Generative AI
&
Large Language Models

(Part I)

Ana Maria Sousa

Conteúdo

1. Generative AI

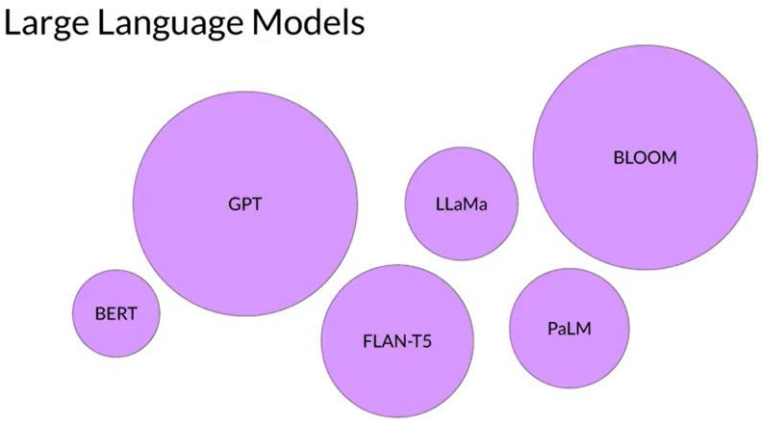
Generative AI is a subset of traditional ML (machine learning) where models are trained to create content that mimics human abilities. The ML algorithms that work behind generative AI do so by exploiting the statistical patterns present in the massive datasets of content that was originally generated by humans.

2. LLMs

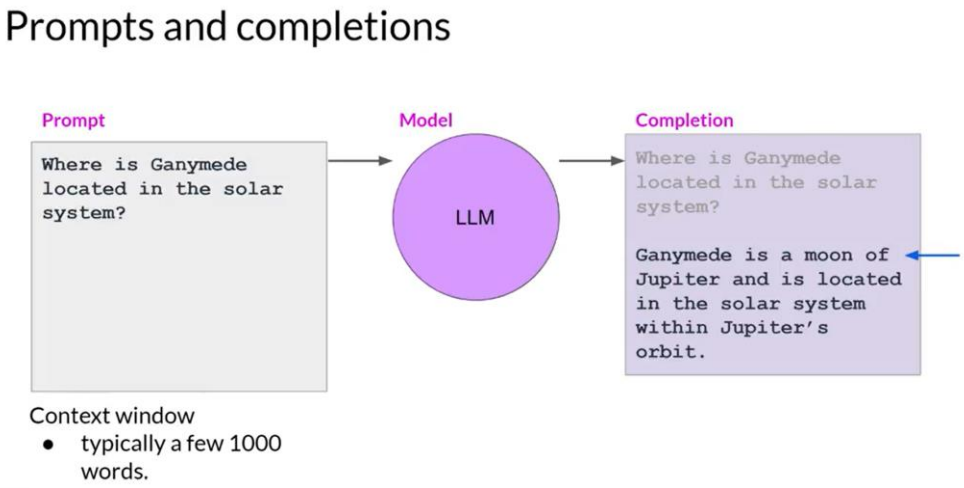
2.1. Definition

Large language models are a specific type of generative AI model that have been trained on massive amounts of text data. All LLMs are powered by the Transformer (Google, 2017) architecture. They are designed to take in input text and repeatedly generate the next token or word that appropriately "completes" the input text.

It is commonly accepted that as the size (in terms of number of parameters) of an LLM increases, so does its understanding of language. At the same time, it is also true the smaller models can be fine-tuned to perform well on specific tasks.



These models can understand and generate human-like text based on the prompts or instructions given to them. These models are trained on a lot of data and can understand and generate human-like language.



2.2. Terminology

2.2.1. Prompt

The input given to an LLM is called the prompt.

2.2.2. Context Window

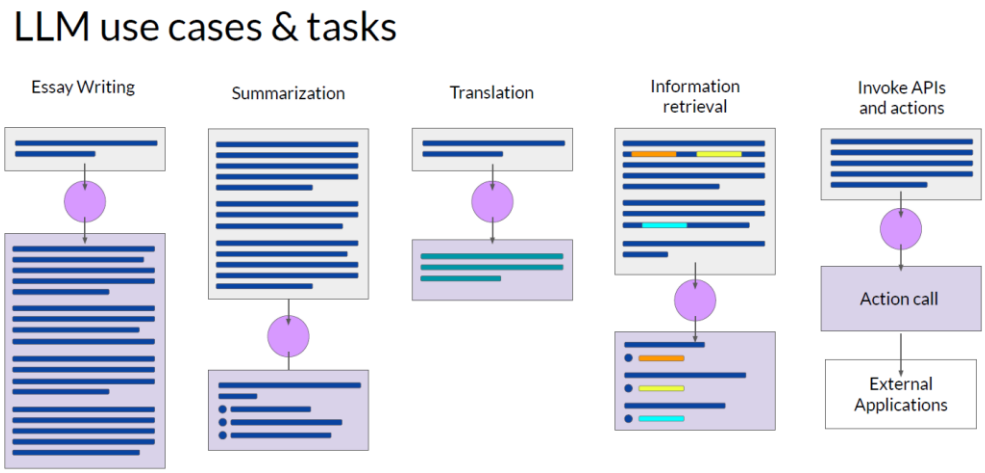
The space/memory that is available to the prompt is called the context window. It is essentially the maximum size of the prompt that the model can handle before it performs poorly. It is limited to a few thousand of words but also varies model to model.

2.2.3. Completion

The output of an LLM when given a prompt is called the completion. Generally, the completion consists of the prompt and the text generated by the model by repeatedly generating the next token or word, though almost all applications omit the prompt from the model's output when showing it to users.

2.3. Use Cases.

They can be used for tasks like chatbots, writing essays, summarizing conversations, translating languages, and even writing code.

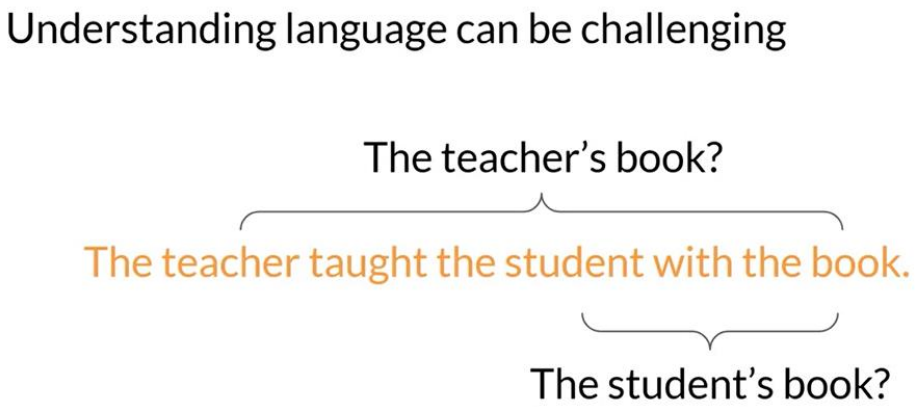


3. Transformers

3.1. Before Transformers

Text generation is not a new paradigm. Before Transformers, text generation was carried out by Recurrent Neural Networks (RNNs). RNNs were capable at their time but were limited by the amount of compute and memory needed to perform generative tasks.

Besides that, language is complex, some words have multiple meanings, homonyms and Words within a sentence structure can be ambiguous or have what we might call syntactic ambiguity. Thus, it's only with the context of the sentence that we can see what kind of bank is meant.



3.2. Why Transformers?

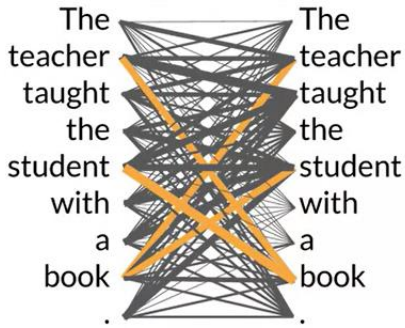
- These kinds of problems are solved (to an extent, of course) by Transformers.
- Can scale efficiently to use multi-core GPUs.
- Can parallel-process input data, allowing the use of massive datasets efficiently.
- Pay "attention" to the input meaning, allowing for better models which can generate more meaningful and relevant text.

3.3. Key Concepts

3.3.1. Self-attention

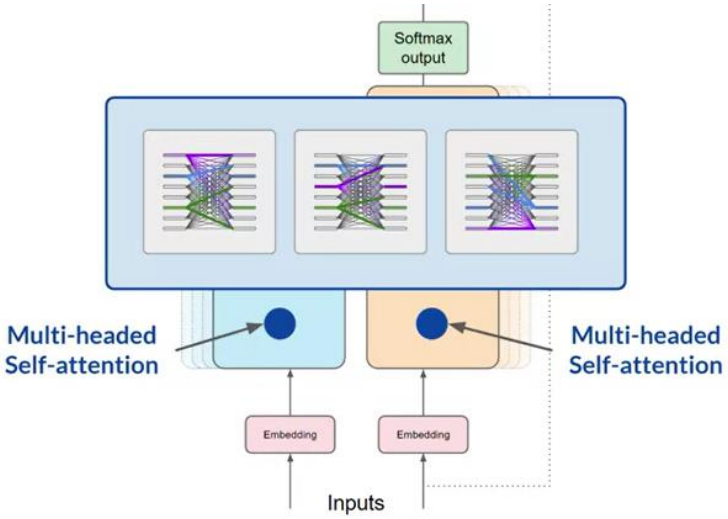
The power of the Transformer architecture lies in its ability to learn the relevance and context of all the words in the prompt with each other. The context is learnt not just with their immediate neighbor, but with every other word.

The model applies "attention weights" to the relationships so that it learns the relevance of each word to every other word. These "attention weights" are learned during training. This is called self-attention. The term originates from the fact that each word in the prompt attends to other words in the same prompt, including itself. This mechanism is what enables Transformers to capture relationships and dependencies between words regardless of their distance from each other in the prompt.



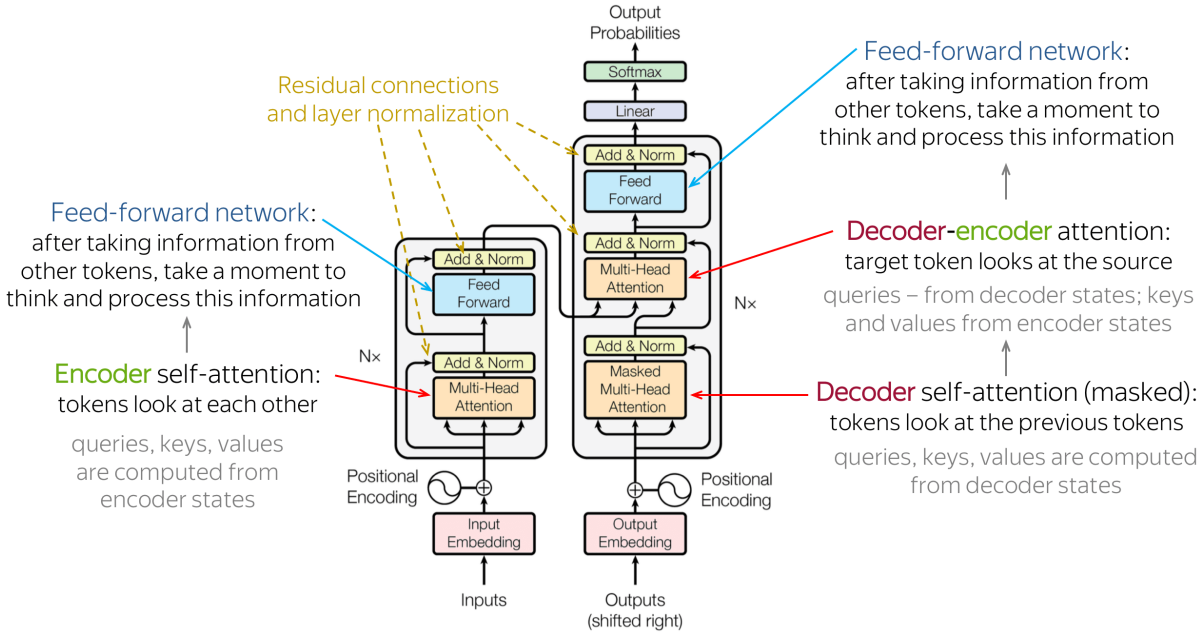
3.3.2. Multi-Headed Attention

Multi-Headed Attention is essentially a for-loop over self-attention. Intuitively, we have multiple questions we'd like to find the best answer for.



Multi-headed attention has no information about the relative position of words in the input sequence. But position can be extremely important in a sentence. Thus, transformers have a positional encoding step.

3.4. Transformers Architecture



The Transformer is an encoder-decoder architecture. Both the encoder and decoder blocks are repeated N times. Typically, N = 6.

3.5. How does Transformers work?

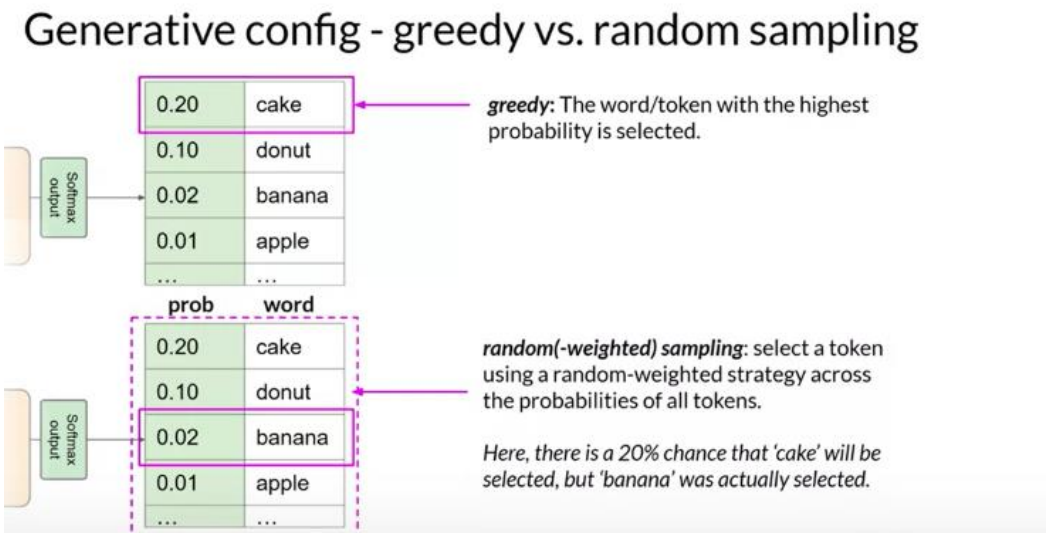
1. Tokenization: the process of converting a sequence of text into smaller parts, known as tokens.
2. Embedding
3. Positional Encoding
4. Self-attention
5. Multi-Headed Self-Attention

6. Feed-Forward Neural Network (FFNN): It is applied to each element in the input sequence separately and identically. The FFNN adds complexity to the model and can be thought of as an extra step of "pre-processing" applied on the output of multi-headed attention.
7. Layer Normalization: A layer normalization is applied to the added output. This is preferred over batch normalization since the batch size is often small and batch normalization tends to perform poorly with text since words tend to have a high variance (due to rare words being considered for a good distribution estimate).
8. Masked Multi-Headed Attention: The first multi-headed attention layer uses masking during training. This allows parallel training. During training, we have the entire expected output sequence. Thus, we do not need to predict word by word. We can instead feed the entire output sequence to the decoder.
9. Cross-Attention
10. Linear + Softmax – Prediction: There is a final linear layer followed by a softmax activation. This converts the output of the decoder to a probability distribution over all the words. In other words, if the dictionary of words has N words, this layer has N units, for each word.

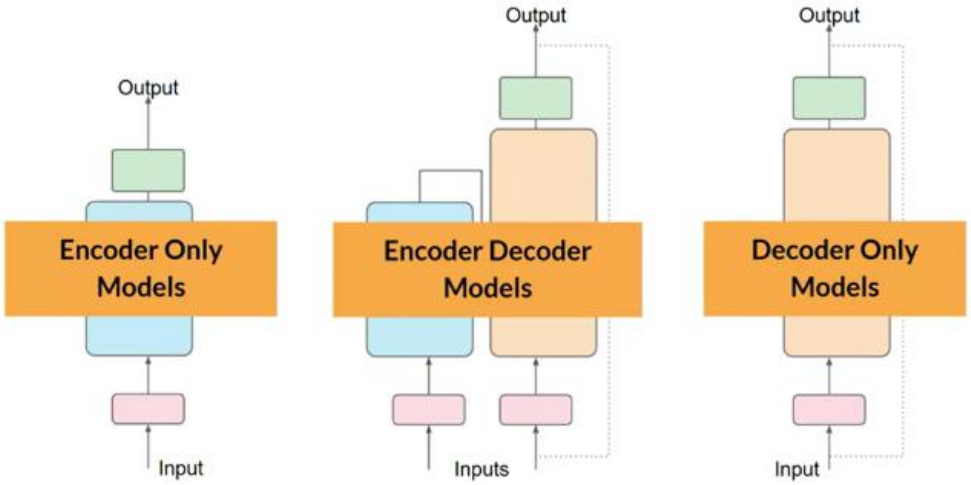
The next word can be predicted from this probability distribution by either taking the one with the maximum probability or using other techniques.

- **Greedy Sampling** : The technique of using the word with the maximum probability is called greedy sampling. This is the most commonly used technique for many models.

- **Random Sampling** : Another approach is called random-weighted sampling. It introduces some variability to the model's output. Instead of taking the word with the maximum probability, the probabilities are used as weights to sample one word at random.



3.6. Types of Configurations of Transformers



3.6.1. Encoder-only Models

They only have encoders. Without some changes, these models always produce an output which is of the same length as the input sequence.

Tasks: It is possible to modify these so that they can be used for tasks such as semantic classification. Sentiment analysis; Named entity recognition; Word classification.

Models: BERT, ROBERTA

3.6.2. Encoder-Decoder Models

This is the model originally described in the Transformers paper and the one detailed here. The output sequence and the input sequence can be of different lengths.

Tasks: It is useful for sequence-to-sequence tasks such as machine translation. Translation, Text summarization, Question answering.

Models: BART, FLAN-T5, T5.

3.6.3. Decoder-only Models

These are some of the most used models. The pretraining of decoder models usually revolves around predicting the next word in the sentence.

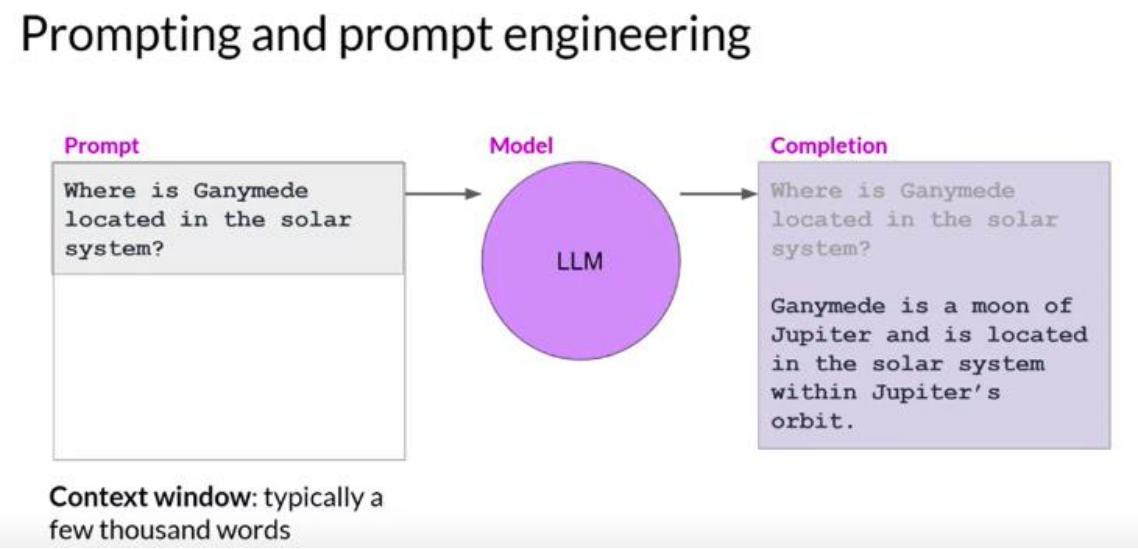
Tasks: These models are best suited for tasks involving text generation.

Models: GPT, BLOOM, LLaMA

4. Prompting & Prompt Engineering

Prompt: The text that is fed to LLMs as input is called the prompt and the act of providing the input is called prompting.

Prompt Eng. : Iterative process of refining and adjusting the prompts provided to a Language Model to elicit the most accurate, relevant, and desirable outputs. This process involves various techniques aimed at guiding the LM's generation towards desired outcomes.



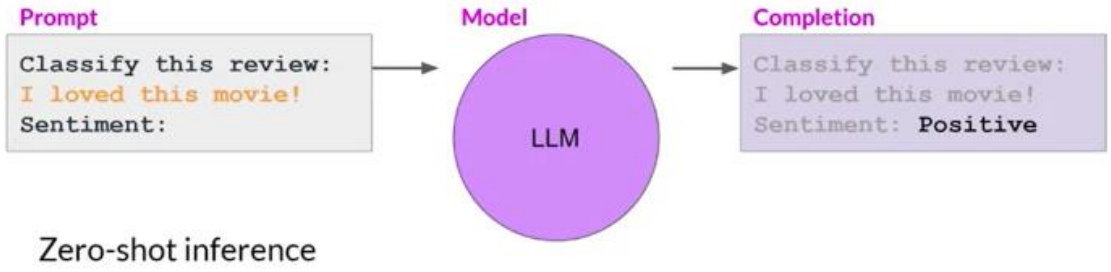
4.1. In-Context Learning (ICL)

In ICL, we add examples of the task we are doing in the prompt. This adds more context for the model in the prompt, allowing the model to "learn" more about the task detailed in the prompt.

For example, if we want the model to classify the sentiment of a movie review, we can include an example review in the prompt. This helps the model understand the task better and generate the desired output.

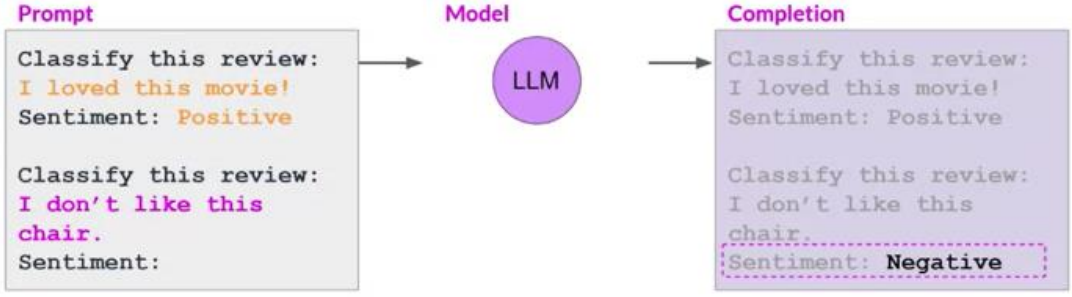
There are different levels of in-context learning. **Zero-shot inference** means the model can understand the task without any examples. **One-shot inference** means we include a single example, and **few-shot inference** means we include multiple examples. These techniques can help smaller models perform better.

4.1.1. Zero-Shot Inference



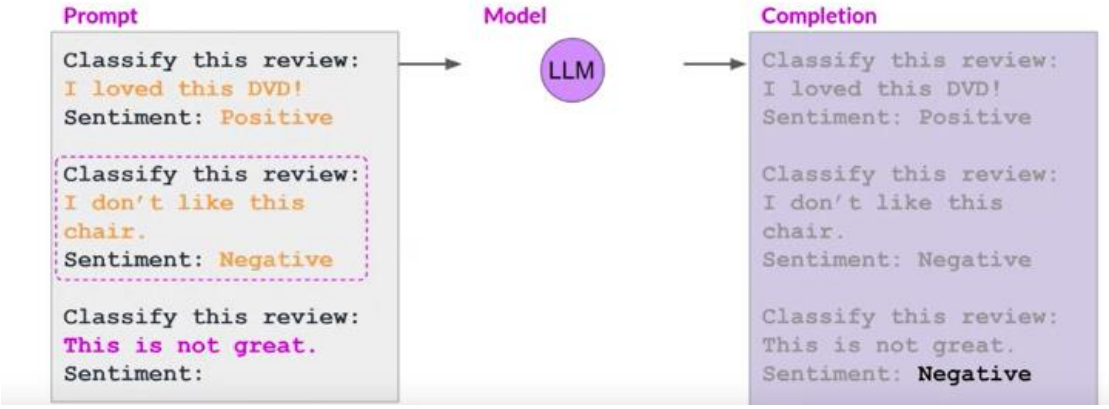
This prompt works well with large LLMs but smaller LLMs might fail to follow the instruction due to their size and fewer number of features.

4.1.2. One-Shot Inference



This is where ICL comes into play. By adding examples to the prompt, even a smaller LLM might be able to follow the instruction and figure out the correct output. Sometimes, a single example won't be enough for the model, for example when the model is even smaller. We'd then add multiple examples in the prompt.

4.1.3. Few-Shot Inference



In summary:

- Larger models are good at zero-shot inference.
- For smaller models, we might need to add examples to the prompt, for few-shot inference.
- There is a limit to how much information we can include in the prompt. If we find that including too many examples doesn't improve the model's performance, we can try fine-tuning the model.
- Fine-tuning is a process of training the model further with new data to make it better at the specific task we want it to perform.
- The size of the model also plays a role in its performance. Larger models can perform multiple tasks and understand language better. Smaller models are generally good at a limited number of tasks.

Summary of in-context learning (ICL)

4.2. Generative configuration parameters for inference

4.2.1. Max New Tokens

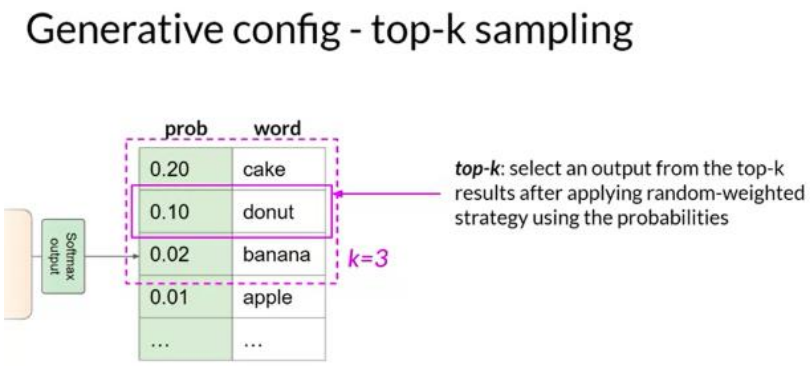
This is used to limit the maximum number of new tokens that should be generated by the model in its output. The model might output fewer tokens (for example, it predicts <EOS> before reaching the limit) but not more than this number.

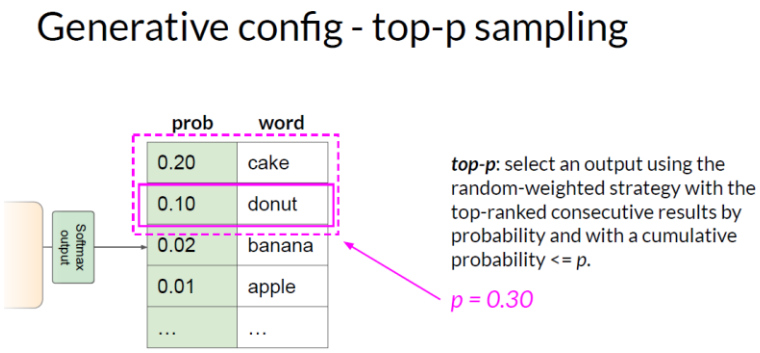
4.2.2. Sample Top-K and Sample Top-P

Sample Top-K and Sample Top-P are used to limit the random sampling of a model.

A top-K value instructs the model to only consider K words with the highest probabilities in its random sampling. his allows the model to have variability while preventing the selection of some highly improbable words in its output.

The top-P value instructs the model to only consider words with the highest probabilities such that their cumulative probability.



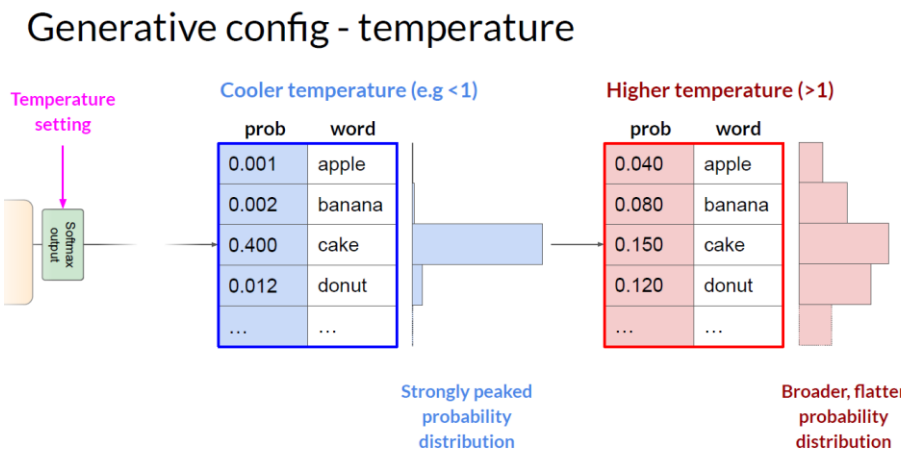


4.2.3. Temperature

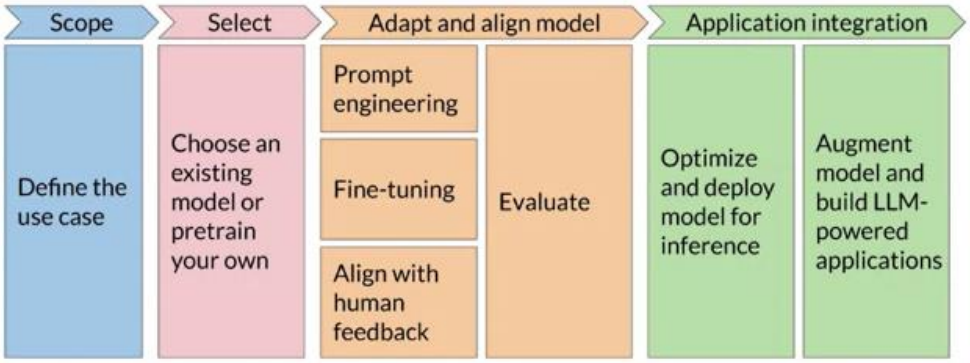
It determines the shape of the probability distribution that the model calculates for the next word. Intuitively, a higher temperature increases the randomness of the model while a lower temperature decreases the randomness of the model.

Cooler temperature (T < 1): the probability distribution is strongly peaked. In other words, one (or a few more) words have very high probabilities while the rest of the words have very low probabilities.

Warmer temperature (T > 1): he probability distribution is broader, flatter and more evenly spread over the tokens.



5. Generative AI project lifecycle



5.1.1. Scope

Define the scope of the project as accurately and as narrowly as we can. LLMs are capable of carrying out various tasks but their ability is dependent by their size and architecture. We need to think about the function(s) that the LLM will have in our application.

5.1.2. Select

Decide whether we want to train our own model from scratch or work with an existing base model. In general, we start with a base model except in a few cases where it might be necessary to train our own model.

5.1.3. Adapt and align model.

We need to evaluate the performance of our model and carry out additional training as needed. Sometimes, prompt engineering could be enough to get our model to perform well. Thus, in general, we'd start with trying, providing examples suitable for our task(s) in the prompt.

There are cases where the model may not perform as well as we need even with ICL. In those cases, we can try fine-tuning the model on some dataset specific to our task, using a supervised learning.

Additionally, to ensure the model responds naturally and safely (no racism, no harmful advice, etc), we might also need to align the model with human feedback. This is achieved through Reinforcement Learning with Human Feedback (RLHF).

For example, we can start with prompt engineering and then evaluating the model's performance. We can then use fine-tuning to improve the model's performance, and then revisit and evaluate prompt engineering one more time to get the performance we need.

5.1.4. Application integration

An important step is to optimize the model for deployment. This ensures optimal utilization of computing resources and guarantees an exceptional user experience. Additionally, it's imperative to factor in any supplementary infrastructure necessary for our Language Learning Model (LLM).

For instance, LLMs often tend to generate fabricated information when faced with unknown queries or struggle with intricate reasoning and mathematical tasks. Thankfully, there exist potent strategies to mitigate or minimize the repercussions of these limitations.

6. Pre-training Large Language Models

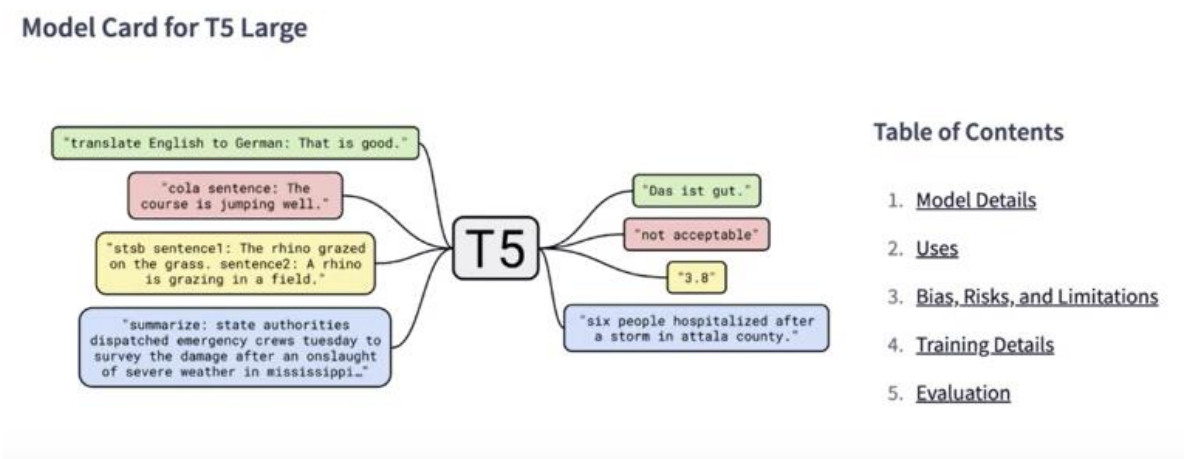
6.1. Choosing a Model

After figuring out the scope of our application, the next step is to select the model we will work with. We have two options:

- Choose a pre-trained foundation model.
- Train our own model to create a custom LLM.

There are specific use cases where the second option might make more sense but in general, we will develop our application using a pre-trained foundation model.

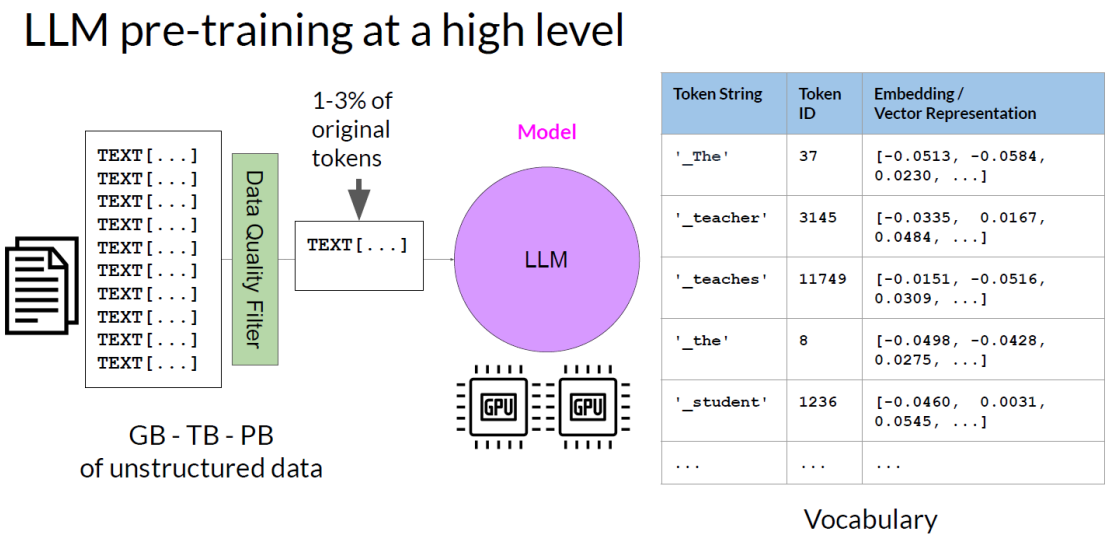
Many of the developers of these models have made available **"hubs"** where we can browse and test the models. One of the most useful features of these hubs is the inclusion of **model cards**. **Model cards** describe important details of a model such as the best use case of the model, how it was trained and its limitation.



6.2. Initial Training Process

The pre-training process is self-supervised. The model internalizes the patterns and structures present in the language. These patterns then unable the model to complete its training objective, which depends on the architecture of the model.

Additionally, since the data is coming from public sources such as the internet, there is often a data quality filter applied before feeding the data to the LLM so that the training data is of high quality, has low bias and does not have harmful content. Due to this, only about 1-3% of the original tokens are used for pre-training.



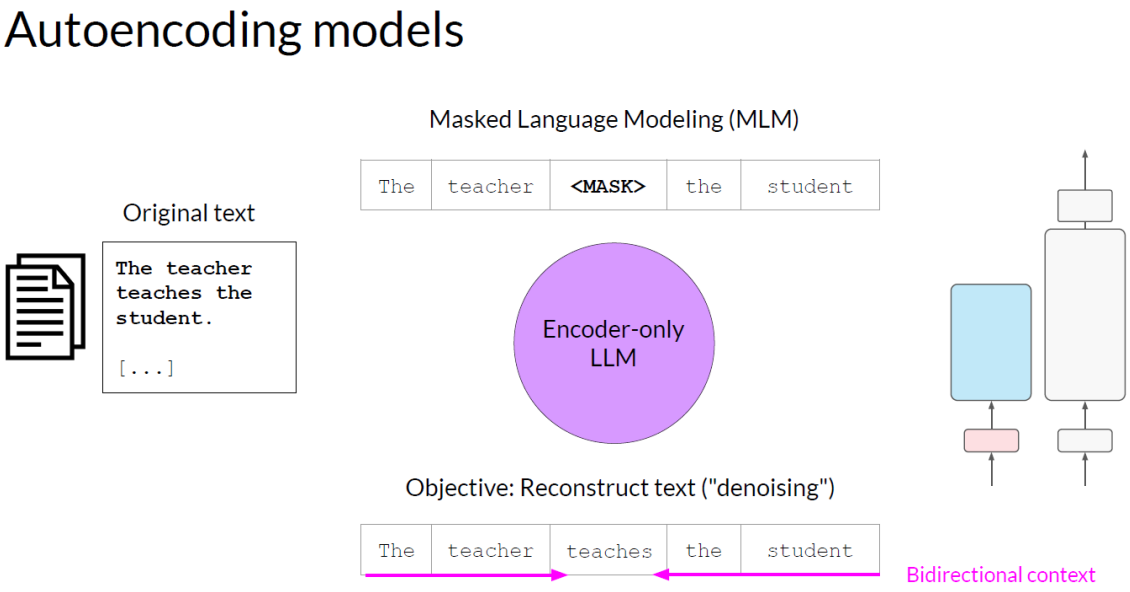
6.3. Training Objectives for Transformer Variants

6.3.1. Encoder-only Models (Autoencoding Models)

They are pre-trained using **Masked Language Modeling (MLM).** In MLM, tokens in the input sequence are randomly masked and the training objective is to predict the masked tokens in order to reconstruct the original input sequence.

This is also called **a denoising objective** since the masking of the tokens can be thought of as adding noise to the input sequence and then predicting the masked tokens can be thought of as removing that noise from the input sequence.

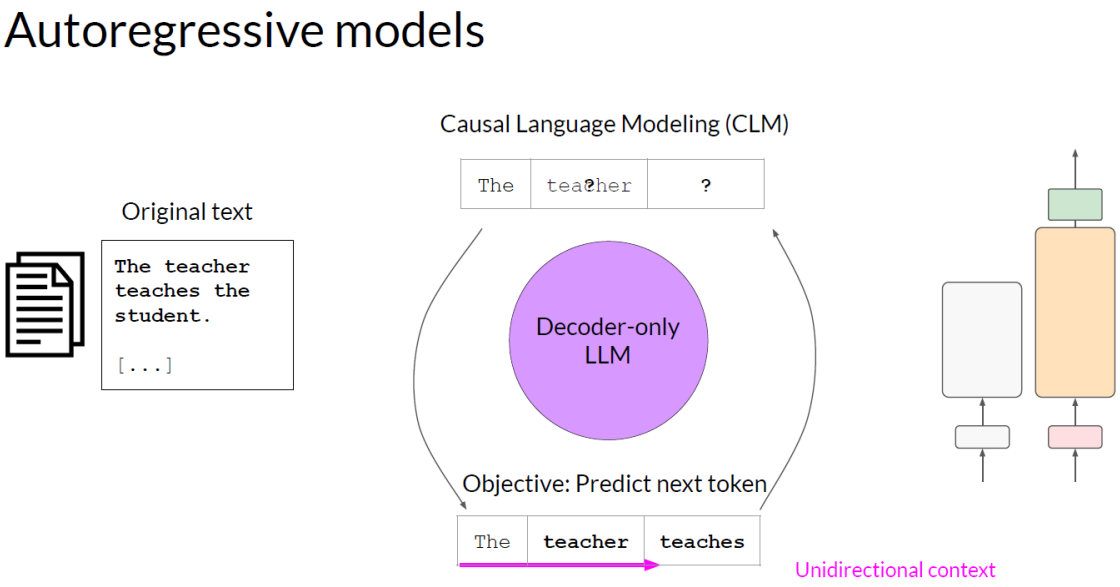
Autoencoding models build bidirectional context representations of the input sequence, meaning that model has an understanding of the full context of the token rather than just the tokens that come before it.



6.3.2. Decoder-only Models (Autoregressive Models)

Autoregressive models, also known as decoder-only variants of Transformers, are trained using a technique called **Causal Language Modeling (CLM).**

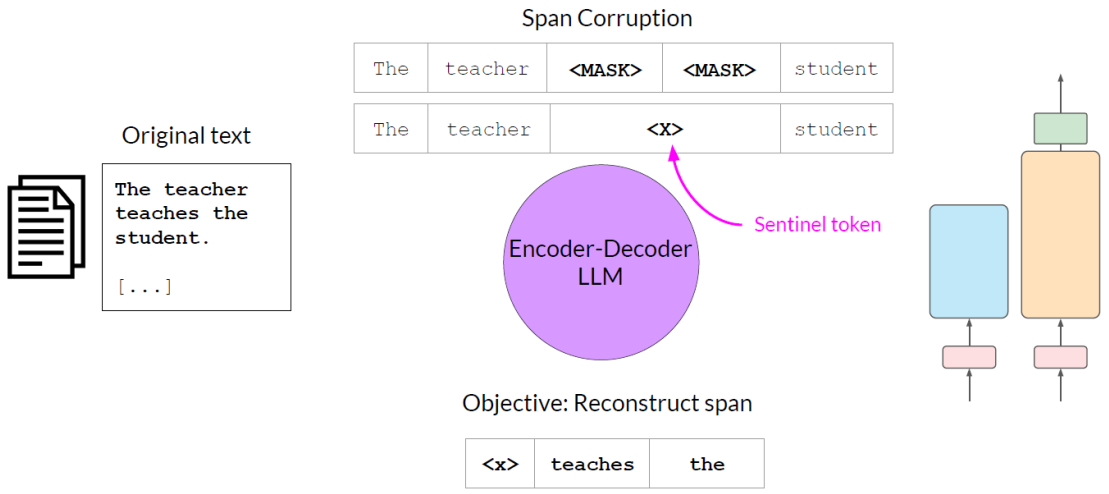
In CLM, the model's objective during training is to predict the next token based on the preceding sequence of tokens. The tokens of the input sequence are masked, and the model can only see the input tokens leading up to the token being predicted at the moment. The model has no knowledge of the tokens that come after this token. The model then iterates over the input sequence one-by-one to predict the next token. Thus, in contrast to autoencoding models, the model builds a unidirectional context for each token.



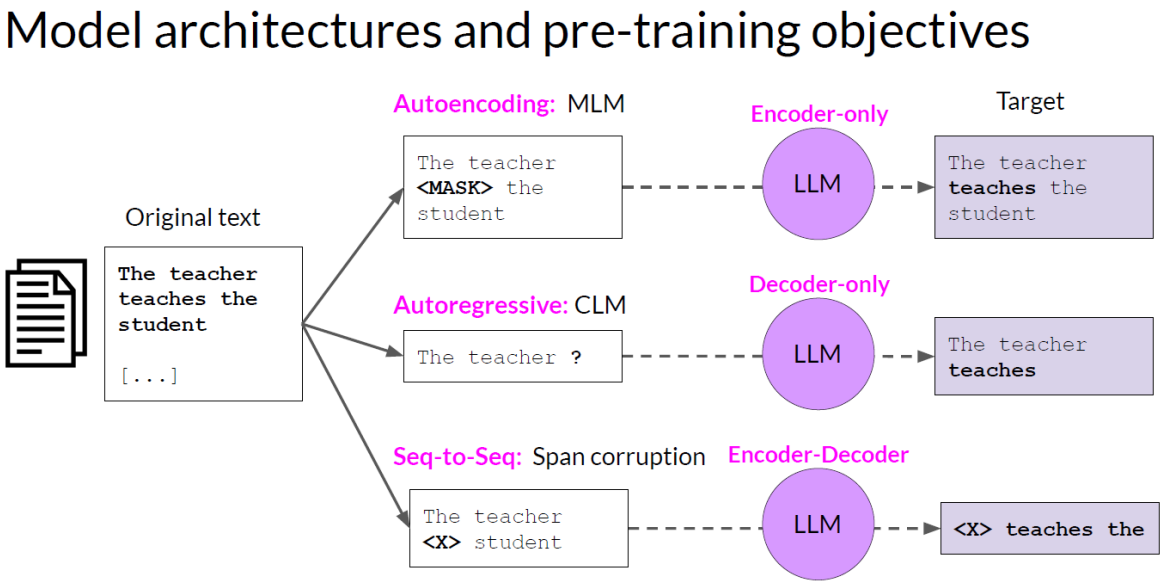
By learning to predict the next token from a vast number of examples, the model builds a statistical representation of the language. Predicting the next token is sometimes called full language modeling by researchers.

6.3.3. Encoder-Decoder Models (Sequence-to-Sequence Models)

The encoder-decoder variants of Transformers are also called sequence-to-sequence models. The exact details of pre-training objective vary from model to model.



For example, **FLAN-T5** is trained using span corruption. In **span corruption,** a part of the input sequence is masked and replaced by a sentinel token. These **sentinel tokens** are special tokens added to the vocabulary that to do not correspond to any actual word from the dataset. The decoder then must reconstruct the sentence autoregressively. The output is the **sentinel token** followed by the predicted tokens.



7. Computational Challenges in Training LLMs

LLMs have a lot of parameters and thus, require vast compute resources for training. One common issue is memory. Many models are too big to be loaded into a single GPU's memory.

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
Activations and temp memory (variable size)	+8 bytes per parameter (high-end estimate)
TOTAL	=4 bytes per parameter +20 extra bytes per parameter

There is a popular technique to reduce memory usage.

7.1.1. Quantization

One way to reduce the memory needed is through a technique called quantization. Quantization reduces the precision of the model weights, which means it uses fewer bits to represent the numbers. This helps to reduce the memory required to store the model.

There are different levels of quantization, such as using 16-bit floating point numbers or even 8-bit integer numbers. By using lower precision, you can significantly reduce the memory footprint of the model.

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

FLAN T5

- Reduce required memory to store and train models
- 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

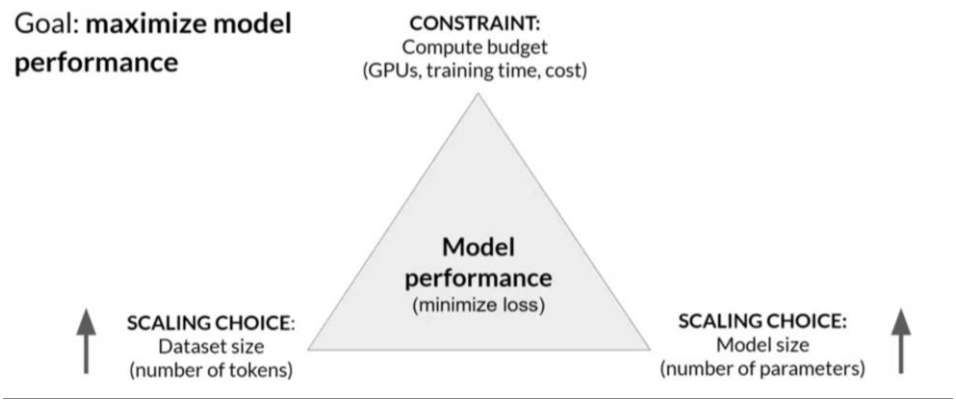
Another option is to use a datatype called BFLOAT16, which is a hybrid between 16-bit and 32-bit precision. BFLOAT16 maintains the dynamic range of 32-bit precision but uses only 16 bits, which saves memory and improves performance.

7.1.2. Scaling Choices

The goal during pre-training an LLM is to maximize the model's performance on its learning objective. This is equivalent to minimizing the loss function.

There are two choices to achieve better performance:

- **Increasing the dataset size** in terms of number of tokens.
- **Increasing the model size** in terms of number of parameters.



8. Compute-Optimal Models

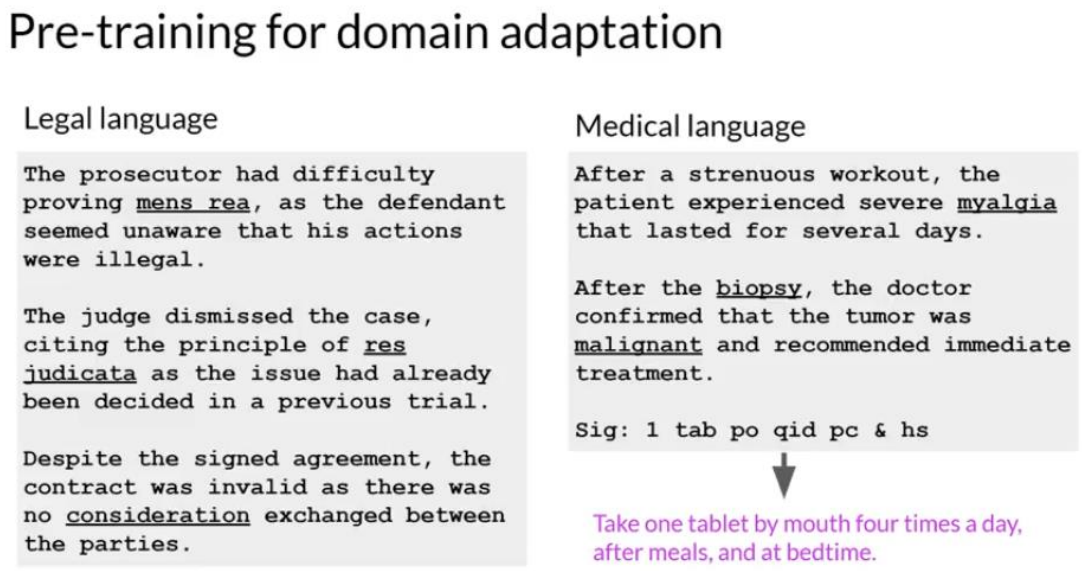
In the paper-Training Compute-Optimal Large Language Models (DeepMind, 2022, popularly referred to as the Chinchilla paper), researchers tried to find the optimal number of parameters and volume of training data for a given compute budget. Such models are called compute-optimal models.

The paper found the following:

- Very large models maybe be over-parameterized and under-trained. They have more parameters than they need to achieve a good understanding of language and they would benefit from seeing more training data.
- Smaller models trained on more data could perform as well as large models.
- Compute-optimal training dataset size is ~20 times the number of parameters.

9. Domain Adaptation

There is a situation where it may be necessary to pre-train our own model. If the domain of the problem we are trying to solve uses vocabulary and language structures that are not commonly used in day-to-day language, we might need to train our own model.



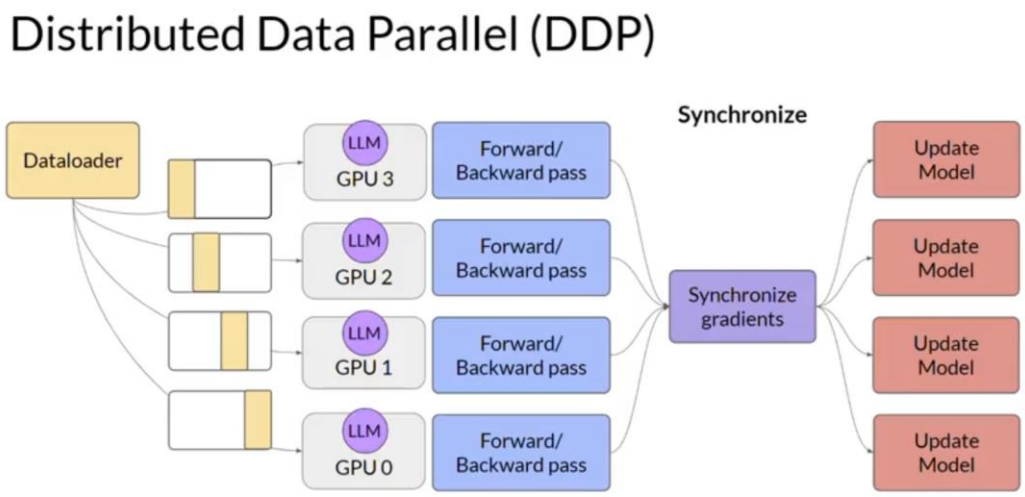
10. Efficient Multi-GPU Compute Strategies

While multiple GPUs are a must when the model is too big to fit on a single GPU, they can be applicable even when the model fits on the GPU since then, we can train the model in parallel by feeding multiple training examples in parallel.

10.1. Distributed Data Parallel (DDP)

DDP is applicable when the model weights and all the additional parameters, gradients, optimizer states, etc., fit on a single GPU. It's a way to train an LLM using parallelism.

I. The model is copied to different GPUs.
II. At the same time, different batches of data are copied to the GPUs and each batch is processed in parallel (both forward and backward pass).
III. At the end, there is a step to synchronize the gradients from each of the GPUs.
IV. These synchronized gradients are used to update the model weights on each of the GPUs, which are always identical.

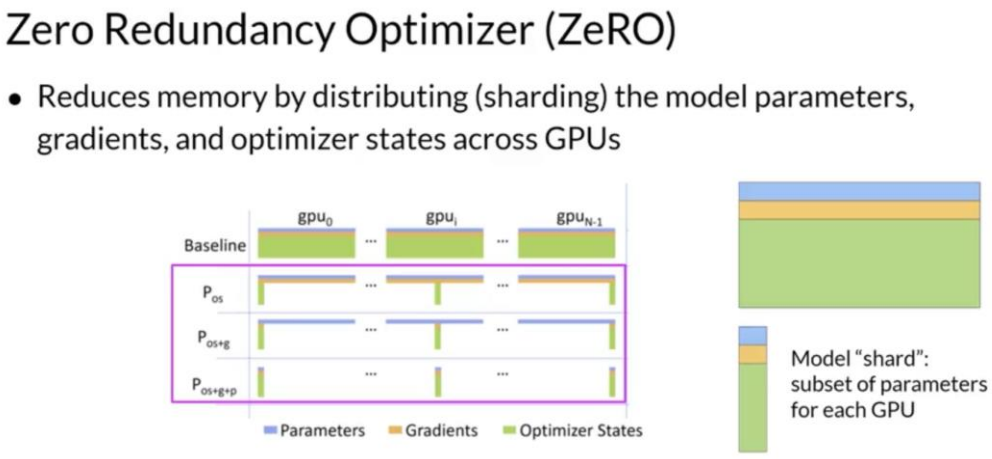


10.2. Fully Sharded Data Parallel (FSDP)

In case of DDP, a full copy of the model is stored on each GPU. In other words, every GPU will be consuming 24 bytes/parameter and will store the same numbers. Thus, there is redundant memory consumption.

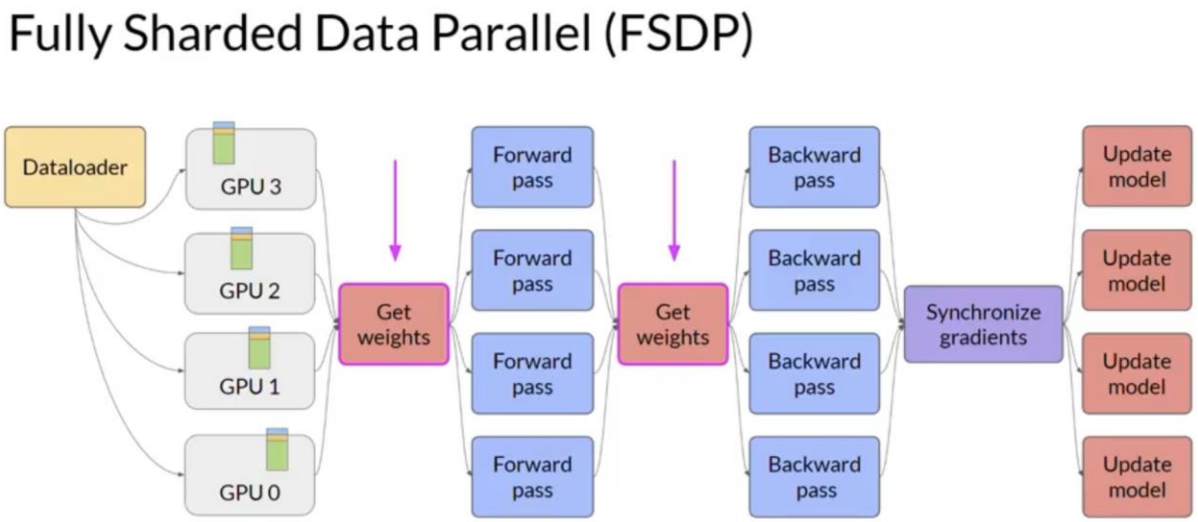
FSDP is an implementation of model sharding, inspired by the paper ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Microsoft,2019). ZeRO stands for Zero Redundancy Optimizer.

ZeRO eliminates this redundancy by distributing (sharding, instead of replicating) model parameters, gradients and optimizer states across GPUs. It also ensures that the communication overhead stays close to that in DDP.



ZeRO offers three strategies for optimization:

- **Stage 1:** Only the optimizer states are sharded across GPUs. It can reduce memory usage by up to a factor of 4.
- **Stage 2:** The gradients as well as the optimizer states are shared across GPUs. It can reduce memory usage by up to a factor of 8.
- **Stage 3:** The model parameters, gradients and optimizer states are sharded across GPUs. With Stage 1 and Stage 2, memory reduction is linear with the number of GPUs. That is, sharding across 64 GPUs could reduce memory by up to a factor of 64.


Fully Sharded Data Parallel (FSDP)
Dataloader
GPU 3
GPU 2
GPU 1
GPU 0
Get weights
Forward pass
Forward pass
Forward pass
Forward pass
Get weights
Backward pass
Backward pass
Backward pass
Backward pass
Synchronize gradients
Update model
Update model
Update model
Update model