**Fine Tuning LLMs**

Large language models (LLMs) are a category of foundation models trained on immense amounts of data making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks. These models are often general and are not equipped for specific tasks. In order to get the full potential of LLMs on a specific task you can use different methods:

* Prompt Engineering
* Retrieval-Augmented Generation (RAG)
* Fine-Tuning

**Fine-Tuning**

Here we focus on Fine Tuning. Large language model (LLM) fine-tuning is the process of taking pre-trained models and further training them on smaller, specific datasets to refine their capabilities and improve performance in a particular task or domain. Fine-tuning is about turning general-purpose models and turning them into specialized models. It bridges the gap between generic pre-trained models and the unique requirements of specific applications, ensuring that the language model aligns closely with human expectations. Think of OpenAI's GPT-3, a state-of-the-art large language model designed for a broad range of [natural language processing (NLP)](https://www.superannotate.com/blog/what-is-natural-language-processing) tasks. Suppose a healthcare organization wants to use GPT-3 to assist doctors in generating patient reports from textual notes. While GPT-3 can understand and create general text, it might not be optimized for intricate medical terms and specific healthcare jargon.

To enhance its performance for this specialized role, the organization fine-tunes GPT-3 on a dataset filled with medical reports and patient notes. It might use different tools to build its own model with the desired interface. Through this process, the model becomes more familiar with medical terminologies, the nuances of clinical language, and typical report structures. After fine-tuning, GPT-3 is primed to assist doctors in generating accurate and coherent patient reports, demonstrating its adaptability for specific tasks.

There are different methods (models) of Fine Tuning:

* Instruction fine-tuning
* Parameter-efficient fine-tuning
* **Adapter-based Fine-tuning**
* **Layer-wise Fine Tuning**

**Parameter-efficient fine-tuning**

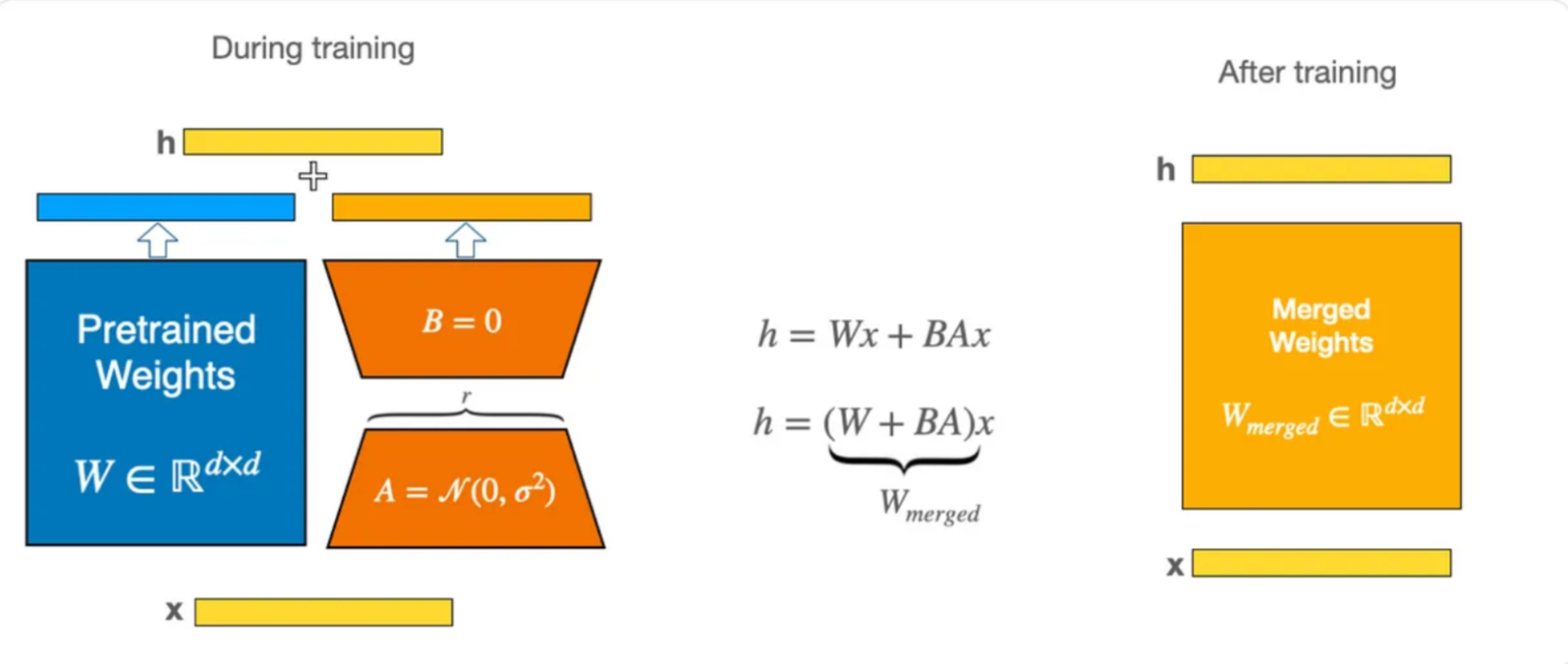
Training a language model is a computationally intensive task. For a full LLM fine-tuning, you need memory not only to store the model, but also the parameters that are necessary for the training process. Your computer might be able to handle the model weights, but allocating memory for optimizing states, gradients, and forward activations during the training process is a challenging task. Simple hardware cannot handle this amount of hurdle. This is where PEFT is crucial. While full LLM fine-tuning updates every model's weight during the supervised learning process, **PEFT methods only update a small set of parameters**. This transfer learning technique chooses specific model components and "freezes" the rest of the parameters. The result is logically having a much smaller number of parameters than in the original model (in some cases, just 15-20% of the original weights; LoRA can reduce the number of trainable parameters by 10,000 times). This makes memory requirements much more manageable. Not only that, but PEFT is also dealing with catastrophic forgetting. Since it's not touching the original LLM, the model does not forget the previously learned information. Full fine-tuning results in a new version of the model for every task you train on. Each of these is the same size as the original model, so it can create an expensive storage problem if you're fine-tuning for multiple tasks.

**Low Rank Adaptation (L0Ra)**

**LoRa** is an improved fine-tuning method where instead of fine-tuning all the weights that constitute the weight matrix of the pre-trained large language model, two smaller matrices that approximate this larger matrix are fine-tuned. These matrices constitute the LoRA adapter. This fine-tuned adapter is then loaded to the pre-trained model and used for inference. It is introduced by a team of Microsoft researchers in 2021**.** In some paper it has been showed that common pre-trained models have a very low intrinsic dimension; in other words, there exists a low dimension re-parameterization that is as effective for fine-tuning as the full parameter space. And thechange in weights during model adaptation also has a low “intrinsic rank”, leading to Low-Rank Adaptation (LoRA) approach.

LORA is designed to fine-tune large-scale models efficiently by targeting a small subset of the model’s weights that have the most significant impact on the task at hand. This contrasts with traditional fine-tuning, where many more weights might be updated. LORA achieves this by:

* Decomposing large matrices of weight changes into smaller matrices that contain the “trainable parameters.”
* Tracking changes to weights instead of updating them directly.

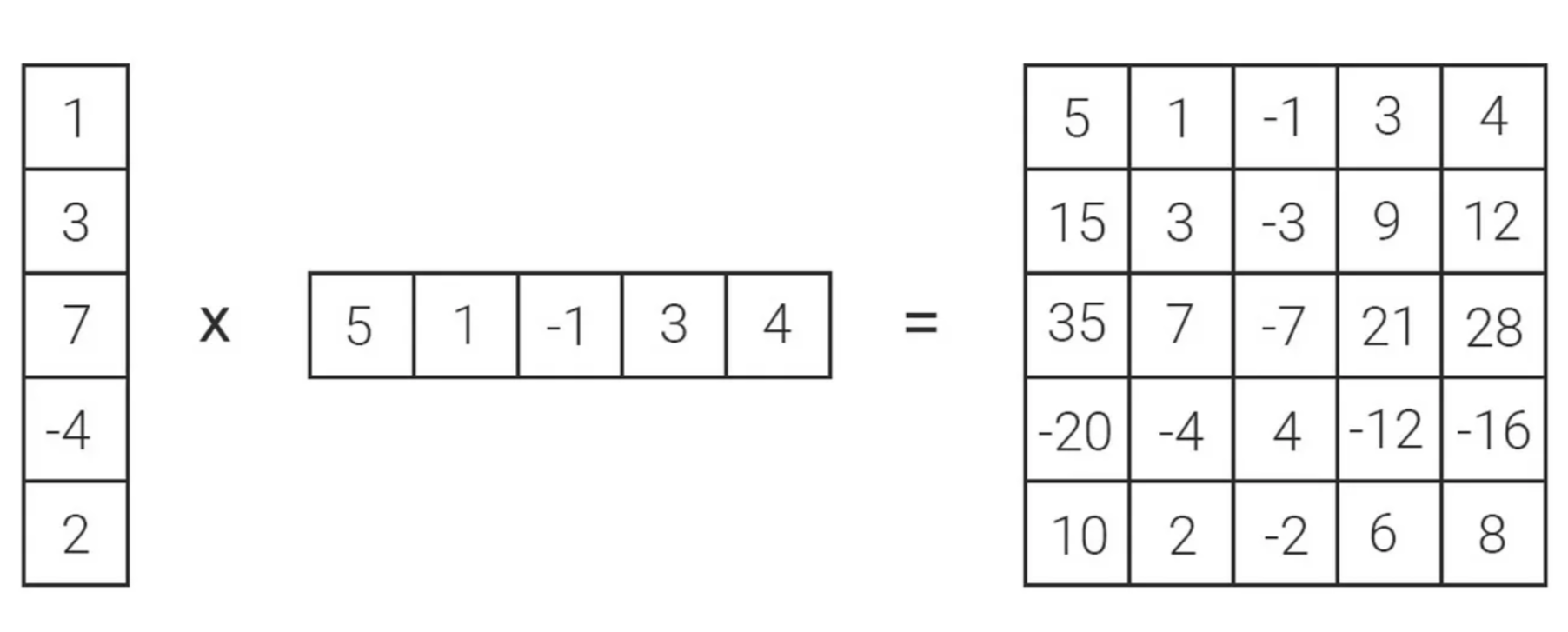
****

This approach offers several advantages:

* Significant reduction in trainable parameters, leading to faster and more efficient fine-tuning.
* Preservation of the original pre-trained weights, allowing for multiple lightweight models for different tasks.
* Compatibility with other parameter-efficient methods, enabling further optimization.
* Comparable performance to fully fine-tuned models in many cases.
* No additional inference latency, as adapter weights can be merged with the base model.

LoRA Shrinks and Speeds up Fine-Tuning through Matrix Decomposition

* This is more for conceptual understanding. Imagine a 5x5 matrix as a storage unit with 25 spaces. LORA breaks it down into two smaller matrices through matrix decomposition with “r” as rank (the dimension): a 5x1 matrix (5 spaces) and a 1x5 matrix (5 spaces).



* This reduces the total storage requirement from 25 to just 10, making the model more compact.
* It also accelerates computations. Working with smaller matrices involves fewer calculations, leading to faster fine-tuning.

**Strategic Focus on Attention Blocks for Maximum Efficiency**

* While LORA can potentially be applied to different parts of a neural network, it’s often strategically used on attention blocks within Transformer models.
* Attention blocks play a key role in LLMs, focusing on the most relevant information during language processing. By selectively adapting these blocks, LORA achieves significant efficiency gains without compromising overall performance.

**Sample Implementation**

from peft import LoraConfig, get\_peft\_model

from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from\_pretrained("your-base-model", device\_map = 'cuda')

config = LoraConfig(

r=32,

lora\_alpha=32,

target\_modules=["query", "value"],

lora\_dropout=0.1,

bias="lora\_only",

modules\_to\_save=["decode\_head"],

)

lora\_model = get\_peft\_model(model, config)

print\_trainable\_parameters(lora\_model)

Enabling LoRA:

1. Tell LoRA which parts to train:
2. Use LoraConfig to specify which parts of the model to update using LoRA.
3. Target the “query” and “value” matrices in the attention blocks.
4. Wrap the model:
5. Enclose the base model with PeftModel to enable LoRA.

**Training both LoRA and classifier:**

1. By default, only LoRA parameters are trained:
2. This means the pre-trained parts and the newly added classifier won’t learn.
3. To train the classifier as well:
4. Use the modules\_to\_save setting to include it in the training.
5. This also saves those parameters when you save the model.
6. And then the train using the parameter efficient model

model\_name = checkpoint.split("/")[-1]

training\_args = TrainingArguments(

output\_dir="outputs",

learning\_rate=5e-4,

num\_train\_epochs=50,

per\_device\_train\_batch\_size=4,

per\_device\_eval\_batch\_size=2,

save\_total\_limit=3,

evaluation\_strategy="epoch",

save\_strategy="epoch",

logging\_steps=5,

remove\_unused\_columns=False,

push\_to\_hub=True,

label\_names=["labels"],

)

trainer = Trainer(

model=lora\_model,

args=training\_args,

train\_dataset=train\_ds,

eval\_dataset=test\_ds,

compute\_metrics=compute\_metrics,

)

trainer.train()