Foundational Large Language Models & Text Generation

Authors: Mohammadamin Barektain, Anant Nawalgaria, Daniel J. Mankowitz, Majd Al Merey, Yaniv Leviathan, Massimo Mascaro, Matan Kalman, Elena Buchatskaya, Aliaksei Severyn, and Antonio Gulli

Google

Acknowledgements

Reviewers and Contributors

Adam Sadvovsky

Yonghui Wu

Andrew Dai

Efi Kokiopolou

Chuck Sugnet

Aleksey Vlasenko

Erwin Huizenga

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Mark Iverson

Designer

Michael Lanning

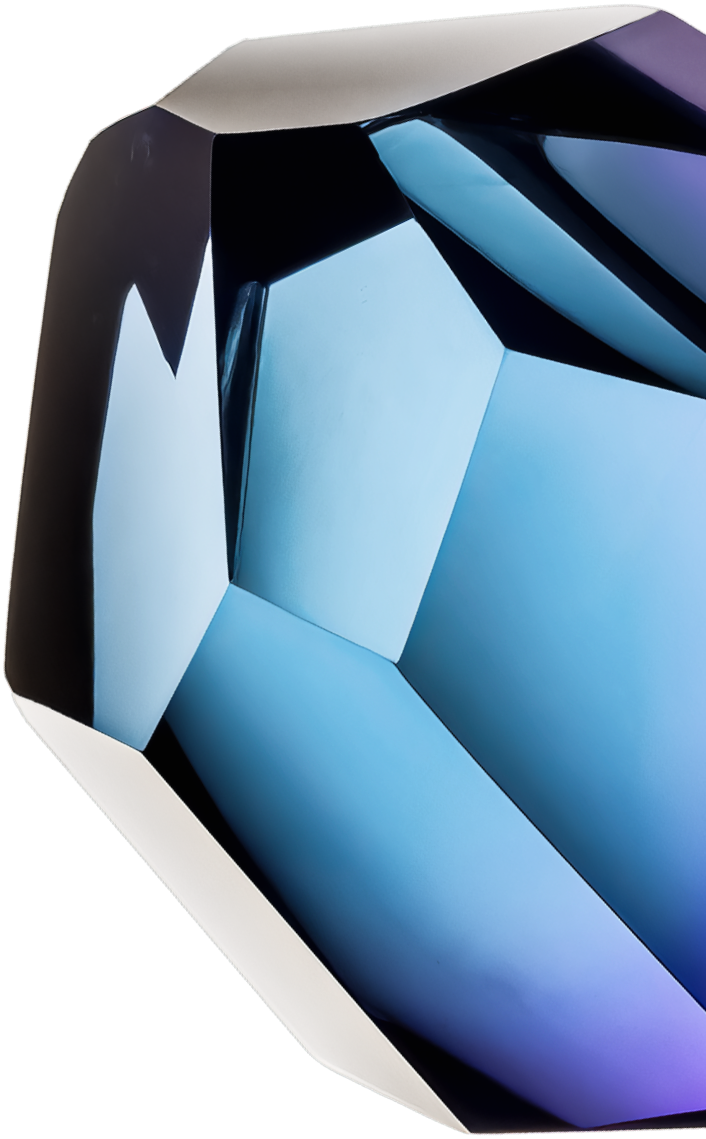
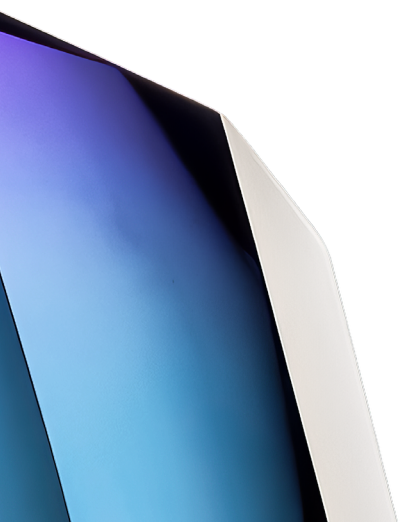


Table of contents



> We believe that this new crop of technologies has the potential to assist, complement, empower, and inspire people at any time across almost any field.

Introduction

The advent of Large Language Models (LLMs) represents a seismic shift in the world of artificial intelligence. Their ability to process, generate, and understand user intent is fundamentally changing the way we interact with information and technology.

An LLM is an advanced artificial intelligence system that specializes in processing, understanding, and generating human-like text. These systems are typically implemented as a deep neural network and are trained on massive amounts of text data. This allows them to learn the intricate patterns of language, giving them the ability to perform a variety of tasks, like machine translation, creative text generation, question answering, text summarization, and many more reasoning and language oriented tasks. This whitepaper dives into the timeline of the various architectures and approaches building up to the large language models and the architectures being used at the time of publication. It also discusses fine-

tuning techniques to customize an LLM to a certain domain or task, methods to make the training more efficient, as well as methods to accelerate inference. These are then followed by various applications and code examples.

Why language models are important

LLMs achieve an impressive performance boost from the previous state of the art across a variety of different and complex tasks which require answering questions or complex reasoning, making feasible many new applications. These include language translation, code generation and completion, text generation, text classification, and question-answering, to name a few. Although foundational LLMs trained in a variety of tasks on large amounts of data perform very well out of the box and display emergent behaviors (e.g. the ability to perform tasks they have not been directly trained for) they can also be adapted to solve specific tasks where performance out of the box is not at the level desired through a process known as fine-tuning. This requires significantly less data and computational resources than training an LLM from scratch. LLMs can be further nudged and guided towards the desired behavior by the discipline of *prompt engineering:* the art and science of composing the prompt and the parameters of an LLM to get the desired response.

The big question is: how do these large language models work? The next section explores the core building blocks of LLMs, focusing on transformer architectures and their evolution from the original 'Attention is all you need' paper[1] to the latest models such as Gemini, Google's most capable LLM. We also cover training and fine-tuning techniques, as well as methods to improve the speed of response generation. The whitepaper concludes with a few examples of how language models are used in practice.

Large language models

A *language model* predicts the probability of a sequence of words. Commonly, when given a prefix of text, a language model assigns probabilities to subsequent words. For example, given the prefix "The most famous city in the US is...", a language model might predict high probabilities to the words "New York" and "Los Angeles" and low probabilities to the words "laptop" or "apple". You can create a basic language model by storing an n-gram table,[2] while modern language models are often based on neural models, such as transformers.

Before the invention of transformers[1], recurrent neural networks (RNNSs) were the popular approach for modeling sequences. In particular, "long short-term memory" (LSTM) and "gated recurrent unit" (GRU) were common architectures.[3] This area includes language problems such as machine translation, text classification, text summarization, and question-answering, among others. RNNs process input and output sequences sequentially. They generate a sequence of hidden states based on the previous hidden state and the current input. The sequential nature of RNNs makes them compute-intensive and hard to parallelize during training (though recent work in state space modeling is attempting to overcome these challenges).

Transformers, on the other hand, are a type of neural network that can process sequences of tokens in parallel thanks to the self-attention mechanism.[1] This means that transformers can better model long-term contexts and are easier to parallelize than RNNs. This makes them significantly faster to train, and more powerful compared to RNNs for handling long-term dependencies in long sequence tasks. However, the cost of self-attention in the original transformers is quadratic in the context length which limits the size of the context, while RNNs have a theoretically infinite context length. Transformers have become the most popular approach for sequence modeling and transduction problems in recent years.

Herein, we discuss the first version of the transformer model and then move on to the more recent advanced models and algorithms.

Transformer

The *transformer architecture* was developed at Google in 2017 for use in a translation model.[1] It's a sequence-to-sequence model capable of converting sequences from one domain into sequences in another domain. For example, translating French sentences to English sentences. The original transformer architecture consists of two parts: an encoder and a decoder. The encoder converts the input text (e.g., a French sentence) into a representation, which is then passed to the decoder. The decoder uses this representation to generate the output text (e.g., an English translation) autoregressively.[1] Notably, the size of the output of the transformer encoder is linear in the size of its input. Figure 1 shows the design of the original transformer architecture.

The transformer consists of multiple layers. A layer in a neural network comprises a set of parameters that perform a specific transformation on the data. In the diagram you can see an example of some layers which include Multi-Head Attention, Add & Norm, Feed-Forward, Linear, Softmax etc. The layers can be sub-divided into the input, hidden and output layers. The input layer (e.g., Input/Output Embedding) is the layer where the raw data enters the network. *Input embeddings* are used to represent the input tokens to the model. *Output embeddings* are used to represent the output tokens that the model predicts. For example, in a machine translation model, the input embeddings would represent the words in the source language, while the output embeddings would represent the words in the target language. The output layer (e.g., Softmax) is the final layer that produces the output of the network. The hidden layers (e.g., Multi-Head Attention) are between the input and output layers and are where the magic happens!

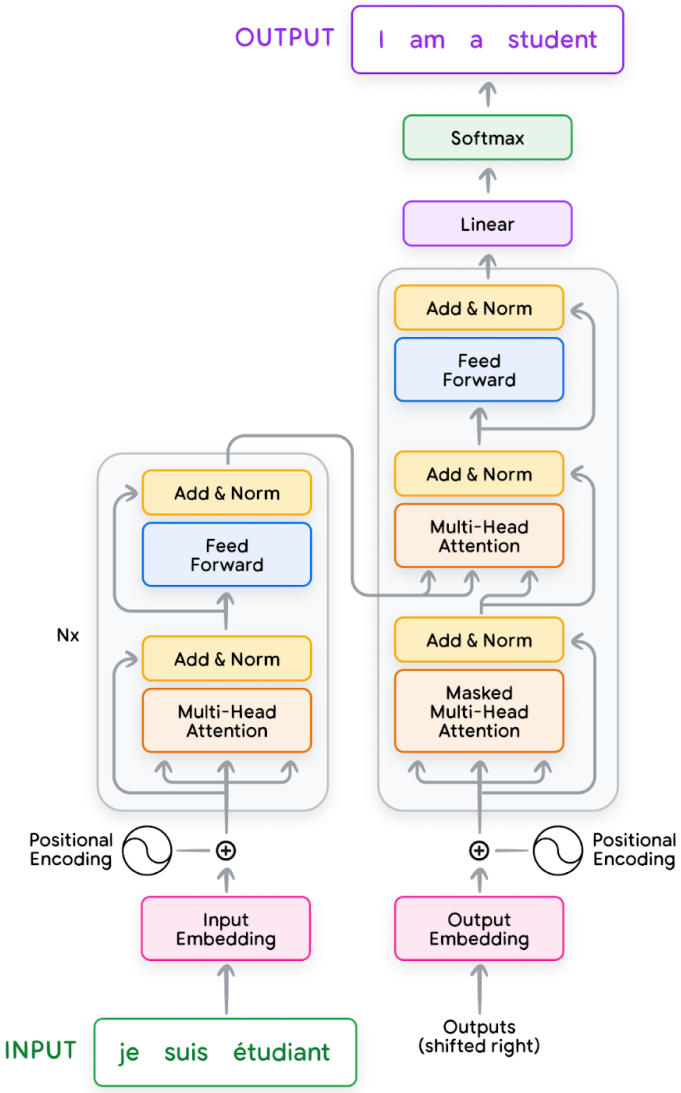


Figure 1. Original Transformer[1] (P.C:[5])

To better understand the different layers in the transformer, let's use a French-to-English translation task as an example. Here, we explain how a French sentence is input into the transformer and a corresponding English translation is output. We will also describe each of the components inside the transformer from Figure 1.

Input preparation and embedding

To prepare language inputs for transformers, we convert an input sequence into tokens and then into input embeddings. At a high level, an input embedding is a high-dimensional vector that represents the meaning of each token in the sentence. This embedding is then fed into the transformer for processing. Generating an input embedding involves the following steps:

1. **Normalization** (Optional): Standardizes text by removing redundant whitespace, accents, etc.
2. **Tokenization**: Breaks the sentence into words or subwords and maps them to integer token IDs from a vocabulary.
3. **Embedding**: Converts each token ID to its corresponding high-dimensional vector, typically using a lookup table. These can be learned during the training process.
4. **Positional Encoding**: Adds information about the position of each token in the sequence to help the transformer understand word order.

These steps help to prepare the input for the transformers so that they can better understand the meaning of the text.

Multi-head attention

After converting input tokens into embedding vectors, you feed these embeddings into the multi-head attention module (see Figure 1). Self-attention is a crucial mechanism in transformers; it enables them to focus on specific parts of the input sequence relevant to the task at hand and to capture long-range dependencies within sequences more effectively than traditional RNNs.

Understanding self-attention

Consider the following sentence: "The tiger jumped out of a tree to get a drink because it was thirsty." Self-attention helps to determine relationships between different words and phrases in sentences. For example, in this sentence, "the tiger" and "it" are the same object, so we would expect these two words to be strongly connected. Self-attention achieves this through the following steps (Figure 2):

1. **Creating queries, keys, and values:** Each input embedding is multiplied by three learned weight matrices (Wq, Wk, Wv) to generate query (Q), key (K), and value (V) vectors. These are like specialized representations of each word.
    - Query: The query vector helps the model ask, "Which other words in the sequence are relevant to me?"
    - Key: The key vector is like a label that helps the model identify how a word might be relevant to other words in the sequence.
    - Value: The value vector holds the actual word content information.
2. **Calculating scores:** Scores are calculated to determine how much each word should 'attend' to other words. This is done by taking the dot product of the query vector of one word with the key vectors of all the words in the sequence.

3. **Normalization:** The scores are divided by the square root of the key vector dimension (dk) for stability, then passed through a softmax function to obtain attention weights. These weights indicate how strongly each word is connected to the others.
4. **Weighted values:** Each value vector is multiplied by its corresponding attention weight. The results are summed up, producing a context-aware representation for each word.

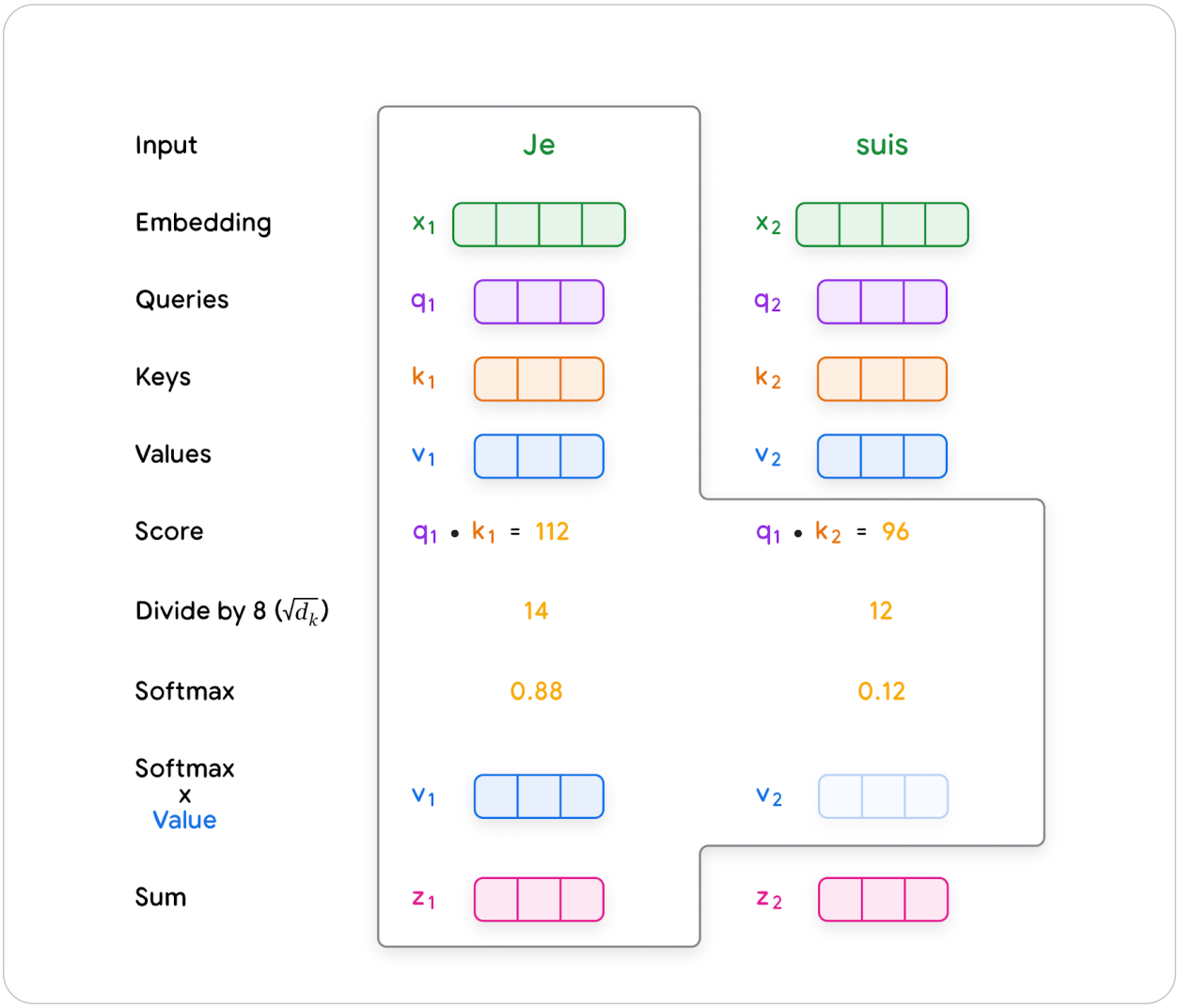


Figure 2. The process of computing self-attention in the multi-head attention module[1] (P.C:[5])

In practice, these computations are performed at the same time, by stacking the query, key and value vectors for all the tokens into Q, K and V matrices and multiplying them together as shown in Figure 3.

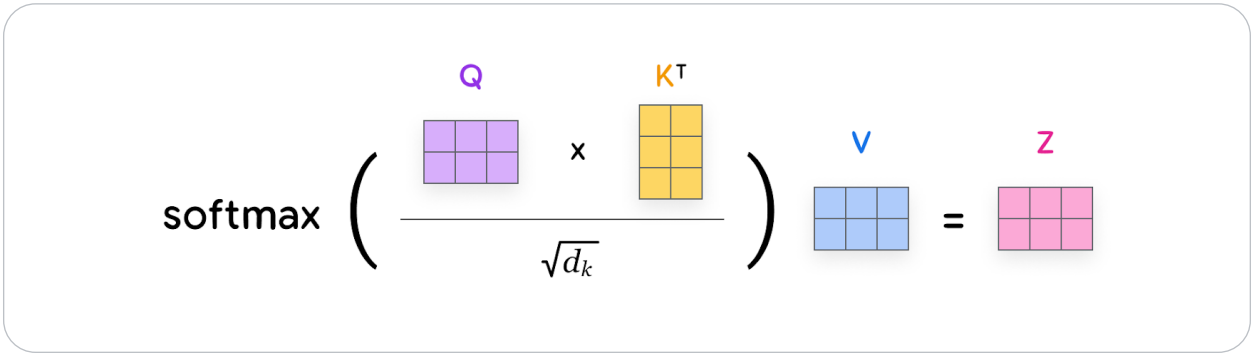


Figure 3. The basic operation of attention,[1] with Q=query, K=Keys and V=Value, Z=Attention, d_k = dimension of queries and keys (P.C:[5])

Multi-head attention: power in diversity

Multi-head attention employs multiple sets of Q, K, V weight matrices. These run in parallel, each 'head' potentially focusing on different aspects of the input relationships. The outputs from each head are concatenated and linearly transformed, giving the model a richer representation of the input sequence.

The use of multi-head attention improves the model's ability to handle complex language patterns and long-range dependencies. This is crucial for tasks that require a nuanced understanding of language structure and content, such as machine translation, text summarization, and question-answering. The mechanism enables the transformer to consider multiple interpretations and representations of the input, which enhances its performance on these tasks.

Layer normalization and residual connections

Each layer in a transformer, consisting of a multi-head attention module and a feed-forward layer, employs layer normalization and residual connections. This corresponds to the *Add and Norm* layer in Figure 1, where 'Add' corresponds to the residual connection and 'Norm' corresponds to layer normalization. Layer normalization computes the mean and variance of the activations to normalize the activations in a given layer. This is typically performed to reduce covariate shift as well as improve gradient flow to yield faster convergence during training as well as improved overall performance.

Residual connections propagate the inputs to the output of one or more layers. This has the effect of making the optimization procedure easier to learn and also helps deal with vanishing and exploding gradients.

The *Add and Norm* layer is applied to both the multi-head attention module and the feed-forward layer described in the following section.

Feedforward layer

The output of the multi-head attention module and the subsequent 'Add and Norm' layer is fed into the feedforward layer of each transformer block. This layer applies a position-wise transformation to the data, independently for each position in the sequence, which allows the incorporation of additional non-linearity and complexity into the model's representations. The feedforward layer typically consists of two linear transformations with a non-linear activation function, such as ReLU or GELU, in between. This structure adds further representational power to the model. After processing by the feedforward layer, the data undergoes another 'Add and Norm' step, which contributes to the stability and effectiveness of deep transformer models.

Encoder and decoder

The original transformer architecture relies on a combination of encoder and decoder modules. Each encoder and decoder consists of a series of layers, with each layer comprising key components: a multi-head self-attention mechanism, a position-wise feed-forward network, normalization layers, and residual connections.

The encoder's primary function is to process the input sequence into a continuous representation that holds contextual information for each token. The input sequence is first normalized, tokenized, and converted into embeddings. Positional encodings are added to these embeddings to retain sequence order information. Through self-attention mechanisms, each token in the sequence can dynamically attend to any other token, thus understanding the contextual relationships within the sequence. The output from the encoder is a series of embedding vectors Z representing the entire input sequence.

The decoder is tasked with generating an output sequence based on the context provided by the encoder's output Z. It operates in a token-by-token fashion, beginning with a start-of-sequence token. The decoder layers employ two types of attention mechanisms: *masked self-attention* and encoder-decoder *cross-attention*. Masked self-attention ensures that each position can only attend to earlier positions in the output sequence, preserving the auto-regressive property. This is crucial for preventing the decoder from having access to future tokens in the output sequence. The encoder-decoder cross-attention mechanism allows the decoder to focus on relevant parts of the input sequence, utilizing the contextual embeddings generated by the encoder. This iterative process continues until the decoder predicts an end-of-sequence token, thereby completing the output sequence generation.

Majority of recent LLMs adopted a *decoder-only* variant of transformer architecture. This approach forgoes the traditional encoder-decoder separation, focusing instead on directly generating the output sequence from the input. The input sequence undergoes a similar

process of embedding and positional encoding before being fed into the decoder. The decoder then uses masked self-attention to generate predictions for each subsequent token based on the previously generated tokens. This streamlined approach simplifies the architecture for specific tasks where encoding and decoding can be effectively merged.

Training the transformer

When talking about machine learning models, it's important to differentiate between training and inference. Training typically refers to modifying the parameters of the model, and involves loss functions and backpropagation. Inference is when model is used only for the predicted output, without updating the model weights. The model parameters are fixed during inference. Up until now we learned how transformers generate outputs during inference. Next, we focus on how to train transformers to perform one or more given tasks.

Data preparation

The first step is data preparation, which involves a few important steps itself. First, clean the data by applying techniques such as filtering, deduplication, and normalization. The next step is tokenization where the dataset is converted into tokens using techniques such as Byte-Pair Encoding[8, 9] and Unigram tokenization.[8, 10] Tokenization generates a vocabulary, which is a set of unique tokens used by the LLM. This vocabulary serves as the model's 'language' for processing and understanding text. Finally, the data is typically split into a training dataset for training the model as well as a test dataset which is used to evaluate the models performance.

Training and loss function

A typical transformer training loop consists of several parts: First, batches of input sequences are sampled from a training dataset. For each input sequence, there is a corresponding target sequence. In unsupervised pre-training, the target sequence is derived from the input sequence itself. The batch of input sequences is then fed into the transformer. The transformer generates predicted output sequences. The difference between the predicted and target sequences is measured using a loss function (often cross-entropy loss)[11]. Gradients of this loss are calculated, and an optimizer uses them to update the transformer's parameters. This process is repeated until the transformer converges to a certain level of performance or until it has been trained on a pre-specified number of tokens.

There are different approaches to formulating the training task for transformers depending on the architecture used:

- **Decoder-only** models are typically pre-trained on the language modeling task (e.g., see endnote[12, 13]). The target sequence for the decoder is simply a shifted version of the input sequence. Given a training sequence like 'the cat sat on the mat' various input/target pairs can be generated for the model. For example the input "the cat sat on" should predict "the" and subsequently the input "the cat sat on the" should predict target sequence "mat".
- **Encoder-only** models (like BERT)[14] are often pre-trained by corrupting the input sequence in some way and having the model try to reconstruct it. One such approach is masked language modeling (MLM).[14] In our example, the input sequence could be "The [MASK] sat on the mat" and the sequence target would be the original sentence.
- **Encoder-decoder** models (like the original transformer) are trained on sequence-to-sequence supervised tasks such as translation (input sequence "Le chat est assis sur le tapis" and target "The cat sat on the mat"), question-answering (where the input sequence is a question and the target sequence is the corresponding answer), and

summarization (where the input sequence is a full article and the target sequence is its corresponding summary). These models could also be trained in an unsupervised way by converting other tasks into sequence-to-sequence format. For example, when training on Wikipedia data, the input sequence might be the first part of an article, and the target sequence comprises the remainder of the article.

An additional factor to consider during training is the 'context length'. This refers to the number of previous tokens the model can 'remember' and use to predict the next token in the sequence. Longer context lengths allow the model to capture more complex relationships and dependencies within the text, potentially leading to better performance. However, longer contexts also require more computational resources and memory, which can slow down training and inference. Choosing an appropriate context length involves balancing these trade-offs based on the specific task and available resources.

The evolution of transformers

The next sections provide an overview of the various transformer architectures. These include encoder-only, encoder-decoder, as well as decoder-only transformers. We start with GPT-1 and BERT and end with Google's latest family of LLMs called Gemini.

GPT-1

GPT-1 (Generative pre-trained transformer version 1)[15] was a *decoder-only* model developed by OpenAI in 2018. It was trained on the BooksCorpus dataset (containing approximately several billion words) and is able to generate text, translate languages, write different kinds of creative content, and answer questions in an informative way. The main innovations in GPT-1 were:

- **Combining transformers and unsupervised pre-training:** Unsupervised pre-training is a process of training a language model on a large corpus of unlabeled data. Then, supervised data is used to fine-tune the model for a specific task, such as translation or sentiment classification. In prior works, most language models were trained using a supervised learning objective. This means that the model was trained on a dataset of labeled data, where each example had a corresponding label. This approach has two main limitations. First, it requires a large amount of labeled data, which can be expensive and time-consuming to collect. Second, the model can only generalize to tasks that are similar to the tasks that it was trained on. Semi-supervised sequence learning was one of the first works that showed that unsupervised pre-training followed by supervised training was superior than supervised training alone.

  Unsupervised pre-training addresses these limitations by training the model on a large corpus of unlabeled data. This data can be collected more easily and cheaply than labeled data. Additionally, the model can generalize to tasks that are different from the tasks that it was trained on. The BooksCorpus dataset is a large (5GB) corpus of unlabeled text that was used to train the GPT-1 language model. The dataset contains over 7,000 unpublished books, which provides the model with a large amount of data to learn from. Additionally, the corpus contains long stretches of contiguous text, which helps the model learn long-range dependencies. Overall, unsupervised pre-training is a powerful technique that can be used to train language models that are more accurate and generalizable than models that are trained using supervised learning alone.

- **Task-aware input transformations:** There are different kinds of tasks such as textual entailment and question-answering that require a specific structure. For example, textual entailment requires a premise and a hypothesis; question-answering requires a context document; a *question* and possible *answers*. One of the contributions of GPT-1 is converting these types of tasks which require structured inputs into an input that the language model can parse, without requiring task-specific architectures on top of the pre-trained architecture. For textual entailment, the premise p and the hypothesis h are

concatenated with a delimiter token ($) in between - [*p, $, h*]. For question answering, the context document *c* is concatenated with the question *q* and a possible answer *a* with a delimiter token in between the question and answer - [*c,q,$,a*].

GPT-1 surpassed previous models on several benchmarks, achieving excellent results. While GPT-1 was a significant breakthrough in natural language processing (NLP), it had some limitations. For example, the model was prone to generating repetitive text, especially when given prompts outside the scope of its training data. It also failed to reason over multiple turns of dialogue and could not track long-term dependencies in text. Additionally, its cohesion and fluency were limited to shorter text sequences, and longer passages would lack cohesion. Despite these limitations, GPT-1 demonstrated the power of unsupervised pre-training, which laid the foundation for larger and more powerful models based on the transformer architecture.

BERT

BERT[14] which stands for Bidirectional Encoder Representations from Transformers, distinguishes itself from traditional encoder-decoder transformer models by being an encoder-only architecture. Instead of translating or producing sequences, BERT focuses on understanding context deeply by training on a masked language model objective. In this setup, random words in a sentence are replaced with a [MASK] token, and BERT tries to predict the original word based on the surrounding context. Another innovative aspect of BERT's training regime is the next sentence prediction loss, where it learns to determine whether a given sentence logically follows a preceding one. By training on these objectives, BERT captures intricate context dependencies from both the left and right of a word, and it can discern the relationship between pairs of sentences. Such capabilities make BERT especially good at tasks that require natural language understanding, such as question-answering, sentiment analysis, and natural language inference, among others. Since this is an encoder-only model, BERT cannot generate text.

GPT-2

GPT-2,[12] the successor to GPT-1, was released in 2019 by OpenAI. The main innovation of GPT-2 was a direct scale-up, with a tenfold increase in both its parameter count and the size of its training dataset:

- **Data:** GPT-2 was trained on a large (40GB) and diverse dataset called WebText, which consists of 45 million webpages from Reddit with a Karma rating of at least three. Karma is a rating metric used on Reddit and a value of three means that all the posts were of a reasonable level of quality.
- **Parameters:** GPT-2 had 1.5 billion parameters, which was an order of magnitude larger than the previous model. More parameters increase the model's learning capacity. The authors trained four language models with 117M (the same as GPT-1), 345M, 762M, and 1.5B (GPT-2) parameters, and found that the model with the most parameters performed better on every subsequent task.

This scaling up resulted in a model that was able to generate more coherent and realistic text than GPT-1. Its ability to generate human-like responses made it a valuable tool for various natural language processing tasks, such as content creation and translation. Specifically, GPT-2 demonstrated significant improvement in capturing long-range dependencies and common sense reasoning. While it performed well in some tasks, it did not outperform state-of-the-art reading comprehension, summarization, and translation. GPT-2's most significant achievement was its ability to perform zero-shot learning on a variety of tasks. Zero-shot task transfer is the ability of a model to generalize to a new task without being trained on it, which requires the model to understand the task based on the given instruction. For example, for an English to German translation task, the model might be given an English sentence followed by the word "German" and a prompt (":"). The model would then be expected to understand that this is a translation task and generate the German translation of the English sentence. GPT-2 was able to perform tasks such as machine translation, text summarization, and reading comprehension without any explicit supervision.

The study discovered that performance on zero-shot tasks increased in a log-linear manner as the model's capacity increased. GPT-2 showed that training on a larger dataset and having more parameters improved the model's ability to understand tasks and surpass the state-of-the-art on many tasks in zero-shot settings.

GPT-3/3.5/4

GPT-3,[13] or the third iteration of the Generative Pre-trained Transformer model, represents a significant evolution from its predecessor, GPT-2, primarily in terms of scale, capabilities, and flexibility. The most noticeable difference is the sheer size of GPT-3, boasting a whopping 175 billion parameters, compared to GPT-2's largest model which had 1.5 billion parameters. This increase in model size allowed GPT-3 to store and recall an even more vast amount of information, understand nuanced instructions, and generate more coherent and contextually relevant text over longer passages.

While GPT-2 could be fine-tuned on specific tasks with additional training data, GPT-3 can understand and execute tasks with just a few examples, or sometimes even without any explicit examples—simply based on the instruction provided. This highlights GPT-3's more dynamic understanding and adaptation abilities, reducing the need for task-specific fine-tuning which was more prevalent in GPT-2.

Finally, GPT-3's large model scale and diverse training corpus have led to better generalization across a broader range of tasks. This means that out-of-the-box, without any further training, GPT-3 exhibits improved performance on diverse NLP challenges, from translation to question-answering, compared to GPT-2. It's also worth noting that the release approach differed: while OpenAI initially held back GPT-2 due to concerns about misuse, they chose to make GPT-3 available as a commercial API, reflecting both its utility and the organization's evolving stance on deployment.

Instruction tuning was then introduced with InstructGPT[17], a version of GPT-3 that was fine-tuned, using Supervised Fine-Tuning, on a dataset of human demonstrations of desired model behaviors. Outputs from this model were then ranked and it was then further fine-tuned using Reinforcement Learning from Human Feedback. This led to improved instruction following in the model. A 1.3B parameter InstructGPT model had better human evaluations than the 175B parameter GPT-3 model. It also showed improvements in truthfulness and reductions in toxicity.

GPT-3.5 models, including GPT-3.5 turbo, improve over GPT-3 as it is capable of understanding and generating code. It's been optimized for dialogue. And it's capable of receiving context windows of up to 16,385 tokens and can generate outputs of up to 4,096 tokens.

GPT-4 extends GPT-3.5 as a large multimodal model capable of processing image and text inputs and producing text outputs.[19] Specifically, accepting text or images as input and outputting text. This model has broader general knowledge and advanced reasoning capabilities. It can receive context windows of up to 128,000 tokens and has a maximum output of 4,096 tokens. GPT-4 demonstrates remarkable versatility by solving complex tasks across diverse fields like mathematics, coding, vision, medicine, law, and psychology – all without specialized instructions. Its performance often matches or even exceeds human capabilities and significantly outperforms earlier models like GPT-3.5.

LaMDA

Google's LaMDA,[20] which stands for 'Language Model for Dialogue Applications' is another contribution to the arena of large-scale language models, designed primarily to engage in open-ended conversations. Unlike traditional chatbots which operate in more constrained and predefined domains, LaMDA is engineered to handle a wide array of topics, delivering

more natural and flowing conversations. LaMDA was trained on dialogue-focused data to encourage ongoing conversational flow, not just isolated responses, ensuring users can have more extensive and explorative dialogues.

While GPT models, especially the later iterations like GPT-3, have strived to address a multitude of tasks simultaneously, from text generation to code writing, LaMDA's primary focus is on maintaining and enhancing conversational depth and breadth. GPT models shine on their ability to produce coherent long-form content and perform various tasks with minimal prompting, whereas LaMDA emphasizes the flow and progression of dialogue, striving to mimic the unpredictability and richness of human conversations.

Gopher

Gopher[22] is a 280 billion parameter language model based on the decoder-only transformer architecture, developed by DeepMind in 2021.[22] It can generate text, translate languages, write different kinds of creative content, and answer your questions in an informative way. Similar to GPT-3, Gopher focused on improving dataset quality and optimization techniques:

- **Dataset:** The researchers curated a high-quality text dataset called MassiveText, which contains over 10 terabytes of data and 2.45B documents from web pages, books, news articles, and code (GitHub). They only trained on 300B tokens, which is 12% of the dataset. Importantly, they improved the quality of the data by filtering it, such as by removing duplicate text and deduplicating similar documents. This significantly improved the model's performance on downstream tasks.
- **Optimization:** The researchers used a warmup learning rate for 1,500 steps and then decayed it using a cosine schedule. They also had an interesting rule that as they increased the model size, they decreased the learning rate and increased the number of tokens in each batch. Additionally, they found that clipping gradients to be a maximum of 1 based on the global gradient norm helped stabilize the training.

Gopher was evaluated on a variety of tasks, including mathematics, common sense, logical reasoning, general knowledge, scientific understanding, ethics, and reading comprehension. Gopher outperformed previous state-of-the-art models on 81% of the tasks. Specifically, Gopher performed well on knowledge-intensive tasks but struggled on reasoning-heavy tasks such as abstract algebra.

The authors also conducted a study on the effect of model size on different types of tasks. Figure 4 shows the results of this ablation study. Specifically, the authors found that increasing the number of parameters had a significant impact on logical reasoning and reading comprehension, but it did not improve performance as much on tasks such as general knowledge, where performance eventually almost plateaued.

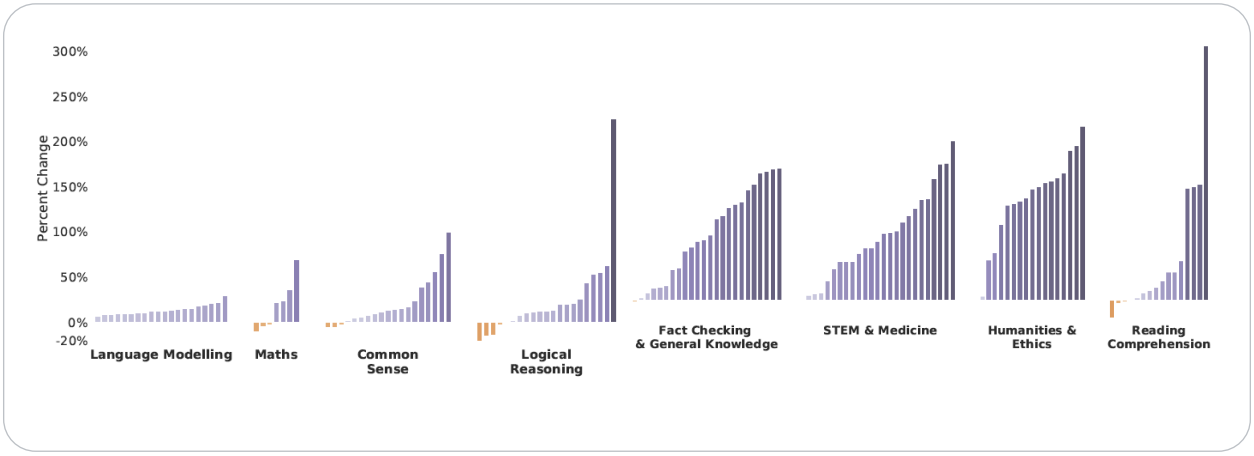


Figure 4. Ablation study[22] on the effect of model size on the performance of Gopher on different types of tasks

GLaM

GLaM (Generalist Language Model)[23] was the first sparsely-activated mixture-of-experts language model. Mixture-of-experts based models are much more computationally efficient given their parameter count. This is achieved by only activating a subset of their parameters

(i.e. experts) for each input token. GLaM consists of 1.2 trillion parameters but uses only ⅓ of the energy used to train GPT-3 and half of the FLOPs for inference while achieving better overall performance compared to GPT-3.

Chinchilla

Until 2022, LLMs were primarily scaled by increasing the model size and using datasets that are relatively small by current standards (up to 300 billion tokens for the largest models). This approach was informed by the Kaplan et al.[24] study, which examined how performance of a language model, measured by cross-entropy loss, varies with changes in computational budget, model size, and dataset size. Specifically, given a 100-fold increase in computational resources (C), Kaplan et al.[24] recommended scaling model size by approximately 28.8 times ($N_{opt} \propto C^{0.73}$), while increasing dataset size by only 3.5 times ($D_{opt} \propto C^{0.27}$).

The Chinchilla paper,[25] revisited the compute optimal scaling laws and used three different approaches to find that near equal scaling in parameters and data is optimal with increasing compute. Thus, a 100-fold increase in compute should translate into a tenfold increase in both data size and model size.

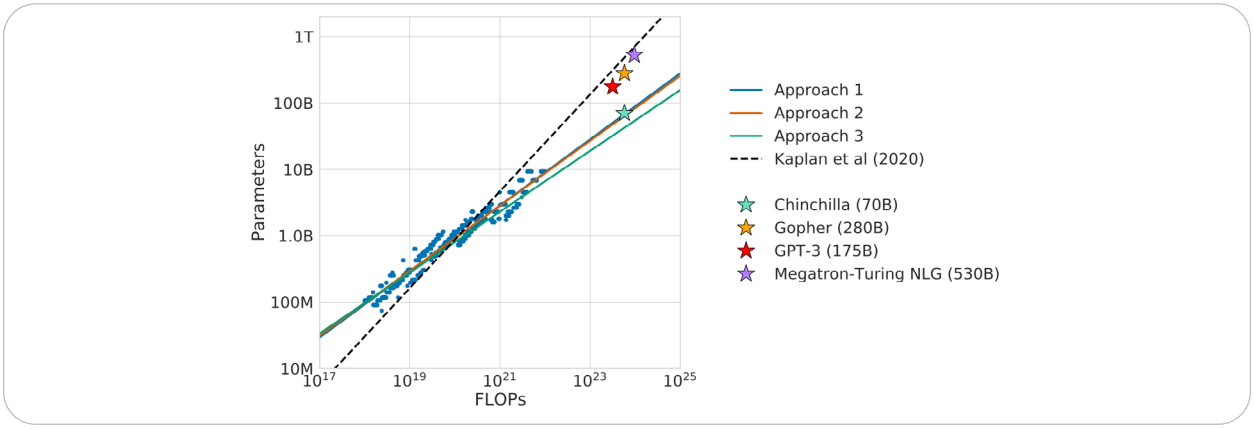


Figure 5. Overlaid predictions from three different approaches from Chinchilla paper,[25] along with projections from Kaplan et al[24]

To verify the updated scaling law, DeepMind trained a 70B parameter model (called Chinchilla) using the same compute budget as the previously trained Gopher model. Chinchilla uniformly and significantly outperformed Gopher (280B),[21] GPT-3 (175B),[13] and Megatron-Turing NLG (530B)[26] on a large range of downstream evaluation tasks. Due to being 4x smaller than Gopher, both the memory footprint and the inference cost of Chinchilla are also smaller.

The findings of Chinchilla had significant ramifications for the development of future LLMs. Focus shifted into finding ways to scale dataset size (while maintaining quality) alongside increasing parameter count. Extrapolating this trend suggests that training dataset size may soon be limited by the amount of text data available. This has led to new research by Muennighoff et al.[27] exploring scaling laws in data-constrained regimes.

PaLM

Pathways language model (PaLM)[28] is a 540-billion parameter transformer-based large language model developed by Google AI. It was trained on a massive dataset of text and code and is capable of performing a wide range of tasks, including common sense reasoning, arithmetic reasoning, joke explanation, code generation, and translation.

At the time of its release, PaLM was also able to achieve state-of-the-art performance on many language benchmarks, for example GLUE and SuperGLUE.[29]

One of the key features of PaLM is its ability to scale efficiently. This is thanks to the Pathways system, which Google developed to distribute the training of large language models across two TPU v4 Pods.

PaLM 2

PaLM 2[30] is a successor to PaLM that was announced in May 2023. Thanks to a number of architectural and training enhancements, PaLM 2 is even more capable than PaLM, with fewer total parameters. It excels at advanced reasoning tasks, including code generation, math, classification, question answering, and translation.

PaLM 2 has also been shown to be more efficient than PaLM and became the basis for a number of commercial models Google released as part of Google Cloud Generative AI.

Gemini

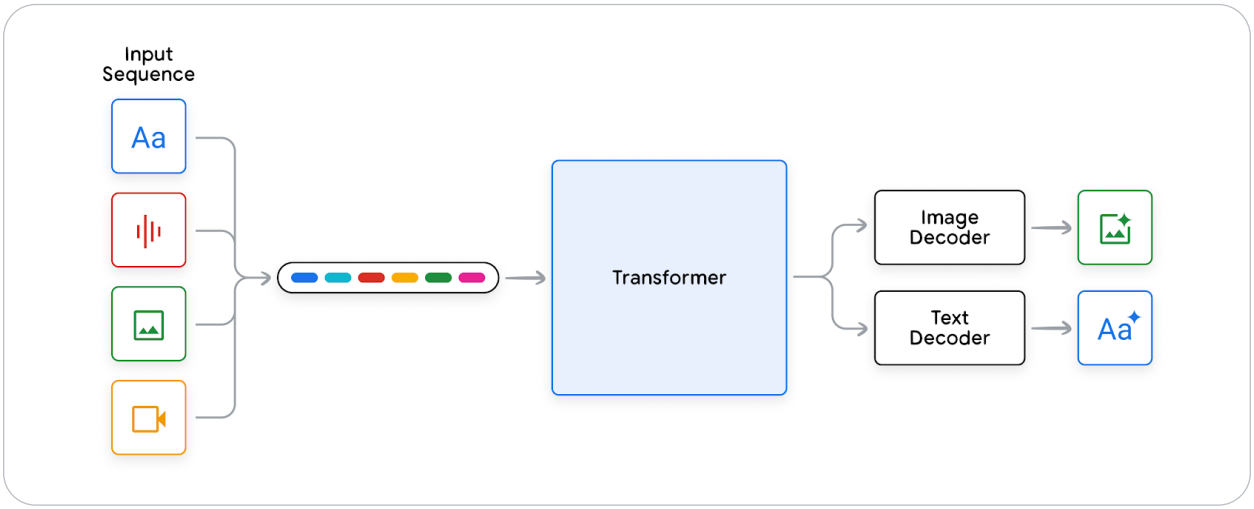


Figure 6. Gemini can receive multi-modal inputs including text, audio, images, and video data. These are all tokenized and fed into its transformer model. The transformer generates an output that can contain images and text

Gemini[31] (Figure 6) is a state-of-the-art multimodal language family of models that can take interleaved sequences of text, image, audio, and video as input. It's built on top of transformer decoders and has architectural improvements for scale as well as optimized inference on Google's Tensor Processing Units (TPUs). In its current 1.5 version, these models are trained to support contexts of different sizes, up to 2M tokens in the Gemini 1.5 Pro version on Vertex AI and employ mechanisms such as multi-query attention for efficiency. Gemini models also employ a Mixture of Experts architecture to optimize efficiency and capabilities of the models. Multimodality allows the models to process text, images and video in input, with more modalities in input and output expected in the future.

The Gemini models are trained on Google's TPUv5e and TPUv4 processors, depending on size and configuration. The pre-training data consists of web documents, books, code, and image, audio, and video data.

Larger models are trained for the compute-optimal number of tokens using the same approach as in Chinchilla paper,[25] while small models are trained on significantly more tokens than compute optimal to improve performance for a given inference budget.

The Gemini family of models is optimized for different sizes: Gemini Ultra, Gemini Pro, Gemini Nano and Flash. Gemini Ultra is used for highly complex tasks and achieves state-of-the-art results in 30 out of 32 benchmark tasks. Gemini Pro enables deployment at scale and Gemini Nano is designed for on-device applications. The Gemini Nano models leverage advancements such as distillation to produce state-of-the-art performance for small language models on tasks such as summarization and reading comprehension. As the Gemini models are natively multi-modal, it can be seen that training across multiple modalities does indeed lead to a model that is capable of achieving strong capabilities in each domain.

During the initial part of 2024, Google introduced the latest model of the Gemini family, Gemini 1.5 Pro,[32] a highly compute-efficient multimodal mixture-of-experts model. This model also dramatically increased the size of the context window to millions of tokens and is capable of recalling and reasoning over those millions of tokens, including multiple long documents and hours of video and audio. Gemini 1.5 Pro demonstrates remarkable capabilities across different domains:

- Code understanding: It can process massive codebases and answer highly specific code-related questions.
- Language learning: The model can learn new languages never observed at training time solely based on reference materials provided within its input
- Multimodal reasoning: It understands images and text, allowing it to locate a famous scene from the novel 'Les Misérables' based on a simple sketch.
- Video comprehension: It can analyze entire movies, answering detailed questions and pinpointing specific timestamps with remarkable accuracy.

Google's Gemini 1.5 Pro model excels at retrieving information from even very long documents. In their study,[32] it demonstrated 100% recall on documents up to 530,000 tokens, and over 99.7% recall on those up to 1 million tokens. Impressively, it maintains 99.2% accuracy when finding information in documents up to 10 million tokens.

Moreover, Gemini 1.5 Pro demonstrates a major leap forward in how well LLMs follow complex instructions. In a rigorous test with 406 multi-step prompts, it significantly outperformed previous Gemini models. The model accurately followed almost 90% of instructions and fully completed 66% of the complex tasks.

Gemini Flash is a new addition to the Gemini model family and the fastest Gemini model served in the API. It's optimized for high-volume, high-frequency tasks at scale, is more cost-efficient to serve and features a breakthrough long context window of 1 million tokens. Although it is a lighter weight model than 1.5 Pro, it is highly capable of multimodal reasoning across vast amounts of information and delivers impressive quality for its size.

Furthermore, recently advanced Gemma is a family of lightweight, state-of-the-art open models built from the same research and technology used to create the Gemini models.[33] The first model by Gemma boasts a large vocabulary of 256,000 words and has been trained on a massive 6 trillion token dataset. This makes it a valuable addition to the openly-available LLM collection. Additionally, the 2B parameter version is intriguing as it can run efficiently on a single GPU.

Gemma 2,[33] developed by Google AI, represents a significant advancement in the field of open large language models. Designed with a focus on efficiency, the 27-billion parameter model boasts performance comparable to much larger models like Llama 3 70B[33] on standard benchmarks. This makes Gemma 2 a powerful and accessible tool for a wide range of AI developers. Its compatibility with diverse tuning toolchains, from cloud-based solutions to popular community tools, further enhances its versatility. With its strong performance, efficient architecture, and accessible nature, Gemma 2 plays a vital role in driving innovation and democratizing AI capabilities.

Other open models

The landscape of open LLMs is rapidly evolving, with a growing number of models where both the code and pre-trained weights are publicly accessible. Below we highlight some of the known examples:

- **LLaMA 2**[34]**:** Released by Meta AI, LLaMA 2 is a family of pretrained and fine-tuned LLMs ranging from 7B to 70B parameters. It shows significant improvements over its predecessor, LLaMA 1, including a 40% larger pre-training dataset (2 trillion tokens), doubled context length (4096 tokens), and the use of grouped-query attention. The fine-tuned version, LLaMA 2-Chat, is optimized for dialogue and shows competitive performance against closed-source models of the same size.
- **LLaMA 3.2**[21]: Released by Meta AI, LLaMA 3.2 is the next generation of their open LLMs. Llama 3.2 includes multilingual text-only models (1B, 3B) and vision LLMs (11B, 90B), with quantized versions of 1B and 3B offering on average up to 56% smaller size and 2-3x speedup, ideal for on-device and edge deployments. LLaMA 3.2 utilizes grouped-query attention and a 128K token vocabulary for enhanced performance and efficiency.
- **Mixtral**[35]: Developed by Mistral AI, Mixtral 8x7B is a Sparse Mixture of Experts (SMoE) model. While its total parameter count is 47B, it utilizes only 13B active parameters per token during inference, leading to faster inference and higher throughput. This model excels in mathematics, code generation, and multilingual tasks, often outperforming LLaMA 2 70B in these domains. Mixtral also supports a 32k token context length, enabling it to handle significantly longer sequences. Its instruction-tuned version, Mixtral 8x7B-Instruct, surpasses several closed-source models on human evaluation benchmarks.
- **Qwen 1.5**[36]: This LLM series from Alibaba comes in six sizes: 0.5B, 1.8B, 4B, 7B, 14B, and 72B. Qwen 1.5 models uniformly support a context length of up to 32k tokens and show strong performance across various benchmarks. Notably, Qwen 1.5-72B outperforms LLaMA2-70B on all evaluated benchmarks, demonstrating exceptional capabilities in language understanding, reasoning, and math.
- **Yi**[37]: Created by 01.AI, the Yi model family includes 6B and 34B base models pre-trained on a massive 3.1 trillion token English and Chinese dataset. Yi emphasizes data quality through rigorous cleaning and filtering processes. The 34B model achieves performance

comparable to GPT-3.5 on many benchmarks and can be efficiently served on consumer-grade GPUs with 4-bit quantization. Yi also offers extensions like a 200k context model, a vision-language model (Yi-VL), and a depth-upscaled 9B model.

- **Grok-1**[38]: Developed by xAI, Grok-1 is a 314B parameter Mixture-of-Experts model with 25% of the weights active on a given token. It is the raw base model checkpoint from the pre-training phase and is not fine-tuned for specific tasks like dialogue. Grok-1 operates with a context length of 8k tokens.

The pace of innovation with LLMs has been rapid and shows no signs of slowing down. There have been many contributions to the field in both the academic and commercial settings. With over 20,000 papers published about LLMs in arxiv.org it is impossible to name all of the models and teams that have contributed to the development of LLMs. However, an abbreviated list of open models of interest could include EleutherAI's GPT-NeoX and GPT-J, Stanford's Alpaca, Vicuna from LMSYS, Grok from xAI, Falcon from TII, PHI from Microsoft, NVLM from Nvidia, DBRX from Databricks, Qwen from Alibaba, Yi from 01.ai, Llama from Meta mentioned above and many others. Some of notable companies developing commercial foundation LLM models include Anthropic, Cohere, Character.ai, Reka, AI21, Perplexity, xAI and many others in addition to Google and OpenAI mentioned in previous sections. It is important when using a model to confirm that the license is appropriate for your use case as many models are provided with very specific terms of use.

Comparison

In this section, we observed how transformer-based language models have evolved. They started as encoder-decoder architectures with hundreds of millions of parameters trained on hundreds of millions of tokens, and have grown to be massive decoder-only architectures with billions of parameters and trained on trillions of tokens. Table 1 shows how the important hyperparameters for all the models discussed in this whitepaper have evolved

over time. The scaling of data and parameters has not only improved the performance of LLMs on downstream tasks, but has also resulted in emergent behaviors and zero- or few-shot generalizations to new tasks. However, even the best of these LLMs still have many limitations. For example, they are not good at engaging in human-like conversations, their math skills are limited, and they might not be aligned with human ethics (e.g., they might be biased or generate toxic responses). In the next section, we learn how a lot of these issues are being addressed.

	Model						
	Attention (2017)	GPT (2018)	GPT-2 (2019)	GPT-3 (2020)	LaMDA (2021)	Gopher (2021)	Chinchilla (2022)
Optimizer	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM-W
# Parameters	213M	117M	1.5B	175B	137B	280B	70B
Vocab size	~37K	~40K	~50K	~50K	~32K	~32K	~32K
Embedding dimension	1024	768	1600	12288	8192	16384	8192
Key dimension	64	64	64	128	128	128	128
# heads (H)	16	12	25	96	128	128	64
# encoder layers	6	N/A	N/A	N/A	N/A	N/A	N/A
# decoder layers	6	12	48	96	64	80	80
Feed forward dimension	4 * 1024	4 * 768	4 * 1600	4 * 12288	8 * 8192	4 * 16384	4 * 8192
Context Token Size	N/A	512	1024	2048	N/A	2048	2048
Pre-Training tokens	~160M[A]	~1.25B[A]	~10B	~300B	~168B	~300B	~1.4T

Table 1. Important hyperparameters for transformers-based large language models

A. This number is an estimate based on the reported size of the dataset.

Fine-tuning large language models

Large language models typically undergo multiple training stages. The first stage, often referred to as pre-training, is the foundational stage where an LLM is trained on large, diverse, and unlabelled text datasets where it's tasked to predict the next token given the previous context. The goal of this stage is to leverage a large, general distribution of data and to create a model that is good at sampling from this general distribution. After language model pretraining, the resulting LLM usually demonstrates a reasonable level of language understanding and language generation skills across a variety of different tasks which are typically tested through zero-shot or few-shot prompting (augmenting the instruction with a few examples / demonstrations). Pretraining is the most expensive in terms of time (from weeks to months depending on the size of the model) and the amount of required computational resources, (GPU/TPU hours).

After training, the model can be further specialized via fine-tuning, typically called instruction-tuning or simply supervised fine-tuning (SFT). SFT involves training an LLM on a set of task-specific demonstration datasets where its performance is also measured across a set of domain-specific tasks. The following are some examples of behaviors that can be improved using fine-tuning:

- Instruction-tuning/instruction following: The LLM is provided as input an instruction to follow which might include summarizing a piece of text, writing a piece of code, or writing a poem in a certain style.[17]
- Dialogue-tuning: This is a special case of instruction tuning where the LLM is fine-tuned on conversational data in the form of questions and responses. This is often called multi-turn dialogue.[39]

- Safety tuning: This is crucial for mitigating risks associated with bias, discrimination, and toxic outputs. It involves a multi-pronged approach encompassing careful data selection, human-in-the-loop validation, and incorporating safety guardrails. Techniques like reinforcement learning with human feedback (RLHF)[40] enable the LLM to prioritize safe and ethical responses.

Fine-tuning is considerably less costly and more data efficient compared to pre-training. Numerous techniques exist to optimize the costs further which are discussed later in this whitepaper.

Supervised fine-tuning

As mentioned in the previous section, SFT is the process of improving an LLM's performance on a specific task or set of tasks by further training it on domain-specific, labeled data. The dataset is typically significantly smaller than the pre-training datasets, and is usually human-curated and of high quality.

In this setting, each data point consists of an input (prompt) and a demonstration (target response). For example, questions (prompt) and answers (target response), translations from one language (prompt) to another language (target response), a document to summarize (prompt), and the corresponding summary (target response).

It's important to note that, while fine-tuning can be used to improve the performance on particular tasks as mentioned above, it can also serve the purpose of helping the LLM improve its behavior to be safer, less toxic, more conversational, and better at following instructions.

Reinforcement learning from human feedback

Typically, after performing SFT, a second stage of fine-tuning occurs which is called *reinforcement learning from human feedback* (RLHF). This is a very powerful fine-tuning technique that enables an LLM to better align with human-preferred responses (i.e. making its responses more helpful, truthful, safer, etc.).

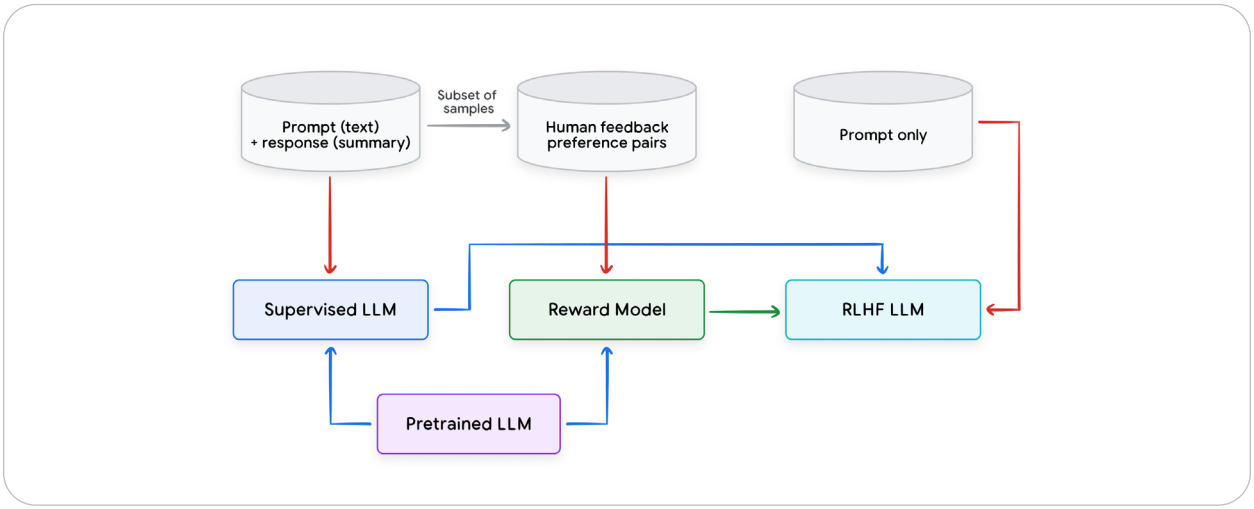


Figure 7. An example RLHF procedure

In contrast to SFT, where an LLM is only exposed to positive examples (e.g. high-quality demonstration data), RLHF makes it possible to also leverage negative outputs thus penalizing an LLM when it generates responses that exhibit undesired properties. Penalizing negative output makes it less likely to generate unhelpful or unsafe responses.

To leverage RLHF, a *reward model* (RM) typically needs to be trained with a procedure similar to that in Figure 7. An RM is usually initialized with a pretrained transformer model, often also one that is SFT. Then it is tuned on human preference data which is either single sided (with a prompt, response and a score) or composed of a prompt and a pair of responses along with

a preference label indicating which of the two responses was preferred. For example, given two summaries, A and B, of the same article, a human rater selects a preferred summary (relying on the detailed guidance). We refer to the provided preference labels as human feedback. Preferences can be in the binary form (e.g. 'good' or 'bad'), on the Likert scale[42], rank order when more than 2 candidates are evaluated, or a more detailed assessment of the summary quality. The preference signal can also incorporate many dimensions that capture various aspects that define a high quality response, e.g., as safety, helpfulness, fairness, and truthfulness.

Figure 7 shows a typical RLHF pipeline where a Reward model is initialized and finetuned on preference pairs. Once an RM has been trained, it's then used by a Reinforcement Learning (RL)[43] policy gradient algorithm, which further finetunes a previously instruction-tuned LLM to generate responses that are better aligned with human preferences.

To better scale RLHF, RL from AI Feedback (RLAIF)[44] leverages AI feedback instead of human feedback to generate preference labels. It's also possible to remove the need for training RLHF by leveraging approaches such as *direct preference optimization* (DPO).[45] Both RLHF and RLAIF can be used on Google Cloud.

Parameter Efficient Fine-Tuning

Both SFT and RLHF are still very costly in terms of compute time and accelerators required, especially when full-fine tuning entire LLMs on the orders of billions of parameters. Luckily, there are some really useful and effective techniques that can make fine-tuning significantly cheaper and faster compared to pre-training and full fine-tuning. One such family of methods is *parameter efficient fine-tuning* (PEFT) techniques.

At a high-level, PEFT approaches append a significantly smaller set of weights (e.g., on the order of thousands of parameters) that are used to 'perturb' the pre-trained LLM weights. The perturbation has the effect of fine-tuning the LLM to perform a new task or set of tasks. This has the benefit of training a significantly smaller set of weights, compared to traditional fine-tuning of the entire model.

Some common PEFT techniques include the adapter, low-rank adaptation, and soft prompting:

- *Adapter-based fine-tuning*[46] employs small modules, called adapters, to the pre-trained model. Only the adapter parameters are trained, resulting in significantly fewer parameters than traditional SFT.
- *Low-Rank Adaptation* (LoRA)[47] tackles efficiency differently. It uses two smaller matrices to approximate the original weight matrix update instead of fine-tuning the whole LLM. This technique freezes the original weights and trains these update matrices, significantly reducing resource requirements with minimum additional inference latency. Additionally, LoRA has improved variants such as QLoRA,[48] which uses quantized weights for even greater efficiency. A nice advantage of LoRA modules is that they can be plug-and-play, meaning you can train a LoRA module that specializes in one task and easily replace it with another LoRA module trained on a different task. It also makes it easier to transfer the model since assuming the receiver has the original matrix, only the update matrices need to be provided.

- *Soft prompting*[49] is a technique for conditioning frozen large language models with learnable vectors instead of hand-crafted text prompts. These vectors, called soft prompts, are optimized on the training data and can be as few as five tokens, making them parameter-efficient and enabling mixed-task inference.

For most tasks, full fine-tuning is still the most performant, followed by LoRA and Soft prompting, but the order is reversed when it comes to cost. All three approaches are more memory efficient than traditional fine-tuning and achieve comparable performance.

Python

```
# Before you start run this command:
# pip install --upgrade --user --quiet google-cloud-aiplatform
# after running pip install make sure you restart your kernel

import vertexai
from vertexai.generative_models import GenerativeModel
from vertexai.preview.tuning import sft

# TODO : Set values as per your requirements
# Project and Storage Constants
PROJECT_ID = '<project_id>'
REGION = '<region>'

vertexai.init(project=PROJECT_ID, location=REGION)

# define training & eval dataset.
TRAINING_DATASET = 'gs://cloud-samples-data/vertex-ai/model-evaluation/
peft_train_sample.jsonl'
# set base model and specify a name for the tuned model
BASE_MODEL = 'gemini-1.5-pro-002'
TUNED_MODEL_DISPLAY_NAME = 'gemini-fine-tuning-v1'

# start the fine-tuning job
sft_tuning_job = sft.train(
   source_model=BASE_MODEL,
   train_dataset=TRAINING_DATASET,
   # # Optional:
   tuned_model_display_name=TUNED_MODEL_DISPLAY_NAME,
)

# Get the tuning job info.
sft_tuning_job.to_dict()

# tuned model endpoint name
tuned_model_endpoint_name = sft_tuning_job.tuned_model_endpoint_name

# use the tuned model
tuned_genai_model = GenerativeModel(tuned_model_endpoint_name)
print(tuned_genai_model.generate_content(contents='What is a LLM?'))
```

Snippet 1. SFT fine tuning on Google cloud

Using large language models

Prompt engineering and sampling techniques have a strong influence on the performance of LLMs. Prompt engineering is the process of designing and refining the text inputs (prompts) that you feed into an LLM to achieve desired and relevant outputs. Sampling techniques determine the way in which output tokens are chosen and influence the correctness, creativity and diversity of the resulting output. We next discuss different variants of prompt engineering and sampling techniques as well as touch on some important parameters that can have a significant impact on LLM performance.

Prompt engineering

LLMs are very powerful, but they still need guidance to unleash their full potential. Prompt engineering is a critical component in guiding an LLM to yield desired outputs. This might include grounding the model to yield factual responses or unleashing the creativity of the model to tell a story or write a song. Examples of prompt engineering include providing clear instructions to the LLM, giving examples, using keywords, and formatting to emphasize important information, providing additional background details etc.

You will often hear the terms zero-shot, few-shot, and chain-of-thought prompting in the context of prompt engineering. We define these terms below:

- Few-shot prompting: This is when you provide the LLM with a task description, as well as a few (e.g. three to five) carefully chosen examples, that will help guide the LLM's response. For example, you might provide the model with the name of a few countries and their capital cities, then ask it to generate the capital for a new country that isn't in the examples.

- Zero-shot prompting: This is when you provide the LLM directly with a prompt with instructions. You usually give the LLM a task description and the LLM relies heavily on its existing knowledge to output the correct response. This requires no additional data or examples, hence the name 'Zero-shot' but can be less reliable than few-shot prompting.
- Chain-of-thought prompting: This technique aims to improve performance on complex reasoning tasks. Rather than simply asking the LLM a question, you provide a prompt that demonstrates how to solve similar problems using step-by-step reasoning. The LLM then generates its own chain of thought for the new problem, breaking it down into smaller steps and explaining its reasoning. Finally, it provides an answer based on its reasoning process.

Prompt engineering is an active area of research.

Sampling Techniques and Parameters

A variety of sampling techniques can be employed to determine how the model chooses the next token in a sequence. They are essential for controlling the quality, creativity, and diversity of the LLM's output. The following is a breakdown of different sampling techniques and their important parameters:

- *Greedy search*[50]: Selects the token with the highest probability at each step. This is the simplest option but it can lead to repetitive and predictable outputs.
- *Random sampling*:[50] Selects the next token according to the probability distribution, where each token is sampled proportionally to its predicted probability. This can produce more surprising and creative text, but also a higher chance of nonsensical output.
- *Temperature sampling*:[50] Adjusts the probability distribution by a temperature parameter. Higher temperatures promote diversity, lower temperatures favor high-probability tokens.

- *Top-K sampling:* Randomly samples from the top K most probable tokens. The value of K controls the degree of randomness.
- *Top-P sampling* (nucleus sampling):[51] Samples from a dynamic subset of tokens whose cumulative probability adds up to P. This allows the model to adapt the number of potential candidates depending on its confidence, favoring more diversity when uncertain and focusing on a smaller set of highly probable words when confident.
- *Best-of-N sampling:* Generates N separate responses and selects the one deemed best according to a predetermined metric (e.g., a reward model or a logical consistency check). This is particularly useful for short snippets or situations where logic and reasoning are key.

By combining prompt engineering with sampling techniques and correctly calibrated hyperparameters, you can greatly influence the LLM's response, making it more relevant, creative, and consistent for your specific needs.

Until now, we have seen the various types of LLM architectures, their underlying technology, as well as the approaches used to train, tune, and adapt these models for various tasks. Let's now look at some key research about how the decoding process in LLMs can be sped up considerably to generate faster responses.

Accelerating inference

The scaling laws for LLMs which were initially explored by the Kaplan et al.[24] study continue to hold today. Language models have been consistently increasing in size and this has been a direct contributor to the vast improvement in these models' quality and accuracy over the last few years. As increasing the number of parameters has improved the quality of LLMs it

has also increased the computational resources needed to run them. Numerous approaches have been used to try and improve the efficiency of LLMs for different tasks as developers are incentivized to reduce cost and latency for model users. Balancing the expense of serving a model in terms of time, money, energy is known as the cost-performance tradeoff and often needs adjusting for particular use cases.

Two of the main resources used by LLMs are memory and computation. Techniques for improving the efficiency or speed of inference focus primarily on these resources. The speed of the connection between memory and compute is also critical, but usually hardware constrained. As LLMs have grown in size 1000x from millions to billions of parameters. Additional parameters increase both the size of memory required to hold the model and computations needed to produce the model results.

With LLMs being increasingly adopted for large-scale and low-latency use cases, finding ways to optimize their inference performance has become a priority and an active research topic with significant advancements. We will explore a number of methods and a few tradeoffs for accelerating inference.

Trade offs

Many of the high yielding inference optimisation methods mandate trading off a number of factors, this can be tweaked on a case-by-case basis allowing for tailored approaches to different inference use cases and requirements. A number of the optimization methods we will discuss later fall somewhere on the spectrum of these tradeoffs.

Trading off one factor against the other (e.g. latency vs quality or cost) doesn't mean that we're completely sacrificing that factor, it just means that we're accepting what might be a marginal degradation in quality, latency or cost for the benefit of substantially improving another factor.

The Quality vs Latency/Cost Tradeoff

It is possible to improve the speed and cost of inference significantly through accepting what might be marginal to negligible drops in the model's accuracy. One example of this is using a smaller model to perform the task. Another example is quantisation where we decrease the precision of the model's parameters thereby leading to faster and less memory intensive calculations.

One important distinction when approaching this trade-off is between the theoretical possibility of a quality loss versus the practical capability of the model to perform the desired task. This is use case specific and exploring it will often lead to significant speedups without sacrificing quality in a meaningful or noticeable way. For example, if the task we want the model to perform is simple, then a smaller model or a quantised one will likely be able to perform this task well. Reduction in parametric capacity or precision does not automatically mean that the model is less capable at that specific task.

The Latency vs Cost Tradeoff

Another name for this tradeoff is the latency vs throughput tradeoff. Where throughput refers to the system's ability at handling multiple requests efficiently. Better throughput on the same hardware means that our LLM inference cost is reduced, and vice versa.

Much like traditional software systems, there are often multiple opportunities to tradeoff latency against the cost of LLM inference. This is an important tradeoff since LLM inference tends to be the slowest and most expensive component in the entire stack; balancing latency and cost intentionally is key to making sure we tailor LLM performance to the product or use case it's being used in. An example would be bulk inference use cases (e.g. offline labeling) where cost can be a more important factor than the latency of any particular request. On the other hand, an LLM chatbot product will place much higher importance on request latency.

Now that we've covered some of the important tradeoffs to consider when optimizing inference, let's examine some of the most effective inference acceleration techniques. As discussed in the tradeoffs section, some optimization techniques can have an impact on the model's output. Therefore we will split the methods into two types: output-approximating and output-preserving.

Output-approximating methods

Quantization

LLMs are fundamentally composed of multiple numerical matrices (a.k.a the model weights). During inference, matrix operations are then applied to these model weights to produce numerical outputs (a.k.a activations). Quantization is the process of decreasing the numerical precision in which weights and activations are stored, transferred and operated upon. The default representation of weights and activations is usually 32 bits floating numbers, with quantization we can drop the precision to 8 or even 4 bit integers.

Quantization has multiple performance benefits, it reduces the memory footprint of the model, allowing to fit larger models on the same hardware, it also reduces the communication overhead of weights and activations within one chip and across chips in a distributed inference setup- therefore speeding up inference as communication is a major contributor to latency. In addition, decreasing the precision of weights/activations can enable faster arithmetic operations on these models as some accelerator hardware (e.g. TPUs/GPUs) natively supports faster matrix multiplication operations for some lower precision representations.

Quantization's impact on quality can be very mild to non-existent depending on the use case and model. Further, in cases where quantisation might introduce a quality regression, that regression can be small compared to the performance gain, therefore allowing for an effective Quality vs Latency/Cost Tradeoff. For example, Benoit Jacob et al.[55] reported a 2X speed-up for a 2% drop in accuracy for the FaceDetection task on MobileNet SSD.

Quantization can be either applied as an inference-only operation, or it can be incorporated into the training (referred to as Quantisation Aware Training QAT). QAT is generally considered to be a more resilient approach as the model is able to recover some of the quantisation-related quality losses during training. To make sure we get the best cost/quality tradeoff, we tweak the quantization strategy (e.g. select different precisions for weights vs activations) and the granularity in which we apply quantisation to Tensors (e.g. channel or group-wise[58]).

Distillation

Using a smaller model to perform a task is one of the most efficient inference optimization techniques, however, smaller models can demonstrate significant regressions on quality compared to their larger counterparts.

Distillation is a set of training techniques that targets improving the quality of a smaller model (the student) using a larger model (the teacher). This method can be effective because larger models outperform smaller ones even if both are trained on the same data, mainly due to parametric capacity and training dynamics. The gap in performance continues as the training dataset grows as illustrated by Figure 8.

It is worth noticing that even at low volumes of training data, large models can already demonstrate better performance than the correspondingly trained smaller models, this fact leads us to the first variant of distillation which is referred to as data distillation or model compression.[56] We use a large model which was trained on the data we have to generate more synthetic data to train the smaller student model, the increase in data volume will help move the the student further along the quality line compared to only training on the original data. Synthetic data needs to be approached carefully as it needs to be of high quality and can lead to negative effects otherwise.

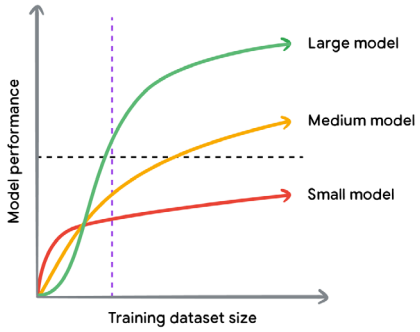


Figure 8. An illustration of the performance of models of various sizes as a function of the training dataset's size

Other distillation techniques attempt to bring the student model closer to the teacher on a more granular level than just synthetic data generation. One prominent technique is knowledge distillation[57], in this approach we attempt to align the output token distribution of the student model to that of the teacher's, this can be much more sample efficient than data distillation. On-policy distillation[59] is another technique that leverages feedback from the teacher model on each sequence generated by the student in a reinforcement learning setup.

Output-preserving methods

These methods are guaranteed to be quality neutral, they cause no changes to the model output which often makes them obvious first steps to optimize inference before facing the more nuanced tradeoffs of the approximating methods

Flash Attention

Scaled Dot-product Attention, which is the predominant attention mechanism in the transformer architecture, is a quadratic operation on the input length. Optimizing the self-attention calculation can bring significant latency and cost wins.

Flash Attention, introduced in by Tri Dao et al.[62], optimizes the attention calculation by making the attention algorithm IO Aware, particularly trying to minimize the amount of data we move between the slow HBM (high bandwidth memory) to the faster memory tier (SRAM/VMEM) in TPUs and GPUs. When calculating attention, the order of operations is changed and multiple layers are fused so we can utilize the faster memory tiers as efficiently as possible.

Flash Attention is an exact algorithm, it maintains the numerical output of the attention computation and can yield significant latency benefits due to reducing the IO overhead, Tri Dao et al.[62] showed 2-4X latency improvements in the attention computation.

Prefix Caching

One of the most compute intensive, and thus slowest, operations in LLM inference is calculating the attention key and value scores (a.k.a KV) for the input we're passing to the LLM, this operation is often referred to as prefill. The final output of prefill is what is termed KV Cache which includes the attention key and value scores for each layer of the transformer for the entire input. This cache is vital during the decoding phase which produces the output tokens, the KV cache allows us to avoid recalculating attention scores for the input on each autoregressive decode step.

Prefix Caching refers to the process of caching the KV Cache itself between subsequent inference requests in order to reduce the latency and cost of the prefill operation. The way the self-attention mechanism works makes reusing KV caches possible because tokens will only pay attention to tokens that came before them in the sequence. If there's new input being appended to input that the model has seen before, then we can potentially avoid recalculating the prefill for the older input.

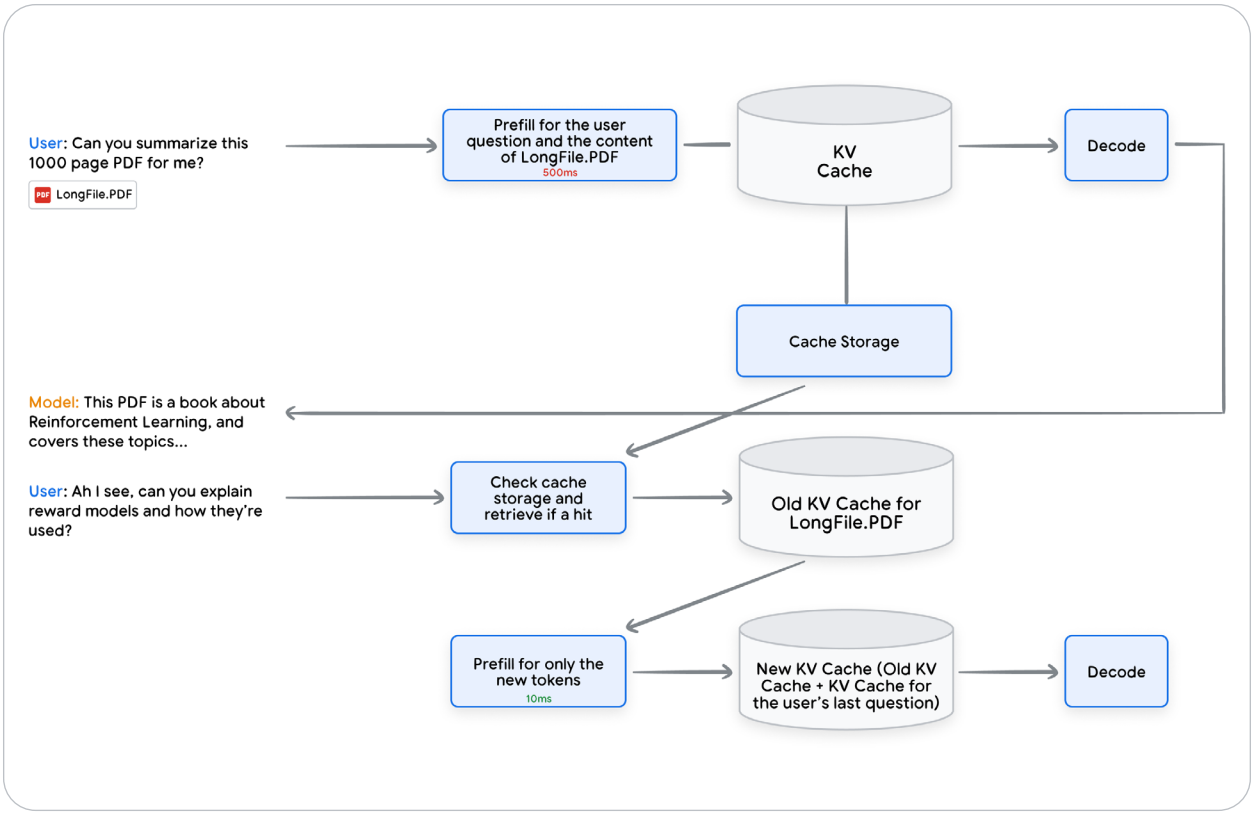


Figure 9. An illustration of Prefix Caching in a chat scenario

Figure 9 illustrates how prefix caching works in a multi-turn scenario with a document upload. On the first user turn, the prefill operation has to process the entire document therefor taking 500ms, the resulting KV cache is then stored so that on the second user turn, we can retrieve the cache directly from storage and avoid recomputing it for the long doc, therefore saving substantial amounts of compute and latency.

Prefix caches can be stored either in memory or on disk and fetched on-demand. One important consideration is making sure that the input structure/schema remains prefix-caching friendly, we should avoid changing the prefix in subsequent requests as that will invalidate the cache for all the tokens that follow For example, putting a fresh timestamp at the very beginning of each request will invalidate the cache completely as every subsequent request will have a new prefix.

Many LLM use cases lend themselves naturally to prefix caching. For example, LLM Chatbots where users will have a multi-turn conversation that can span 10s of 1000s of tokens and we can avoid recalculating the KV cache for the previous parts of the conversation. Large document/code uploads is another use case where the artifact the user uploads will remain unchanged from one request to the next. All that's changing are the questions the user is asking, so caching the KV cache for the document (especially for larger artifacts) can result in significant latency and cost savings.

Prefix caching is available as a service called Context Caching on Google AI studio[52] and Vertex AI on Google Cloud[53].

Speculative Decoding

The first phase of LLM inference, known as prefill, is compute bound due large matrix operations on many tokens occurring in parallel. The second phase, known as decode, is generally memory bound as tokens are auto-regressively decoded one at a time.

It is not easy to naively use additional parallel compute capacity to speed up decode given the need to wait for the current token to be produced before we can calculate what the next token should be (as per the self-attention mechanism), the decode process is inherently serial.

Speculative decoding (Leviathan at al.[63]) aims to overcome this limitation in decode by finding a way to utilize the spare compute capacity to make each decode step faster. The main idea is to use a much smaller secondary model (often referred to as the drafter) to run ahead of the main model and predict more tokens. (e.g. 4 tokens ahead). This will happen very quickly as the drafter is much faster and smaller than the main model. We then use the main model to verify the hypotheses of the drafter in parallel for each of the 4 steps (i.e. the first token, the first two tokens, the first 3 tokens and finally all 4 tokens), and we then select the accepted hypothesis with the maximum number of tokens. For example:

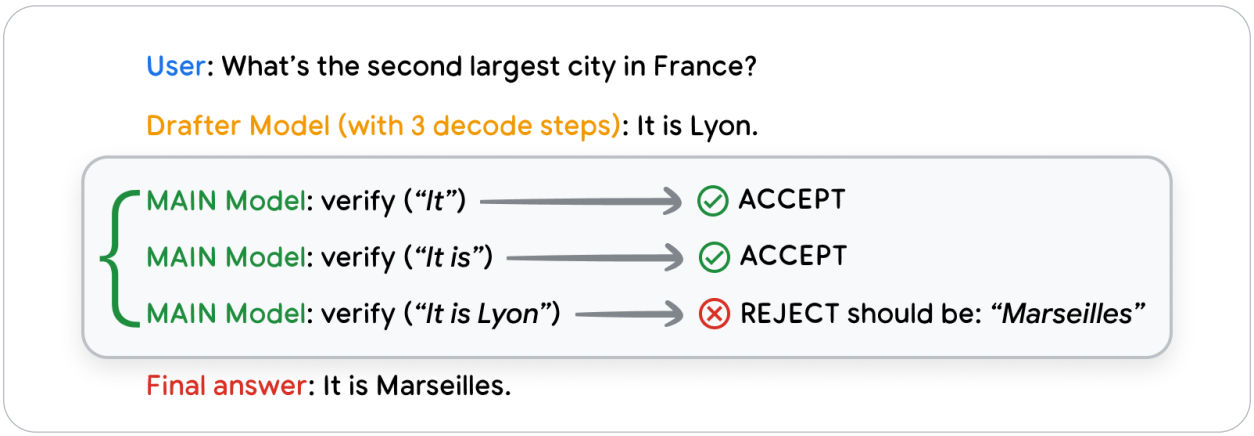


Figure 10. An illustration of speculative decoding over 3 tokens

Note that the 3 main model steps run in parallel. And because we are not compute bound in decode, we can use the spare capacity to get much better decode latencies. In the example above, let's say a single main model step needs 10ms, while the drafter needs 1ms. Without speculative decoding, we need 3 * 10ms = 30ms to produce the response, with speculative

decoding, there's only one main model step on the critical path due to parallelization, so we need 3 * 1ms + 10ms = 13ms. A significant latency improvement. This technique is completely quality neutral, the main model will reject any tokens that it wouldn't have predicted itself in the first place, so the only thing speculative decoding does is run ahead and present hypotheses that the main model can accept or reject in parallel.

One important condition for speculative decoding to work effectively is that the drafter model has good levels of alignment with the main model, otherwise we won't be able to accept any of the tokens. So investing in the training quality of the drafter model is worthwhile to get better latencies.

Now that we have seen some methods to make LLM generate their responses faster, let's look at some examples of how these models can be applied to various tasks to get an idea how to use them.

Batching and Parallelization

Most of the optimization techniques we've discussed so far are specific to Machine Learning and Transformer architecture in particular. However, much like any software system, there are opportunities to improve throughput and latency through a combination of 1) batching less compute-intensive operations (i.e. we can run multiple requests on the same hardware simultaneously to better utilize the spare compute) and 2) parallelizing the more compute-intensive parts of the computations (i.e. we can divide the computation and split it amongst more hardware instances to get more compute capacity and therefore better latencies

Batching in LLMs is most useful on the decode side - as we explained in the Speculative Decoding section, decode is not compute-bound and therefore there's an opportunity to batch more requests. We need to be careful that we batch computations in a way that

enables utilization of the spare capacity which is possible to do on accelerators (e.g. TPUs and GPUs). We also need to make sure we remain within the memory limits, as decode is a memory intensive operations, batching more requests will put more pressure on the free memory available. Batching has become an important component in most high-throughput LLM inference setups.

Parallelization is also a widely used technique given the variety of opportunities in transformers for horizontal scaling across more hardware instances. There are multiple parallelism techniques across the model input (Sequence parallelism) the model layers (Pipeline parallelism), and within a single layer (Tensor parallelism). One of the most important considerations for parallelism is the cost of communication and synchronization between the different shards that we distribute to other machines. Communication is a significant overhead and can erode the benefits of adding more computational capacity if we're not careful about which parallelization strategy to use. On the other hand, selecting the right strategy to balance the need for additional compute and the communication cost can yield significant latency wins.

Now that we have seen some methods to make LLM generate their responses faster, let's look at some examples of how these models can be applied to various tasks to get an idea how to use them.

Applications

Large language models are revolutionizing the way we interact with and process information. With their unprecedented ability to understand context and generate content, they're transforming numerous applications in the worlds of text, code, images, audio and video. Here we collected a few examples of application areas, but the reader should keep in mind that this is not a comprehensive list and that many new ideas are emerging continuously

about how to best utilize the capabilities of these new tools. For more information about optimally building and deploying functioning applications based on the following mentioned use cases, refer to the subsequent whitepapers.

It is also very simple to generate text-based responses for your use case using either the Google Cloud Vertex AI SDK or the Developer focused AI studio. Snippet 3 shows code examples from these SDKs to generate responses to text prompts using the Gemini model. Note that the multimodal aspects of Gemini are covered in their respective dedicated whitepapers.

Python

```
# Before you start run this command:
# pip install --upgrade --user --quiet google-cloud-aiplatform
# after running pip install make sure you restart your kernel

import vertexai
from vertexai.language_models import TextGenerationModel
from vertexai.preview.generative_models import GenerationConfig,GenerativeModel

# Set values as per your requirements
PROJECT_ID = '<project_id>' # set to your project_id
vertexai.init(project=PROJECT_ID, location='us-central1')

PROMPT= 'What is a LLM?' # set your prompt here
model = GenerativeModel('gemini-1.5-pro-002')

# call the Gemini API
response = model.generate_content(
   PROMPT)

print(response.text)

# google AI Studio SDK
import google.generativeai as genai
import os

# update with your API key
genai.configure(api_key=os.environ["GOOGLE_API_KEY"])

# choose the model
model = genai.GenerativeModel('gemini-pro')

response = model.generate_content('What is a LLM?') # set your prompt here

print(response.text)
```

Snippet 3. Using Vertex AI and Google AI studio SDKs for unimodal text gene

Code and mathematics

Generative models can comprehend and generate code and algorithms to supercharge developers by assisting them across many application areas. Some of the popular use cases for code include:

- **Code generation:** LLMs can be prompted in natural language to generate code in a specific programming language to perform certain operations. The output can be used as a draft.
- **Code completion:** LLMS can proactively suggest useful code as the user types it. This can save developers time and improve code quality.
- **Code refactoring and debugging:** LLMs can help reduce technical debt by refactoring and debugging code to improve quality, efficiency and correctness.
- **Code translation:** LLMs can significantly help developer time and effort by helping to convert code from one programming language to another. For example, an LLM might convert Python code to Java.
- **Test case generation:** LLMs can be prompted to generate unit tests for a provided codebase which saves considerable time and reduces errors.
- **Code documentation and understanding:** LLMs can be used in a conversational manner to engage in a natural language chat to help you understand a codebase. They can also generate appropriate comments, understand copyright status, and create release notes.

Recently, a number of exciting advancements have been made in the space of competitive coding and mathematics. *AlphaCode 2*,[64] combines Gemini's reasoning capabilities with search and the use of tools to solve competitive coding problems. It receives as input a description of a problem to solve, and outputs a code solution that solves the problem. It

now ranks among the top 15% competitive coders on the popular Codeforces competitive coding platform. *FunSearch*[65] uses an evolutionary procedure which is based on pairing a pre-trained LLM with a systematic evaluator. It solved the cap set problem[66], an open problem in mathematics, and also discovered more efficient bin-packing algorithms which are used in many applications such as making data centers more efficient. Another recent approach called *AlphaGeometry* tackles the problem of finding proofs for complex geometric theorems. It comprises a neuro-symbolic system made up of a neural language model and a symbolic deduction engine. AlphaGeometry managed to solve 25 out of 30 Olympiad geometry problems, where the average human gold medalist scores on average 25.9. [67]

Machine translation

LLMs are capable of generating fluid, high-quality and contextually accurate translations. This is possible due to the LLM's deep understanding of linguistic nuances, idioms, and context. The following are some possible real world use cases:

- **Instant messaging apps:** In messaging platforms, LLMs can provide on-the-fly translations that feel natural. Unlike previous algorithms that might translate word-for-word, LLMs understand slang, colloquialisms, and regional differences, enhancing cross-language communication.
- **E-commerce:** On global platforms like AliExpress, product descriptions are automatically translated. LLMs help with ensuring cultural nuances and idiomatic expressions in product details are appropriately translated, leading to fewer misunderstandings.
- **Travel apps:** In apps like Google Translate, travelers get real-time spoken translations. With LLMs, the translated conversations are smoother, making interactions in foreign countries more effortless.

Text summarization

Text summarization is a core capability of many of the LLMs mentioned in this whitepaper. There are a number of natural potential use cases which include:

- **News aggregators:** LLMs could craft summaries that capture not only the main events but also the sentiment and tone of the article, providing readers with a more holistic understanding.
- **Research databases:** LLMs could help researchers generate abstracts that encapsulate the core findings and implications of scientific papers.
- **Chat management:** In platforms like Google Chat, LLM-based systems could generate thread summaries that capture the urgency and tone, aiding users in prioritizing their responses.

Question-answering

The older generation of QA systems often worked by keyword matching, frequently missing out on the contextual depth of user queries. LLMs, however, dive deep into context. They can infer user intent, traverse vast information banks, and provide answers that are contextually rich and precise. Some of the examples where this could be used include:

- **Virtual assistants:** LLMs can offer detailed explanations of a weather forecast considering the user's location, time of year, and recent weather trends.
- **Customer support:** In business platforms, LLM-based bots could provide answers that take into account the user's purchase history, past queries, and potential issues, offering personalized assistance.

- **Academic platforms:** On academic platforms like Wolfram Alpha, LLMs could cater to user queries by understanding the depth and context of academic questions, offering answers that suit everyone from a high school student to a postgraduate researcher.

The quality of the generated answers, as well as the corresponding citations and sources can be significantly improved by using advanced search systems (such as those based on Retrieval Augmented Generation (RAG) architectures) to expand the prompt with relevant information, as well as post-hoc grounding after the response has been generated. Clear instructions, roles of what should and should not be used to answer the question, and advanced prompt engineering approaches (such as chain of thought and search/RAG architectures), combined with a lower temperature value amongst other things can also help greatly.

Chatbots

Earlier chatbots followed scripted pathways, leading to 'mechanical' conversations. LLMs transform this space by offering dynamic, human-like interactions. They can analyze sentiment, context, and even humor, making digital conversations feel more authentic. Some examples of where this can be used include:

- **Customer service:** A chatbot on retail platforms like Zara could not only answer product-related queries but also offer fashion advice based on current trends.
- **Entertainment:** On Media LLM-driven chatbots could engage with users dynamically, reacting to live events in the stream and moderating chats with contextual understanding.

Content generation

Text generation isn't new, but what LLMs bring to the table is the unprecedented ability to generate human-like text that's contextually relevant and rich in detail. Earlier models would often lose context or coherence over longer passages. LLMs, with their vast knowledge and nuanced understanding, can craft text spanning various styles, tones, and complexities, mixing factuality with creativity (depending on the context) effectively bridging the gap between machine-generated and human-written content. The following are some real-world examples:

- **Content creation:** Platforms could utilize LLMs to help marketers develop advertisements. Instead of generic content, the LLMs could generate creative, targeted, and audience-specific messages.
- **Scriptwriting:** LLMs could potentially assist with producing scripts for movies or TV shows. Writers could input themes or plot points, and the model can suggest dialogues or scene descriptions, enhancing the creative process.

Text generation is a wide task encompassing a variety of use cases that might range from those where correctness of the generated output is more or less important than its creativity/diversity of the language. The sampling methods and parameters like temperature should be tuned accordingly. For more information, see the prompt engineering and architecting for LLM applications whitepapers.

Natural language inference

Natural language inference (NLI) is the task of determining whether a given textual hypothesis can be logically inferred from a textual premise.

Traditional models struggled with nuanced relationships or those that require a deeper understanding of context. LLMs, with their intricate grasp of semantics and context, excel at tasks like these, bringing accuracy levels close to human performance. The following are some real-world examples:

- **Sentiment analysis:** Businesses could utilize LLMs to infer customer sentiment from product reviews. Instead of just basic positive or negative tags, they could extract nuanced emotions like ‘satisfaction,’ ‘disappointment,’ or ‘elation’.
- **Legal document review:** Law firms could employ LLMs to infer implications and intentions in contracts, ensuring there are no contradictions or potentially problematic clauses.
- **Medical diagnoses:** By analyzing patient descriptions and histories, LLMs could assist doctors in inferring potential diagnoses or health risks, ensuring early intervention.

The whitepapers on domain specific LLMs, prompt engineering, and architecting for LLM applications give further insight into these use cases.

Text classification

Text classification involves categorizing text into predefined groups. While traditional algorithms were efficient, they often struggled with ambiguous or overlapping categories. LLMs, given their deep understanding of context, can classify text with higher precision, even when faced with subtle distinctions. Some examples of this include:

- **Spam detection:** Email services could utilize LLMs to classify emails as spam or legitimate. Instead of just keyword-based detection, the models understand the context and intent, potentially reducing false positives.

- **News categorization:** News platforms could employ LLMs to categorize articles into topics like ‘technology,’ ‘politics,’ or ‘sports,’ even when articles blur the boundaries between categories.
- **Customer feedback sorting:** Businesses could analyze customer feedback through LLMs to categorize them into areas like ‘product design,’ ‘customer service,’ or ‘pricing,’ ensuring targeted responses.
- **Evaluating LLMs as autorater:** LLMs could be used to rate, compare and rank the generated outputs of other LLMs as well.

Text analysis

LLMs excel at deep text analysis – extracting patterns, understanding themes, and gleaning insights from vast textual datasets. Where traditional tools would scratch the surface, LLMs delve deep, offering rich and actionable insights. Some potential real-world examples are:

- Market research: Companies could leverage LLMs to analyze consumer conversations on social media, extracting trends, preferences, and emerging needs.
- Literary analysis: Academics could employ LLMs to understand themes, motifs, and character developments in literary works, offering fresh perspectives on classic and contemporary literature.

Multimodal applications

Multimodal LLMs, capable of processing and generating text, images, audio, and video, have opened up a new frontier in AI, offering a range of exciting and innovative applications across various sectors. The following are some examples:

Creative content generation:

- Storytelling: An AI system could watch an image or video and spin a captivating narrative, integrating details from the visual with its knowledge base.
- Advertising and marketing: Generating targeted and emotionally resonant advertisements based on product photos or videos.

Education and accessibility:

- Personalized learning: Tailoring educational materials to individual learning styles by combining text with interactive visual and audio elements.
- Assistive technology: Multimodal LLMs could power tools that describe images, videos, and audio for visually or hearing-impaired individuals.

Business and industry:

- Document understanding and summarization: Automatically extracting key information from complex documents, combining text and visuals like invoices and contracts.
- Customer service: Multimodal chatbots can understand and respond to customer queries combining text and images, offering a richer and more personalized experience. Science and research:

- Medical diagnosis: Analyzing medical scans and reports together, identifying potential issues and providing insights for doctors.
- Bioinformatics and drug discovery: Integrating knowledge from diverse data sources like medical images, protein structures, and research papers to accelerate research.

These examples are just the tip of the iceberg. As research progresses, the applications of multimodal LLMs are only expected to grow, transforming our daily lives in diverse and profound ways. Multimodal LLMs also benefit greatly from the existing methodologies of Unimodal LLMs (i.e., text based LLMs).

LLMs, thanks to their ability to understand and process language, are reshaping how we interact with, generate, and analyze text across diverse sectors. As they continue to evolve, their applications will only grow, boosting the ability for machines and humans to have rich natural language interactions.

Summary

In this whitepaper we have discussed the basics of transformers, upon which all modern-day LLMs are based. We detailed the evolution of the various LLM model architectures and their components. We've also seen the various methodologies you can use to train and fine-tune models efficiently and effectively. We briefly discussed prompt engineering and sampling techniques that greatly influence the output of an LLM, and also touched on possible applications of this technology. There are a number of key takeaways to keep in mind:

- The transformer architecture is the basis for all modern-day LLMs. Across the various architectures mentioned in this whitepaper we see that it's important not only to add more parameters to the model, but the composition of the dataset is equally important.
- The order and strategies used for fine-tuning is important and may include multiple steps such as Instruction Tuning, Safety Tuning, etc. Supervised Fine Tuning (SFT) is important in capturing the essence of a task. RLHF, and potentially RLAIF, can be used to shift the distribution from the pretraining distribution to a more desired one through the power of the reward function, that can reward desirable behaviors and penalize undesirable ones.
- Making inference from neural models efficient is an important problem and an active field of research. Many methods exist to reduce serving costs and latency with minimal impact to model performance, and some exact acceleration methods guarantee identical model outputs.
- Large language models can be used for a variety of tasks including summarization, translation, question answering, chat, code generation, and many more. You can create your own tasks using the Vertex and Makersuite text generation services which leverage Google's latest language models. After the model has been trained and tuned, it is important to experiment with engineering prompts. You should use the technique most appropriate for the task-at-hand because LLMs can be sensitive to prompts k. Furthermore, it is also possible to enhance task specific performance or creativity and diversity by tweaking the parameters corresponding to sampling techniques such as Top-K, Top-P, and Max decoding steps to find the optimum mix of correctness, diversity, and creativity required for the task at hand.

Endnotes

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I., 2017, Attention is all you need. *Advances in Neural Information Processing Systems*, 30.
2. Wikipedia, 2024, Word n-gram language model. Available at: https://en.wikipedia.org/wiki/Word_n-gram_language_model.
3. Sutskever, I., Vinyals, O., & Le, Q. V., 2014, Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems,* 27.
4. Gu, A., Goel, K., & Ré, C., 2021, Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*.
5. Jalammar, J. (n.d.). The illustrated transformer. Available at: https://jalammar.github.io/illustrated-transformer/.
6. Ba, J. L., Kiros, J. R., & Hinton, G. E., 2016, Layer normalization. *arXiv preprint arXiv:1607.06450*.
7. He, K., Zhang, X., Ren, S., & Sun, J., 2016, Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.
8. HuggingFace., 2024, Byte Pair Encoding. Available at: https://huggingface.co/learn/nlp-course/chapter6/5?fw=pt.
9. Kudo, T., & Richardson, J., 2018, Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.
10. HuggingFace, 2024, Unigram tokenization. Available at: https://huggingface.co/learn/nlp-course/chapter6/7?fw=pt.
11. Goodfellow et. al., 2016, Deep Learning. MIT Press. Available at: http://www.deeplearningbook.org.
12. Radford, Alec et al., 2019, Language models are unsupervised multitask learners.
13. Brown, Tom, et al., 2020, Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877-1901.
14. Devlin, Jacob, et al., 2018, BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

15. Radford, A., & Narasimhan, K., 2018, Improving language understanding by generative pre-training.

16. Dai, A., & Le, Q., 2015, Semi-supervised sequence learning. *Advances in Neural Information Processing Systems*.

17. Ouyang, Long, et al., 2022, Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems,* 35, 27730-27744.-27744.

18. OpenAI., 2023, GPT-3.5. Available at: https://platform.openai.com/docs/models/gpt-3-5.

19. OpenAI., 2023, GPT-4 Technical Report. Available at: https://arxiv.org/abs/2303.08774.

20. Thoppilan, Romal, et al., 2022, Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

21. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. Available at: https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.

22. Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., ... & Irving, G., 2021, Scaling language models: Methods, analysis & insights from training Gopher. Available at: https://arxiv.org/pdf/2112.11446.pdf.

23. Du, N., He, H., Dai, Z., Mccarthy, J., Patwary, M. A., & Zhou, L., 2022, GLAM: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning* (pp. 2790-2800). PMLR.

24. Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D., 2020, Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

25. Hoffmann, Jordan, et al., 2022, Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

26. Shoeybi, Mohammad, et al., 2019, Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

27. Muennighoff, N. et al., 2023, Scaling data-constrained language models. *arXiv preprint arXiv:2305.16264*.

28. Chowdhery, Aakanksha, et al., 2023, Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240), 1-113.

29. Wang, Alex, et al.,2019, SuperGLUE: A stickier benchmark for general-purpose language understanding systems. *Advances in Neural Information Processing Systems*, 32.

30. Anil, Rohan, et al., 2023, Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

31. DeepMind, 2023, Gemini: A family of highly capable multimodal models. Available at: https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf.

32. DeepMind, 2024, Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. Available at: https://storage.googleapis.com/deepmind-media/gemini/gemini_v1_5_report.pdf.

33. Google Developers, 2024, Introducing PaLi-Gemma, Gemma 2, and an upgraded responsible AI toolkit. Available at: https://developers.googleblog.com/en/gemma-family-and-toolkit-expansion-io-2024/.

34. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., ... & Jegou, H., 2023, Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

35. Jiang, A. Q., 2024, Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

36. Qwen, 2024, Introducing Qwen1.5. Available at: https://qwenlm.github.io/blog/qwen1.5/.

37. Young, A., 2024, Yi: Open foundation models by 01.AI. *arXiv preprint arXiv:2403.04652*.

38. Grok-1, 2024, Available at: https://github.com/xai-org/grok-1.

39. Duan, Haodong, et al., 2023, BotChat: Evaluating LLMs' capabilities of having multi-turn dialogues. *arXiv preprint arXiv:2310.13650*.

40. Google Cloud, 2024, *Tune text models with reinforcement learning from human feedback*. Available at: https://cloud.google.com/vertex-ai/generative-ai/docs/models/tune-text-models-rlhf.

41. Bai, Yuntao, et al., 2022, Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*.

42. Wikipedia, 2024, Likert scale. Available at: https://en.wikipedia.org/wiki/Likert_scale.

43. Sutton, R. S., & Barto, A. G., 2018, *Reinforcement learning: An introduction*. MIT Press.

44. Bai, Yuntao, et al, 2022, Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*.

45. Rafailov, Rafael, et al., 2023, Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*.

46. Houlsby, Neil, et al., 2019, Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning* (pp. 2790-2799). PMLR.

47. Hu, Edward J., et al., 2021, LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

48. Dettmers, Tim, et al., 2023, QLoRA: Efficient finetuning of quantized LLMs. *arXiv preprint arXiv:2305.14314*.

49. Lester, B., Al-Rfou, R., & Constant, N., 2021, The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*.

50. HuggingFace., 2020, How to generate text? Available at: https://huggingface.co/blog/how-to-generate.

51. Google AI Studio Context caching. Available at: https://ai.google.dev/gemini-api/docs/caching?lang=python.

52. Vertex AI Context caching overview. Available at: https://cloud.google.com/vertex-ai/generative-ai/docs/context-cache/context-cache-overview.

53. Gu, A., Goel, K., & Ré, C., 2021, Efficiently modeling long sequences with structured state spaces. Available at: https://arxiv.org/abs/2111.00396.

54. Hubara et al., 2016, Quantized neural networks: Training neural networks with low precision weights and activations. Available at: https://arxiv.org/abs/1609.07061.

55. Benoit Jacob et al., 2017, Quantization and training of neural networks for efficient integer-arithmetic-only inference. Available at: https://arxiv.org/abs/1712.05877.

56. Bucila, C., Caruana, R., & Niculescu-Mizil, A., 2006, Model compression. *Knowledge Discovery and Data Mining*. Available at: https://www.cs.cornell.edu/~caruana/compression.kdd06.pdf.

57. Hinton, G., Vinyals, O., & Dean, J., 2015, Distilling the knowledge in a neural network. Available at: https://arxiv.org/abs/1503.02531.

58. Zhang, L., Fei, W., Wu w., He Y., Lou Z., Zhou H., 2023, Dual Grained Quantisation: Efficient Finegrained Quantisation for LLM. Available at: https://arxiv.org/abs/2310.04836.

59. Agarwal, R., Vieillard, N., Zhou, Y., Stanczyk, P., Ramos, S., Geist, M., Bachem, O., 2024, On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes. Available at: https://arxiv.org/abs/2306.13649.

60. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J., 2017, Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. Available at: https://arxiv.org/abs/1701.06538.

61. Schuster, T., Fried, D., & Jurafsky, D., 2022, Confident adaptive language modeling. Available at: https://arxiv.org/abs/2207.07061.

62. Tri Dao et al. "FlashAttention. Available at: https://arxiv.org/abs/2205.14135.

63. Leviathan, Y., Ram, O., Desbordes, T., & Haussmann, E., 2022, Fast inference from transformers via speculative decoding. Available at: https://arxiv.org/abs/2211.17192.

64. Li, Y., Humphreys, P., Sun, T., Carr, A., Cass, S., Hawkins, P., ... & Bortolussi, L., 2022, Competition-level code generation with AlphaCode. *Science*, 378(1092-1097). DOI: 10.1126/science.abq1158.

65. Romera-Paredes, B., Barekatain, M., Novikov, A., Novikov, A., Rashed, S., & Yang, J., 2023, Mathematical discoveries from program search with large language models. *Nature*. DOI: 10.1038/s41586-023-06924-6.

66. Wikipedia., 2024, Cap set. Available at: https://en.wikipedia.org/wiki/Cap_set.

67. Trinh, T. H., Wu, Y., & Le, Q. V. et al., 2024, Solving olympiad geometry without human demonstrations. *Nature,* 625, 476–482. DOI: 10.1038/s41586-023-06747-5.