Building LLM Applications: Large Language Models (Part 6)

Vipra Singh · Follow

38 min read · Feb 10, 2024

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Posts in this Series

1. *Introduction*
2. *Data Preparation*
3. *Sentence Transformers*
4. *Vector Database*
5. *Search & Retrieval*
6. *LLM (This Post)*
7. *Open-Source RAG*

8. *Evaluation*
9. *Serving LLMs*
10. *Advanced RAG*

. . .

Table of Contents

Greetings!

So till now, we have learnt how the Raw Data is transformed and stored in Vector Databases. Then relevant chunks are retrieved back from the Vector Database based on the user prompt. This completes the Retrieval Part of the Application.

Next, we will focus on the Generation part of the RAG Application. So for text generation, we will be using Large Language Models.

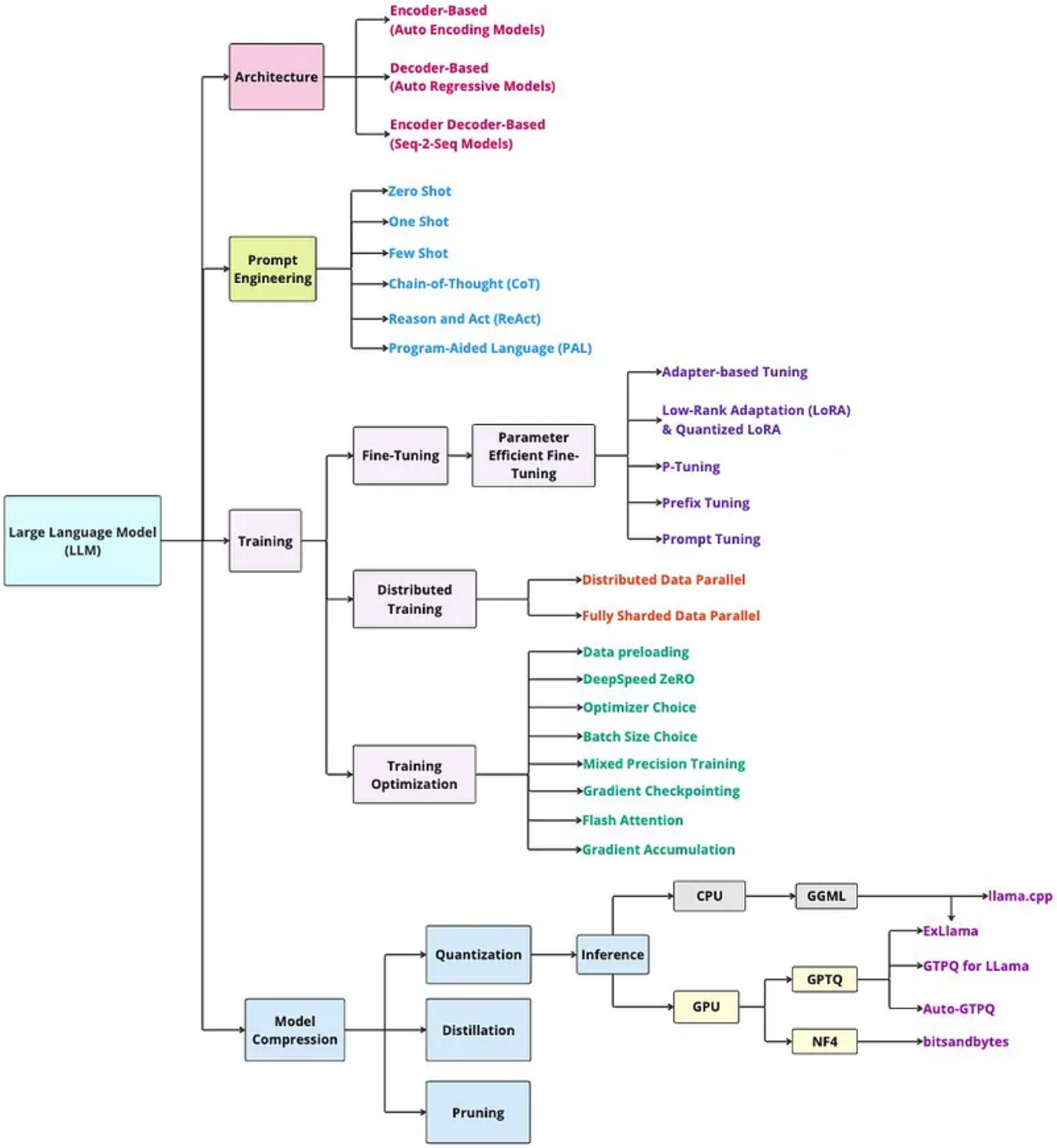
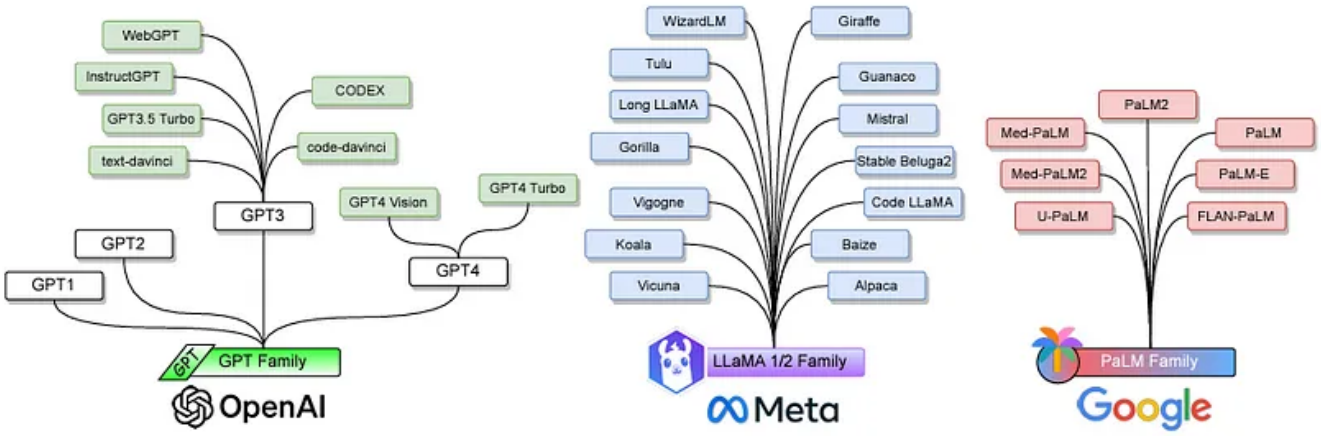


Image by Author

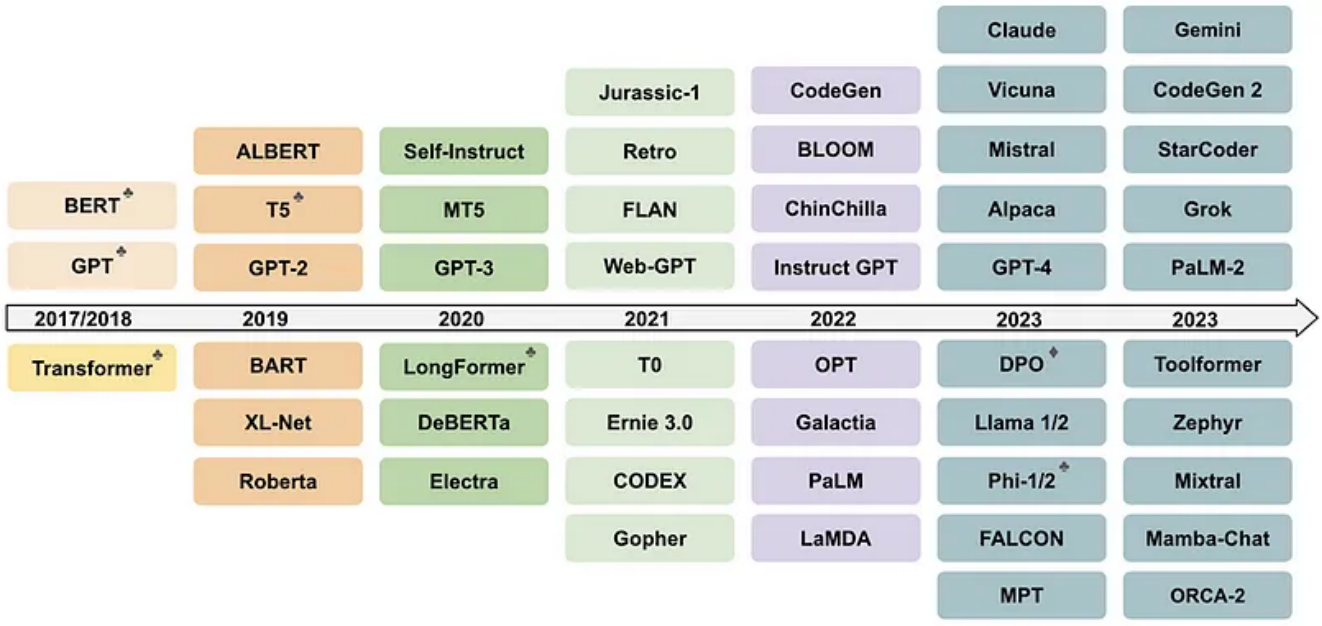
1. What Are Large Language Models?

Large Language Models (LLM) are very large deep learning models that are pre-trained on vast amount of data. The underlying transformer is a set of neural networks that consist of an encoder and a decoder with self-attention

capabilities. The encoder and decoder extract meanings from a sequence of text and understand the relationships between words and phrases in it.



Popular LLM Families.

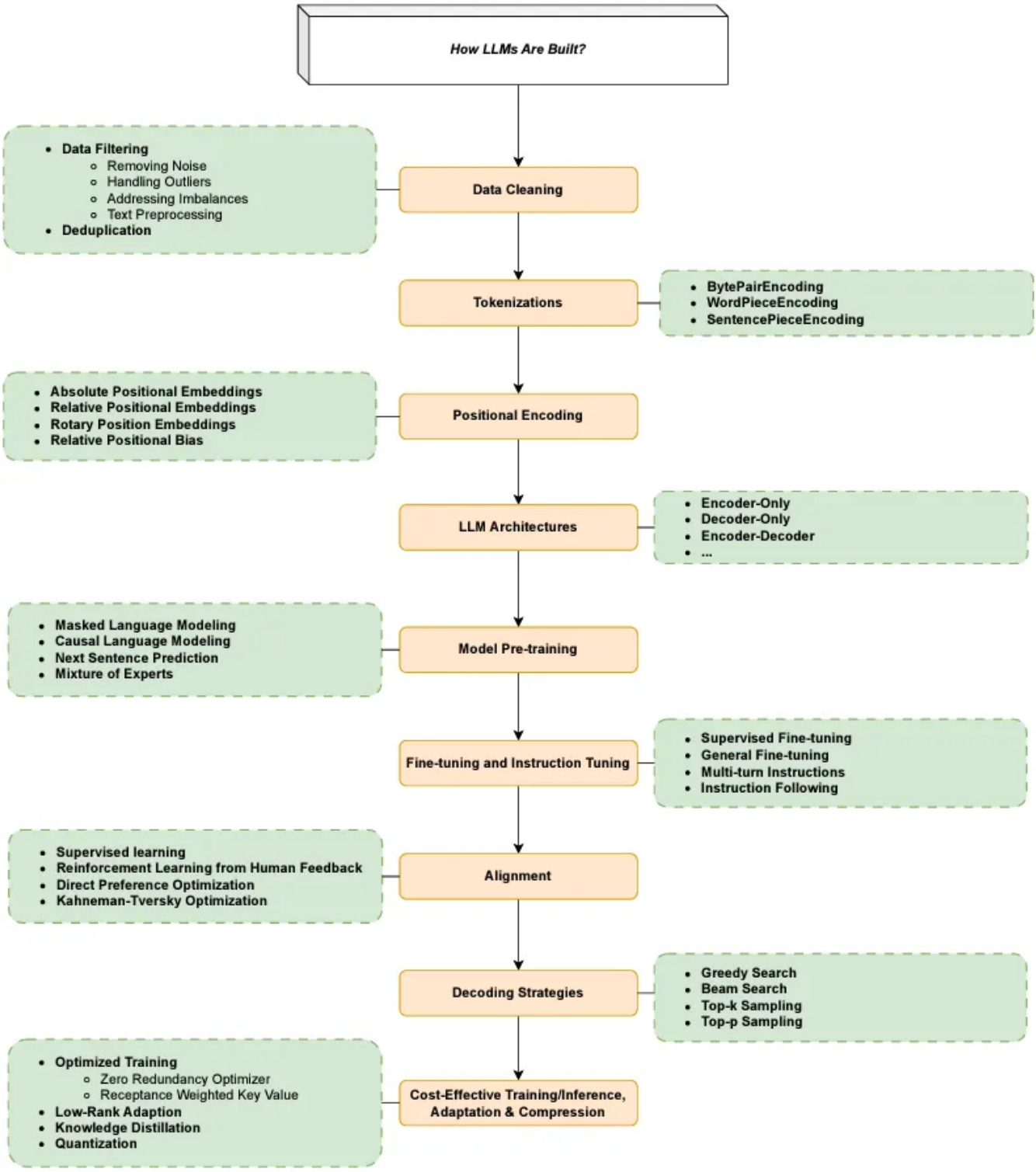


Timeline of some of the most representative LLM frameworks (so far).

Transformer neural network architecture allows the use of very large models, often with hundreds of billions of parameters. Such large-scale models can ingest massive amount of data, often from the internet, but also

from sources such as the Common Crawl, which comprises more than 50 billion web pages, and Wikipedia, which has approximately 57 million pages.

One of the key instruments of NLP applications is **language modeling.**

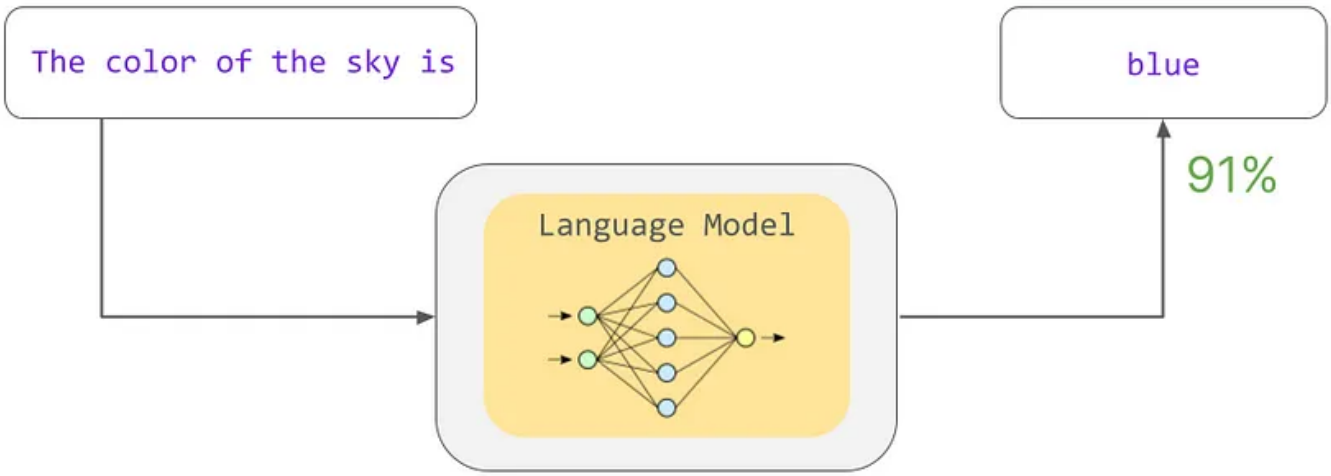


This figure shows different components of LLMs.

2. Language Modeling (LM)

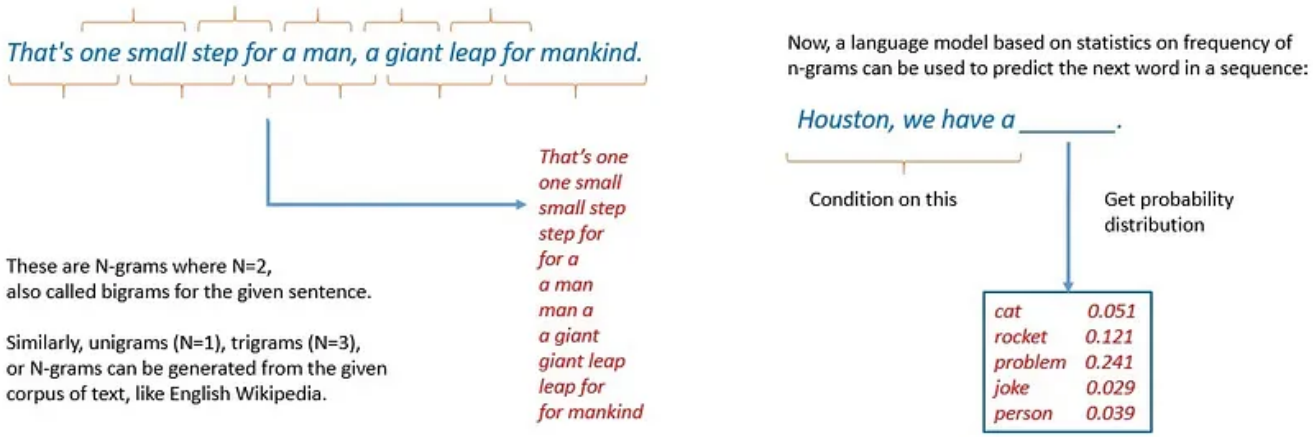
Can the processes of language and communication be reduced to computation?

Language models **generate probabilities by learning from one or more text corpus.** A text corpus is a language resource consisting of a large and structured set of texts in one or more languages. Text corpus can contain text in one or multiple languages and is often annotated.



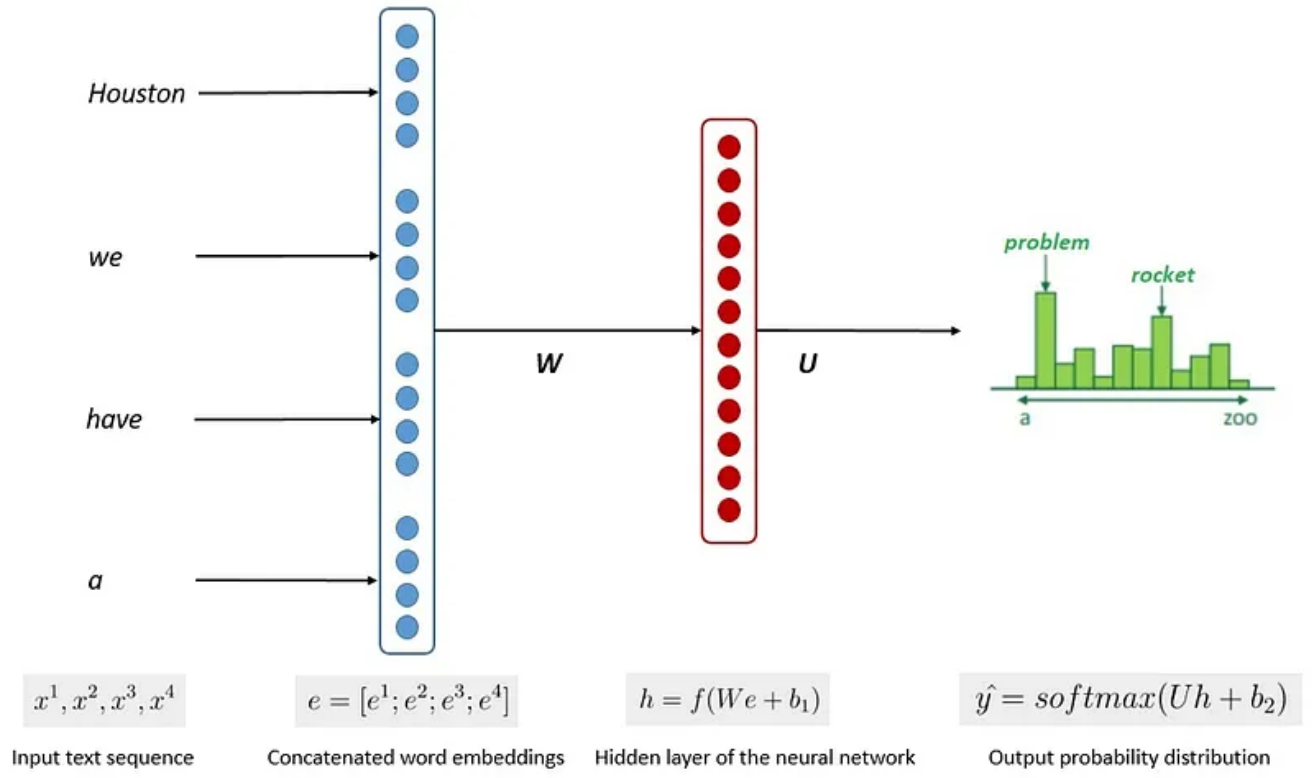
A language model can predict the most probable word (or words) to follow this phrase, based on the statistical patterns it has learned during training. In the figure, a Language Model may estimate a 91% probability that the word blue follows the sequence of words The color of the sky is.

One of the **earliest approaches for building a language model is based on the n-gram.** An n-gram is a contiguous sequence of n items from a given text sample. Here, the model assumes that the probability of the next word in a sequence depends only on a fixed-size window of previous words:



N-gram

However, n-gram language models have been **largely superseded by neural language models.** It's based on neural networks, a computing system inspired by biological neural networks. These models make use of continuous representations or embeddings of words to make their predictions:



Neural Networks

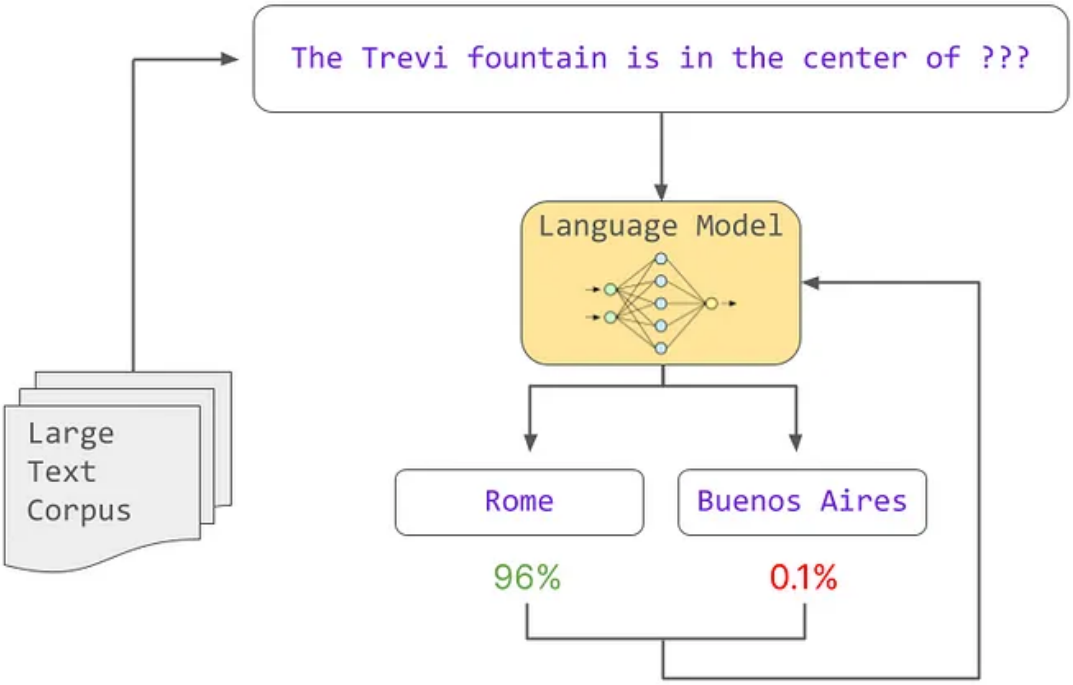
Neural networks represent words distributed as a non-linear combination of weights. Hence, it can avoid the curse of dimensionality in language modeling. There have been several neural network architectures proposed for language modeling.

3. Foundation Models and LLMs

This is quite a departure from the earlier approach in NLP applications, where specialized language models were trained to perform specific tasks. On the contrary, **researchers have observed many emergent abilities in the LLMs**, abilities that they were never trained for.

For instance, LLMs have been shown to perform multi-step arithmetic, unscramble a word's letters, and identify offensive content in spoken languages. Recently, ChatGPT, a popular chatbot built on top of OpenAPI's GPT family of LLMs, has cleared professional exams like the US Medical Licensing Exam!

A foundation model generally refers to **any model trained on broad data that can be adapted to a wide range of downstream tasks**. These models are typically created using deep neural networks and trained using self-supervised learning on many unlabeled data.



During the training process, text sequences are extracted from the corpus and truncated. The language model calculates probabilities of the missing words, which are then slightly adjusted and fed back to the model to match the ground truth, via a gradient descent-based optimization mechanism. This process is repeated over the whole text corpus.

Nevertheless, LLMs are typically trained on language-related data like text. But a **foundation model is usually trained on multimodal data**, a mix of text, images, audio, etc. More importantly, a foundation model is intended to serve as the basis or foundation for more specific tasks:

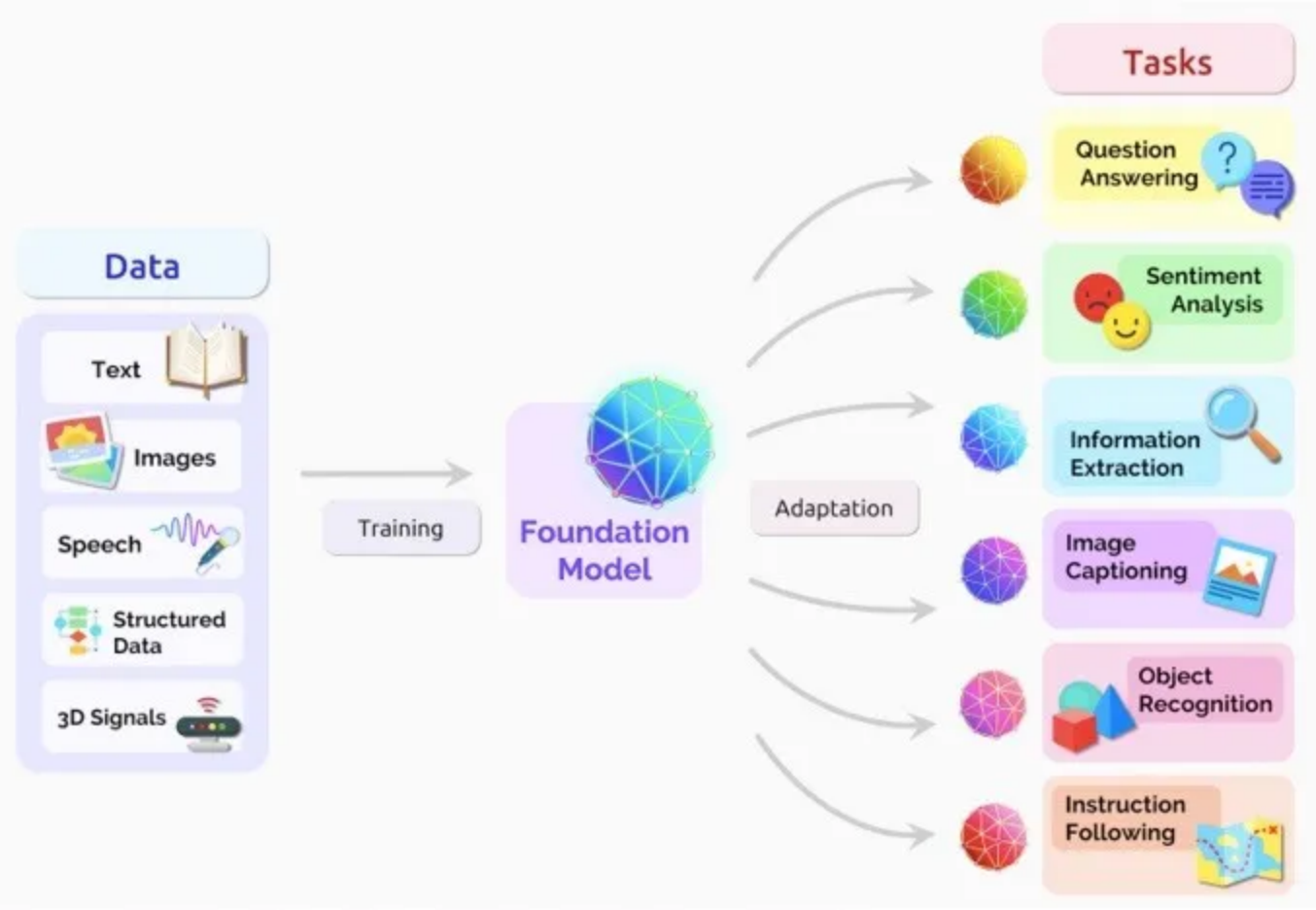


Image Credit: https://research.aimultiple.com/foundation-models/

Foundation models are **typically fine-tuned with further training for various downstream cognitive tasks.** Fine-tuning refers to the process of taking a pre-trained language model and training it for a different but related task using specific data. The process is also known as transfer learning.

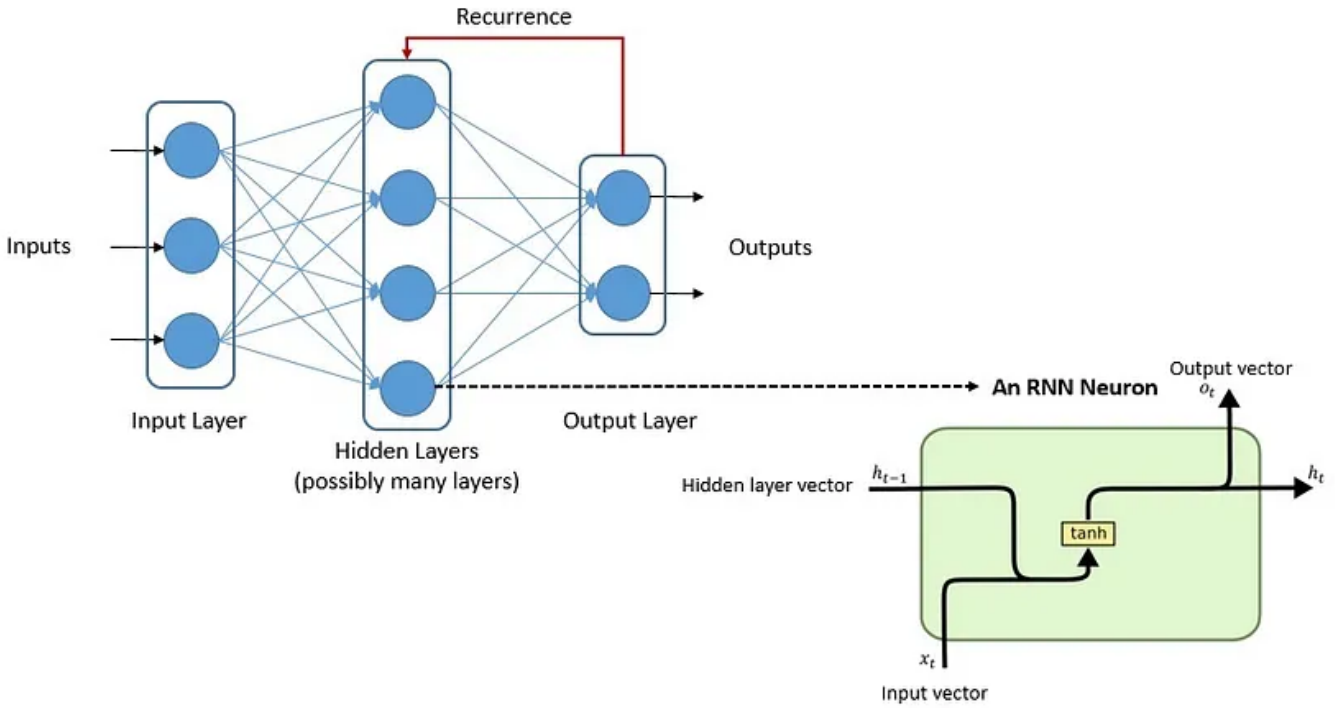
4. Architecture of LLMs

Most of the early LLMs were created using RNN models with LSTMs and GRUs. However, they faced challenges, mainly in performing NLP tasks at massive scales. But, this is precisely where LLMs were expected to perform. This led to the creation of Transformers!

Earlier Architecture of LLMs

When it started, LLMs were largely created using self-supervised learning algorithms. Self-supervised learning refers to the **processing of unlabeled data to obtain useful representations** that can help with downstream learning tasks.

Quite often, self-supervised learning algorithms use a model based on an artificial neural network (ANN). We can create ANN using several architectures, but the most widely used architecture for LLMs were the Recurrent Neural Network (RNN).



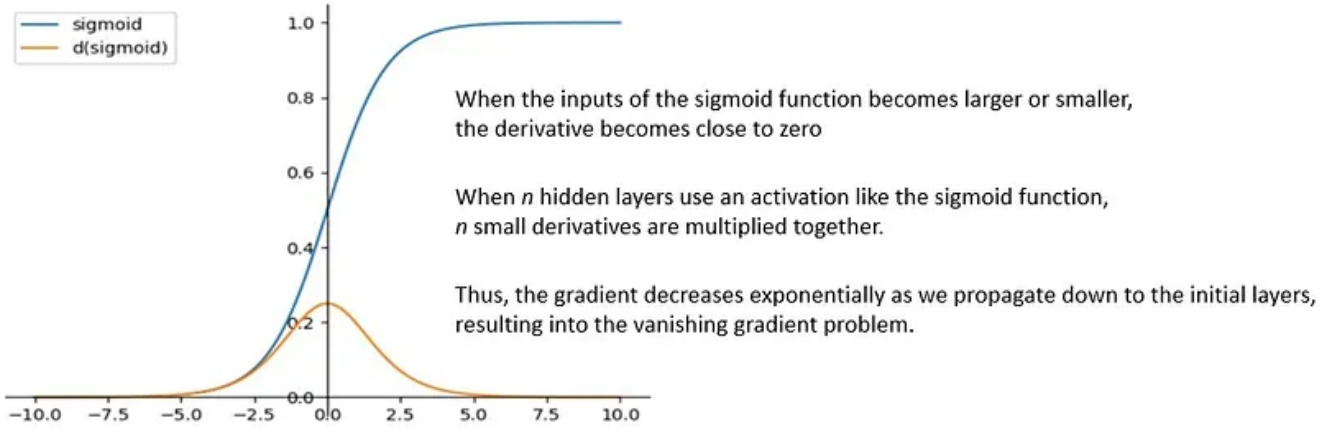
RNN

Now, RNNs can use their internal state to process variable-length sequences of inputs. An RNN **has both long-term memory and short-term memory.** There are variants of RNN like Long-short Term Memory (LSTM) and Gated Recurrent Units (GRU).

Problems with LSTMs & GRUs

A RNN that uses LSTM units is very slow to train. Moreover, we need to feed the data sequentially or serially for such architectures. This does not allow us to parallelize and use available processor cores.

Alternatively, an **RNN model with GRU trains faster but performs poorly on larger datasets.** Nevertheless, for a long time, LSTMs and GRUs remained the preferred choice for building complex NLP systems. However, such models also suffer from the vanishing gradient problem:



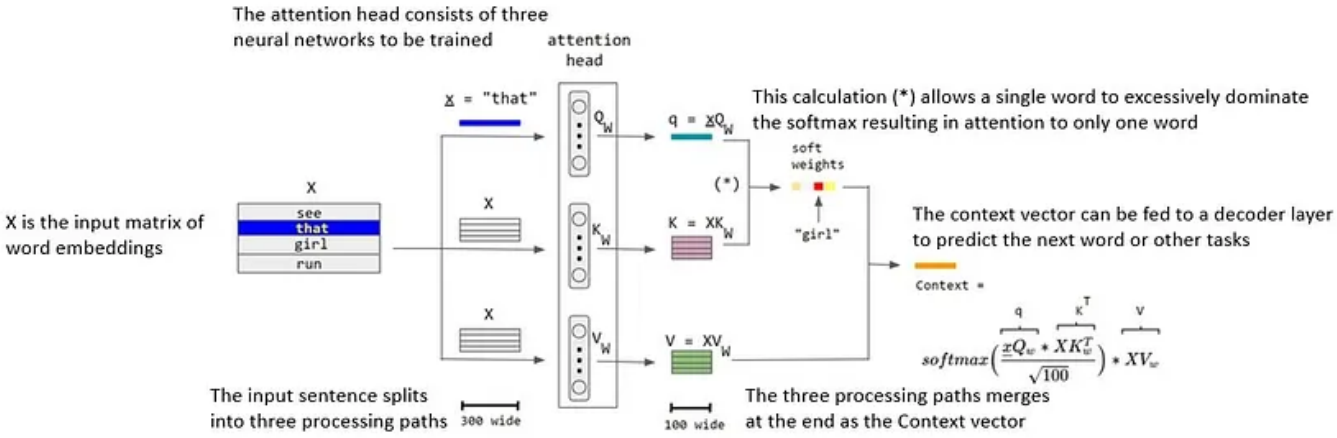
Vanishing Gradient

Attention Mechanism

Some of the problems with RNNs were partly addressed by adding the attention mechanism to their architecture. In recurrent architectures like LSTM, the amount of information that can be propagated is limited, and the window of retained information is shorter.

However, with the attention mechanism, this information window can be significantly increased. Attention is **a technique to enhance some parts of the input data while diminishing other parts.** The motivation behind this is

that the network should devote more focus to the important parts of the data:



A self-attention scheme used in unsupervised training of a language model

Image Credit: https://en.wikipedia.org/wiki/Attention_(machine_learning)

There is a **subtle difference between attention and self-attention**, but their motivation remains the same. While the attention mechanism refers to the ability to attend to different parts of another sequence, self-attention refers to the ability to attend to different parts of the current sequence.

Self-attention allows the model to access information from any input sequence element. In NLP applications, this **provides relevant information about far-away tokens**. Hence, the model can capture dependencies across the entire sequence without requiring fixed or sliding windows.

Arrival of Transformers

The RNN models with attention mechanisms saw significant improvement in their performance. However, **recurrent models are, by their nature, difficult to scale**. But, the self-attention mechanism soon proved to be quite

powerful, so much so that it did not even require recurrent sequential processing.

The introduction of transformers by the Google Brain team in 2017 is perhaps one of the most important inflection points in the history of LLMs. A transformer is **a deep learning model that adopts the self-attention mechanism** and processes the entire input all at once:

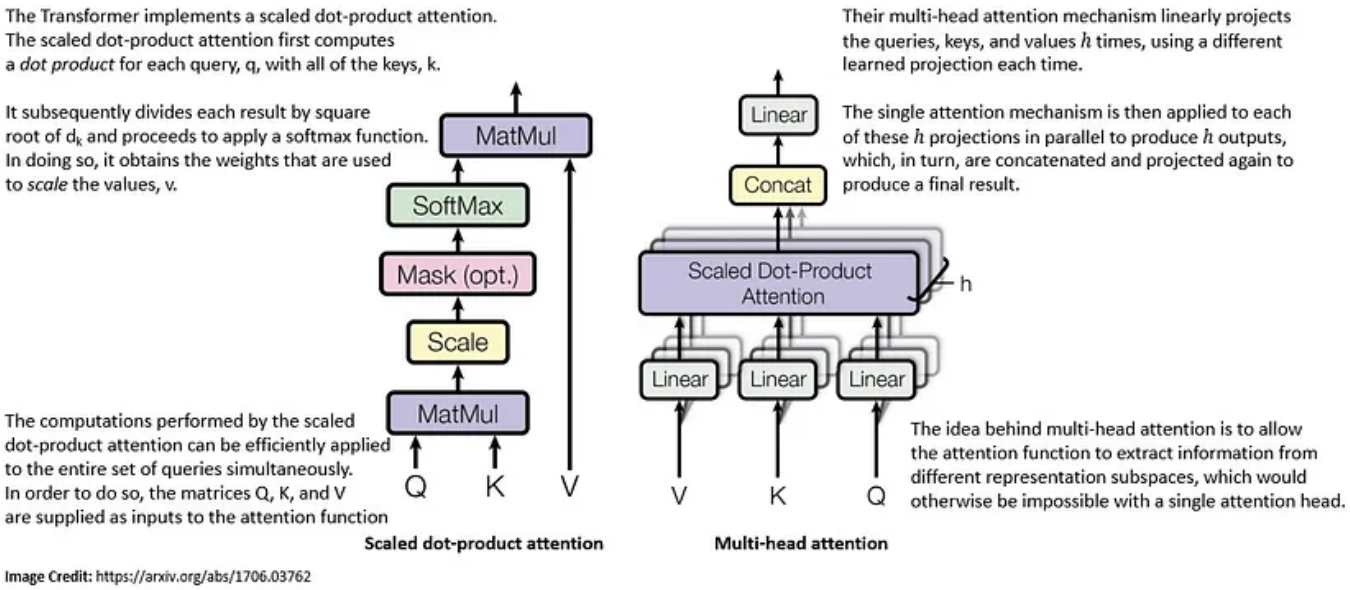


Image Credit: https://arxiv.org/abs/1706.03762

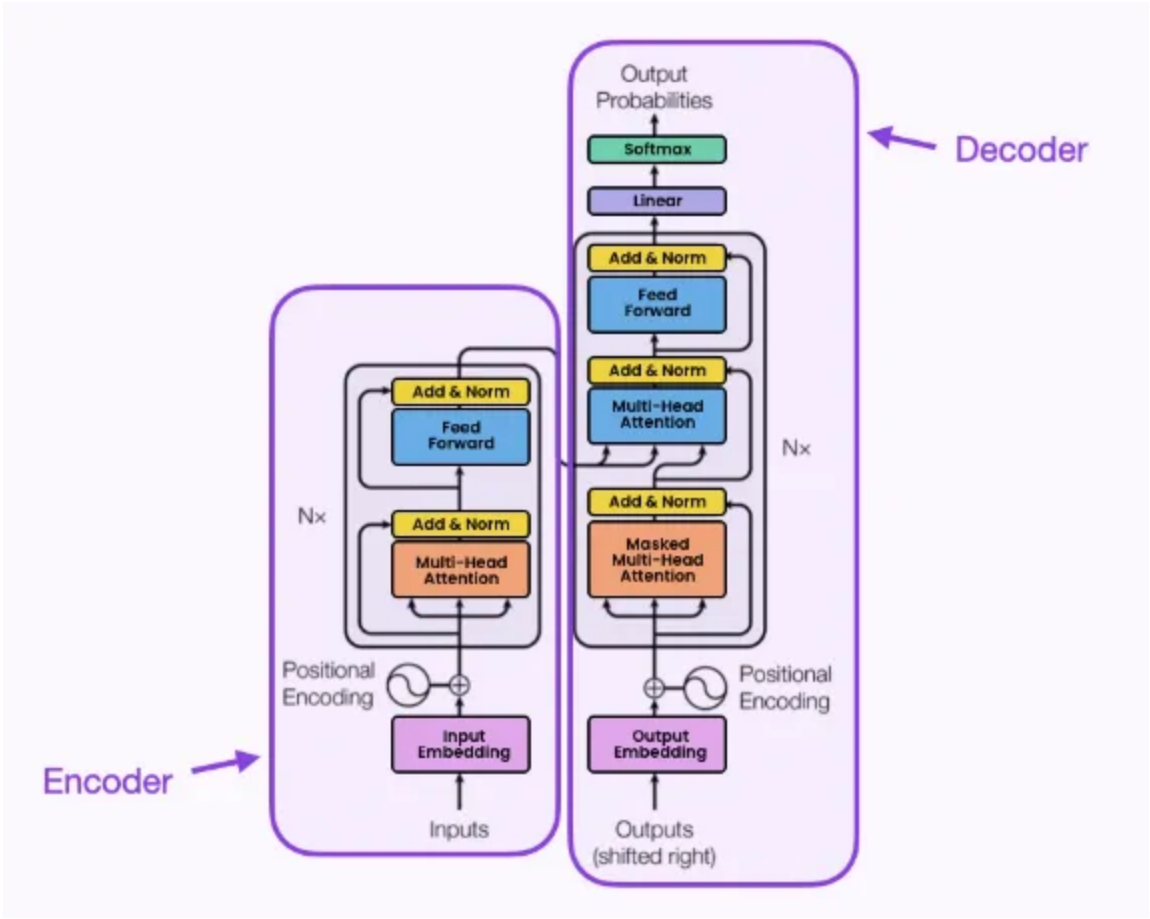
As a significant change to the earlier RNN-based models, **transformers do not have a recurrent structure.** With sufficient training data, the attention mechanism in the transformer architecture alone can match the performance of an RNN model with attention.

Another significant advantage of using the transformer model is that they are **more parallelized and require significantly less training time.** This is exactly the sweet spot we require to build LLMs on a large corpus of text-based data with available resources.

Encoder-Decoder Architecture

Many ANN-based models for natural language processing are built using encoder-decoder architecture. For instance, seq2seq is a family of algorithms originally developed by Google. It turns one sequence into another sequence by using RNN with LSTM or GRU.

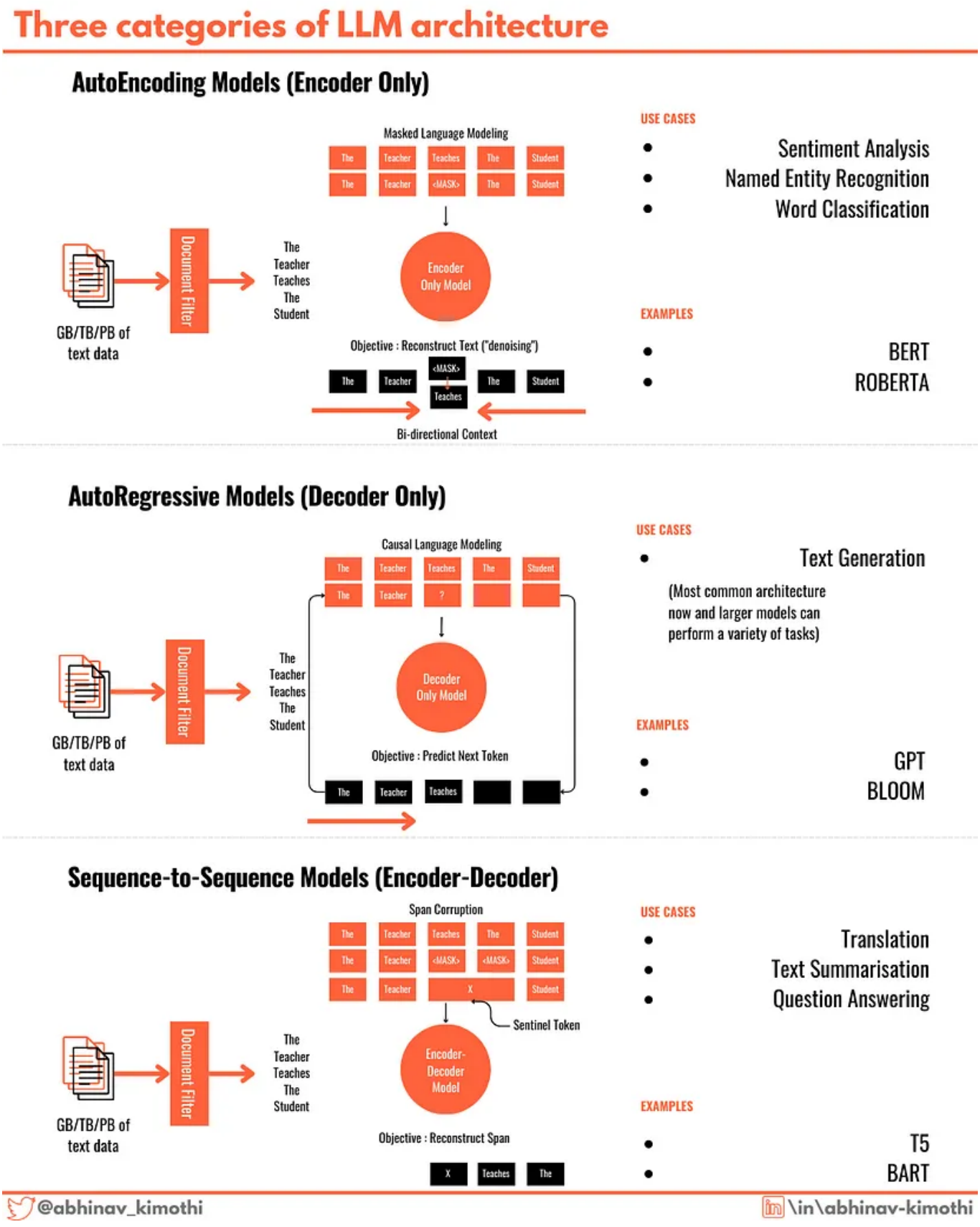
The original **transformer model also used the encoder-decoder architecture.** The encoder consists of encoding layers that process the input iteratively, one layer after another. The decoder consists of decoding layers that do the same thing to the encoder's output:



The Transformer — High-Level Architecture

The function of each **encoder layer is to generate encodings that contain information about which parts of the input are relevant to each other.** The output encodings are then passed to the next encoder as its input. Each encoder consists of a self-attention mechanism and a feed-forward neural network.

Further, each **decoder layer takes all the encodings and uses their incorporated contextual information** to generate an output sequence. Like encoders, each decoder consists of a self-attention mechanism, an attention mechanism over the encodings, and a feed-forward neural network.



5. Pre-Training

During this phase, the model is pre-trained on a large amount of unstructured textual datasets in a self-supervised manner. The main challenge in pretraining is computational cost.

GPU RAM required to store 1B parameter model
=> 1 parameter –> 4 bytes (32-bit float)
=> 1B parameter –> $4*10^9$ bytes = 4GB

GPU RAM Required for 1B paramter model = 4GB@32 bit full precission

Let's calculate the memory required to train the 1B parameter model:

```
Model Parameter --> 4 bytes per parameter
Gradients --> 4 bytes per parameter
ADAM Optimizer (2 states) --> 8 bytes per parameter
Activations and temp memory (variable size) --> 8 bytes per parameter (high-end
==> 4 bytes parameter + 20 extra bytes per paramter
```

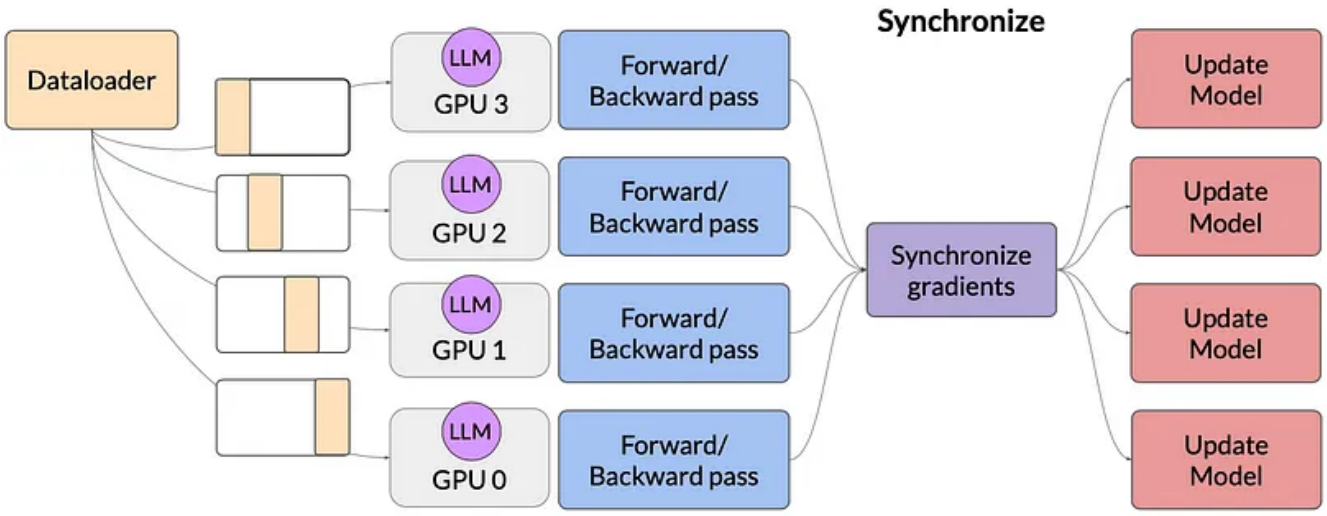
So, the memory needed to train is ~20X the memory needed to store the model.
Memory needed to store 1B parameter model = 4GB@32 bit full precision
Memory needed to train 1B parameter model = 80GB@32 bit full precision

6. Data Parallel Training Techniques

6.1. Distributed Data Parallel (DDP)

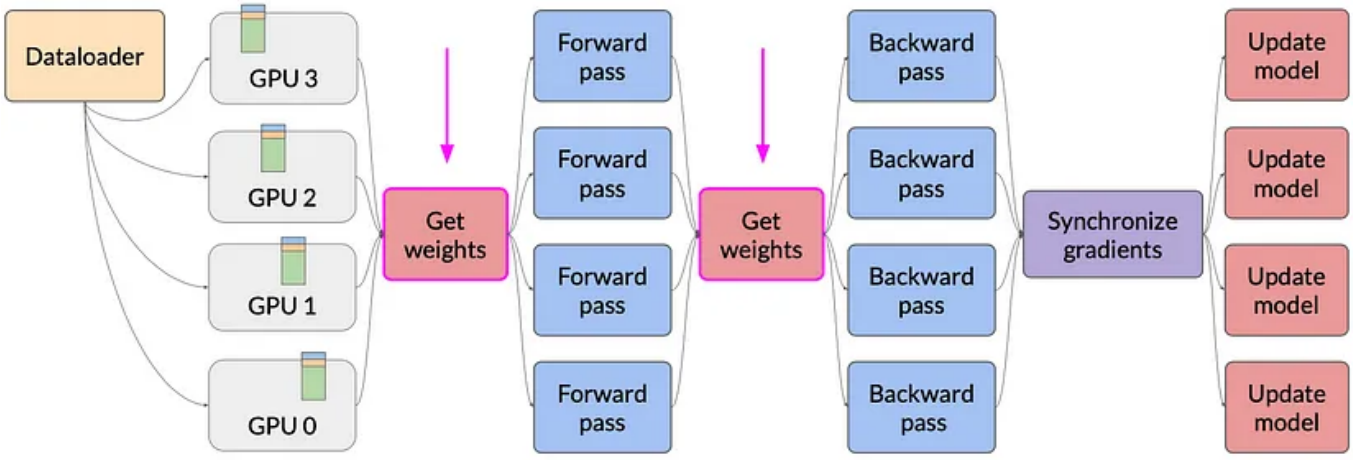
Distributed Data Parallel (DDP) requires model weights and all other additional parameters, gradients, and optimizer states that are needed for

training fit in a single GPU. If the model is too big, model sharding should be used instead.



6.2. Fully Sharded Data Parallel (FSDP)

Fully Sharded Data Parallel (FSDP) reduces memory by distributing (sharding) the model parameters, gradients, and optimizer states across GPUs.



7. Fine-Tuning

Fine-tuning helps us get more out of pre-trained large language models (LLMs) by adjusting the model weights to better fit a specific task or domain. This means you can get higher quality results than plain prompt engineering at a fraction of the cost and latency.

The evolutionary development of fine tuning methods in recent years. Models on the same branch have some common features. The vertical position of the models shows the timeline of their release dates.

Why fine-tune LLM?

Compared to prompting, fine-tuning is often far more effective and efficient for steering an LLM's behavior. By training the model on a set of examples,

you're able to shorten your well-crafted prompt and save precious input tokens without sacrificing quality. You can also often use a much smaller model. That, in turn, translates to reduced latency and inference costs.

For example, a fine-tuned Llama 7B model can be astronomically more **cost-effective** (around 50 times) on a per-token basis compared to an off-the-shelf model like GPT-3.5, with comparable performance.

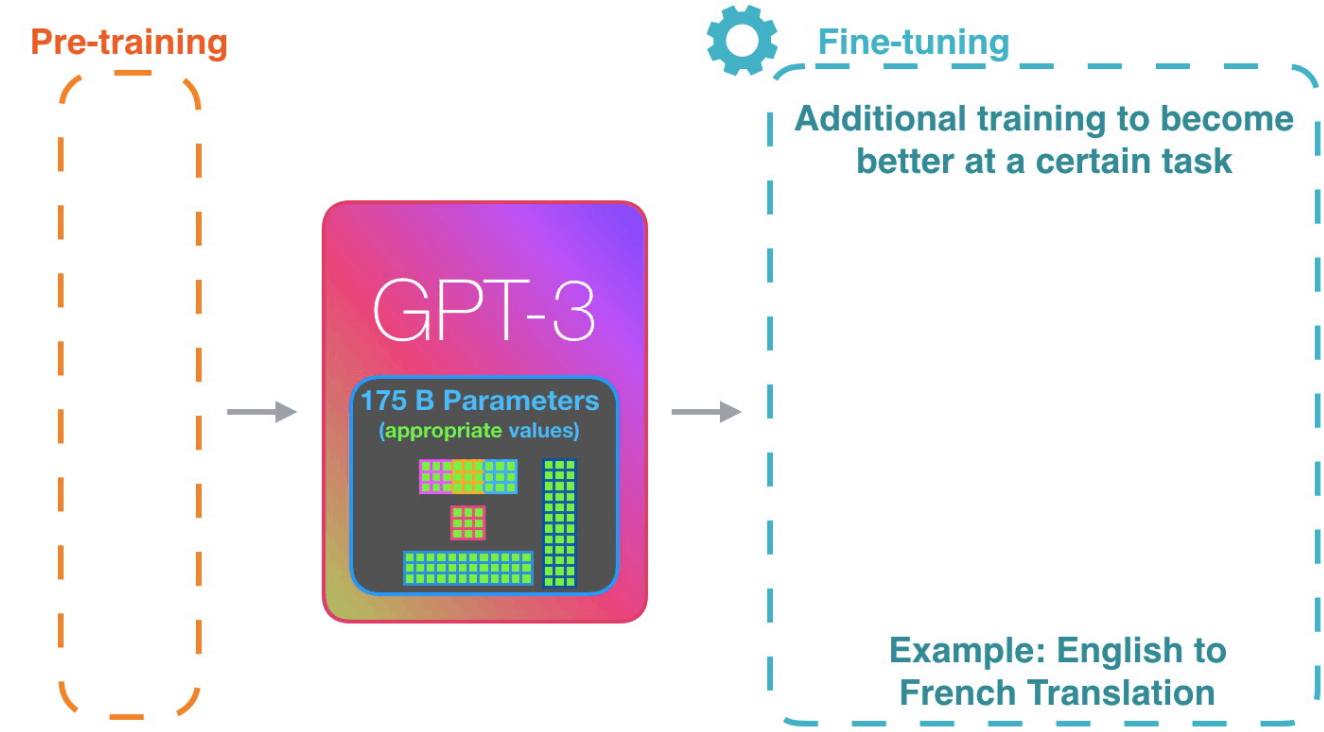
How Fine-Tuning Works ?

As mentioned, fine-tuning is tweaking an already-trained model for some other task. The way this works is by taking the weights of the original model and adjusting them to fit a new task.

Models when trained learn to do some specific task, for example, GPT-3 has been trained on a massive dataset and as a result, it has learned to generate stories, poems, songs, letters, and a lot of other things. One can take this ability of GPT-3 and fine-tune it on a specific task like generating answers to customer queries in a specific manner.

There are different ways and techniques to fine-tune a model, the most popular being *transfer learning*. Transfer learning comes out of the computer vision world, it is the process of freezing the weights of the initial layers of a network and only updating the weights of the later layers. This is because the lower layers, the layers closer to the input, are responsible for learning the general features of the training dataset. And the upper layers, closer to the output, learn more specific information which is directly tied to generating the correct output.

Here is a quick visualization of how fine-tuning works:

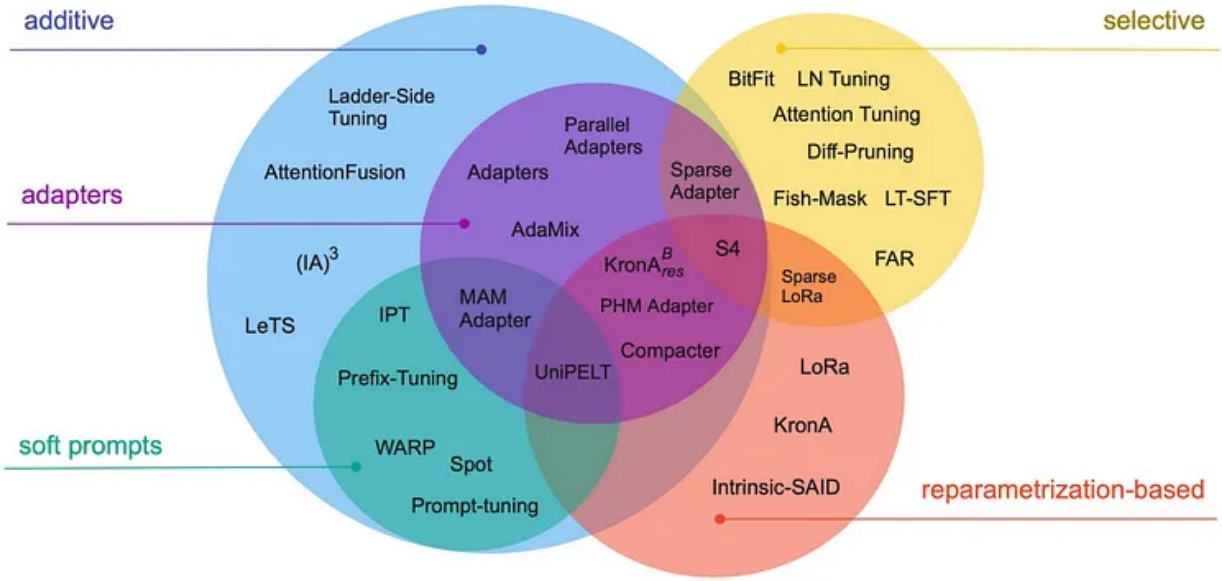


Source

7.1. PEFT

PEFT, Parameter Efficient Fine-Tuning, is a set of techniques or methods to fine-tune a large model in the most compute and time-efficient way possible, without losing any performance which we might see from full fine-tuning. This is done because with models growing bigger and bigger like BLOOM which has a whooping **176 billion** parameters, it is almost impossible to fine tune them without spending tens of thousands of dollars. But it is sometimes almost necessary to use such big models for better performance. This is where PEFT comes in. It helps you solve the problems faced during such big models.

Here are some PEFT techniques:



Source

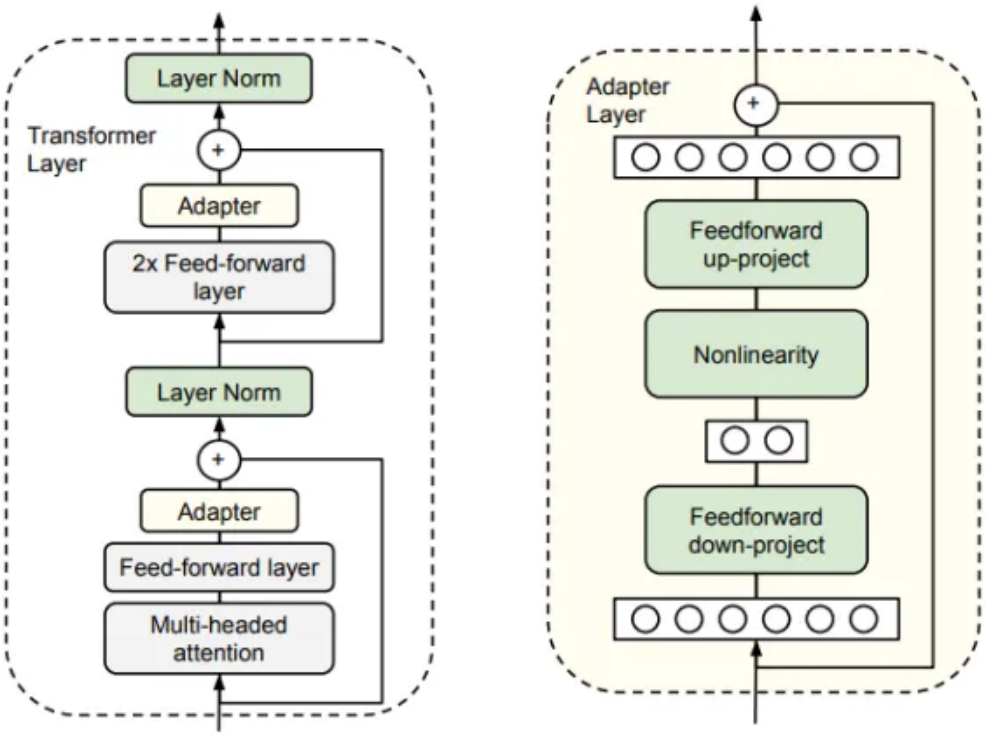
7.2. Transfer Learning

Transfer learning is when we take some of the learned parameters of a model and use them for some other task. This sounds similar to fine-tuning but is different. In finetuning, we re-adjust all the parameters of the model or *freeze* some of the weights and adjust the rest of the parameters. But in transfer learning, we use some of the learned parameters from a model and use them in other networks. This gives us more flexibility in terms of what we can do. For example, we cannot change the architecture of the model when fine-tuning, this limits us in many ways. But when using transfer learning, we use only a part of the trained model, which we can then attach to any other model with any architecture.

Transfer learning is often seen in NLP tasks with LLMs where people use the encoder part of the transformer network from a pre-trained model like T5 and train the later layers.

7.3. Adapters

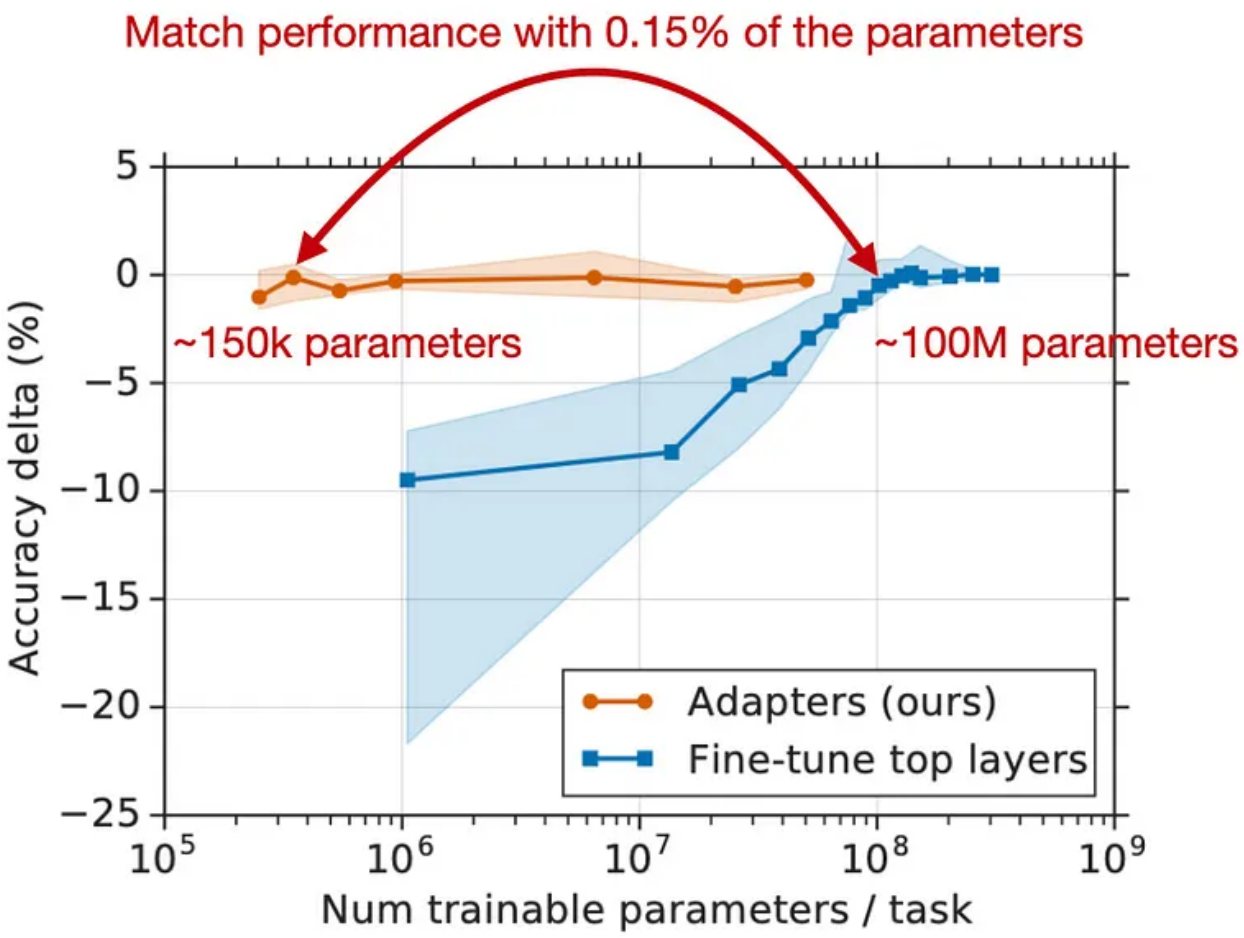
Adapters were one of the first parameter-efficient fine-tuning techniques released. In the paper, showed that we can add more layers to the pre-existing transformer architecture and only finetune them instead of the whole model. They showed that this technique resulted in similar performance when compared to complete fine-tuning.



Source

On the left, there is the modified transformer architecture with added adapter layers. We can see adapter layers are added after the attention stack and the feed-forward stack. And on the right, we can see the architecture of the adapter layer itself. The adapter layer comprises a bottleneck architecture, it takes the input and narrows it down to a smaller dimension representation and then passes it through a non-linear activation function, and then scales it back up to the dimension of the input. This makes sure that the next layer in the transformer stack will be able to receive the generated output from the adapter layer.

In the paper, the authors show that this method of fine-tuning is comparable to complete fine-tuning while consuming much less compute resources and training time. They were able to attain 0.4% of full fine-tuning on the GLUE benchmark while adding 3.6% of the parameters.



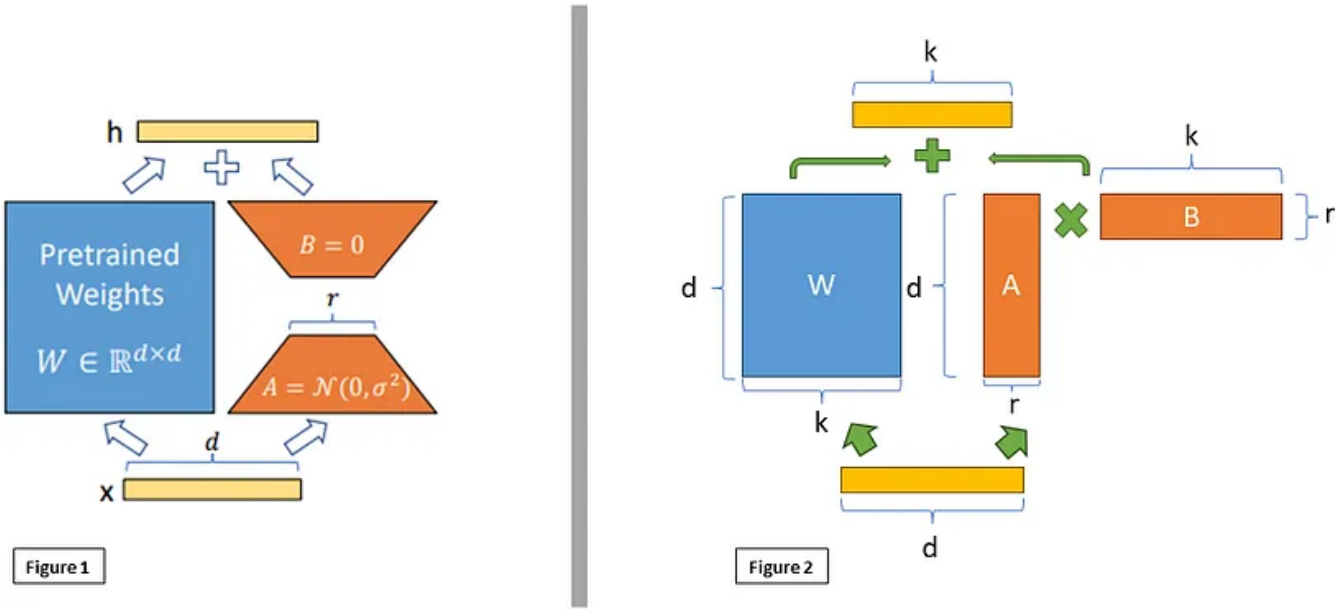
Source

7.4. LoRA — Low-Rank Adaptation

LoRA is a similar strategy to Adapter layers but it aims to further reduce the number of trainable parameters. It takes a more mathematically rigorous approach. LoRA works by modifying how the updatable parameters are trained and updated in the neural network.

Let's explain mathematically. We know that the weights matrices of a pre-trained neural network are full rank, meaning each weight is unique and can't be made by combining other weights. But in this paper authors showed that when pretrained language models are adjusted to a new task the weights have a lower "intrinsic dimension". Meaning, that the weights can be represented in a smaller matrix, or that it has a lower rank. This in turn means that during backpropagation, the weight update matrix has a lower rank, as most of the necessary information has already been captured by the pre-training process and only task-specific adjustments are made during fine-tuning.

A much simpler explanation is that during finetuning only a very few weights are updated a lot as most of the learning is done during the pretraining phase of the neural network. LoRA uses this information to reduce the number of trainable parameters.



LoRA

Using GPT-3 175B as an example, LoRA research team demonstrated that a very low rank (i.e., r in Figure 1 can be one or two) suffices even when the

full rank (i.e., d) is as high as 12,288, making LoRA both storage and compute efficient.

Figure 2 demonstrates that matrix A[d X r] and B[r X k] will be [d X k] while we can vary the r. A very small r will lead to fewer parameters to turn. While it will shorten the training time, it also could result in information loss and decrease the model performance as r becomes smaller. However, with LoRA even at low rank order, performance was as good as or better than fully trained models.

LoRA Fine-tuning with HuggingFace

To implement LoRA finetuning with HuggingFace, you need to use the PEFT library to inject the LoRA adapters into the model and use them as the update matrices.

```
from transformers import AutoModelForCausalLM
from peft import get_peft_config, get_peft_model, LoraConfig, TaskType

model = AutoModelForCausalLM.from_pretrained(model_name_or_path, device_map="aut
peft_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM, inference_mode=False, r=32, lora_alpha=16, lor
    target_modules=['query_key_value'] # optional, you can target specific layer
) # create LoRA config for the finetuning

model = get_peft_model(model, peft_config) # create a model ready for LoRA finet
model.print_trainable_parameters()
# trainable params: 9,437,184 || all params: 6,931,162,432 || trainable%: 0.1361
```

Once this is done, you can train the model as you would normally do. But this time it will take much less time and compute resources as it normally

does.

Efficiency of LoRA

Authors in the paper show that LoRA can **outperform** full finetuning **with only 2% of total trainable parameters.**

Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 ($\text{Adapter}^{\text{H}}$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($\text{Adapter}^{\text{H}}$)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

Table 4: Performance of different adaptation methods on GPT-3 175B. We report the logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched, and Rouge-1/2/L on SAMSum. LoRA performs better than prior approaches, including full fine-tuning. The results on WikiSQL have a fluctuation around $\pm 0.5\%$, MNLI-m around $\pm 0.1\%$, and SAMSum around $\pm 0.2/\pm 0.2/\pm 0.1$ for the three metrics.

Source

As for the number of parameters it trains, we can largely control that using the rank *r* parameter. For example, let's say the weight updation matrix has 100,000 parameters, *A* being 200 and *B* being 500. The weight updation matrix can be decomposed into smaller matrixes of lower dimensions, *A* being *200 x 3* and *B* being *3 x 500*. This gives us *200 x 3 + 3 x 500 =* ***2100*** trainable parameters only, which is only **2.1%** of the total number of parameters. This can be further reduced as we can decide to only apply LoRA to specific layers only.

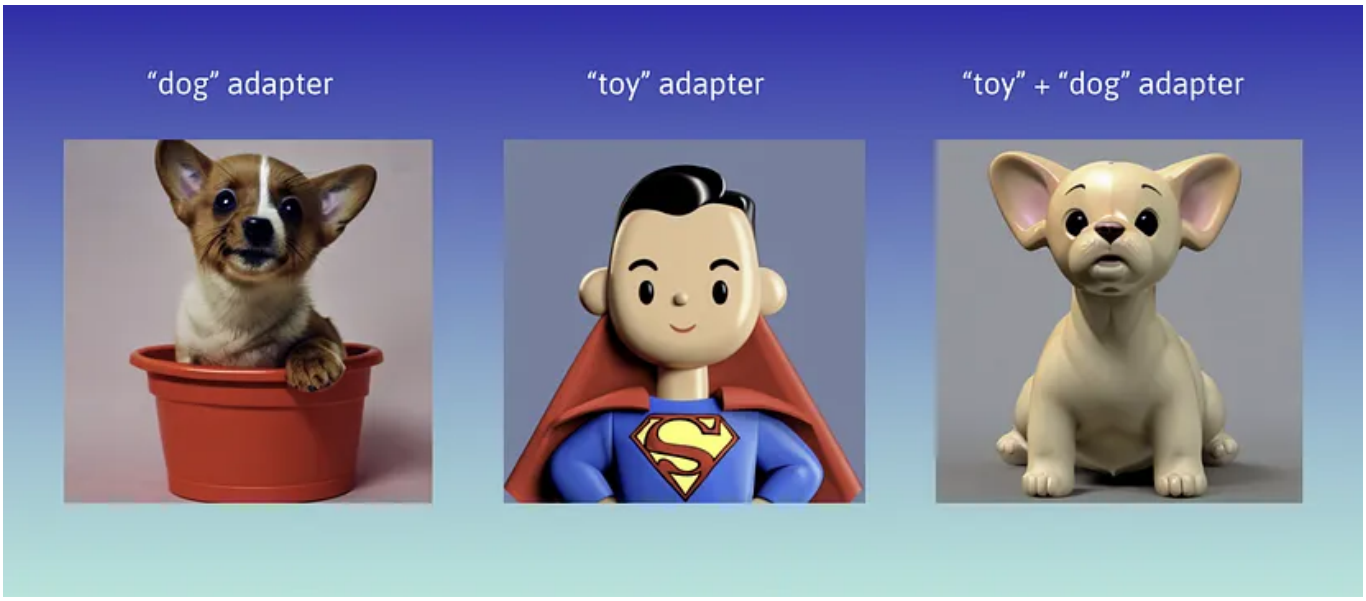
As the number of parameters trained and applied are MUCH smaller than the actual model, the files can be as small as **8MB**. This makes loading, applying, and transferring the learned models much easier and faster.

You can read the LoRA paper if you want to learn more and do a deeper dive into the topic.

LoRA in Stable Diffusion

One of the most interesting use cases of LoRA can be shown in image generation applications. Images have an inherent *style* that can be visually seen. Instead of training massive models to get specific styles of images out of models, users can now only train LoRA weights and use them with techniques like Dreambooth to achieve really good quality images with a lot of customizability.

LoRA weights can also be combined with other LoRA weights and be used in a weighted combination to generate images that carry multiple styles. You can find a ton of LoRA adapters online and load them into your models on CivitAI.



Source

7.5. QLoRA

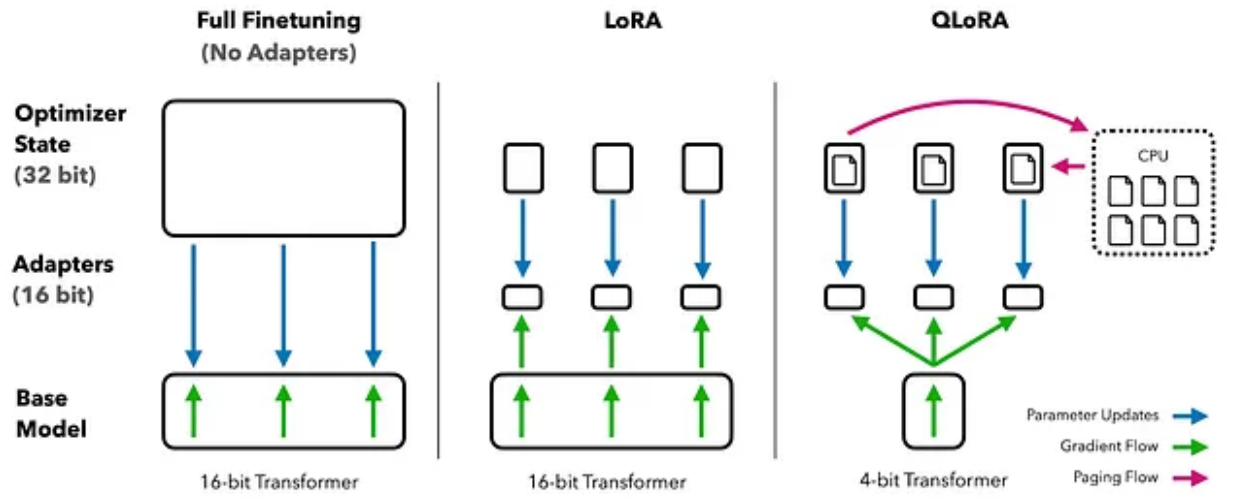


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

QLoRA

How is it different from LoRA

- It is a 4-bit transformer.

- QLoRA is a finetuning technique that combines a high-precision computing technique with a low-precision storage method. This helps keep the model size small while still making sure the model is still highly performant and accurate.
- QLoRA uses LoRA as an accessory to fix the errors introduced during the quantization errors.

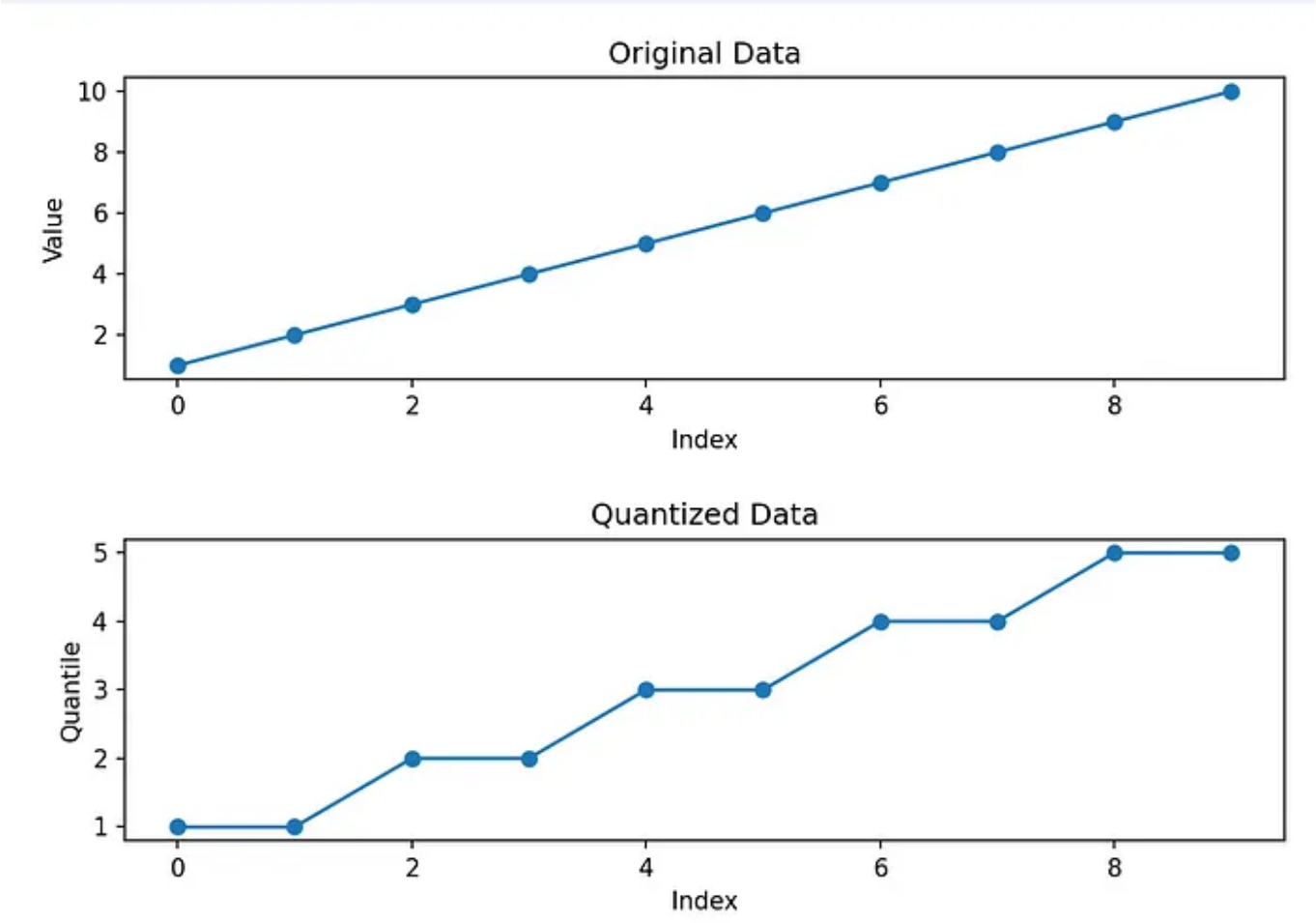
Working of QLoRA

- QLoRA works by introducing 3 new concepts that help to reduce memory while retaining the same quality performance. These are *4-bit Normal Float, Double Quantization,* and *Paged Optimizers.*

4-bit Normal Float (NF4)

- 4-bit NormalFloat is a new data type and a key ingredient to maintaining 16-bit performance levels. Its main property is this: Any bit combination in the data type, e.g. 0011 or 0101, gets assigned an equal number of elements from an input tensor.
- 4-bit Quantization of weights and PEFT and train injected adapter weights (LORA) in 32-bit precision.
- QLoRA has one storage data type (NF4) and a computation data type (16-bit BrainFloat).
- We dequantize the storage data type to the computation data type to perform the forward and backward pass, but we only compute weight gradients for the LORA parameters which use 16-bit BrainFloat.

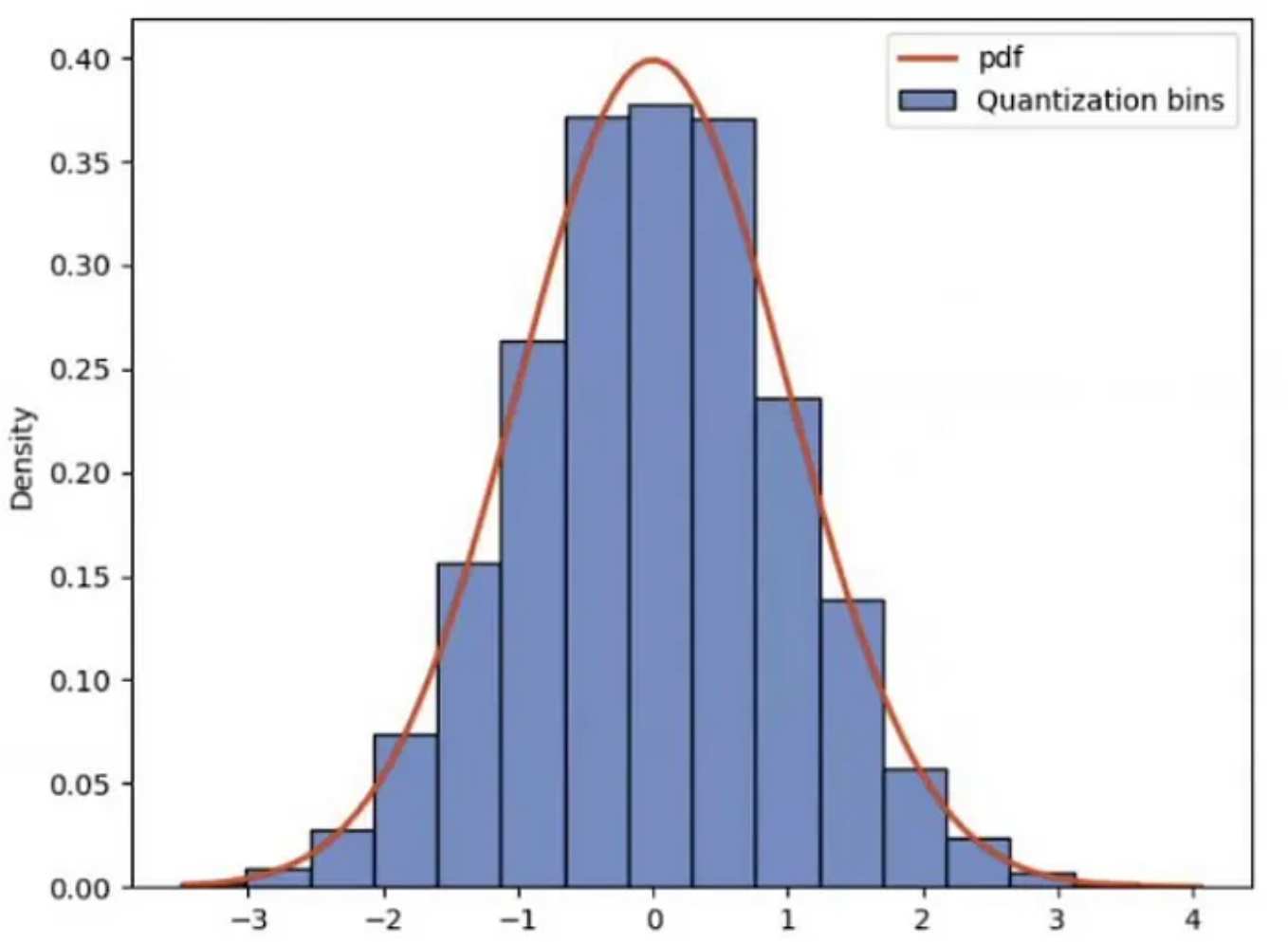
> *1. **Normalization:** The weights of the model are first normalized to have zero mean and unit variance. This ensures that the weights are distributed around zero and fall within a certain range.*
>
> *2. **Quantization:** The normalized weights are then quantized to 4 bits. This involves mapping the original high-precision weights to a smaller set of low-precision values. In the case of NF4, the quantization levels are chosen to be evenly spaced in the range of the normalized weights.*
>
> *3. **Dequantization:** During the forward pass and backpropagation, the quantized weights are dequantized back to full precision. This is done by mapping the 4-bit quantized values back to their original range. The dequantized weights are used in the computations, but they are stored in memory in their 4-bit quantized form.*



There are “buckets” or “bins” of data where the data is quantized. Both the numbers 2 and 3 fall into the same quantile, 2. This quantization process allows you to use fewer numbers by “rounding off” to the nearest quantile.

Dequantization

- Double quantization refers to the unique practice of quantizing the quantization constants utilized in the 4-bit NF quantization process. While it may seem inconspicuous, this approach has the potential to yield an average savings of 0.5 bits per parameter, as highlighted in the associated research paper. This optimization proves particularly beneficial within the context of QLoRA, which employs Block-wise k-bit Quantization. In contrast to quantizing all weights collectively, this method involves segregating weights into distinct blocks or chunks that undergo independent quantization.



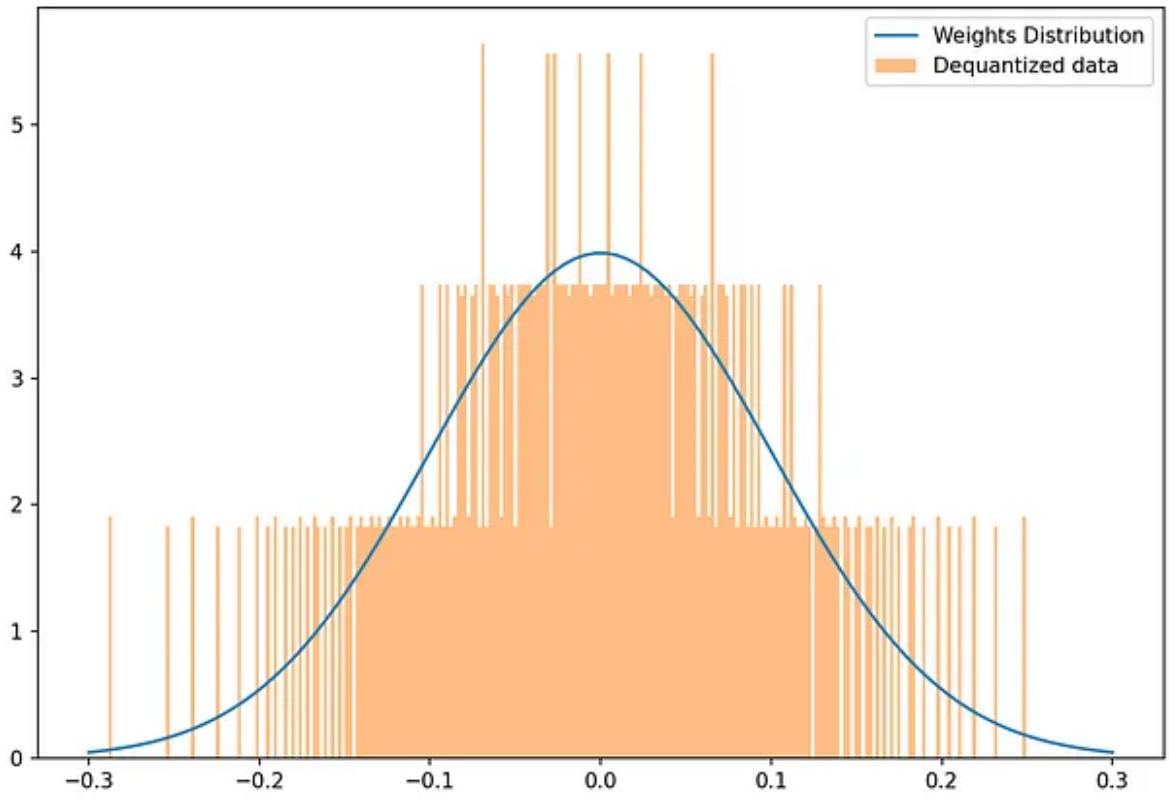
- The block-wise quantization method results in the generation of multiple quantization constants. Interestingly, these constants can undergo a secondary round of quantization, offering an opportunity for additional space savings. This strategy remains effective due to the limited number of quantization constants, mitigating the computational and storage demands associated with the process.

Paged Optimizers

- As demonstrated earlier, quantile quantization involves the creation of buckets or bins to encompass a wide range of numerical values. This process results in multiple distinct numbers being mapped to the same bucket, exemplified by the conversion of both 2 and 3 into the value 3

during quantization. Consequently, a dequantization of weights introduces an error of 1.

- Visualizing these errors across a broader weight distribution in a neural network reveals the inherent challenges of quantile quantization. This discrepancy underscores why QLoRA functions more as a fine-tuning mechanism than a standalone quantization strategy, despite its applicability for 4-bit inference. During fine-tuning with QLoRA, the LoRA tuning mechanism comes into play, involving the creation of two smaller weight update matrices. These matrices, maintained in a higher precision format such as brain float 16 or float 16, are then utilized to update the neural network weights.



- It’s noteworthy that throughout backpropagation and the forward pass, the weights of the network undergo de-quantization, ensuring that actual

training occurs in higher precision formats. Although the storage remains in lower precision, this deliberate choice introduces quantization errors. However, the model training process itself exhibits the capacity to adapt and mitigate these inefficiencies inherent in the quantization process. In essence, the LoRA training approach with higher precision aids the model in learning about and actively reducing quantization errors.

QLoRA finetuning with HuggingFace

To do QLoRA finetuning with HuggingFace, you need to install both the BitsandBytes library and the PEFT library. The BitsandBytes library takes care of the 4-bit quantization and the whole low-precision storage and high-precision compute part. The PEFT library will be used for the LoRA finetuning part.

```
import torch
from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

model_id = "EleutherAI/gpt-neox-20b"
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

# setup bits and bytes config
model = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=bnb_c

model.gradient_checkpointing_enable()

model = prepare_model_for_kbit_training(model) # prepares the whole model for kb
config = LoraConfig(
    r=8,
    lora_alpha=32,
```

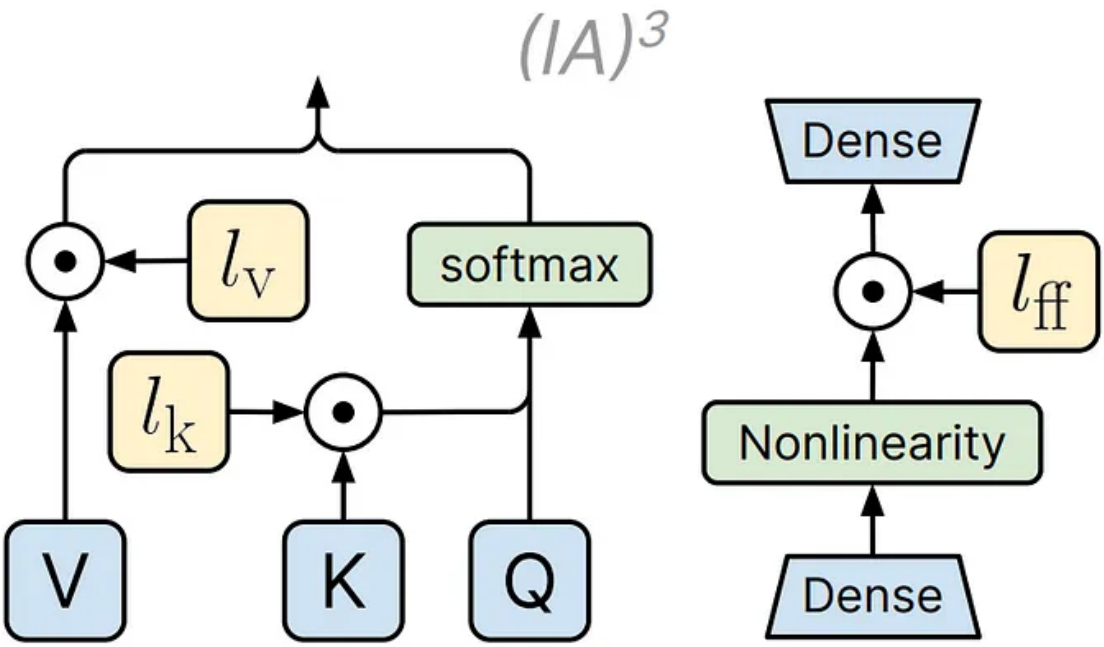
```
        target_modules=["query_key_value"],
        lora_dropout=0.05,
        bias="none",
        task_type="CAUSAL_LM"
    )
    model = get_peft_model(model, config) # Now you get a model ready for QLoRA trai
```

And then once again you can move to normal training using the HF trainer. Check out this colab notebook as a guide for QLoRA training.

7.6. IA3

IA3 (Infused Adapter by Inhibiting and Amplifying Inner Activations) is an adapter-based technique that is somewhat similar to LoRA. The goal of the authors was to replicate the advantages of ICL (in context learning or Few-Shot prompting) without the issues that come with it. ICL can get messy in terms of cost and inference as it requires prompting the model with examples. Longer length prompts require more time and computation to process. But ICL is perhaps the easiest way to get started working with models.

IA3 works by introducing **rescaling vectors** that target the activations of the model. A total of 3 vectors are introduced, *lv, ik,* and *lff.* These vectors target the *value, keys* in the attention layer, and the *non-linear* layer in the dense layers. These vectors are multiplied elementwise to the default values in the model. Once injected, these parameters are then learned during the training process, while the rest of the model remains frozen. These learned vectors essentially rescale or optimize the targeted pretrained model weights for the task at hand.



Source

So far this seems like a basic adapter type PEFT method. But that's not all. The authors also use 3 loss terms to enhance the learning process. The 3 losses are *LLM, LUL,* and *LLN*. *LLM* is the standard cross-entropy loss, which increases the likelihood of generating the correct response. Then there is *LUL* which is *Unlikelihood Loss*. This loss term reduces the probability of incorrect outputs using Rank Classification. Finally, we have *LLN*, which is a length-normalized loss that applies a softmax cross-entropy loss to length-normalized log probabilities of all output choices. Multiple losses are used here to ensure faster and better learning of the model. Because we are trying learn using few-shot examples, these losses are necessary.

Now let's talk about two very important concepts in IA3. Rank Classification and Length Normalization.

In Rank Classification a model is asked to rank a set of responses by their correctness. This is done by calculating the probability scores for the potential responses. The LUL is then used to reduce the probability of the

wrong responses and as a result, increase the probability of the correct response. But with Rank classification, we face a critical problem, which is that the responses with fewer tokens will rank higher, because of how probability works. A smaller amount of generated tokens ensures a higher probability as the probability of every generated token is < 1. To fix this, the authors propose dividing the score of the response by the number of tokens in the response. Doing this will normalize the scores. One very important thing to note here is that normalization is done over log probabilities, not raw probabilities. Log probabilities are negative and between zero to one.

Example Usage

For the task of sequence classification, one can initialize the IA3 config for a Llama model as follows:

```
peft_config = IA3Config(
    task_type=TaskType.SEQ_CLS, target_modules=["k_proj", "v_proj", "down_proj"]
)
```

7.7. P-Tuning

The P-tuning method aims to optimize the representation of the prompt which is passed to the model. In the P-Tuning paper, the authors emphasize how prompt engineering is a very strong technique when working with large language models. The p-Tuning method builds up on top of prompt engineering and tries to further improve the effectiveness of a good prompt.

P-Tuning works by creating a small *encoder network* for your prompt that creates a *soft prompt* for your passed prompt. To tune your LLM using P-

tuning, you are supposed to create a *prompt template* that represents your prompt. And a context *x* which is used in the template to get label *y*. This is the approach mentioned in the paper. The tokens used for the prompt template are trainable and learnable parameters, these are called *pseudo tokens*. We also add a prompt encoder which then helps us update pseudo tokens to the specific task at hand. The prompt encoder is usually a *bi-LSTM* network that learns the optimal representation of the prompt for the model and then passes the representation to it. The LSTM network is attached to the original model. Only the encoder network and the pseudo tokens are trained here, the weights of the original network remain unaffected. Once the training is done, the LSTM head is discarded as we have the *hi* which can be used directly.

In short, the prompt encoder only changes the embeddings of the passed prompt to better represent the task, everything else remains unchanged.

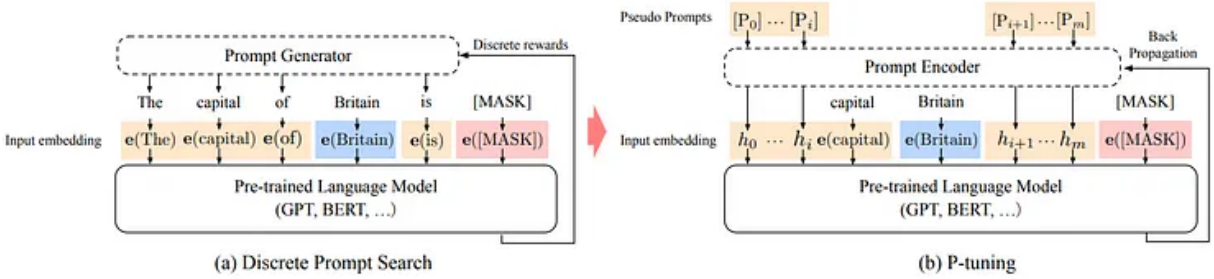


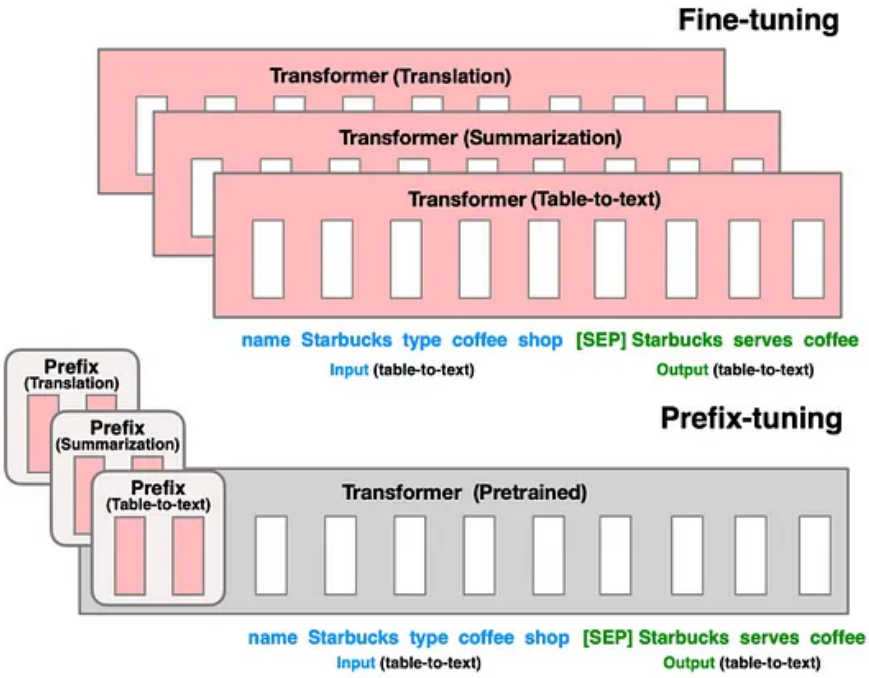
Figure 2. An example of prompt search for "The capital of Britain is [MASK]". Given the context (blue zone, "Britain") and target (red zone, "[MASK]"), the orange zone refer to the prompt tokens. In (a), the prompt generator only receives discrete rewards; on the contrary, in (b) the pseudo prompts and prompt encoder can be optimized in a differentiable way. Sometimes, adding few task-related anchor tokens (such as "capital" in (b)) will bring further improvement.

Source

7.8. Prefix Tuning

Prefix tuning can be considered the next version of P-Tuning. The authors of P-Tuning published a paper on P-Tuning V-2 addressing the issues of P-Tuning. In this paper, they implemented the Prefix tuning introduced in this

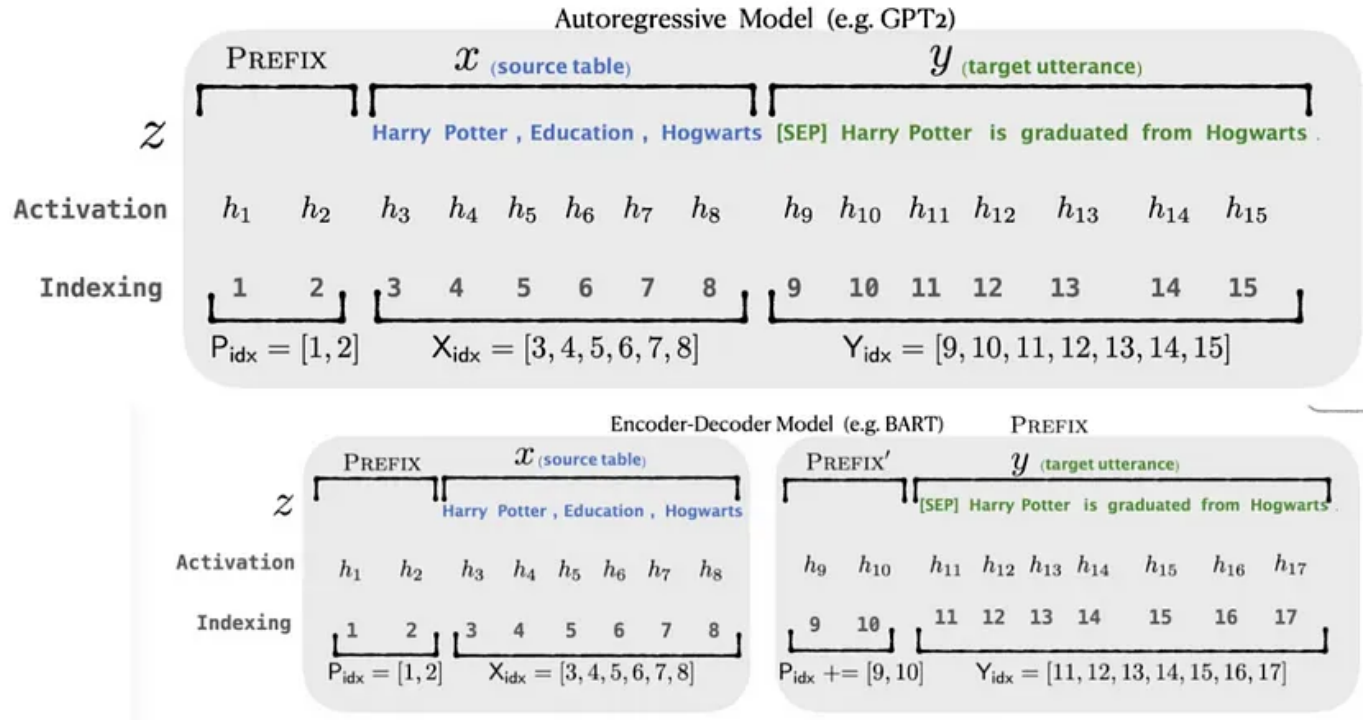
paper. Prefix tuning and P-Tuning do not have a lot of differences but can still lead to different results. Let's dive into a deeper explanation.



Source

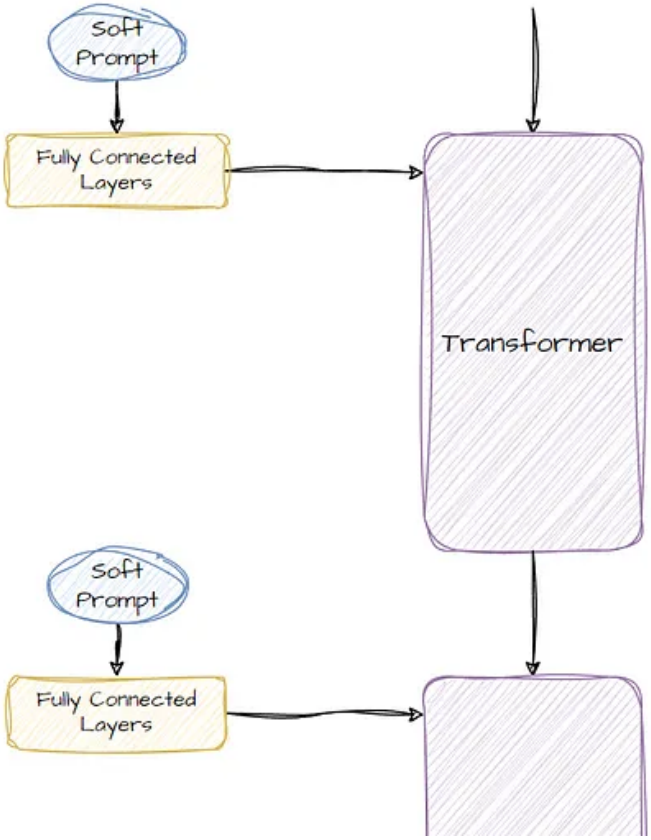
In P-Tuning, we added learnable parameters only to the input embeddings but in Prefix Tuning we add them **to all the layers of the network**. This ensures that the model itself learns more about the task it is being finetuned on. We append learnable parameters to the prompt and to every layer activation in the transformer layers. The difference from P-Tuning is that instead of completely modifying the prompt embeddings, we only add very few learnable parameters at the start of the prompt at every layer.

Here's a visual explanation:



Prefix Tuning

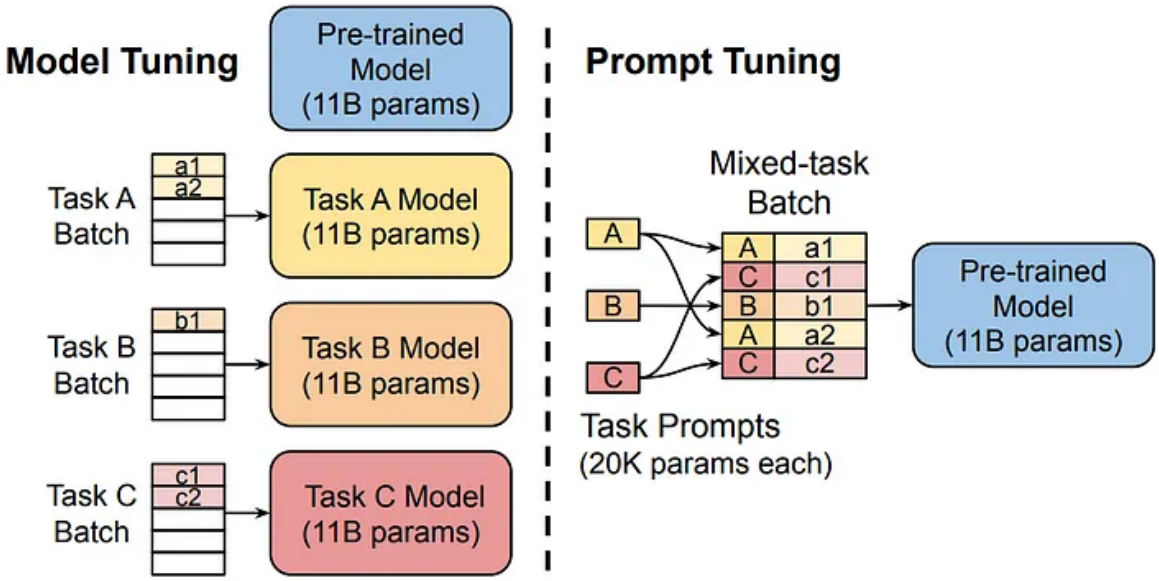
At every layer in the transformer, we concatenate a soft prompt with the input which has learnable parameters. These learnable parameters are tuned using a very small MLP, only 2 fully connected layers. This is done because in the paper authors note that directly updating these prompt tokens is very sensitive to learning rate and initialization. The soft prompts increase the number of trainable parameters but substantially increase the learning ability of the model too. The MLP or fully connected layers can be dropped later as we only care about the soft prompts, which will be appended to the input sequences during inference and will guide the model.



Source

7.9. Prompt Tuning (Not Prompt Engineering)

Prompt tuning was one of the first papers to build upon the idea of fine tuning only with soft prompts. Prompt tuning is a very simple and easy-to-implement idea. It involves prepending a specific prompt to the input and using virtual tokens or new trainable tokens for that specific prompt. These new virtual tokens can be finetuned during the process to learn a better representation of the prompt. This means that the model is tuned to understand the prompt better. Here is a comparison of prompt tuning with full fine-tuning from the paper:



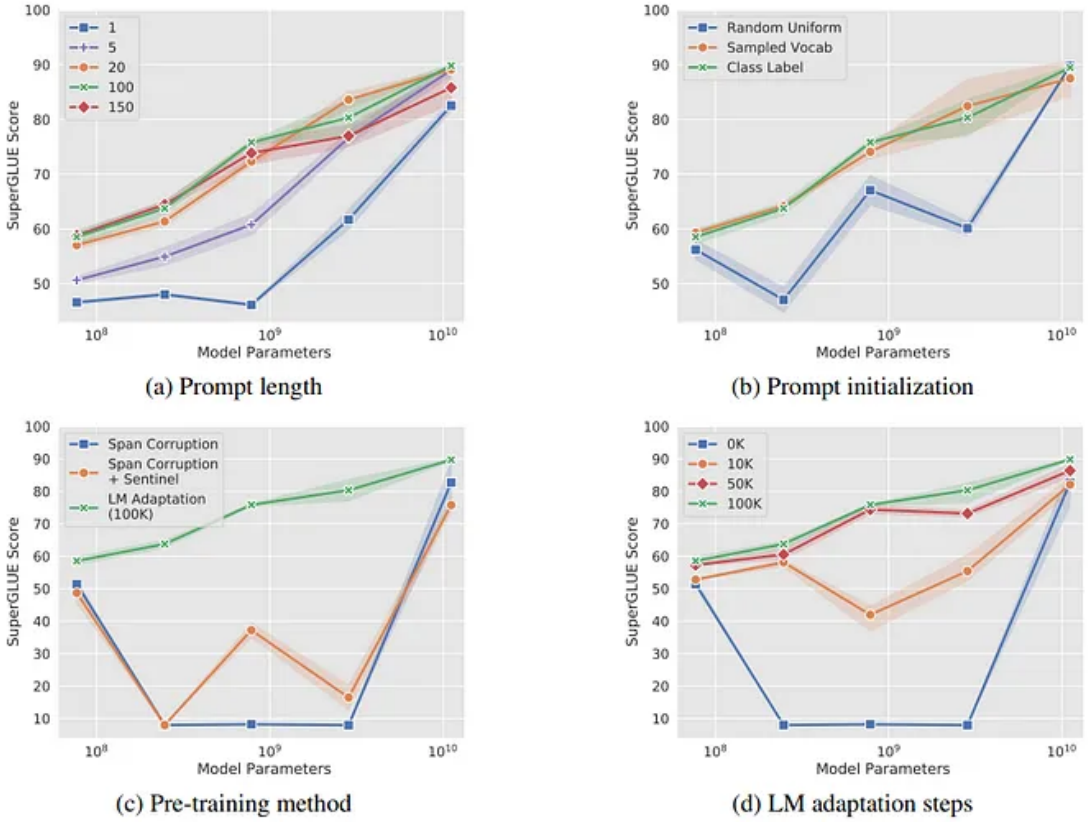
Source

Here you can see that full model tuning requires multiple copies of the model to exist if we want to use the model for multiple tasks. But with Prompt Tuning, you only need to store the learned virtual tokens of the prompt tokens. So for example, if you use a prompt like *"Classify this tweet: {tweet}"* the goal will be to learn new better embeddings for the prompt. And during inference, only these new embeddings will be used to generate the outputs. This allows the model to tune the prompt to help itself generate better outputs during inference.

Efficiency of Prompt Tuning

The biggest advantage of using prompt tuning is the small size of learned parameters. The files can be in **KBs.** As we can determine the dimension size and number of parameters to use for the new tokens, we can greatly control the number of parameters we are going to learn. In the paper, the authors show how even with a very small number of trainable tokens method

performs really well. And the performance only goes up as bigger models are used. You can read the paper here.



Source

Another big advantage is that we can use the same model **without any changes** for multiple tasks, as the only thing being updated are the embeddings of the prompt tokens. Meaning you can use the same model for a tweet classification task and for a language generation task without any changes to the model itself, given the model is big and sophisticated enough to perform those tasks. But a big limitation is that the model itself doesn't learn anything new. This is purely a prompt optimization task. This means if the model has never trained on a sentiment classification dataset, prompt tuning might not be of any help. It is **very important** to note that this method optimizes the prompts, not the model. So, if you cannot handcraft a *hard*

prompt that can do the task relatively well, there is no use of trying to optimize for a *soft* prompt using prompt optimization techniques.

Hard Prompt & Soft Prompt

Hard Prompts can be seen as the idea of a defined prompt which is static, or at best a template. A generative AI application can also have multiple prompt templates at its disposal to make use of.

> Hard prompts are manually handcrafted text prompts with discrete input tokens. ~ HuggingFace

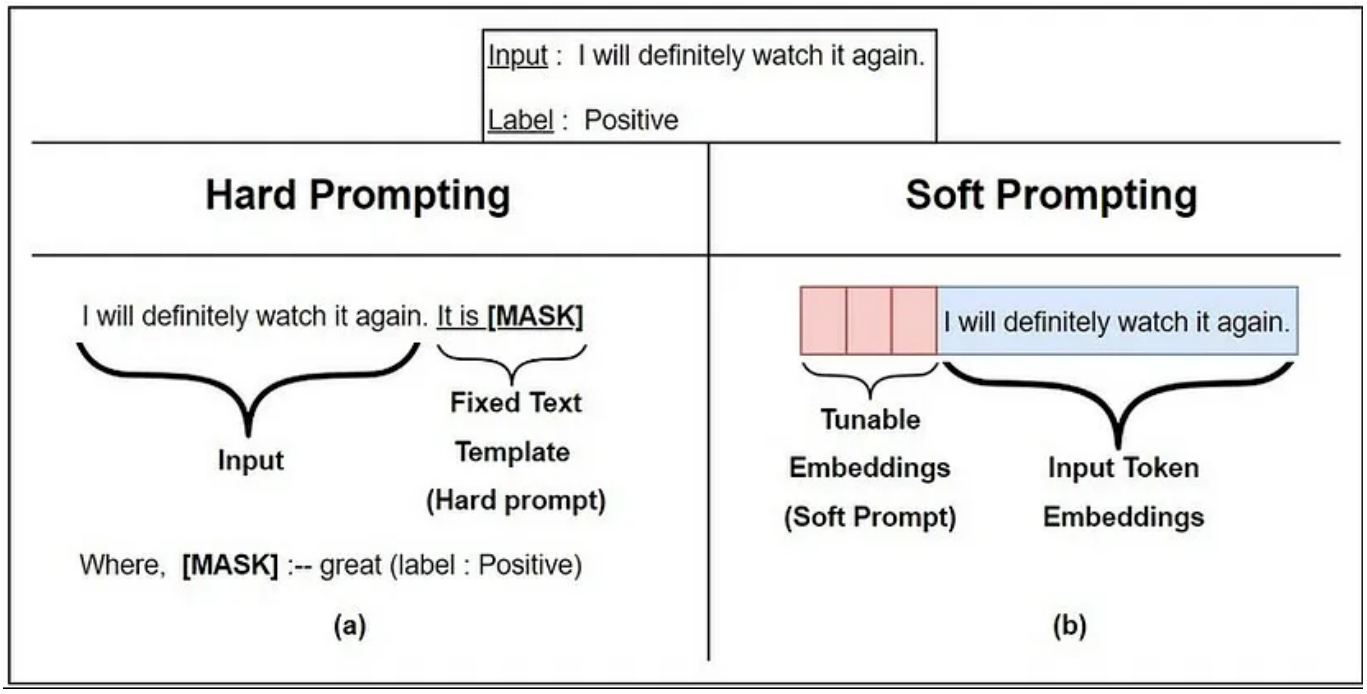
Prompt templating allows for prompts to be *stored, re-used, shared,* and *programmed.* And generative prompts can be incorporated in programs for programming, storage and re-use.

Soft prompts are created during the process of prompt tuning.

Unlike hard prompts, soft prompts cannot be viewed and edited in text. Prompts consist of an embedding, a string of numbers, that derives knowledge from the larger model.

So for sure, a disadvantage is the lack of interpretability of soft prompts. The AI discovers prompts relevant for a specific task but can't explain why it chose those embeddings. Like deep learning models themselves, soft prompts are opaque.

Soft prompts act as a substitute for additional training data.



The comparison between the Hard Prompting and Soft Prompting for T5

7.10. LoRA vs Prompt Tuning

Now we have explored various PEFT techniques. Now the question becomes whether to use an additive technique like Adapter and LoRA or you use a Prompt based technique like P-Tuning and Prefix Tuning.

On comparing LoRA vs P-Tuning and Prefix Tuning, one can say for sure LoRA is the best strategy in terms of getting the most out of the model. But it might not be the most efficient based on your needs. If you want to **train** the model on a much different task than what it has been trained on, LoRA is without a doubt the best strategy for tuning the model efficiently. But if your task is more or less already understood by the model, but the challenge is to properly prompt the model, then you should use Prompt Tuning techniques. Prompt Tuning doesn't modify many parameters in the model and mainly focuses on the passed prompt instead.

One important point to note is that LoRA decomposes the weight updation matrix into smaller rank matrices and uses them to update the weights of the

model. Even though trainable parameters are low, LoRA updates all the parameters in the targeted parts of the neural network. Whereas in Prompt Tuning techniques, a few trainable parameters are added to the model, this usually helps the model adjust to and understand the task better but does not help the model learn new properties well.

7.11. LoRA and PEFT in comparison to full Fine Tuning

PEFT, Parameter Efficient Fine Tuning, is proposed as an alternative to full Finetuning. For most of the tasks, it has already been shown in papers that PEFT techniques like LoRA are comparable to full finetuning, if not better. But, if the new task you want the model to adapt to is completely different from the tasks the model has been trained on, PEFT might not be enough for you. The limited number of trainable parameters can result in major issues in such scenarios.

If we are trying to build a code generation model using a text-based model like LLaMA or Alpaca, we should probably consider fine-tuning the whole model instead of tuning the model using LoRA. This is because the task is too different from what the model already knows and has been trained on. Another good example of such a task is training a model, which only understands English, to generate text in the Nepali language.

8. LLM Inference

When using a large language model (LLM) for inference, we can often configure various parameters to fine-tune its output and performance. Here's a breakdown of some key parameters:

1. Top-k Sampling:

- Samples only the top-k most likely tokens at each step, encouraging diversity and preventing repetition. Higher k leads to more diverse but potentially less coherent outputs.

2. Temperature:

- Influences the probability distribution over possible next tokens, controlling randomness and "creativity."
- Lower temperature generates more likely but potentially repetitive text, while higher temperature encourages diversity and less predictable outputs.

3. Top-P (Nucleus) Sampling:

- Top-P or nucleus sampling limits the selection of tokens to a subset of the vocabulary with cumulative probability mass up to a threshold value. It helps in controlling the diversity of generated output.

4. Maximum Length:

- Sets the maximum number of tokens the LLM generates, preventing excessively long outputs.

5. Context Prompting:

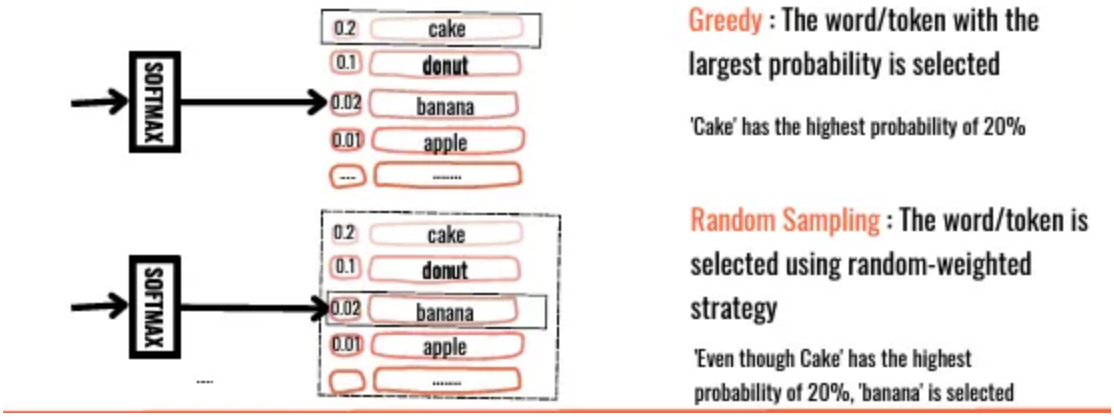
- By providing a specific context prompt or input, you can guide the model to generate text that aligns with that context. This can help ensure that the generated output is relevant and coherent within the given context.

6. Repetition Penalty:

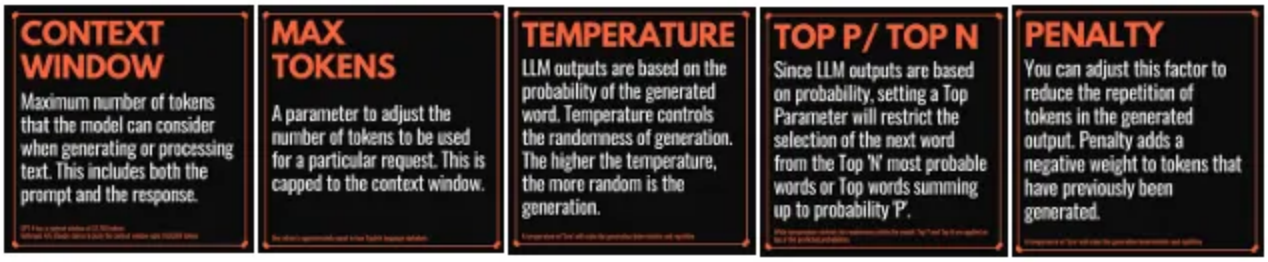
- Penalizes sequences with repeated n-grams, encouraging diversity and originality.

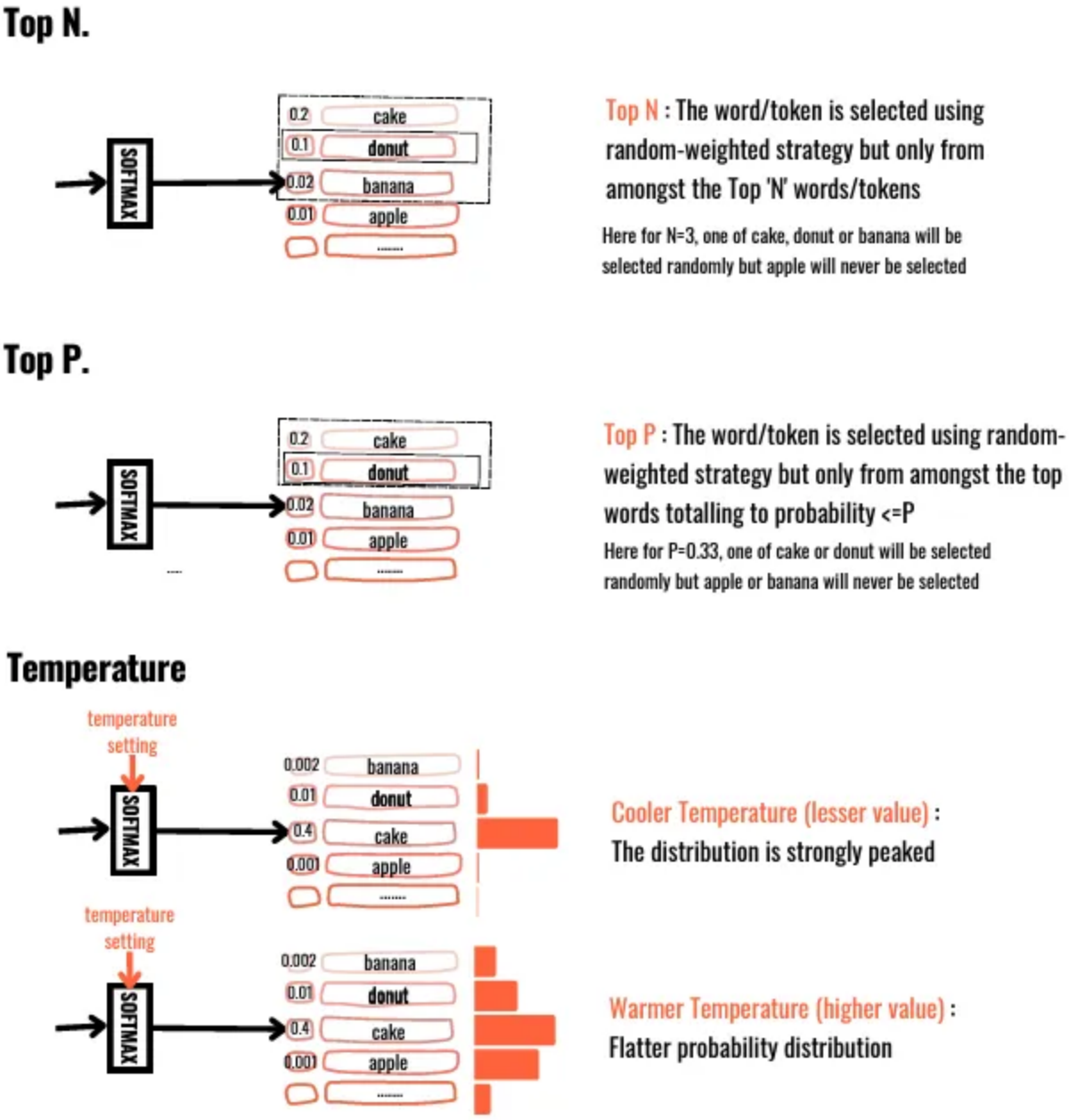
7. Sampling:

- Chooses between deterministic (greedy) and random sampling-based generation. Greedy mode picks the most likely token at each step, while random sampling introduces randomness.
- Greedy mode prioritizes accuracy, while random sampling encourages diversity and creativity.
- **Beam Search:** Maintains multiple potential sequences, expanding the most promising ones at each step, aiming for more coherent and accurate outputs compared to top-k sampling.



Credits : Abhinav Kimothi





Credits : Abhinav Kimothi

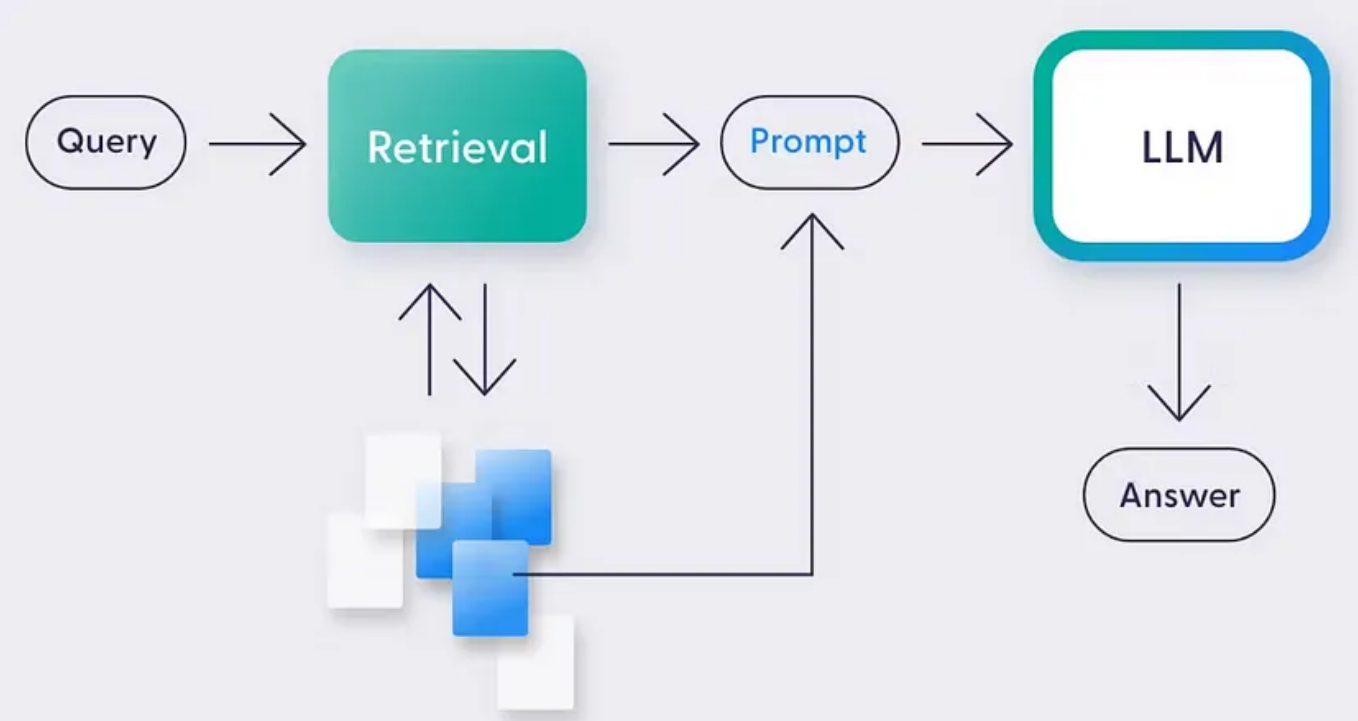
9. Prompt Engineering

Prompt Engineering, also known as **In-Context Prompting**, refers to methods for how to communicate with LLM to steer its behavior for desired

outcomes *without* updating the model weights. It is an empirical science and the effect of prompt engineering methods can vary a lot among models, thus requiring heavy experimentation and heuristics.

What is a Prompt?

The natural language instruction in which we interact with an LLM is called a **Prompt**. The construction of prompts is called **Prompt Engineering**.



Role of a Prompt

The inferencing that an LLM does and completes the instruction given in the prompt is called **In-Context Learning**.

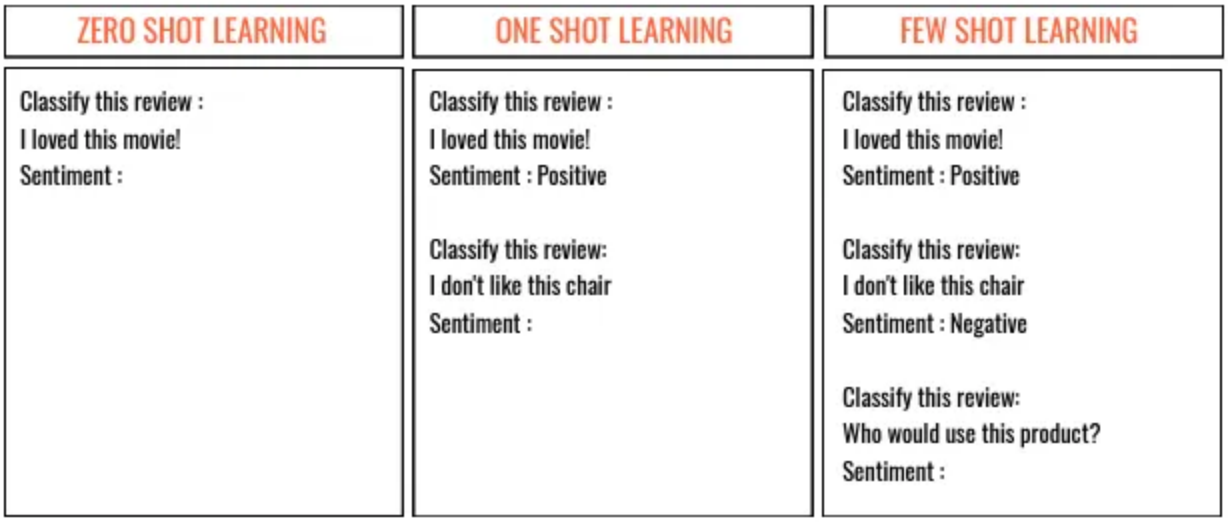
Few-Shot Prompting

The ability of the LLM to respond to the instruction in the prompt without any example is called **Zero-Shot Learning**.

When a single example is provided, it's called **One-Shot Learning.**

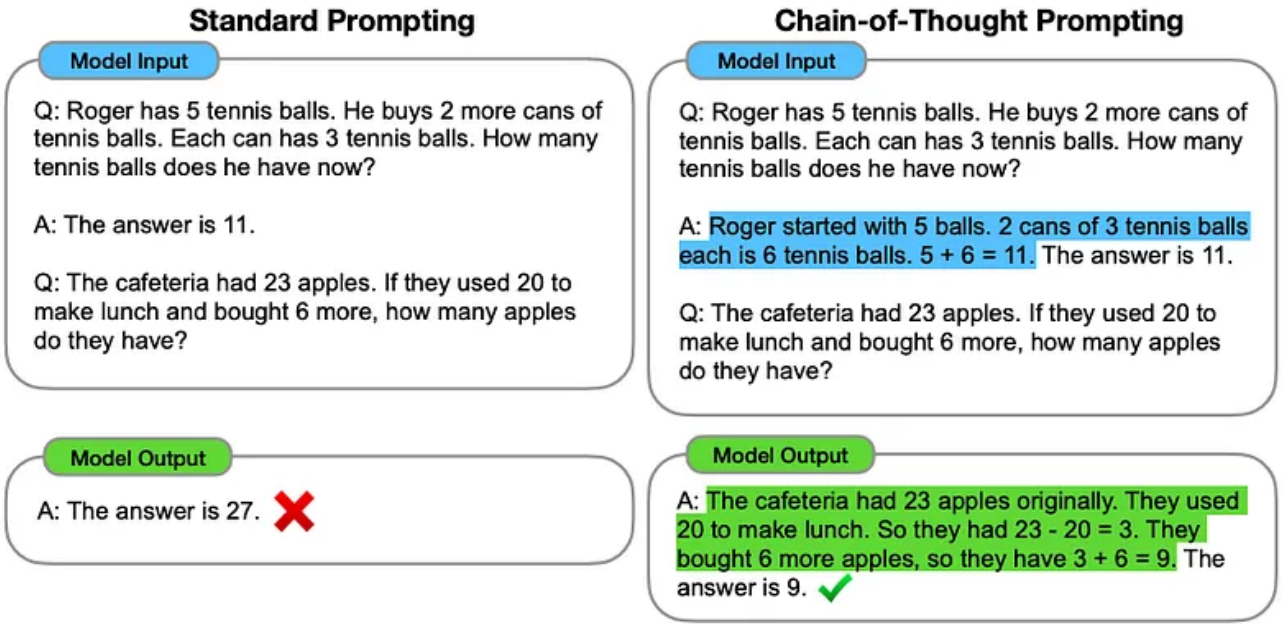
If more than one example is provided, it's called **Few-Shot Learning.**

Context Window, or the maximum number of tokens that an LLM can provide and inference on, is critical in the **Zero/One/Few Shot Learning.**



Credits: Abhinav Kimothi

9.1. Chain-of-Thought (CoT) Prompting

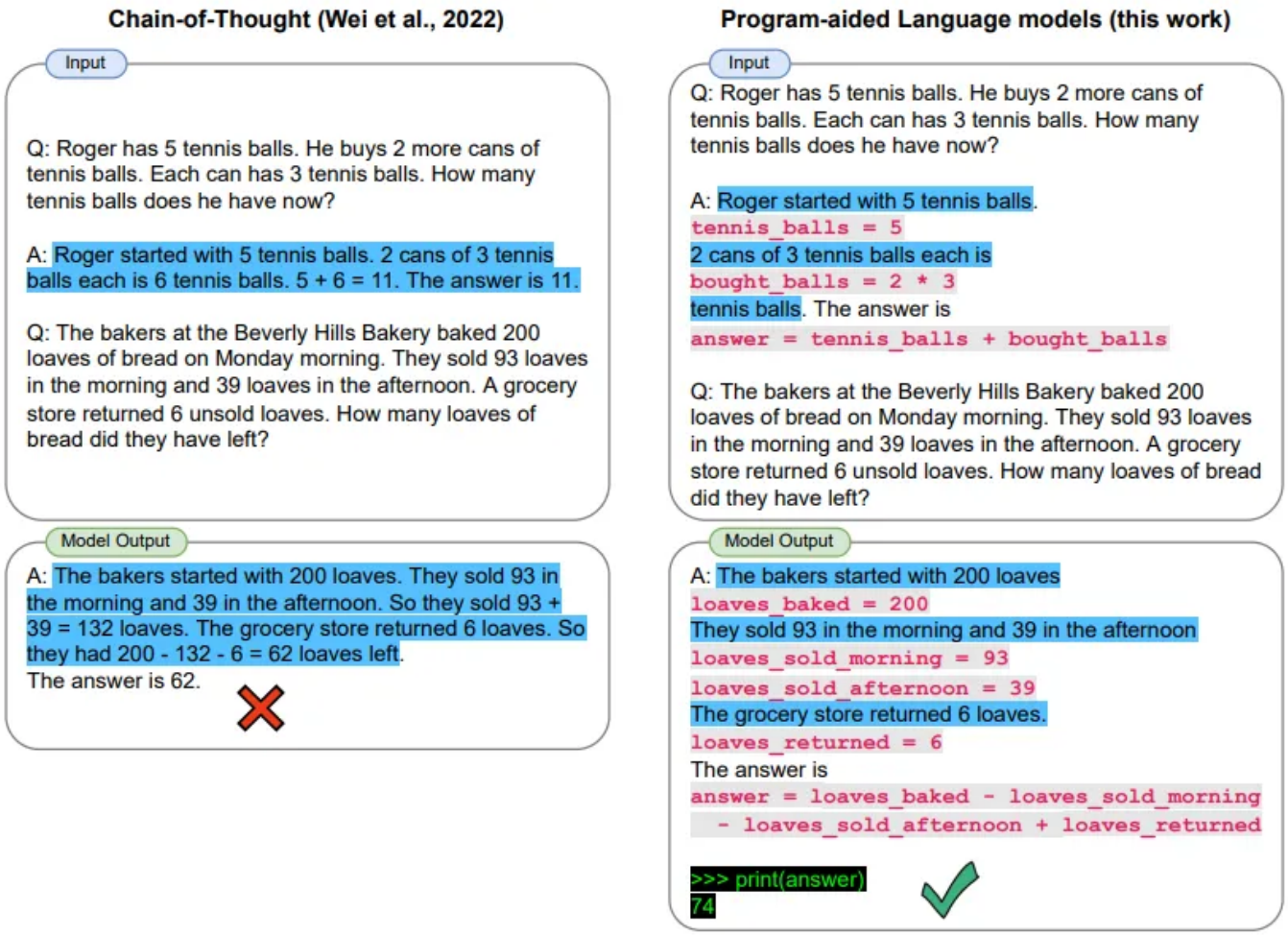


Source

Chain-of-thought (CoT) prompting (Wei et al. 2022) generates a sequence of short sentences to describe reasoning logics step by step, known as *reasoning chains* or *rationales*, to eventually lead to the final answer. The benefit of CoT is more pronounced for **complicated reasoning tasks** while using **large models** (e.g. with more than 50B parameters). Simple tasks only benefit slightly from CoT prompting.

9.2. PAL (Program-Aided Language Models)

Gao et al., (2022)(opens in a new tab) presents a method that uses LLMs to read natural language problems and generate programs as the intermediate reasoning steps. Coined, program-aided language models (PAL), it differs from chain-of-thought prompting in that instead of using free-form text to obtain solution it offloads the solution step to a programmatic runtime such as a Python interpreter.



Source

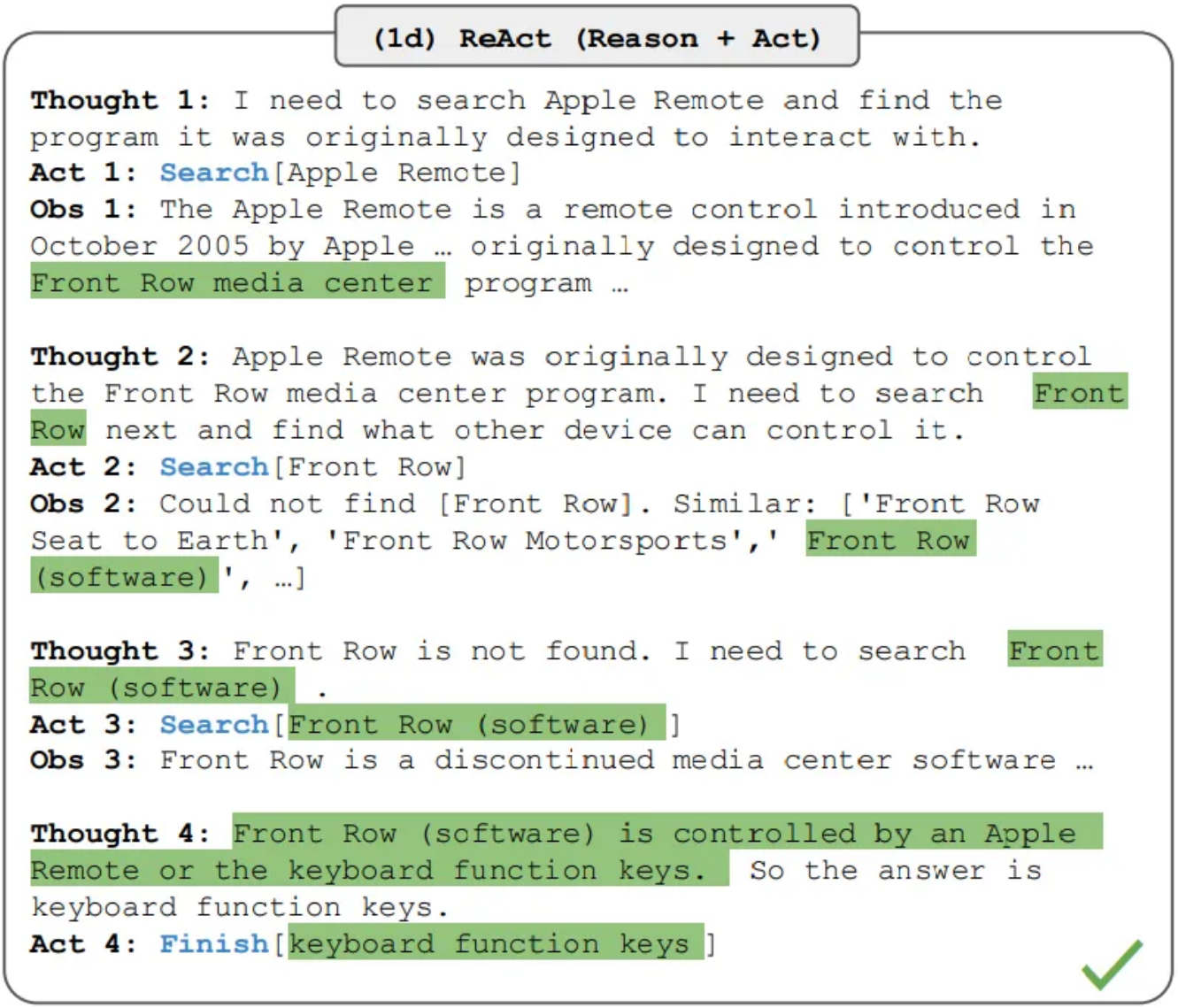
9.3. ReAct Prompting

ReAct is inspired by the synergies between “acting” and “reasoning” which allow humans to learn new tasks and make decisions or reasoning.

CoT’s lack of access to the external world or inability to update its knowledge can lead to issues like fact hallucination and error propagation.

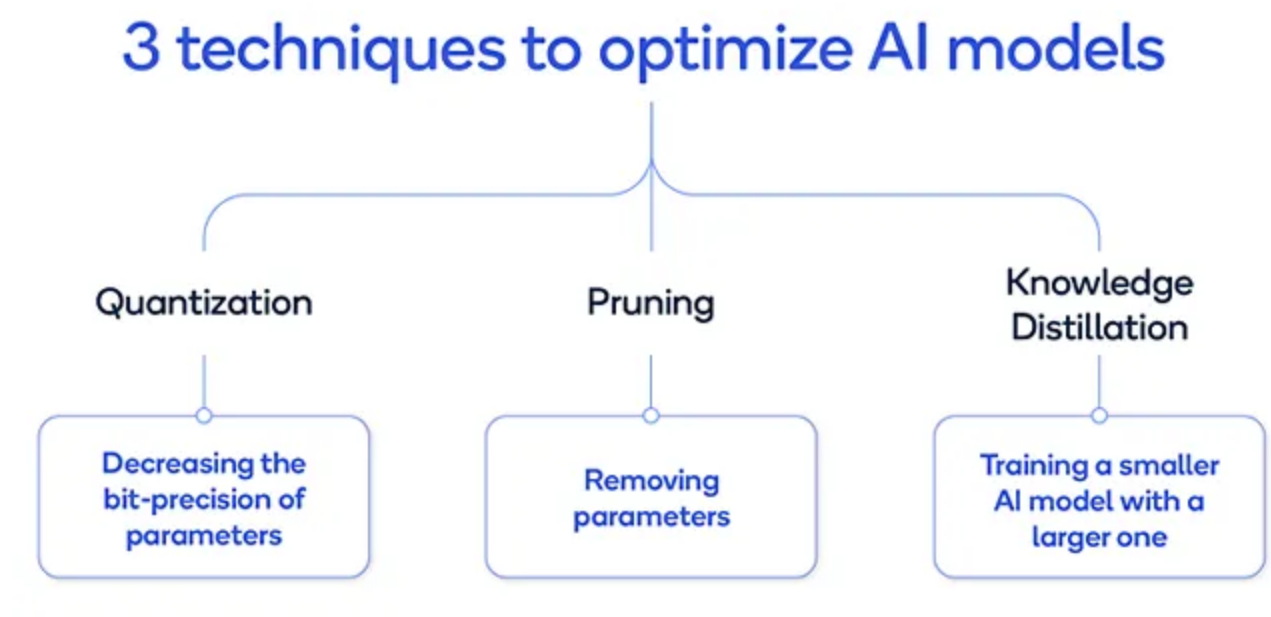
ReAct is a general paradigm that combines reasoning and acting with LLMs. ReAct prompts LLMs to generate verbal reasoning traces and actions for a task. This allows the system to perform dynamic reasoning to create, maintain, and adjust plans for acting while also enabling interaction to external environments (e.g., Wikipedia) to incorporate additional

information into the reasoning. The figure below shows an example of ReAct and the different steps involved to perform question answering.

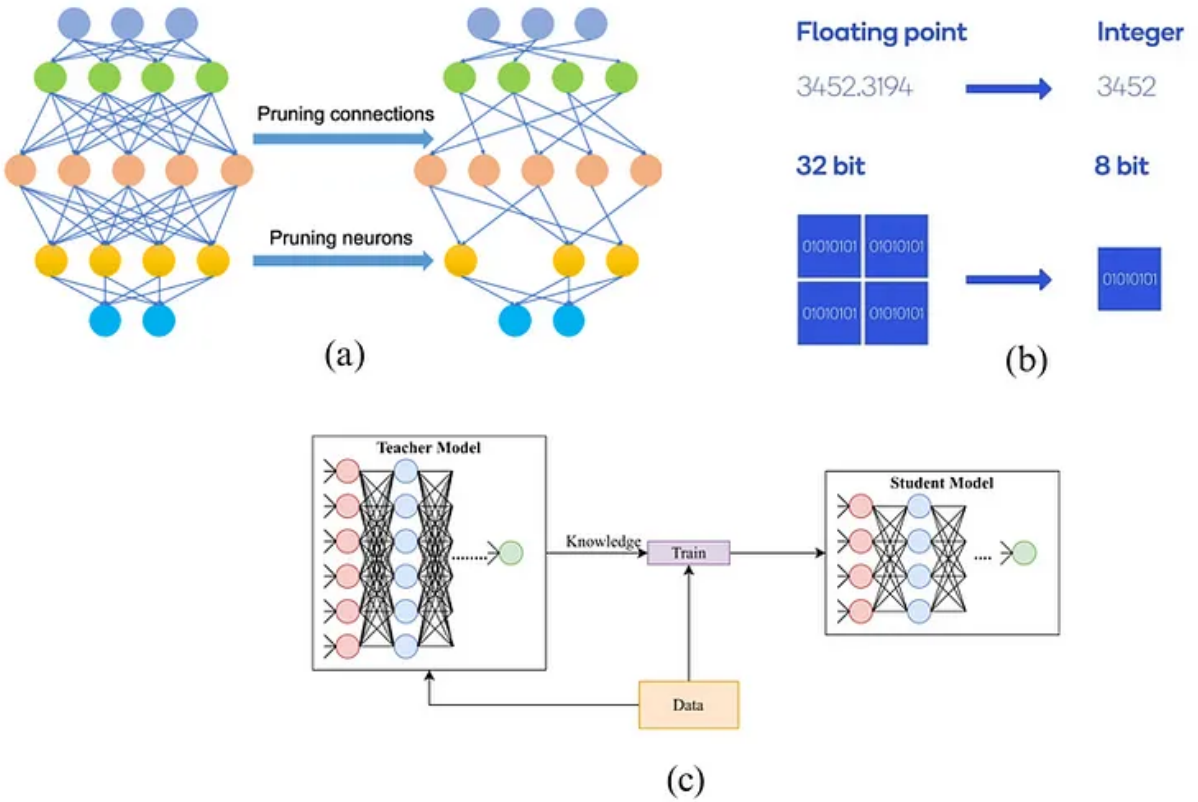


Source

10. Model Optimization Techniques



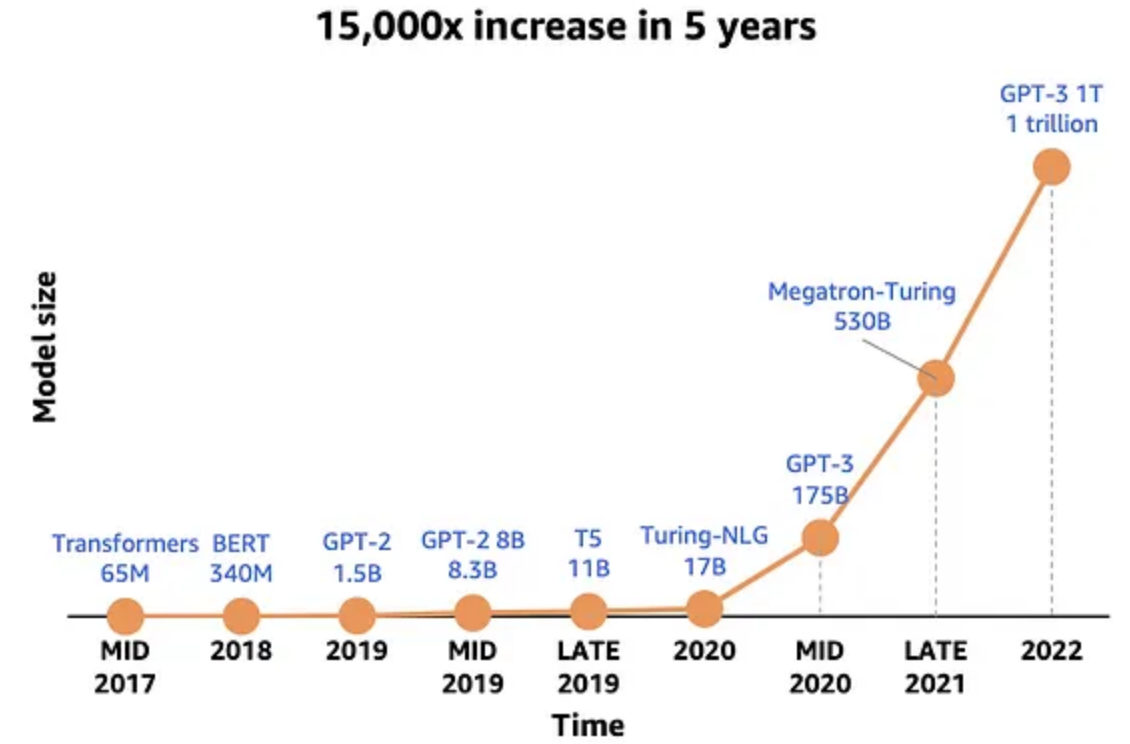
Source



Model compression methods: (a) pruning, (b) quantization, and © knowledge distillation

10.1. Quantization

Model Quantization is a technique used to reduce the size of large neural networks, including large language models (LLMs) by modifying the precision of their weights. LLM Quantization is enabled thanks to empiric results showing that while some operations related to neural network training and inference must leverage high precision, in some cases it's possible to use significantly lower precision (float16 for example) reducing the overall size of the model, allowing it to be run using less powerful hardware with an **acceptable reduction of its capabilities and accuracy.**



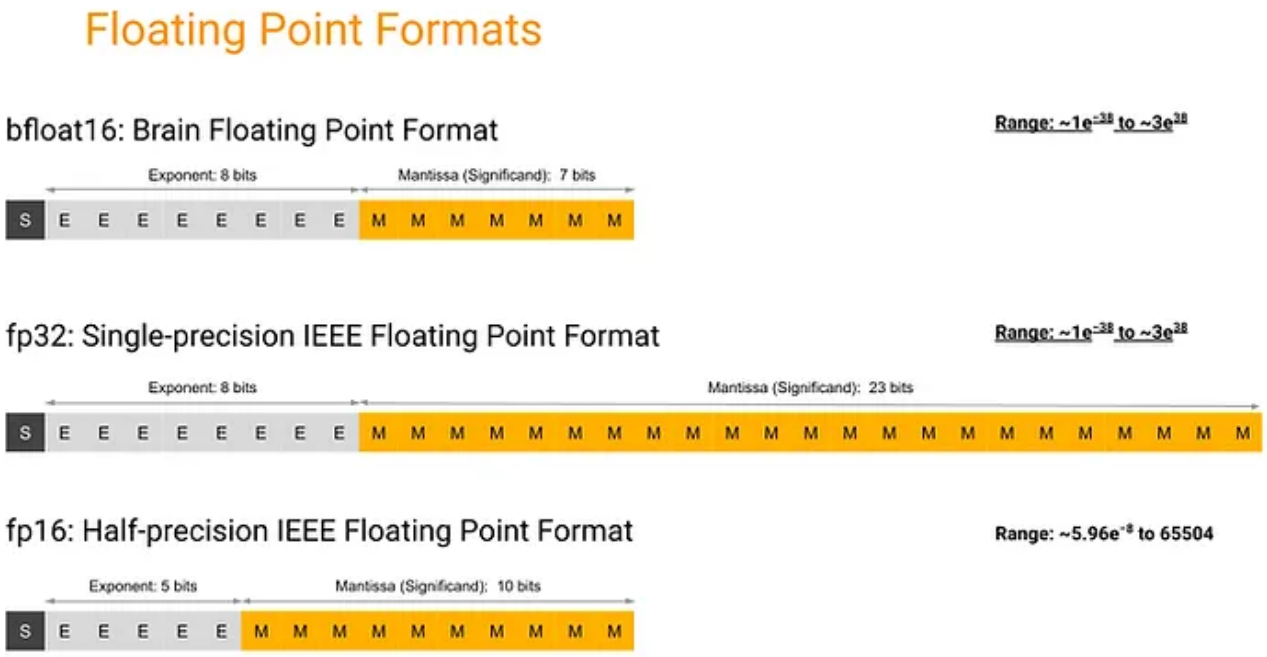
Trend of model sizes | AWS reinvent

Precision Trade-off

Tensors | Source

Generally, using high precision in neural networks is associated with better accuracy and more stable training(read more about it here). Using high precision is also more computationally expensive as it requires **more hardware** and more **expensive hardware.** Research mostly done by Google and NVIDIA regarding the possibility of using lower precision for some neural network operations showed that lower precision can be leveraged for some training and inference operations.

Aside from the research, both companies developed hardware and frameworks to support lower precision operations. For example, the NVIDIA T4 accelerators are lower precision GPUs with Tensor Cores technology that is significantly more efficient than that of the K80. Google's TPUs introduced the concept of bfloat16, a special primitive data type optimized for neural networks. **The fundamental idea behind lower precision is that neural networks don't always need to use ALL the range that 64-bit floats to allow them to perform well.**



The bfloat16 numerical format | Google

As neural networks became increasingly large, the importance of leveraging lower precision had a significant impact on the ability to use them. With LLMs, this became even more crucial.

For reference, an A100 GPU by Nvidia has 80GB of memory in its most advanced version. In the table below we can see that the LLama2–70B model requires 138 GB of memory approximately, meaning that to host it, we will need multiple A100s. Distributing models over multiple GPUs means paying for more GPUs as well as overhead infrastructure. A quantized version, on the other hand, requires around 40 GB of memory, therefore it can fit easily into one A100, reducing the cost of inference significantly. This example doesn't even mention the fact that within the single A100, using quantized models would result in faster execution of most of the individual computation operations.

Model	Original Size (FP16)	Quantized Size (INT4)
Llama2-7B	13.5 GB	3.9 GB
Llama2-13B	26.1 GB	7.3 GB
Llama2-70B	138 GB	40.7 GB

Example of 4-bit quantization using llama.cpp, size may vary slightly depending on the method

How does quantization shrink models?

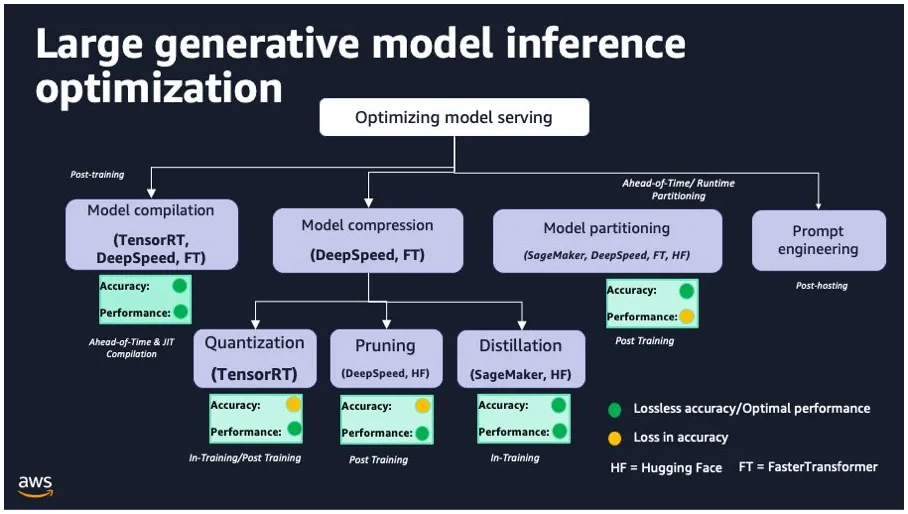
Quantization significantly decreases the model's size by **reducing the number of bits required for each model weight.** A typical scenario would be the reduction of the weights from FP16 (16-bit Floating-point) to INT4 (4-bit Integer). This allows for models to run on **cheaper hardware** and/or with **higher speed**. By reducing the precision of the weights, the **overall quality of the LLM can also suffer some impact.**

Studies show that this impact varies depending on the techniques used and that larger models suffer less from change in precision. Larger models (over ~70B) can maintain their capacities even when converted to 4-bit, with some **techniques such as the NF4 suggesting no impact on their performance.** Therefore, **4-bit appears to be the best compromise** between performance and size/speed for these **larger models**, while 6 or 8-bit might be better for smaller models.

Types of LLM Quantization

It's possible to divide the techniques of obtaining quantized models into two:

1. **Post-Training Quantization (PTQ)**: converting the weights of an already trained model to a lower precision without any retraining. Though straightforward and easy to implement, PTQ might degrade the model's performance slightly due to the loss of precision in the value of the weights.
2. **Quantization-Aware Training (QAT)**: Unlike PTQ, QAT integrates the weight conversion process during the training stage. This often results in superior model performance, but it's more computationally demanding. A highly used QAT technique is the QLoRA.



This post will only focus on **PTQ** strategies and the key distinctions between them.

Larger Quantized Model vs Smaller Non-Quantized

Acknowledging that reducing the precision will reduce the accuracy of the model, should you prefer a **smaller full-precision model** or a **larger quantized model** with a comparable inference cost? Although the ideal choice might vary due to diverse factors, recent research by Meta offers some insightful guidelines.

While we expect that reducing the precision would result in the reduction of the accuracy, Meta researchers have demonstrated that in some cases, not only does the **quantized model demonstrate superior performance**, but it also allows for **reduced latency** and enhanced throughput. The same trend can be observed when comparing an **8-bit 13B model with a 16-bit 7B model**. In essence, when comparing models with similar inference costs, **the larger quantized models can outperform their smaller, non-quantized counterparts**. This advantage becomes even more pronounced with larger networks, as they exhibit a smaller quality loss when quantized.

Where to find already Quantized models?

Fortunately, it is possible to **find many versions of models** already quantized using **GPTQ** (some compatible with ExLLama), **NF4** or **GGML** on the Hugging Face Hub. A quick glance would reveal that a substantial chunk of these models has been quantified by TheBloke, an influential and respected figure in the LLM community. This user has published several models with different types of quantization methods so one can choose to use the best fit for each particular use-case.

To easily experiment with these models open up a Google Colab and make sure you change your **runtime to a GPU** (a free one is available for use). Start by installing the transformers library maintained by Hugging Face and all necessary libraries. Since we will be using a model **quantized using Auto-GPTQ** the respective libraries will also be required:

```
!pip install transformers
!pip install accelerate

# Due to using GPTQ
```

```
!pip install optimum
!pip install auto-gptq
```

You might need to restart the runtime so that the installs are available. Then simply load the already quantized model, in this case we are loading a Llama-2–7B-Chat **model previously quantized** using Auto-GPTQ, as shown below:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

model_id = "TheBloke/Llama-2-7b-Chat-GPTQ"
tokenizer = AutoTokenizer.from_pretrained(model_id, torch_dtype=torch.float16, d
model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16
```

Quantizing a LLM

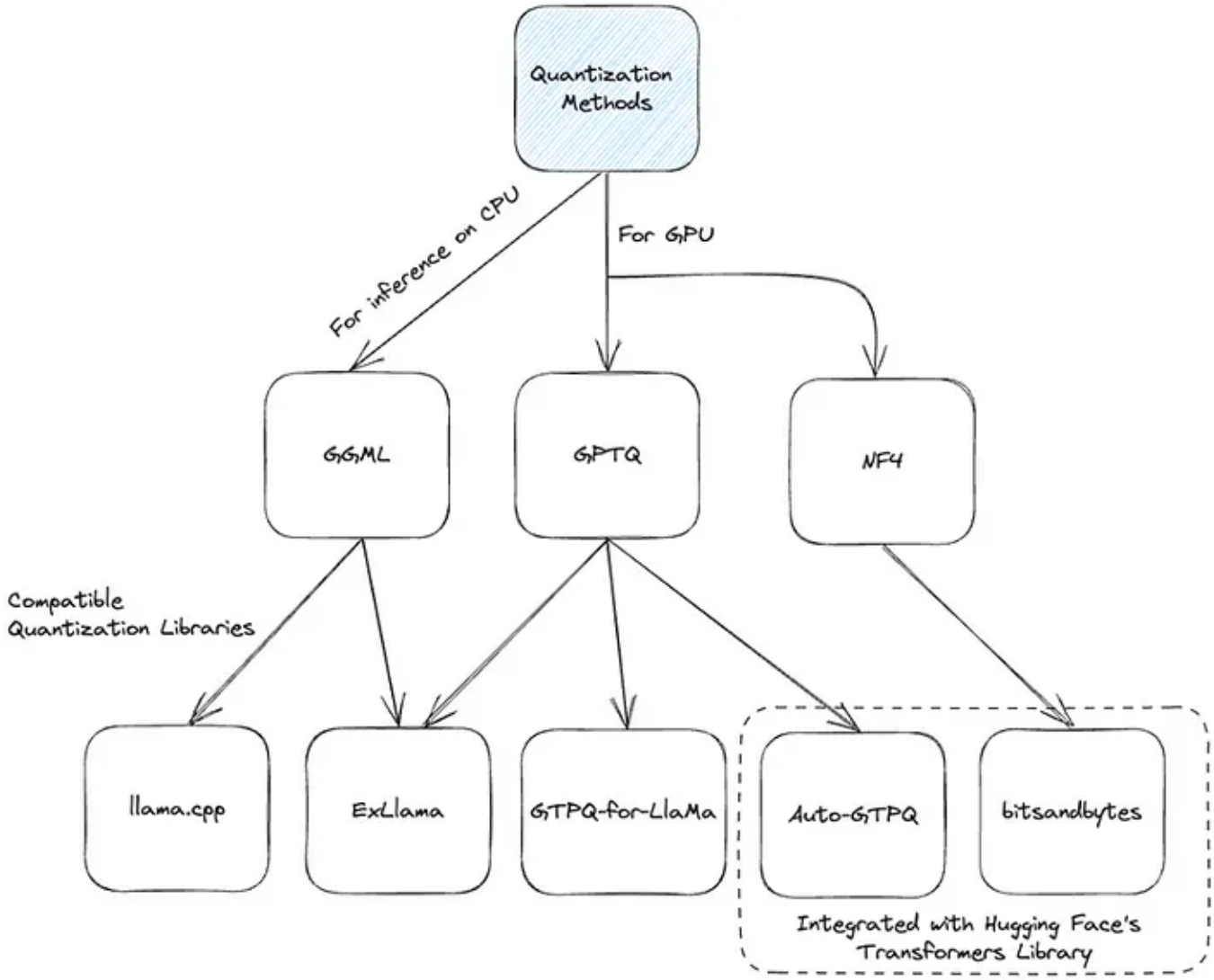
As highlighted earlier, a plethora of quantized models already reside on the **Hugging Face Hub**, eliminating the necessity to compress a model personally in many scenarios. However, in same cases you may want to **use models which are not yet quantized** or you may want to compress the model yourself. This can be achieved by using a dataset tailored to your specific domain.

To demonstrate how to easily quantize a model using **AutoGPTQ** along with the Transformers library, we employed a streamlined variant of the AutoGPTQ interface found in Optimum — Hugging Face's solution for refining training and inference:

```
from transformers import AutoModelForCausalLM, AutoTokenizer, GPTQConfig

model_id = "facebook/opt-125m"

tokenizer = AutoTokenizer.from_pretrained(model_id)

quantization_config = GPTQConfig(bits=4, dataset = "c4", tokenizer=tokenizer)

model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto", quanti
```

Model compression can be time-consuming. For instance, a 175B model demands at least **4 GPU hours**, especially with expansive datasets like “c4”. Notably, the number of bits in the quantization process or the dataset can be easily modified by the parameters of GPTQConfig. Changing the dataset will impact how the quantization is done so, if possible, **use a dataset that resembles data seen in inference** to maximize performance.

Quantization Techniques



Techniques

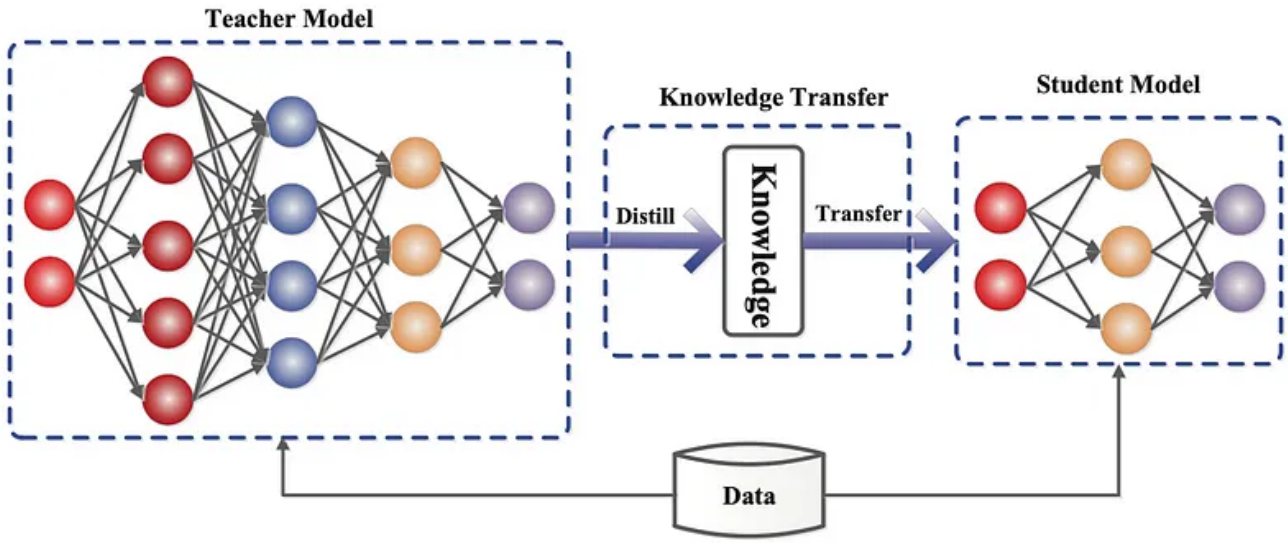
Several state-of-the-art methods have emerged in the arena of model quantization. Let's delve into some prominent ones:

1. **GPTQ:** With some implementation options AutoGPTQ, ExLlama and GPTQ-for-LLaMa, this method focuses mainly on GPU execution.
2. **NF4:** Being implemented on the bitsandbytes library it works closely with the Hugging Face transformers library. It is primarily **used by QLoRA** methods and loads models in 4-bit precision for fine-tuning.
3. **GGML:** This C library works closely with the llama.cpp library. It features a unique binary format for LLMs, allowing for fast loading and ease of

reading. Notably, its recent **shift to the GGUF format** ensures future extensibility and compatibility.

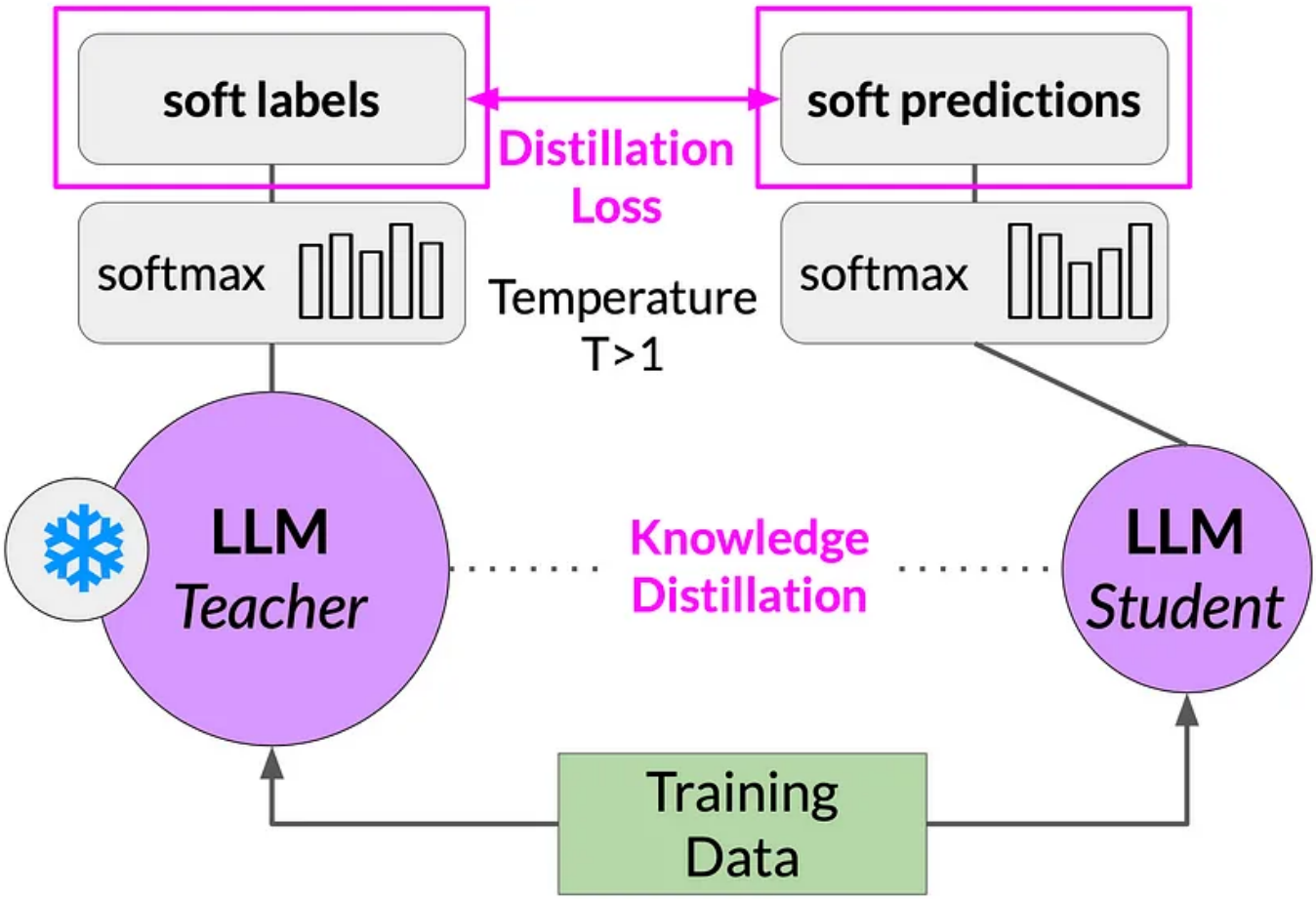
Many quantization libraries support several different quantization strategies (e.g. 4-bit, 5-bit, and 8-bit quantization), each of which offers different trade-offs between efficiency and performance.

10.2. Distillation



Source

Knowledge Distillation (**KD**; Hinton et al. 2015, Gou et al. 2020) is a straightforward way to build a smaller, cheaper model (*"student model"*) to speed up inference by transferring skills from a pre-trained expensive model (*"teacher model"*) into the student. There is no much restriction on how the student architecture should be constructed, except for a matched output space with the teacher to construct a proper learning objective.



Distillation

The teacher model is already fine-tuned on the training data. So, the probability distribution likely closely matches the ground truth data and won't have many variations in tokens.

So, when temperature > 1 then, probability distribution becomes broader.
T > 1 => Teacher's output –> soft labels and Student's output –> soft predictions
T = 1 => Teacher's output –> hard labels and Student's output –> hard predictions

```
Distillation is not as effective for generative decoder models. Its effective for encoder only models, such as BERT, which have a lot of representation redundancy.
```

10.3. Pruning

Network pruning is to reduce the model size by trimming unimportant model weights or connections while the model capacity remains. It may or may not require re-training. Pruning can be **unstructured** or **structured**.

- *Unstructured pruning* is allowed to drop any weight or connection, so it does not retain the original network architecture. Unstructured pruning often does not work well with modern hardware and doesn't lead to actual inference speedup.
- *Structured pruning* aims to maintain the dense matrix multiplication form where some elements are zeros. They may need to follow certain pattern restrictions to work with what hardware kernel supports. Here we focus on structured pruning to achieve *high sparsity* in transformer models.

A routine workflow to construct a pruned network has three steps:

1. Train a dense network until convergence;
2. Prune the network to remove unwanted structures;
3. Optionally retrain the network to recover the performance with new weights.

The idea of discovering a sparse structure within a dense model via network pruning while the sparse network can still maintain similar performance is motivated by **Lottery Ticket Hypothesis** (**LTH**): A randomly initialized, dense, feed-forward network contains a pool of subnetworks and among them, only a subset (a sparse network) are *"winning tickets"* which can achieve the optimal performance when trained in isolation.

We will continue with the Evaluation of LLMs in our next blog.

Conclusion

In this blog we explored the text generation part of the Retrieval-Augmented Generation (RAG) application, emphasizing the use of Large Language Models (LLM). It covers language modeling, pre-training challenges, quantization techniques, distributed training methods, and fine-tuning for LLMs. Parameter Efficient Fine-Tuning (PEFT) techniques, including Adapters, LoRA, and QLoRA, are discussed. Prompting strategies, model compression methods like pruning and quantization, and various quantization techniques (GPTQ, NF4, GGML) are introduced. The blog concludes with insights into distillation and pruning for model size reduction.

Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentations, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

Below is a consolidated list of references:

1. https://www.assemblyai.com/blog/the-full-story-of-large-language-models-and-rlhf/
2. https://www.baeldung.com/cs/large-language-models
3. https://www.appypie.com/blog/architecture-and-components-of-llms
4. https://jalammar.github.io/illustrated-transformer/

5. https://www.mercity.ai/blog-post/fine-tuning-llms-using-peft-and-lora
6. https://www.linkedin.com/pulse/transforming-ai-landscape-loras-technical-llm-training-prasun-mishra/
7. https://www.researchgate.net/figure/The-comparison-between-the-previous-T5-prompt-tuning-method-part-a-and-the-introduced_fig1_366062946
8. https://www.deepset.ai/blog/llm-finetuning
9. https://www.qualcomm.com/news/onq/2023/12/optimizing-generative-ai-for-edge-devices
10. https://www.tensorops.ai/post/what-are-quantized-llms
11. https://lilianweng.github.io/posts/2023-01-10-inference-optimization/
12. https://techtalkverse.com/post/artificial-intelligence/llm-basics/#optimization-techniques

Thank you for reading!

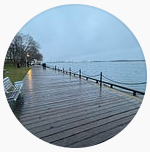
If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 👏 or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.
- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides my future posts.

Connect with me!

Vipra

Large Language Models | Fine Tuning | Training | Prompt Engineering | Optimization

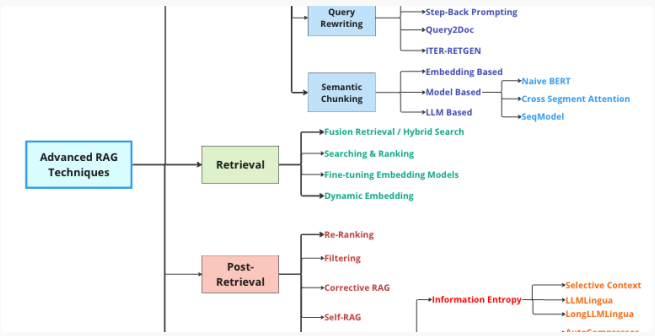


Written by Vipra Singh

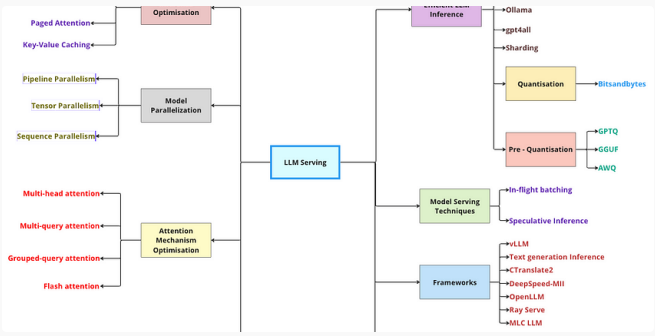
1K Followers

Follow

More from Vipra Singh



Vipra Singh



Vipra Singh

Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✨ · 48 min read · Apr 27, 2024

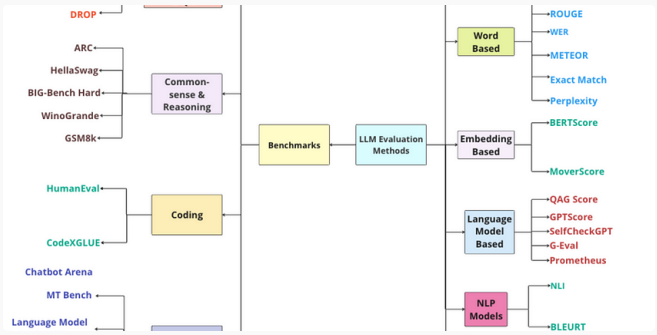
384 2

Building LLM Applications: Serving LLMs (Part 9)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✨ · 50 min read · Apr 17, 2024

546 3



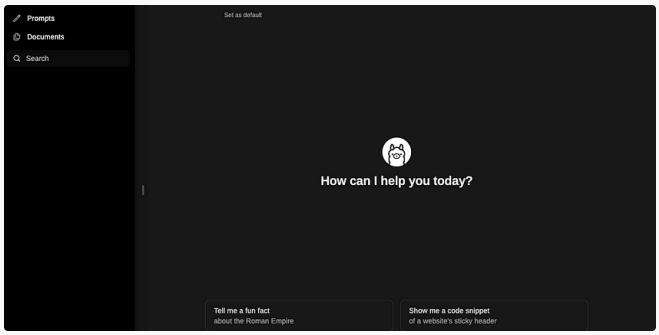
Vipra Singh

Building LLM Applications: Evaluation (Part 8)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✨ · 48 min read · Apr 7, 2024

276 1



Vipra Singh

Building LLM Applications: Open-Source RAG (Part 7)

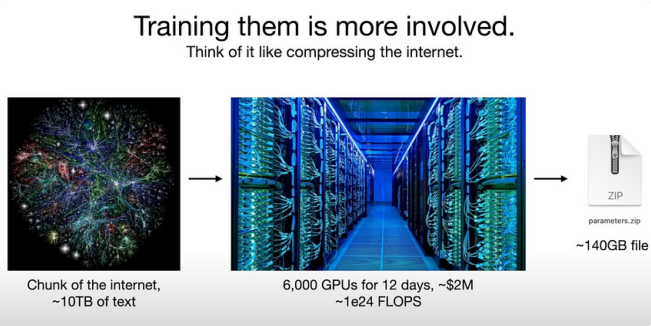
Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generatio...

✨ · 27 min read · Mar 16, 2024

223 1

See all from Vipra Singh

Recommended from Medium



Wayland Zhang

What is Large Language Model? (LLMs: Zero-to-Hero)

This is the first article of my Zero-to-Hero series. In this article we will take a high-level...

5 min read · Jan 24, 2024

21 1

Paul Iusztin in Decoding ML

The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-...

15 min read · May 4, 2024

1.4K 10

Lists

Natural Language Processing

1494 stories · 1011 saves

ChatGPT prompts

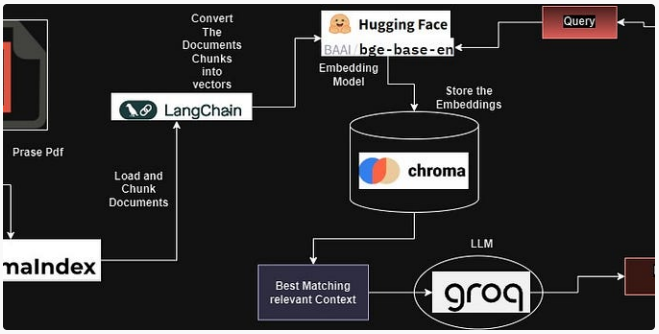
47 stories · 1642 saves

AI Regulation

6 stories · 473 saves

Generative AI Recommended Reading

52 stories · 1106 saves



Plaban Nayak in The AI Forum

RAG on Complex PDF using LlamaParse, Langchain and Groq

Retrieval-Augmented Generation (RAG) is a new approach that leverages Large Languag...

13 min read · Apr 7, 2024

678 9

Mahesh

How to Productionize Large Language Models (LLMs)

Understand LLMOps, architectural patterns, how to evaluate, fine tune & deploy...

94 min read · Mar 27, 2024

186 2

Mastering LLM (Large Language Model)

[Updated]Recent 32 Large Language Models (LLMs) Intervie...

Recent 11 large language models (LLMs) Interview Questions (Answered) for your nex...

15 min read · Apr 6, 2024

125 2

Fareed Khan in Level Up Coding

Building a Million-Parameter LLM from Scratch Using Python

A Step-by-Step Guide to Replicating LLaMA Architecture

25 min read · Dec 7, 2023

2.6K 32

See more recommendations