Week 1- Introduction to LLMs and the generative AI project lifecycle

Learning Objectives

- Discuss model pre-training and the value of continued pre-training vs fine-tuning
- Define the terms Generative AI, large language models, prompt, and describe the transformer architecture that powers LLMs

- Describe the steps in a typical LLM-based, generative AI model lifecycle and discuss the constraining factors that drive decisions at each step of the model lifecycle
- Discuss computational challenges during model pre-training and determine how to efficiently reduce memory footprint
- Define the term scaling law and describe the laws that have been discovered for LLMs related to training dataset size, compute budget, inference requirements, and other factors.

Generative AI and LLMs

Generative AI is a subset of traditional machine learning. The machine learning models that underpin generative AI have learned these abilities by finding statistical patterns in massive datasets of content that humans originally generated

The more parameters the model has the more memory the model has and with that is capable of more complex tasks.

Prompts and completions

The way you interact with language models is quite different from other machine learning and programming paradigms.
In those cases, you write computer code with formalized syntax to interact with libraries and APIs. In contrast, large language models can take natural language or human written instructions and perform tasks much as a human would. ***The text that you pass to an LLM is known as a prompt.*** The ***space or memory that is available to the prompt is called the context window***, and this is typically large enough for a few thousand words but differs from model to model.
The prompt is passed to the model, then predicts the next words. **The output of the model is called a completion**, and the **act of using the model to generate text is known as inference**. The completion is comprised of the text contained in the original prompt, followed by the generated text.

LLM use cases and texts

The core technique is next-word generation, but has a lot more applications besides the chatbot like:

- Summarize
- Translate
- Code generation (natural language -> machine code)
- NER (name entity recognition)
- Augmented LLMs (RAGs)

Developers have discovered that as the ***scale of foundation models grows from hundreds of millions of parameters to billions, even hundreds of billions, the subjective understanding of language that a model possesses also increases.*** This language understanding stored within the parameters of the model is what processes, reasons, and ultimately solves the tasks you give it, ***but it's also true that smaller models can be fine-tuned to perform well on specific focused tasks.***

Text Generation before Transformers

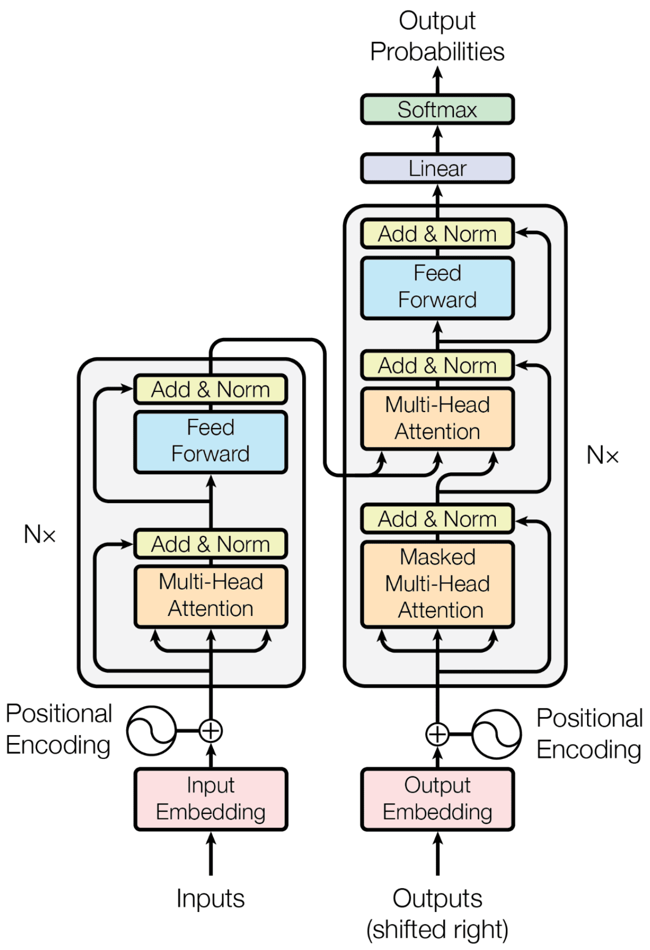
Where based on ***RNNs***. RNNs while powerful for their time, were limited by the amount of computing and memory needed to perform well at generative tasks

Seeing just a couple of previous orders the model can be wrong more times, the model should have the notion of all documents. And in some times the context and semantics can be hard even for humans.

Transformers model changed that

- This novel approach unlocked the progress in generative AI that we see today. It can be scaled efficiently to use multi-core GPUs, it can parallel process input data, making use of much larger training datasets, and crucially, it's able to learn to pay attention to the meaning of the words it's processing.

Transformers Architecture



The power of the transformer architecture lies in its ability to learn the relevance and context of all the words in a sentence. To apply attention weights to those relationships so that the model learns the relevance of each word to each other words no matter where they are in the input. This gives the algorithm the ability to learn who has the book, who could have the book, and if it's even relevant to the wider context of the document.

The transformer architecture is split into two distinct parts, the encoder and the decoder.

Machine-learning models are just big statistical calculators and they work with numbers, not words. So before passing texts into the model to process, you must first **tokenize** the words. Simply put, this converts the words into numbers, with each number representing a position in a dictionary of all the possible words that the model can work with.

Now that your input is represented as numbers, you can pass it to the **embedding layer**. This layer is a trainable vector embedding space, a high-dimensional space where each token is represented as a vector and occupies a unique location within that space. Each token ID in the vocabulary is matched to a multi-dimensional vector, and the intuition is that these vectors learn to encode the meaning and context of individual tokens in the input sequence. Embedding vector spaces have been used in natural language processing for some time, previous generation language algorithms like Word2vec use this concept. How you can calculate the distance between the words as an angle, which gives the model the ability to mathematically understand language

As you add the token vectors into the base of the encoder or the decoder, you also add **positional encoding.** The model processes each of the input tokens in parallel. So by adding the positional encoding, you preserve the information about the word order and **don't lose the relevance of the position of the word in the sentence.**

Once you've summed the input tokens and the positional encodings, you pass the resulting vectors to the **self-attention layer.** Here, the model analyzes the relationships between the tokens in your input sequence. As you saw earlier, this allows the model to attend to different parts of the input sequence to better capture the contextual dependencies between the words. The self-attention weights that are learned during training and stored in these layers reflect the importance of each word in that input sequence to all other words in the sequence.

But this does not happen just once, the transformer architecture has **multi-headed self-attention**. This means that multiple sets of self-attention weights or heads are learned in parallel independently of each other. The number of attention heads included in the attention layer varies from model to model, but numbers in the range of 12-100 are common. **The intuition here is that each self-attention head will learn a different aspect of language.**

Now that all of the attention weights have been applied to your input data, the output is processed through a **fully connected feed-forward network.**

The ***output of this layer is a vector of logits proportional to the probability score for every token in the tokenizer dictionary***. You can then pass these logits to a final softmax layer, where they are normalized into a probability score for each word.

Generating text with Transformers

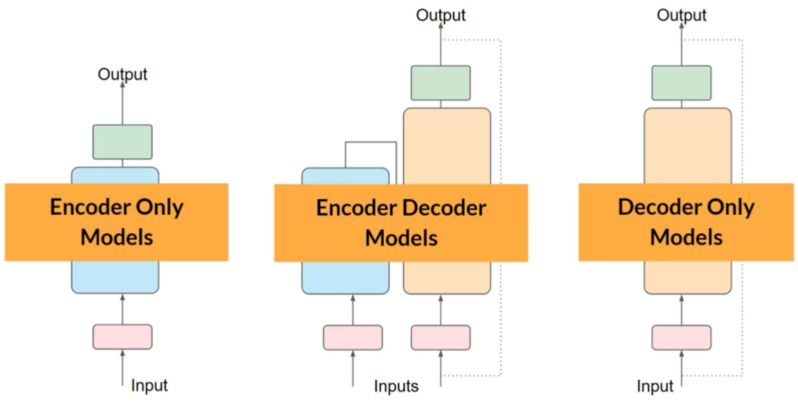
When the data *leaves* the encoder is a deep representation of the structure and meaning of the input sequence. This representation is inserted into the middle of the decoder to influence the decoder's self-attention mechanisms.

Next, a start of sequence token is added to the input of the decoder. This triggers the decoder to predict the next token, which it does based on the contextual understanding that it's being provided by the encoder. The output of the decoder's self-attention layers gets passed through the decoder feed-forward network and a final softmax output layer. At this point, we have our first token. The loop continues, passing the output token back to the input to trigger the generation of the next token, until the model predicts an end-of-sequence token. At this point, the final sequence of tokens can be detokenized into words, and you have your output.

The complete transformer architecture consists of an encoder and decoder components.

The ***encoder*** encodes input sequences into a deep representation of the structure and meaning of the input.

The ***decoder***, working from input token triggers, uses the encoder's contextual understanding to generate new tokens. It does this in a loop until some stop condition has been reached.



While the translation example you explored here used both the encoder and decoder parts of the transformer, you can split these components apart for variations of the architecture.

Encoder-only models also work as sequence-to-sequence models, but without further modification, the input sequence and the output sequence or the same length. Their use is less common these days, but by adding additional layers to the architecture, you can train encoder-only models to perform classification tasks such as sentiment analysis, ***BERT*** is an example of an encoder-only model.

Encoder-decoder models, as you've seen, perform well on ***sequence-to-sequence*** tasks such as translation, where the input sequence and the output sequence can be different lengths. You can also scale and train this type of model to perform general text generation tasks. Examples of encoder-decoder models include BART as opposed to BERT and T5, the model that you'll use in the labs in this course. Finally, decoder-only models are some of the most commonly used today. Again, as they have scaled, their capabilities have grown. These models can now generalize to most tasks.

Popular ***decoder-only*** models include the **GPT** family of models, **BLOOM**, **Jurassic**, **LLaMA**, and many more.

Prompting and Prompt Engineering

The text that you feed into the model is called the prompt, the act of generating text is known as inference, and the output text is known as the completion. The full amount of text or the memory that is available to use for the prompt is called the context window.

The work to develop and improve the prompt is known as prompt engineering.

However, one powerful strategy to get the model to produce better outcomes is to include examples of the task that you want the model to carry out inside the prompt. Providing examples inside the context window is called in-context learning.

ICL (In Context Learning)

Help LLMs learn more about the task being asked by including examples or additional data in the prompt.

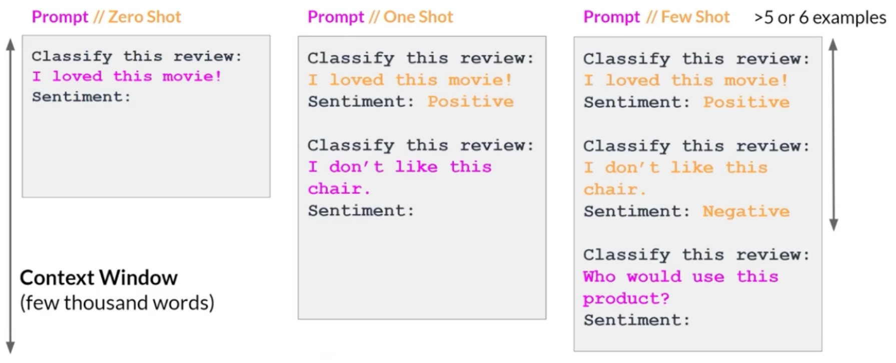
Here is a concrete example.

```
Classify this review:
I loved this movie!
Sentiment:
```

Within the prompt shown here, you ask the model to classify the sentiment of a review. So whether the review of this movie is positive or negative, the prompt consists of the instruction, "Classify this review," followed by some context, which in this case is the review text itself, and an instruction to produce the sentiment at the end. This method, **including your input data within the prompt**, is called ***zero-shot inference.*** Lager models are surprisingly good at this kind of task, but **smaller models,** like GPT-2, **do not** always perform well on this. **This is where providing an example within the prompt can improve performance.**

```
Classify this review:
I loved this movie!
Sentiment:Positive

Classify this review:
I don't like this chair.
Sentiment:
```

The inclusion of a single example is known as **one-shot inference**, in contrast to the zero-shot prompt you supplied earlier. Sometimes a single example won't be enough for the model to learn what you want it to do. So you can extend the idea of giving a single example to include multiple examples. This is known as **few-shot inference**.



```
Classify this review:
I loved this movie!
Sentiment:
```

```
Classify this review:
I loved this movie!
Sentiment: Positive

Classify this review:
I don't like this
chair.
Sentiment:
```

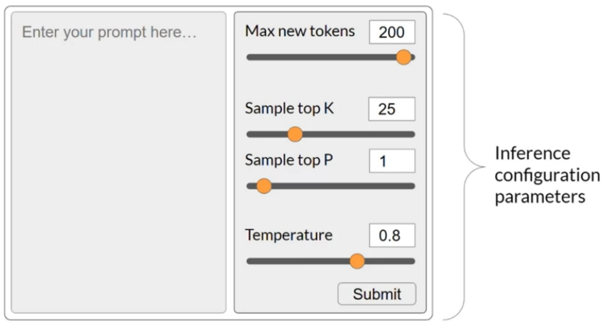
```
Classify this review:
I loved this movie!
Sentiment: Positive

Classify this review:
I don't like this
chair.
Sentiment: Negative

Classify this review:
Who would use this
product?
Sentiment:
```

Generally, if you find that your model isn't performing well when, say, including five or six examples, you should try ***fine-tuning your model instead***. Fine-tuning performs additional training on the model using new data to make it more capable of the task you want it to perform.

As larger and larger models have been trained, it's become clear that the ability of models to perform multiple tasks and how well they perform those tasks depends strongly on the scale of the model.
Models with more parameters can capture more understanding of language. The largest models are surprisingly good at zero-shot inference and can infer and complete many tasks that they were not specifically trained to perform. In contrast, smaller models are generally only good at a small number of tasks. Typically, those that are similar to the task that they were trained on.

Generative Configuration

Inference Parameters



Max new tokens 200
Sample top K 25
Sample top P 1
Temperature 0.8

Each model exposes a set of configuration parameters that can influence the model's output during inference. Note that these ***are different than the training parameters*** which are learned during training time.
Instead, these configuration parameters are invoked at inference time and give you control over things like the maximum number of tokens in the completion, and how creative the output is.
Max new tokens are probably the simplest of these parameters, and you can use them to ***limit the number of tokens that the model will generate.*** You can think of this as putting a cap on the number of times the model will go through the selection process. Here you can see examples of max new tokens being set to 100, 150, or 200. Remember it's max new tokens, not a hard number of new tokens generated, the model can predict the end-of-sequence before the max number number of tokens.

Most large language models by default will operate with so-called **greedy decoding.** This is the simplest form of next-word prediction, where the ***model will always choose the word with the highest probability***. This method can work very well for short generations but is susceptible to *repeated* words or *repeated sequences* of words. If you want to generate text that's more natural, more creative and avoids repeating words, you need to use some other controls.
Random sampling *is the easiest way to introduce some variability*. Instead of selecting the most probable word every time with random sampling, the model chooses an output word at random using the probability distribution to weight the selection.
Let's explore ***top k*** and ***top p*** sampling techniques to help limit the random sampling and increase the chance

that the output will be sensible. Two Settings, top p and top k are sampling techniques that we can use to help limit the random sampling and increase the chance that the output will be sensible.

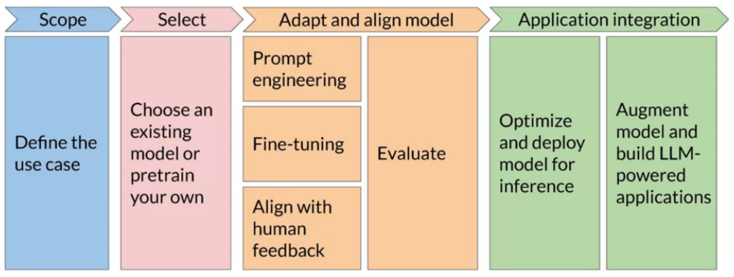
- ***To limit the options while still allowing some variability,*** you can specify a ***top k*** value which instructs the model to ***choose from only the k tokens with the highest probability***. This method can help the model have some randomness while preventing the selection of highly improbable completion words. This in turn makes your text generation more likely to sound reasonable and to make sense.
- Alternatively, you can use the ***top p*** setting to limit the random sampling to the predictions whose combined probabilities are not lower than p. The model then uses the random probability weighting method to choose from these tokens.

> *With top k, you specify the number of tokens to randomly choose from, and with top p, you specify the total probability that you want the model to choose from.*

One more parameter that you can use to control the ***randomness*** of the model output is known as ***temperature***. This parameter influences the shape of the probability distribution that the model calculates for the next token. Broadly speaking, the higher the temperature, the higher the randomness, and the lower the temperature, the lower the randomness. The temperature value is a scaling factor that's applied within the final softmax layer of the model that impacts the shape of the probability distribution of the next token. In contrast to the top k and top p parameters, changing the temperature actually alters the predictions that the model will make.

If you choose a ***low value of temperature,*** *say less than one, the resulting probability distribution from the softmax layer is more strongly peaked with the* ***probability being concentrated in a smaller number of words.*** *If instead you set the* ***temperature to a higher value****, say, greater than one, then the model will calculate a* ***broader flatter probability distribution*** *for the next token.*

Generative AI project lifecycle


Scope
Select
Adapt and align model
Application integration
Define the use case
Choose an existing model or pretrain your own
Prompt engineering
Fine-tuning
Align with human feedback
Evaluate
Optimize and deploy model for inference
Augment model and build LLM-powered applications

The most important step in any project is to ***define the scope as accurately and narrowly as you can.*** Do you need the model to be able to carry out many different tasks, including long-form text generation or with a high degree of capability, or is the task much more specific like named entity recognition so that your model only needs to be good at one thing.

Once you're happy, and you've scoped your model requirements enough to begin development. **Your first decision will be whether to train your own model from scratch or work with an existing base model**. In general, you'll start with an existing model, although there are some cases where you may find it necessary to train a model from scratch.

With your model in hand, the next step is to **assess its performance** and carry out additional training if needed for your application. *Prompt engineering* can sometimes be enough to get your model to perform well, so you'll likely start by trying in-context learning, using examples suited to your task and use case. There are still cases, however, where the model may not perform as well as you need, even with *one or a few short inferences,* and in that case, you can try *fine-tuning* your model.
As models become more capable, it's becoming increasingly important to ensure that they behave well and in a way that is ***aligned with human preferences*** in deployment. An important aspect of all of these techniques is ***evaluation***, this can be done with some metrics and benchmarks that can be used to determine how well your model is performing or how well aligned it is to your preferences.

Finally, when you've got a model that meets your performance needs and is well aligned, you can ***deploy*** it into your infrastructure and integrate it with your application. At this stage, an important step is to optimize your model for deployment.

The last but very important step is to consider any ***additional infrastructure*** that your application will require to work well.

Lab1 - Summarize Dialogue

Tools:

- SageMaker
- Torch
- Hugging Face
  - Transformers
  - datasets
- Techniques
  - Prompt Eng
  - Try out inference parameters
- `Model`: pre-trained Large Language Model (LLM) FLAN-T5 from Hugging Face (base)
  - Load the FLAN-T5 model, creating an instance of the `AutoModelForSeq2SeqLM` class with the `.from_pretrained()` method.
- Dataset: Dialogsum
  - This dataset contains 10,000+ dialogues with the corresponding manually labelled summaries and topics.
- Main Tasks

    - Check the model output with
        - Zero-Shot inference
        - One-shot inference
        - Few Shot inference
    - Change the inference parameters and see how different the output
        - Putting the parameter `do_sample = True`, you activate various decoding strategies which influence the next token from the probability distribution over the entire vocabulary. You can then adjust the outputs changing `temperature` and other parameters (such as `top_k` and `top_p`).
        - `max_new_tokens=X, do_sample=True/False, temperature=Y`
- In this case, the model output improved with a one-shot inference but the addition of more examples on the prompt(few-shot inference) doesn't improve the output that much.
    - This is a very inexpensive way to try out these models and even figure out which model should you fine-tune. We chose Plan T5 because it works across a large number of tasks. But if you have no idea how a model is you just get it off of some model hub somewhere. These are the first step. Prompt engineering, zero-shot, one-shot, few-shot is almost always the first step when you're trying to learn the language model that you've been handed and the dataset.
    - We see a case where the few-shot didn't do much better than the one-shot. This is something that you want to pay attention to because, in practice, people often try to just keep adding more and more shots, five shots, six shots. Typically, in my experience, above five or six shots, so full prompt and then completions, you don't gain much after that. Either the model can do it or it can't do it and goes about five or six.

Lab steps

Load the FLAN-T5 model, creating an instance of the `AutoModelForSeq2SeqLM` class with the `.from_pretrained()` method.

To perform encoding and decoding, you need to work with text in a tokenized form. **Tokenization** is the process of splitting texts into smaller units that can be processed by the LLM models.

- Download the tokenizer for the FLAN-T5 model using `AutoTokenizer.from_pretrained()` method. Parameter `use_fast` switches on fast tokenizer. At this stage, there is no need to go into the details of that, but you can find the tokenizer parameters in the documentation.

LLM pre-trained and scaling laws

Once you have scoped out your use case, and determined how you'll need the LLM to work within your application, your next step is to select a model to work with. Your first choice will be to either work with an existing model or train your own from scratch. There are specific circumstances where training your own model from scratch might be advantageous, and you'll learn about those later in this lesson. In general, however, you'll begin the process of developing your application using an existing foundation model. Many open-source models are available for members of the AI community like you to use in your application.

The developers of some of the major frameworks for building generative AI applications like Hugging Face and PyTorch, have curated hubs where you can browse these models. A really useful feature of these hubs is the inclusion of model cards, that describe important details including the best use cases for each model, how it was trained, and known limitations.

Pre-training large language models

Model architectures and pre-training objectives

LLMs encode a deep statistical representation of language. This understanding is developed during the models' pre-training phase when the model learns from vast amounts of unstructured textual data. This 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 that have been 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 architecture of the model, as you'll see shortly. During pre-training, the model weights get updated to minimize the loss of the training objective.

The encoder generates an embedding or vector representation for each token. Pre-training also requires a large amount of computing and the use of GPUs

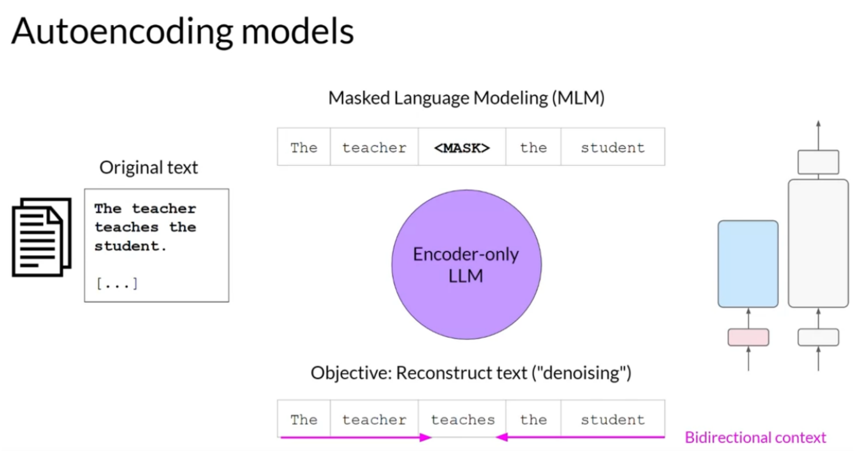
Note, when you scrape training data from public sites such as the Internet, you often need to process the data to increase quality, address bias, and remove other harmful content. As a result of this data quality curation, often only 1-3% of tokens are used for pre-training.

Earlier this week, you saw that there were three variances of the transformer model; encoder-only encoder-decoder models, and decode-only. Each of these is trained on a different objective, and so learns how to carry out different tasks.

Encoder-only models are also known as ***Autoencoding models***, and they are pre-trained using masked language modelling. Here, tokens in the input sequence or randomly masked, and the training objective is to predict the mask tokens in order to reconstruct the original sentence. This is also called a denoising objective. Autoencoding models spilt bi-directional representations of the input sequence, meaning that the model has an understanding of the full context of a token and not just of the words that come before. Encoder-only models are ideally suited to tasks that benefit from this bi-directional context.

- Sentiment Analysis (sentence level class)
- Named Entity Recognition (NER) (token level class)
- Word Classification

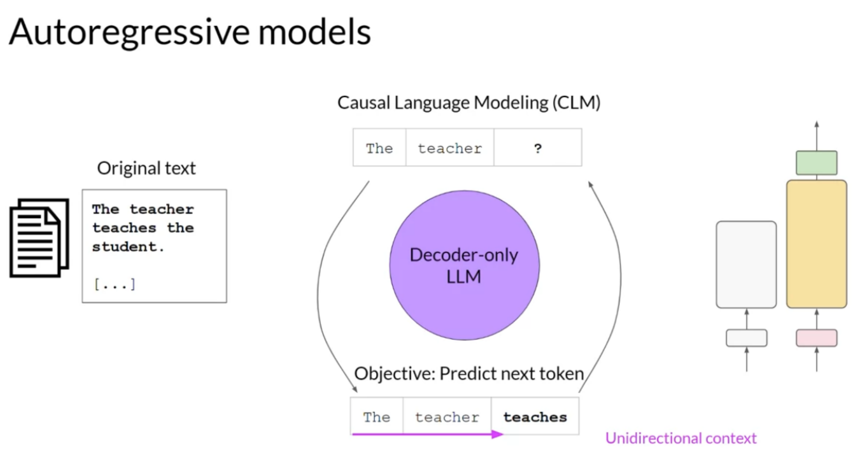
Some well-known examples of an autoencoder model are *BERT* and *RoBERTa*.


Autoencoding models
Masked Language Modeling (MLM)
The teacher <MASK> the student
Original text
The teacher teaches the student.
[...]
Encoder-only LLM
Objective: Reconstruct text ("denoising")
The teacher teaches the student
Bidirectional context

Decoder-only or autoregressive models, which are pre-trained using causal language modelling. Here, the training objective is to predict the next token based on the previous sequence of tokens. *Predicting the next token is sometimes called full language modelling by researchers.*
Decoder-based autoregressive models, mask the input sequence and can only see the input tokens leading up to the token in question. The model has no knowledge of 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, this means that the context is unidirectional.
By learning to predict the next token from a vast number of examples, the model builds up a statistical representation of language. Models of this type make use of 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*.

Good use cases:

- Text Generation
- Other emergent behavior
  - Depends on model size



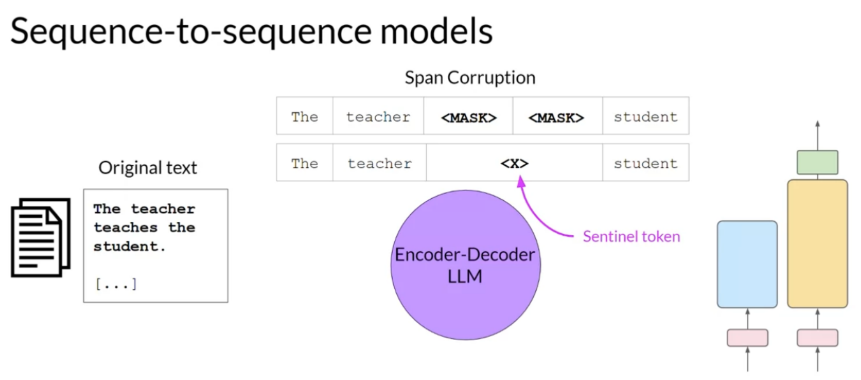
The final variation of the transformer model is the ***sequence-to-sequence model*** that uses both the encoder and decoder parts of the original transformer architecture. The exact details of the pre-training objective vary from model to model. A popular sequence-to-sequence model T5, pre-trains the encoder using span corruption, which masks random sequences of input tokens. Those mass sequences are then 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.

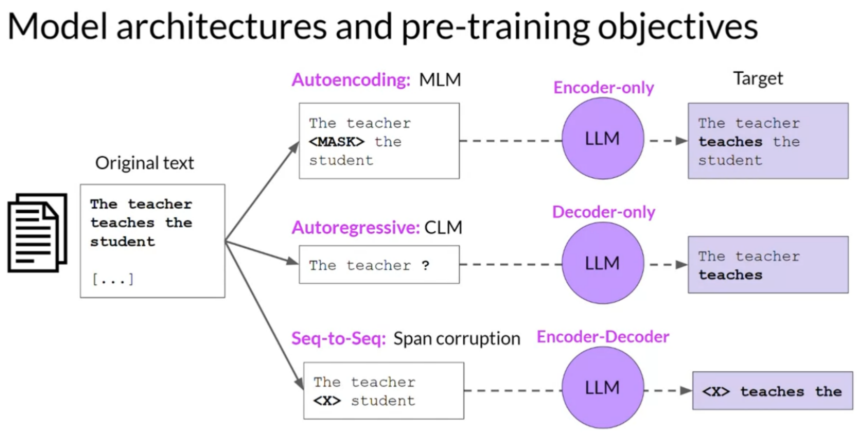
Good use cases: (They are generally useful in cases where you have a body of texts as both input and output)

- Translation
- Text summarization
- Question answering

Example models:

- T5
- BART





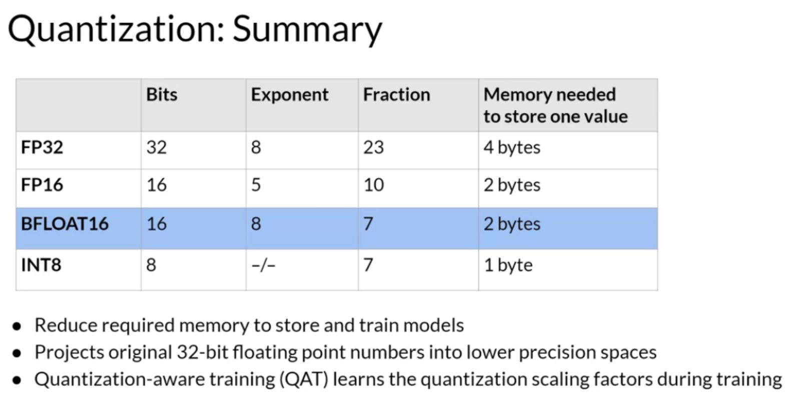
One additional thing to keep in mind is that larger models of any architecture are typically more capable of carrying out their tasks well. Researchers have found that the larger a model, the more likely it is to work as you needed to without additional in-context learning or further training. This observed trend of increased model capability with size has driven the development of larger and larger models in recent years. This growth has been fueled by inflection points and research, such as the introduction of the highly scalable transformer architecture, access to massive amounts of data for training, and the development of more powerful computing resources. This steady increase in model size led some researchers to *hypothesize the existence of a new Moore's law for LLMs.* Like them, we may be asking, can we just keep adding parameters to increase performance and make models smarter? Where could this model growth lead? While this may sound great, it turns out that training these enormous models is difficult and very expensive, so much so that it may be infeasible to continuously train larger and larger models.

Computational challenges

One technique that you can use to reduce the memory is called **quantization**. The main idea here is that you 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. The corresponding data types used in deep learning frameworks and libraries are FP32 for 32-bit full position, FP16, or Bfloat16 for 16-bit half-precision, and int8 eight-bit integers. The range of numbers you can represent 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.

One datatype in particular 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. BFLOAT16 is often described as a truncated 32-bit float, as it captures the full dynamic range of the full 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 not only saves memory but also increases model performance by speeding up calculations. The downside is that BF16 is not well suited for integer calculations, but these are relatively rare in deep learning.

Let's summarize what you've learned here and emphasize the key points you should take away from this discussion. Remember that the goal of quantization is to reduce the memory required to store and train models by reducing the precision off 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 the training process.

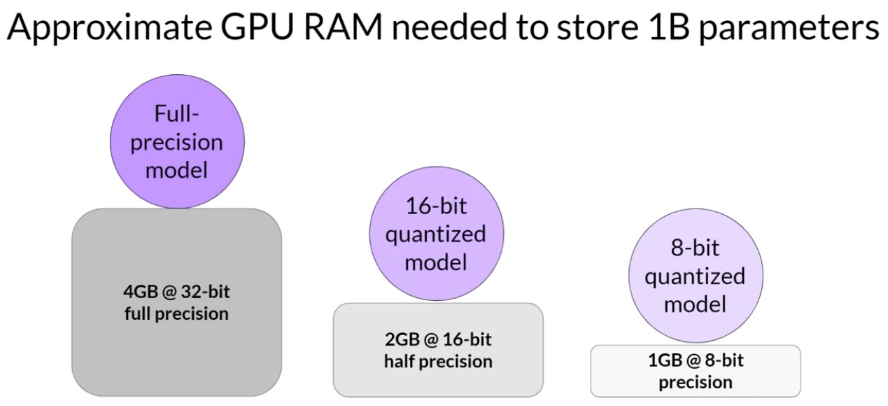


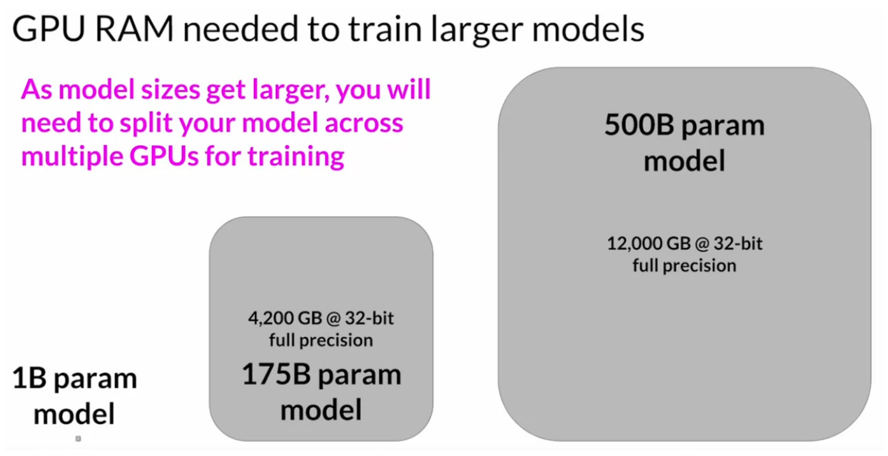
Quantization: Summary

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

- 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

By applying quantization, you can reduce your memory consumption required to store the model parameters down to only two gigabytes using 16-bit half-precision of 50% saving and you could further reduce the memory footprint by another 50% by representing the model parameters as eight-bit integers, which requires only one gigabyte of GPU RAM. Note that in all these cases you still have a model with one billion parameters.

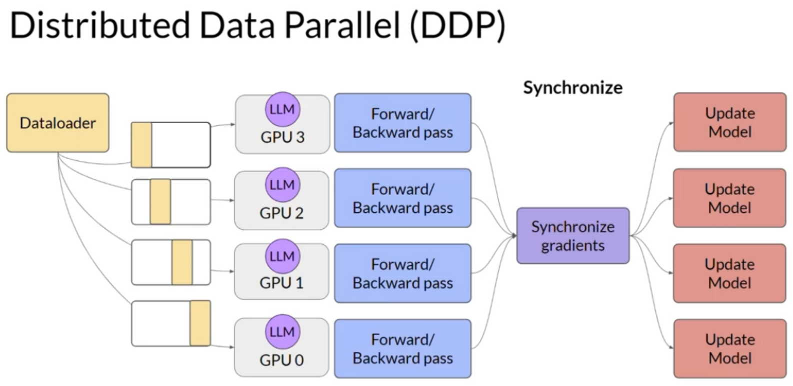

Approximate GPU RAM needed to store 1B parameters
Full-precision model
4GB @ 32-bit full precision
16-bit quantized model
2GB @ 16-bit half precision
8-bit quantized model
1GB @ 8-bit precision


GPU RAM needed to train larger models
As model sizes get larger, you will need to split your model across multiple GPUs for training
500B param model
12,000 GB @ 32-bit full precision
4,200 GB @ 32-bit full precision
175B param model
1B param model

Efficient multi-GPU compute strategies

Distributed Data-Parallel (DDP)

The first 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 Pi torches distributed data-parallel, or DDP for short. DDP copies your model onto each GPU and sends batches of data to each of the GPUs 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. Note that DDP requires that your model weights and all of the additional parameters, gradients, and optimizer states that are needed for training, fit onto a single GPU.

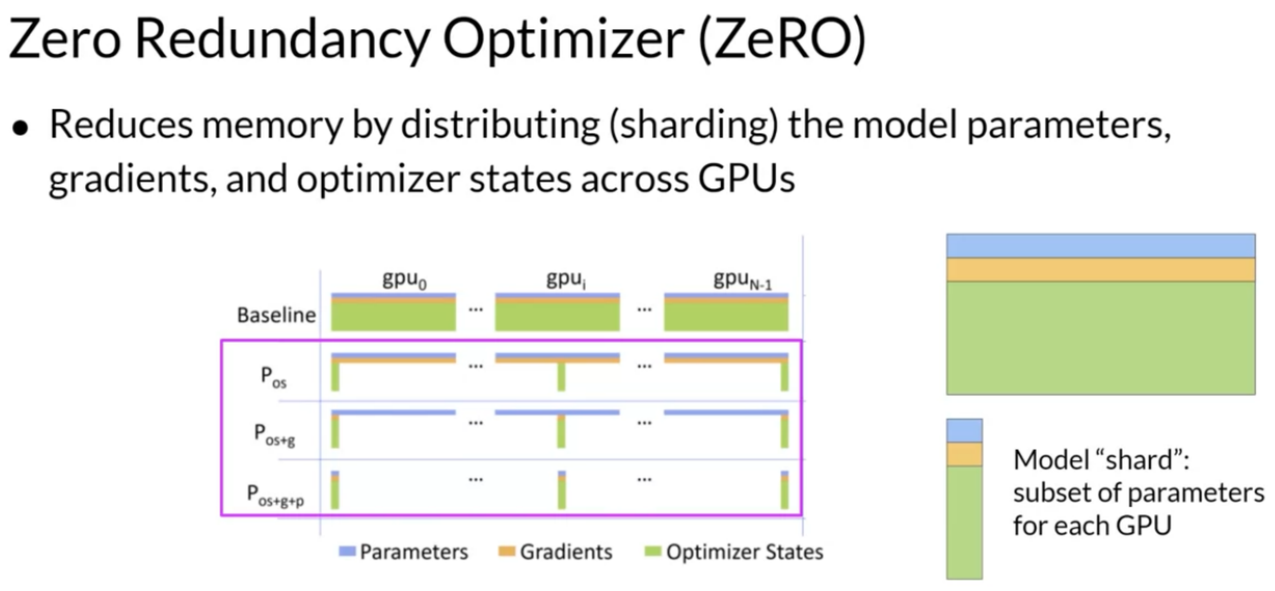


Modal Sharding

If your model is too big for this, you should look into another technique called ***modal sharding.***

A popular implementation of modal sharding is Pi Torch is fully sharded data parallel, or FSDP for short. 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 the goal of ZeRO is to optimize memory by distributing or sharding model states across GPUs with ZeRO data overlap. This allows you to scale model training across GPUs when your model doesn't fit in the memory of a single chip.

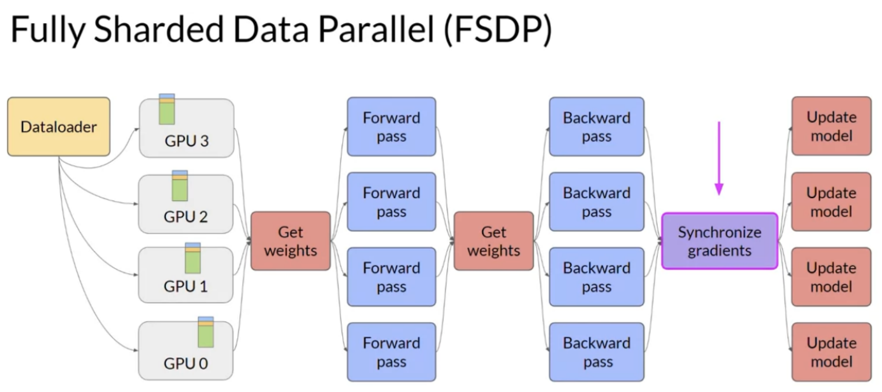
Earlier this week, you looked at all of the memory components required for training LLMs, the largest memory requirement was for the optimizer states, which take up twice as much space as the weights, followed by the weights themselves and the gradients. Let's represent the parameters as this blue box, the gradients and yellow and the optimizer states in green. One limitation of the model replication strategy that I showed before is that you need to keep a full model copy on each GPU, which leads to redundant memory consumption. You are storing the same numbers on every GPU. 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 previously discussed ADP. ZeRO offers three optimization stages. ZeRO Stage 1, shots only optimizer states across GPUs, this can reduce your memory footprint by up to a factor of four. ZeRO Stage 2 also shots the gradients across chips. When applied together with Stage 1, this can reduce your memory footprint by up to eight times. Finally, ZeRO Stage 3 shots all components including the model parameters across GPUs. When applied together with Stages 1 and 2, memory reduction is linear with the number of GPUs. For example, sharding across 64 GPUs could reduce your memory by a factor of 64. Let's apply this concept to the visualization of GDP and replace the LLM by the memory representation of model parameters, gradients, and optimizer states.



Zero Redundancy Optimizer (ZeRO)

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

Fully Shared Data Parallell (FSDP)



Fully Sharded Data Parallel (FSDP)

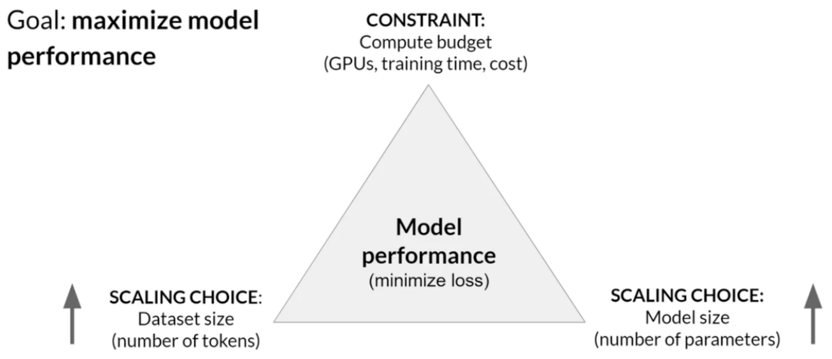
When you use FSDP, you distribute the data across multiple GPUs as you saw in GDP. But with FSDP, you also distributed or shard the model parameters, and gradients, and optimised the states across the GPU nodes using one of the strategies specified in the ZeRO paper. With this strategy, you can now work with models too big to fit on a single chip. In contrast to GDP, 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. Each CPU requests data from the other GPUs on-demand to materialise the sharded data into uncharted data for the duration of the operation. After the operation, you release the uncharted non-local data back to the other GPUs as original sharded data You can also choose to keep it for future operations during backward pass for example. Note, that this requires more GPU RAM again, this 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 allows you to reduce your overall GPU memory utilization. Optionally, you can specify that FSDP offloads part of the training computation to GPUs to reduce your GPU memory utilization further. To manage the trade-off between performance and memory utilisation, you can configure the level of sharding using FSDP is a charting factor. A sharding factor of one removes the sharding and replicates the full model similar to DDP. If you set the sharding factor to the maximum available GPUs, you 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.

- Helps to reduce overall GPU memory utilization
- supports offloading to CPU if needed
- Configure the level of sharding via `sharding factor`

FSDP for both small and large models and seamlessly scale your model training across multiple GPUs..

Scaling laws and compute-optimal models

Scaling choices for pre-training


Goal: **maximize model performance**
CONSTRAINT: Compute budget (GPUs, training time, cost)
Model performance (minimize loss)
SCALING CHOICE: Dataset size (number of tokens)
SCALING CHOICE: Model size (number of parameters)

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 be able to achieve the same performance as much larger ones if they are trained on larger datasets. In this table, you can see a selection of models along with 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 compute optimal. For a 70 billion parameter model, the ideal training dataset contains 1.4 trillion tokens or 20 times the number of parameters.

You can probably expect to see a deviation from the bigger is always better trends of 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 really interesting. It was trained in a compute optimal way following the Chinchilla loss and so achieves good performance with the size of 50 billion parameters.

Pre-training for domain adaption

There's one situation where you may find it necessary to pre-train your model from scratch. If your target domain uses vocabulary and language structures that are not commonly used in day-to-day language. (Medical, legal, etc...)

BloombergGPT: domain adaptation for finance

51% Finance Data | 49% General Data