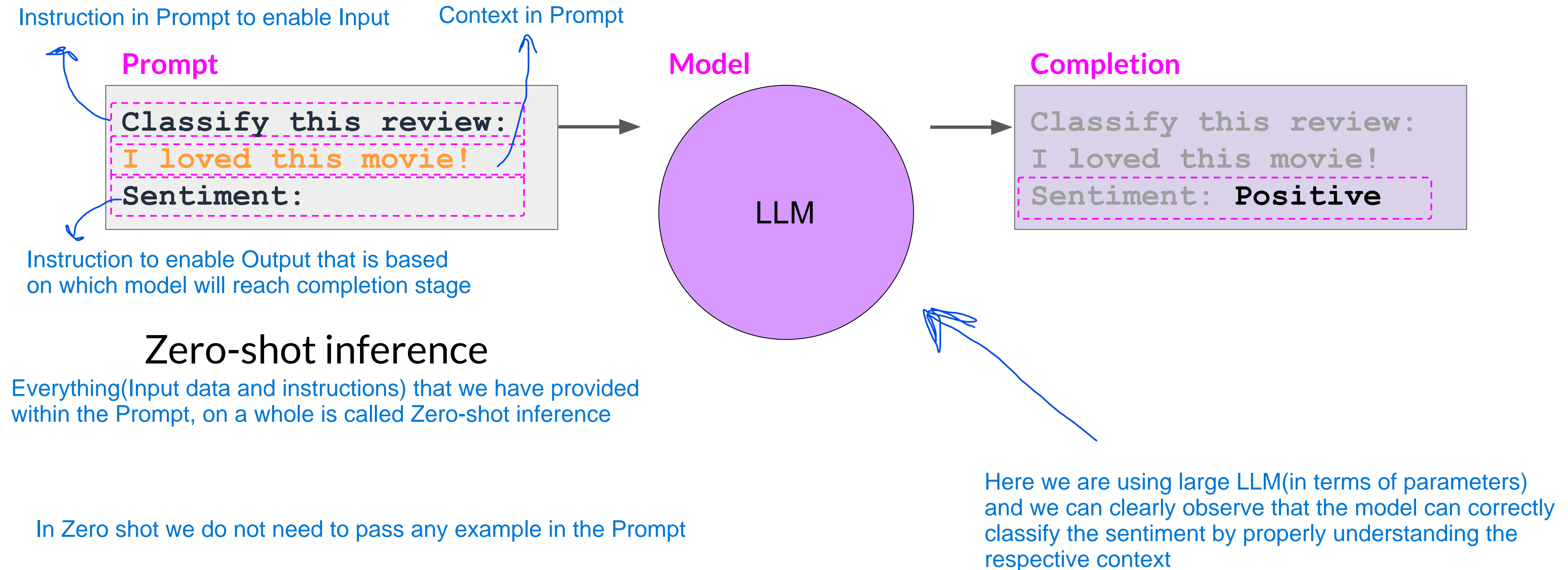
This slide already discussed in above slide



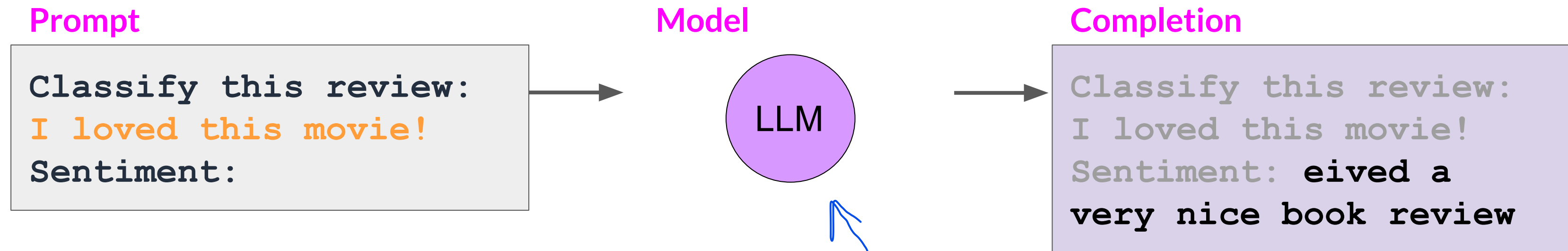
Context window: typically a few thousand words

One powerful strategy to get the model to produce better outcomes is to include examples of the task that you want the model to carry out inside the prompt. Providing examples inside the context window is called in-context learning.

In-context learning (ICL) - zero shot inference



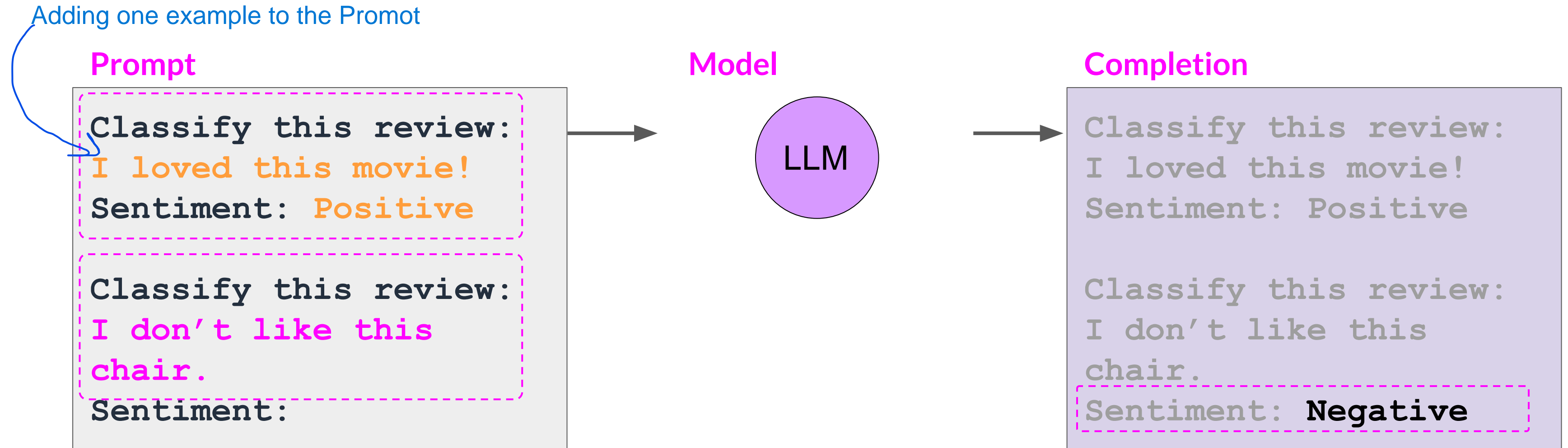
In-context learning (ICL) - zero shot inference



On the other hand if we would have used the smaller LLM model then then we can clearly see in the completion that the sentiment is not captured appropriately which signifies that the model was not able to map the deep contextual understanding. This can resolved by either fine tuning smaller LLM model or by using the large LLM model(which will be computationally expensive)

When we say fine tuning on top of smaller LLM model then one most powerful way is to give examples in the prompt so that even smaller LLM models can do the job more precisely. This same thing is shown in the next slide.

In-context learning (ICL) - one shot inference

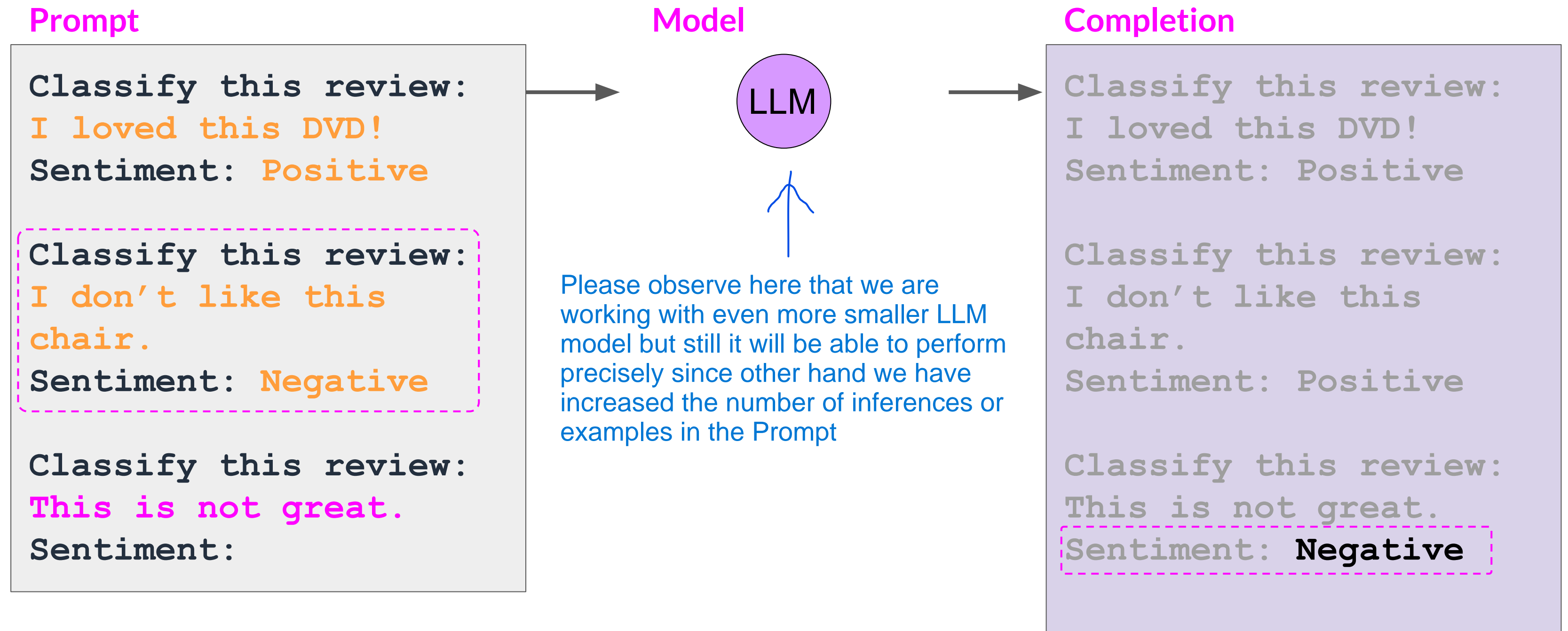


One-shot inference

When an example(Single example) is included in the Prompt that demonstrates the tasks to be carried out from the model is known as One-shot inference

In-context learning (ICL) - few shot inference

Sometimes a single example won't be enough for the model to learn what you want it to do. So you can extend the idea of giving a single example to include multiple examples. This is known as few-shot inference.



Summary of in-context learning (ICL)

Prompt // Zero Shot

Classify this review:
I loved this movie!
Sentiment:

Prompt // One Shot

Classify this review:
I loved this movie!
Sentiment: Positive

Classify this review:
I don't like this
chair.
Sentiment:

Prompt // Few Shot >5 or 6 examples

Classify this review:
I loved this movie!
Sentiment: Positive

Classify this review:
I don't like this
chair.
Sentiment: Negative

Classify this review:
Who would use this
product?
Sentiment:

Context Window
(few thousand words)

The significance of scale: task ability

BERT*
110M

BLOOM
176B



TO summarize we can clearly understand the importance or Significance of scale (model with large memory or parameter). Models with large scale(parameters) can make prediction or generate appropriate output with nearly no inferences or examples in the Prompt. Whereas, Small models will require more and more examples in Prompt to carry out the same task

Ques: Which in-context learning method involves creating an initial prompt that states the task to be completed and includes a single example question with answer followed by a second question to be answered by the LLM?

Ans: One Shot inference

*Bert-base

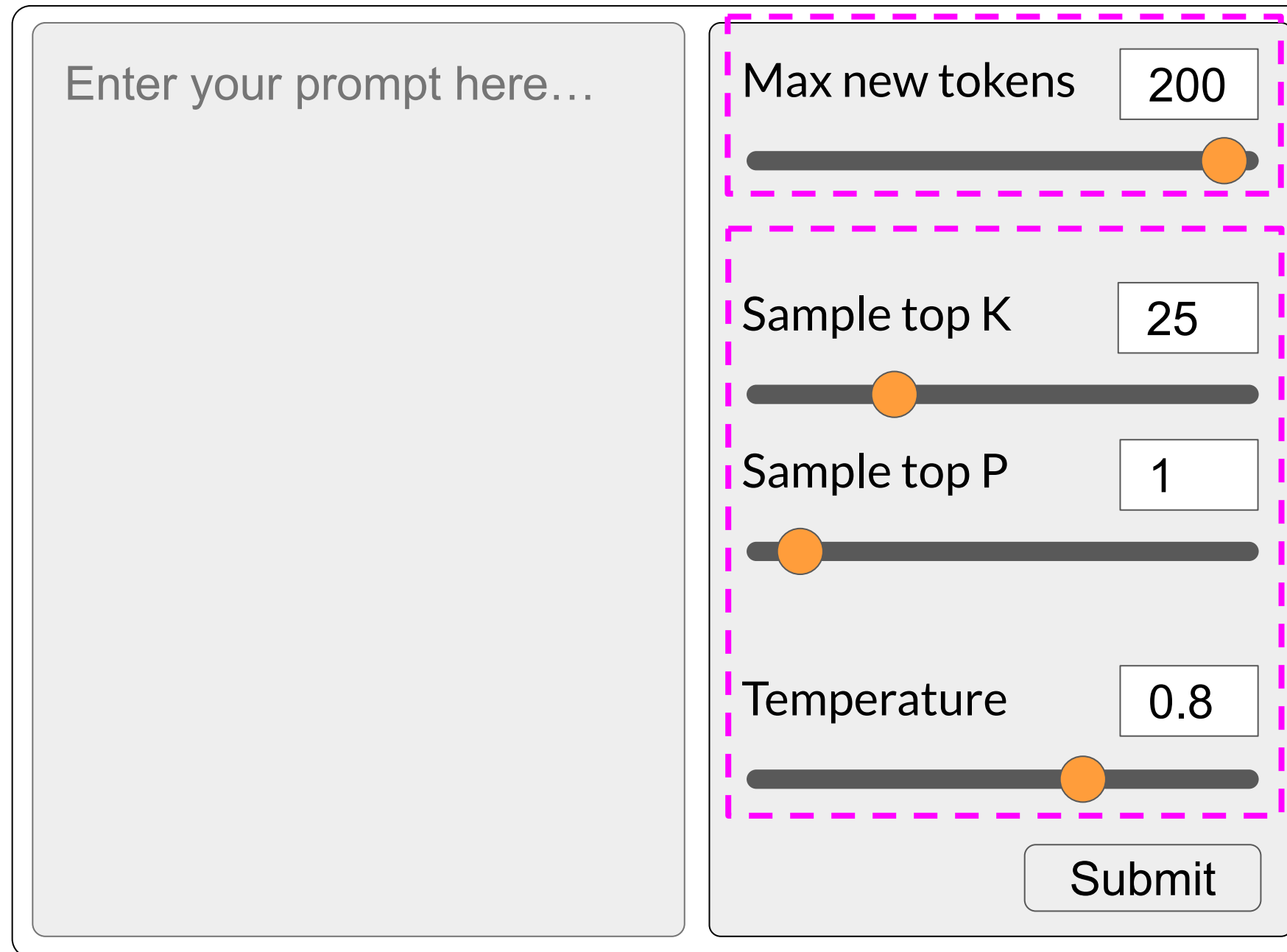
Generative configuration parameters for inference

Agenda: We'll examine some of the methods and associated configuration parameters that we can use to influence the way that the model makes the final decision about next-word generation.

One way is adding examples but on top of it will discuss different configurations that can be applied. Simple example to understand in the Chat GPT feature in edge has 3 options Creative, Balanced and More precise (which are actually the configurations)

Generative configuration - inference parameters

Below is the kind of UI that we get for LLM playground in Hugging Face



The image shows a UI for an LLM playground. On the left is a large text input area with the placeholder "Enter your prompt here...". To the right of the input area is a panel containing four inference parameters, each with a slider and a numeric input field. The parameters are: "Max new tokens" (slider at 200), "Sample top K" (slider at 25), "Sample top P" (slider at 1), and "Temperature" (slider at 0.8). A "Submit" button is located at the bottom right of the parameter panel. A dashed pink box highlights the four parameter rows. A bracket on the right side of the parameter panel points to the text "Inference configuration parameters".

Parameter	Value
Max new tokens	200
Sample top K	25
Sample top P	1
Temperature	0.8

Inference
configuration
parameters

Please note that these configuration parameters are different from the parameters that model learn during training (Learning Parameters)

Generative configuration - max new tokens

Instead, these configuration parameters are invoked at inference time(that is at the Prompt stage where we are passing input)

Enter your prompt here...

Max new tokens 200

Sample top K 25

Sample top P 1

Temperature 0.8

Submit

Max new tokens

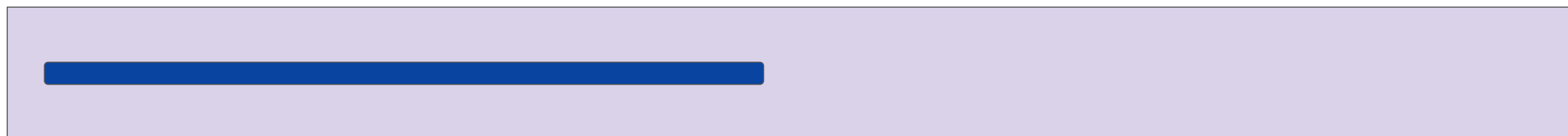
This configuration parameter represents the Maximum token size in the Completion stage that is the maximum number of characters that can be generated as an Output.

OR

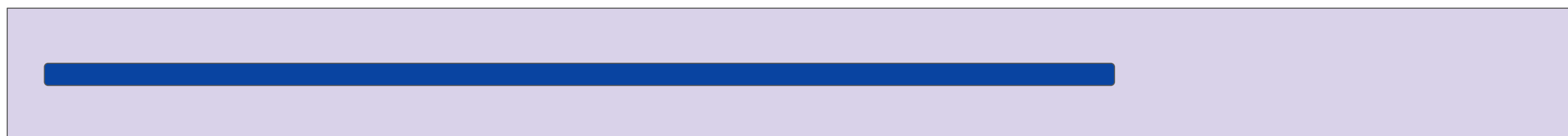
In other words this is putting the cap on the number of times model will go through the selection process(Recall Step 7 in End to End Transformer architecture)

Generative config - max new tokens

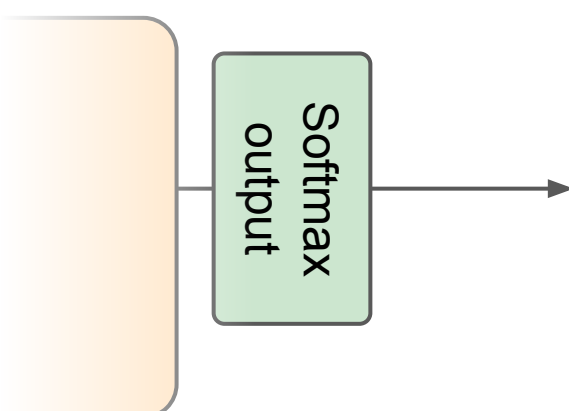
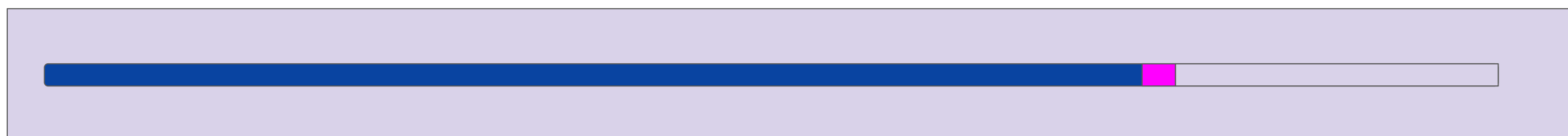
max_new_tokens = 100



max_new_tokens = 150

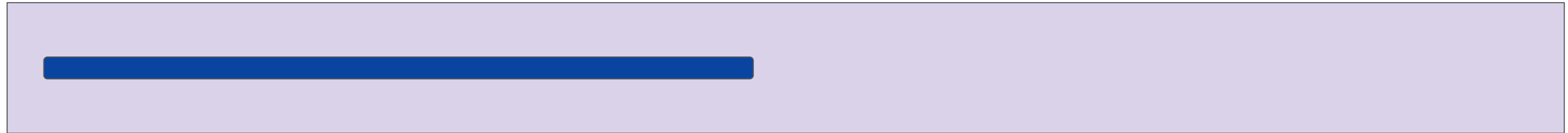


max_new_tokens = 200

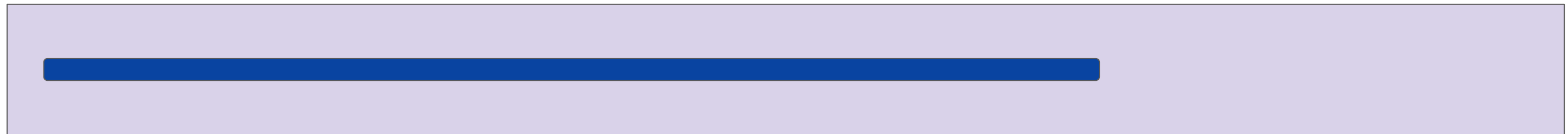


Generative config - max new tokens

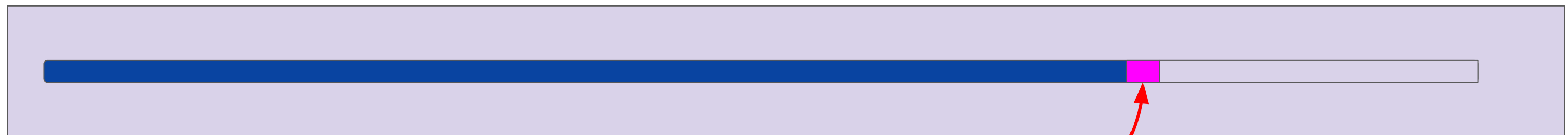
max_new_tokens = 100



max_new_tokens = 150



max_new_tokens = 200



Stop token

Since this is the maximum limit of the new token

Generative config - greedy vs. random sampling

Token
probability

Softmax
output

0.20	cake
0.10	donut
0.02	banana
0.01	apple
...	...

prob

word

Softmax
output

0.20	cake
0.10	donut
0.02	banana
0.01	apple
...	...

greedy: The word/token with the highest probability is selected.

The simplest form of next-word prediction, where the model will always choose the word with the highest probability is called Greedy Sampling. This method can work very well for short generation but is susceptible to repeated words or repeated sequences of words. If you want to generate text that's more natural, more creative and avoids repeating words, you need to use Random Sampling

random(-weighted) sampling: select a token using a random-weighted strategy across the probabilities of all tokens.

Here, there is a 20% chance that 'cake' will be selected, but 'banana' was actually selected.

The output from the transformer's SoftMax layer is a probability distribution across the entire dictionary of words that the model uses.

Generative configuration - top-k and top-p

Two Settings, top p and top k are sampling techniques that we can use to help limit the random sampling and increase the chance that the output will be sensible.

Enter your prompt here...

Max new tokens

200

Sample top K

25

Sample top P

1

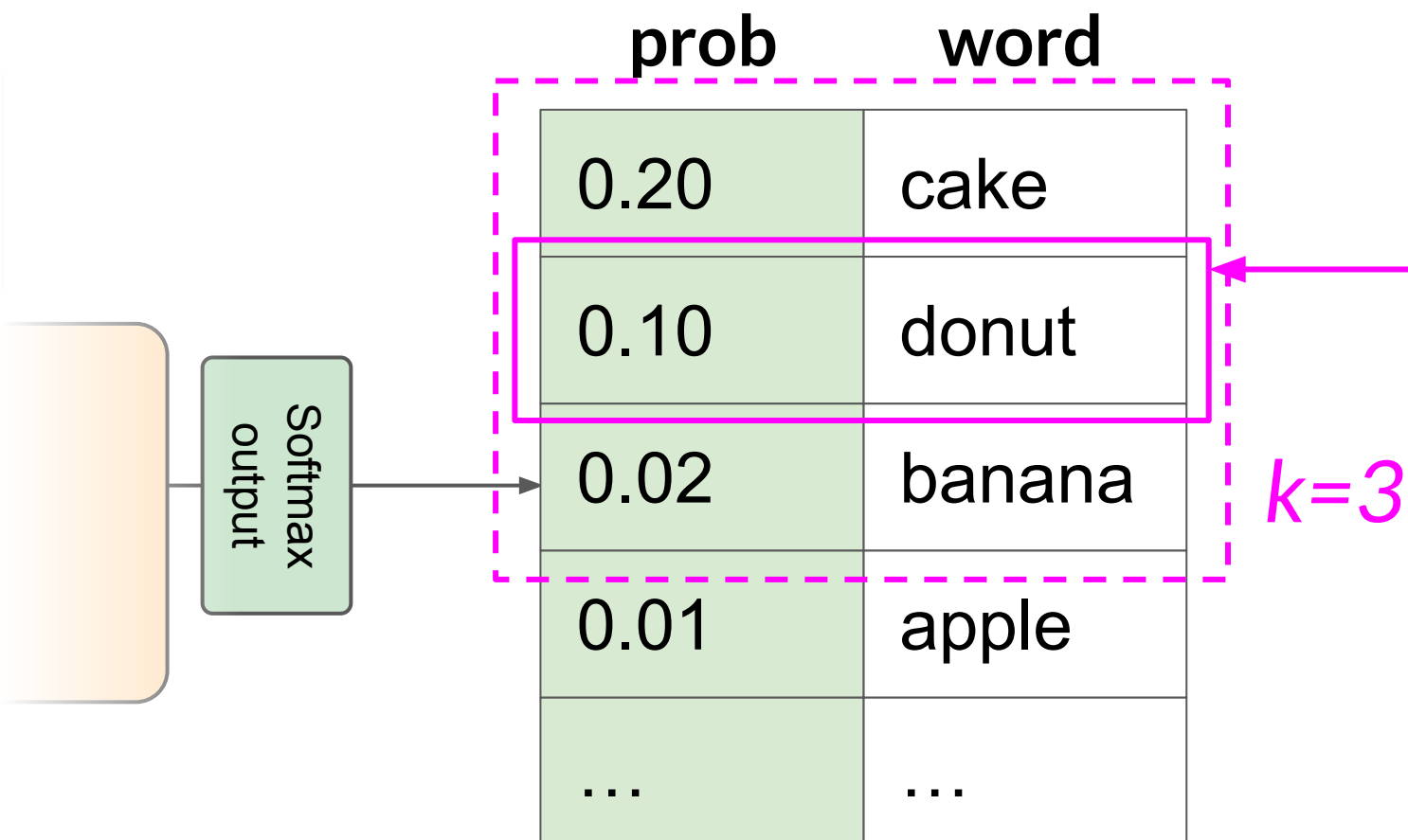
Temperature

0.8

Submit

Top-k and top-p sampling

Generative config - top-k sampling

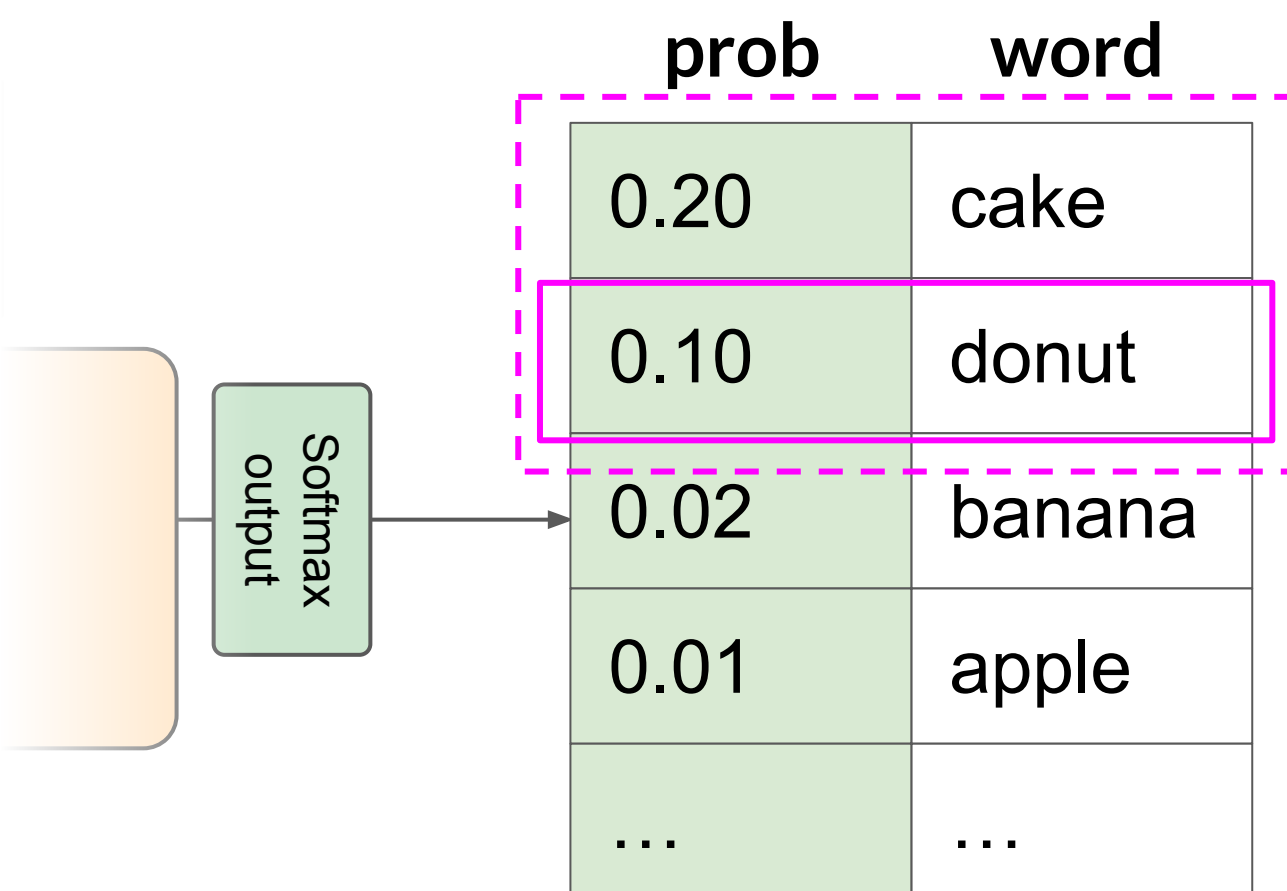


Please observe here that Probabilities are arranged in descending order

top-k: select an output from the top-k results after applying random-weighted strategy using the probabilities

This method can help the model have some randomness while preventing the selection of highly improbable completion words. This in turn makes your text generation more likely to sound reasonable and to make sense. In short we are overall trying to combine both Greedy and Random sampling approaches

Generative config - top-p sampling



top-p: select an output using the random-weighted strategy with the top-ranked consecutive results by probability and with a cumulative probability $\leq p$.

$p = 0.30$

Starting from top keep aggregating probabilities until $\leq p$. The moment aggregation of probabilities violates the condition $\leq p$ stop at that point and apply random sampling on those set words before that stopping point.

To summarize, with top k, you specify the number of tokens to randomly choose from, and with top p, you specify the total probability that you want the model to choose from.

Generative configuration - temperature

- Temperature controls the randomness of the model output
- It influences the shape of the probability distribution that the model calculates for the next token.
- the higher the temperature, the higher the randomness, and the lower the temperature, the lower the randomness.

Enter your prompt here...

Max new tokens

200

Sample top K

25

Sample top P

1

Temperature

0.8

Submit

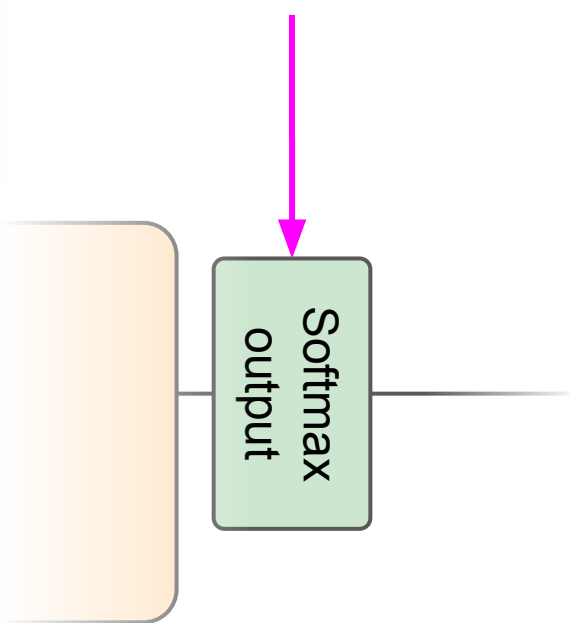
Temperature

Generative config - temperature

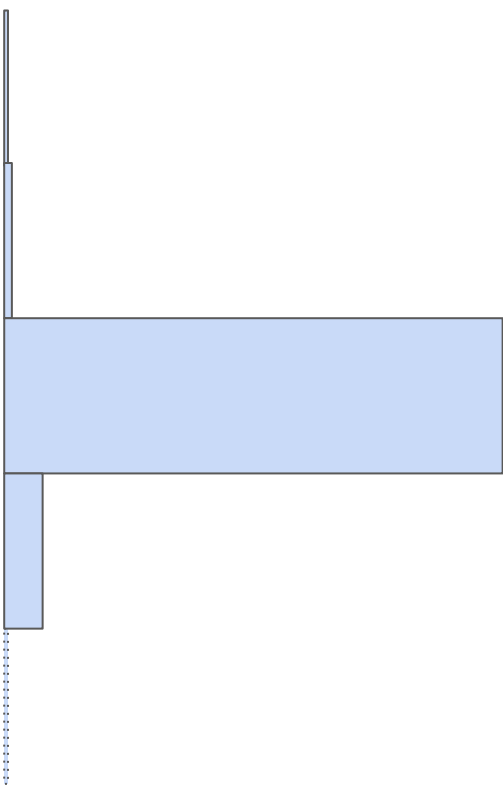
Leads to biasness or factual or less creative

Cooler temperature (e.g <1)

Temperature
setting



prob	word
0.001	apple
0.002	banana
0.400	cake
0.012	donut
...	...

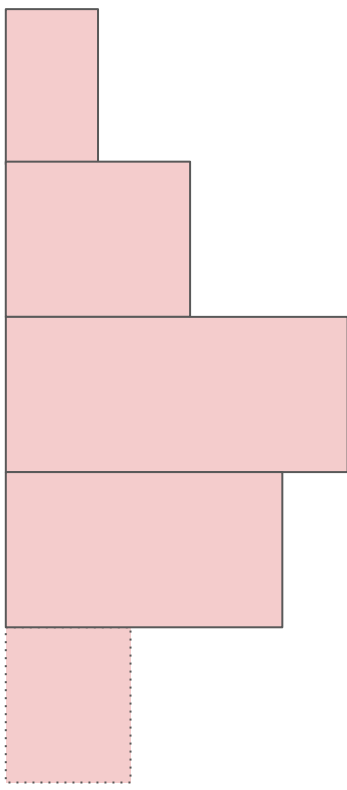


Strongly peaked
probability
distribution

Leads to creativity

Higher temperature (>1)

prob	word
0.040	apple
0.080	banana
0.150	cake
0.120	donut
...	...



Broader, flatter
probability
distribution

If you leave the temperature value equal to one, this will leave the SoftMax function as default and the unaltered probability distribution will be used.

Here we can clearly observe that when temperature is low or cooler, randomness is low and that's why cake becomes highly Probable($Pr=0.4$) as compared to other words. Hence we get Strongly peaked Probability distribution with probability being more concentrated on only a single word cake.

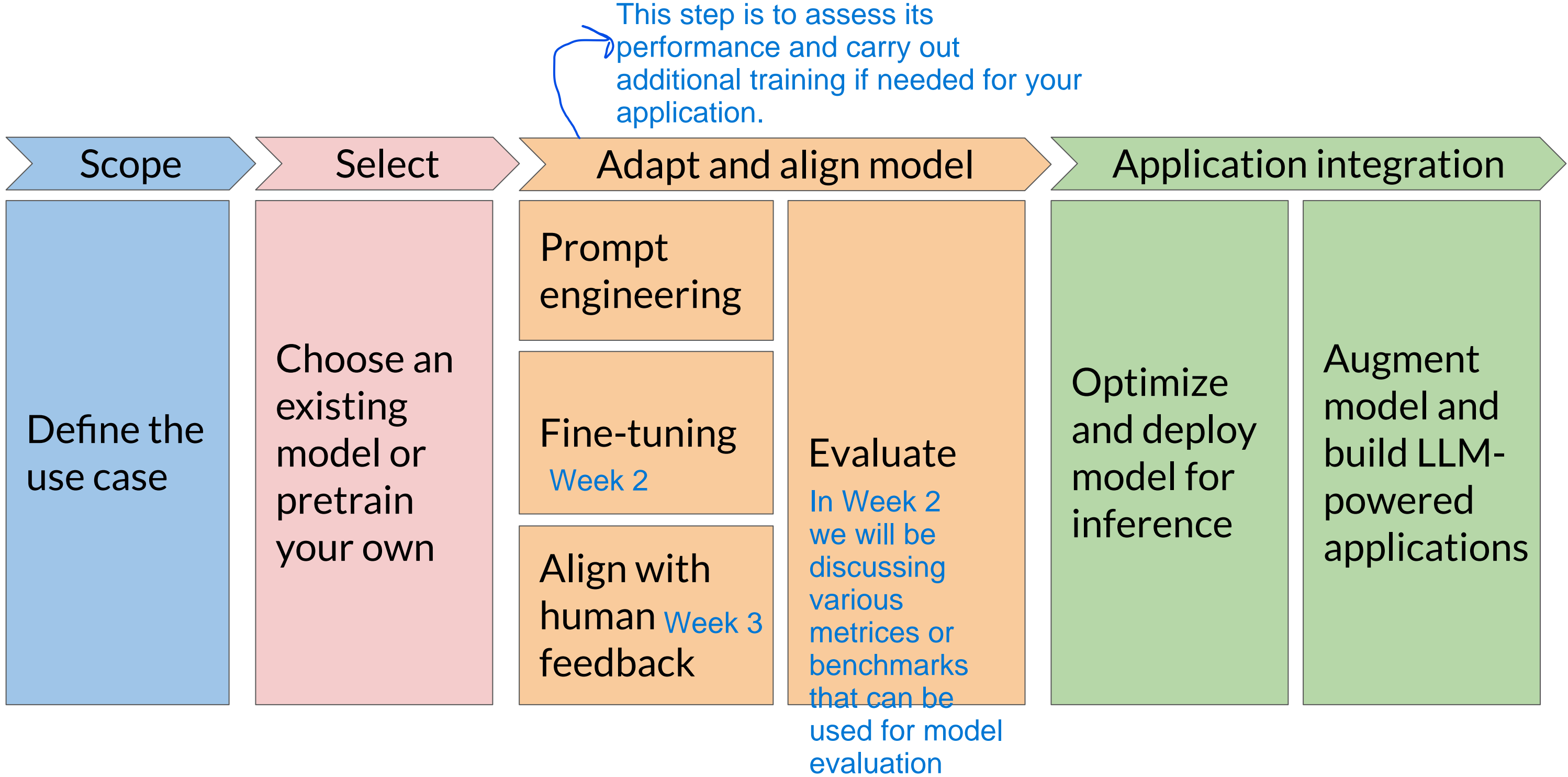
Whereas when temperature is high or warm then there is some degree of variations or randomness where probability is not concentrated only over a single word, producing more broader probability distribution

Ques: Which configuration parameter for inference can be adjusted to either increase or decrease randomness within the model output layer?

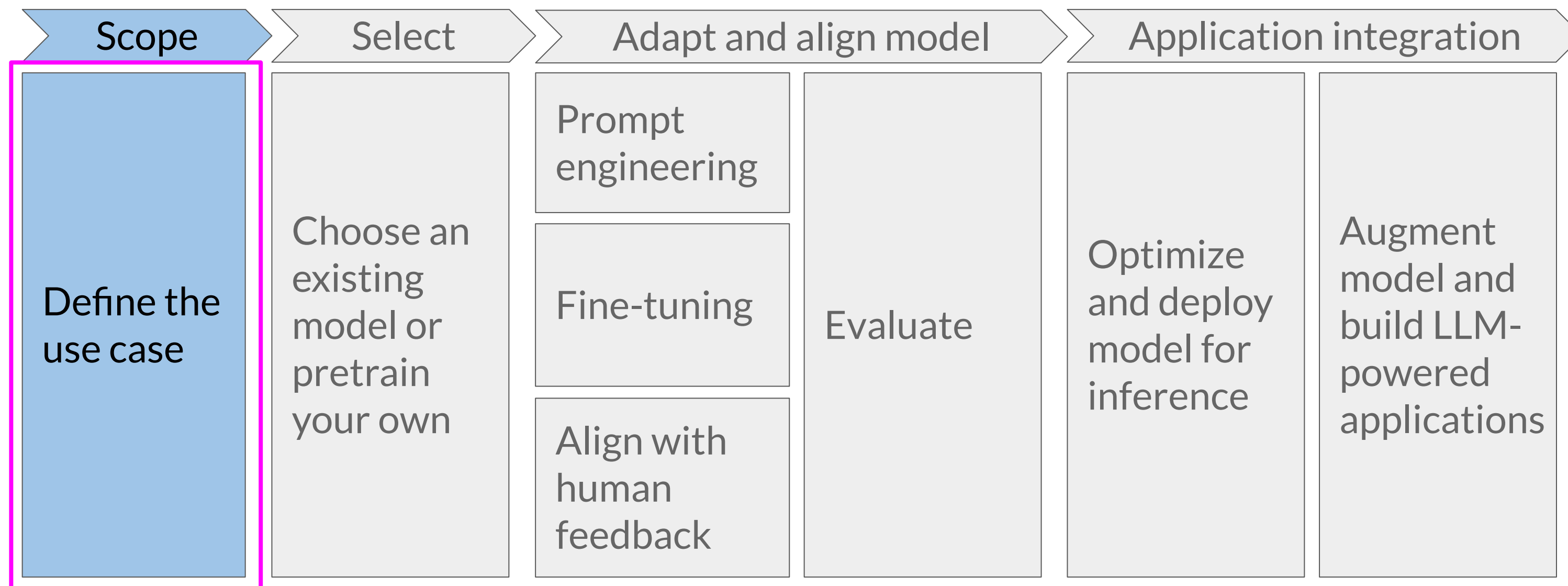
Ans: Temperature

Generative AI project lifecycle

Generative AI project lifecycle



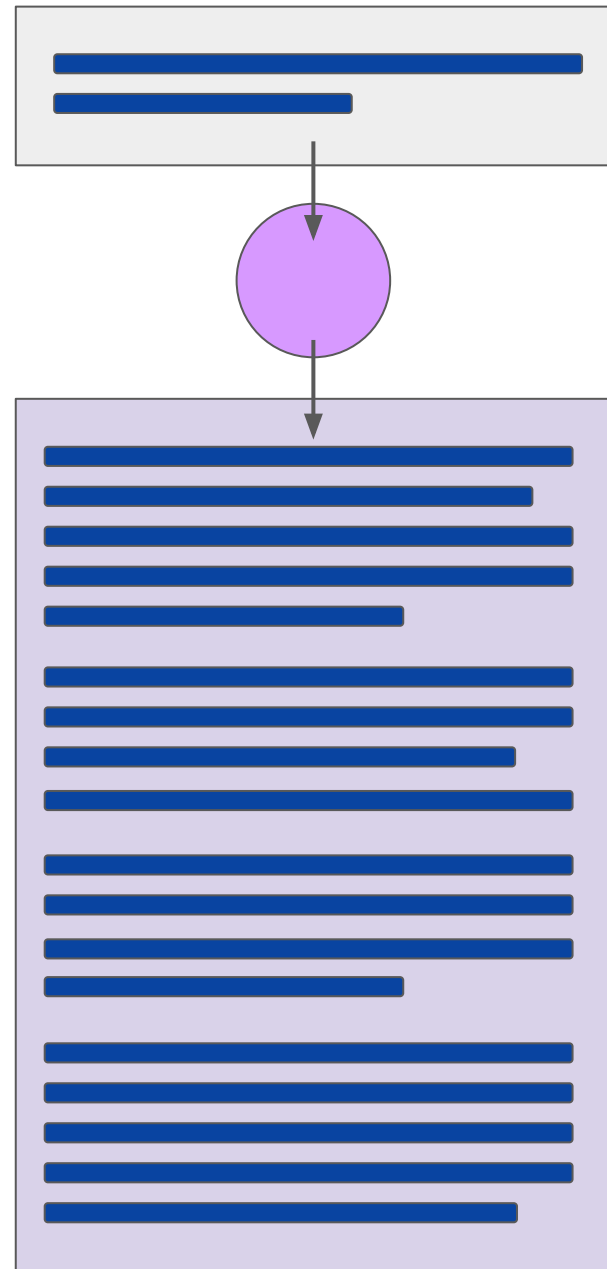
Generative AI project lifecycle



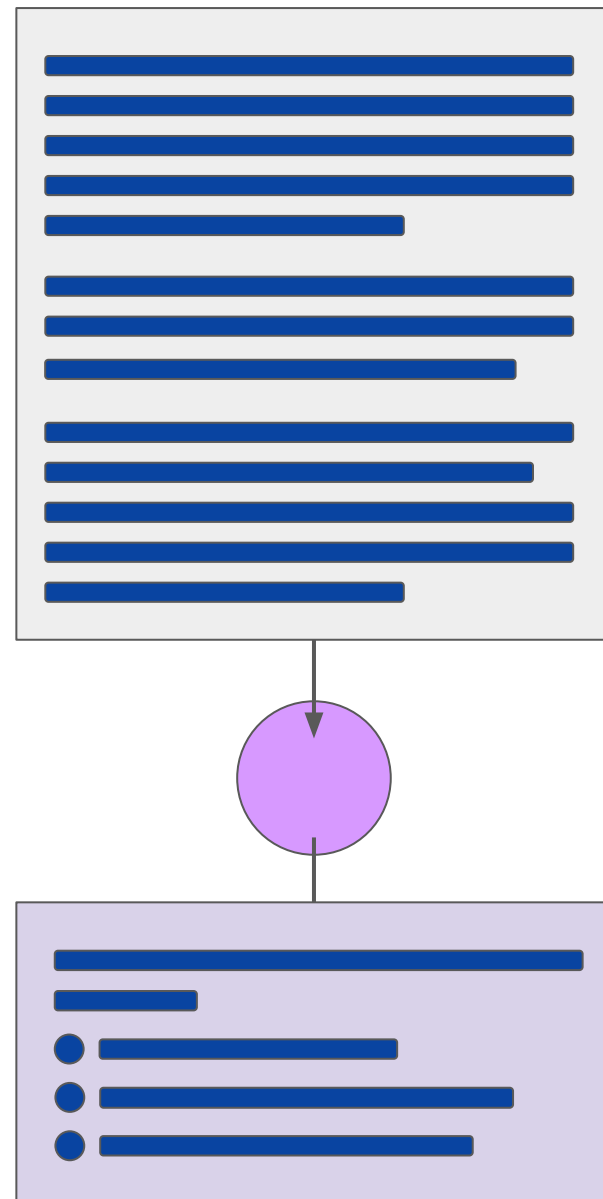
Good at many tasks?

Things to consider while defining the scope of the use case

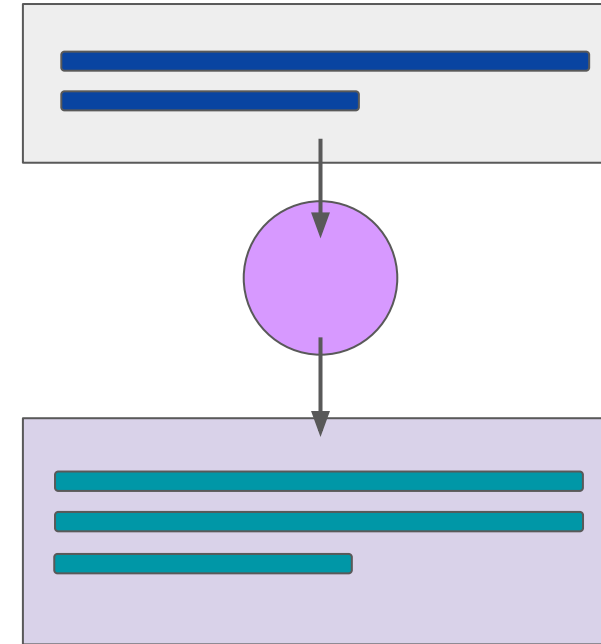
Essay Writing



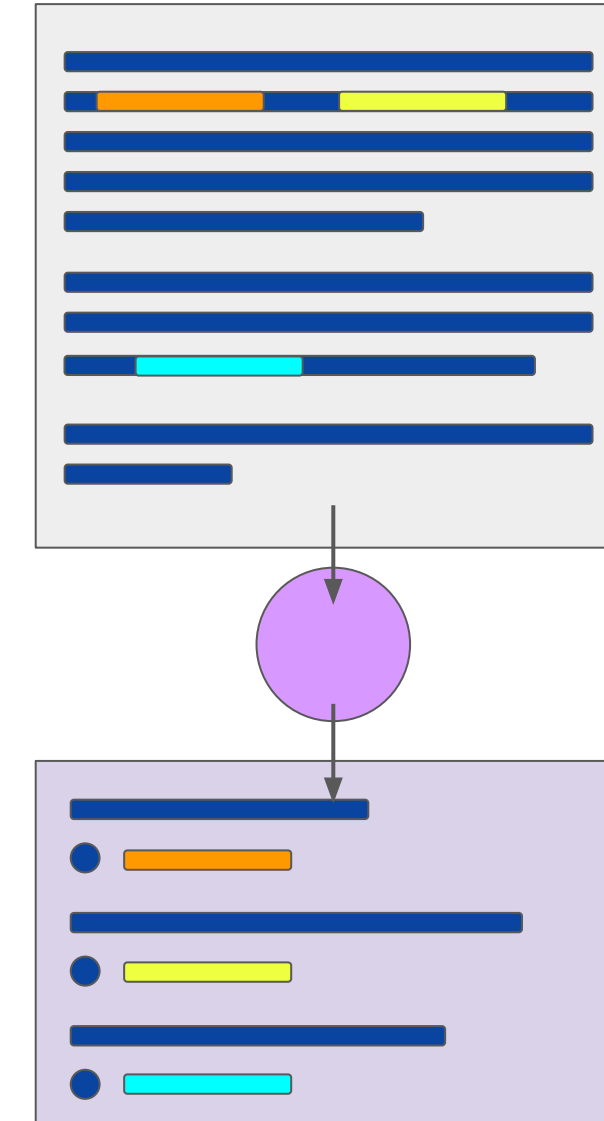
Summarization



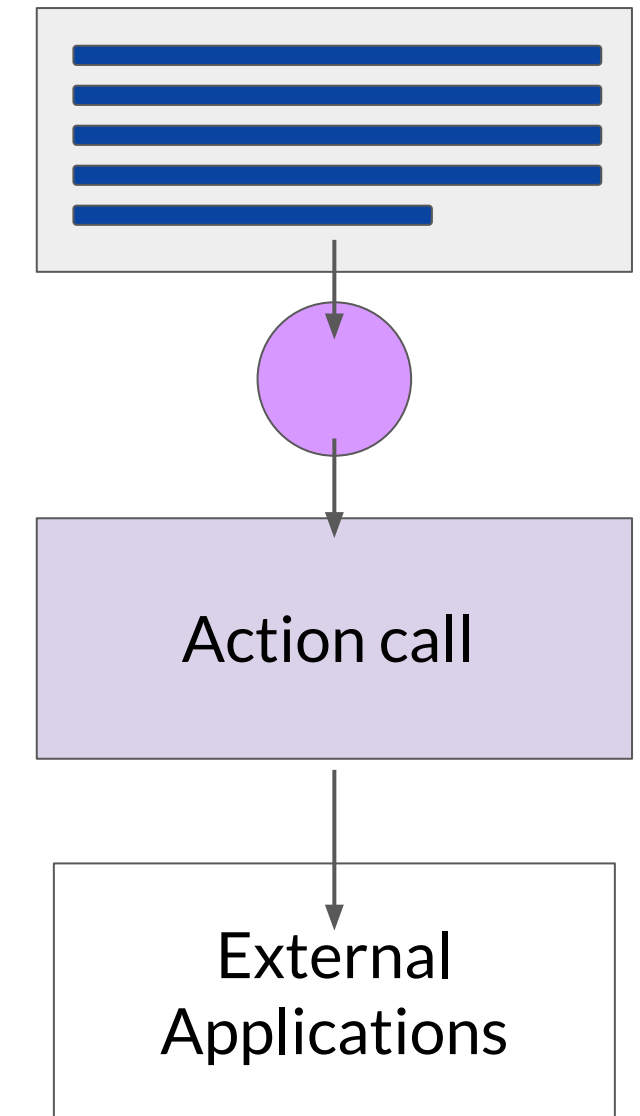
Translation



Information retrieval

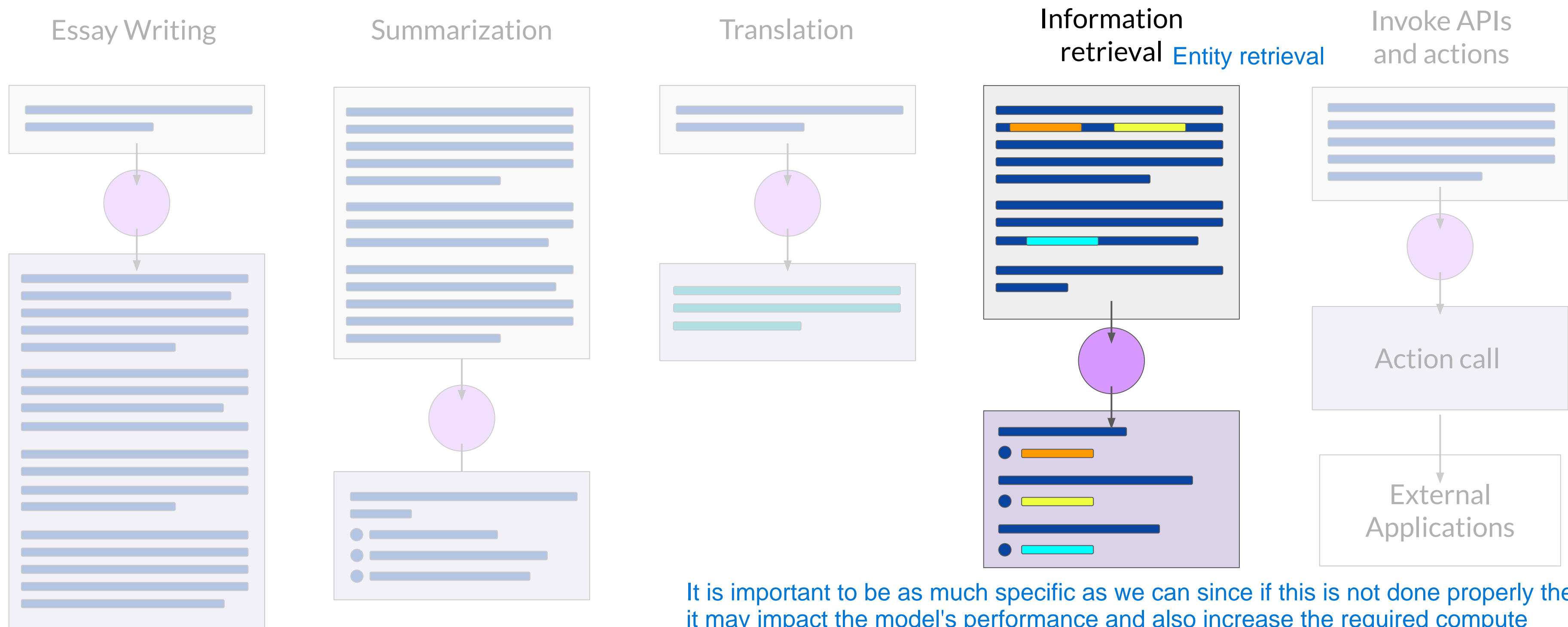


Invoke APIs and actions



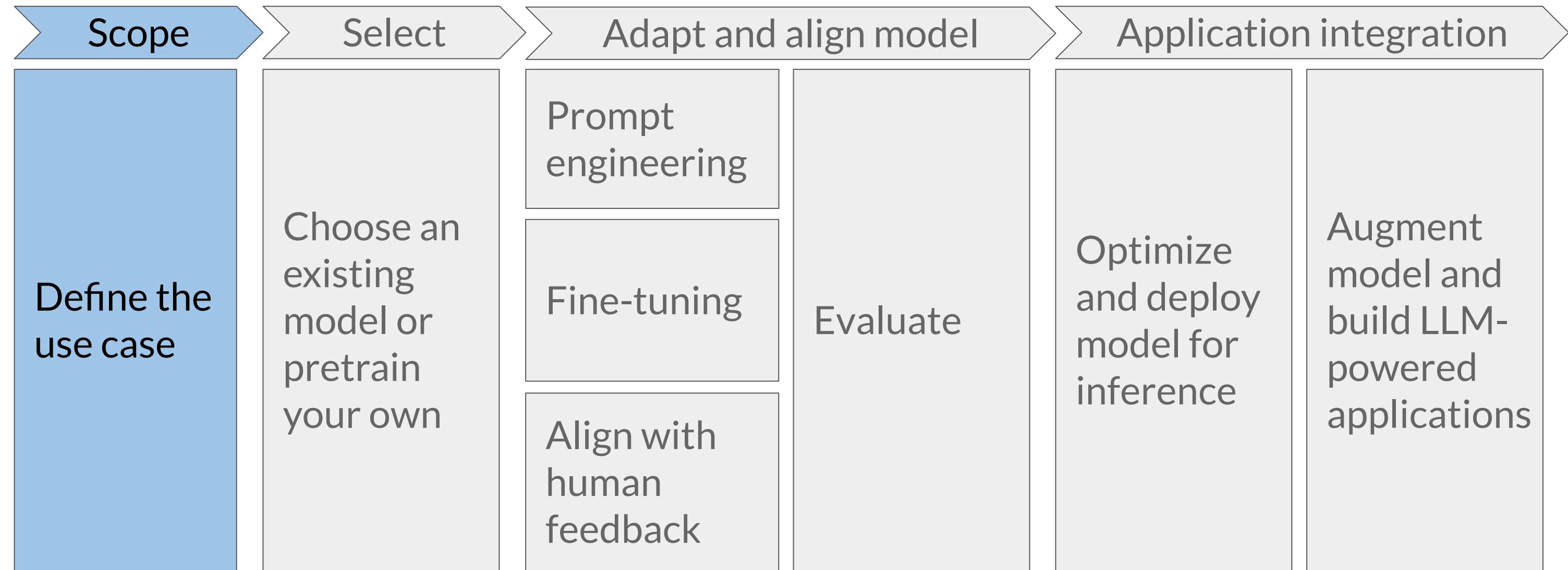
Or good at a single task?

Things to consider while defining the scope of the use case

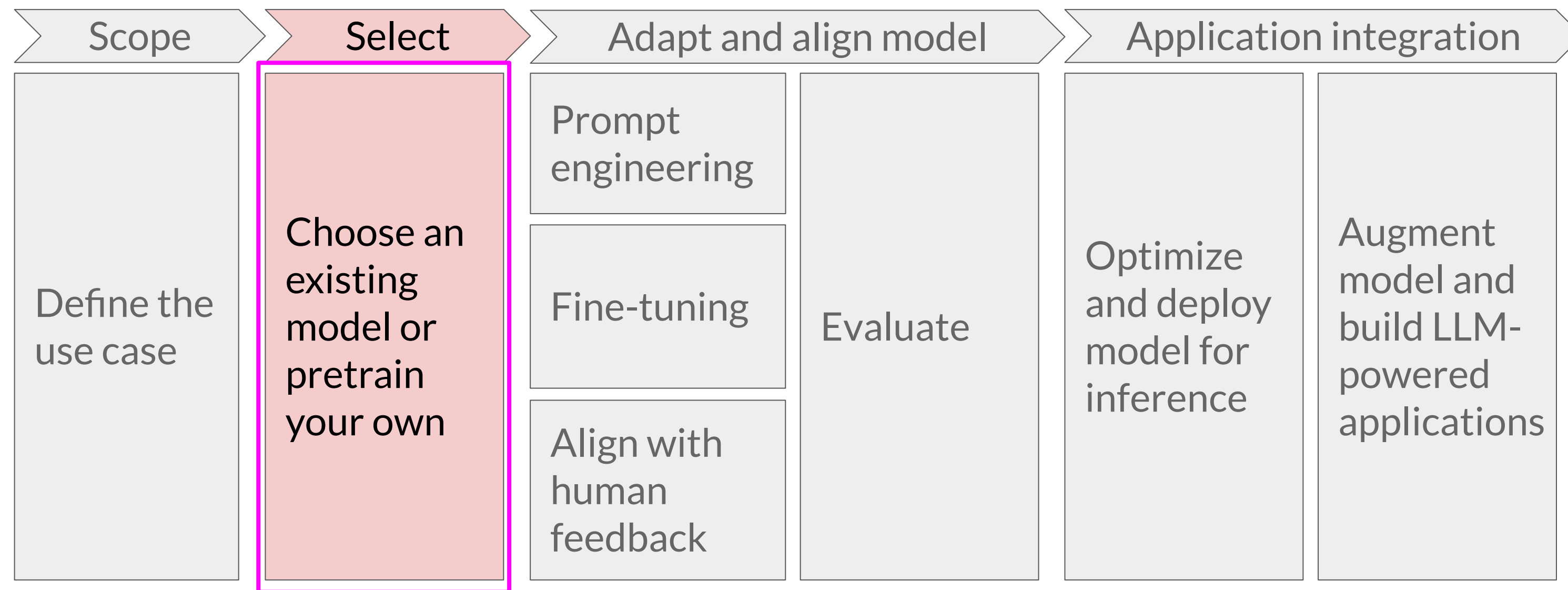


It is important to be as much specific as we can since if this is not done properly then it may impact the model's performance and also increase the required compute resources as well

Generative AI project lifecycle

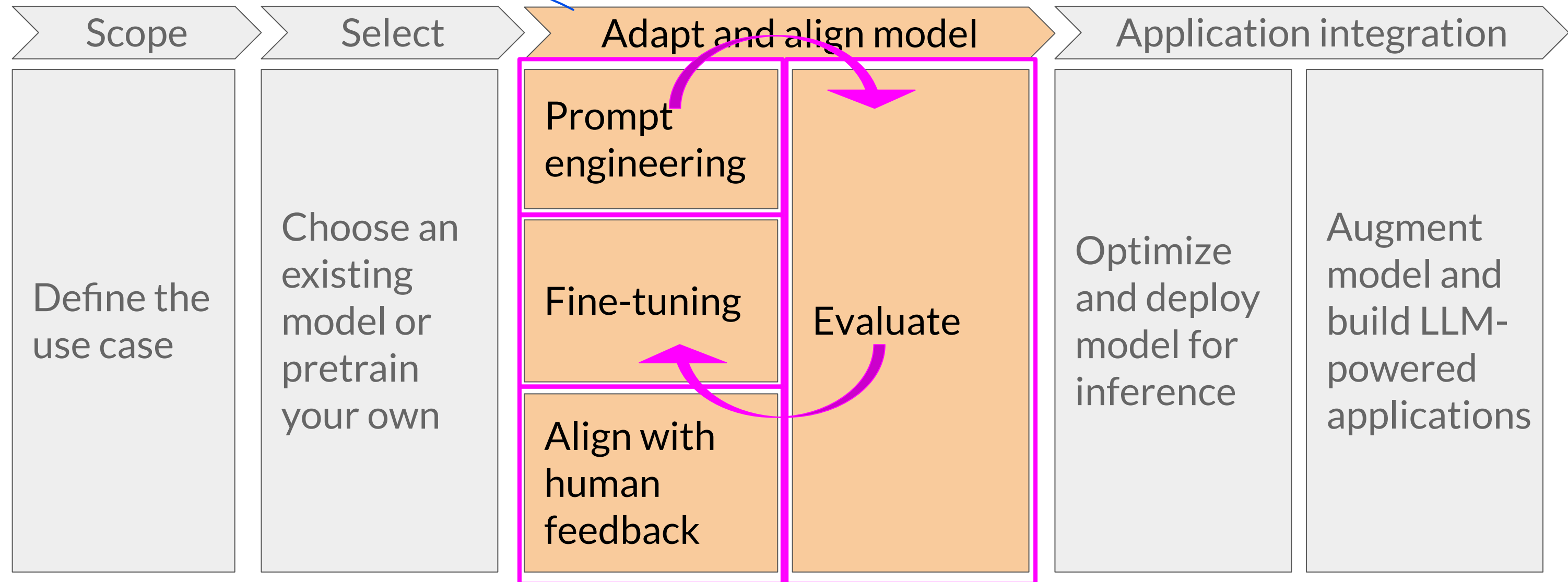


Generative AI project lifecycle

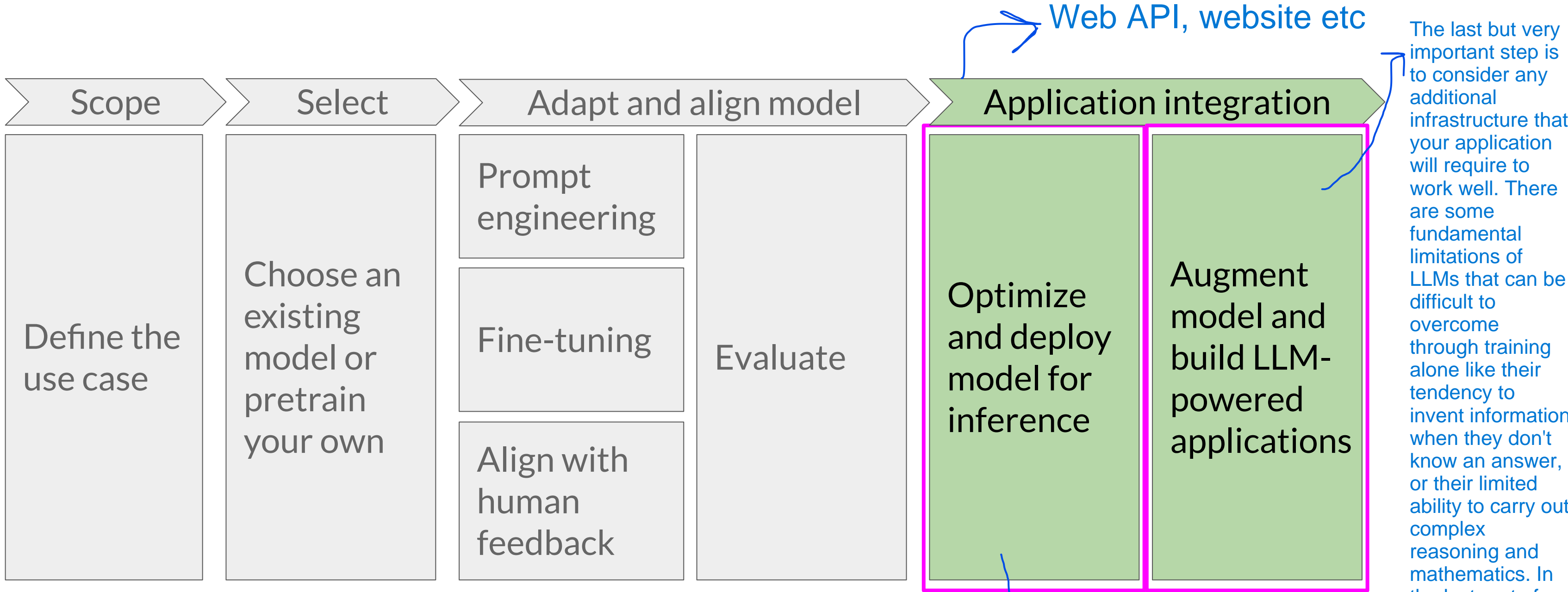


Generative AI project lifecycle

➤ Note that this adapt and aligned stage of app development can be highly iterative. You may start by trying prompt engineering and evaluating the outputs, then using fine tuning to improve performance and then revisiting and evaluating prompt engineering one more time to get the performance that you need.



Generative AI project lifecycle



At this stage, an important step is to optimize your model for deployment. This can ensure that you're making the best use of your compute resources and providing the best possible experience for the users of your application.

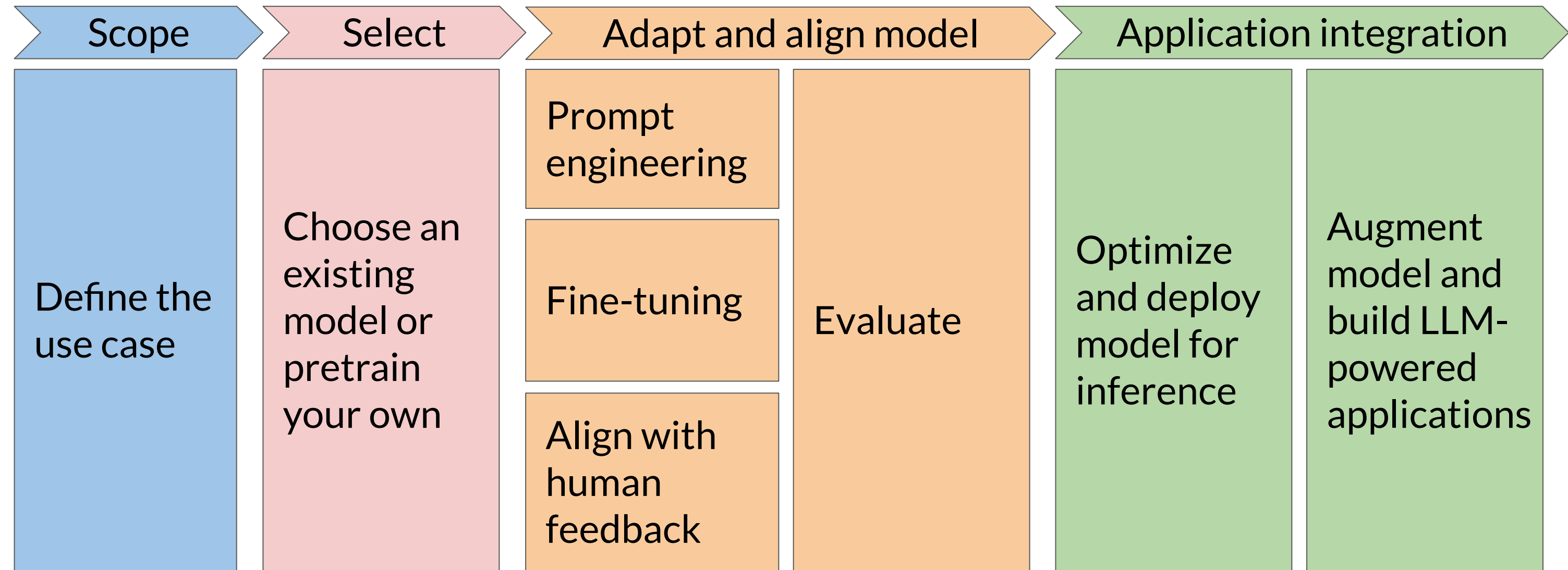
The last but very important step is to consider any additional infrastructure that your application will require to work well. There are some fundamental limitations of LLMs that can be difficult to overcome through training alone like their tendency to invent information when they don't know an answer, or their limited ability to carry out complex reasoning and mathematics. In the last part of this course, you'll learn some powerful techniques that you can use to overcome these limitations.

- Amazon SageMaker is the Jupyter based IDE
- torchdata library used to load the data and with some other aspects of Pytorch as well
- transformers library is used to play with LLMs that is the open source library created by hugging face
- datasets is the library within transformers that have a lot of dataset that can be used for training, fine tuning or experimentation.
- We are going to FLAN-T5 LLM model for 1 lab
- Baseline Human summary is the labeled output(this is not something LLM has generated but some human has done it manually to make a dataset)
- Prompt engineering, zero-shot, one-shot, few shot is almost always the first step when you're trying to learn the language model that you've been handed and dataset.
- Typically, in my experience, above five or six shots, so full prompt and then completions, you really don't gain much after that. Either the model can do it or it can't do it and going about five or six

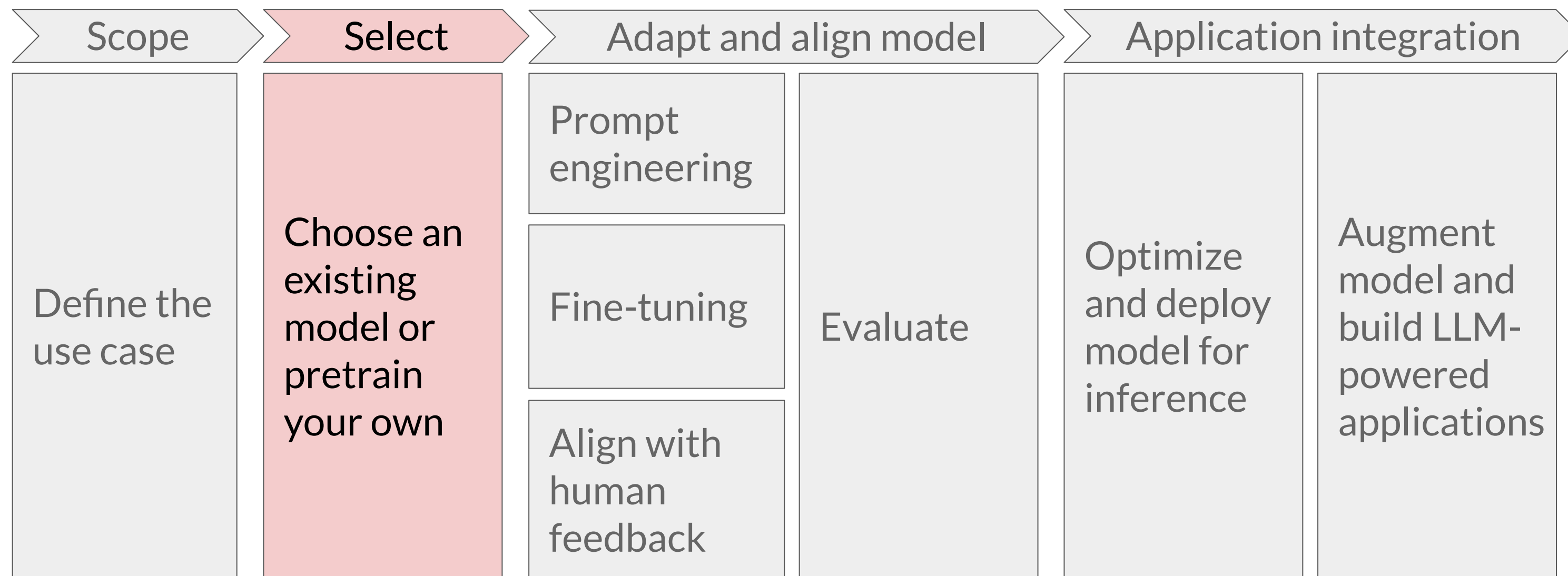
Pre-training and scaling laws



Generative AI project lifecycle

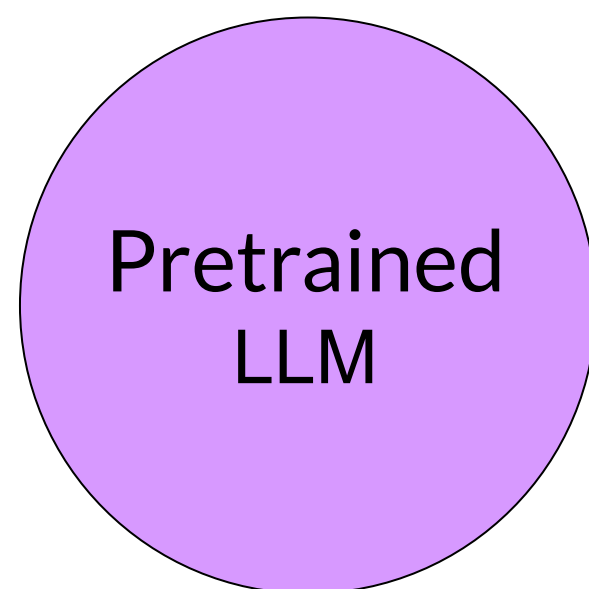


Generative AI project lifecycle

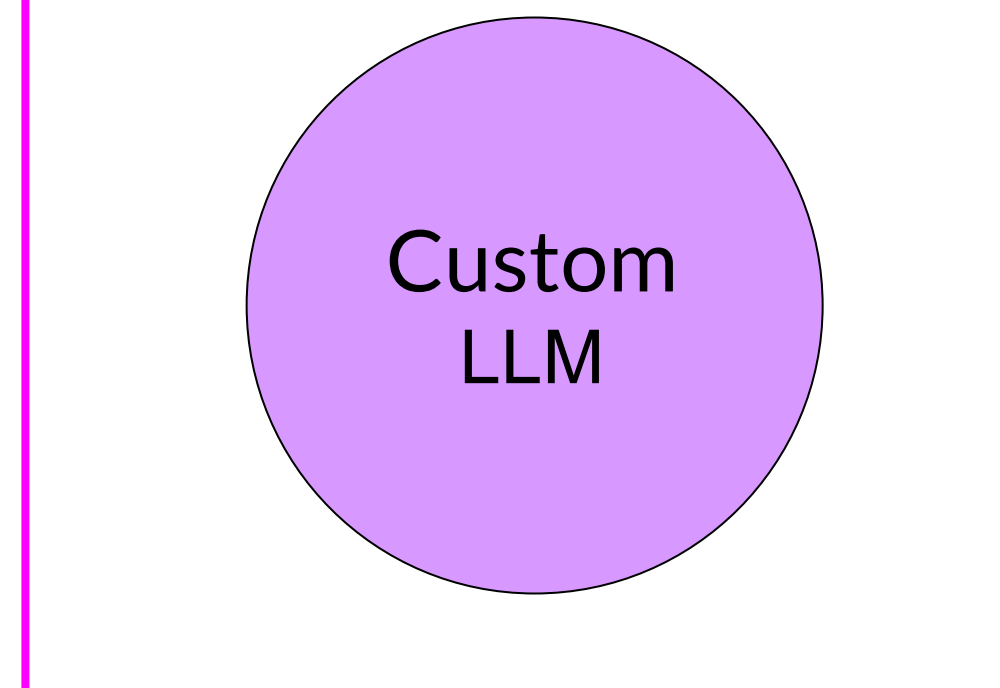


Considerations for choosing a model

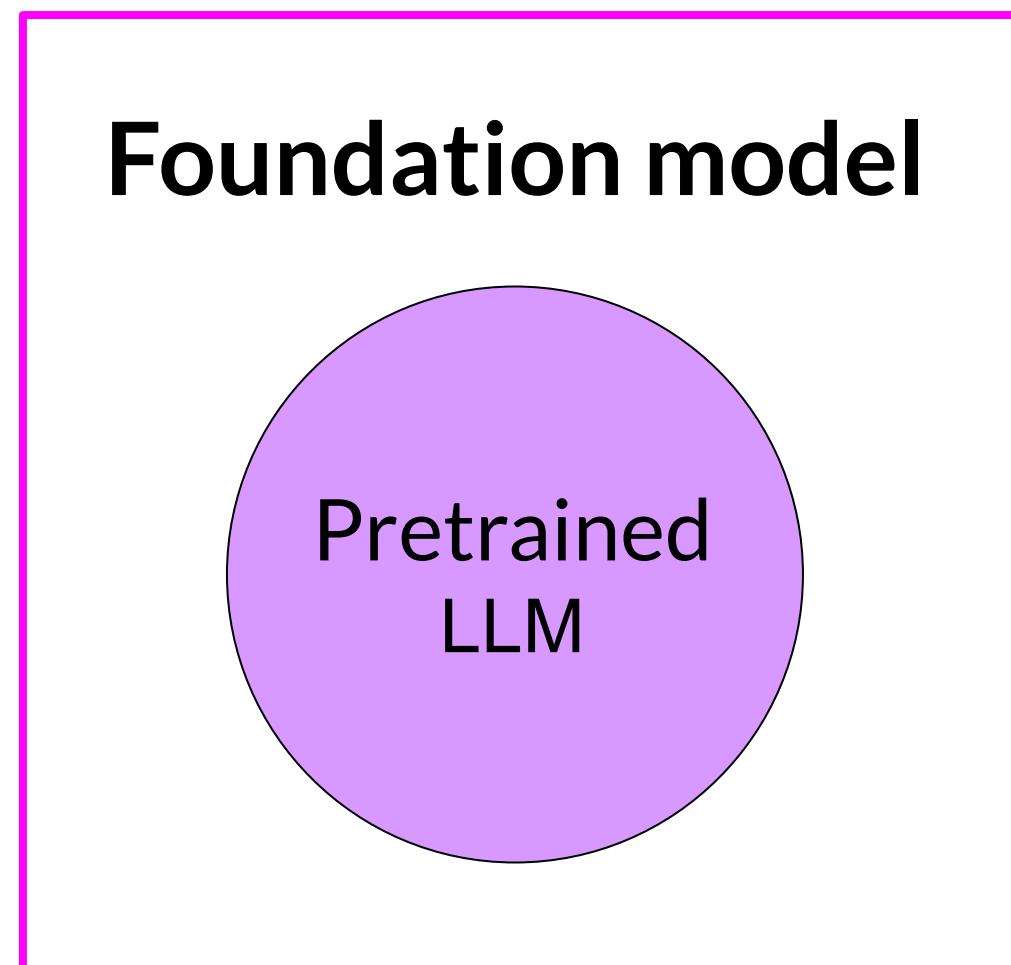
Foundation model



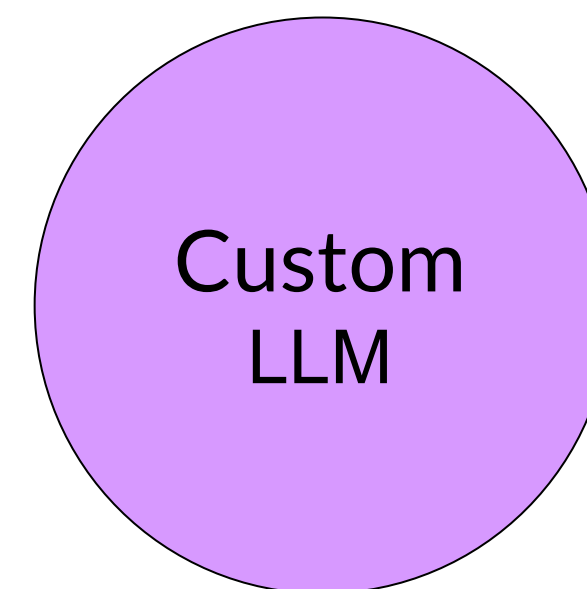
Train your own model



Considerations for choosing a model



Train your own model



Model hubs

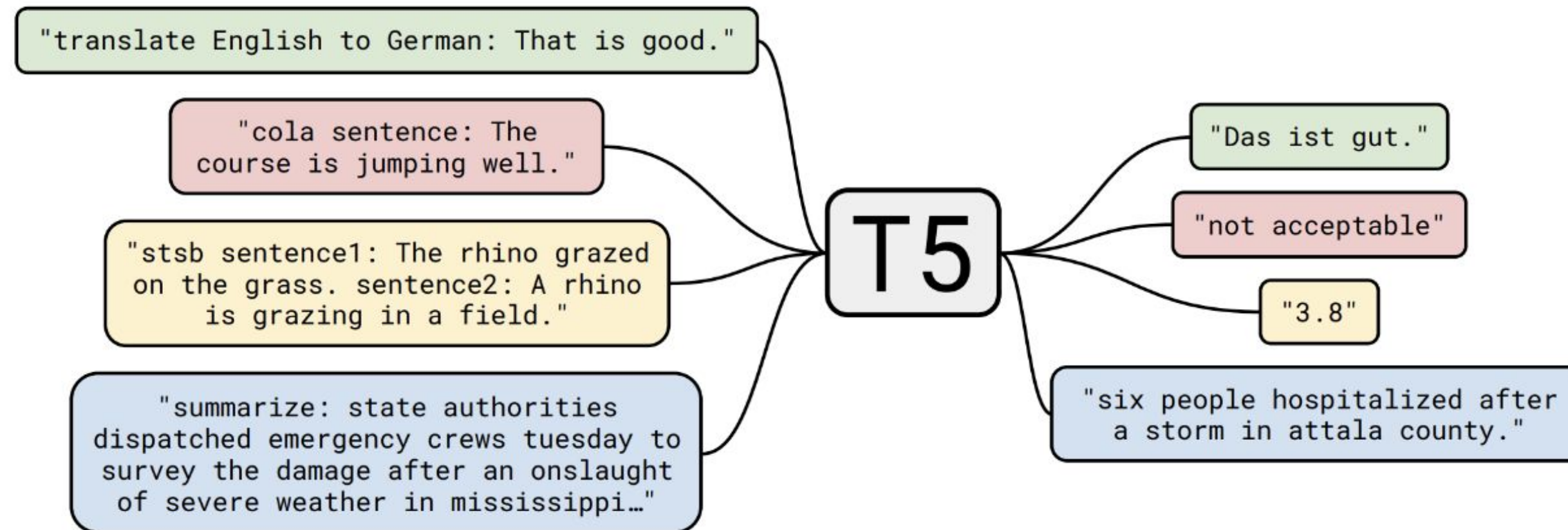
The developers of some of the major frameworks for building generative AI applications like Hugging Face and PyTorch, have curated hubs where you can browse these models(LLM models).

Model Card for T5 Large

A really useful feature of these hubs is the inclusion of model cards, that describe important details including the best use cases for each model, how it was trained, and known limitations.

Table of Contents

1. Model Details
2. Uses
3. Bias, Risks, and Limitations
4. Training Details
5. Evaluation



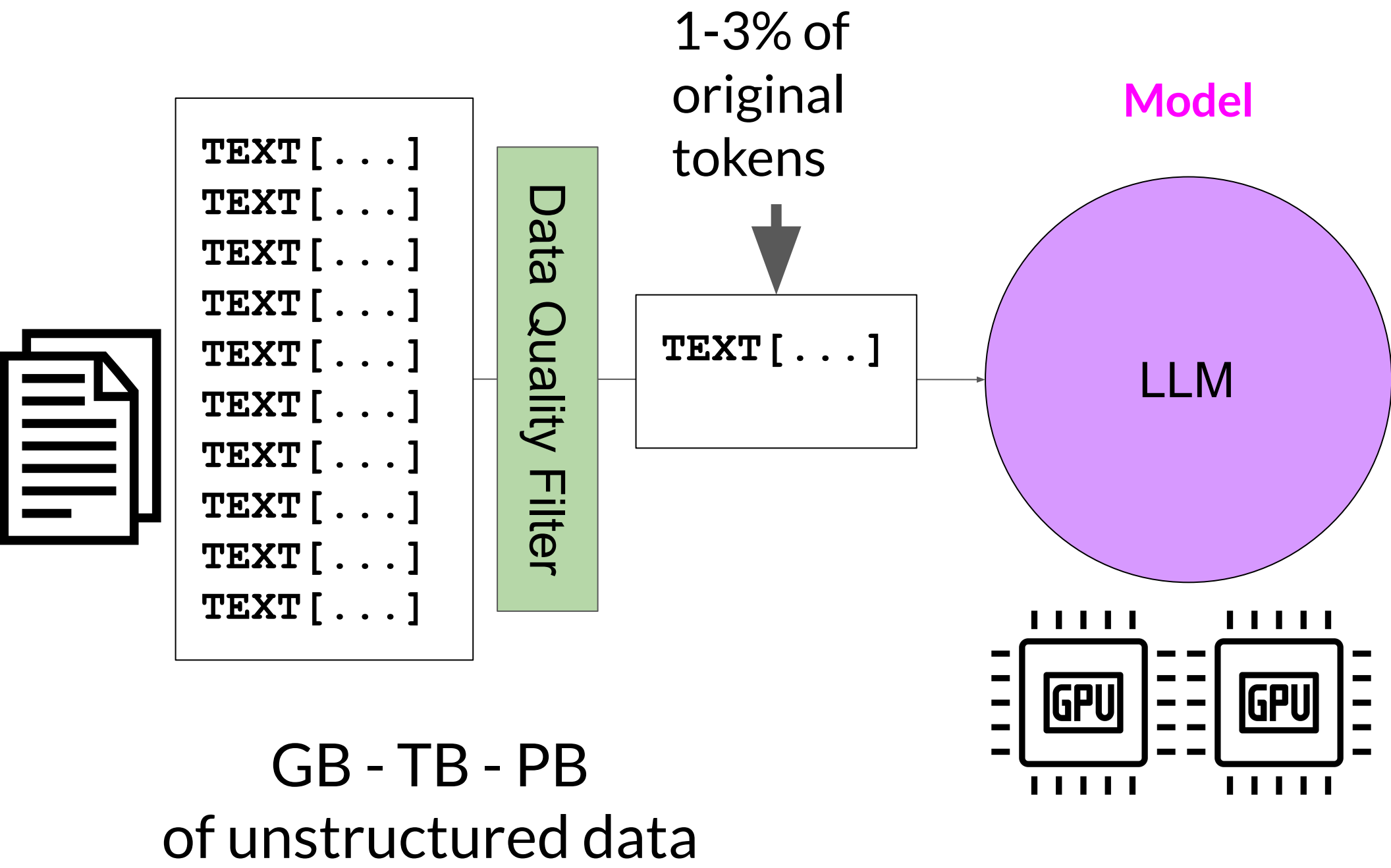
To help us develop intuition about which model to use for a particular task, let's take a closer look at how large language models are trained. With this knowledge in hand, you'll find it easier to navigate the model hubs and find the best model for your use case.

Model architectures and pre-training objectives

To begin, let's take a high-level look at the initial training process for LLMs. This phase is often referred to as pre-training.

LLM pre-training at a high level

[Read this sticky note](#)



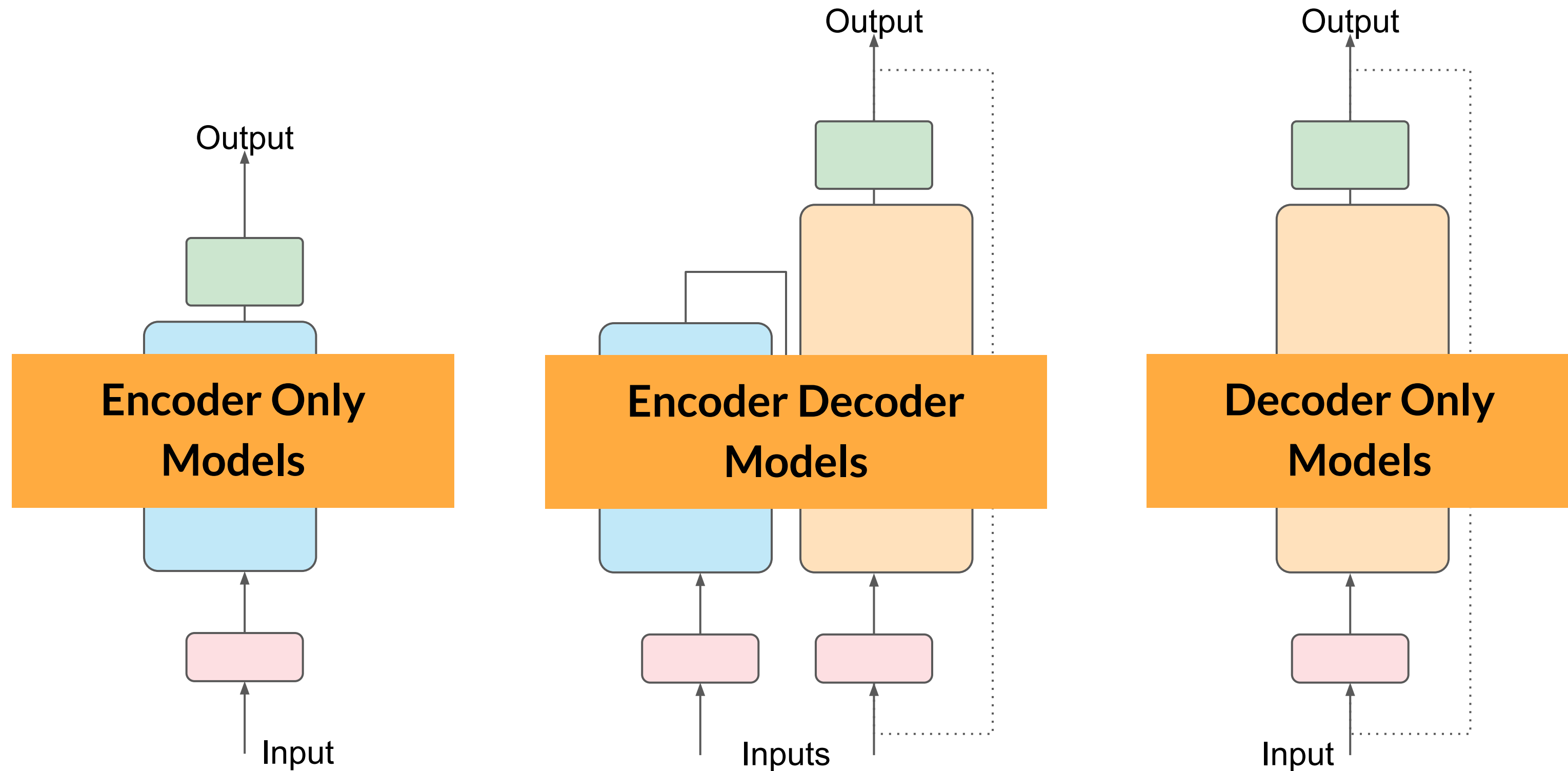
GB - TB - PB
of unstructured data

Token String	Token ID	Embedding / Vector Representation
'_The '	37	[-0.0513, -0.0584, 0.0230, ...]
'_teacher '	3145	[-0.0335, 0.0167, 0.0484, ...]
'_teaches '	11749	[-0.0151, -0.0516, 0.0309, ...]
'_the '	8	[-0.0498, -0.0428, 0.0275, ...]
'_student '	1236	[-0.0460, 0.0031, 0.0545, ...]
...

Vocabulary

Transformers

This was already explained in the earlier task



Autoencoding models

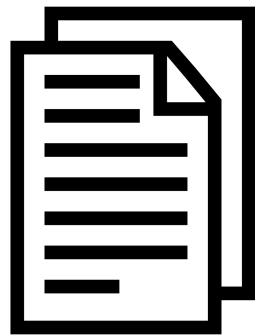
Encoder only Model for Transformers are also known as Autoencoding models and they are pre-trained using masked language modeling.

Here, tokens in the input sequence are randomly masked, and the training objective is to predict the masked tokens in order to reconstruct the original sentence. This is also called a denoising objective.

Masked Language Modeling (MLM)

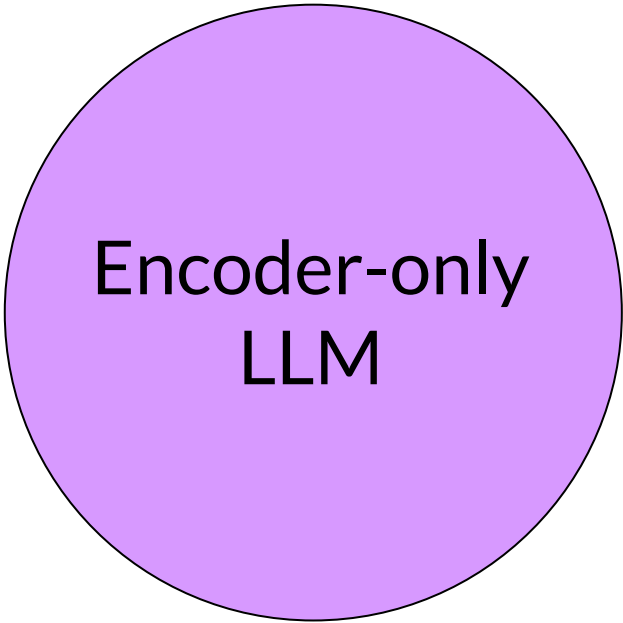
The	teacher	<MASK>	the	student
-----	---------	--------	-----	---------

Original text



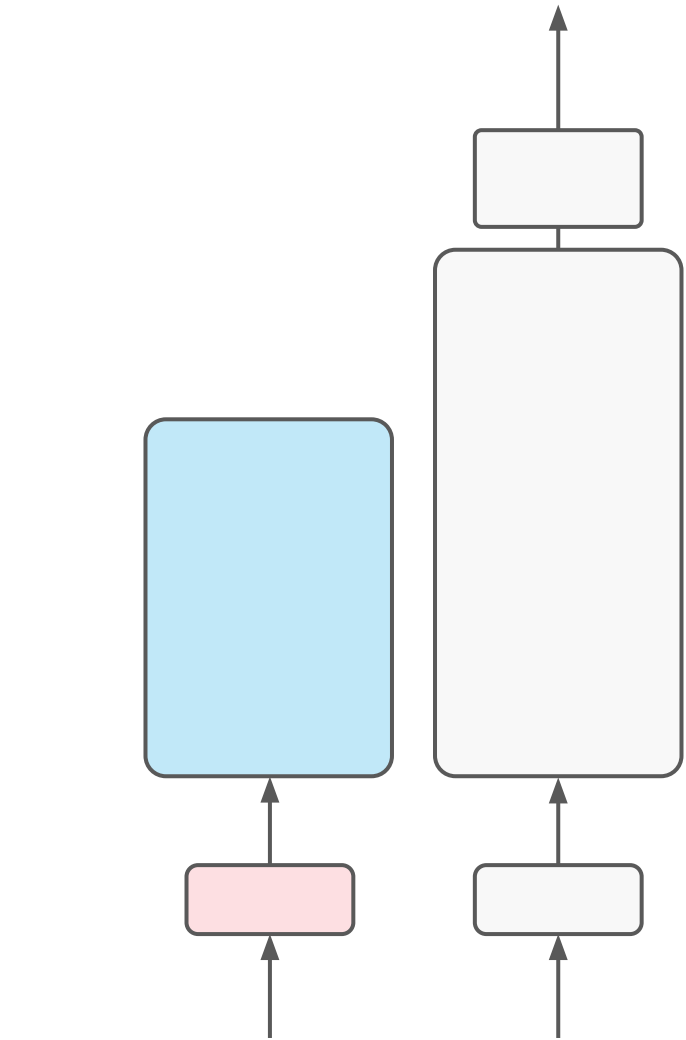
The teacher
teaches the
student.

[...]



Objective: Reconstruct text ("denoising")

The	teacher	teaches	the	student
-----	---------	---------	-----	---------



Bidirectional context

Autoencoding models are the bi-directional representations of the input sequence, meaning that the model has an understanding of the full context of a token and not just of the words that come before.

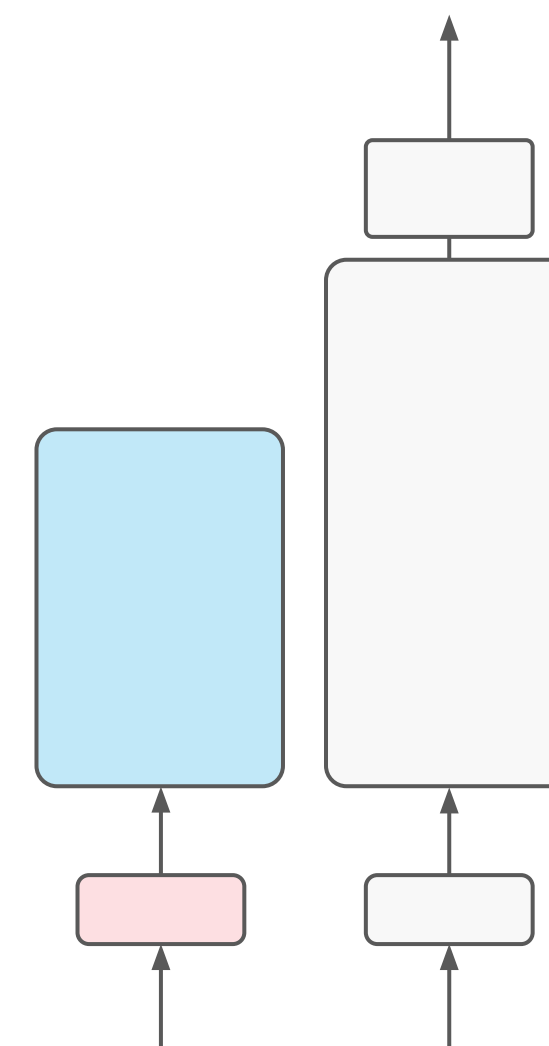
Autoencoding models

Good use cases:

- Sentiment analysis
- Named entity recognition
- Word classification

Example models:

- BERT
- ROBERTA




Autoregressive models

Decoder only is also called Autoregressive models which are pre-trained using Casual language modelling (CLM).

Here, the training objective is to predict the next token based on the previous sequence of tokens. Predicting the next token is sometimes called full language modeling by researchers.

Causal Language Modeling (CLM)

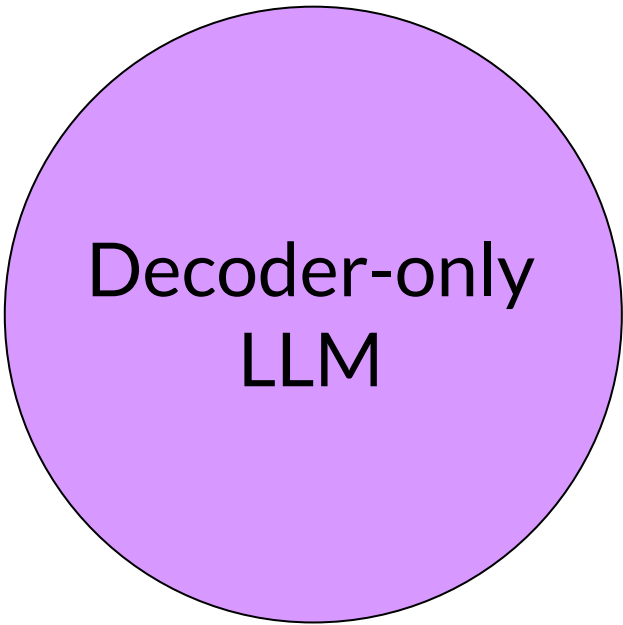


Original text

The teacher teaches the student.
[...]

Decoder-based autoregressive models, mask the input sequence and can only see the input tokens leading up to the token in question. The model has no knowledge of the end of the sentence. The model then iterates over the input sequence one by one to predict the following token. In contrast to the encoder architecture, this means that the context is unidirectional. By learning to predict the next token from a vast number of examples, the model builds up a statistical representation of language.

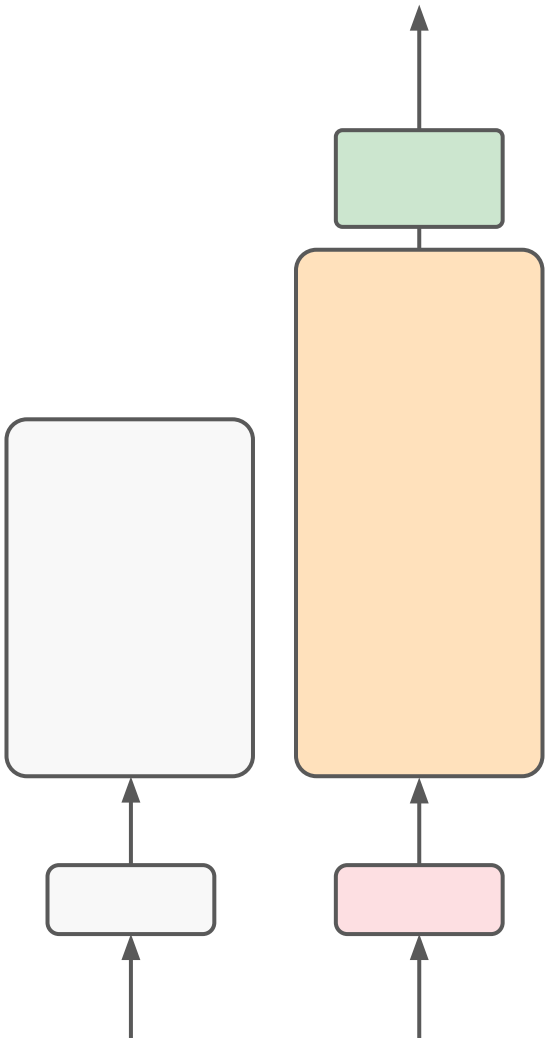
The	tea ch er	?
-----	----------------------	---



Objective: Predict next token

The	teacher	teaches
-----	----------------	----------------

Unidirectional context



Autoregressive models

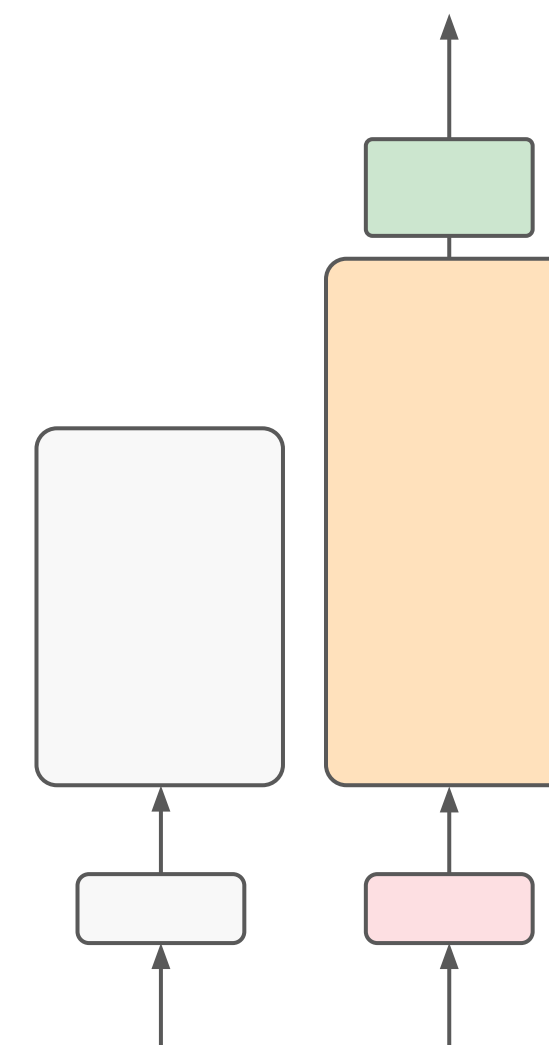
Good use cases:

- Text generation
- Other emergent behavior
 - Depends on model size

Decoder-only models are often used for text generation, although larger decoder-only models show strong zero-shot inference abilities, and can often perform a range of tasks well.

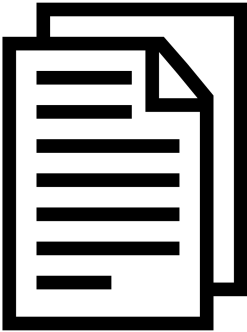
Example models:

- GPT
- BLOOM



Sequence-to-sequence models

Variant of LLM that uses the both Encoder and Decoder is know as Sequence-to-sequence models which are pre-trained using Span Corruption that masks the random sequence of the input tokens. These masked sequence of tokens are then replaced with Sentinel token which is represented by x below



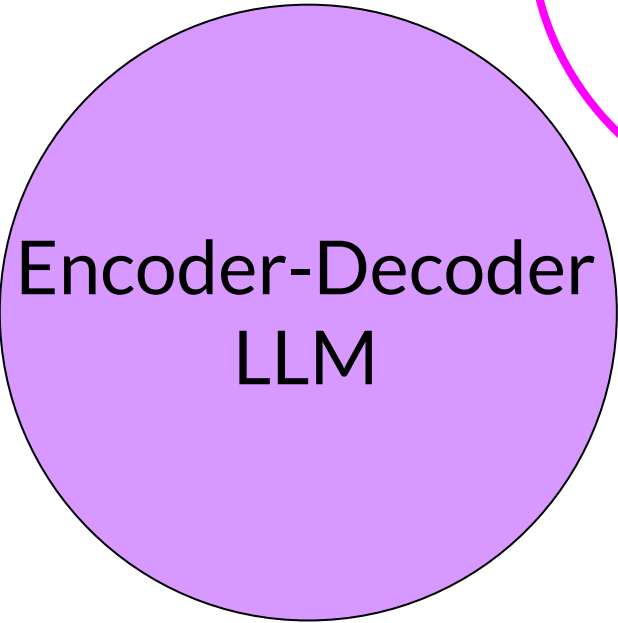
Original text

The teacher teaches the student.

[...]

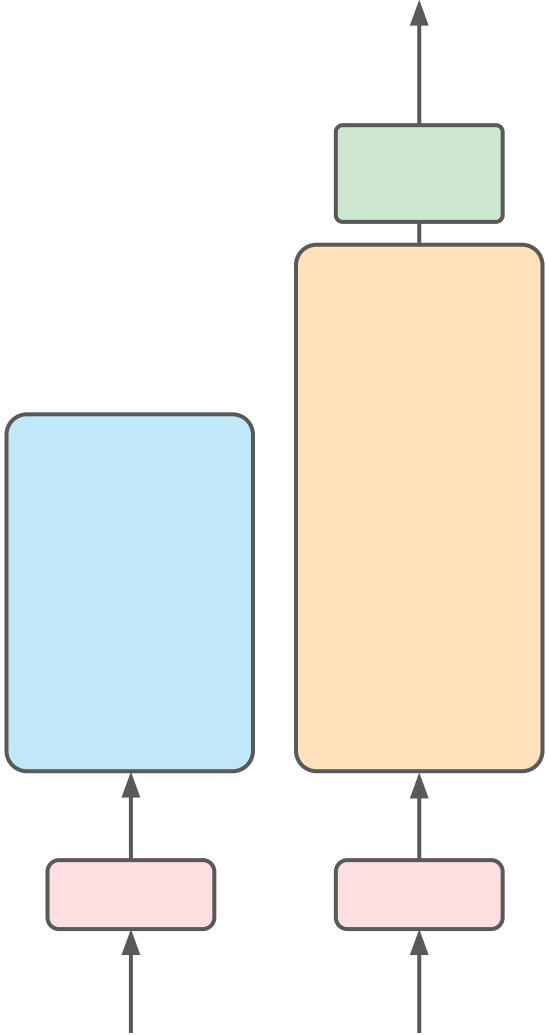
Span Corruption

The	teacher	<MASK>	<MASK>	student
The	teacher	<X>		student



Sentinel token

Sentinel tokens are special tokens added to the vocabulary, but do not correspond to any actual word from the input text.



The decoder is then tasked with reconstructing the mask token sequences auto-regressively. The output is the Sentinel token followed by the predicted tokens.

Objective: Reconstruct span

<x>	teaches	the
-----	---------	-----

Sequence-to-sequence models

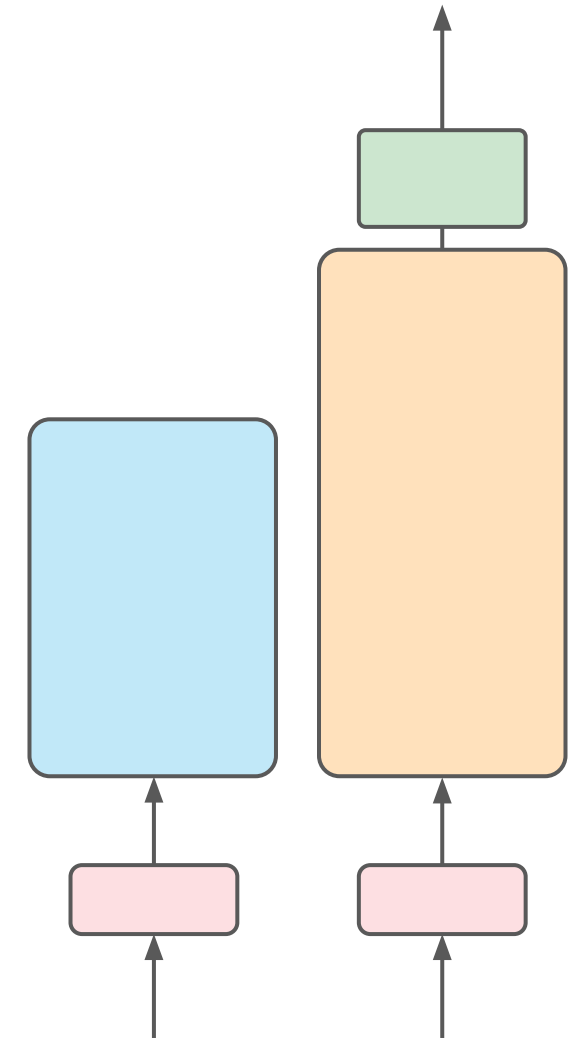
Good use cases:

- Translation
- Text summarization
- Question answering

They are generally useful in cases where you have a body of texts as both input and output.

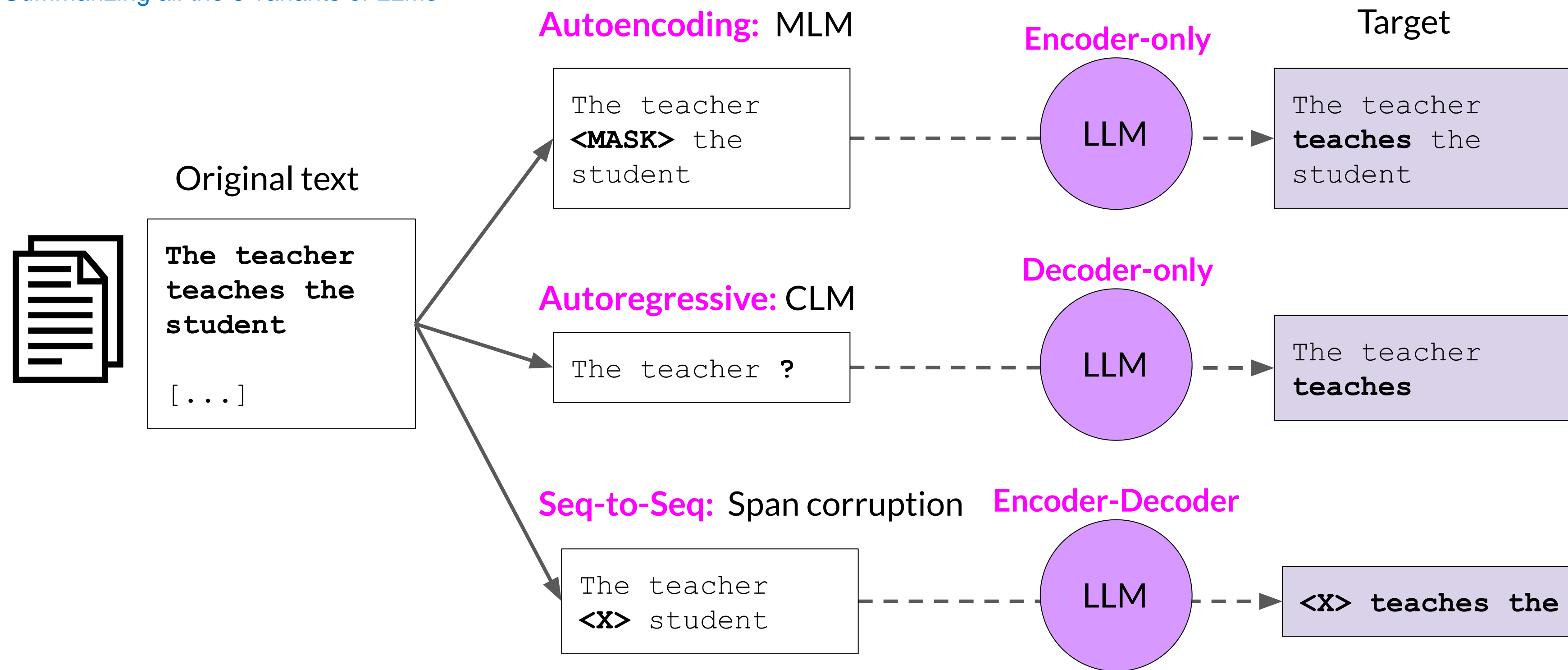
Example models:

- T5
- BART



Model architectures and pre-training objectives

Summarizing all the 3 variants of LLMs



The significance of scale: task ability

BERT*
110M

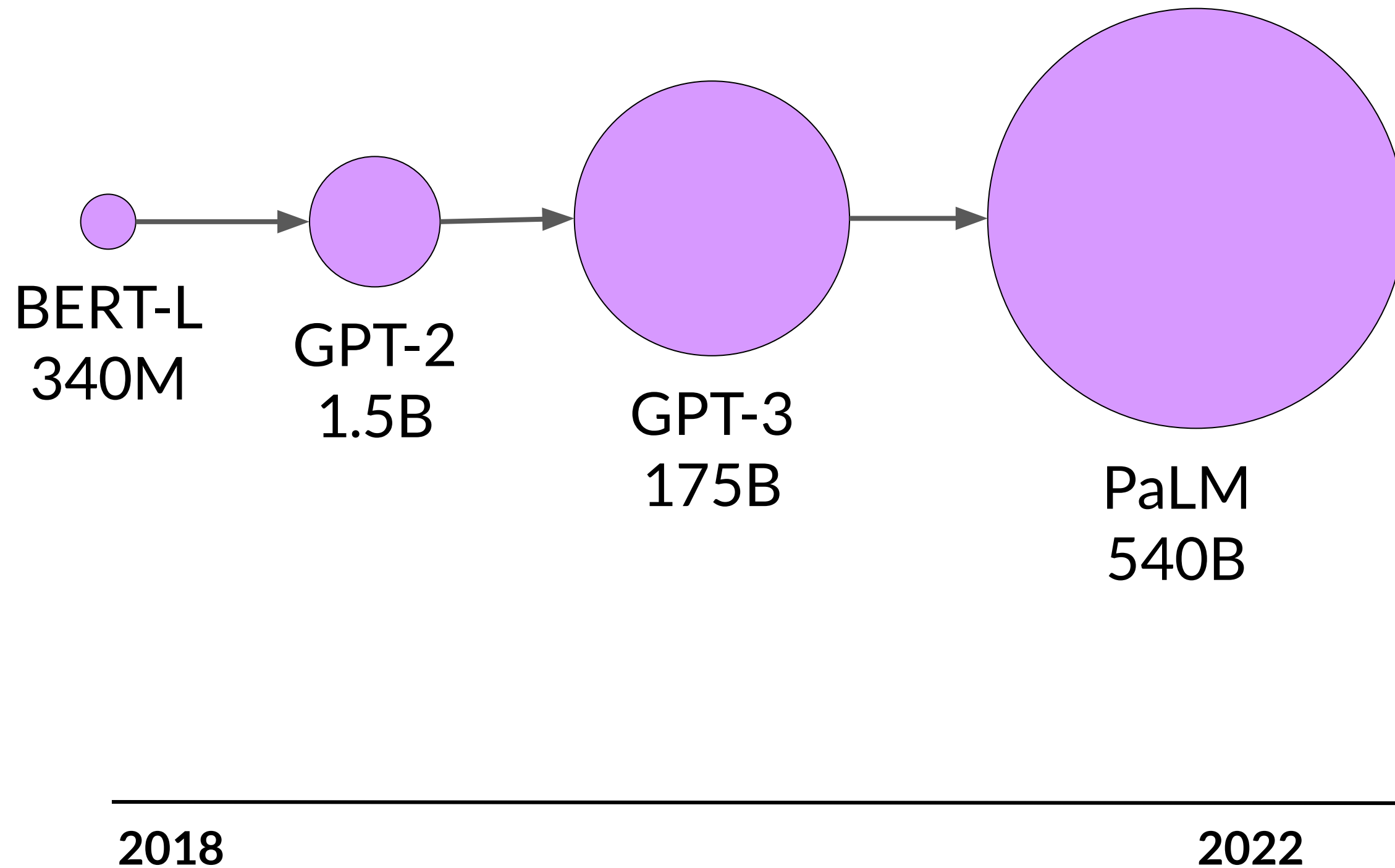
BLOOM
176B



Researchers have found that the larger a model, the more likely it is to work as you needed to without additional in-context learning or further training.

*Bert-base

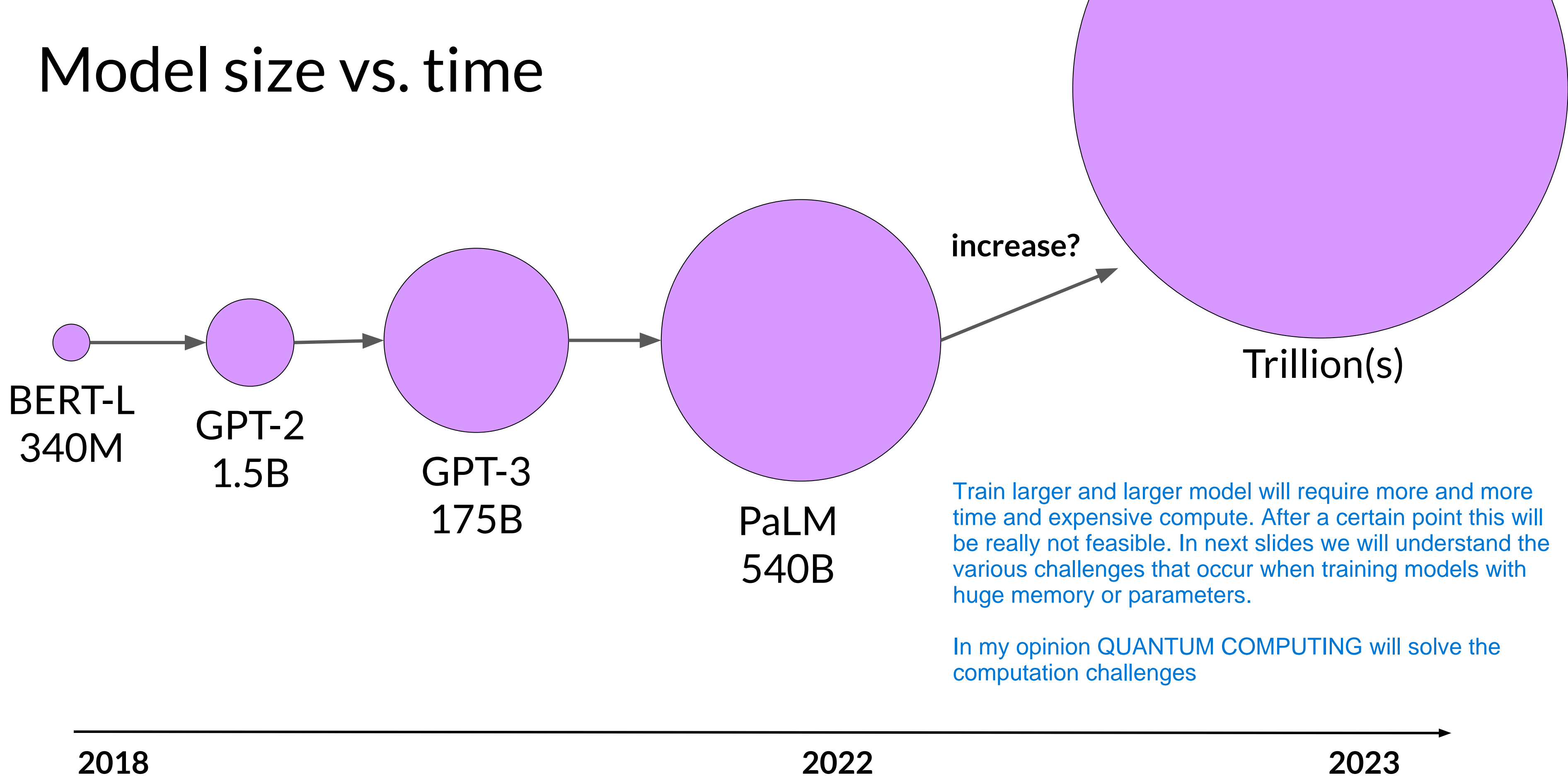
Model size vs. time



Growth powered by:

- Introduction of transformer
- Access to massive datasets
- More powerful compute resources

Model size vs. time



You'll encounter these out-of-memory issues when LLM model are trained using Nvidia GPUs because most LLMs are huge, and require a ton of memory to store and train all of their parameters

Computational challenges

```
OutOfMemoryError: CUDA out of memory.
```



CUDA, short for Compute Unified Device Architecture, is a collection of libraries and tools developed for Nvidia GPUs. Libraries such as PyTorch and TensorFlow use CUDA to boost performance on metrics multiplication and other operations common to deep learning.

Approximate GPU RAM needed to store 1B parameters

1 parameter = 4 bytes (32-bit float)

1B parameters = 4×10^9 bytes = 4GB

A parameter is generally a Real number representation that occupies 4 bytes of memory location

Say a LLM based model needs to be trained over 1 billion parameters. Then Memory needs to be train such model will be 4GB

**4GB @ 32-bit
full precision**

four gigabyte of
GPU RAM at
32-bit full precision

Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, <https://github.com/facebookresearch/bitsandbytes>

Additional GPU RAM needed to train 1B parameters

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter
Adam optimizer (2 states)	+8 bytes per parameter
Gradients	+4 bytes per parameter
Activation and temp memory (variable size)	+8 bytes per parameter (high-end estimate)
TOTAL	= 4bytes per parameter +20 extra bytes per parameter

Read this sticky note

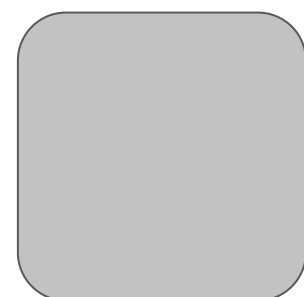
~20 extra bytes per parameter

Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, <https://github.com/facebookresearch/bitsandbytes>

Approximate GPU RAM needed to train 1B-params

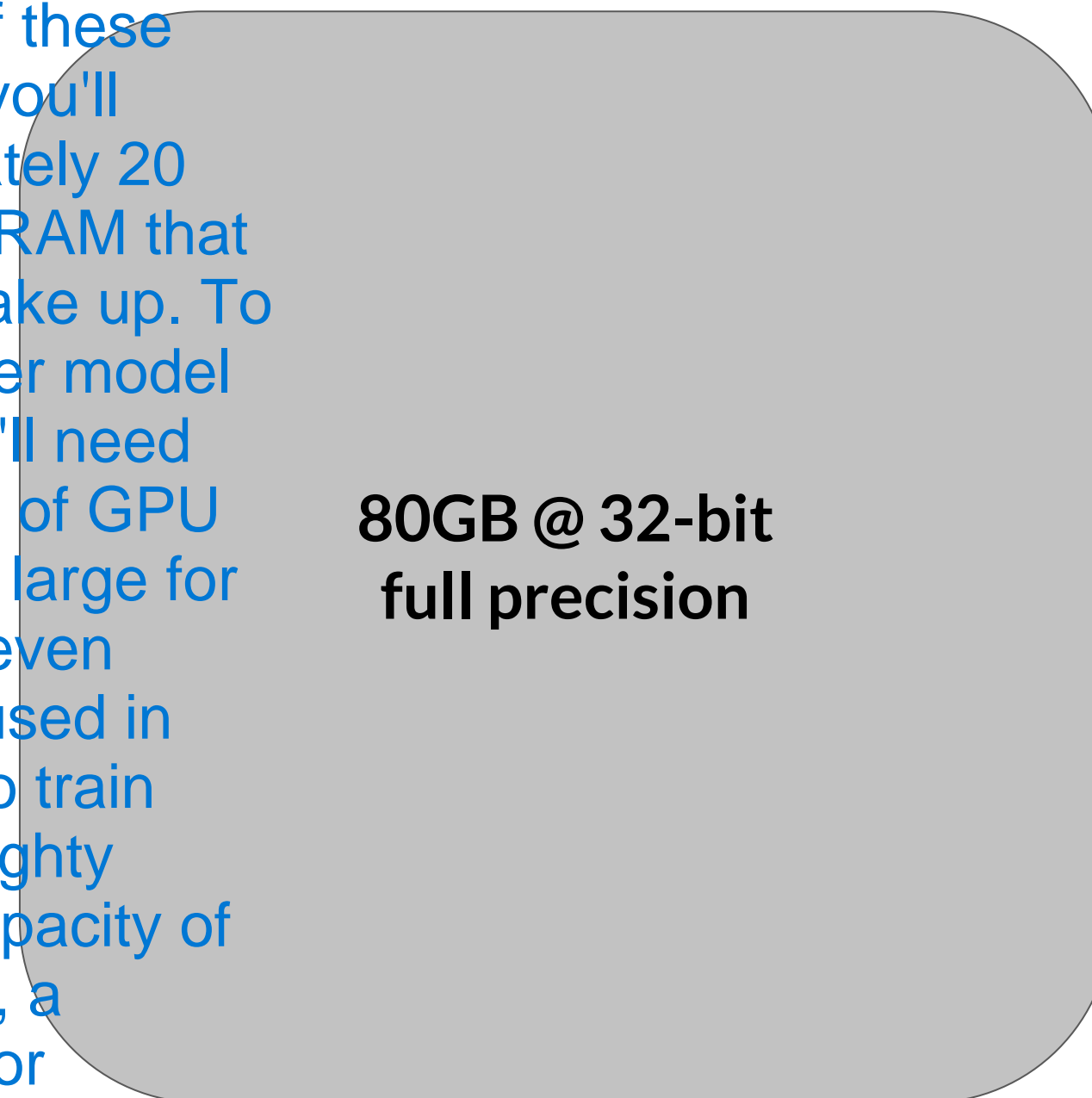
Memory needed to store model

**4GB @ 32-bit
full precision**



Memory needed to train model

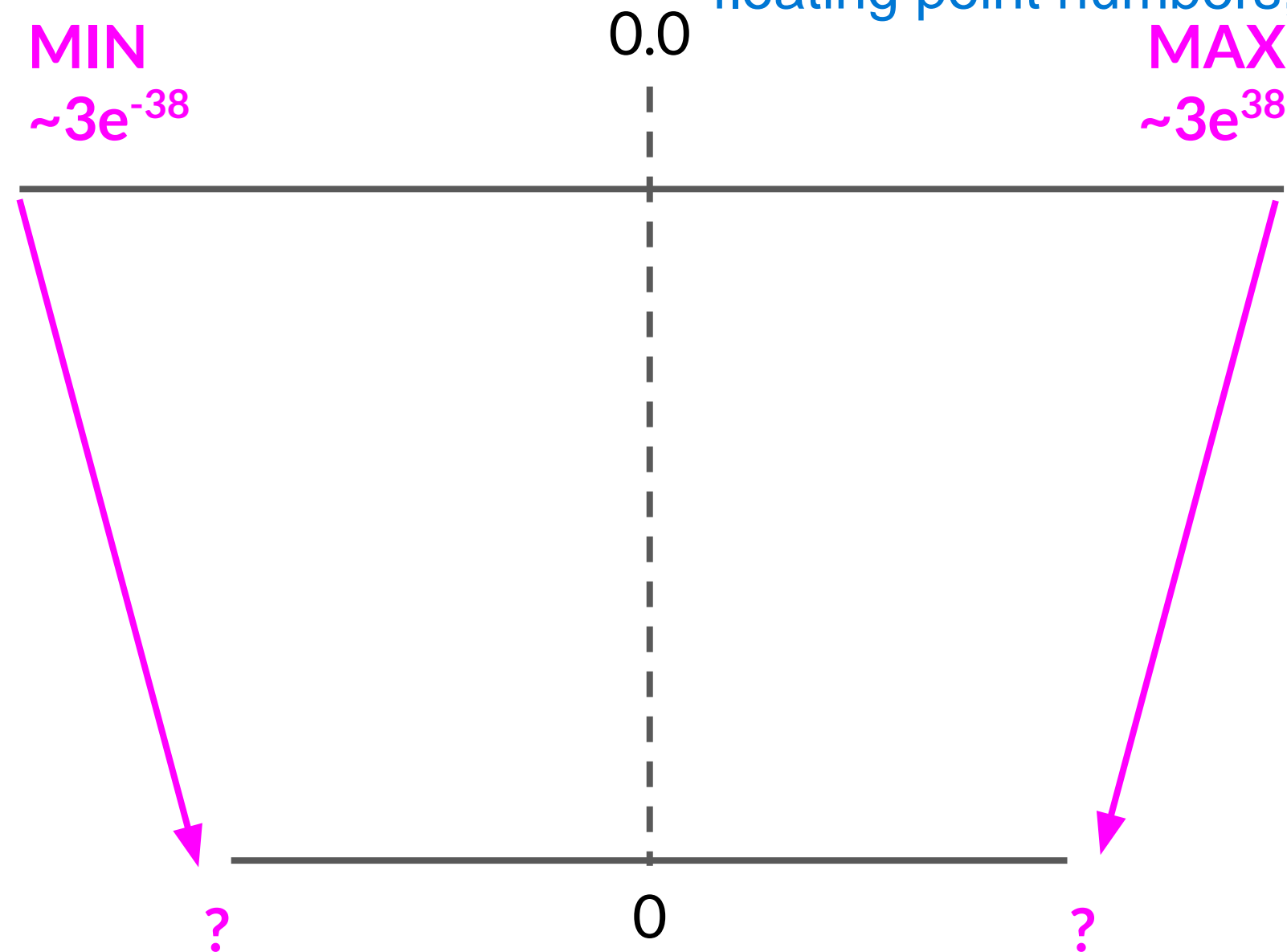
**80GB @ 32-bit
full precision**



In fact, to account for all of these overhead during training, you'll actually require approximately 20 times the amount of GPU RAM that the model weights alone take up. To train a one billion parameter model at 32-bit full precision, you'll need approximately 80 gigabyte of GPU RAM. This is definitely too large for consumer hardware, and even challenging for hardware used in data centers, if you want to train with a single processor. Eighty gigabyte is the memory capacity of a single Nvidia A100 GPU, a common processor used for machine learning tasks in the Cloud.

Quantization

- What options do you have to reduce the memory required for training? One technique that you can use to reduce the memory is called quantization.
- The main idea here is that you reduce the memory required to store the weights of your model by reducing their precision from 32-bit floating point numbers to 16-bit floating point numbers, or eight-bit integer numbers.



FP32

32-bit floating point

Range: (Range of FP32)
From $\sim 3e^{-38}$ to $\sim 3e^{38}$

FP16 | BFLOAT16 | INT8

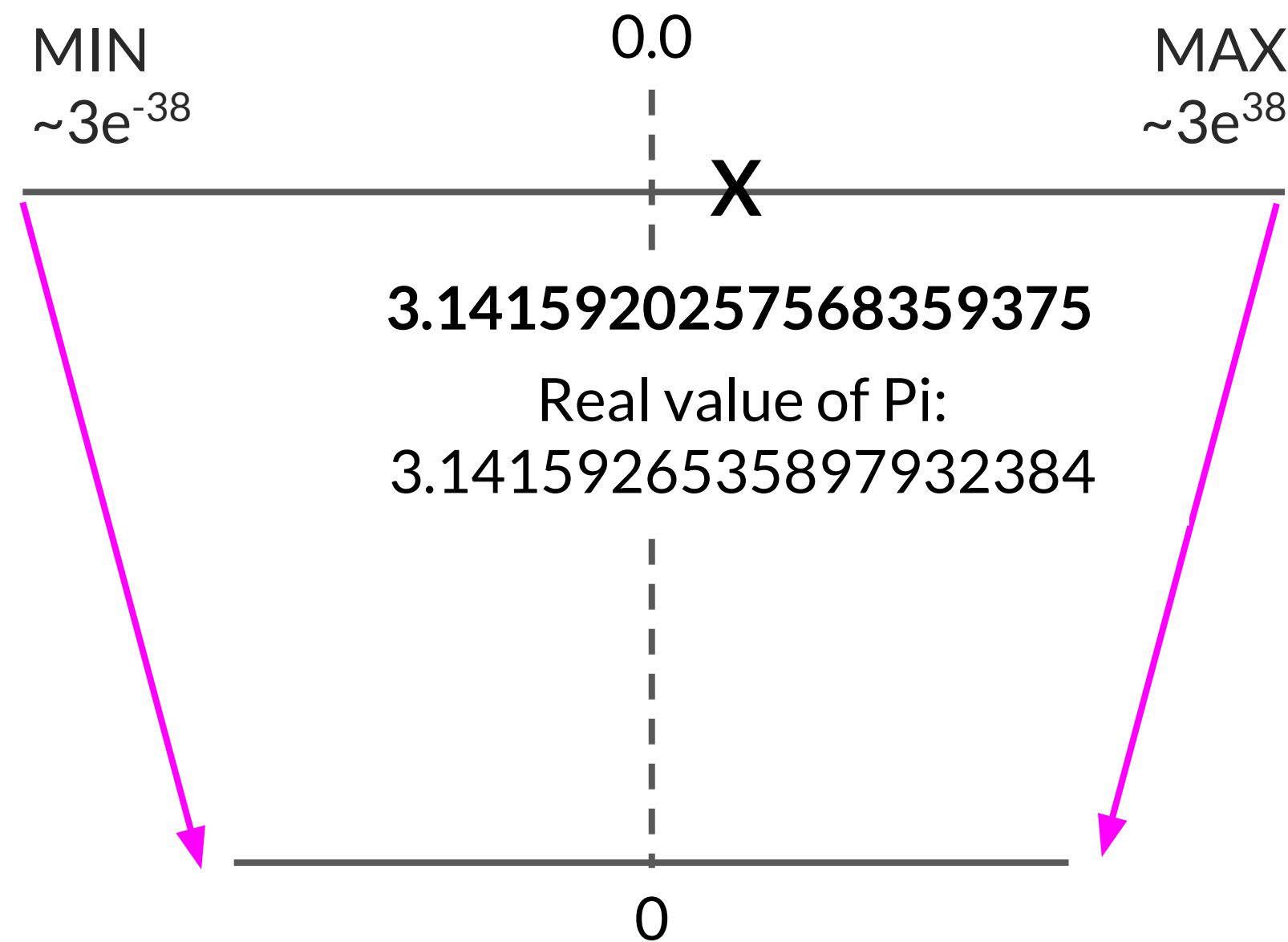
16-bit floating point | 8-bit integer

- Quantization statistically projects the original 32-bit floating point numbers into a lower precision space, using scaling factors calculated based on the range of the original 32-bit floating point numbers.

Quantization: FP32

Let's understand Quantization with the help of an example:

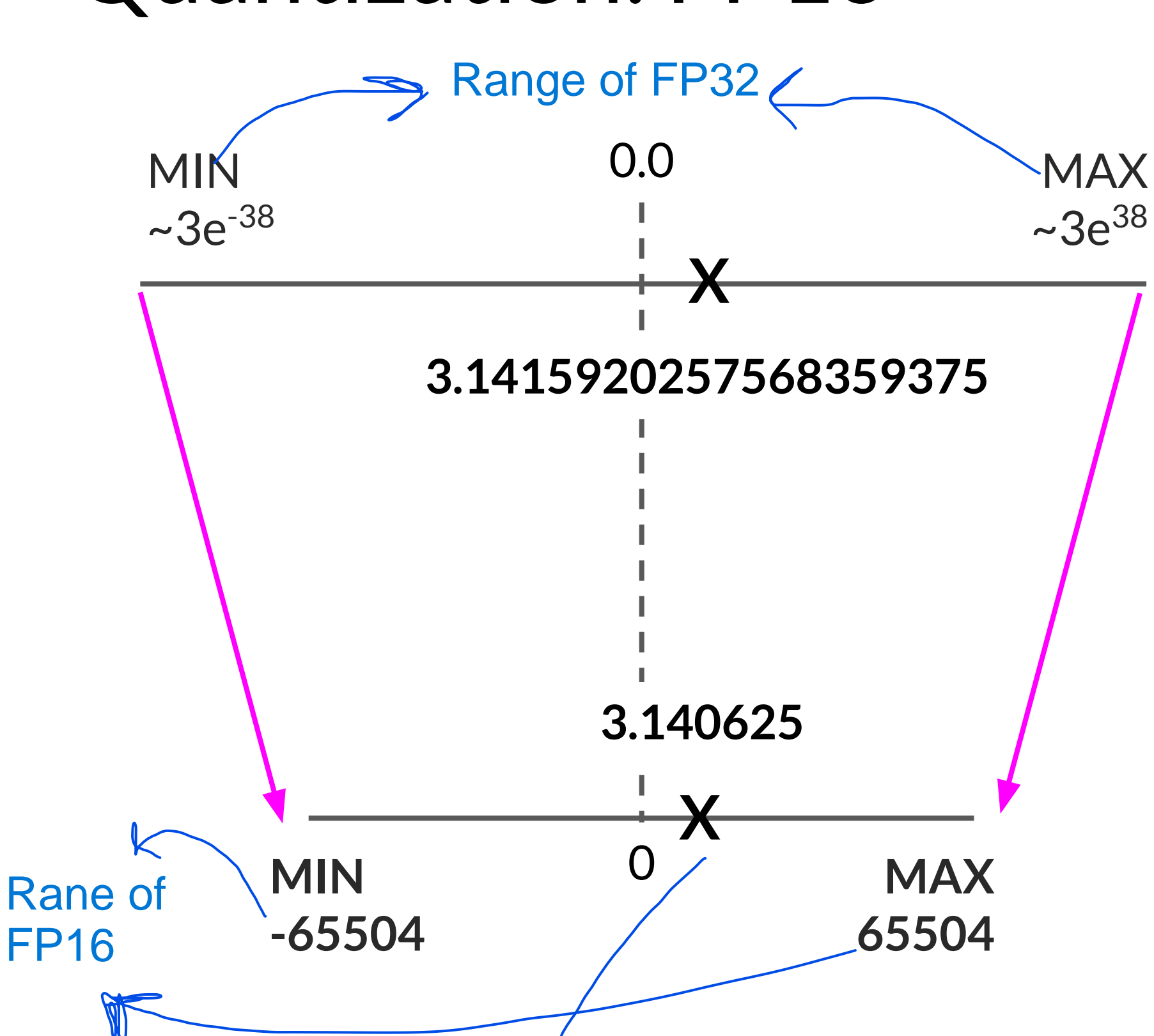
Let's store Pi: 3.141592



FP32

0	10000000	1001001000011111011000
<hr/>		
Sign	Exponent	Fraction
1 bit	8 bits	23 bits
<hr/>		
Mantissa / Significand = Precision		

Quantization: FP16



Let's store Pi: 3.141592

FP32	4 bytes memory	
0	10000000	1001001000011111011000
Sign 1 bit	Exponent 8 bits	Fraction 23 bits

FP16	2 bytes memory	
0	10000	1001001000
Sign 1 bit	Exponent 5 bits	Fraction 10 bits

In this example we can clearly observe that Quantization FP16 brought our memory requirement to half

Notice that you lose some precision with this projection. There are only six places after the decimal point now.

Quantization: BFLOAT16

[← Read this sticky note](#)

Let's store Pi: 3.141592

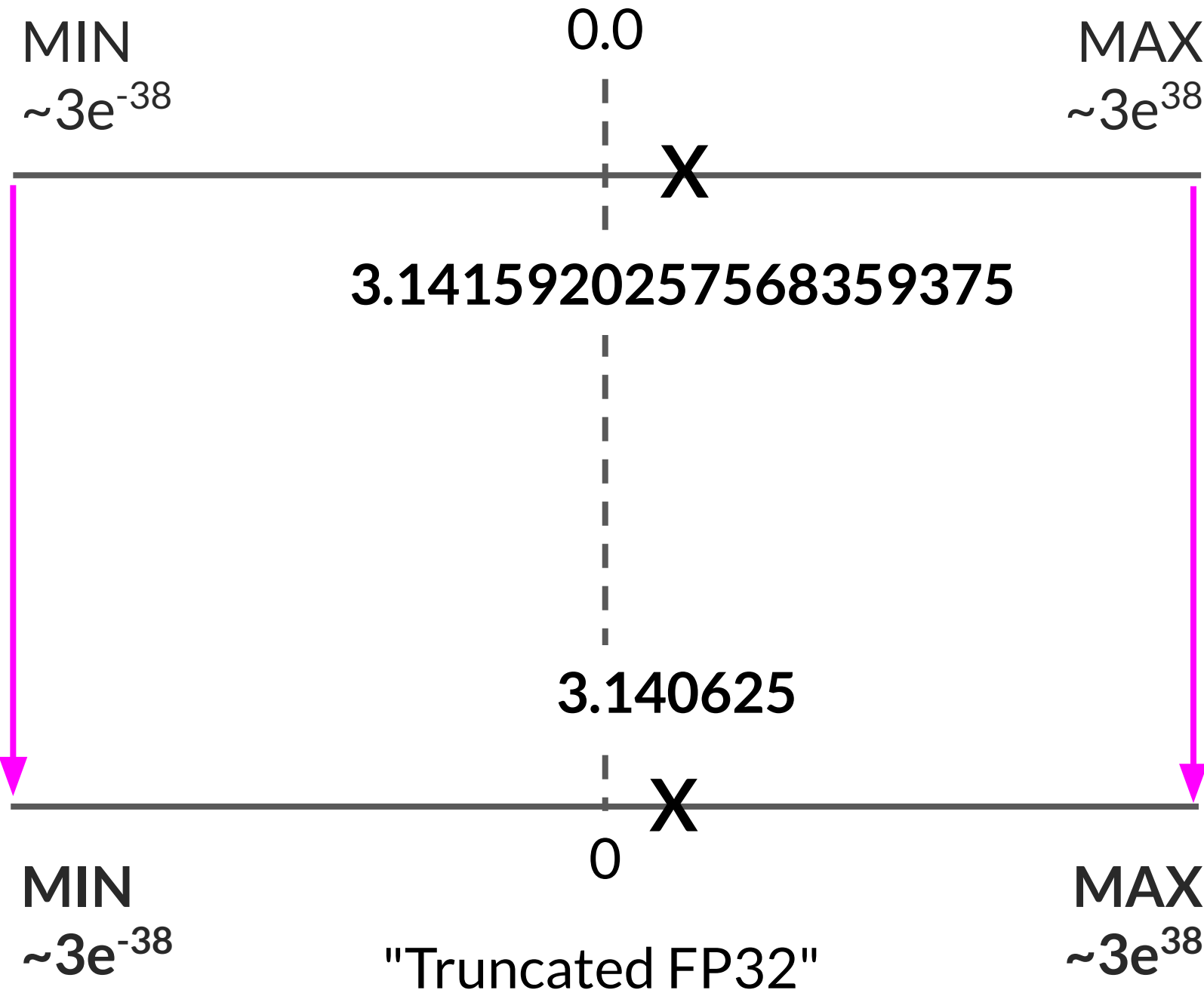
FP32 4 bytes memory

0	10000000	1001001000011111011000
<hr/>		
Sign 1 bit	Exponent 8 bits	Fraction 23 bits

BFLOAT16 | BF16

2 bytes memory

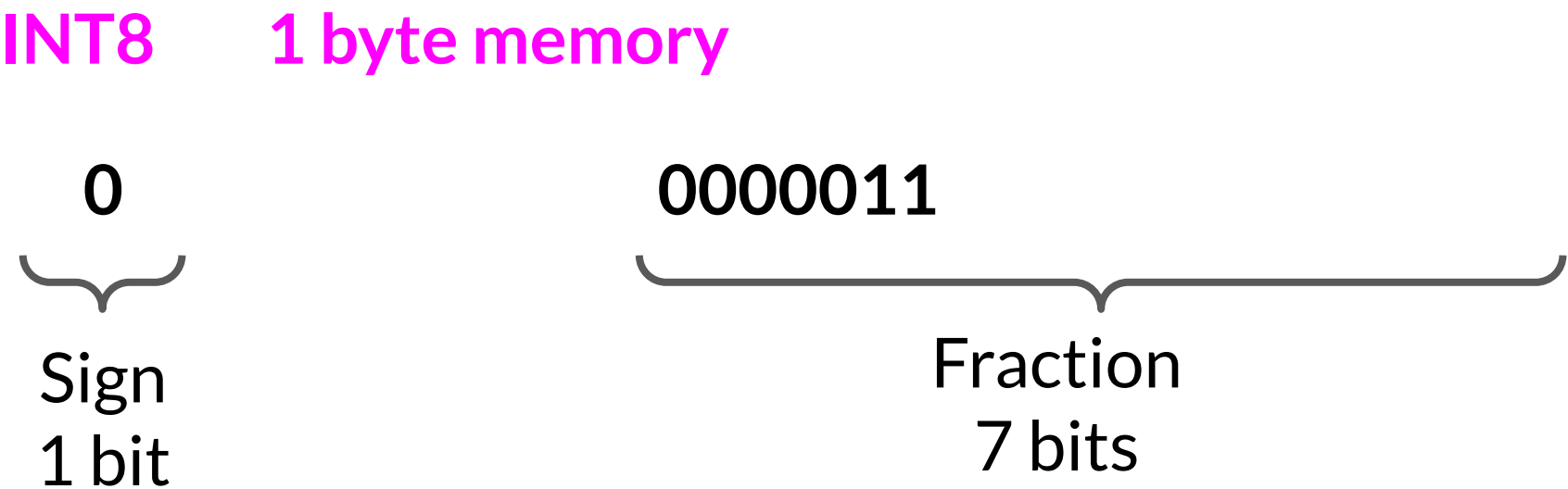
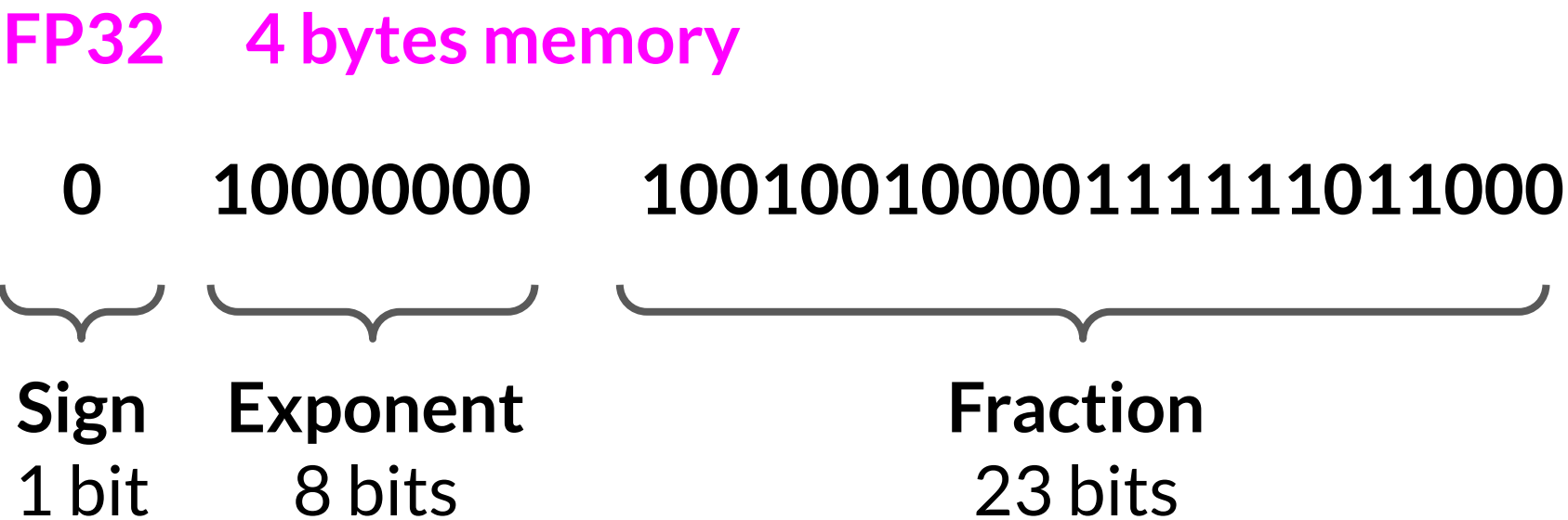
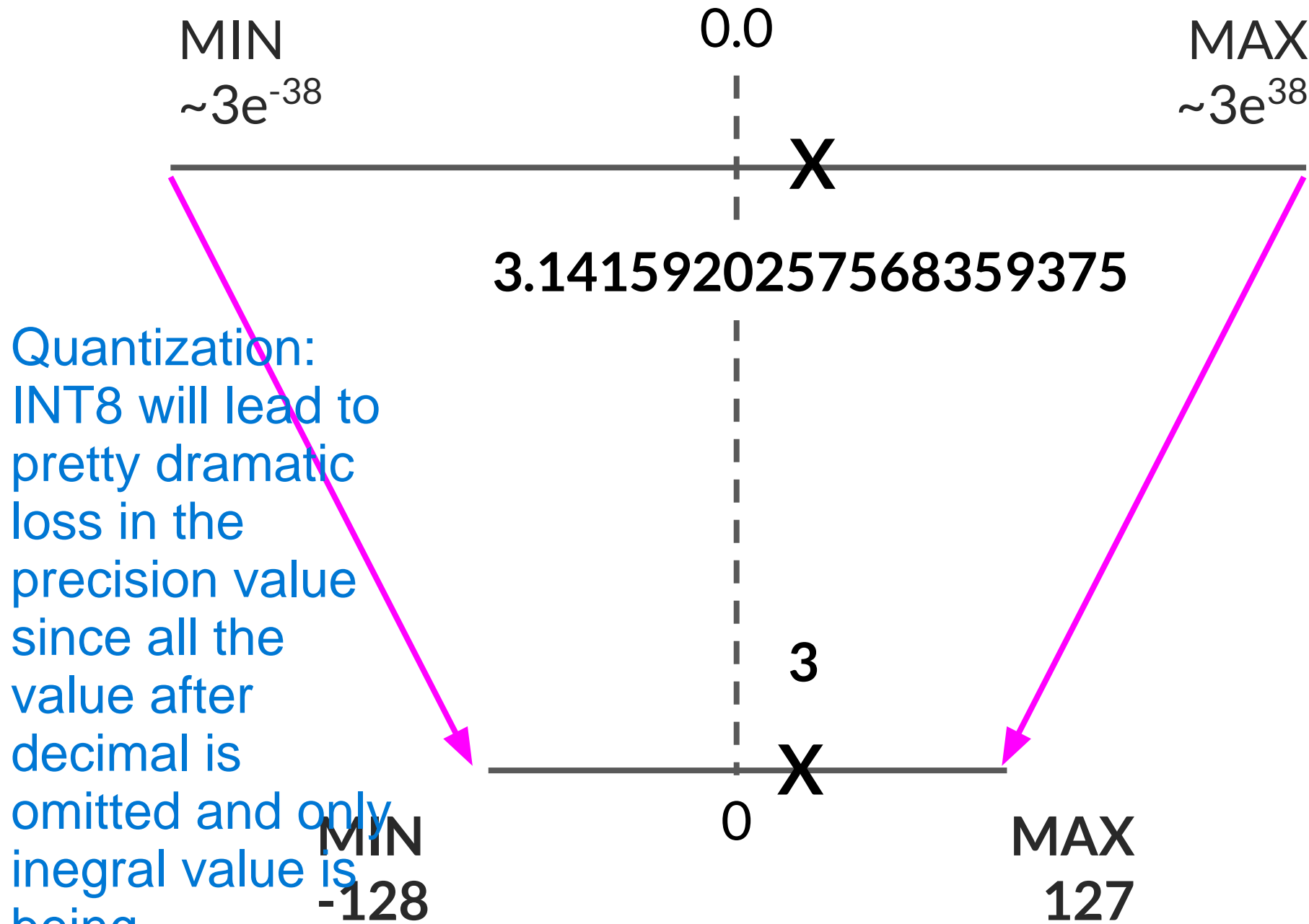
0	10000000	1001001
<hr/>		
Sign 1 bit	Exponent 8 bits	Fraction 7 bits



Quantization: INT8

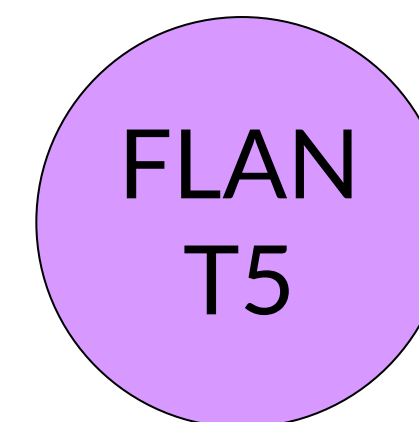
Let's store Pi: 3.141592

Quantization:
INT8 will lead to pretty dramatic loss in the precision value since all the value after decimal is omitted and only integral value is being considered leading to huge loss in precision



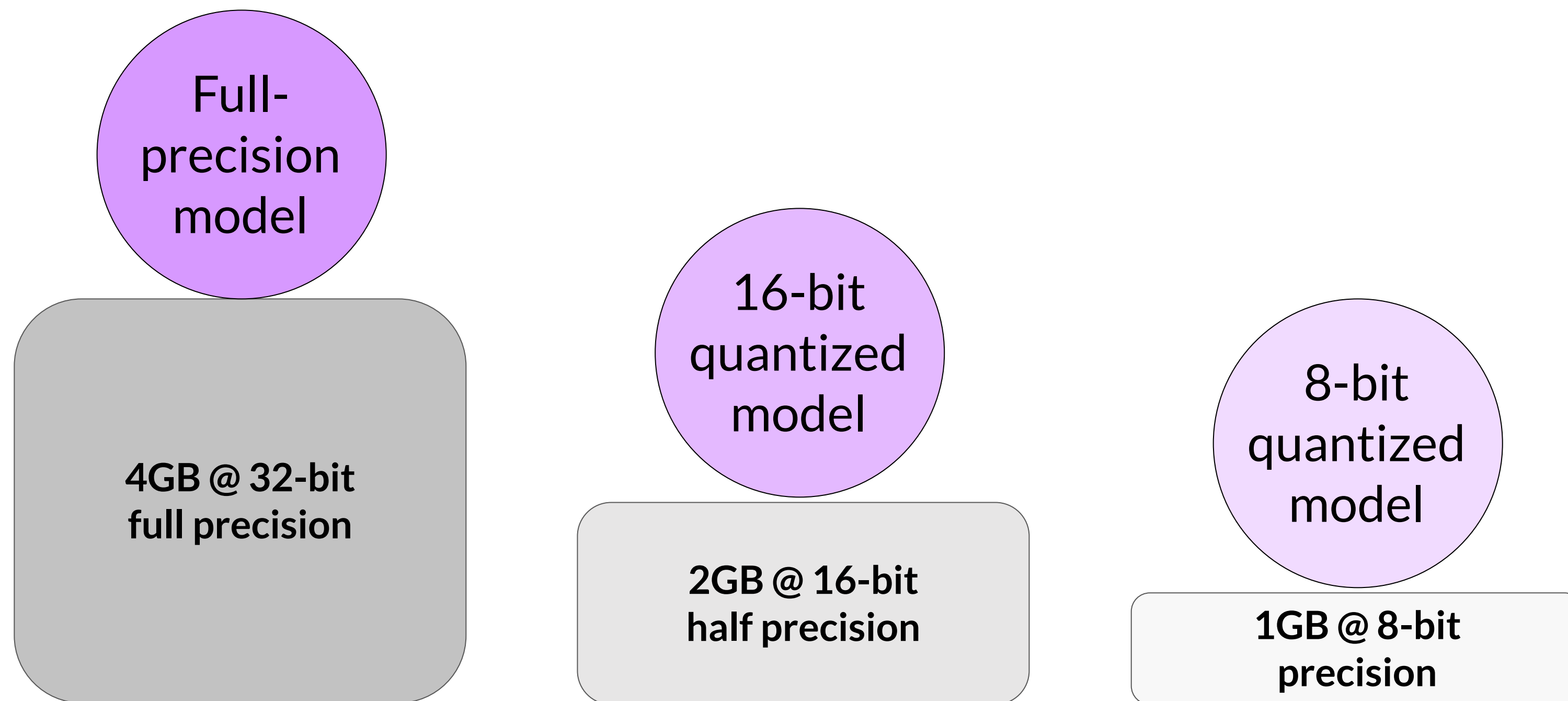
Quantization: Summary

	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
BFLOAT16	16	8	7	2 bytes
INT8	8	-/-	7	1 byte



- Reduce required memory to store and train models ([Goal of quantization](#))
- Projects original 32-bit floating point numbers into lower precision spaces
- Quantization-aware training (QAT) learns the quantization scaling factors during training
- BFLOAT16 is a popular choice

Approximate GPU RAM needed to store 1B parameters



Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, <https://github.com/facebookresearch/bitsandbytes>

Approximate GPU RAM needed to train 1B-params

**80GB @ 32-bit
full precision**

**40GB @ 16-bit
half precision**

**20GB @ 8-bit
precision**

80GB is the maximum memory for the Nvidia A100 GPU, so to keep the model on a single GPU, you need to use 16-bit or 8-bit quantization.

Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, <https://github.com/facebookresearch/bitsandbytes>

GPU RAM needed to train larger models

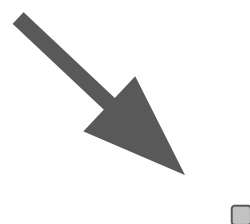
**1B param
model**

**175B param
model**

**500B param
model**

**14,000 GB @ 32-bit
full precision**

**40,000 GB @ 32-bit
full precision**



GPU RAM needed to train larger models

As model sizes get larger, you will need to split your model across multiple GPUs for training

This can be achieved using Distributed Computing Techniques

1B param model

14,000 GB @ 32-bit full precision

175B param model

500B param model

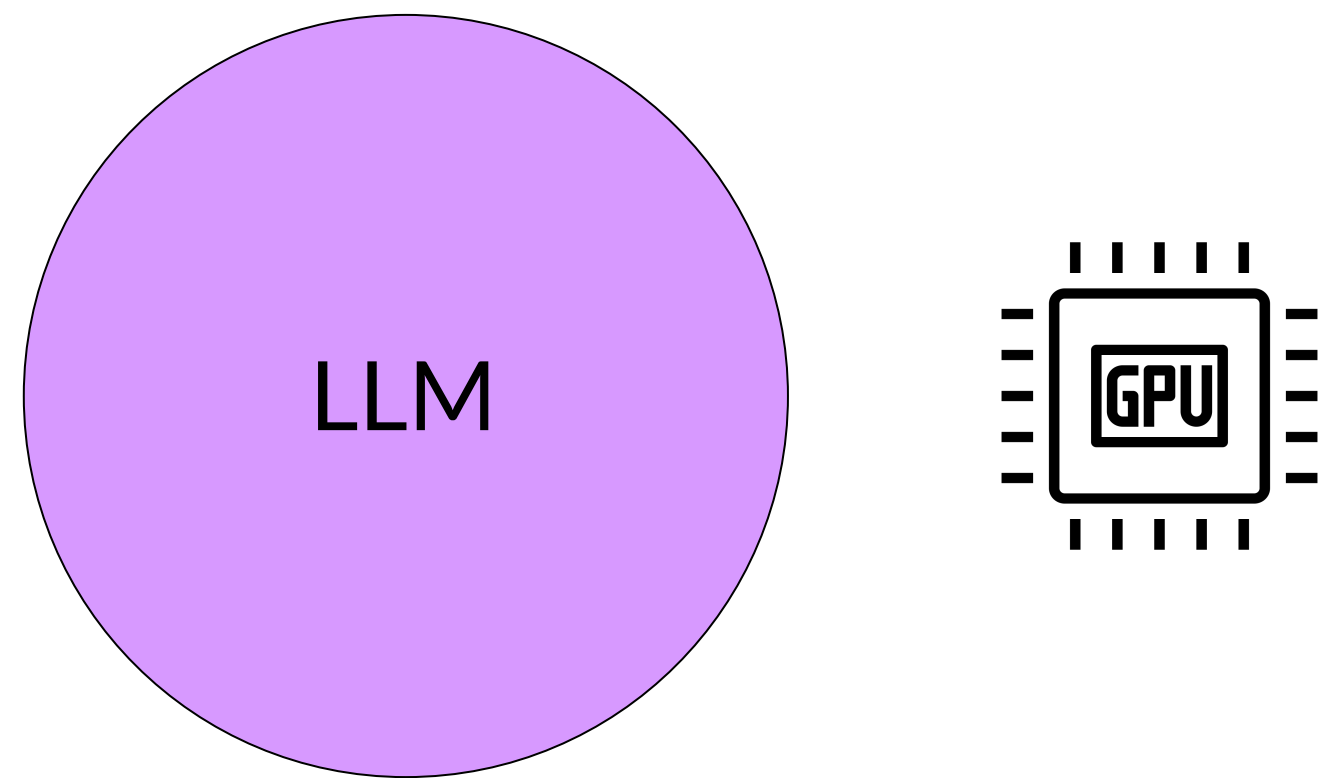
40,000 GB @ 32-bit full precision

This huge compute resources requirement is enough to justify that why it's not feasible for everyone or businesses to train LLMs from scratch and instead utilize the existing LLMs to solve a particular use case

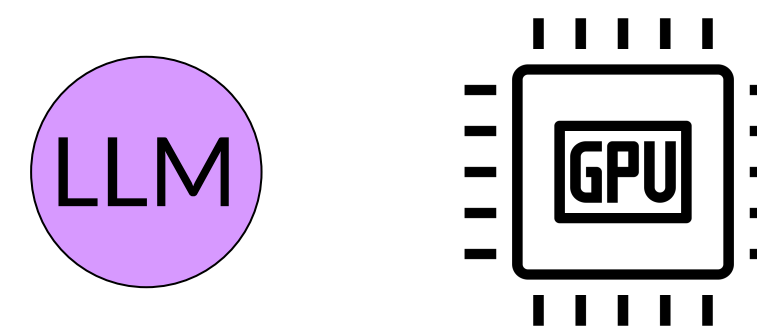
To help you understand more about the technical aspects of training across GPUs, we've prepared this slide in "Efficient Multi-GPU Compute Strategies". It's very detailed, but it will help you understand some of the options that exist for developers like you to train larger models. One should feel free to skip this.

Efficient Multi-GPU Compute Strategies

When to use distributed compute

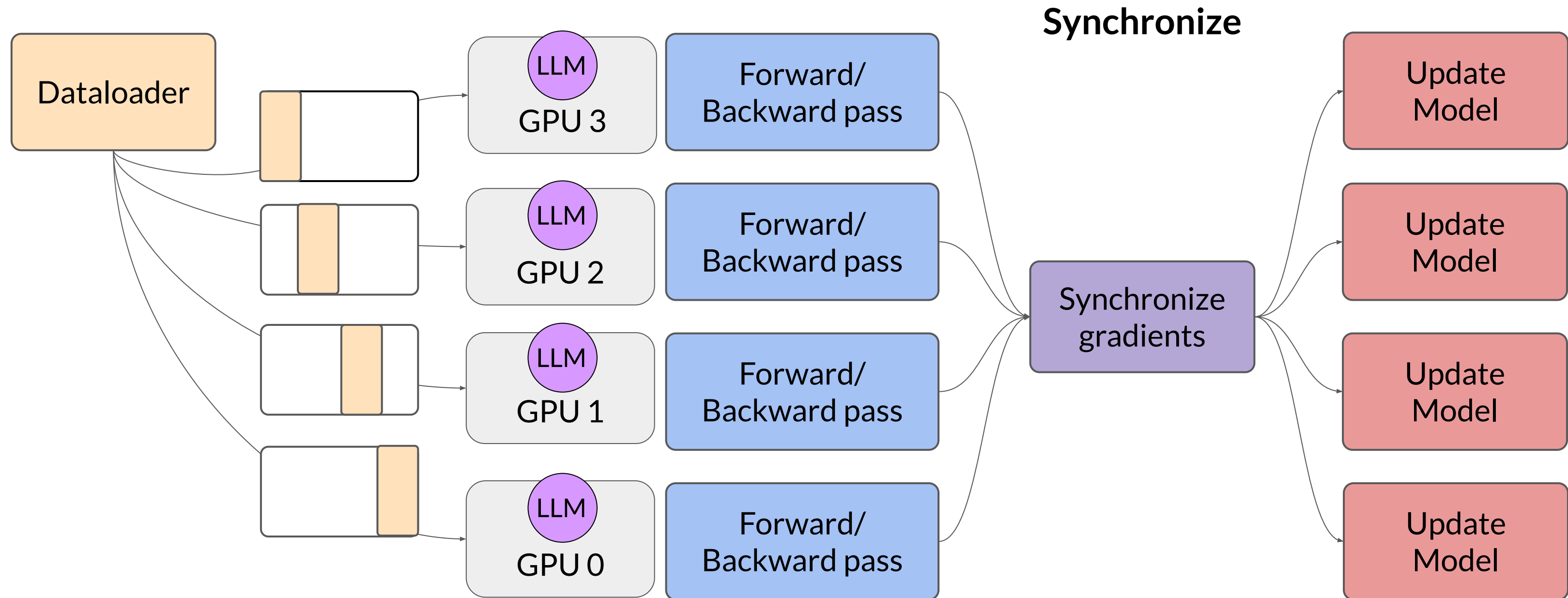


Model too big for single GPU



Model fits on GPU, train data in parallel

Distributed Data Parallel (DDP)



Fully Sharded Data Parallel (FSDP)

- Motivated by the “ZeRO” paper - zero data overlap between GPUs

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Samyam Rajbhandari*, Jeff Rasley*, Olatunji Ruwase, Yuxiong He
`{samyamr, jerasley, olruwase, yuxhe}@microsoft.com`

Sources:

Rajbhandari et al. 2019: “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models”

Zhao et al. 2023: “PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel”

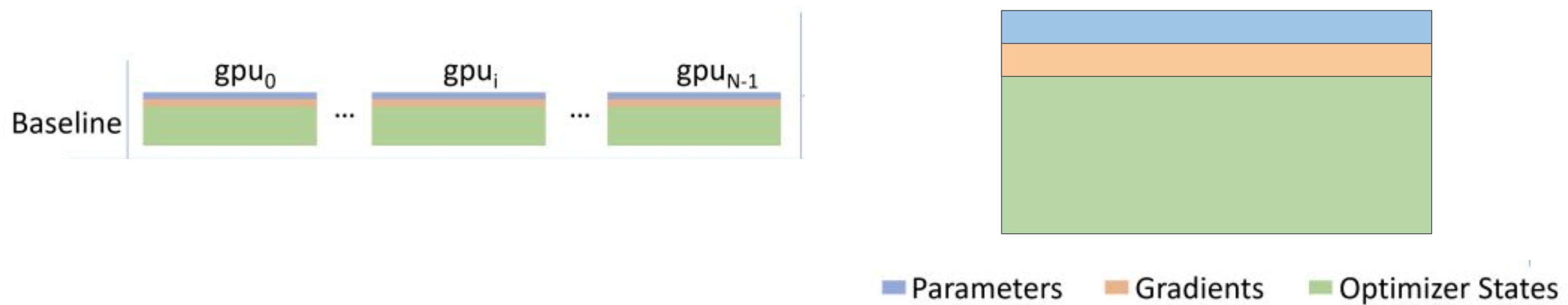
Recap: Additional GPU RAM needed for training

	Bytes per parameter
Model Parameters (Weights)	4 bytes per parameter
Adam optimizer (2 states)	+8 bytes per parameter
Gradients	+4 bytes per parameter
Activations and temp memory (variable size)	+8 bytes per parameter (high-end estimate)
TOTAL	=4 bytes per parameter +20 extra bytes per parameter

Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, <https://github.com/facebookresearch/bitsandbytes>

Memory usage in DDP

- One full copy of model and training parameters on each GPU



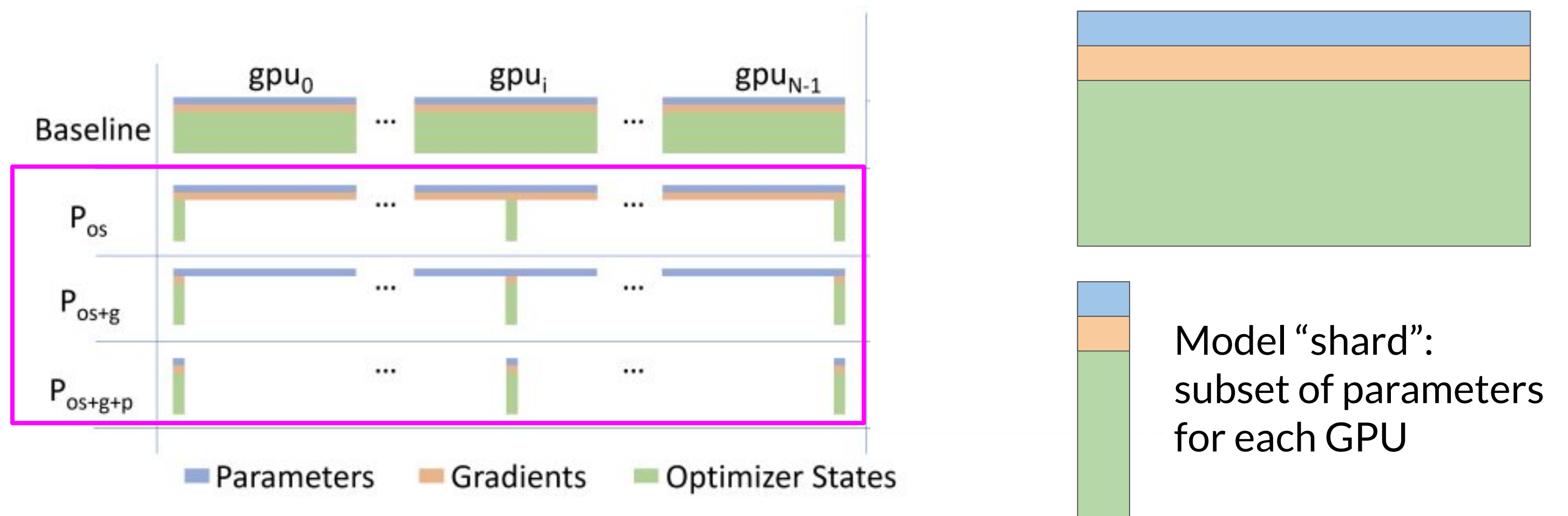
Sources:

Rajbhandari et al. 2019: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models"

Zhao et al. 2023: "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel"

Zero Redundancy Optimizer (ZeRO)

- Reduces memory by distributing (sharding) the model parameters, gradients, and optimizer states across GPUs



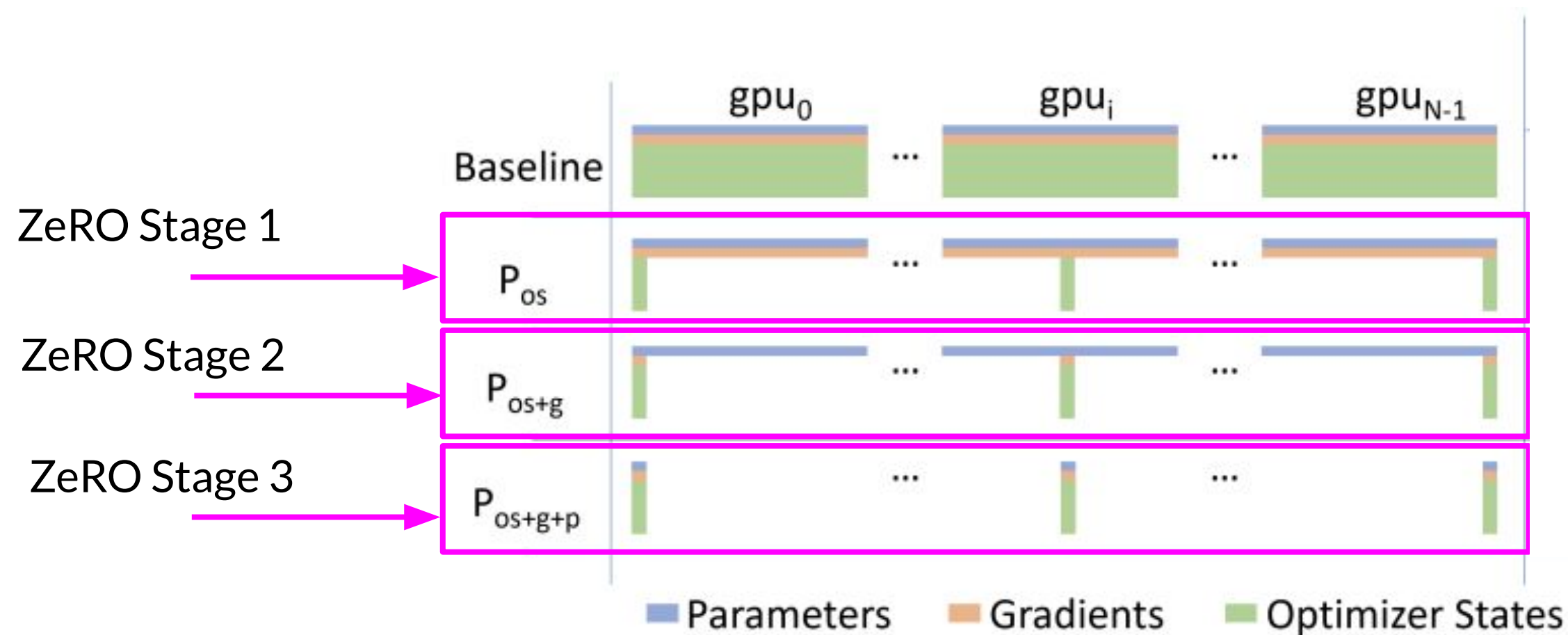
Sources:

Rajbhandari et al. 2019: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models"

Zhao et al. 2023: "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel"

Zero Redundancy Optimizer (ZeRO)

- Reduces memory by distributing (sharding) the model parameters, gradients, and optimizer states across GPUs

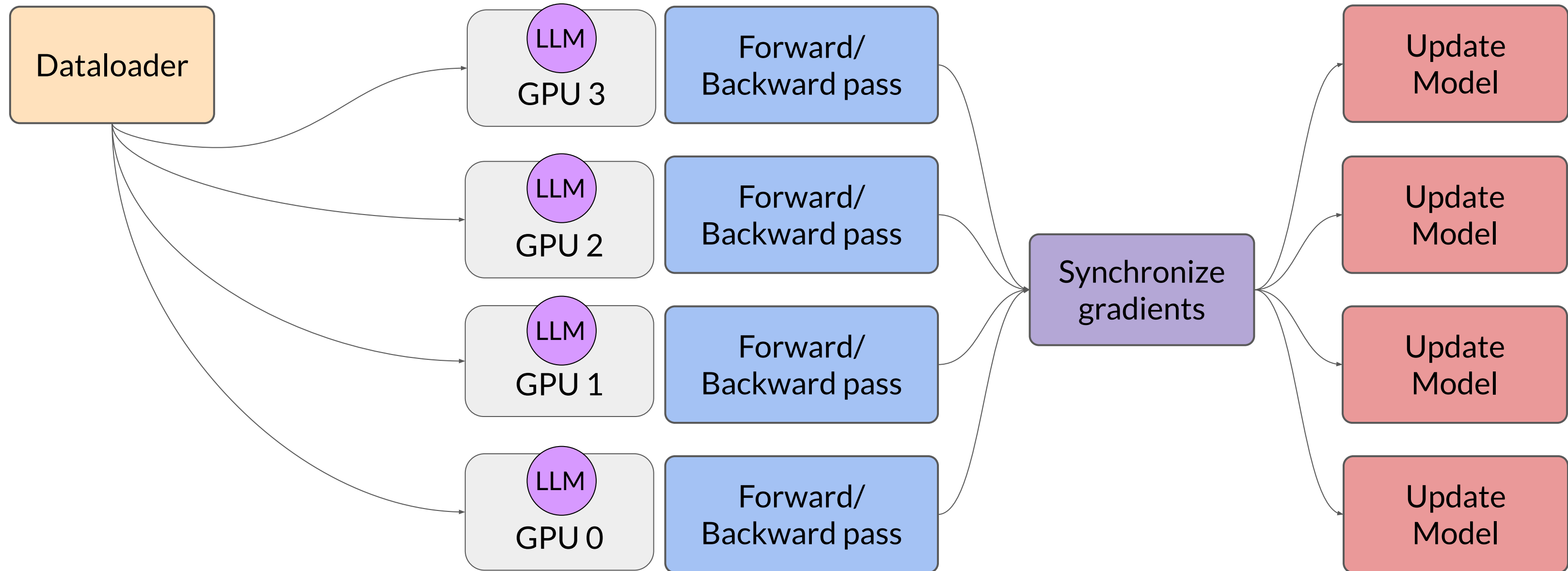


Sources:

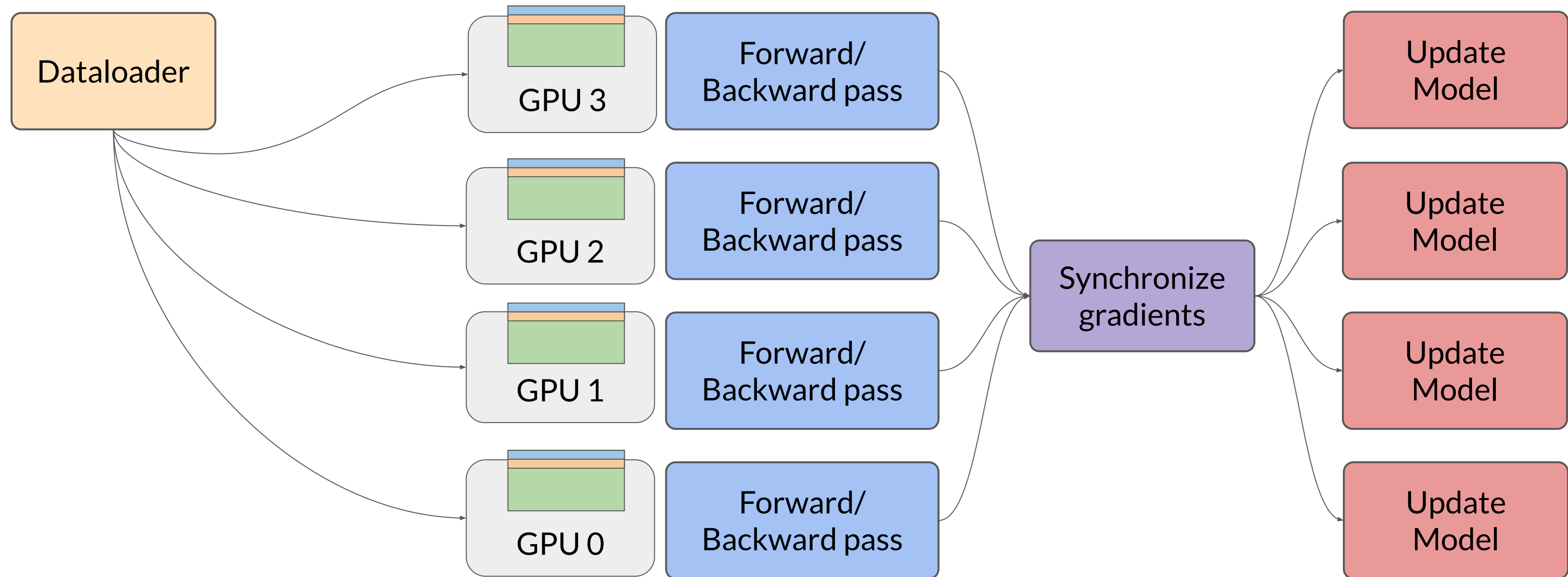
Rajbhandari et al. 2019: “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models”

Zhao et al. 2023: “PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel”

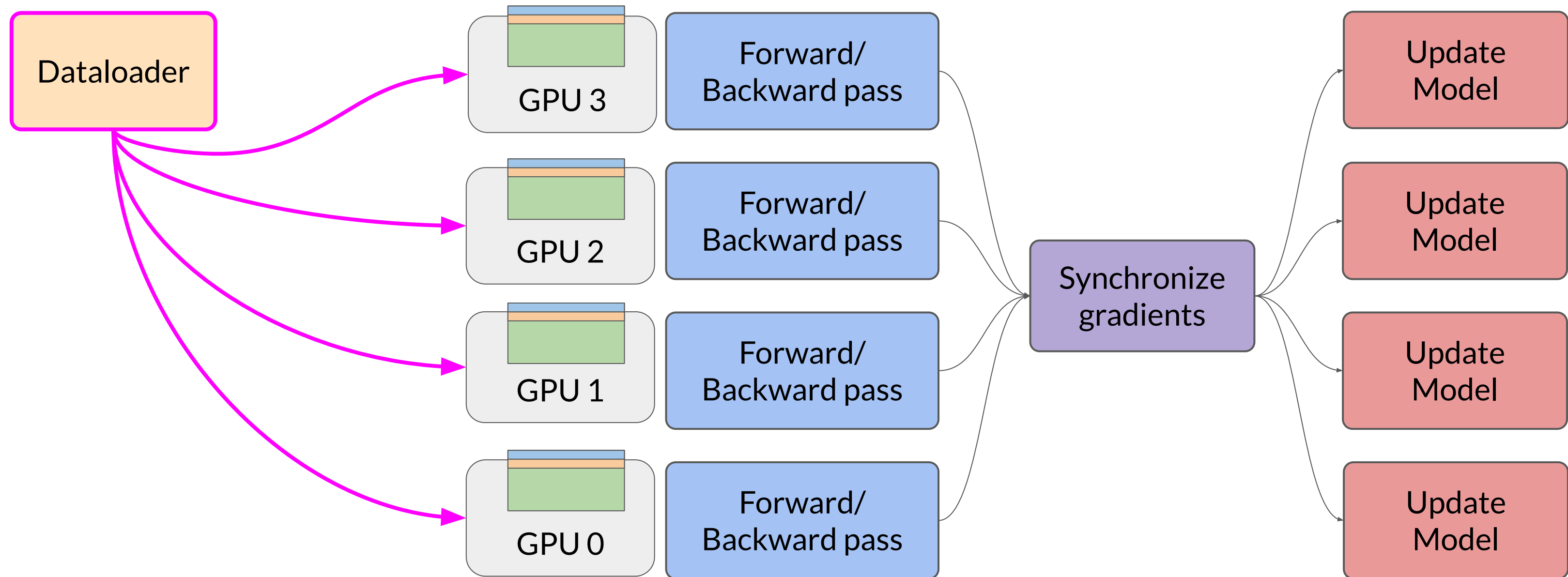
Distributed Data Parallel (DDP)



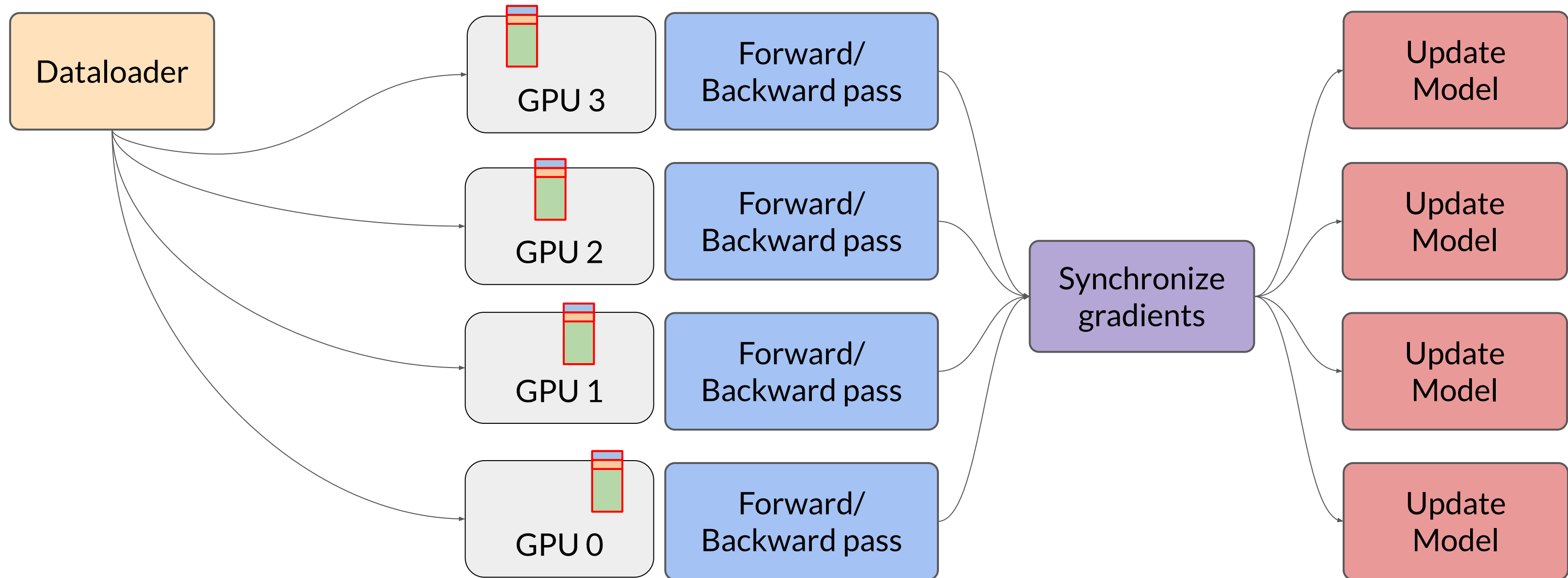
Distributed Data Parallel (DDP)



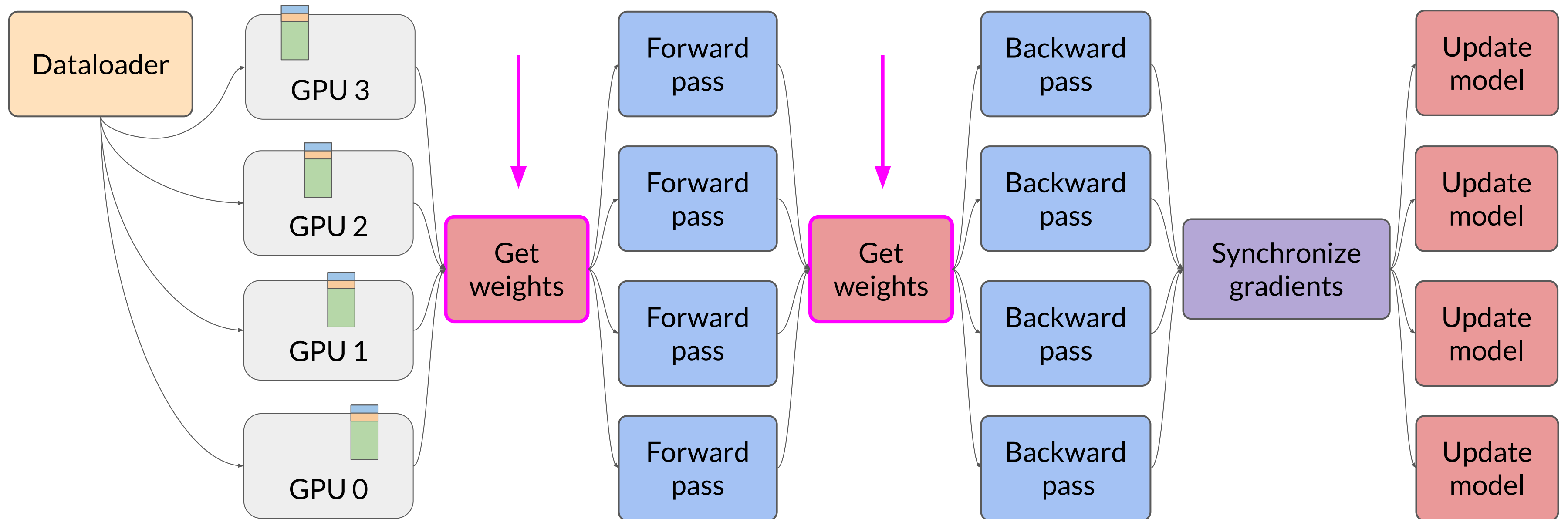
Fully Sharded Data Parallel (FSDP)



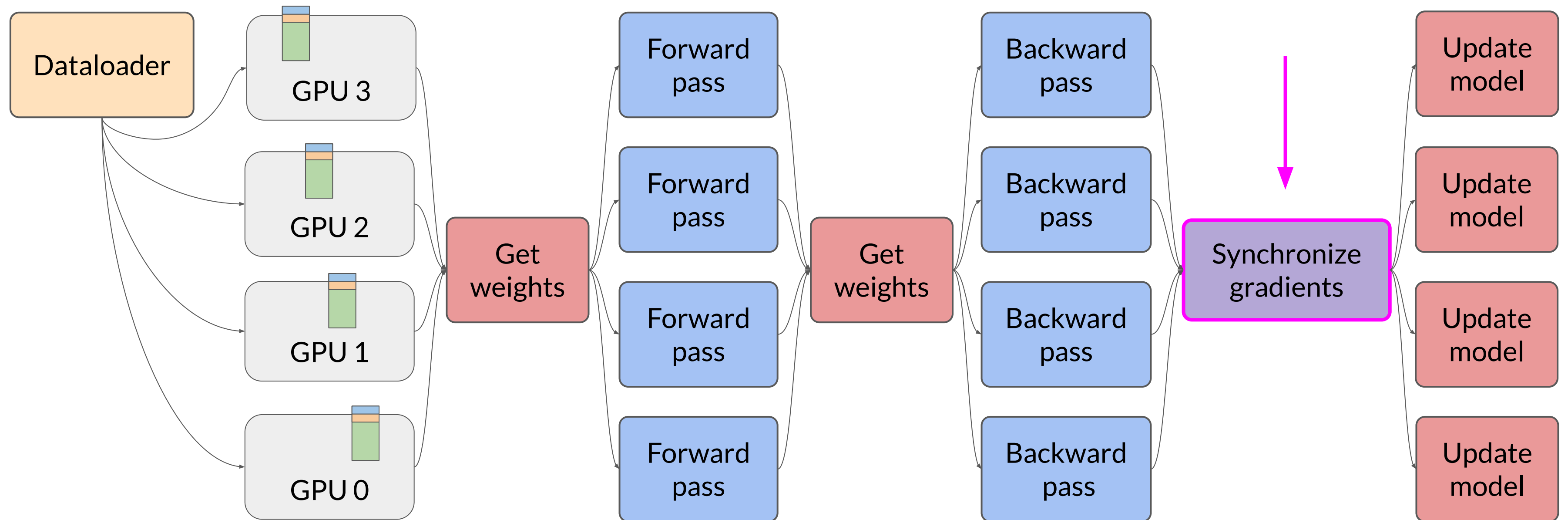
Fully Sharded Data Parallel (FSDP)



Fully Sharded Data Parallel (FSDP)



Fully Sharded Data Parallel (FSDP)

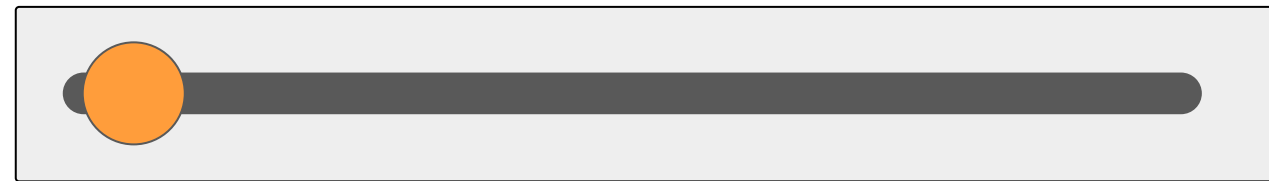


Fully Sharded Data Parallel (FSDP)

- Helps to reduce overall GPU memory utilization
- Supports offloading to CPU if needed
- Configure level of sharding via `sharding_factor`

Full replication (no sharding)

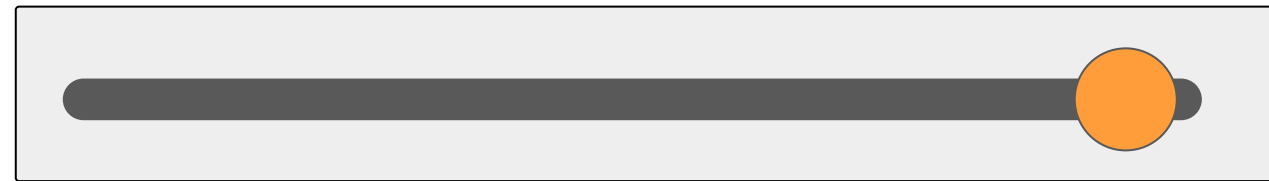
1 GPU



max. number of GPUs

Full sharding

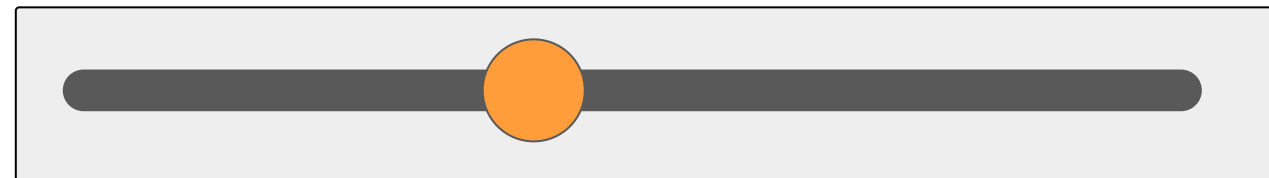
1 GPU



max. number of GPUs

Hybrid sharding

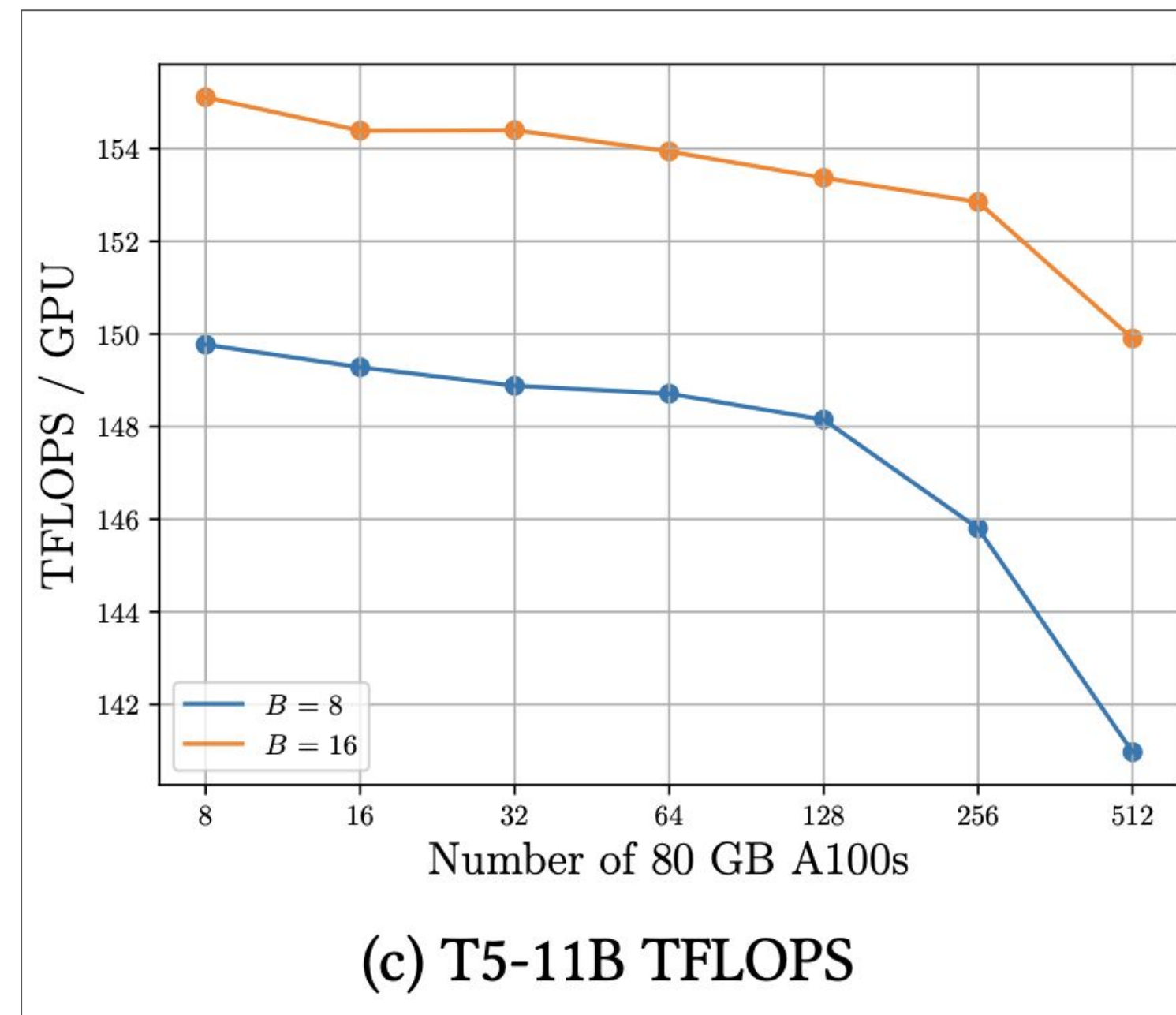
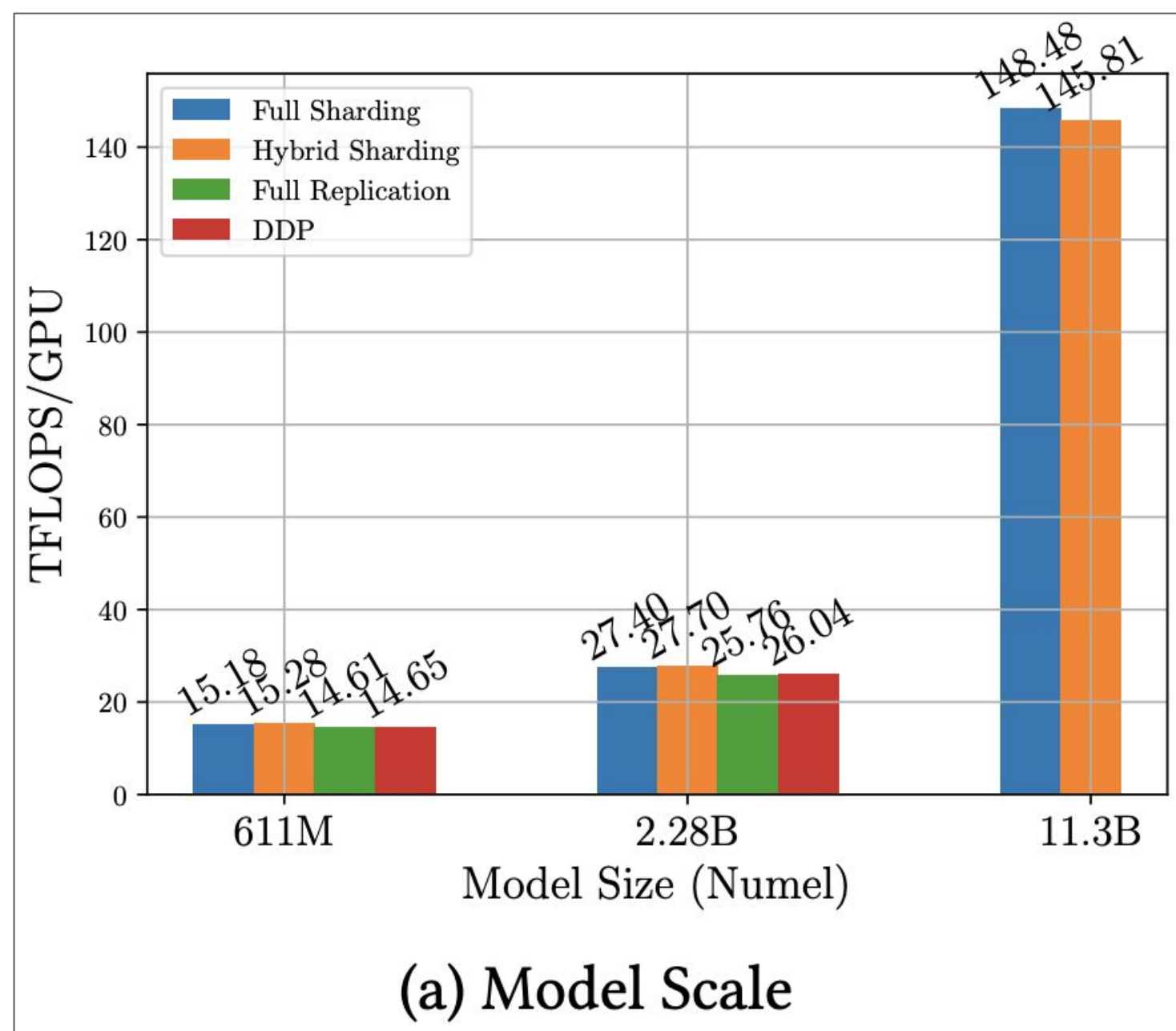
1 GPU



max. number of GPUs

Impact of using FSDP

Note: 1 teraFLOP/s = 1,000,000,000,000
(one trillion) floating point operations per second



Zhao et al. 2023: "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel"

CONTINUE

Scaling laws and compute-optimal models

Here you'll learn about research that has explored the relationship between model size, training, configuration and performance in an effort to determine just how big models need to be.

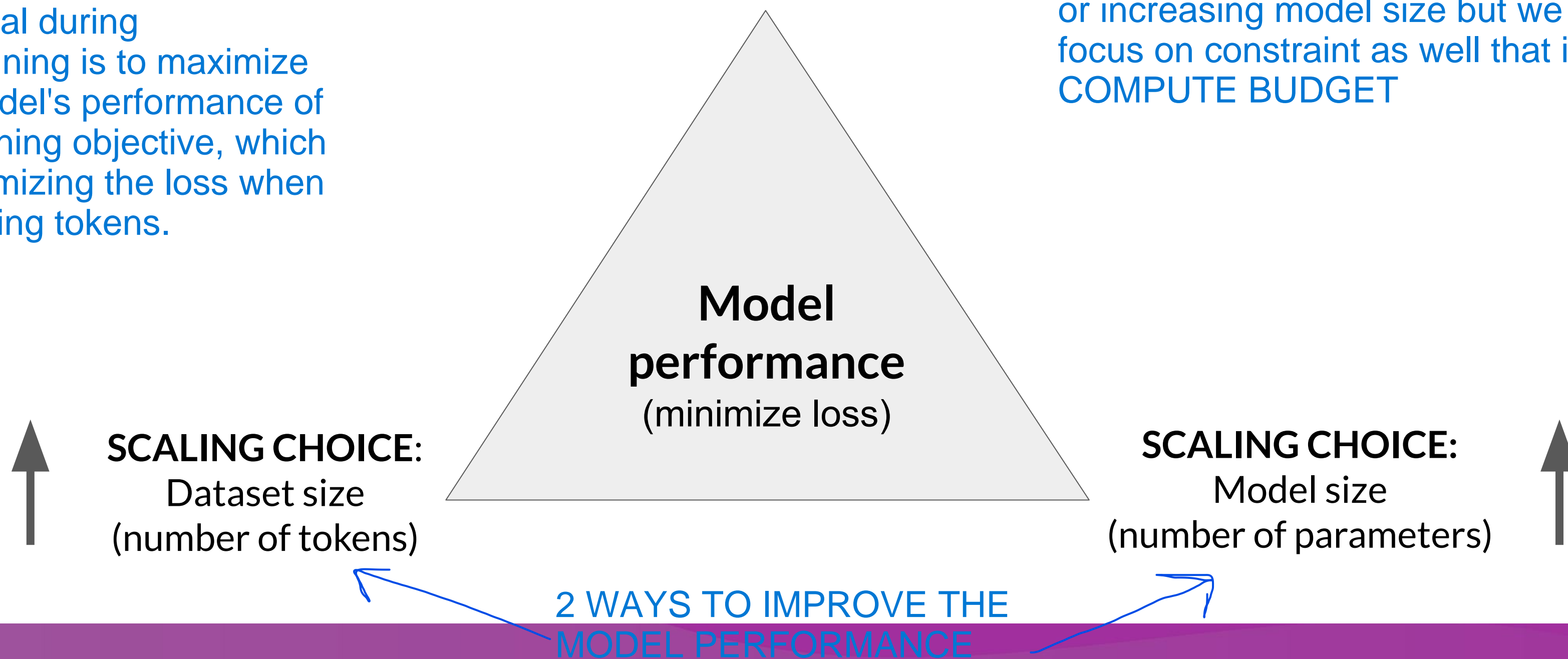
Scaling choices for pre-training

Goal: maximize model performance

The goal during pre-training is to maximize the model's performance of its learning objective, which is minimizing the loss when predicting tokens.

CONSTRAINT:
Compute budget
(GPUs, training time, cost)

In practical performance can be improved by either increasing data size or increasing model size but we need to focus on constraint as well that is **COMPUTE BUDGET**



Compute budget for training LLMs

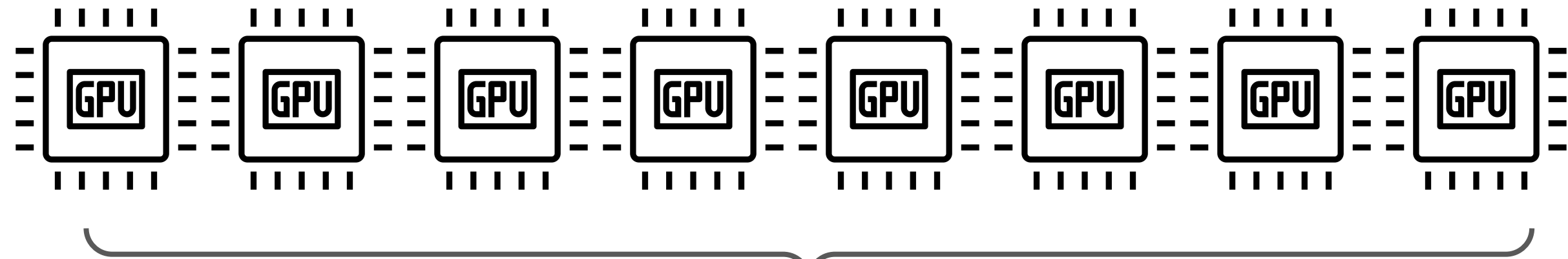
FLOP means floating point operation

unit of compute that quantifies the required resources.

1 “petaflop/s-day” =

floating point operations performed at rate of 1 petaFLOP per second for one day

NVIDIA V100s



Note: 1 petaFLOP/s = 1,000,000,000,000,000
(one quadrillion) floating point operations per second

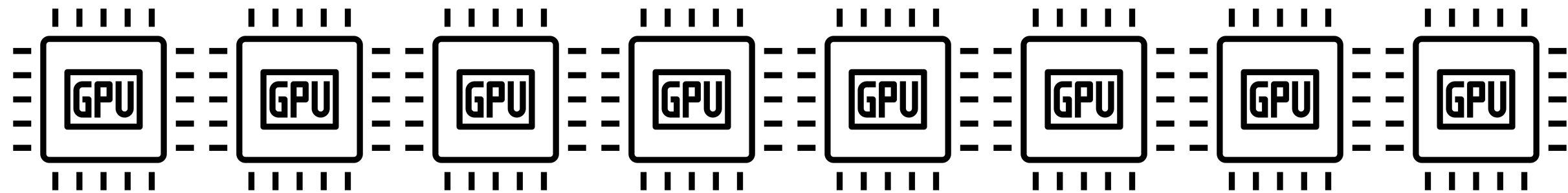
1 petaflop/s-day is these chips (i.e 8 NVIDIA V100s chips)
running at full efficiency for 24 hours

Compute budget for training LLMs

1 “petaflop/s-day” =

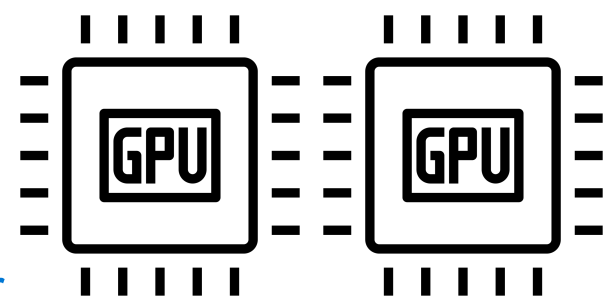
floating point operations performed at rate of 1 petaFLOP per second for one day

NVIDIA V100s



OR

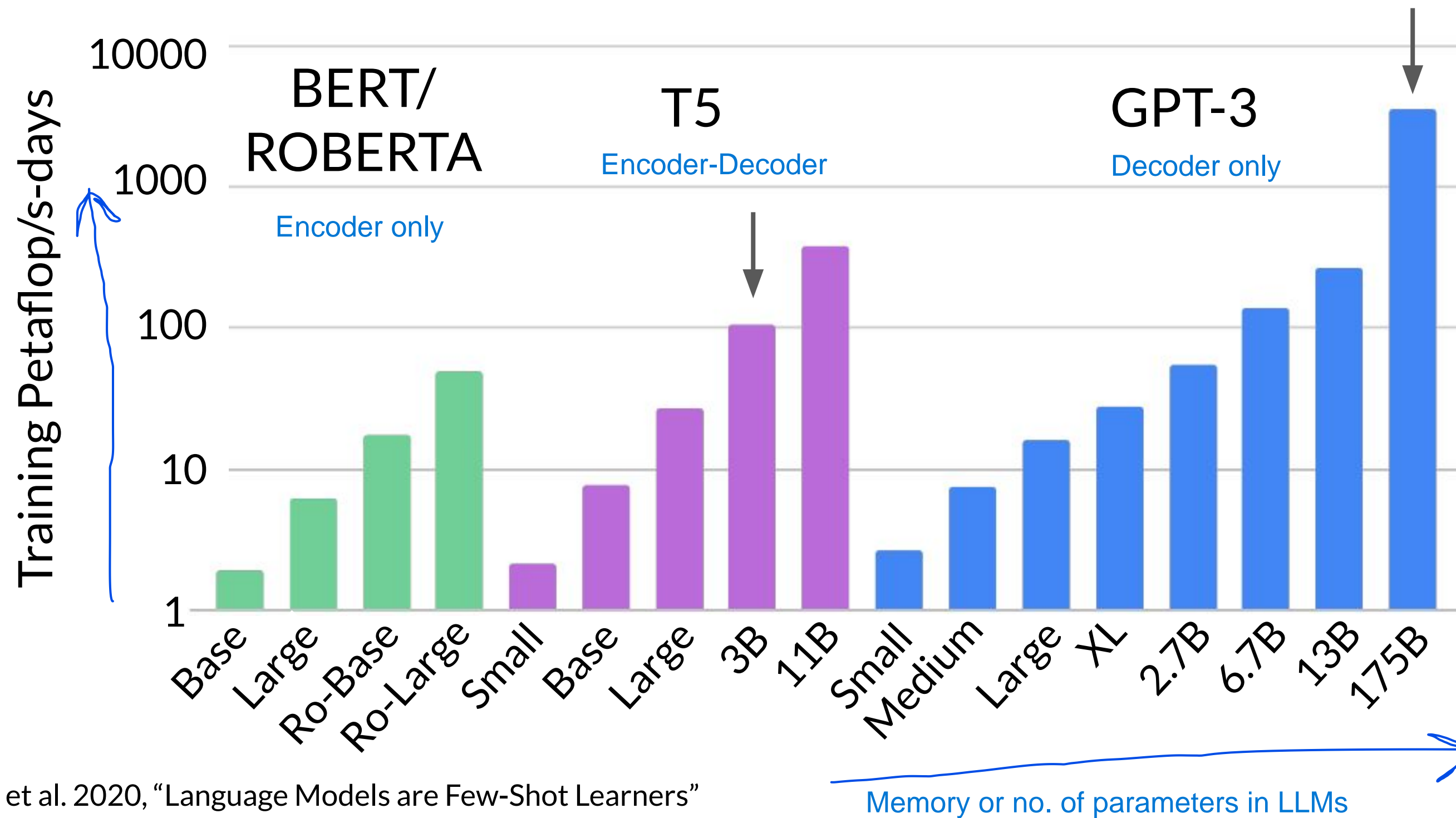
NVIDIA A100s



1 petaflop/s-day is these chips running at full efficiency for 24 hours

If you have a more powerful processor that can carry out more operations at once, then a petaFLOP per second day requires fewer chips. For example, two NVIDIA A100 GPUs give equivalent compute to the eight V100 chips.

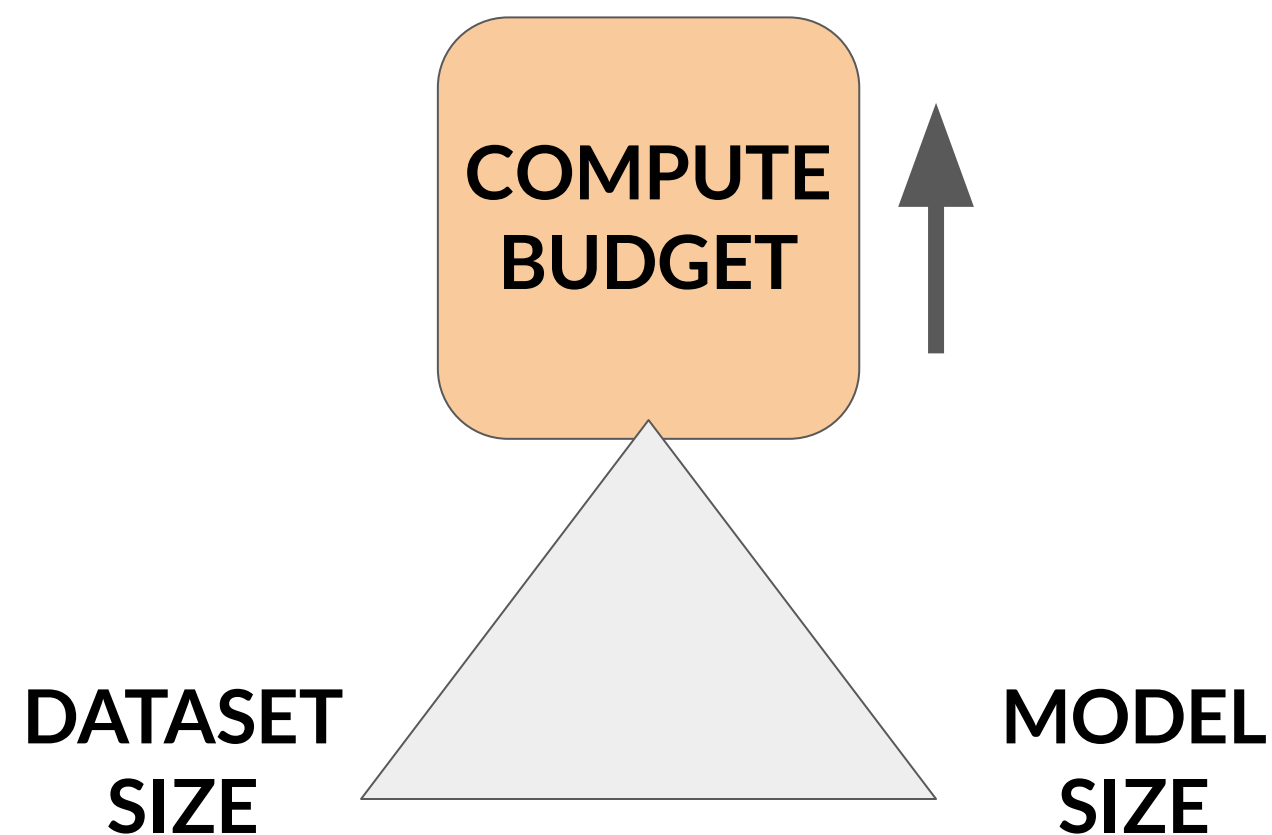
Number of petaflop/s-days to pre-train various LLMs



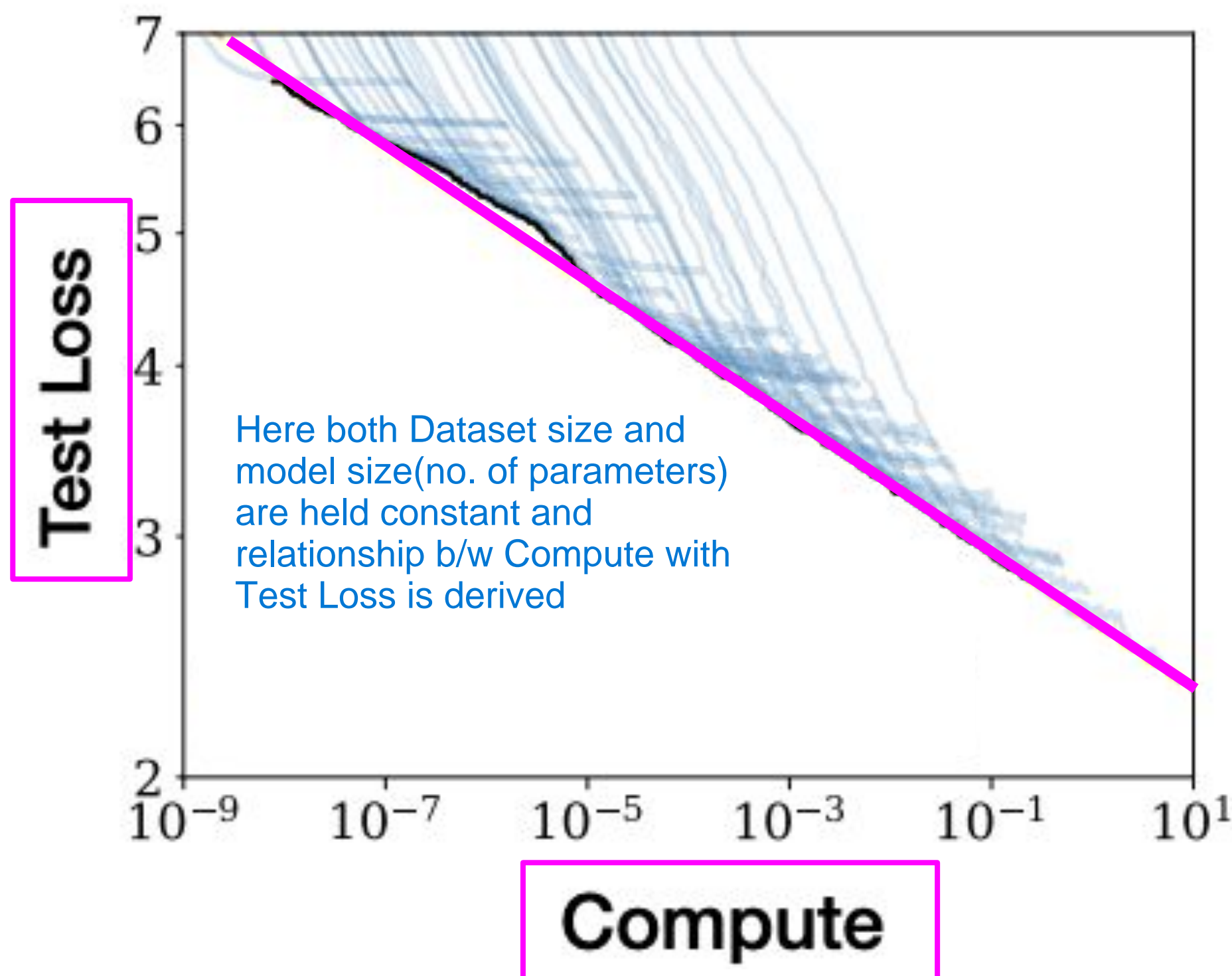
Source: Brown et al. 2020, "Language Models are Few-Shot Learners"

Compute budget vs. model performance

[Read this sticky note](#)

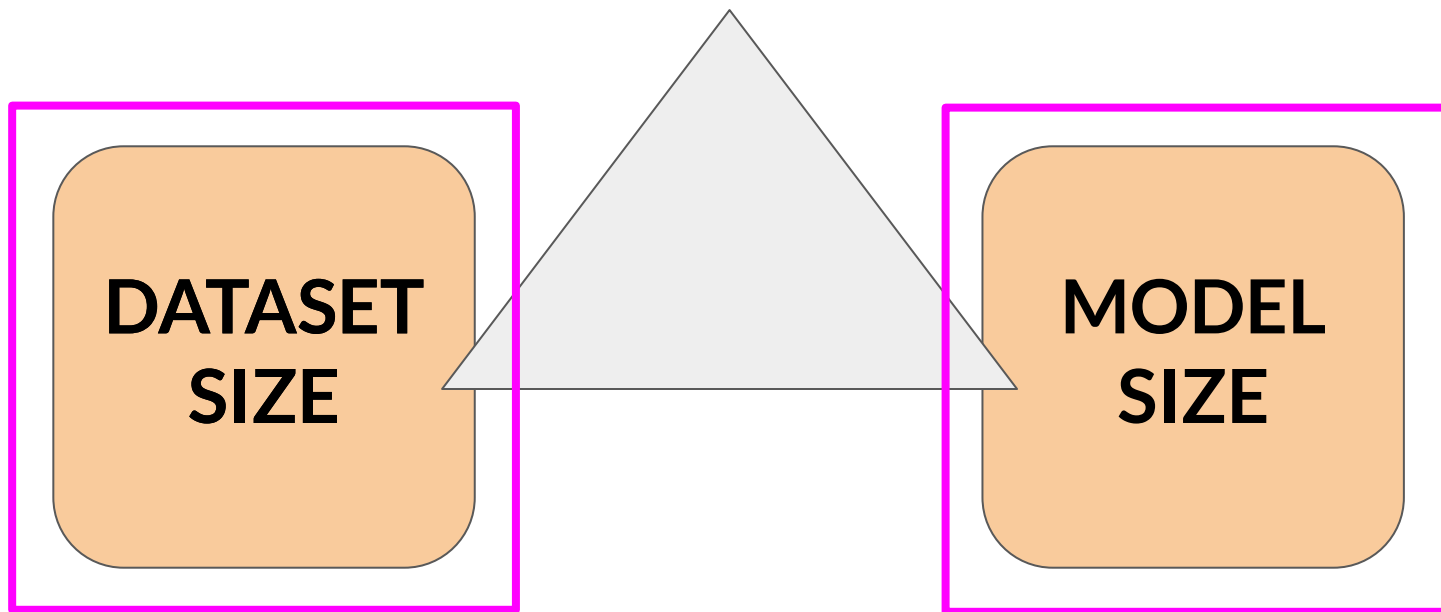


On a whole to summarize, as we increase the compute size (represented along x axis in petaflops/sec-day) loss decreases which means LLM model performs better



Source: Kaplan et al. 2020, "Scaling Laws for Neural Language Models"

Dataset size and model size vs. performance



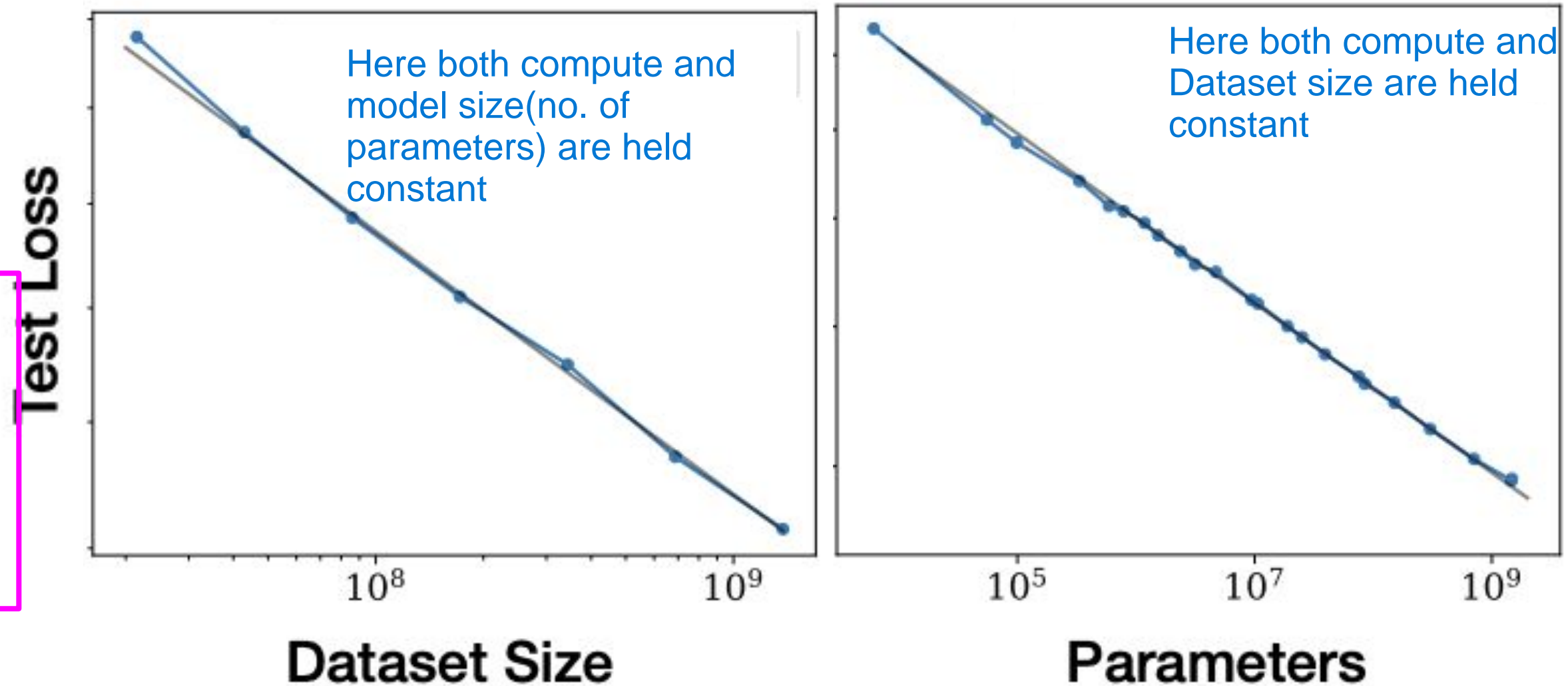
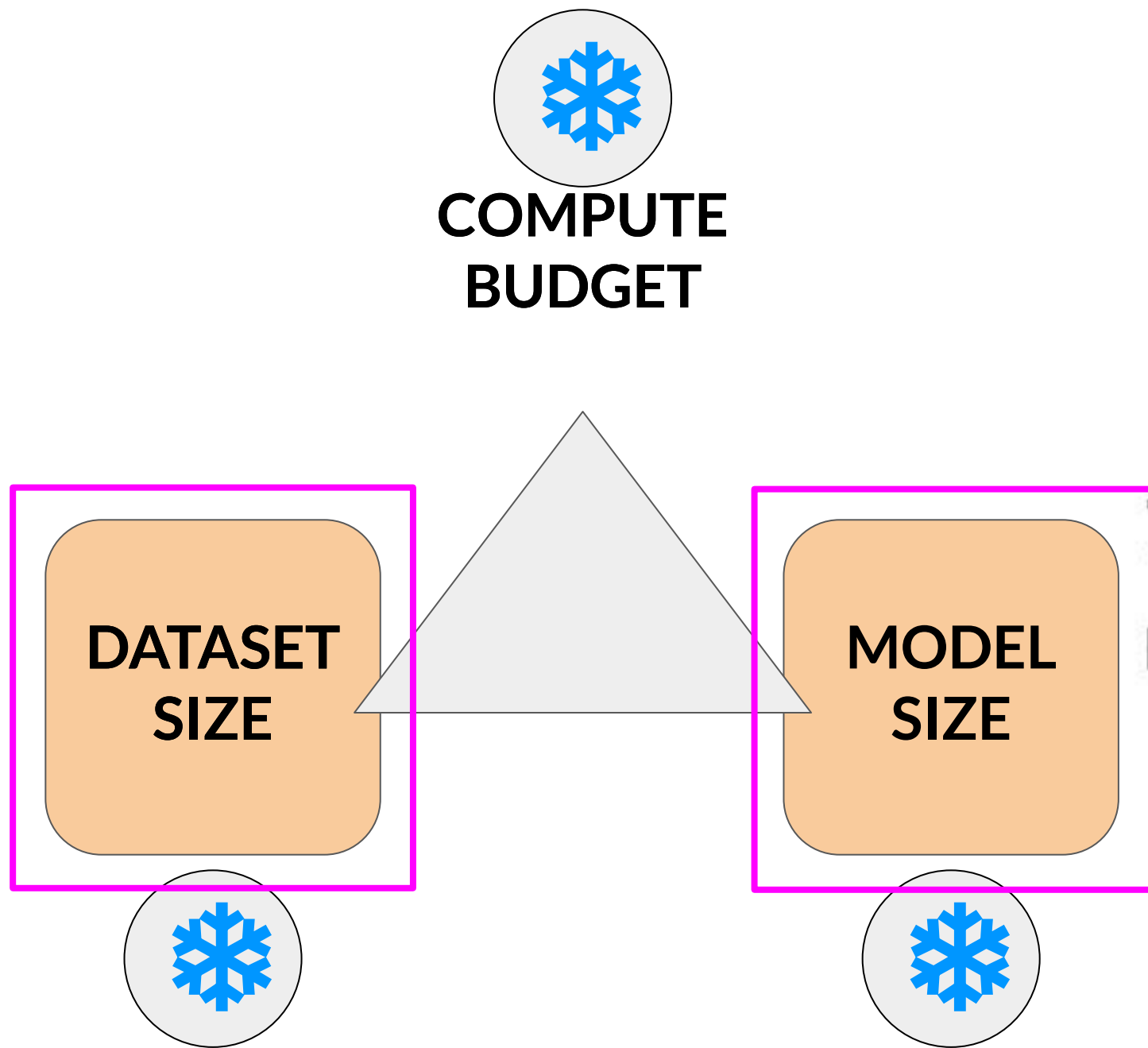
Compute resource constraints

- Hardware
- Project timeline
- Financial budget

In practice however, the compute resources you have available for training will generally be a hard constraint set by factors such as the hardware you have access to, the time available for training and the financial budget of the project. If you hold your compute budget fixed, the two levers you have to improve your model's performance are the size of the training dataset and the number of parameters in your model.

Source: Kaplan et al. 2020, "Scaling Laws for Neural Language Models"

Dataset size and model size vs. performance



Open AI researchers observed that similar to Compute, both Dataset size and Model size hold power law relationship with Test loss. In short which means that performance of model increases:

- With increase in Dataset size
- With increase in Model size
- With increase in Compute size

Source: Kaplan et al. 2020, "Scaling Laws for Neural Language Models"

Chinchilla paper

AIM OF CHINCILLA PAPER: The goal was to find the optimal number of parameters and volume of training data for a given compute budget.

We will now be having a question in our mind that what's the ideal balance b/w these three quantities (Compute, Dataset size and Model size)?

In 2022 a research paper was published called Chinchilla paper that carried out a detailed study of the performance of language models of various sizes and quantities of training data. The goal was to find the optimal number of parameters and volume of training data for a given compute budget.

Training Compute-Optimal Large Language Models

Jordan Hoffmann*, Sebastian Borgeaud*, Arthur Mensch*, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals and Laurent Sifre*

*Equal contributions

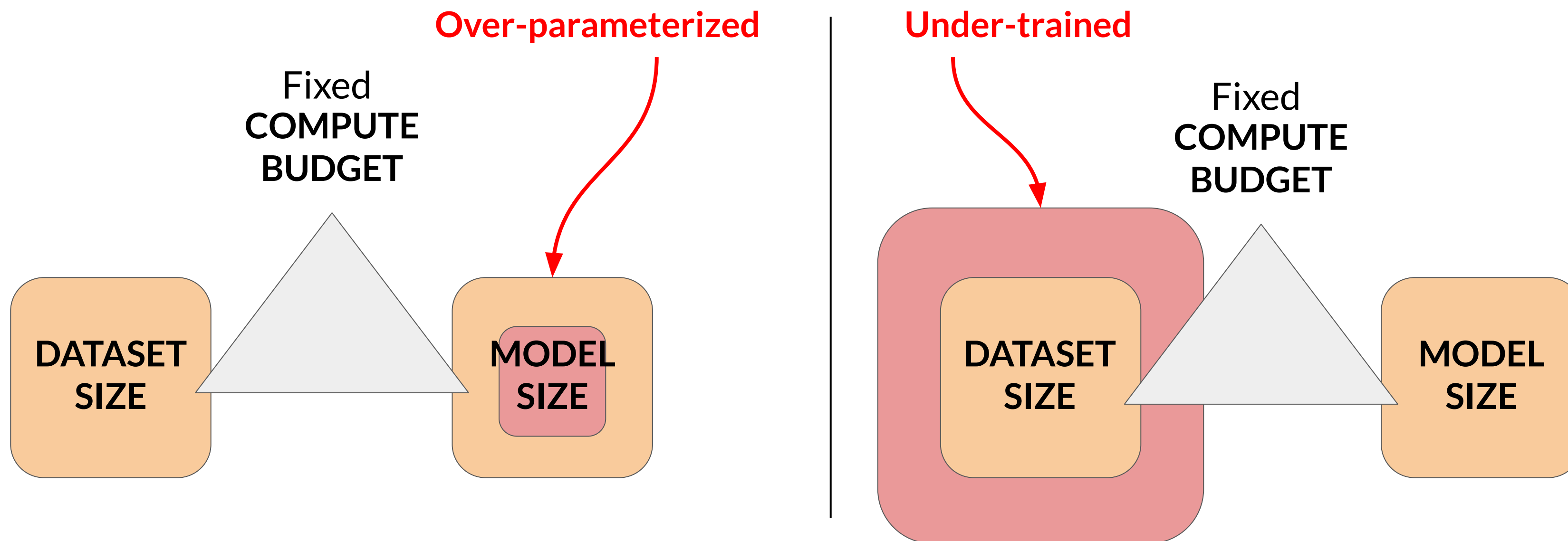
We investigate the optimal model size and number of tokens for training a transformer language model under a given compute budget. We find that current large language models are significantly under-trained, a consequence of the recent focus on scaling language models whilst keeping the amount of training data constant. By training over 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens, we find that for compute-optimal training, the model size and the number of training tokens should be scaled equally: for every doubling of model size the number of training tokens should also be doubled. We test this hypothesis by training a predicted compute-optimal model, *Chinchilla*, that uses the same compute budget as *Gopher* but with 70B parameters and 4× more more data. *Chinchilla* uniformly and significantly outperforms *Gopher* (280B), GPT-3 (175B), Jurassic-1 (178B), and Megatron-Turing NLG (530B) on a large range of downstream evaluation tasks. This also means that *Chinchilla* uses substantially less compute for fine-tuning and inference, greatly facilitating downstream usage. As a highlight, *Chinchilla* reaches a state-of-the-art average accuracy of 67.5% on the MMLU benchmark, greater than a 7% improvement over *Gopher*.

Jordan et al. 2022

Compute optimal models

Here we will be discussing some of the findings of the Chinchilla Paper

- Very large models may be **over-parameterized** and **under-trained**
 - Like GPT3
 - Models are using more parameters as compared to actual need to have good understanding of language
 - Dataset that promotes underfitting
- Smaller models trained on more data could perform as well as large models



Chinchilla scaling laws for model and dataset size

For a 70 billion parameter model, the ideal training dataset contains 1.4 trillion tokens or 20 times the number of parameters.

Training dataset size proposed by Chinchilla

Actual Training dataset size on top of which respective LLM get trained

Model	# of parameters	Compute-optimal* # of tokens (~20x)	Actual # tokens
Chinchilla	70B	~1.4T	1.4T
LLaMA-65B	65B	~1.3T	1.4T
GPT-3	175B	~3.5T	300B
OPT-175B	175B	~3.5T	180B
BLOOM	176B	~3.5T	350B

One important takeaway from the Chinchilla paper is that the optimal training dataset size for a given model is about 20 times larger than the number of parameters in the model

Compute optimal training dataset size is ~20x number of parameters

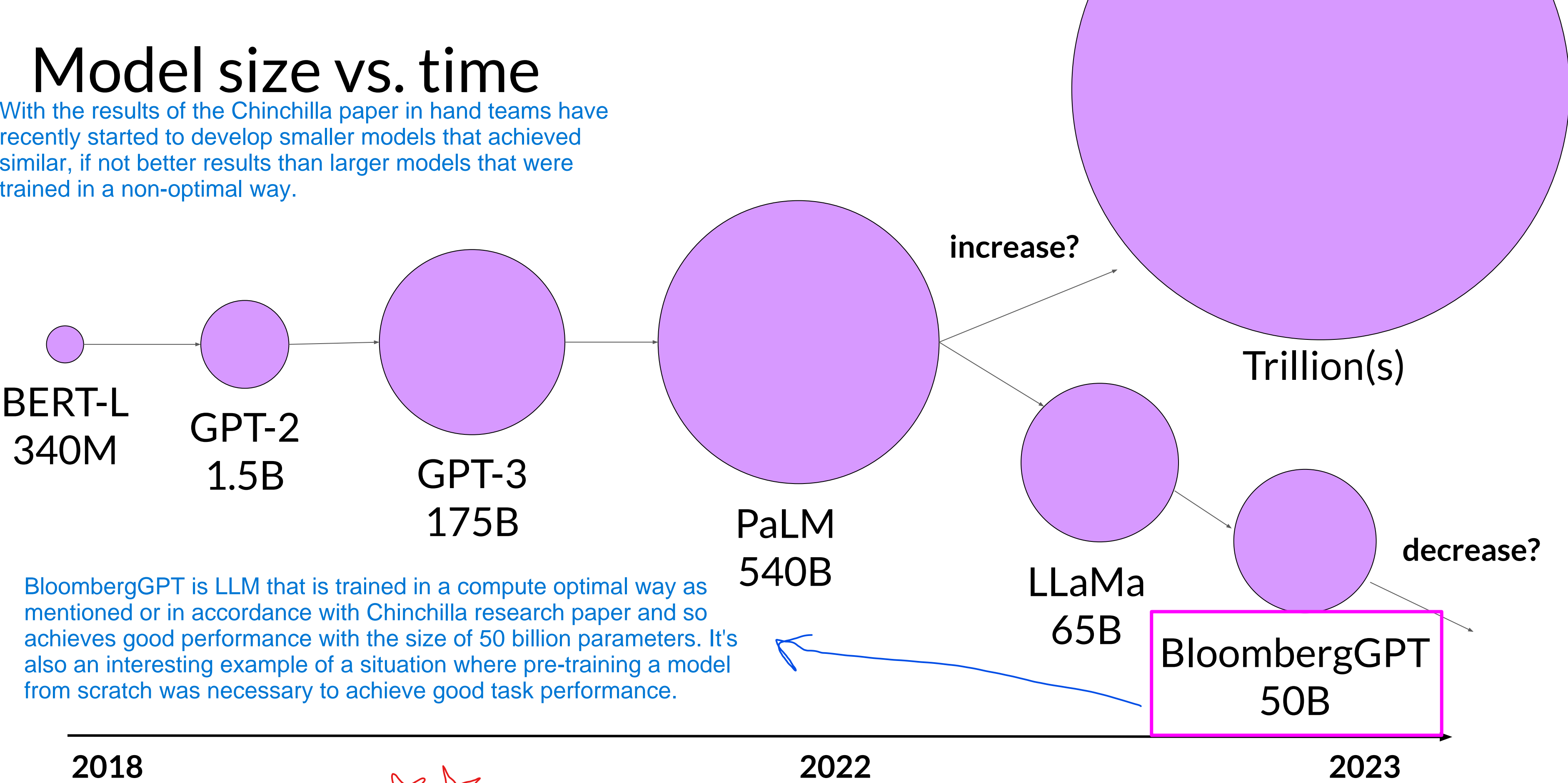
The last three models in the table were trained on datasets that are smaller than the Chinchilla optimal size. These models may actually be under trained.

Sources: Hoffmann et al. 2022, "Training Compute-Optimal Large Language Models"
Touvron et al. 2023, "LLaMA: Open and Efficient Foundation Language Models"

* assuming models are trained to be compute-optimal per Chinchilla paper

Model size vs. time

With the results of the Chinchilla paper in hand teams have recently started to develop smaller models that achieved similar, if not better results than larger models that were trained in a non-optimal way.



BloombergGPT is LLM that is trained in a compute optimal way as mentioned or in accordance with Chinchilla research paper and so achieves good performance with the size of 50 billion parameters. It's also an interesting example of a situation where pre-training a model from scratch was necessary to achieve good task performance.

If your target domain uses vocabulary and language structures that are not commonly used in day to day language. You may need to perform domain adaptation to achieve good model performance in which we may need a bit to train model from scratch in the form of pre-training for domain adaptation.

Pre-training for domain adaptation

Pre-training for domain adaptation

Legal language

Pre-training for domain adaptation

Legal language

The prosecutor had difficulty proving mens rea, as the defendant seemed unaware that his actions were illegal.

The judge dismissed the case, citing the principle of res judicata as the issue had already been decided in a previous trial.

Despite the signed agreement, the contract was invalid as there was no consideration exchanged between the parties.

For example, imagine you're a developer building an app to help lawyers and paralegals summarize legal briefs. Legal writing makes use of very specific terms like mens rea in the first example and res judicata in the second. These words are rarely used outside of the legal world, which means that they are unlikely to have appeared widely in the training text of existing LLMs. As a result, the models may have difficulty understanding these terms or using them correctly. Another issue is that legal language sometimes uses everyday words in a different context, like consideration in the third example. Which has nothing to do with being nice, but instead refers to the main element of a contract that makes the agreement enforceable

Pre-training for domain adaptation

Legal language

The prosecutor had difficulty proving mens rea, as the defendant seemed unaware that his actions were illegal.

The judge dismissed the case, citing the principle of res judicata as the issue had already been decided in a previous trial.

Despite the signed agreement, the contract was invalid as there was no consideration exchanged between the parties.

Medical language

After a strenuous workout, the patient experienced severe myalgia that lasted for several days.

After the biopsy, the doctor confirmed that the tumor was malignant and recommended immediate treatment.

Sig: 1 tab po qid pc & hs



Take one tablet by mouth four times a day, after meals, and at bedtime.

This last example of medical language may just look like a string of random characters, but it's actually a shorthand used by doctors to write prescriptions. This text has a very clear meaning to a pharmacist, take one tablet by mouth four times a day, after meals and at bedtime.

BloombergGPT: domain adaptation for finance

BloombergGPT
is used for
building domain
specific LLMs

BloombergGPT: A Large Language Model for Finance

Shijie Wu^{1,*}, Ozan İrsoy^{1,*}, Steven Lu^{1,*}, Vadim Dabravolski¹, Mark Dredze^{1,2},
Sebastian Gehrmann¹, Prabhanjan Kambadur¹, David Rosenberg¹, Gideon Mann¹

¹ Bloomberg, New York, NY USA

² Computer Science, Johns Hopkins University, Baltimore, MD USA

gmann16@bloomberg.net

Abstract

The use of NLP in the realm of financial technology is broad and complex, with applications ranging from sentiment analysis and named entity recognition to question answering. Large Language Models (LLMs) have been shown to be effective on a variety of tasks; however, no LLM specialized for the financial domain has been reported in literature. In this work, we present BLOOMBERGGPT, a 50 billion parameter language model that is trained on a wide range of financial data. We construct a 363 billion token dataset based on Bloomberg's extensive data sources, perhaps the largest domain-specific dataset yet, augmented with 345 billion tokens from general purpose datasets. We validate BLOOMBERGGPT on standard LLM benchmarks, open financial benchmarks, and a suite of internal benchmarks that most accurately reflect our intended usage. Our mixed dataset training leads to a model that outperforms existing models on financial tasks by significant margins without sacrificing performance on general LLM benchmarks. Additionally, we explain our modeling choices, training process, and evaluation methodology. As a next step, we plan to release training logs (Chronicles) detailing our experience in training BLOOMBERGGPT.

This is the research that is published in 2023 where BloombergGPT is pre-trained for domain adaptation OF Finance.

~51%

**Financial
(Public & Private)**

~49%

**Other
(Public)**

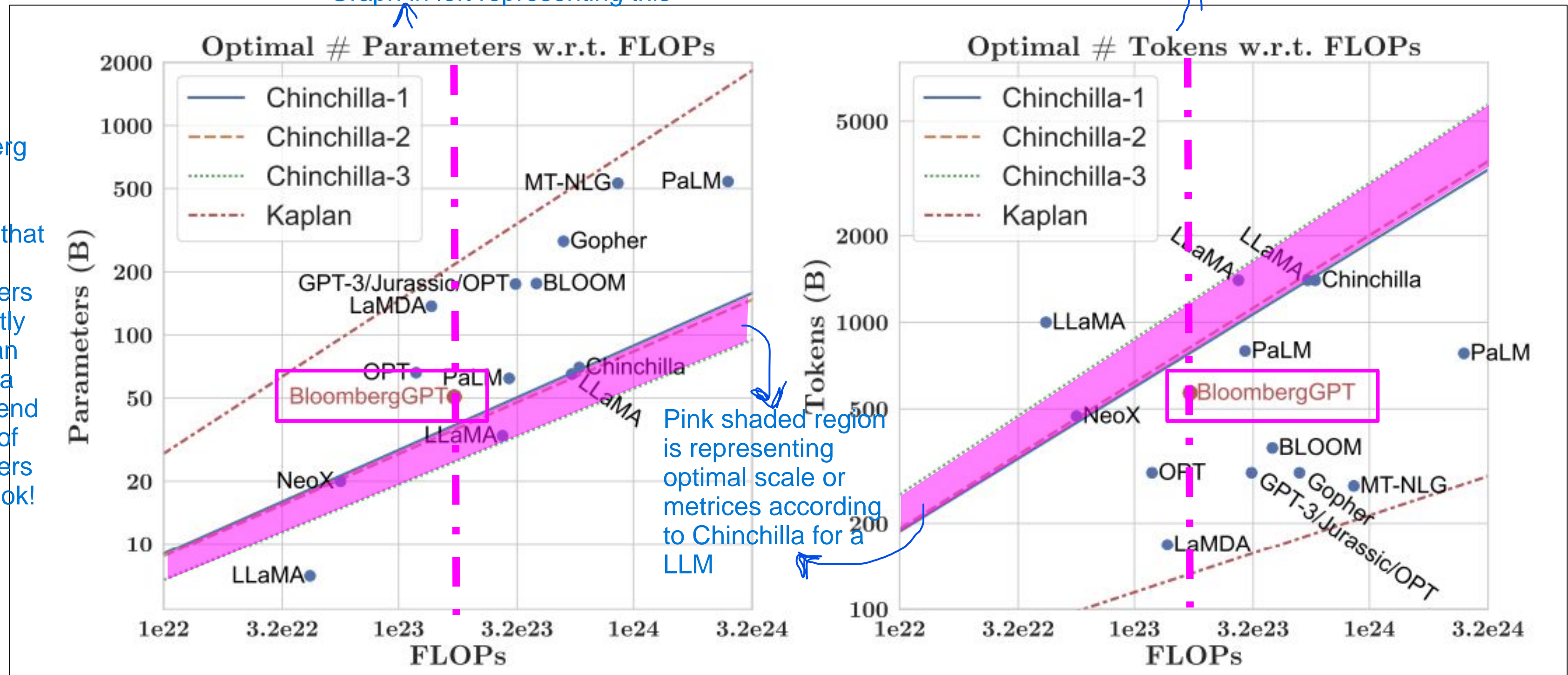
Here model was pre-trained using dataset that has vocabulary containing 51% belonging to Financial domain whereas 49% generally used daily vocabularies. In this was we are giving equal importance expecting that model will work fine for both financial domain language and general daily used language

BloombergGPT relative to other LLMs

Graph in left representing this

Graph in right representing this

For Bloomberg we can clearly observe that actual parameters are slightly more than Chinchilla recommend number of parameters which is ok!



Source: Wu et al. 2023, "BloombergGPT: A Large Language Model for Finance"

This graph is explained in next slide which is very important

But on the other hand actual Tokens or Training dataset size for BloombergGPT is lower as compared to that of Chinchilla's recommended token size. It's important to note that this does not mean that Bloomberg is bad here. We are getting less token size or training dataset because there are not enough Financial vocabularies available. Hence, we can conclude that we should not follow Chinchilla's recommended metrics hardly instead can do required adjustments or tradeoffs as well. Here one can think that if we don't have enough financial vocabularies then let's increase the generally used daily vocabularies in the training dataset. Say with this intuition we prepared training dataset consisting of 70% daily used vocabulary and 30% financial vocabularies. Now our model will be having token size or training dataset in accordance with Chinchilla's recommendation. On first view it may seem fine but let me give another angle as well. We are solving financial use case so ideally training dataset should have more finance related vocabularies that are not present in daily used vocabularies so that model can build deep contextual understanding more w.r.t financial domain and not w.r.t General used languages. Therefore this case will be omitted and the same explains why we should not follow Chinchilla's recommendation (called Chinchilla Scaling Laws) hardly and should be making necessary tradeoffs

Key takeaways



Ques: Which of the following statements about pretraining scaling laws are correct? Select all that apply

Ans:

correct ans: To scale our model, we need to jointly increase dataset size and model size, or they can become a bottleneck for each other.

correct ans: There is a relationship between model size (in number of parameters) and the optimal number of tokens to train the model with.

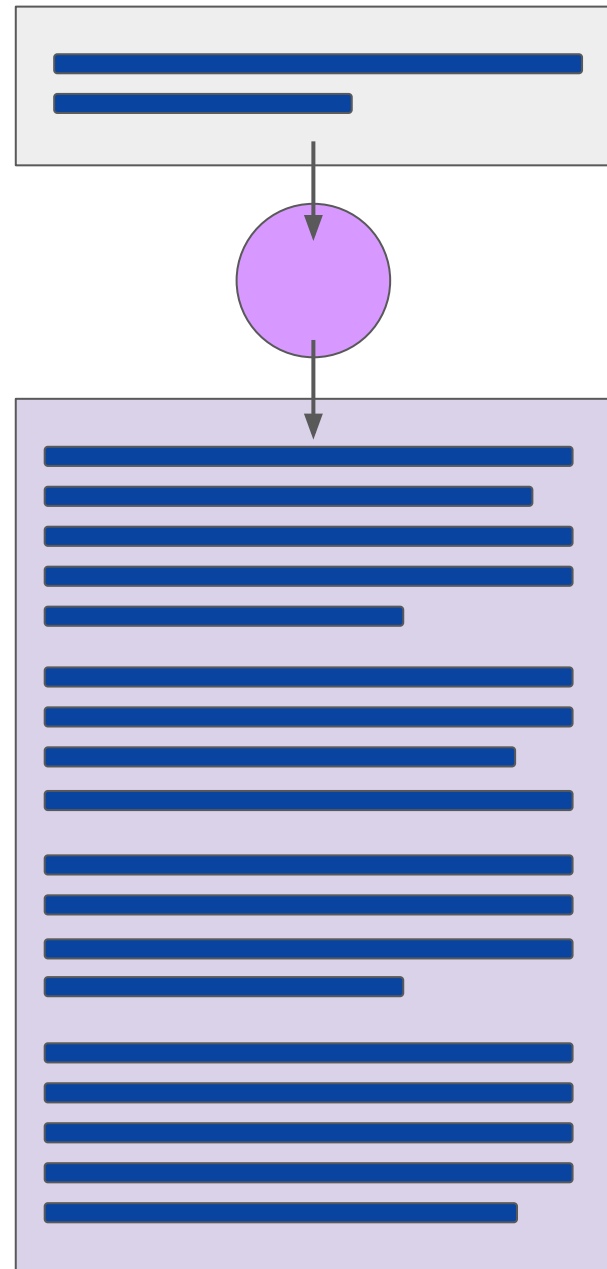
correct ans: When measuring compute budget, we can use "PetaFlops per second-Day" as a metric.

✘✘✘ wrong ans: You should always follow the recommended number of tokens, based on the chinchilla laws, to train your model.

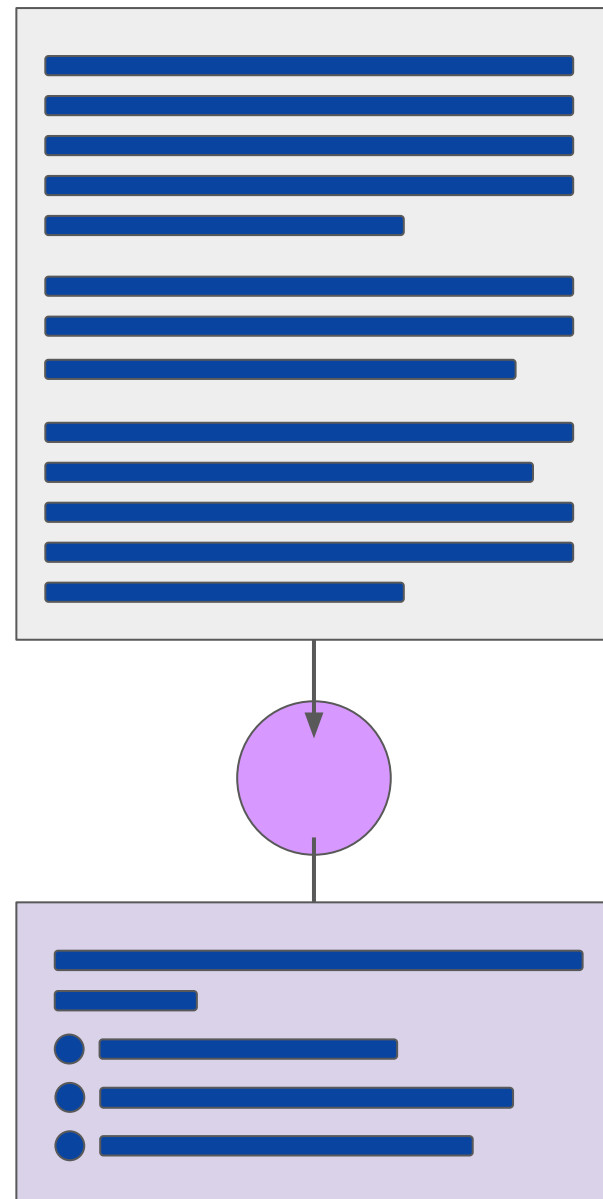
expalination for wrong example: Although compute optimization is important, it can be challenging to obtain a sufficient amount of data tokens. In the case of BloombergGPT, they had a limited token count and even used fewer tokens due to early stopping. While chinchilla laws offer valuable guidance, they should not be strictly followed as rigid rules.

LLM use cases & tasks

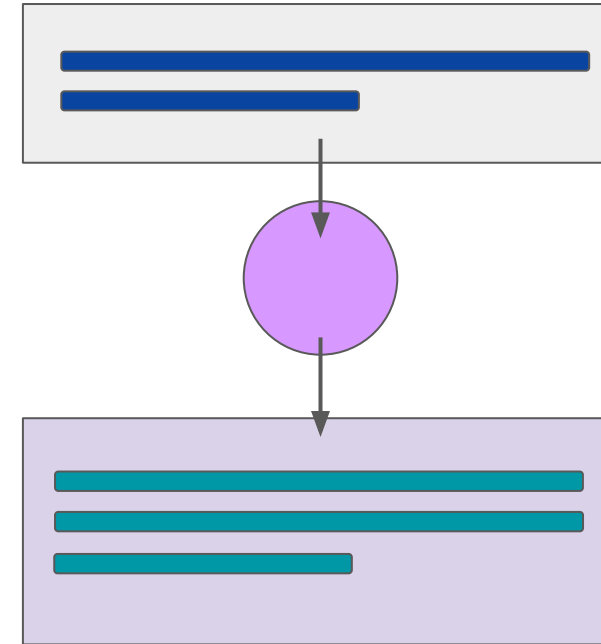
Essay Writing



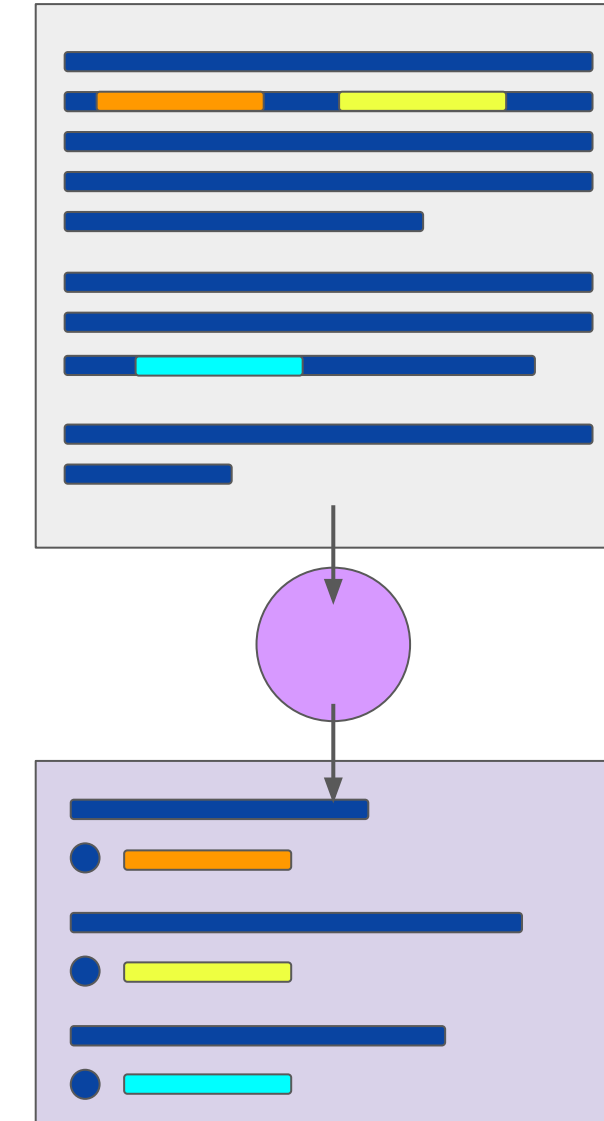
Summarization



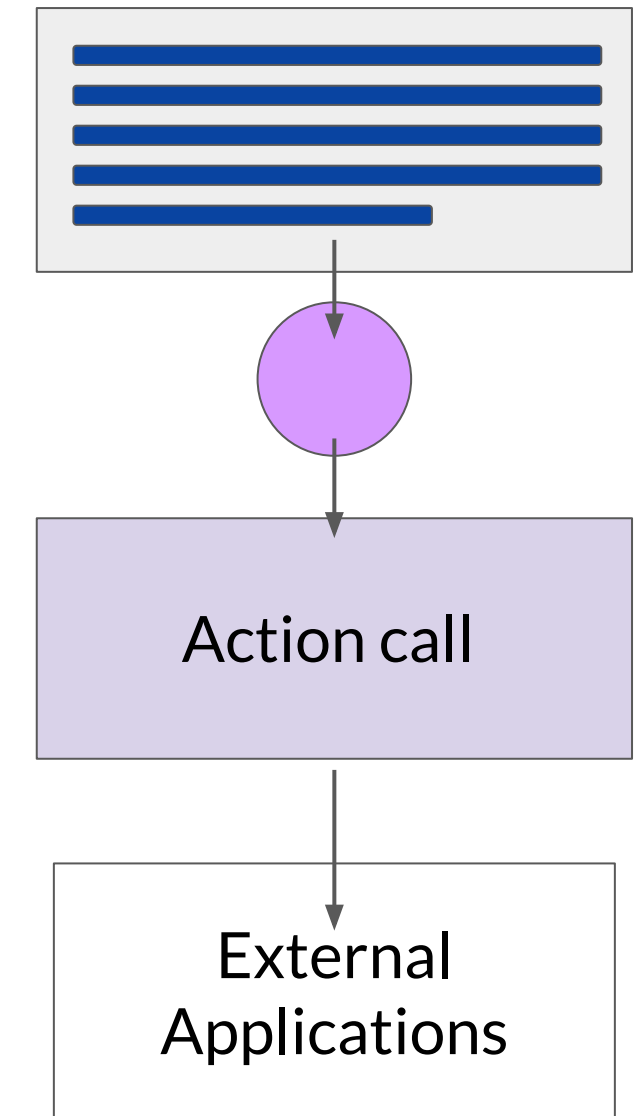
Translation



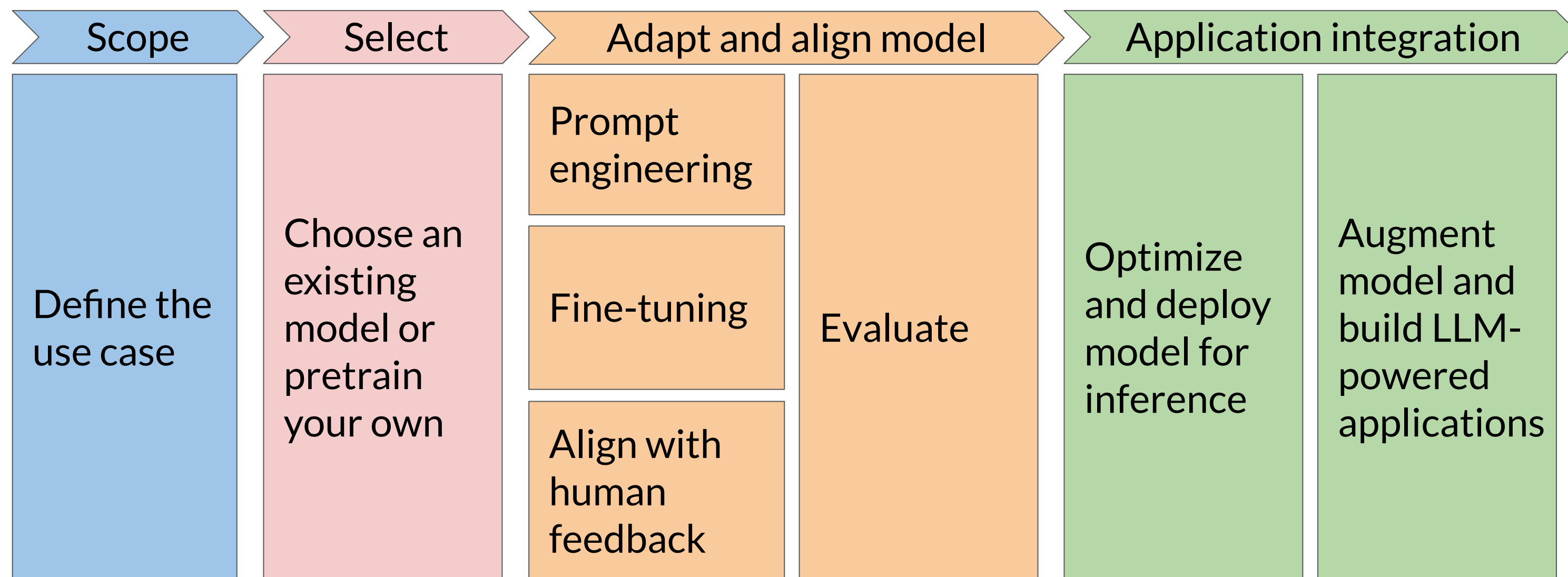
Information retrieval



Invoke APIs and actions



Generative AI project lifecycle



BloombergGPT: A Large Language Model for Finance: Research Paper link: <https://arxiv.org/pdf/2303.17564.pdf>