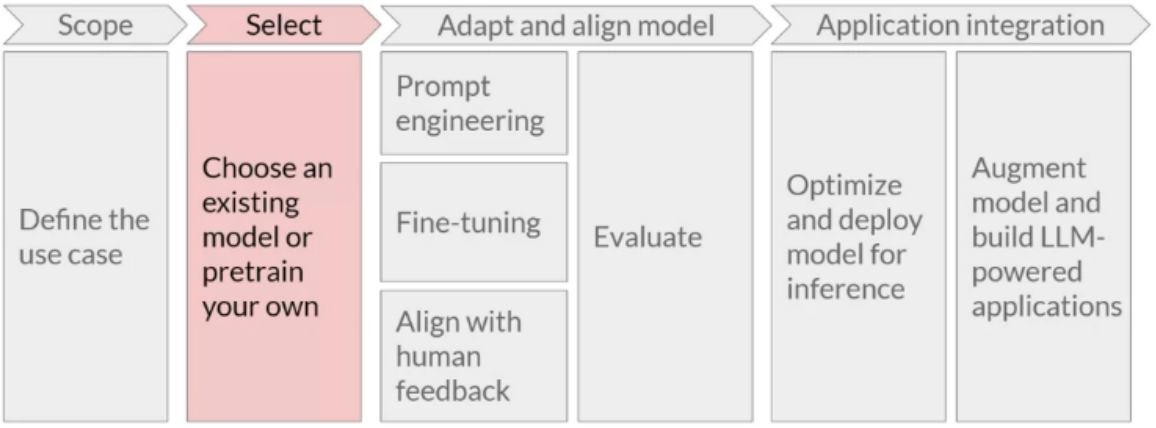
LLM pre-training and scaling laws

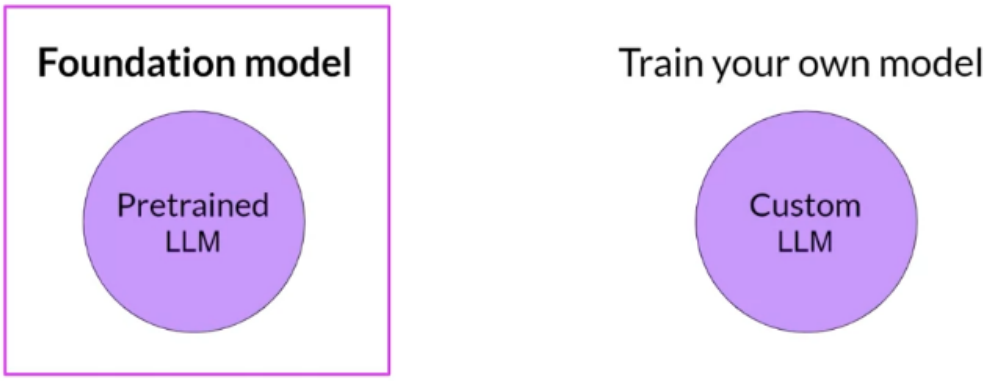
> Source: Coursera

▼ Pre-training large language models

In the generative AI project life cycle, we select a model once the use case is scoped out, and we determine how the LLM will work within the application.

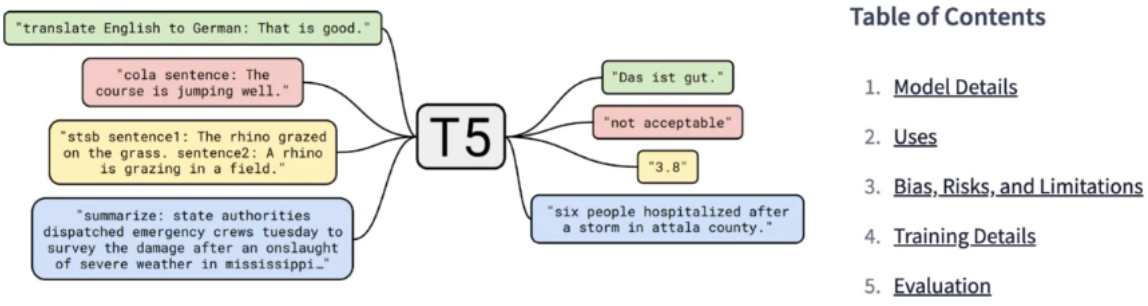


The first choice will be to either work with an existing model or train the own one from scratch. In general, we'll begin developing an application using an existing foundation model.



Many open-source models are available for members of the AI community to use in the application. The developers of some of the major frameworks for building generative AI applications, like Hugging Face and PyTorch, have curated hubs where we can browse these models.

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



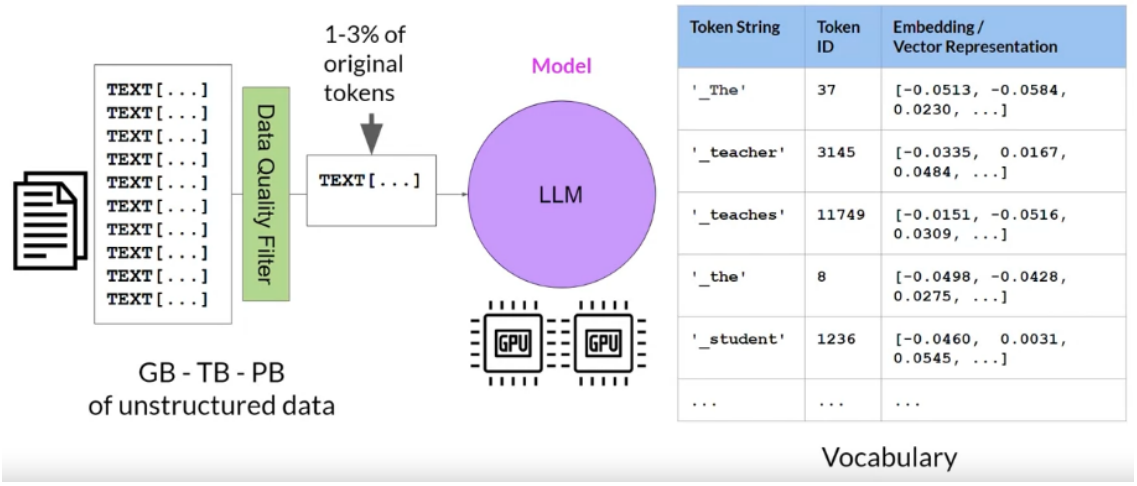
Model hubs - Model Card for T5 Large

The exact chosen model will depend on the details of the task that needs to be carried out. The variance in the transformer model architecture is suited to different language tasks largely because of differences in how the models are trained.

▼ Model architectures and pre-training objectives

LLMs encode a deep statistical representation of language. This understanding is developed during the model's pre-training phase when the model learns from vast amounts of unstructured textual data, which can be gigabytes, terabytes, and even petabytes of text. This data is pulled from many sources, including scrapes off the Internet and corpora of texts assembled specifically for training language models.

In this self-supervised learning step, the model internalizes the patterns and structures present in the language. These patterns then enable the model to complete its training objective, which depends on the model's architecture.



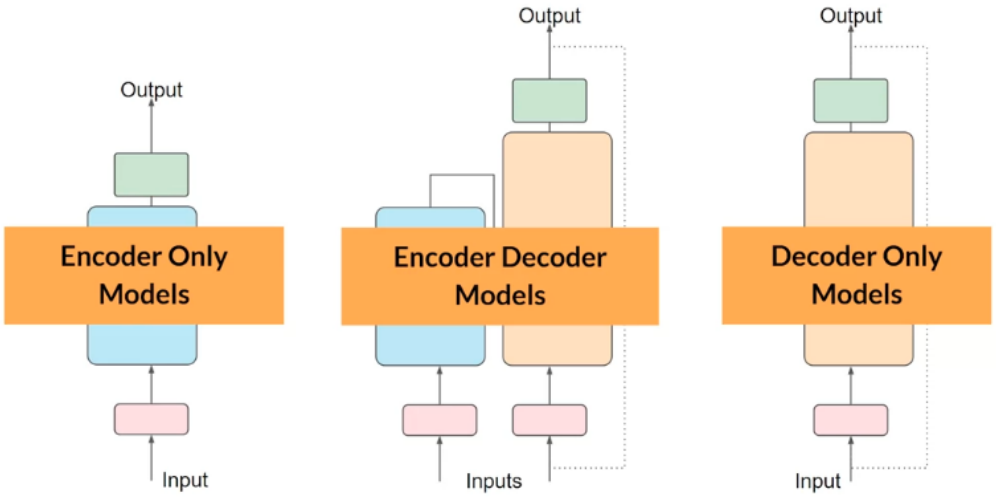
LLM pre-training at a high level

During pre-training, the model weights are updated to minimize the loss of the training objective. The encoder generates an embedding or vector representation for each token. Pre-training also requires extensive computing and the use of GPUs.

Note: The training data from public sites such as the Internet often needs to process the data to increase quality, address bias, and remove other harmful content. As a result of this data quality curation, only 1-3% of tokens are often used for pre-training. This should be considered when estimating how much data needs to be collected to pre-train the model.

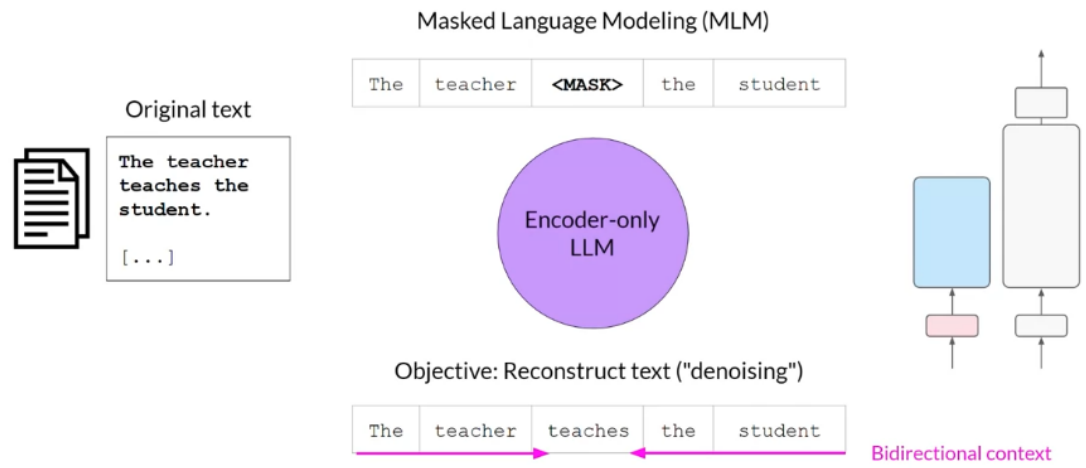
▼ The transformer models

The transformer model had three variances: encoder-only, encoder-decoder models, and decode-only. Each is trained on a different objective and learns to perform different tasks.



▼ Encoder-only models

Encoder-only models, also known as Autoencoding models, are pre-trained using masked language modelling. Here, tokens in the input sequence are randomly masked, and the training objective is to predict the mask tokens to reconstruct the original sentence. This is also called a denoising objective.

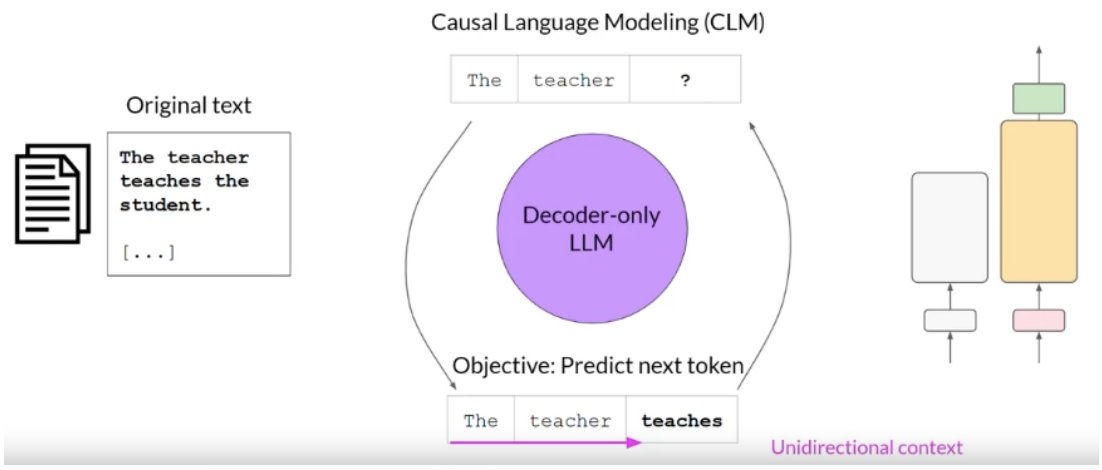


Autoencoding models split bidirectional representations of the input sequence, meaning that they understand the full context of a token and not just the words that come before it. Encoder-only models are ideally suited to tasks that benefit from this bidirectional context.

They can be used to perform sentence classification tasks, such as sentiment analysis or token-level tasks, like named entity recognition or word classification. Some well-known examples of an autoencoder model are BERT and RoBERTa.

▼ Decoder-only or autoregressive models

They are pre-trained using causal language modelling. The training objective is to predict the next token based on the previous sequence of tokens. Researchers sometimes call this prediction full language modelling.



Decoder-based autoregressive models mask the input sequence and can only see the input tokens leading up to the token in question. The model does not know the end of the sentence. The model then iterates over the input sequence one by one to predict the following token.

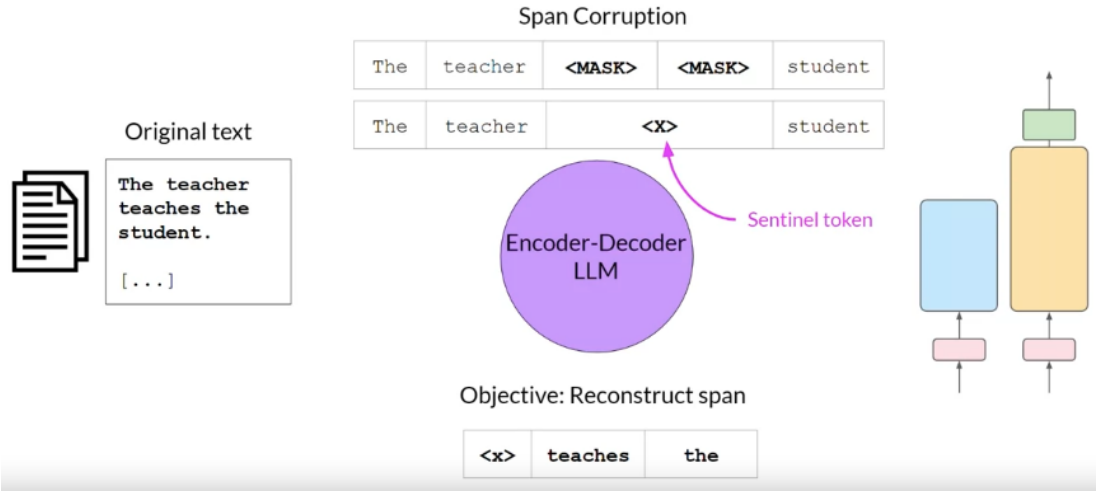
In contrast to the encoder architecture, the context is unidirectional. The model builds up a statistical representation of language by learning to predict the next token from many examples. Models of this type use the decoder component of the original architecture without the encoder.

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

well. Well-known examples of decoder-based autoregressive models are GBT and BLOOM.

▼ Encoder-Decoder model or sequence-to-sequence model

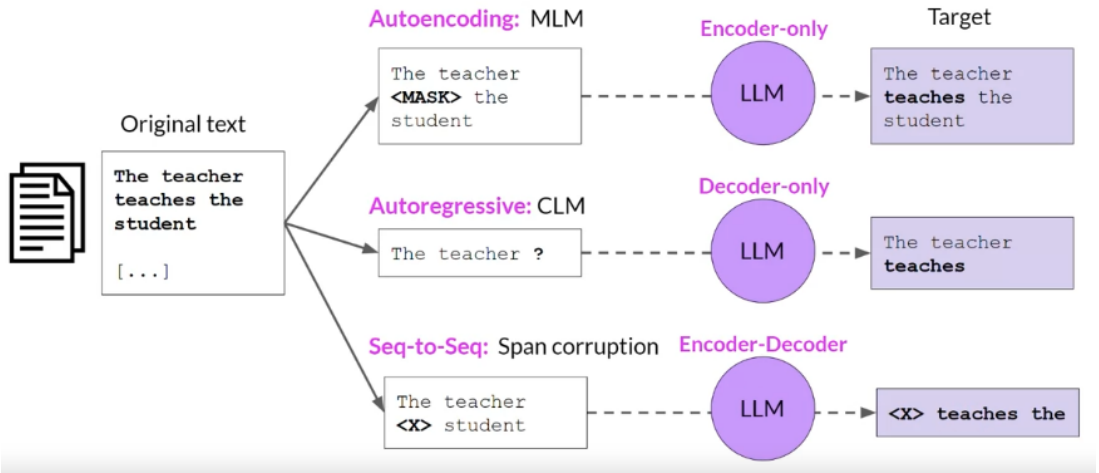
The exact details of the pre-training objective vary from model to model. A popular sequence-to-sequence model, T5, pre-trains the encoder using span corruption, which masks random sequences of input tokens. Those mass sequences are replaced with a unique Sentinel token, shown here as x. Sentinel tokens are special tokens added to the vocabulary but do not correspond to any actual word from the input text.



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

The sequence-to-sequence models can be used for translation, summarization, and question-answering. They are generally helpful when the body of texts has both input and output. Besides T5, another well-known encoder-decoder model is BART.

▼ Summary



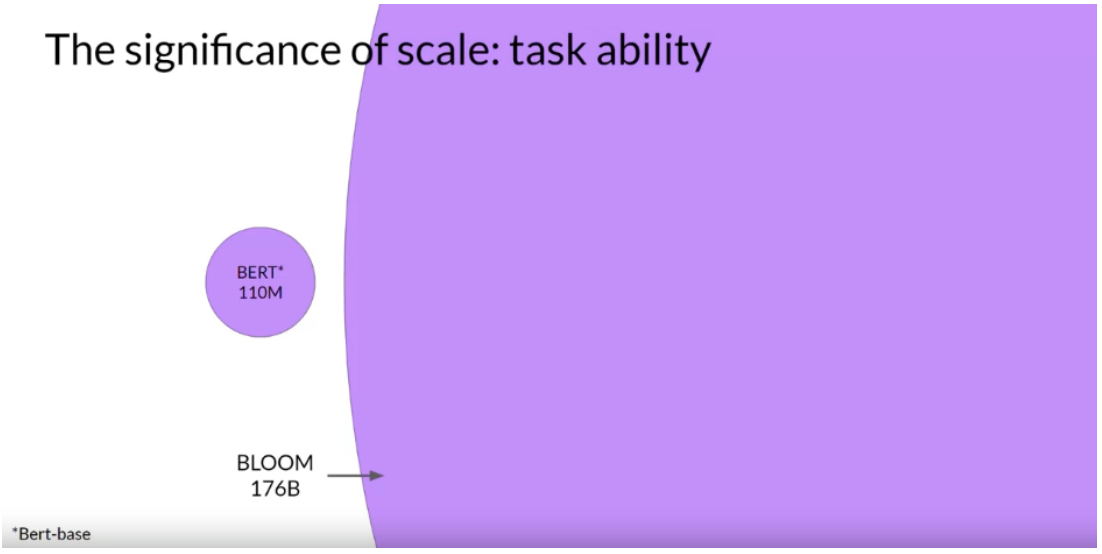
Autoencoding models are pre-trained using masked language modelling. They correspond to the encoder part of the original transformer architecture and are often used with sentence or token classification.

Autoregressive models are pre-trained using causal language modelling. Models of this type use the decoder component of the original transformer architecture and are often used for

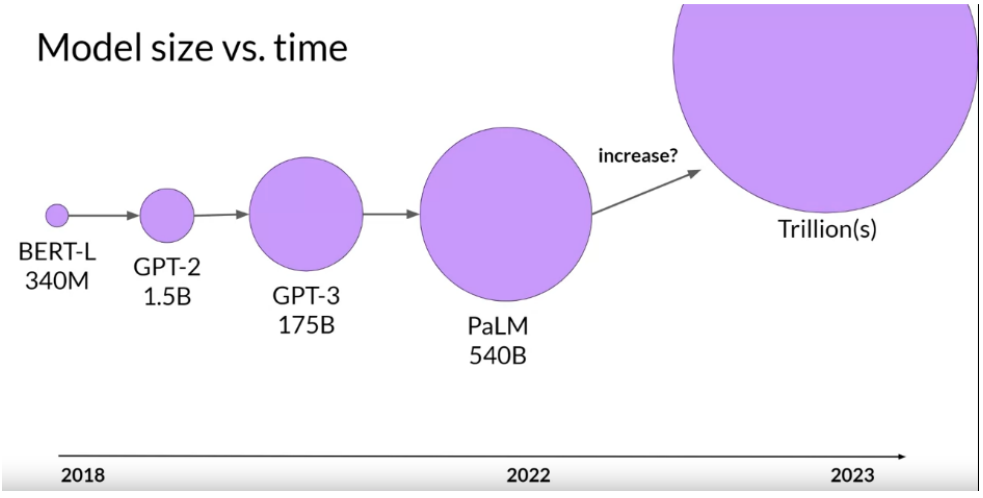
text generation.

Sequence-to-sequence models use both the encoder and decoder parts of the original transformer architecture. The exact details of the pre-training objective vary from model to model. The T5 model is pre-trained using span corruption. Sequence-to-sequence models are often used for translation, summarization, and question-answering.

The larger architecture models are typically more capable of carrying out their tasks well. Researchers have found that the larger a model, the more likely it is to work as needed without additional in-context learning or further training.



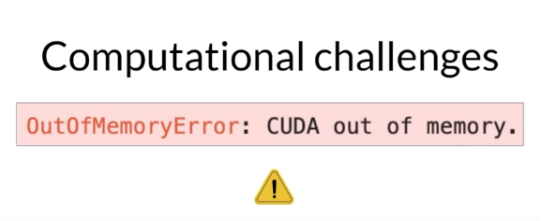
This observed trend of increased model capability with size has driven the development of larger and larger models in recent years. This growth has been powered by inflexion points and research, such as introducing the highly scalable transformer architecture, access to massive data for training, and developing more powerful computing resources.



This steady increase in model size led some researchers to hypothesise the existence of a new Moore's law for LLMs. Can we keep adding parameters to increase performance and make models smarter? Where could this model growth lead? While this may sound great, training these enormous models is difficult and very expensive, so much so that it may be infeasible to train larger and larger models continuously.

▼ Computational challenges of training LLMs

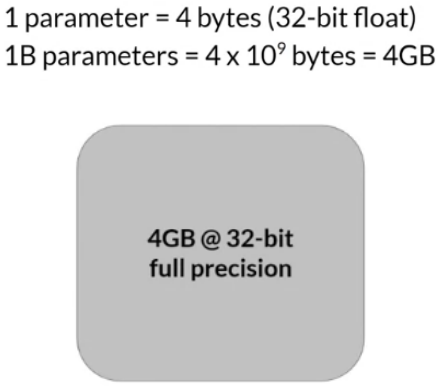
▼ Computational Challenges



One of the most common issues in training large language models is running out of memory. This error message is common when training or loading a model on Nvidia GPUs.

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

These are out-of-memory issues because most LLMs are huge and require a ton of memory to store and train their parameters.



Typically, a 32-bit float represents a single parameter: how computers represent real numbers. A 32-bit float takes up four bytes of memory. So, it needs four bytes times one billion parameters or four gigabytes of GPU RAM at 32-bit full precision to store one billion parameters. It is a lot of memory, and note if only accounted for the memory to store the model weights so far.

Additional components that use GPU memory during training must be planned to train the model. These include two Adam optimizer states, gradients, activations, and temporary variables needed by functions. This can easily lead to 20 extra bytes of memory per model parameter.

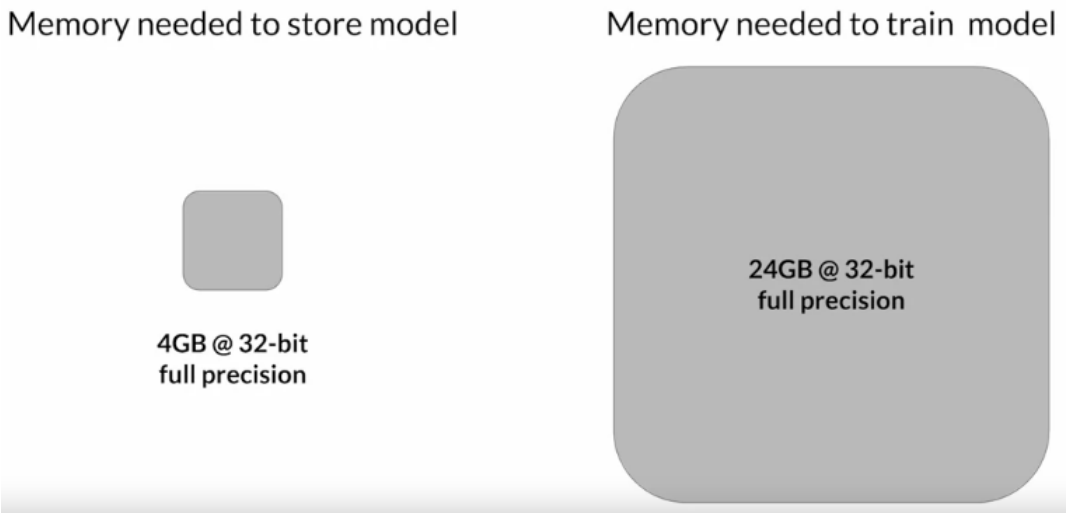
Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#tlanatomy-of-models-memory, https://rithub.com/facebookresearch/bitsandbytes

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

~20 extra bytes per parameter

Additional GPU RAM needed to train 1B parameters

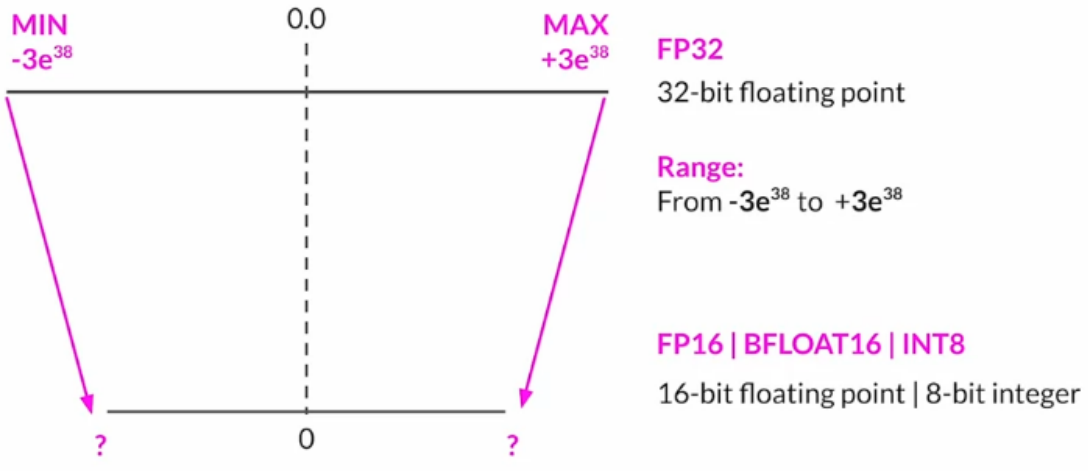
To account for these overheads during training, approximately 6 times the amount of GPU RAM that the model weights alone take up is required. To train a one billion parameter model at 32-bit full precision, approximately 24 gigabytes of GPU RAM is needed. This is too large for consumer hardware and even challenging for hardware used in data centers.



Approximate GPU RAM needed to train 1B-params

▼ Quantization

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

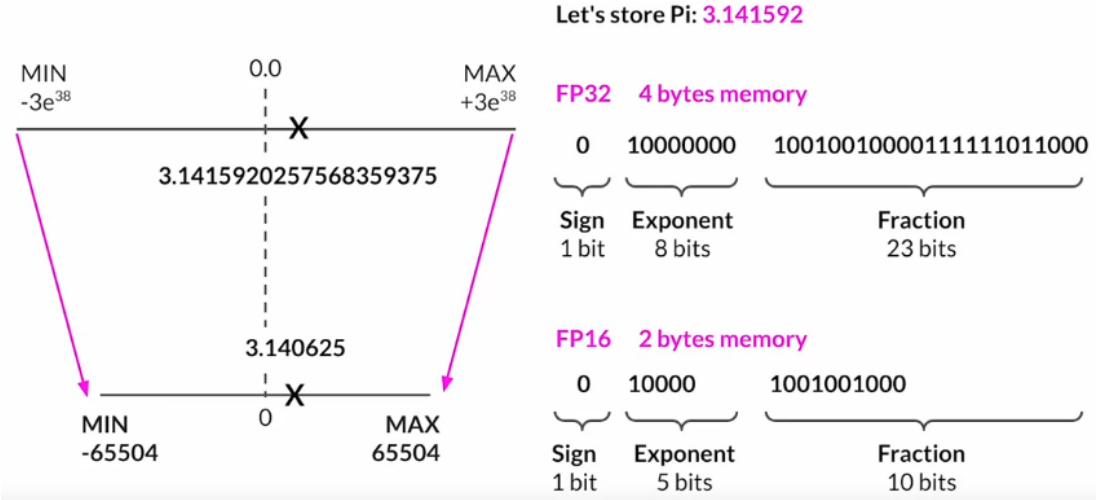


▼ FP16

The corresponding data types used in deep learning frameworks and libraries are FP32 for the 32-bit full position, FP16 or Bfloat16 for 16-bit half-precision, and int8 eight-bit integers. The range of numbers represented with FP32 goes from approximately -3*10^38 to 3*10^38. By default, model weights, activations, and other model parameters are stored in FP32. Quantization statistically projects the original 32-bit floating point numbers into a lower

precision space, using scaling factors calculated based on the range of the original 32-bit floating point numbers.

Example: Suppose a PI is stored in six decimal places in different positions. Floating point numbers are stored as bits, zeros, and ones. The 32 bits to store numbers in full precision with FP32 consist of one bit for the sign where zero indicates a positive number, and one a negative number. Then, eight bits are for the exponent of the number, and 23 bits represent the fraction of the number. The fraction is also referred to as the mantissa or significant. It represents the precision bits of the number.



Quantization: FP16

If we convert the 32-bit floating point value back to a decimal value, you notice a slight loss in precision. For reference, 3.1415926535897932384 is the actual value of Pi to 19 decimal places. If we project this FP32 representation of Pi into the FP16, 16-bit lower precision space. The 16 bits consist of one bit for the sign for FP32.

But now, FP16 only assigns five bits to represent the exponent and 10 bits to represent the fraction. Therefore, the range of numbers representing FP16 is vastly smaller, from negative 65,504 to positive 65,504. The original FP32 value gets projected to 3.140625 in the 16-bit space.

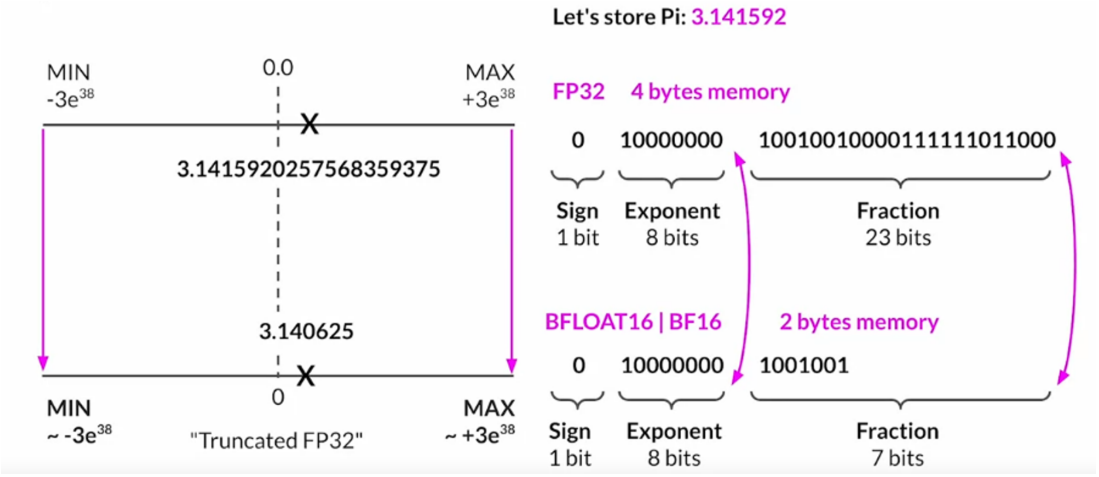
We lose some precision with this projection. There are only six places after the decimal point now. This loss in precision is acceptable in most cases because we're trying to optimize for memory footprint.

Storing a value in FP32 requires four bytes of memory. In contrast, storing a value on FP16 requires only two bytes of memory, so quantization reduces the memory requirement by half.

▼ BFLOAT16

The AI research community has explored ways to optimize 16-bit quantization. One datatype, BFLOAT16, has recently become a popular alternative to FP16. BFLOAT16, short for Brain Floating Point Format, developed at Google Brain, has become a popular choice in deep learning.

Many LLMs, including FLAN-T5, have been pre-trained with BFLOAT16. BFLOAT16, or BF16, is a hybrid between half-precision FP16 and full-precision FP32. BF16 significantly helps with training stability and is supported by newer GPUs, such as NVIDIA's A100.

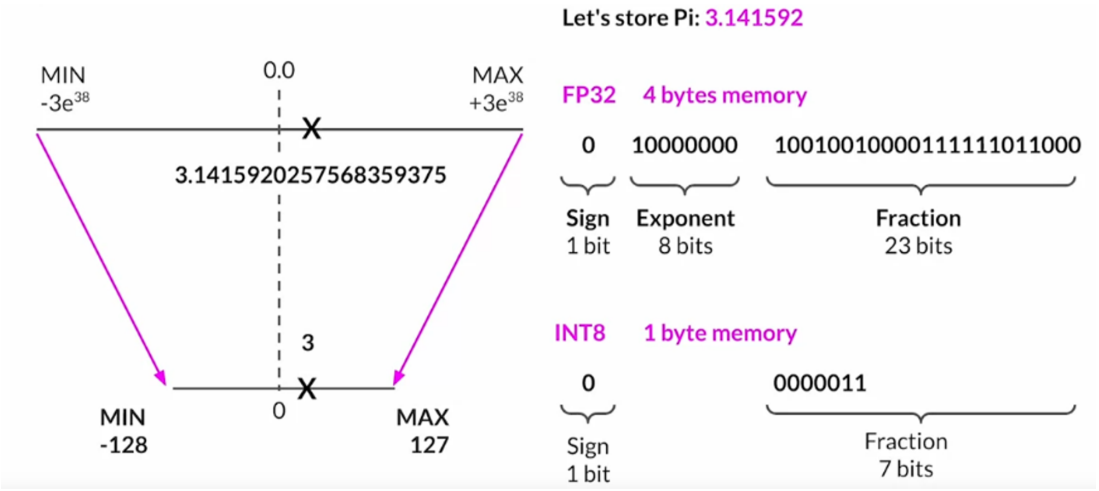


Quantization: BFLOAT16

BFLOAT16 is often described as a truncated 32-bit float, as it captures the full dynamic range of the total 32-bit float that uses only 16-bits. BFLOAT16 uses the full eight bits to represent the exponent but truncates the fraction to just seven bits. This saves memory and increases model performance by speeding up calculations. The downside is that BF16 is not well suited for integer calculations, which are relatively rare in deep learning.

▼ INT8

What happens if we quantize Pi from the 32-bit into the even lower precision eight-bit space? If we use one bit for the sign, INT8 values are represented by the remaining seven bits. This gives us a range to represent numbers from -128 to 127, and unsurprisingly, Pi gets projected 2 or 3 in the 8-bit lower precision space.



Quantization: INT8

This reduces the new memory requirement from 4 bytes originally to 1 byte, but it obviously results in a pretty dramatic loss of precision.

▼ Summary

Quantization's goal is to reduce the memory required to store and train models by reducing the precision of the model weights. Quantization statistically projects the original 32-bit floating point numbers into lower precision spaces using scaling factors calculated based on the range

of the original 32-bit floats. Modern deep learning frameworks and libraries support quantization-aware training, which learns the quantization scaling factors during training.

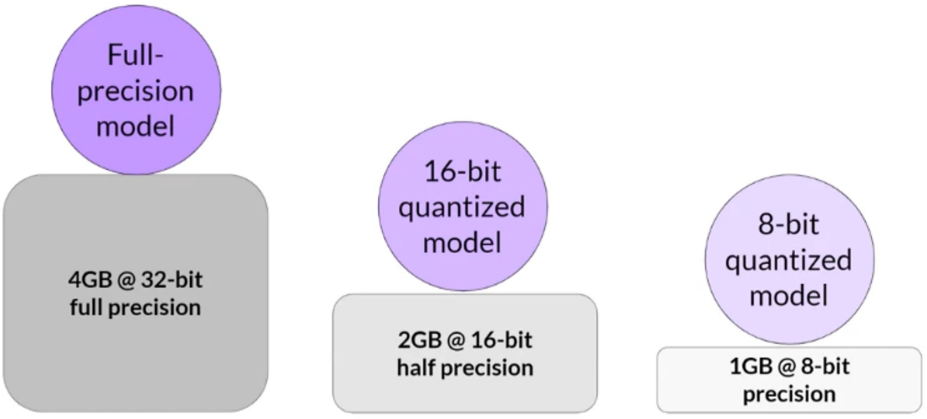
	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
BFLOAT16	16	8	7	2 bytes
INT8	8	-/-	7	1 byte

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

BFLOAT16 has become a popular choice for precision in deep learning. It maintains the dynamic range of FP32 but reduces the memory footprint by half. Many LLMs, including FLAN-T5, have been pre-trained with BFOLAT16.

Applying quantization can reduce the memory consumption required to store the model parameters to only two gigabytes using a 16-bit half-precision of 50% saving. Representing the model parameters as eight-bit integers could further reduce the memory footprint by another 50%, requiring only one gigabyte of GPU RAM.

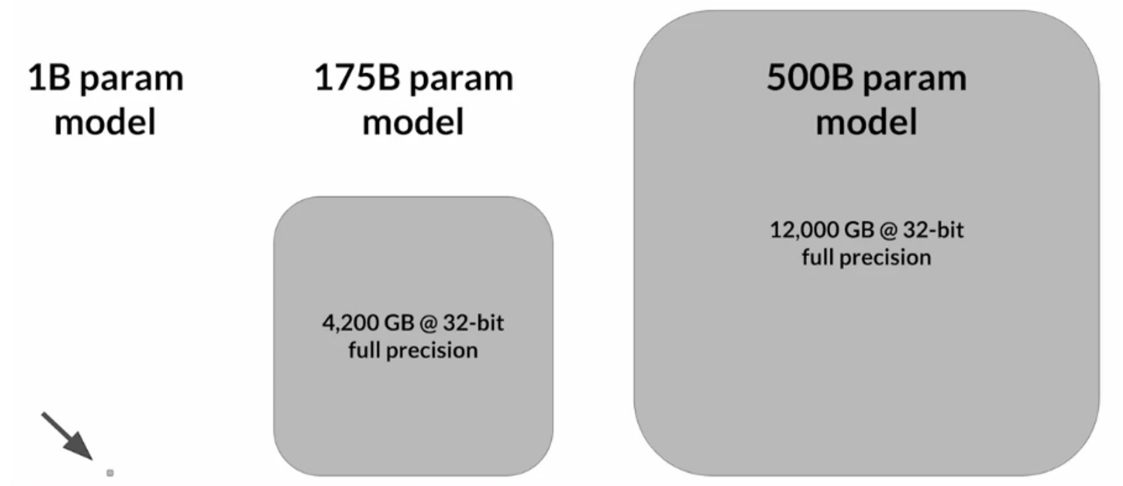
Sources: https://huggingface.co/docs/transformers/v4.20.1/en/perf_train_gpu_one#anatomy-of-models-memory, https://github.com/facebookresearch/bitsandbytes



Approximate GPU RAM needed to store 1B parameters.

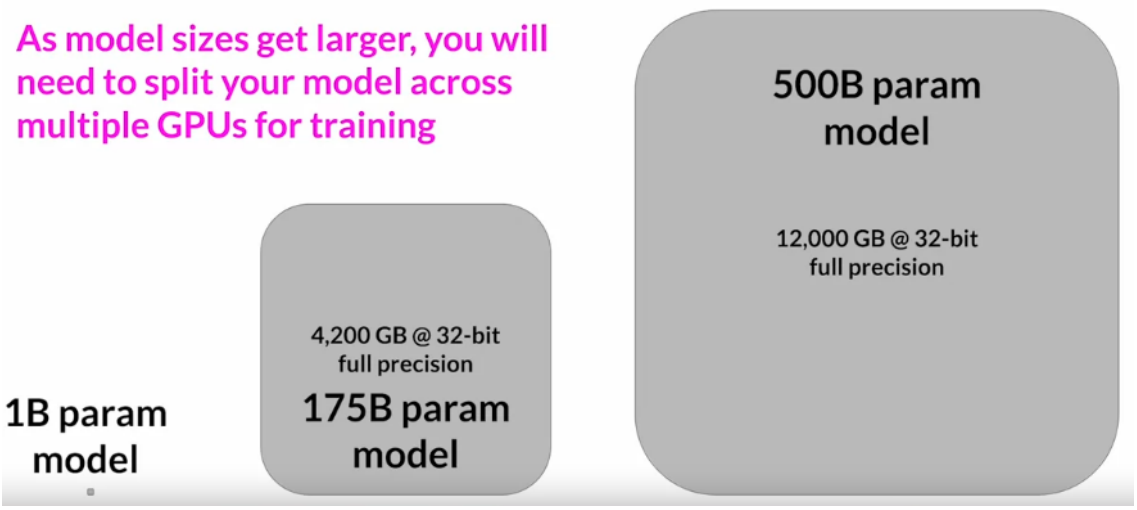
Note: After all, the model still has one billion parameters. The circles representing the models are the same size. Quantization will give the same degree of savings when it comes to training.

However, many models now have sizes over 50 billion or even 100 billion parameters. To train them, they need up to 500 times more memory capacity, tens of thousands of gigabytes.



GPU RAM needed to train larger models

These enormous models dwarf the one billion parameter model we've considered, shown here to scale on the left. As models scale beyond a few billion parameters, training them on a single GPU becomes impossible.

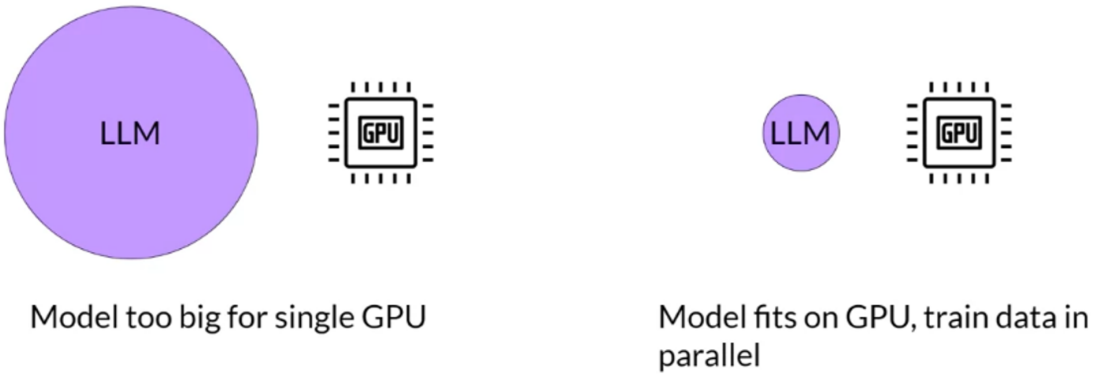


GPU RAM needed to train larger models

Instead, we turn to distributed computing techniques while we train the model across multiple GPUs. This could require access to hundreds of GPUs, which is very expensive. That is another reason we won't pre-train the model from scratch most of the time.

However, an additional training process called fine-tuning requires storing all training parameters in memory, and we'll likely want to fine-tune a model at some point.

▼ Efficient multi-GPU compute strategies



When to use distributed compute

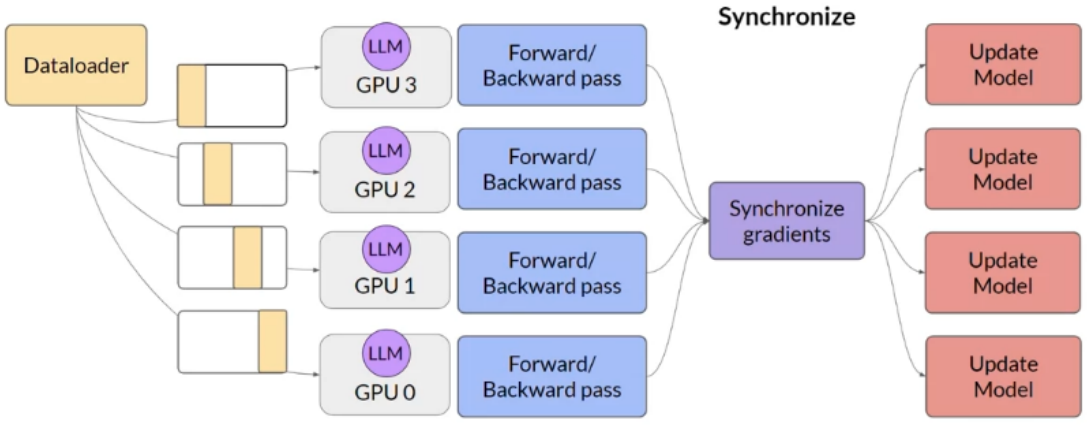
Using multi-GPU computing strategies is necessary when the model becomes too big to fit in a single GPU. But even if the model does fit onto a single GPU, there are benefits to using multiple GPUs to speed up the training. Knowing how to distribute computing across GPUs is useful, even when working with a small model.

▼ Distributed Data Parallel (DDP)

There are 2 cases we should consider when distributing data across GPUs.

▼ The model still fits on a single GPU

The 1st step in scaling model training is to distribute large data sets across multiple GPUs and process these batches of data in parallel.



A popular implementation of this model replication technique is Pytorch distributed data-parallel, or DDP for short. DDP copies the model onto each GPU and sends batches of data to each GPU in parallel. Each data set is processed in parallel, and then a synchronization step combines the results of each GPU, which in turn updates the model on each GPU, which is always identical across chips.

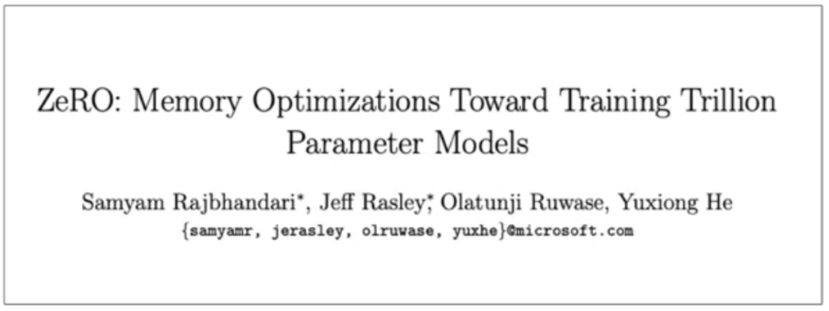
This implementation allows parallel computations across all GPUs, resulting in faster training. DDP requires that your model weights and all the additional parameters, gradients, and optimizer states needed for training fit onto a single GPU.

▼ Fully Sharded Data Parallel (FSDP)

> Sources: Rajbhandari et al. 2019: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", Zhao et al. 2023: "PyTorch

> FSDP: Experiences on Scaling Fully Sharded Data Parallel."

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



Fully Sharded Data Parallel (FSDP)

If the model is too big for DDP, we should look into another technique called modal sharding. Pytorch is a popular implementation of modal sharding with fully sharded data parallel (FSDP).

FSDP is motivated by a paper published by researchers at Microsoft in 2019 that proposed a technique called ZeRO. ZeRO stands for zero redundancy optimizer, and ZeRO aims to optimize memory by distributing or sharding model states across GPUs with ZeRO data overlap. This allows us to scale model training across GPUs when the model doesn't fit in the memory of a single chip.

▼ ZeRO redundancy optimizer

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

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

Additional GPU RAM needed for training

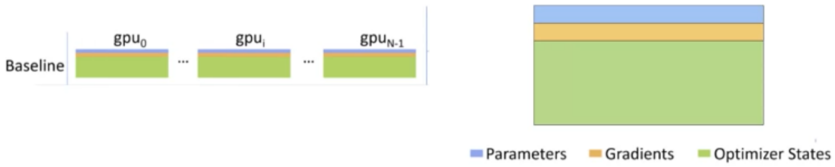
With the memory components required for training LLMs, the largest memory requirement was for the optimizer states, which take up twice as much space as the weights, followed by the weights and the gradients.

▼ Memory usage in DDP

One limitation of the model replication strategy is the need to keep a full model copy on each GPU, which leads to redundant memory consumption. We are storing the same numbers on every GPU.

> Sources: Rajbhandari et al. 2019: "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", Zhao et al. 2023: "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel"

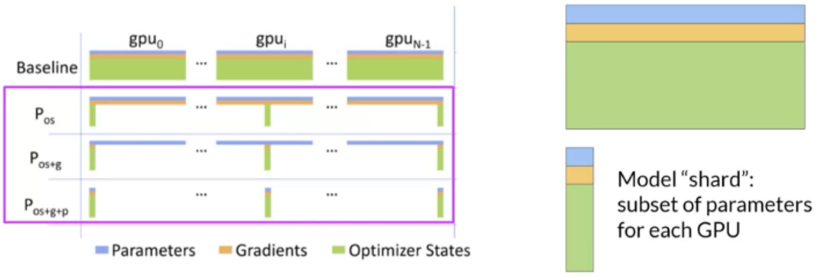
- One full copy of model and training parameters on each GPU



Memory usage in DDP

▼ Zero Redundancy Optimizer (ZeRO)

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



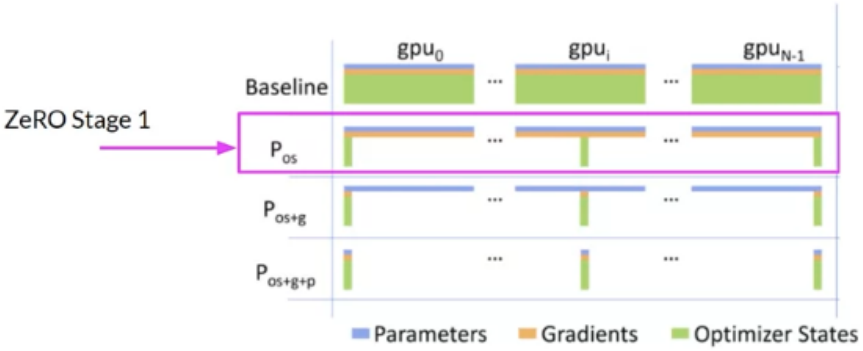
Zero Redundancy Optimizer (ZeRO)

ZeRO, on the other hand, eliminates this redundancy by distributing, also referred to as sharding, the model parameters, gradients, and optimizer states across GPUs instead of replicating them.

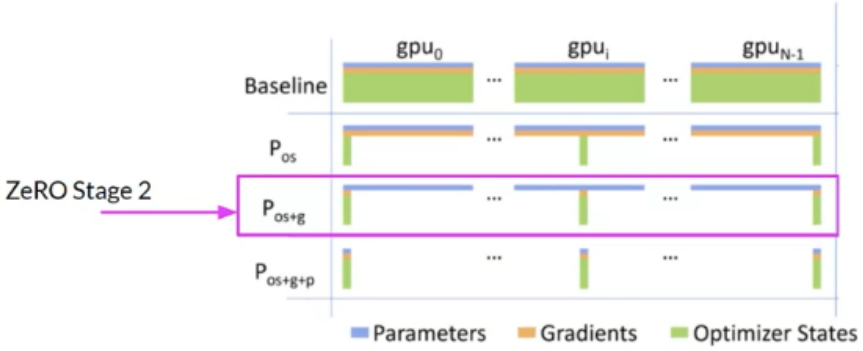
At the same time, the communication overhead for a sinking model state stays close to that of the DDP.

ZeRO offers three optimization stages.

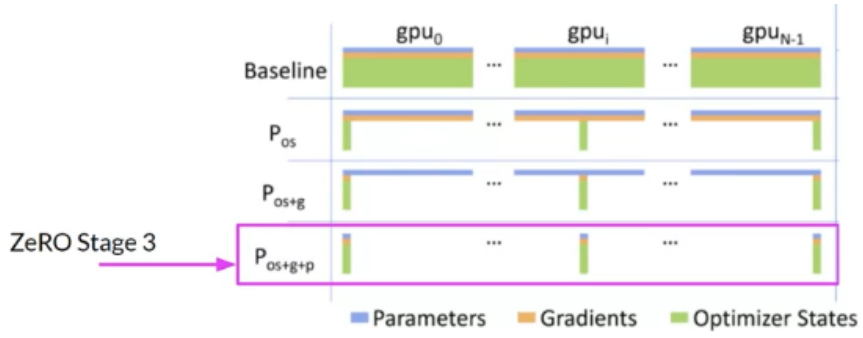
ZeRO Stage 1, shots only optimizer states across GPUs, can reduce the memory footprint by up to a factor of four.



ZeRO Stage 2 also sorts the gradients across chips. When applied together with Stage 1, this can reduce the memory footprint by up to eight times.



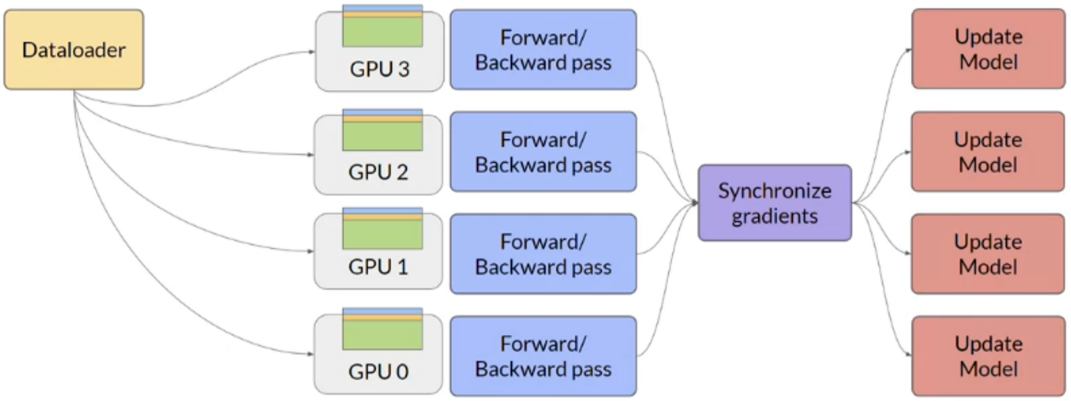
Finally, ZeRO Stage 3 sorts all components, including the model parameters, across GPUs.



When applied together with Stages 1 and 2, memory reduction is linear with a number of GPUs. For example, sharding across 64 GPUs could reduce your memory by a factor of 64.

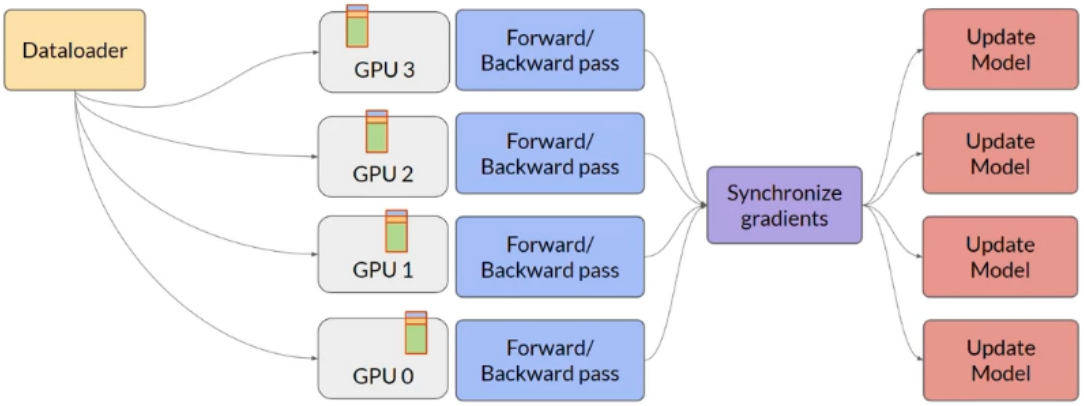
▼ Apply ZeRO technique into DDP and FSDP

Let's apply this concept to the visualization of DDP and replace the LLM with the memory representation of model parameters, gradients, and optimizer states.



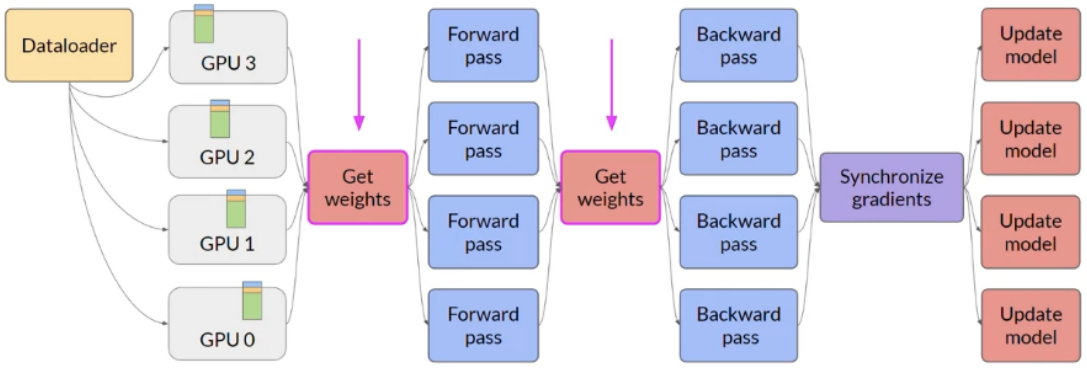
Distributed Data Parallel (DDP)

When using FSDP, we distribute the data across multiple GPUs, as in DDP. However, FSDP also distributes or shards the model parameters and gradients and optimizer's states across the GPU nodes using one of the strategies specified in the ZeRO paper.



Fully Sharded Data Parallel (FSDP)

This strategy allows us to work with models too big to fit on a single chip. In contrast to DDP, where each GPU has all of the model states required for processing each batch of data available locally, FSDP requires you to collect this data from all of the GPUs before the forward and backward pass.

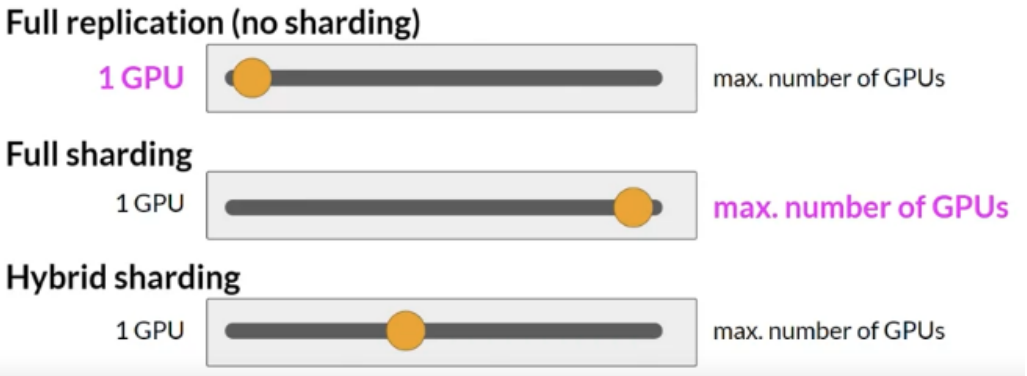


Fully Sharded Data Parallel (FSDP)

Each CPU requests data from the other GPUs on-demand to materialize the sharded data into unsharded data for the duration of the operation. After the operation, we release the

unsharded non-local data back to the other GPUs as original sharded data We can also choose to keep it for future operations during backward pass. This will requires more GPU RAM and is a typical performance versus memory trade-off decision.

In the final step after the backward pass, FSDP synchronizes the gradients across the GPUs in the same way they were for DDP. Model sharding 'S' described with FSDP reduce the overall GPU memory utilization. Optionally, FSDP offloads part of the training computation to CPUs to further reduce the GPU memory utilization. Configure the level of sharding using FSDP is charting factor can help manage the trade-off between performance and memory utilization.

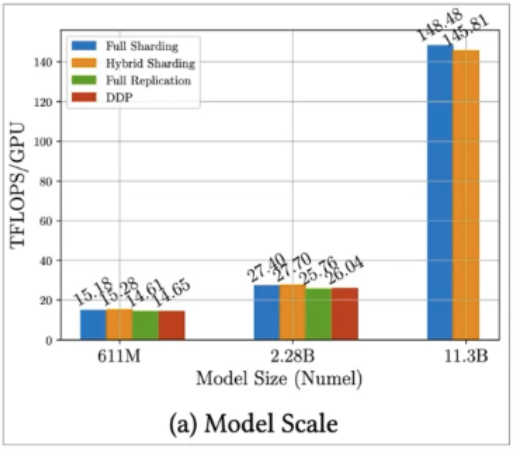


Configure level of sharding via sharding factor

A sharding factor of one basically removes the sharding and replicates the full model similar to DDP. Seting the sharding factor to the maximum number of available GPUs turn on full sharding. This has the most memory savings, but increases the communication volume between GPUs. Any sharding factor in-between enables hyper sharding.

Below is how FSDP performs in comparison to DDP measured in teraflops per GPU. These tests were performed using a maximum of 512 NVIDIA V100 GPUs, each with 80 gigabytes of memory. One teraflop corresponds to one trillion floating-point operations per second.

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

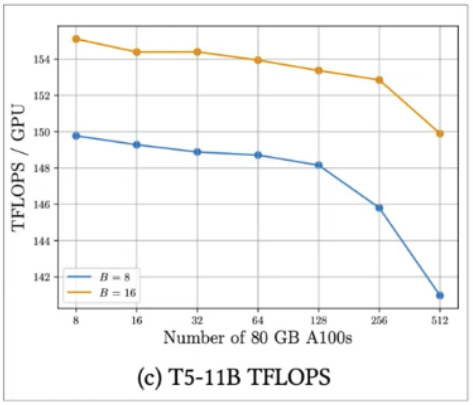


Impact of using FSDP

The first figure shows FSDP performance for different size T5 models. The different performance numbers for FSDP, full sharding in blue, hyper shard in orange and full replication in green. For reference, DDP performance is shown in red. For the first 25 models with 611 million parameters and 2.28 billion parameters, the performance of FSDP and DDP is similar.

Now, if the model size beyond 2.28 billion, such as 25 with 11.3 billion parameters, DDP runs out-of-memory. FSDP on the other hand can easily handle models this size and achieve much higher teraflops when lowering the model's precision to 16-bit.

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



Impact of using FSDP

The second figure shows 7% decrease in per GPU teraflops when increasing the number of GPUs from 8-512 for the 11 billion T5 model, plotted here using a batch size of 16 in orange and a batch size of eight in blue.

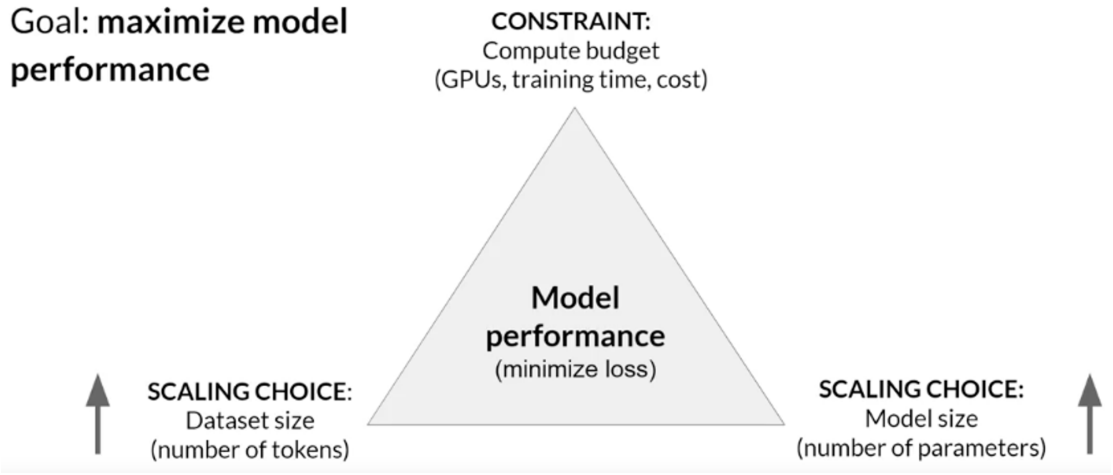
> Zhao et al. 2023: "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel"

As the model grows in size and is distributed across more and more GPUs, the increase in communication volume between chips starts to impact the performance, slowing down the computation.

In summary, can use FSDP for both small and large models and seamlessly scale the model training across multiple GPUs. Given the expense and technical complexity of training models across GPUs, some researchers have been exploring ways to achieve better performance with smaller models.

▼ Scaling laws and compute-optimal models

This research explored the relationship between model size, training, configuration and performance to determine how big models must be. The goal during pre-training is to maximize the model's performance of its learning objective, minimizing the loss when predicting tokens.

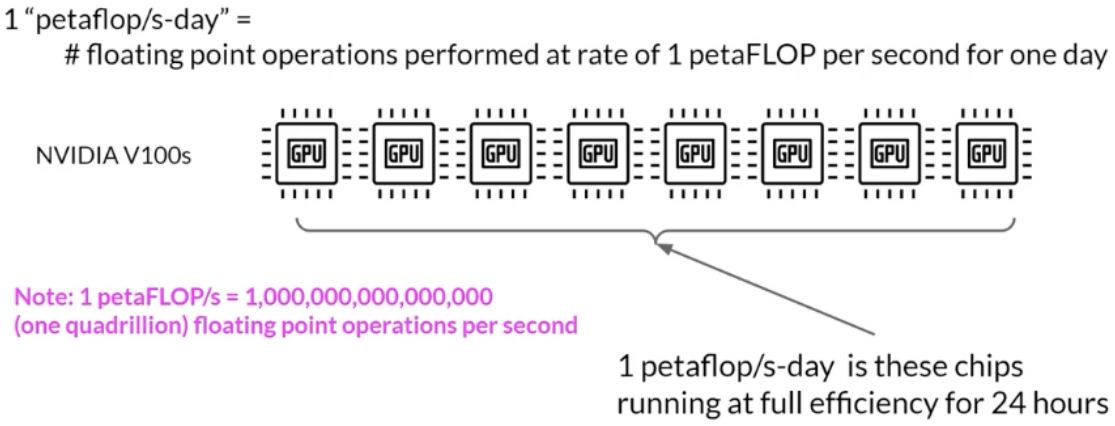


Scaling choices for pre-training

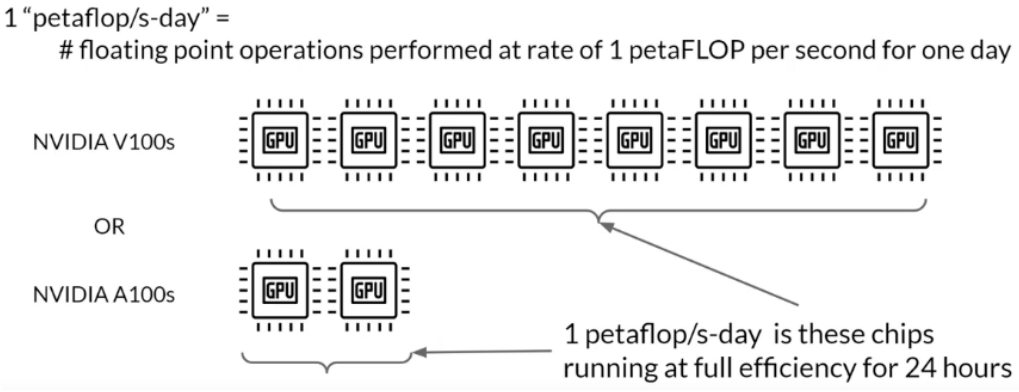
Two options to achieve better performance are increasing the dataset size on which to train the model and increasing the number of parameters in the model. In theory, we could scale either of these quantities to improve performance. However, another issue to consider is the compute budget, which includes factors like the number of GPUs that can be accessed and the time available for training models.

▼ Compute budget for training LLMs

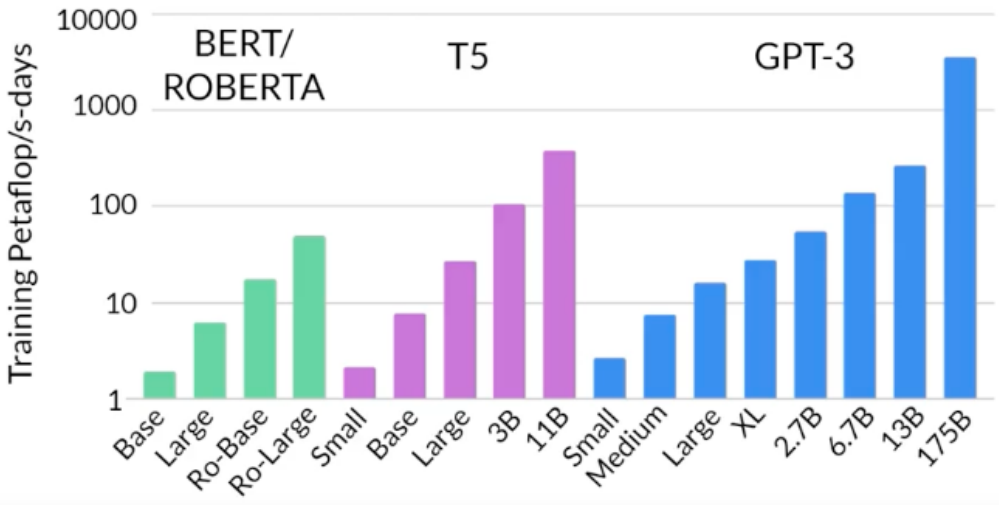
A petaFLOP per second day measures the number of floating point operations performed at a rate of one petaFLOP per second, running for an entire day.



Note: one petaFLOP corresponds to one quadrillion floating point operations per second. When specifically thinking about training transformers, one petaFLOP per second day is approximately equivalent to eight NVIDIA V100 GPUs operating efficiently for one full day.



With a more powerful processor that can carry out more operations simultaneously, a petaFLOP per second day requires fewer chips. For example, two NVIDIA A100 GPUs compute equivalent to the eight V100 chips.



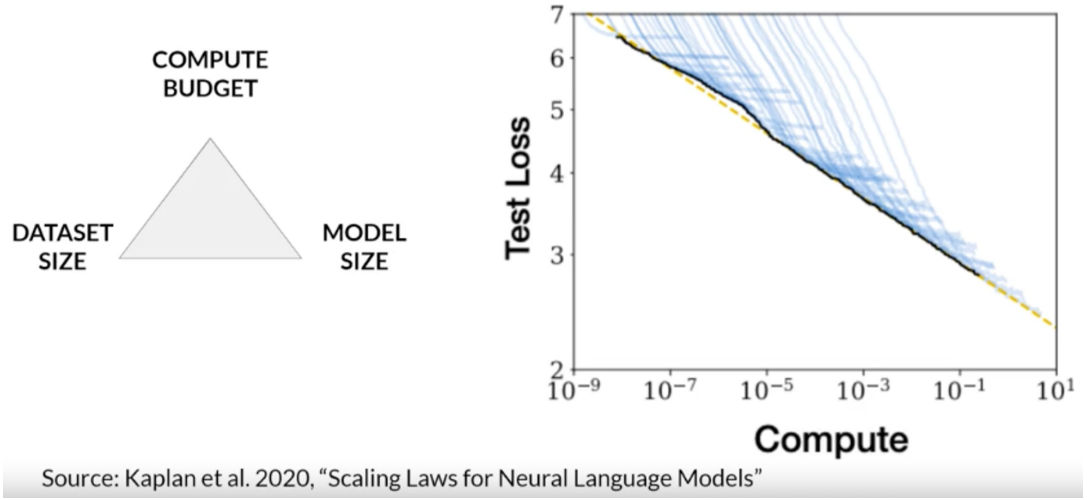
Number of petaflop/s-days to pre-train various LLMs, Source: Brown et al. 2020, "Language Models are Few-Shot Learners"

This chart compares the petaFLOP per second days required to pre-train different variances between Bert and Roberta, both encoder-only models. T5 and encoder-decoder model and GPT-3, which is a decoder-only model. The difference between the models in each family is the number of trained parameters, ranging from a few hundred million for Bert base to 175 billion for the largest GPT-3 variant.

The y-axis is logarithmic. Each increment vertically is a power of 10. Here, we see that T5 XL with three billion parameters required nearly 100 petaFLOP per second day. While the larger GPT-3 175 billion parameter model required approximately 3,700 petaFLOP per second day.

▼ Compute budget vs. model performance

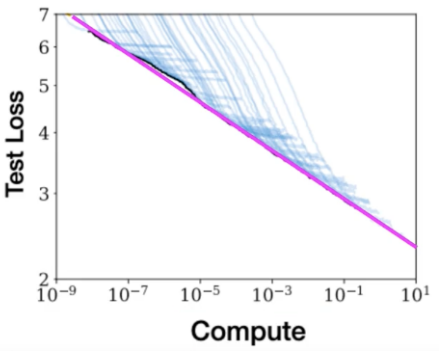
This chart clarifies that many computers are required to train the largest models. The bigger models take more computing resources to train and require more data to achieve good performance.



Compute budget vs. model performance, Source: Kaplan et al. 2020, "Scaling Laws for Neural Language Models"

It shows well-defined relationships between these three scaling choices. Researchers have explored the trade-offs between training dataset size, model size, and compute budget.

The figure from a paper by researchers at OpenAI explores the impact of computing budget on model performance. The y-axis is the test loss, a proxy for model performance, where smaller values are better. The x-axis is the computed budget in units of petaFLOP per second day. The larger numbers can be achieved by either using more computing power or training for longer, or both. Each thin blue line shows the model loss over a single training run. Looking at where the loss starts to decline more slowly for each run reveals a clear relationship between the compute budget and the model's performance.

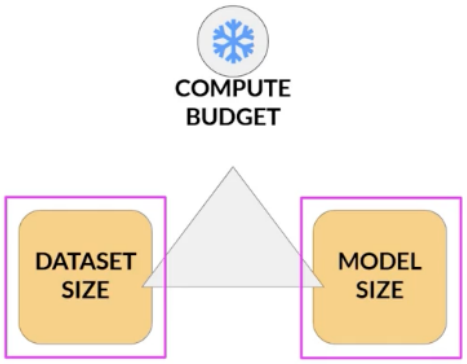


This pink line shows a power-law relationship that can approximate this. A power law is a mathematical relationship between two variables, where one is proportional to the other raised to some power.

Power-law relationships appear as straight lines when plotted on a graph where both axes are logarithmic. The relationship here holds as long as the model and training dataset sizes don't inhibit the training process.

Taken at face value, this would suggest increasing the compute budget to achieve better model performance.

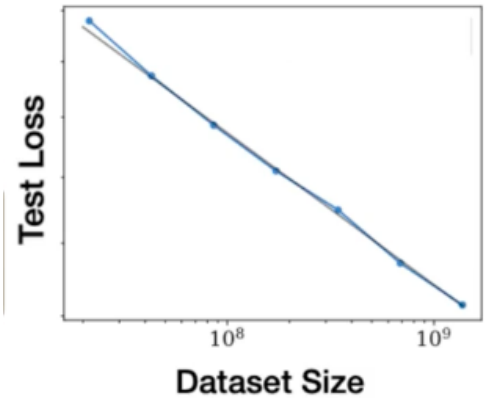
In practice, however, the computing resources available for training will generally be a hard constraint set by factors such as the accessed hardware, the time available for training and the project's financial budget.



Compute resource constraints

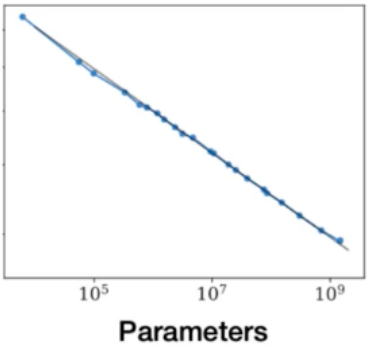
- Hardware
- Project timeline
- Financial budget

If the compute budget is fixed, the two levers to improve the model's performance are the size of the training dataset and the number of parameters in the model. The OpenAI researchers found that these two quantities also show a power-law relationship with a test loss when the other two variables are fixed.



This is another figure from the paper exploring the impact of training dataset size on model performance. Here, the compute budget and model size are held fixed, and the size of the training dataset varies. The graph shows that as the training data volume increases, the model's performance continues to improve.

The second graph holds the compute budget and training dataset size constant. Models of varying numbers of parameters are trained. As the model increases in size, the test loss decreases, indicating better performance.



▼ The optimal model in Chinchilla paper

So, what's the ideal balance between these three quantities? Well, it turns out many people are interested in this question. Both research and industry communities have published much empirical data for pre-training compute optimal models.

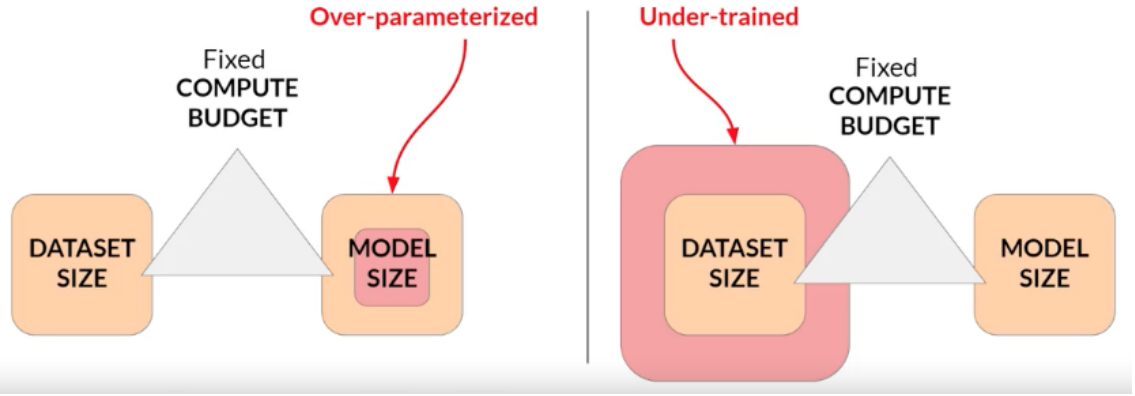
Training Compute-Optimal Large Language Models

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

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

Chinchilla paper, Jordan et al. 2022

In a paper published in 2022, a group of researchers led by Jordan Hoffmann, Sebastian Borgeaud and Arthur Mensch carried out a detailed study of the performance of language models of various sizes and quantities of training data. The goal was to find the optimal number of training data parameters and volume for a given compute budget. The author states that the optimal model is Chinchilla. This paper is often referred to as the Chinchilla paper.



Compute optimal models: Very large models may be over-parameterized and under-trained

Let's take a look at some of their findings. The Chinchilla paper hints that many of the 100 billion-parameter large language models like GPT-3 may actually be over-parameterized, meaning they have more parameters than they need to achieve a good understanding of language and under-trained so that they would benefit from seeing more training data. The authors hypothesized that smaller models may achieve the same performance as larger ones if trained on larger datasets.

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

Compute optimal training datasize
is ~**20x** number of parameters

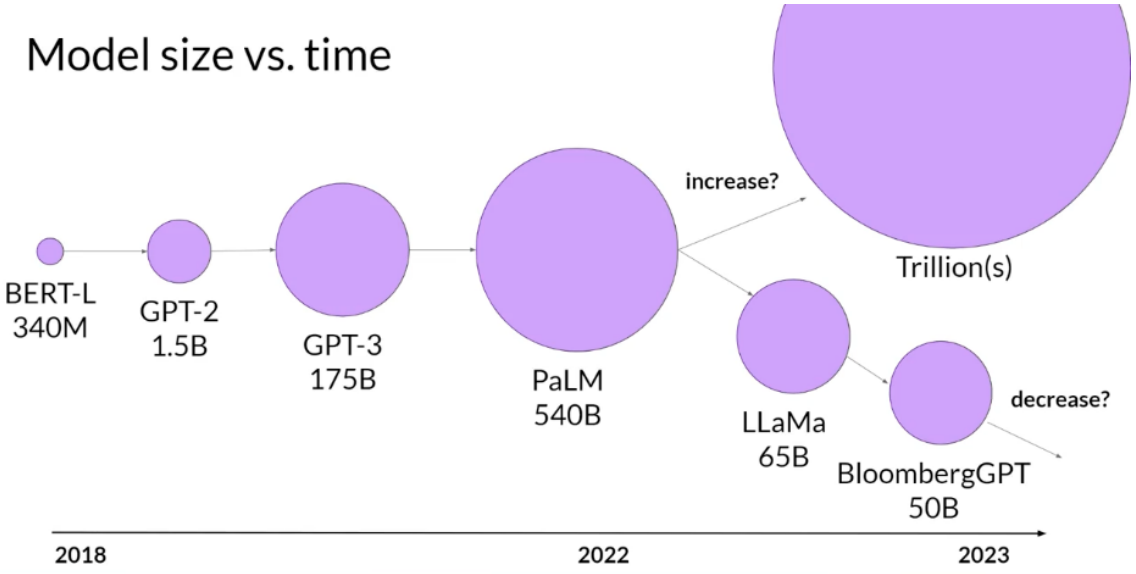
Chinchilla scaling laws for model and dataset size, Sources: Hoffmann et al. 2022, "Training Compute-Optimal Large Language Models" Touvron et al. 2023, "LLaMA: Open and Efficient Foundation Language Models", * assuming models are trained to be compute-optimal per Chinchilla paper

In this table, you can see a selection of models, their size, and information about the dataset they were trained on. One important takeaway from the Chinchilla paper is that the optimal training dataset size for a given model is about 20 times larger than the number of parameters in the model. Chinchilla was determined to be computationally optimal. For a 70 billion parameter model, the ideal training dataset contains 1.4 trillion tokens or 20 times the number of parameters.

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

In contrast, LLaMA was trained on a dataset size of 1.4 trillion tokens, close to the Chinchilla recommended number. Another important result from the paper is that the compute-optimal Chinchilla model outperforms non-compute-optimal models such as GPT-3 on a wide range of downstream evaluation tasks.

With the results of the Chinchilla paper in hand, teams have recently started developing smaller models that achieved similar, if not better, results than larger models that were trained in a non-optimal way. Moving forward, we can probably expect to see a deviation from the bigger, which has always been a better trend in the last few years as more teams or developers like you start to optimize their model design.



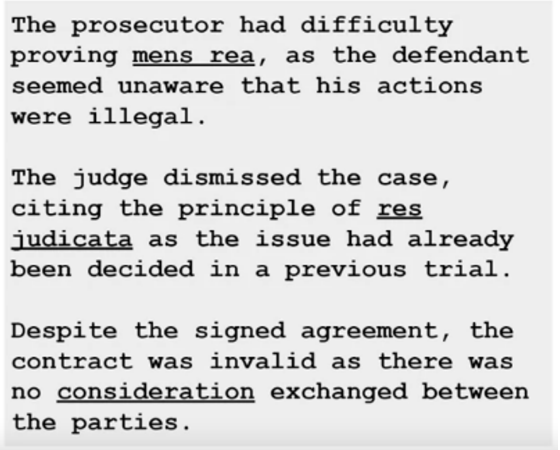
The last model shown on this slide, Bloomberg GPT, is interesting. It was trained in a compute optimal way following the Chinchilla loss and achieved good performance with 50 billion parameters. It's also an interesting example of a situation where pre-training a model from scratch was necessary to achieve good task performance.

▼ Pre-training for domain adaptation

Working with an existing LLM helps save a lot of time and can get to a working prototype much faster. However, there's one situation where it’s necessary to pre-train the model from scratch.

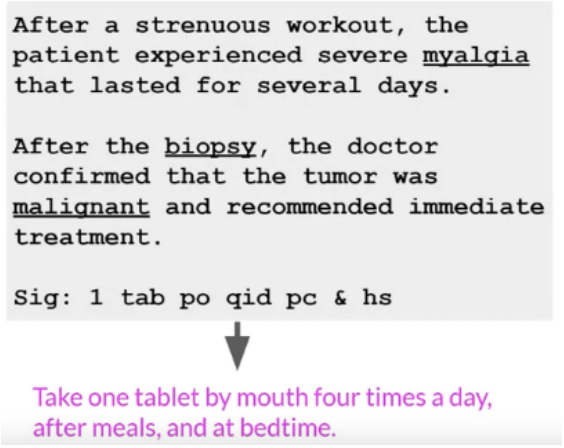
If the target domain uses vocabulary and language structures not commonly used in daily language, we may need domain adaptation to achieve good model performance.

For example, building an app to help lawyers and paralegals summarize legal briefs. Legal writing uses specific terms like “mens rea” in the first example and “res judicata” in the second. These words are rarely used outside of the legal world, which means they are unlikely to have appeared widely in the training text of existing LLMs. As a result, the models may have difficulty understanding or using these terms correctly.



Legal language

Another issue is that legal language sometimes uses everyday words in a different context, like consideration in the third example, which has nothing to do with being nice. Instead, it refers to the main element of a contract that makes the agreement enforceable.



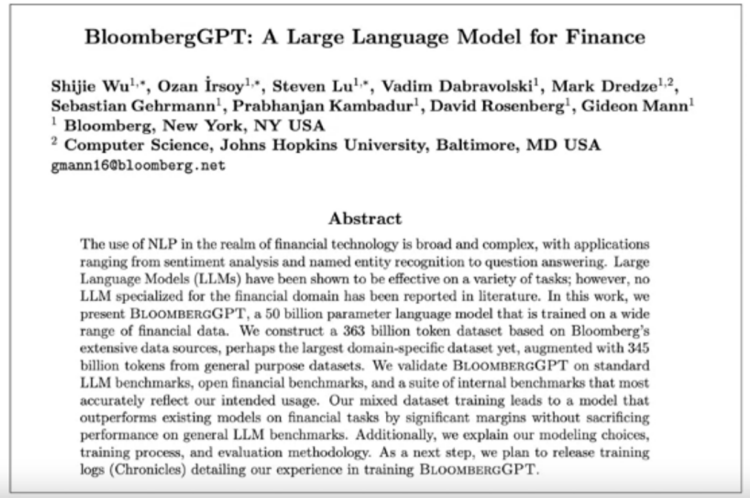
Medical language

For similar reasons, using an existing LLM in a medical application is challenging. The medical language contains many uncommon words to describe medical conditions and procedures. These may not appear frequently in training datasets consisting of web scrapes and book texts. Some domains also use language in a highly idiosyncratic way.

This last example of medical language may look like a string of random characters, but it's a shorthand doctors use to write prescriptions. This text means pharmacists should take one tablet by mouth four times a day, after meals, and at bedtime.

Because models learn their vocabulary and understanding of language through the original pretraining task, pretraining the model from scratch will result in better models for highly specialized domains like law, medicine, finance or science.

Return to BloombergGPT, first announced in 2023 in a paper by Shijie Wu, Steven Lu, and colleagues at Bloomberg. BloombergGPT is an example of a large language model pre-trained for a specific domain, in this case, finance. The Bloomberg researchers combined finance and general purpose tax data to pre-train a model that achieves Bestinclass results on financial benchmarks. While also maintaining competitive performance on general-purpose LLM benchmarks.



BloombergGPT: A Large Language Model for Finance

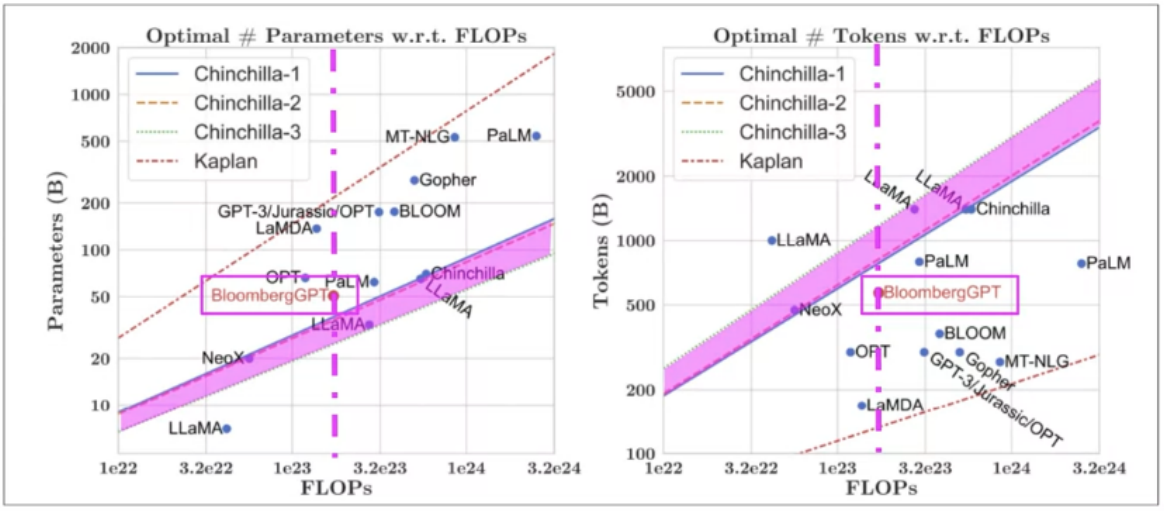
Shijie Wu[1,*], Ozan İrsoy[1,*], Steven Lu[1,*], Vadim Dabravolski[1], Mark Dredze[1,2], Sebastian Gehrmann[1], Prabhanjan Kambadur[1], David Rosenberg[1], Gideon Mann[1]
[1] Bloomberg, New York, NY USA
[2] Computer Science, Johns Hopkins University, Baltimore, MD USA
gmann16@bloomberg.net

Abstract

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

BloombergGPT: domain adaptation for finance

The researchers chose 51% financial data and 49% public data. In their paper, the Bloomberg researchers describe the model architecture in more detail. They also discuss how they started with chinchilla scaling laws for guidance and where they had to make tradeoffs.



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

These two graphs compare several LLMs, including BloombergGPT, to scaling laws that researchers have discussed. On the left, the diagonal lines trace the optimal model size in billions of parameters for a range of computing budgets. On the right, the lines trace the computed optimal training data set size measured in the number of tokens.

The dashed pink line on each graph indicates the compute budget the Bloomberg team had available for training their new model. The pink-shaded regions correspond to the computed optimal scaling loss determined in the Chinchilla paper.

Regarding model size, BloombergGPT roughly follows the Chinchilla approach for the given compute budget of 1.3 million GPU hours, or roughly 230,000,000 petaflops. The model is slightly above the pink-shaded region, suggesting the number of parameters is fairly close to optimal. However, the number of tokens used to pre-train BloombergGPT 569B is below the recommended Chinchilla value for the available compute budget.

The smaller-than-optimal training data set is due to the limited availability of financial domain data. Showing that real-world constraints may force us to make trade-offs when pre-training our models.

▼ Reading: Domain-specific training: BloombergGPT

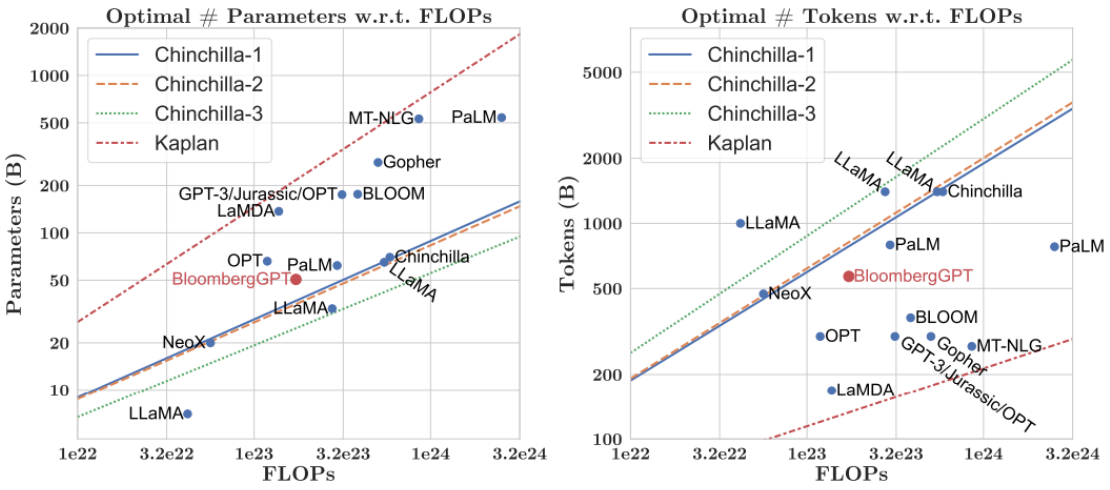


Figure 1: Kaplan et al. (2020) and Chinchilla scaling laws with prior large language model and BLOOMBERGGPT parameter and data sizes. We adopt the style from Hoffmann et al. (2022).

2 Dataset

- 2.1 Financial Datasets (363B tokens – 54.2% of training)
  - 2.1.1 Web (298B tokens – 42.01% of training) . . .
  - 2.1.2 News (38B tokens – 5.31% of training) . . . .
  - 2.1.3 Filings (14B tokens – 2.04% of training) . . .
  - 2.1.4 Press (9B tokens – 1.21% of training) . . . .
  - 2.1.5 Bloomberg (5B tokens – 0.70% of training) .
- 2.2 Public Datasets (345B tokens – 48.73% of training) .
  - 2.2.1 The Pile (184B tokens – 25.9% of training) .
  - 2.2.2 C4 (138B tokens – 19.48% of training) . . . .
  - 2.2.3 Wikipedia (24B tokens – 3.35% of training) .

BloombergGPT, developed by Bloomberg, is a large Decoder-only language model. It underwent pre-training using an extensive financial dataset comprising news articles, reports, and market data to increase its understanding of finance and enable it to generate finance-related natural language text. The datasets are shown in the image above.

During BloombergGPT's training, the authors used the Chinchilla Scaling Laws to guide the number of parameters in the model and the volume of training data measured in tokens. The Chinchilla recommendations are represented by the lines Chinchilla-1, Chinchilla-2, and Chinchilla-3 in the image, and BloombergGPT is close to them.

While the recommended configuration for the team's available training compute budget was 50 billion parameters and 1.4 trillion tokens, acquiring 1.4 trillion tokens of training data in the finance domain proved challenging. Consequently, they constructed a dataset containing just 700 billion tokens, less than the compute-optimal value. Furthermore, the training process terminated after processing 569 billion tokens due to early stopping.

The BloombergGPT project is a good illustration of pre-training a model for increased domain specificity and the challenges that may force trade-offs against compute-optimal models and training configurations.

Read the BloombergGPT article here.

▼ Week 1 quiz

1. Question 1

Interacting with Large Language Models (LLMs) differs from traditional machine learning models. Working with LLMs involves natural language input, known as a _____, resulting in output from the Large Language Model, known as the _____ .

Choose the answer that correctly fills in the blanks. (1 point)

○ **prediction request, prediction response**
● **prompt, completion**
○ **tunable request, completion**
○ **prompt, fine-tuned LLM**

2. Question 2

Large Language Models (LLMs) are capable of performing multiple tasks supporting a variety of use cases. Which of the following tasks supports the use case of converting code comments into executable code? (1 point)

● **Translation**
○ **Information Retrieval**
○ **Invoke actions from text**
○ **Text summarization**

3. Question 3

What is the *self-attention* that powers the transformer architecture? (1 point)

● **A mechanism that allows a model to focus on different parts of the input sequence during computation.**
○ **A measure of how well a model can understand and generate human-like language.**

○ **A technique used to improve the generalization capabilities of a model by training it on diverse datasets.**

○ **The ability of the transformer to analyze its own performance and make adjustments accordingly.**

4. Question 4

Which of the following stages are part of the generative AI model lifecycle mentioned in the course? (Select all that apply)

1 point

☑ **Defining the problem and identifying relevant datasets.**

☑ **Deploying the model into the infrastructure and integrating it with the application.**

☑ **Selecting a candidate model and potentially pre-training a custom model.**

☐ **Performing regularization**

☑ **Manipulating the model to align with specific project needs.**

5. Question 5

"RNNs are better than Transformers for generative AI Tasks."
Is this true or false? (1 point)

○ **True**

● **False**

6. Question 6

Which transformer-based model architecture has the objective of guessing a masked token based on the previous sequence of tokens by building bidirectional representations of the input sequence. (1 point)

○ **Sequence-to-sequence**
○ **Autoregressive**
● **Autoencoder**

7. Question 7

Which transformer-based model architecture is well-suited to the task of text translation? (1 point)

● **Sequence-to-sequence**
○ **Autoencoder**
○ **Autoregressive**

8. Question 8

Do we always need to increase the model size to improve its performance? (1 point)

○ **True**
● **False**

9. Question 9

Scaling laws for pre-training large language models consider several aspects to maximize the performance of a model within a set of constraints and available scaling choices. Select all alternatives that should be considered for scaling when performing model pre-training. (1 point)

☑ **Compute budget: Compute constraints**

☑ **Dataset size: Number of tokens**

☐ **Batch size: Number of samples per iteration**

☑ **Model size: Number of parameters**

10. Question 10

"You can combine data parallelism with model parallelism to train LLMs."
Is this true or false? (1 point)

● **True**

○ **False**

▼ Reading: Week 1 resources

Below you'll find links to the research papers discussed in this weeks videos. You don't need to understand all the technical details discussed in these papers - **you have already seen the most important points and must answer the quizzes** in the lecture videos.

However, if you'd like to examine the original research more closely, you can read the papers and articles via the links below.

Generative AI Lifecycle

- **Generative AI on AWS: Building Context-Aware, Multimodal Reasoning Applications**—This O'Reilly book explores all phases of the generative AI lifecycle, including model selection, fine-tuning, adapting, evaluation, deployment, and runtime optimizations.

Transformer Architecture

- **Attention is All You Need** - This paper introduced the Transformer architecture with the core "self-attention" mechanism. This article was the foundation for LLMs.
- **BLOOM: BigScience 176B Model**—BLOOM is an open-source LLM with 176B parameters trained in an open and transparent way. This paper presents a detailed discussion of the dataset and process used to train the model. You can also see a high-level overview of the model here.
- **Vector Space Models** - Series of lessons from DeepLearning.AI's Natural Language Processing specialization discussing the basics of vector space models and their use in language modelling.

Pre-training and scaling laws

- **Scaling Laws for Neural Language Models is an** empirical study by researchers at OpenAI that explores the scaling laws for large language models.

Model architectures and pre-training objectives

- **What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?** - The paper examines modelling choices in large pre-trained language models and identifies the optimal approach for zero-shot generalization.
- **HuggingFace Tasks and Model Hub** - Collection of resources to tackle varying machine learning tasks using the HuggingFace library.
- **LLaMA: Open and Efficient Foundation Language Models** - Article from Meta AI proposing Efficient LLMs (their model with 13B parameters outperforms GPT3 with 175B parameters on most benchmarks)

Scaling laws and compute-optimal models

- **Language Models are Few-Shot Learners** This paper investigates the potential of few-shot learning in Large Language Models.
- **Training Compute-Optimal Large Language Models** Study from DeepMind to evaluate the optimal model size and number of tokens for training LLMs. Also known as "Chinchilla Paper".
- **BloombergGPT: A Large Language Model for Finance**—LLM trained specifically for the finance domain, a good example of which tried to follow chinchilla laws.

▼ Copyright Notice