


Fine-tuning LLMs

Large Language Models (LLMs) have revolutionized the field of natural language processing. We have a really good set of llm models, but they are trained in a generic manner, we have seen a few methods to make the model perform and output in a way we want it:

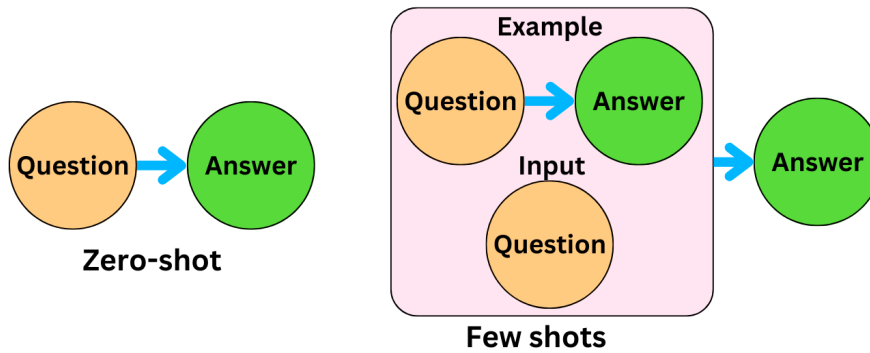
Prompt Engineering

- **Method:** Carefully crafting the prompts given to the LLM can influence its output. By providing informative and specific prompts, we can guide the model towards desired responses.
- **When to use:** This technique is particularly effective when the task involves generating text, such as writing emails or creating summaries.
-  [Harnessing the LLM APIs and OpenAI Ecosystem.ipynb](#)

Worse	Better
How do I add numbers in Excel?	How do I add up a row of dollar amounts in Excel? I want to do this automatically for a whole sheet of rows with all the totals ending up on the right in a column called "Total".
Who's president?	Who was the president of Mexico in 2021, and how frequently are elections held?
Write code to calculate the Fibonacci sequence.	Write a TypeScript function to efficiently calculate the Fibonacci sequence. Comment the code liberally to explain what each piece does and why it's written that way.
Summarize the meeting notes.	Summarize the meeting notes in a single paragraph. Then write a markdown list of the speakers and each of their key points. Finally, list the next steps or action items suggested by the speakers, if any.

Few-Shot Learning

- **Method:** The LLM is trained on a small number of examples from the target task. The model learns to generalize from these examples and produce accurate outputs on unseen data.
- **When to use:** This technique is useful when obtaining large amounts of labeled data is challenging or expensive.



```
[ ] messages = [
  {"role": "system", "content": "you are an helpful assistant"},
  {
    "role": "user",
    "content": "Convert this instruction to JSON: Create a project named ProjectA with a deadline of 2024-12-31 and priority high."
  },
  {
    "role": "assistant",
    "content": " {\"name\": \"ProjectA\", \"deadline\": \"2024-12-31\", \"priority\": \"high\"}"
  },
  {
    "role": "user",
    "content": "Convert this instruction to JSON: Schedule a meeting with Bob on 2024-07-10 at 10:00 AM with the subject Quarterly Review"
  },
  {
    "role": "assistant",
    "content": "{\"participant\": \"Bob\", \"date\": \"2024-07-10\", \"time\": \"10:00 AM\", \"subject\": \"Quarterly Review\"}"
  },
  {
    "role": "user",
    "content": "Convert this instruction to JSON: Add a new user named Alice with email alice@scaler.com and role admin."
  }
]

response = get_completion_from_messages(messages)
print(response)
```

But these are **not actually finetuning the model**, we are just ordering the same model to behave in a certain way giving ideas and examples.

If we want to make the **model learn** and **give it inbuilt information** of what we want then there are two ways:

Pre-Training vs Fine-tuning

LLMs are trained on massive datasets to understand and generate human language.

Two crucial phases in the training process are **pre-training** and **fine-tuning**.

Let's explore the key differences between these two stages.

1. Definition and Purpose

Pre-Training

- The **goal of pre-training** is to **create a robust language model that has a broad, generalized understanding of language**.
 - This process equips the model with the ability to generate coherent text, predict the next word in a sequence, and perform other language-related tasks without any task-specific tuning.

Fine-Tuning

- The **aim of fine-tuning** is to **specialize the pre-trained model for a particular application, improving its performance on that task by providing it with domain-specific knowledge** or correcting any biases introduced during pre-training.

2. Training Data

Pre-Training

- **Data Volume:** Pre-training requires **vast amounts of data**, often comprising billions of words or even entire internet-scale datasets. Common sources include books, articles, websites, and social media.
- **Data Diversity:** The datasets used for pre-training are **highly diverse**, covering a wide range of topics, writing styles, and languages. This diversity helps the model learn general linguistic structures and patterns.

Fine-Tuning

- **Data Volume:** Fine-tuning typically uses much **smaller datasets**, often ranging from a few thousand to a few million examples, depending on the complexity of the task.
- **Data Specificity:** The data used for fine-tuning is **usually domain-specific** and annotated, meaning that it includes **labeled examples** relevant to the target task. For instance, a model fine-tuned for sentiment analysis would be trained on a dataset where texts are labeled with sentiments like positive, negative, or neutral.

- **Data Preparation:** Fine-tuning requires carefully curated and often annotated data to guide the model's learning towards the desired outcome.

3. Learning Objectives

Pre-Training

- **Generalization:** Pre-training **focuses on capturing general language knowledge, including syntax, semantics, and general world knowledge embedded in the text.**

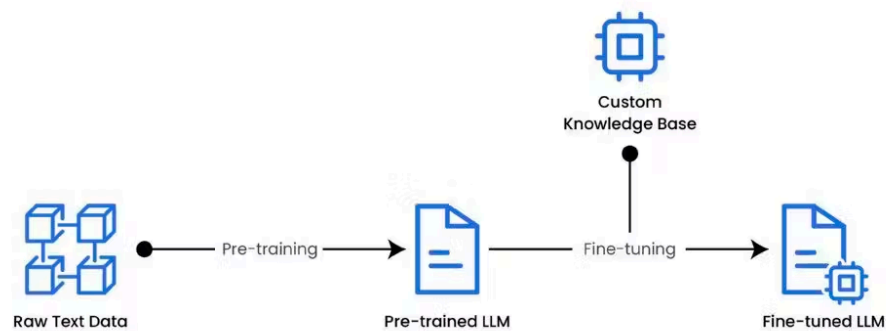
Fine-Tuning

- **Specialization:** During fine-tuning, the **model is trained to excel at a particular task**, often at the expense of some of its general language abilities. The trade-off between generalization and specialization is a key aspect of fine-tuning.

Feature	Pre-training	Fine-tuning
Data:	Massive, unlabeled data	Smaller, labeled data
Objective:	General language understanding	Task-specific performance
Process:	Unsupervised learning	Supervised learning
Result:	General-purpose model	Specialized model

One more thing to note:

- **Pre-training is computationally intensive and time-consuming**, often requiring days or weeks of training on high-performance hardware (like GPUs or TPUs) across multiple machines.
- **Fine-tuning is usually much faster than pre-training**, often taking only a few hours or days, depending on the dataset size and computational resources.



Fine-tuning Process

Fine-tuning makes sense in the following scenarios:

1. **Task-Specific Requirements:** When you need a model to perform a specific task (e.g., sentiment analysis, machine translation) that the general-purpose pre-trained model is not optimized for.
2. **Domain-Specific Data:** When the target application involves a specialized domain (e.g., legal, medical, technical) where the pre-trained model lacks sufficient expertise.
3. **Performance Improvement:** When the pre-trained model's performance on a specific task is not satisfactory, and fine-tuning can improve accuracy, relevance, or other key metrics.
4. **Custom Outputs:** When you need the model to generate outputs that align with specific guidelines, styles, or formats that are not captured during pre-training.

Pretraining makes sense in the following scenarios:

- **No pretrained model is enough:** Assume you are building something for a local language like bengali or malayali, and no other model is available.
- **Have a huge dataset for your task:** for finetuning according to your needs you dont need that much big a dataset but for pretrainig you need a very big one
- **You dont need a generalized model:** maybe your task is very specific, and requires a very specific type of information or thinking. And being trained on previous data may in turn damage your requirements.
- **Want a much smaller model:** Now if you have a huge dataset, and dont need a generalise model, you could even work with a much smaller model as you dont

have to teach it that much. In this case the training time and resources would also be less. (Still a lot more than fine tuning though)

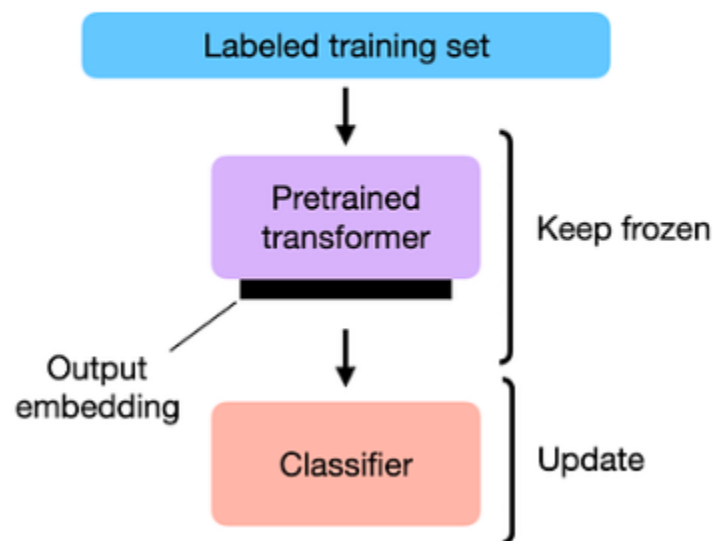
Finetuning techniques

1. Feature-Based Fine-Tuning

Feature-based fine-tuning involves using the representations (or features) learned by the pre-trained model as inputs for a separate model or downstream task, without modifying the pre-trained model's parameters.

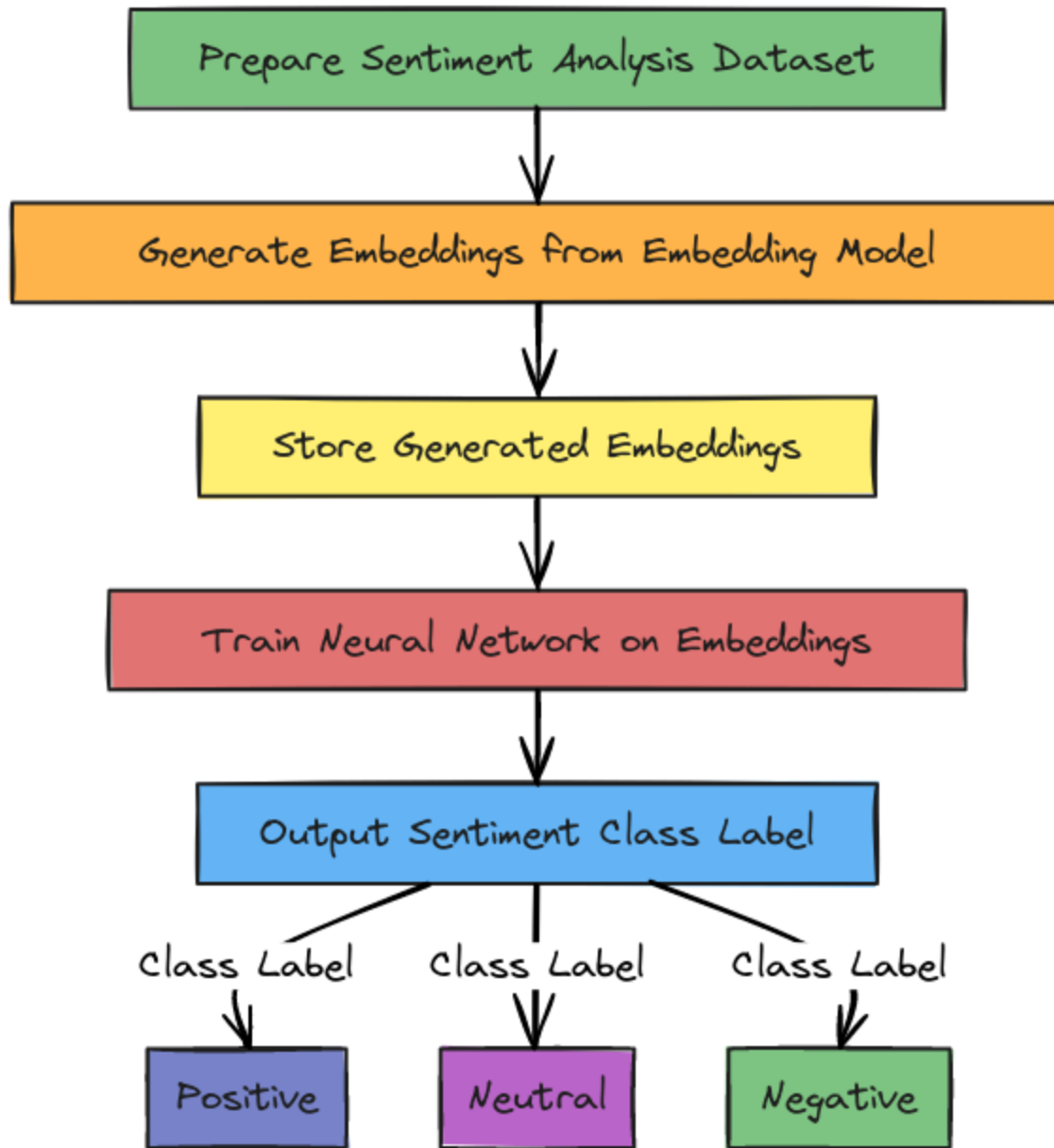
This approach leverages the powerful features that the LLM has learned during pre-training to solve specific tasks by extracting these features and applying them in a different context.

FEATURE-BASED APPROACH



Key Characteristics:

- **Fixed Pre-Trained Model:** The pre-trained model's **parameters remain unchanged**. The model **acts as a feature extractor**, providing embeddings or other representations from the input data.
 - The input text is processed to **extract features such as word embeddings**, character n-grams, or syntactic information.
 - **Example: we want to create sentiment analysis system with three class outputs : positive, neutral and negative.**
 - Prepare a sentiment analysis dataset
 - Generate embeddings for your complete dataset from any embedding model and store it.
 - Train a new NN for sentiment analysis on these embeddings, with the labels.
 - So instead of training on sentences or text now your nn model is trained on embeddings directly making it pretty simple.
 - The output would be a class label
 - Everytime you get a sentence on inference time you first use the same models for embeddings and pass it to the NN for answer.



- **External Model Utilization:** A separate, often smaller model or classifier (such as a linear classifier or a neural network) is trained on top of the features extracted by the LLM. This external model is fine-tuned on the task-specific data.
- **Efficiency:** Since the pre-trained model is not updated, this approach is computationally efficient and requires fewer resources. It is particularly useful when the downstream task does not require extensive changes to the LLM's representations.
- **Flexibility:** This method allows for easy switching between different tasks,
 - Assume you have a dataset of customer surveys
 - You generate embeddings for all of it

- Now the same embeddings can be used to train sentiment analysis and classify if it is a feedback, a complaint, or a praise for the product.
- So in this case you just have to train the NN differently on these tasks not the entire model.

Disadvantages:

- **Limited Adaptability:** May be less flexible in adapting to complex tasks or when the original pre-training data is not well-suited for the target task.

Fine-tuning vs. RAG		
What's the difference?		
	RAG	Fine-tuning
Use external system?	RAG allows for dynamic insertion of relevant content from external systems	Fine-tuning dataset should be large enough to cover knowledge sources
Modify model behavior or domain-specific knowledge?	Feed domain knowledge directly into the prompt. No model changes.	Encode domain knowledge into model by modifying model weights
Is the task standardized?	RAG works well for Q/A-type use-cases	Fine-tuning works well for standardized tasks (structured output)
Training data available?	RAG doesn't use labelled training data	Leverage input-output pairs to understand nuances and intricacies

Parameter-Based Fine-Tuning

Parameter-based fine-tuning involves **directly modifying the weights (parameters) of the pre-trained model** to adapt it to a specific task.

This method allows the **model to learn task-specific patterns** and optimize its internal representations for better performance on the target task.

Key Characteristics:

- **Updating the Pre-Trained Model:** The parameters of the pre-trained model are adjusted during fine-tuning. **The entire model, or a portion of it, is trained on the task-specific dataset.**
- **Task Specialization:** This approach leads to a highly specialized model that is finely tuned for the target task.

The model's internal representations are optimized to perform well on this specific task, often leading to higher accuracy and better performance compared to feature-based fine-tuning.

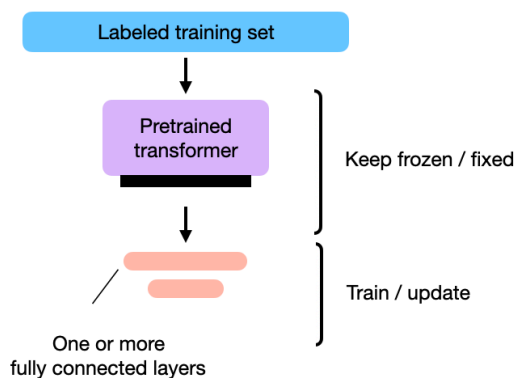
Disadvantages:

- **Computational Cost:** Can be computationally expensive, especially for large LLMs and large datasets.
- **Risk of Catastrophic Forgetting:** There is a risk of the model forgetting information learned during pre-training.

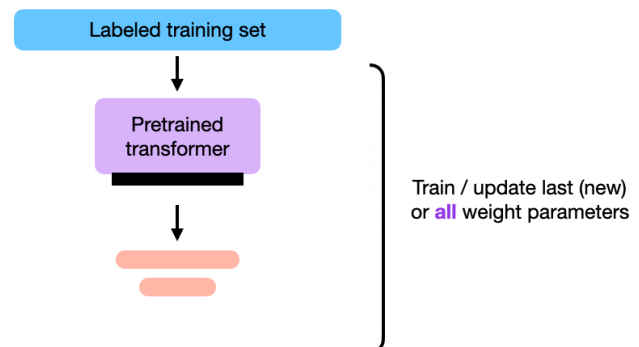
Example:

- Finetuning the model on a medical dataset to make it more precise on medical questions.

1) FINETUNE LAST LAYERS



2) FINETUNE ALL LAYERS



LLMs vs. SLMs

Large Language Models (LLMs) and Small Language Models (SLMs) are both powerful tools in the field of natural language processing. While they share a common foundation, their scale and capabilities differ significantly.

LLM Key Characteristics:

- **Size and Complexity:** LLMs have a large number of parameters, often exceeding hundreds of billions. This enables them to capture complex patterns and nuances in language.
- **Training Data:** LLMs are trained on massive datasets, including books, articles, websites, and other forms of text, allowing them to learn from diverse sources of information.
- **Versatility:** LLMs are highly versatile and can be fine-tuned for specific tasks such as translation, summarization, question answering, and more. They can also perform well in zero-shot or few-shot learning scenarios.
- **Resource Requirements:** Due to their size, LLMs require significant computational resources for both training and inference. They often need specialized hardware like GPUs or TPUs and considerable memory.

SLM Key Characteristics:

Small Language Models (SLMs) are language models with a **significantly smaller number of parameters compared to LLMs**. These models are designed to be **efficient** and **lightweight**, making them suitable for deployment in environments with limited computational resources. SLMs can still perform well on specific tasks, especially when they are fine-tuned or trained on domain-specific data.

The value at which a model converts from SLM to LLM is vague. Some people like to call models with 100s of millions of parameters, some say less than a billion parameters to be SLM, some argue till 10 billion

- **Size and Efficiency:** SLMs have far fewer parameters, often ranging from a few million to a few billion. This makes them more efficient in terms of computational resource requirements.
- **Faster Inference:** Due to their smaller size, SLMs are faster during inference, making them ideal for real-time applications where latency is a critical factor.
- **Lower Resource Requirements:** SLMs can be deployed on devices with limited computational power, such as mobile phones or edge devices. They do not require the extensive hardware infrastructure needed by LLMs.
- **Task-Specific:** While SLMs are less versatile than LLMs, they can be highly effective for specific tasks, especially when trained or fine-tuned on relevant datasets.

Examples:

- DistilBERT: https://huggingface.co/docs/transformers/en/model_doc/distilbert
- Tiny GPT family: <https://huggingface.co/papers/2312.16862>
- Tiny LLaMA chat 1.1B:
<https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0>
- Gemma 2-2B: <https://huggingface.co/google/gemma-2-2b>

If you notice all of these are above a billion as they are more generalized models. But you can even use a smaller model for simpler tasks with decent result

https://huggingface.co/facebook/m2m100_418M : Text translation

LLM

- Up to 2 trillion parameters
- Trained on big, varied data sets
- Trained in months
- Advanced computing power needed
- Ideal for complex and general tasks
- Expensive to fine tune
- Higher latency
- High token cost
- Cannot run on device or on edge devices

SLM

- 100M - 10B parameters
- Trained on small, niche data sets
- Trained in days
- Can work with basic computing power
- Ideal for specific tasks
- Cheaper to fine tune
- Lower latency
- Low token cost (up to 1/50th of the price)
- Can run on device or on edge devices

PEFT

Parameter-Efficient Fine-Tuning (PEFT) refers to **a set of techniques** designed to fine-tune large pre-trained models by updating only a **small subset of the model's parameters**. **Instead of adjusting all the parameters of the model**, which can be in the billions, PEFT focuses on optimizing a smaller number of parameters, leading to a more resource-efficient and faster fine-tuning process.



There are these major methods of PEFT finetuning:

Adapters

Adapters are small, task-specific modules inserted into the layers of an LLM. These modules are trained while the original model layers are frozen, meaning their weights are not updated during fine-tuning.

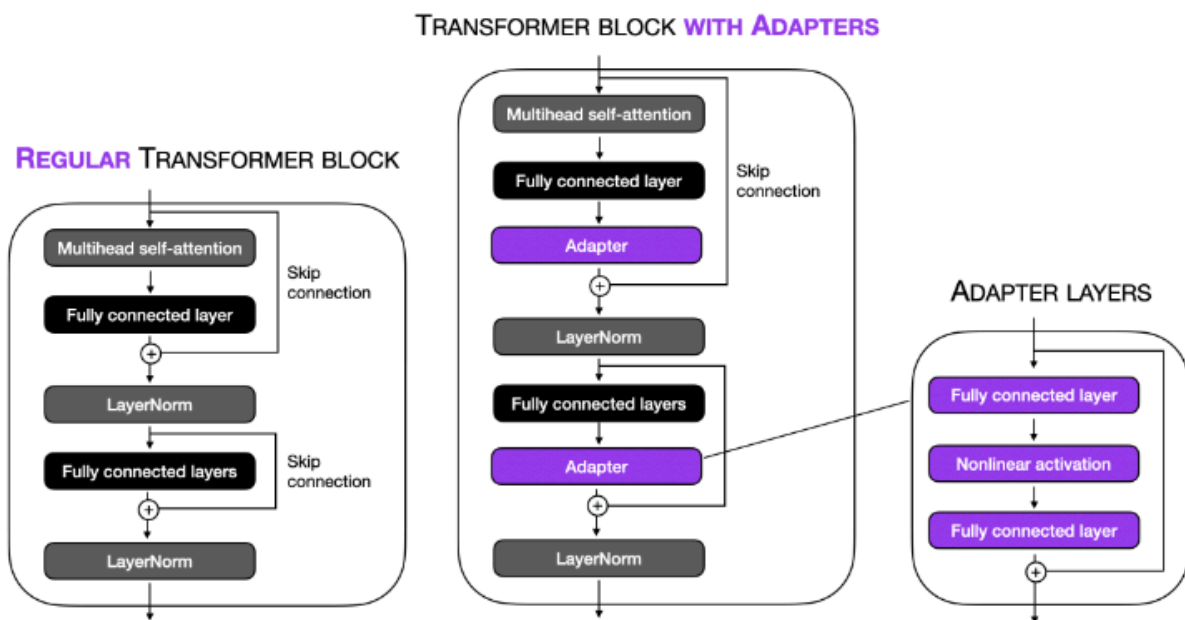
How Adapters Work:

- **Adding Adapters:** Imagine the LLM as a big machine. We add small, extra parts called adapters to different places in the machine. These adapters are like mini-brains that can learn new things.

- **Learning New Things:** When we want to teach the LLM something new, we only train the adapters. This is like teaching the mini-brains without changing the whole machine.
- **Using the New Knowledge:** Once the adapters have learned, the LLM can use their knowledge to do new tasks or do old tasks better.

How to use Adapters

- Adapter layers add minimal additional parameters to the pretrained model. These adapters are **inserted between existing layers** of the network.
- Adapters usually are a **small neural network module typically consisting of two fully connected layers with a non-linear activation function between them.**
- Typically, adapters are inserted at multiple points within the model.
 - Often, **the top layers (closer to the output) are fine-tuned because they are more specialized and closer to making the final prediction.**
 - The decision on which layers to unfreeze or train along with adapters is often based on experimentation. You might try different combinations and see which setup gives the best performance on your specific task. It's all a black box after all.



Function of Adapters:

- **Learning Task-Specific Features:** During training, these adapter layers adjust their weights to tune to features specific to the task at hand, like sentiment

analysis or text classification. The rest of the model's parameters remain frozen (unchanged).

- **Influencing Output:** When data is passed through the model during inference (prediction), it goes through both the original layers and the adapter layers. **The adapters tweak the data slightly based on the task they were fine-tuned for,** thus improving the model's performance on that specific task.

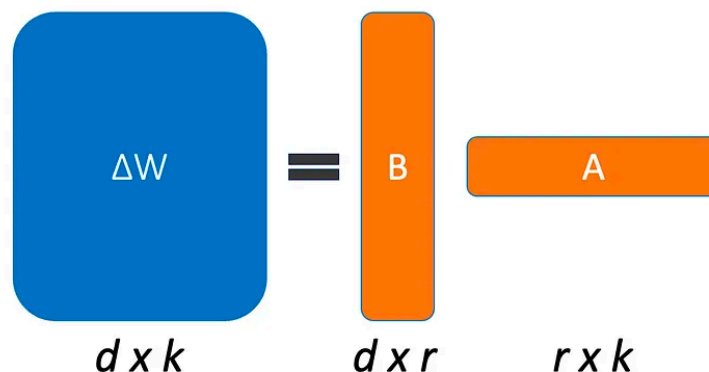
The learning process is almost the same that you learned in neural networks finetuning lectures

LORA

LoRA (Low-Rank Adaptation) is a specific technique within Parameter-Efficient Fine-Tuning (PEFT), designed to further reduce the computational and memory requirements for fine-tuning large models.

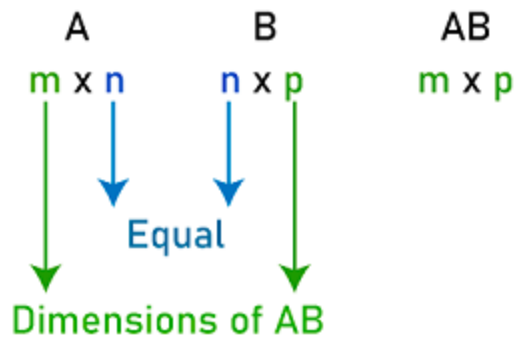
LoRA is similar to adapters, but instead of adding full trainable layers, it introduces a more efficient way to modify the weights of the pre-trained model by adding low-rank matrices to specific parts of the model.

- LoRA represents the weight updates ΔW with two smaller matrices (called update matrices) through low-rank decomposition.



- Remember the matrix multiplication rule

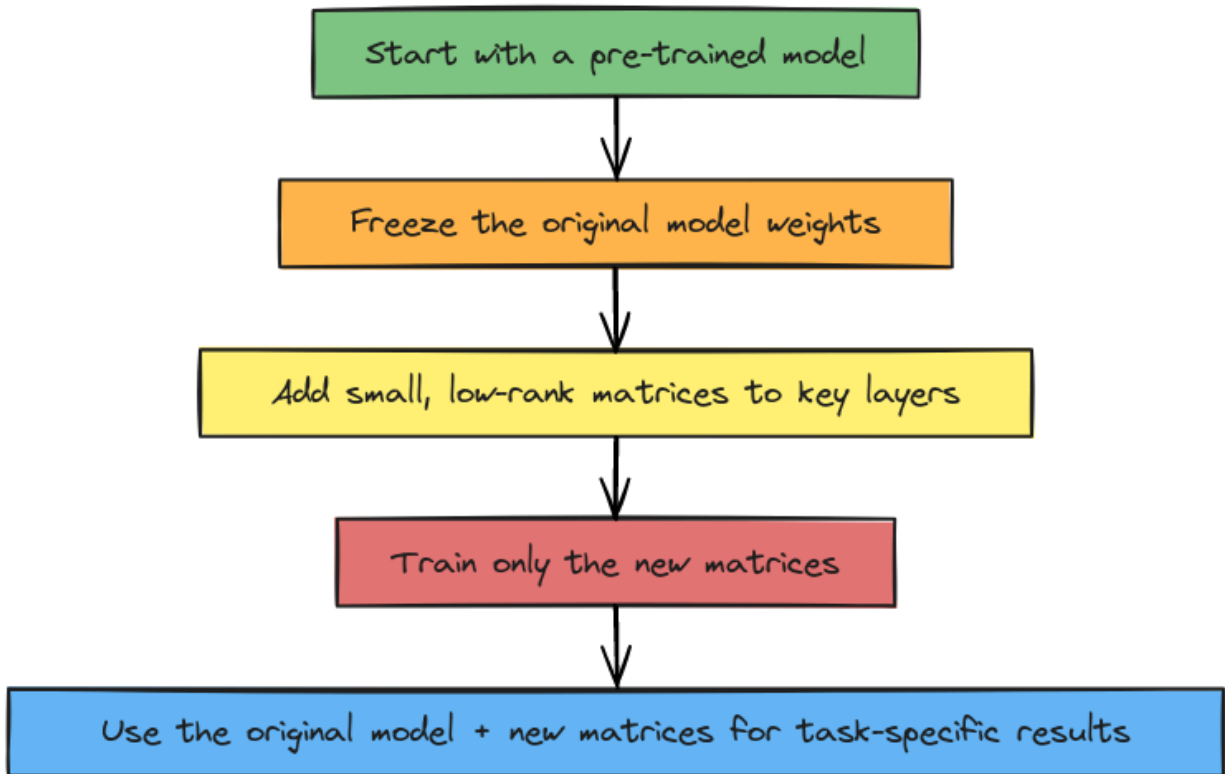
Rule For Matrix Multiplication



- replaced by the product of two smaller matrices
- **Let's say you have a large 100x100 matrix in your model. That means you have 10,000 numbers (parameters) to manage. With LoRA, you could approximate this with two low-rank matrices, each 100x10 and 10x100. This gives you only 2,000 parameters (100x10 + 10x100) to train—80% fewer parameters! This leads to a significant reduction in both computational load and memory usage.**
- Remember this is just an approximation of the original matrix
- The original weight matrix remains frozen and doesn't receive any further updates.

When the model processes input data (like a sentence), the original weights (which are frozen) still do their job. However, the low-rank matrices added by **LoRA adjust the data slightly at specific points in the model**. These adjustments are what help the model improve on a new task.

- **Think of it like adding a new filter to a camera lens: the original camera still takes the picture, but the filter gives it a new look based on what you need.**



Why Use LoRA?

- **Efficiency:** LoRA uses fewer parameters, which makes fine-tuning faster and less expensive.
- **Memory Savings:** Since you're only training small matrices, it takes up much less memory than training the entire model.
- **Scalability:** LoRA works well with very large models, making it possible to fine-tune them on specific tasks without needing enormous resources.

