2 Background

2.1

Q: What are the main four major aspects of LLMs covered in [1]?

A:
pre-training, adaptation tuning, utilization, and capacity evaluation

a. pre-training (how to pretrain a capable LLM)

b. adaptation (how to effectively adapt pre-trained LLMs for better use)

c. utilization (how to use LLMs for solving various downstream tasks)

d. capability evaluation (how to evaluate the abilities of LLMs and existing empirical findings)

2.2

Q: What is the difference between PLM and LLM?

A:
The term large language models (LLM) represents for the PLMs of significant size (containing tens or hundreds of billions of parameters)

1. Large Language Models (LLMs) have emergent abilities that were not present in smaller pre-trained language models (PLMs). These abilities significantly improve their performance on complex tasks.

2. LLMs are set to change the manner in which humans develop and interact with AI algorithms. This impact is considerably different from the impact of smaller PLMs.

3. Access to LLMs is typically through a prompting interface, such as the GPT-4 API, which requires humans to learn how to effectively communicate with LLMs to maximize their potential.

4. The evolution of LLMs has blurred the lines between research and engineering because their development relies on practical skills in handling large-scale datasets and managing distributed, parallel training processes.

2.3

Q: What are the main three types of LLM architecture? What is the difference? Name a few examples for each type

A:

LLMs can be categorized into three major types, namely encoder-decoder, causal decoder, and prefix decoder.

a. Encoder-decoder Architecture

The initial Transformer model is built on the encoder-decoder architecture, containing two stacks of Transformer blocks as the encoder and decoder.

There are only a small number of LLMs that are built based on the encoder-decoder architecture, such as Flan-T5.

b. Causal Decoder Architecture

The causal decoder architecture incorporates the unidirectional attention mask, to guarantee that each input token can only attend to the past tokens and itself.

The causal decoders have been widely adopted as the architecture of LLMs by various existing LLMs, such as OPT, BLOOM, and Gopher.

c. Prefix Decoder Architecture

The prefix decoder architecture (non-causal decoder) revises the masking mechanism of causal decoders, to enable performing bidirectional attention over the prefix tokens and unidirectional attention only on generated tokens.

Examples are GLM-130B and U-PaLM.

2.4

Q: What is language modeling? What is the difference between causal language modeling and masked language modeling?

A:

LM aims to model the generative likelihood of word sequences, so as to predict the probabilities of future (or missing) tokens.

a. Causal Language Models (CLM) generate text in a forward direction, learning to predict each subsequent token based on the tokens that came before it.

b. Masked Language Models (MLM) can take into account the entire context, learning to understand and predict tokens based on all surrounding text, which allows them to better understand the meaning of words in context.

2.5

Q: What is a text classification task? What kind of model do we use for this task? Do you need to retrain a classification head for a new downstream task?

A:

Text classification task involves assigning predefined categories (or labels) to text. For example, an email might be classified as "spam" or "not spam," or a movie review might be categorized as expressing a "positive" or "negative" sentiment. The task is a fundamental one in the field of natural language processing (NLP), as it forms the basis for many applications that need to understand and organize text at scale.

To perform text classification, one would typically use a model that has been trained to understand language patterns. The types of models can include:

a. Traditional Machine Learning Models: Such as Naive Bayes, Support Vector Machines (SVM), or Random Forests, which often rely on bag-of-words or TF-IDF (Term Frequency-Inverse Document Frequency) features.

b. Neural Network Models: Such as Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs), including their variants like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) networks.

c. Transformer-based Models: Modern approaches often utilize transformer-based architectures like BERT (Bidirectional Encoder Representations from Transformers), RoBERTa, or GPT (Generative Pre-trained Transformer) models.

For most modern applications, transformer-based models are preferred due to their superior ability to capture complex language patterns and context. These models are pre-trained on large corpora of text in a self-supervised fashion and can then be fine-tuned on a specific text classification task.

2.6

Q: What is a summarization task? Sometimes it is also referred to as a sequence-to-sequence task. What kind of model do we use for this task?

A:

Text Summarization methods are grouped into two main categories: Extractive and Abstractive. Extractive Text Summarization, where the model “extracts” the most important sentences from the original text, is the more traditional method. Extractive Text Summarization does not alter the original language used in the text.

Here are some kinds of models can be used in summarization task.

a. TextRank algorithm. Google uses an algorithm called PageRank in order to rank web pages in their search engine results. TextRank implements PageRank in a specialized way for Text Summarization, where the highest “ranking” sentences in the text are the ones that describe it the best.

b. LexRank which works similarly to TextRank, utilizing the singular value decomposition of a word-sentence matrix.

c. T5. It is a Transformer based architecture that uses a text-to-text approach.

2.7

Q: Why Adam and AdamW are most commonly used optimizers for training LLMs? If training a model with N parameters, we know that the optimizer needs to store N gradients. Does Adam or AdamW introduces extra overhead?

A:

a. Adam and AdamW are widely used optimizers for training large language models (LLMs) due to several reasons:

1. Adaptive Learning Rates: Both Adam and AdamW use adaptive learning rates for each parameter. This means they compute individual learning rates for different parameters from estimates of first and second moments of the gradients. This feature is particularly useful for complex models like LLMs, where different parameters may require different learning strategies.
2. Efficient with Large Datasets and Parameters: They are efficient for large-scale optimization problems common in LLMs, where datasets and model sizes are substantial.
3. Balancing Exploration and Exploitation: These optimizers strike a balance between exploration of the parameter space (which SGD with momentum does) and exploitation (which is typical of algorithms that adapt the learning rate based on the variance).
4. Robustness: Adam and AdamW are more robust to hyperparameter settings, especially the learning rate, which makes them a preferred choice when fine-tuning LLMs.

Relationship: AdamW is a variant of Adam that modifies the weight decay component, decoupling it from the gradient updates. This can lead to better training dynamics and ultimately better performance on some tasks.

b. Both Adam and AdamW introduce extra overhead in terms of memory usage:

1. Adam optimizer stores:

   First moment estimates (the mean of the gradients), which is an array of size N.

   Second moment estimates (the uncentered variance of the gradients), which is another array of size N.

   Additionally, for each parameter, Adam keeps track of a time step counter to compute the bias-corrected first and second moment estimates.

2. AdamW optimizer stores the same information as Adam but handles the weight decay component differently during the parameter update step.

Thus, both optimizers require storage for at least 3N parameters (the original parameters, the first moment, and the second moment), not including the additional memory for the time step counter. This means that the memory requirements for Adam and AdamW are typically more than double that of simpler algorithms like stochastic gradient descent (SGD) which only needs to store the gradients and the parameters. However, despite the extra memory overhead, the benefits they provide for the convergence and performance of training large models usually outweigh the cost, particularly when using distributed training across multiple GPUs or TPUs where memory resources are abundant.

2.8

Q: What is a learning rate scheduler? Explain the behavior of torch.optim.lr scheduler.CosineAnnealingLR .

A:

a. A learning rate scheduler is a strategy or a method used during the training of neural networks to change the learning rate during training. The learning rate is a critical hyperparameter that can influence model performance and convergence rates. If it's too high, the training may not converge; if it's too low, training may take too long or get stuck in a local minimum.

b. Torch.optim.lr_scheduler.CosineAnnealingLR is one such learning rate scheduler provided by PyTorch. It implements a cosine annealing schedule, which is characterized by the following behavior:

   Initial Phase : The learning rate starts from a specified initial learning rate (typically higher).

Middle Phase : As training progresses, the learning rate decreases following a cosine curve from its initial value down to a minimum value. This part of the curve is like the first part of the cosine function, starting at the top of the wave and ending at the bottom.

End Phase : The learning rate then increases again following the curve, but this is often not used in practice since the idea is to decrease the learning rate to fine-tune the model's weights.

The CosineAnnealingLR scheduler can be visualized as a smooth curve resembling the shape of a cosine wave. The purpose of this is to make large adjustments to the learning rate at the beginning of training when we are far from the optimal weights and smaller adjustments as we get closer to the optimal weights. This allows for rapid learning at first but then fine-tuning as the model approaches convergence.

2.9

Q: What is tokenization? Name a few different tokenization methods. What is the tokenization method used by LLaMA?

A:

Tokenization is the process of converting a sequence of characters into a sequence of tokens. In the context of natural language processing (NLP), tokens are typically words, numbers, or punctuation marks. This process is a fundamental step in many NLP tasks because it structures the input text into a form that can be analyzed and used by algorithms.

Here are a few different tokenization methods:

1. Whitespace Tokenization : The simplest form of tokenization that splits text on whitespace. It is a naive approach and often does not suffice for complex text structures.
2. Dictionary-based Tokenization : Involves using a dictionary or a lexicon to identify tokens, especially useful for languages where whitespace is not used to demarcate words, like Chinese or Japanese.
3. Rule-based Tokenization : Uses a set of defined rules and regular expressions to identify tokens. This may involve looking for patterns in the text to split it into tokens, such as punctuation or capitalization.
4. Subword Tokenization : Breaks words into smaller units (subwords or characters), which can be beneficial for handling unknown words, morphologically rich languages, or agglutinative languages where words are constructed by combining multiple morphemes.
   - Byte Pair Encoding (BPE) : Starts with a large corpus of text and iteratively merges the most frequent pair of bytes (or characters) to create common subword units.
   - WordPiece : Similar to BPE, but it merges the pair of tokens that minimizes the likelihood of the training data given the model.
5. Morphological Tokenization : Breaks down words into morphemes, which are the smallest grammatical units in a language, such as stems, root words, prefixes, and suffixes.

The LLaMA tokenizer is a BPE model based on sentencepiece . One quirk of sentencepiece is that when decoding a sequence, if the first token is the start of the word (e.g. “Banana”), the tokenizer does not prepend the prefix space to the string.

2.10

Q: What are the pertaining data used by LLaMA? How many tokens are in the entire training set for training LLaMA?

A:
The training data of Meta LLaMa is mostly from large public websites and forums such as

a. Webpages scraped by CommonCrawl.

b. Open source repositories of source code from GitHub.

c. Wikipedia in 20 different languages.

LLaMA1 foundational models were trained on a data set with 1.4 trillion tokens, most of them are drawn from publicly available data sources.

2.11

Q: What is perplexity [4]? What is it used for? How to calculate it?

A:
Perplexity (PPL) is one of the most common metrics for evaluating language models.
It may be used to compare probability models. A low perplexity indicates the probability distribution is good at predicting the sample.
Perplexity is defined as the exponentiated average negative log-likelihood of a sequence. If we have a tokenized sequence X= (x0, x1, …, xt), then the perplexity of X is calculated as the following

$$\mathrm{PPL}(X) = \exp\{-\frac{1}{t}\sum_i^t \log p_\theta\left(x_i \mid x_{<i}\right)\}$$

where

$$\log p\_(\theta \backslash \text{left})\ (x\_i \mid x\_(< i)\ )$$

is the log-likelihood of the i-th token conditioned on the preceding tokens.

reference: [Perplexity of fixed-length models (huggingface.co)](https://huggingface.co/docs/transformers/perplexity)

2.12

Q: How to generate text from LLMs [5]? What are the decoding methods for outputs?

A:

a. Improved transformer architecture

Massive unsupervised training data

Better decoding methods

b. Decoding methods includes:

Greedy Search

Beam search

Sampling & Top-K Sampling

Top-p (nucleus) sampling

Among them, *top-p* and *top-K* sampling seem to produce more fluent text than traditional *greedy* and *beam* search on open-ended language generation.

2.13

Q: What is prompt learning?

A:

Prompt learning refers to a method used in machine learning, particularly in the context of models like GPT (Generative Pre-trained Transformer), where the model is given a prompt or an instruction in natural language, and it generates a response or continuation based on that prompt.

In traditional machine learning, models are trained on large datasets with fixed inputs and outputs, and the model learns to map from input to output. However, in prompt learning, instead of pre-defining a rigid structure, the model is given prompts that are more flexible and can vary widely.

2.14

Q: What is in-context learning?

A:
In-context learning refers to the model's ability to learn from the context provided within a prompt in real-time, without any additional training. This is a form of few-shot learning, where the model can pick up on the intended task and produce relevant outputs based on a few examples.

Here's how in-context learning works:

1. Example Providing : The user provides examples of the task within the prompt. This could be as simple as a couple of instances of input-output pairs that show the model what is expected. For example, if the model is to perform translations, a prompt might include one or two examples of sentences in English followed by their French translations.
2. Task Description : Sometimes, along with examples, a brief description of the task is also provided within the prompt. This description helps to set the context and clarifies what the model is expected to do.
3. Prompt Adjustment : The model uses the information from the examples and task description to adjust its parameters on the fly. It essentially "understands" the task at hand and applies this understanding to the new input it receives within the prompt.
4. Generation : Based on the in-context examples and descriptions, the model generates an output that aligns with the demonstrated examples.

For example, if you provide a language model with a few examples of text where sentiment is labeled (e.g., "Text: 'I love sunny days.' Sentiment: Positive"), it can then infer that it should label the sentiment of new texts you provide in a similar format.

2.15

Q: What is zero-shot and few-shot learning?

A:
Zero-shot and few-shot learning are concepts in machine learning that describe a model's ability to handle tasks it has not been explicitly trained to perform.

Zero-Shot Learning: Zero-shot learning refers to the ability of a model to correctly perform a task without having seen any examples of that specific task during its training. It relies on the model's ability to generalize from the training it has received on other tasks or data. In the context of large language models like GPT-3 or GPT-4, zero-shot learning means the model can understand and execute a task given only a description of the task at inference time. For example, if you ask a model to translate a sentence from English to French without ever having provided an example of such a translation, and it successfully does so, that's zero-shot learning.

Few-Shot Learning: Few-shot learning, on the other hand, involves providing the model with a very small number of examples (or "shots") of the desired task when prompting it. This can be as few as one or several examples from which the model is expected to understand the task and

generalize to new instances of it. The provided examples serve to guide the model in understanding the context and the nature of the task. Few-shot learning is particularly useful for tasks where there is not enough data to traditionally train a machine learning model.

2.16

Q: What is instruction tuning? Particularly, how is it applied to train LLaMA? Why is instruction tuning supervised training?

A:

Instruction tuning is a training regime specifically designed to improve the performance of language models on tasks specified by natural language instructions. This technique has been used to fine-tune models like GPT-3 and similar large language models to better understand and respond to user instructions.

Here's how instruction tuning typically works:

1. Dataset Creation : A dataset is created with a wide range of tasks, each accompanied by a natural language instruction explaining what the model should do. These tasks are often derived from existing benchmarks that assess different capabilities, such as reasoning, language understanding, translation, summarization, and more.
2. Fine-Tuning : The model, already pre-trained on a large corpus of text data, is further fine-tuned on this specially crafted dataset. During fine-tuning, the model learns to follow the provided instructions and produce the correct outputs.
3. Performance Improvement : By learning from these varied tasks and associated instructions, the model becomes better at understanding and executing a wide array of new instructions it hasn't seen before.

Instruction tuning is considered a form of supervised training because it relies on labeled data where the correct responses to tasks are provided. The model uses these correct responses to learn how to perform tasks as specified by the instructions. This differs from unsupervised learning, where the model would try to learn patterns and representations from the data without any labeled responses.

The instruction tuning is applied to train LLaMA by following steps:

a. Define the use case and create a prompt template for instructions

b. Create an instruction dataset

c. Instruction-tune Llama 2 using `trl` and the `SFTTrainer`

d. Test Model and run Inference

(reference: https://www.philschmid.de/instruction-tune-llama-2)

The supervised aspect comes from the nature of the task and output relationship in the training data: for every instruction (input), there is a correct task output (target), and the model's goal during fine-tuning is to predict the correct output given the instruction. This direct mapping from input to output with labeled examples is the hallmark of supervised learning.

2.17

Q: What is Alpaca dataset [6]? Pick a training sample and describe it.

A:
Stanford Alpaca is an open-source project that demonstrates the capabilities of an instruction-following LLaMA model.

Alpaca is a dataset of 52,000 instructions and demonstrations generated by OpenAI's engine. This instruction data can be used to conduct instruction-tuning for language models and make the language model follow instruction better.`text-davinci-003`.

```json
{
  "instruction": "Create a classification task by clustering the given list of items.",
  "input": "Apples, oranges, bananas, strawberries, pineapples",
  "output": "Class 1: Apples, Oranges\nClass 2: Bananas, Strawberries\nClass 3: Pineapples",
  "text": "Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request.\n\n### Instruction:\nCreate a classification task by clustering the given list of items.\n\n### Input:\nApples, oranges, bananas, strawberries, pineapples\n\n### Response:\nClass 1: Apples, Oranges\nClass 2: Bananas, Strawberries\nClass 3: Pineapples",
}
```

The data fields are as follows

- `instruction`: describes the task the model should perform. Each of the 52K instructions is unique.

- `input`: optional context or input for the task. For example, when the instruction is "Summarize the following article", the input is the article. Around 40% of the examples have an input.

- `output`: the answer to the instruction as generated by ./`text-davinci-003`

- `text`: the , and formatted with the prompt template used by the authors for fine-tuning their models.

reference:

https://github.com/tatsu-lab/stanford_alpaca#data-release

[tatsu-lab/alpaca · Datasets at Hugging Face](https://huggingface.co/datasets/tatsu-lab/alpaca)

2.18

Q: What is human alignment? Why is it important?

A:

10. Human alignment, in the context of artificial intelligence (AI), refers to the development of AI systems that understand and can act according to human values, intentions, and ethical principles. It's about creating AI that can make decisions and take actions that are aligned with what is beneficial for humans and our varied societies.

    The importance of human alignment in AI can be understood in terms of the following considerations:

    1. Safety : As AI systems become more capable, they also become more powerful in their potential impact on the world. If an AI's goals are not aligned with human values, even a seemingly benign task could lead to unintended negative consequences at scale. For instance, an AI instructed to maximize production in a factory without regard for safety could cause harm to workers.
    2. Ethics and Morality : AI should operate in ways that are ethically acceptable to the communities they serve. This includes respecting privacy, fairness, and not discriminating against individuals or groups. An AI system that is not aligned might inadvertently perpetuate biases present in its training data or make decisions that are ethically questionable.
    3. Legal and Regulatory Compliance : AI systems must operate within the legal frameworks of the societies they are deployed in. This means compliance with regulations regarding data protection, consumer rights, and other legal requirements. Human-aligned AI would inherently respect and adhere to these regulations.
    4. Avoiding Misuse : AI systems that are closely aligned with human intentions are less likely to be misused in ways that could be harmful. By ensuring that AI systems understand and adhere to human ethical standards, we reduce the risk of them being employed for malicious purposes.

3 Preliminary

3.1 Gradient Accumulation

Q: Consider a linear model with MSE loss, and mathematically explain why this division step can yield identical results.

A:
Assume n is number for images in a batch, the MSE can be denoted as:

$$MSE\_batch = 1/n \sum\_(i = 1)\^n (Y\_i - Y\hat{}\_i)\^2$$

If Gradient Accumulation is applied during training, where 'm' represents the number of sections it divides into, 'q' signifies the number of images in one section, and 'n' is the total number of images, the MSE can be expressed as:

$$\begin{aligned} MSE\_GA &= 1/m \sum\_(j = 1)\^m \left[1/q \sum\_(i \right. \\ &= (j-1) * q)\^((j-1) * q + q) (Y\_i - Y\hat{}\_i)\^2 \left. \right] \\ &= 1/mq \sum\_(i = 1)\^mq (Y\_i - Y\hat{}\_i)\^2 \\ &= 1/n \sum\_(i = 1)\^n (Y\_i - Y\hat{}\_i)\^2 = MSE\_batch \end{aligned}$$

Q: Mathematically explain the batch normalization layer behavior and why gradient accumulation yields non-identical results in this case.

A:
Assume n is number for images in a batch, the Batch Mean and Batch Variance can be denoted as:

$$\mu\_batch = 1/n \sum\_(i = 1)\^n x\_i$$
$$\sigma\_batch\^2 = 1/n \sum\_(i = 1)\^n (x\_i - \mu\_batch)\^2$$
$$x\_BN = (x - \mu\_batch)/\sigma\_batch$$

If Gradient Accumulation is applied during training, where 'm' represents the number of sections it divides into, 'q' signifies the number of images in one section, and 'n' is the total number of images, the Batch Mean and Batch Variance for each section can be denoted as:

$$\mu\_(GA\_m) = 1/q \sum\_(i = 1)\^q x\_i$$
$$\sigma\_(GA\_m)\^2 = 1/q \sum\_(i = 1)\^q (x\_i - \mu\_(GA\_m))\^2$$
$$x\_(BN\_m) = (x\_(GA\_m) - \mu\_(GA\_m))/ \left[\sigma\_GA \right] \_m$$

Since:

$$\mu\_batch \neq \mu\_(GA\_m)$$
$$\sigma\_batch\^2 \neq \sigma\_(GA\_m)\^2$$

Therefore, the batch normalization layer can lead to non-identical results when gradient accumulation is employed during training.

3.2 Gradient Checkpointing

Q: Explain why model inference does not have such "memory blow-up" problem.

A:
During inference, only a single forward pass occurs, and there is no backpropagation. Therefore, inference doesn't involve multiple forward and backward passes, and the model doesn't need to calculate gradients, update parameters, or store all the activation values and gradients. As a result, inference requires significantly less memory compared to model training.

Q: What are the memory requirement and forward computation steps for the two strategies in big O notation?

A:

	Default	Memory-Poor (1st strategy)	$\sqrt{n}$ (2^{nd} strategy)
Memory Requirement	$O(n)$	$O(1)$	$O(\sqrt{n})$
Computation Requirement	$O(n)$	$O(n\text{^}2)$	$O(n)$
Forward calcs per node	1	1 to n	1 to 2

3.3 Low-Rank Adaptation (LoRA)

Q1: What is matrix rank?

A:
The rank of a matrix is the maximum number of linearly independent rows or columns it contains, indicating the count of unique rows or columns.

Q2: What are three decomposed matrices by SVD?

A:

$$A = U\Sigma V\text{^}T$$

U: Left singular vector, it can be used to transform data into a new basis
Σ: Diagonal singular values, this is a diagonal matrix that contain the singular value of the original matrix
$V\text{^}T$: Right singular vector, it can be used to transform data or perform dimensionality reduction

---

Q3: U and V are orthogonal matrices. Why does it imply $UU\text{^}T = I$, $VV\text{^}T = I$?

A:
An orthogonal matrix is defined such that when it is multiplied by its transpose, the result is the identity matrix. Therefore, if U and V are orthogonal matrices, it implies that that $UU\text{^}T = I$and $VV\text{^}T = I$.

Q4: If a matrix $W \in R\_(n * n)$ is full rank, what is its rank?

A:
If a matrix $W \in R\_(n * n)$ is full rank, it implies that both its row rank and column rank are equal to n.

Q5: Suppose a full rank matrix $W \in R\_(n * n)$ represents an image. After we apply SVD to this matrix, we modify the singular matrix by only keeping its top-k singular values and discarding the rest (i.e., set the rest of the singular values to zero). Then, we reconstruct the image by multiplying U, modified S, and V. What would the reconstructed image look like? What if you increase the values of k (i.e., keep more singular values)?

A:
When retaining only the top-k singular values to reconstruct the image, you will notice that the image loses some of its finer details, leading to a decrease in image quality compared to the original. As you increase the value of k, the reconstructed image will progressively resemble the original image more closely.

Q6: If a matrix $W \in R\_(n * n)$ is low rank, what does its singular matrix look like?

A:
If W is not full rank, the singular value matrix will not be a perfect diagonal identity matrix. Instead, it will have only a limited number of non-zero singular values, while the remaining singular values will be very close to or equal to zero.

Q7: If the top-k singular values of a matrix $W \in R\_(n * n)$ are large, and the rest are near zero, this matrix W exhibits low-rank or near-low-rank behavior. Can you represent W by two low-rank matrices, A and B? If so, what are those two matrices' expressions in terms of U, S, and V ? Do you think those two matrices are a good approximation of W (i.e., $W \approx AB$)?

A:
If top-k singular values of a matrix $W \in R\_(n * n)$ are large, and the rest are near zero W can be represent by two low rank matrices A and B.

$$A = U[:, 1:k]$$

$$B = \Sigma[1:k, :] * V\text{^}T$$

Where k is the first k columns of the matrix Σ
The approximation of W is giving by $W \approx AB$. The quality of this approximation depends on the choice of k. If the top-k singular values are indeed significantly larger than the rest, then this approximation is often quite good.

Q8: The above operation is called truncated SVD. Under what situation do you think truncated SVD fails to make a good approximation? Think about the singular matrix.

A:
If the singular values exhibit a relatively uniform distribution, implying that there is no substantial decrease in magnitude beyond the initial top few singular values, truncating to a small k might not effectively capture the majority of the information contained in the original matrix. In such situations, the approximation is likely to be less accurate.

Q: Derive the equation in terms of r that specifies the conditions under which the total number of parameters in matrices A and B is smaller than that of W0.

A:
We know: $W\_0 \in R\^(n \times n), A \in R\^(n \times r), B \in R\^(r \times n)$
So, the total number of parameters in $W\_0 = n\^2$
The total number for parameters in A and B is $n \times r + r \times n = 2rn$, where $r \ll n$ so $2rn \ll n\^2$
Therefore, the total number of parameters in matrices A and B is smaller than that of W_0

Q: Consider carefully why LoRA can save computation and memory costs during the fine-tuning stage.

A:
During the fine-tuning's backward propagation, we keep the pre-trained weights frozen and only update the weights of matrices A and B. Since the parameters of A and B are significantly fewer than those of the original model, both the computation and memory costs are much lower compared to directly updating the original model.

Q: Can you think of a way to eliminate this drawback?

A:
To address the drawbacks of adding matrices A and B, several approaches can be considered. Firstly, we can apply Mixed Precision Training techniques to the A and B matrices as well as the original parameters. This can reduce space and time costs while ensuring precision. Secondly, the weights of matrices A and B can be merged with the pre-trained matrix during fine-tuning. This would eliminate the A and B matrices, thus maintaining the same forward pass time as before.

3.4 Mixed Precision Training

Q: Check the paper and answer why we need an FP32 master copy of weights and why the memory requirement is reduced despite storing an additional copy of weights in FP16.

A:
Compared to FP16, FP32 requires more memory but offers higher precision. Retaining an FP32 master copy of weights maximizes the preservation of the model's precision. This is because during gradient accumulation, if the gradients are very small, FP16 might choose a value larger or smaller than the actual value. Therefore, choosing FP32 helps to minimize errors caused by loss of information.

The reason why the memory requirement is reduced despite storing an additional copy of weights in FP16 is that the memory space required for FP16 is only half that of FP32. This significantly reduces the total computational demand during forward and backward propagation.