> 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 NLP models 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 model long-term contexts more effectively 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. Although they have infinite context length, in practice they struggle to utilize it due to vanishing gradient problem. Transformers have become the most popular approach for sequence modeling and transfer learning 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!

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[68] 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.

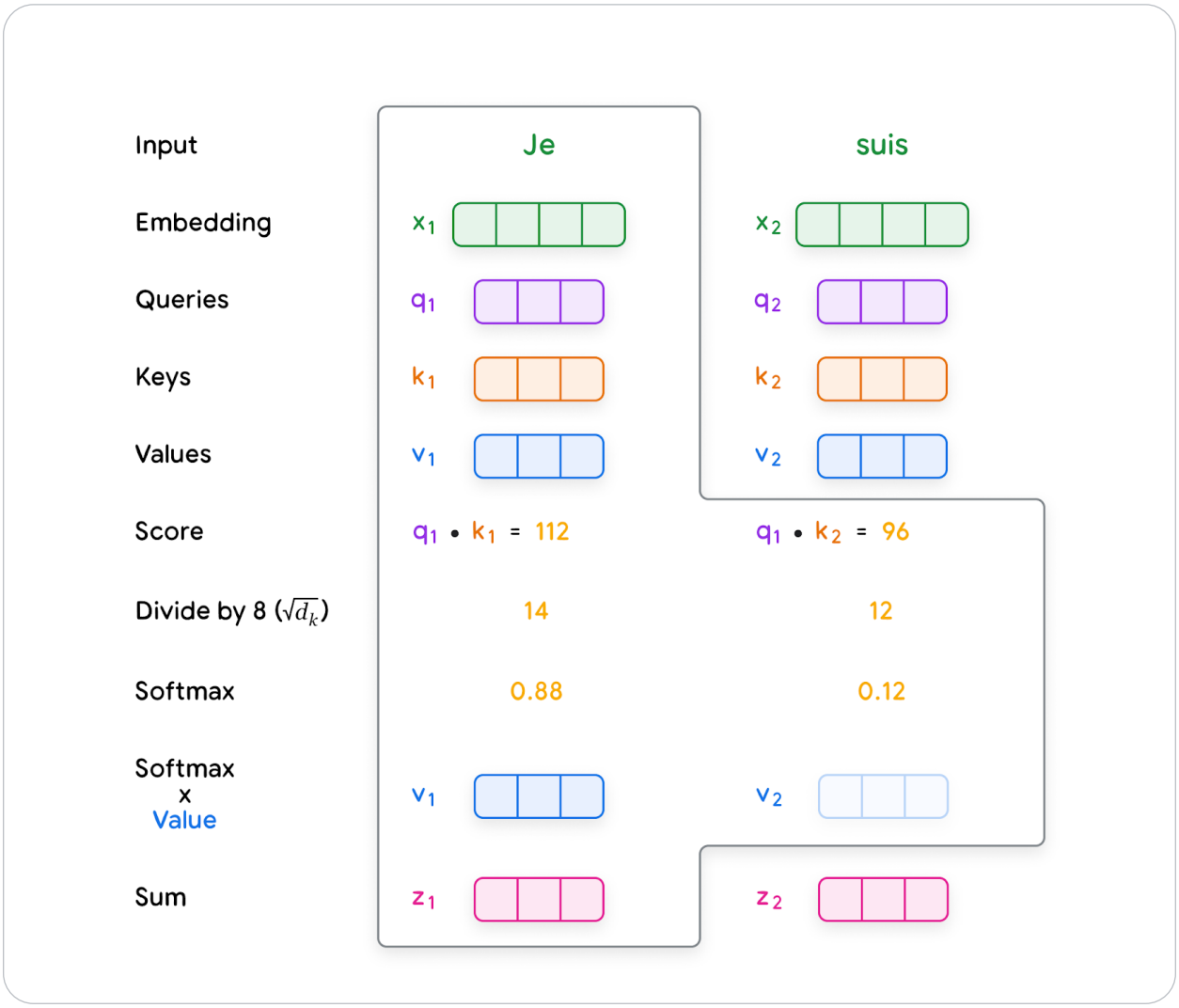

Figure 2. The process of computing self-attention in the multi-head attention module[1] (P.C:[5])

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.

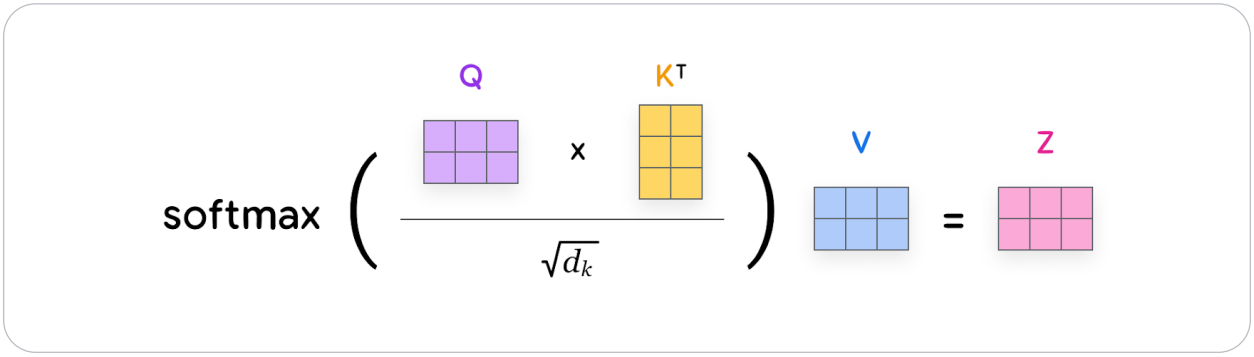

Figure 3. The basic operation of attention,[1] with Q=query, K=Keys and V=Value, Z=Attention, d_k = dimension of queries and keys (P.C:[5])

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.

Mixture of Experts (MoE)

A Mixture of Experts (MoE) is a an architecture that combines multiple specialized sub-models (the "experts") to improve overall performance, particularly on complex tasks. It's a form of ensemble learning, but with a key difference: instead of simply aggregating the predictions of all experts, it learns to route different parts of the input to different experts. This allows the model to specialize, with each expert becoming proficient in a specific sub-domain or aspect of the data. Here's a more technical breakdown describing the main components of an MoE:

- **Experts:** These are the individual sub-models, each designed to handle a specific subset of the input data or a particular task. They can be any type of model (e.g., neural networks, decision trees, etc.), but in the context of large language models, they are typically themselves transformer-based architectures.
- **Gating Network (Router):** This is a crucial component that learns to route the input to the appropriate expert(s). It takes the input and produces a probability distribution over the experts. This distribution determines how much each expert should "contribute" to the final prediction. The gating network is also typically a neural network.
- **Combination Mechanism:** This combines the outputs of the experts, weighted by the probabilities from the gating network, to produce the final prediction. A common approach is a weighted average.

In practice, A Mixture of Experts (MoE) architecture combines multiple specialized sub-models, called “experts,” to tackle complex tasks. Instead of simply averaging all expert predictions, an MoE uses a “gating network” to intelligently route different parts of the input to the mostt relevant experts. Both the experts and the gating network receive the input. Each expert processes the input and generates its output. Simultaneously, the gating network analyzes the input and produces a probability distribution over the experts, indicating how much each expert should contribute to the final result. These probabilities then weight the outputs of the experts, and the weighted combination becomes the final prediction. This allows different experts to specialize in handling specific types of data or sub-tasks, improving overall performance and, through “sparse activation,” potentially reducing computational cost by only activating a subset of experts for any given input.

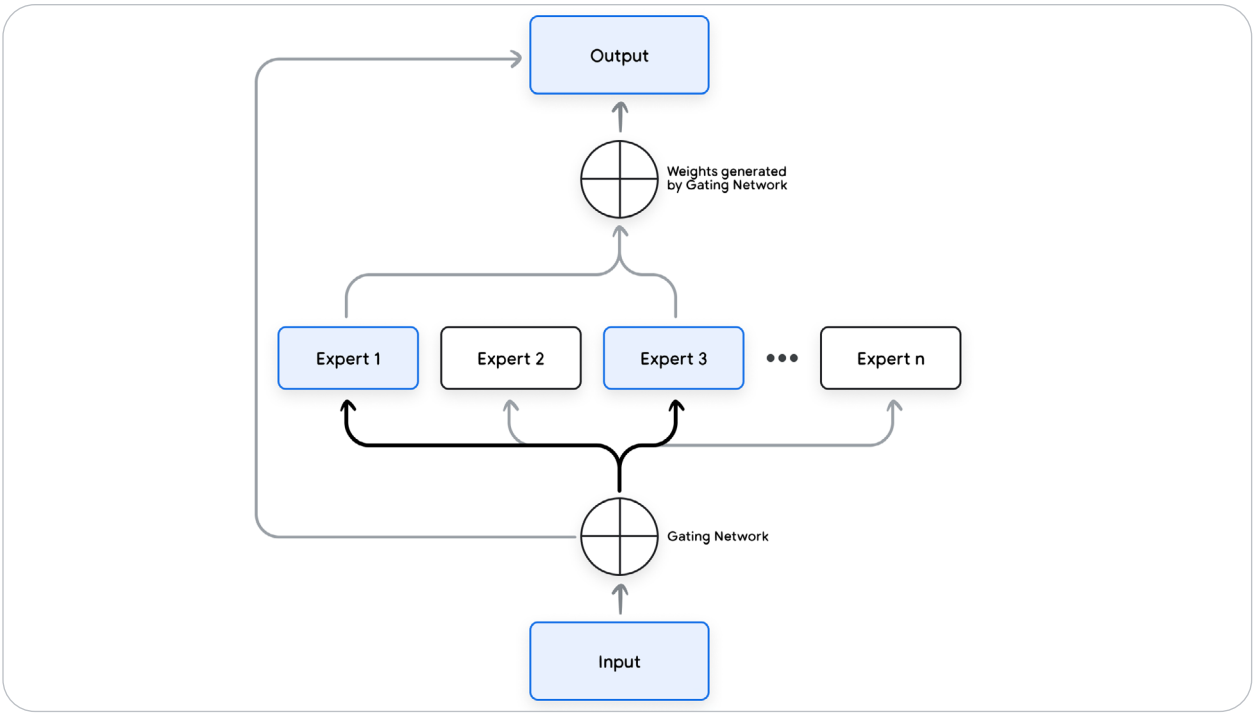

Figure 4. Mixture of experts ensembling[70]

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.

- 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.

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.

As of this writing, Gemini 2.0 Flash Thinking offers an unparalleled balance of quality, as measured by its ELO score, and affordability, with a cost per million tokens that is ten times lower than comparable models; its position on a quality-versus-cost graph (where further up and to the right is superior) demonstrates its transformative development. Moreover, the picture highlights the rapid advancements in reasoning and thinking capabilities within the AI field, with a 27-fold improvement observed in the last three months.

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.