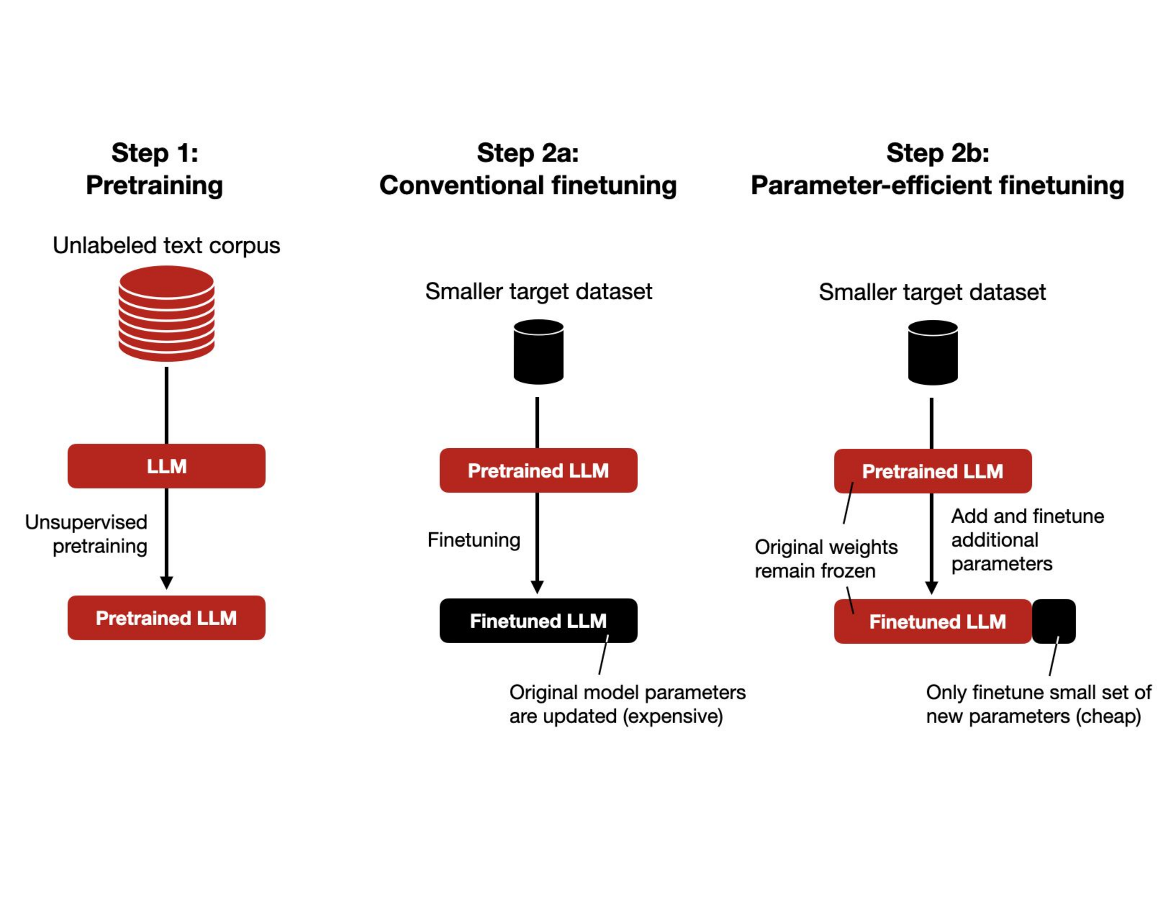
# Pre-training, Full and Parameter Efficient Fine-Tuning LLMs

Dipanjan (DJ) Sarkar

# Pre-training vs Full Fine-tuning vs Parameter Efficient Fine-tuning




Analytics Vidhya

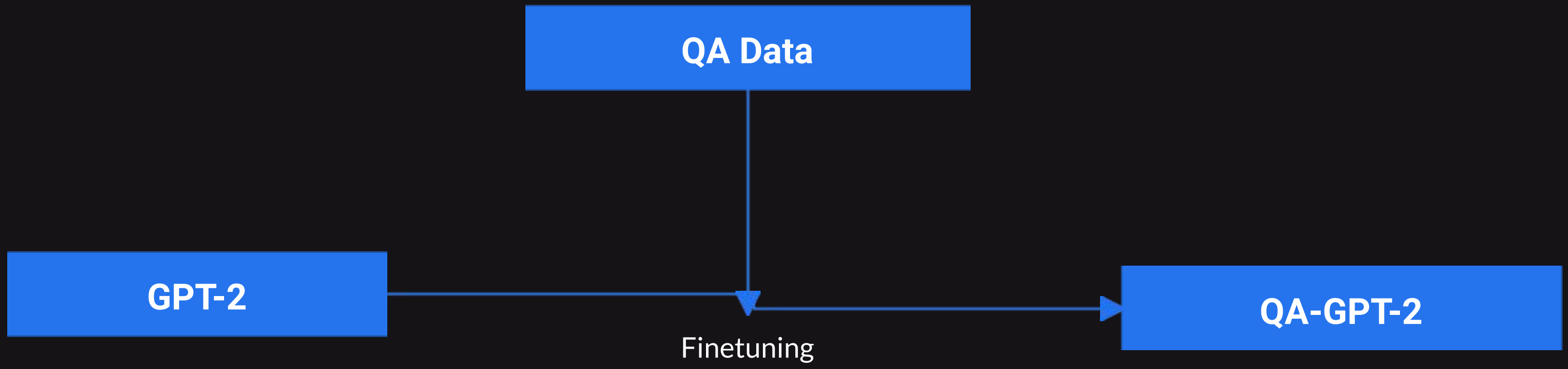
## Why Finetuning LLMs?

Pretrained LLMs are trained on massive internet datasets.

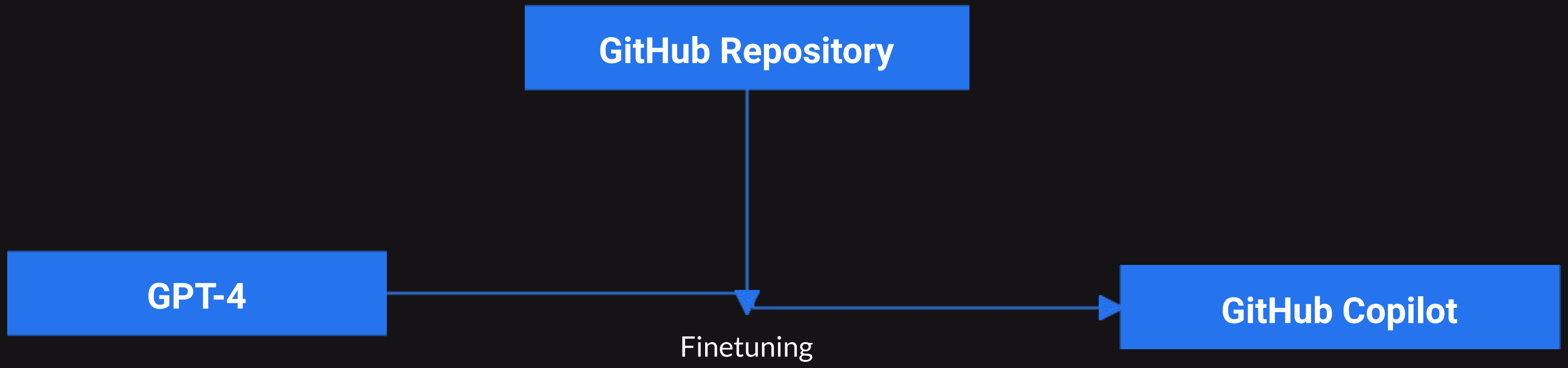
# Reason

Parameters and Domain Adaptation

# Why Finetuning LLMs?

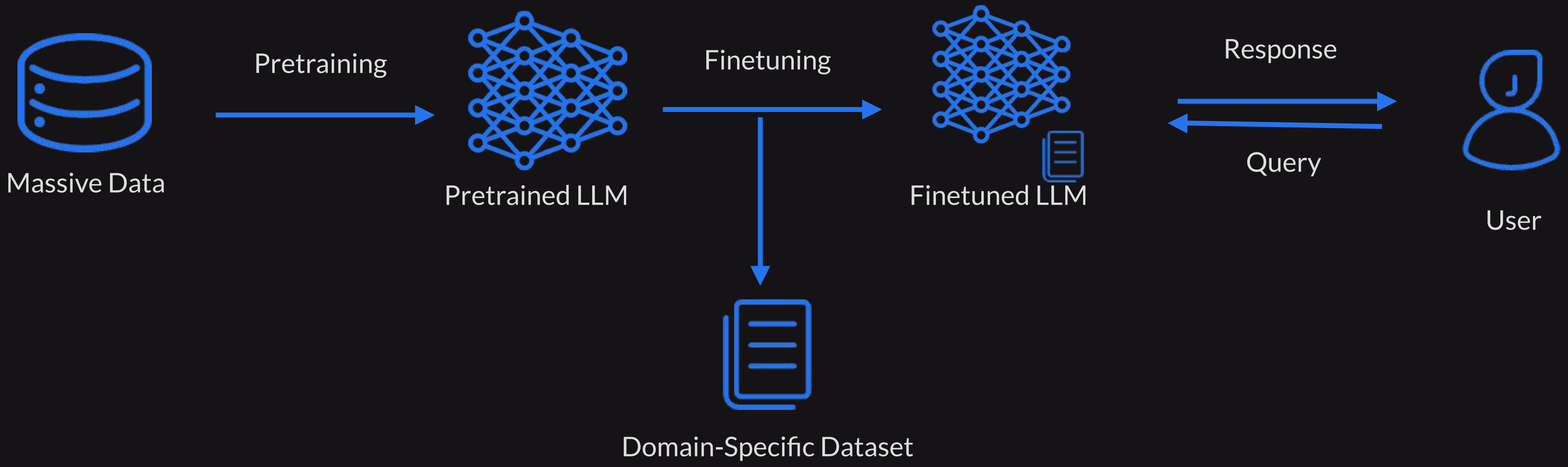


# Why Finetuning LLMs?



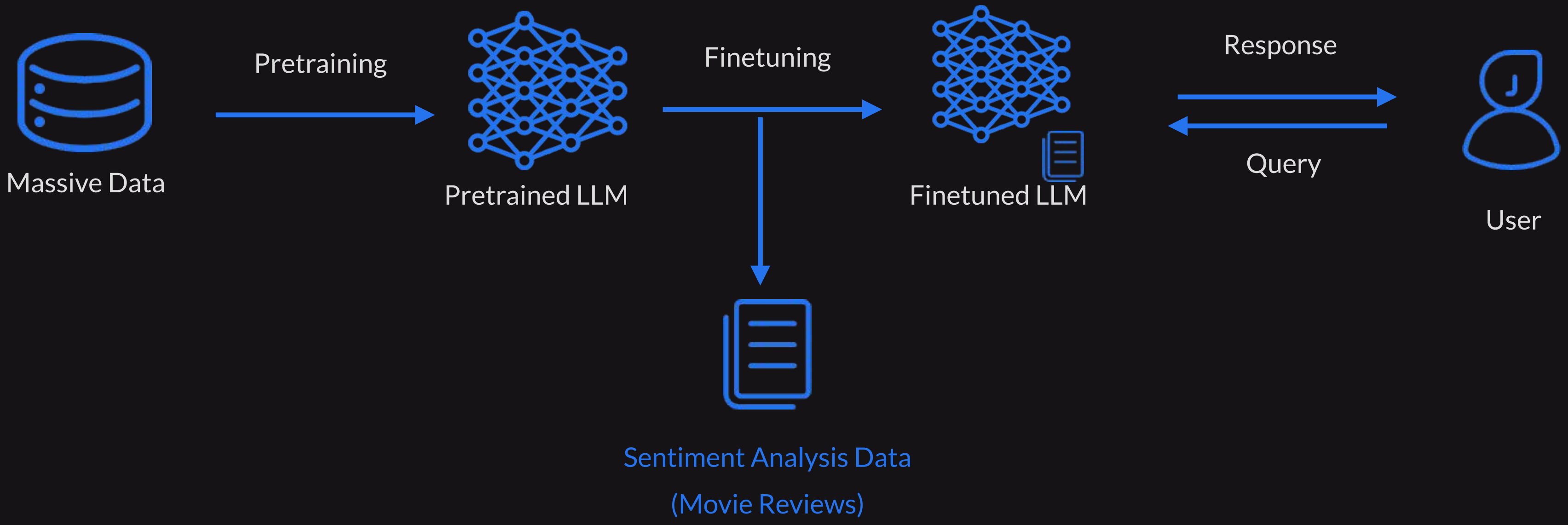

Analytics Vidhya

# What is Finetuning LLMs?




Analytics Vidhya

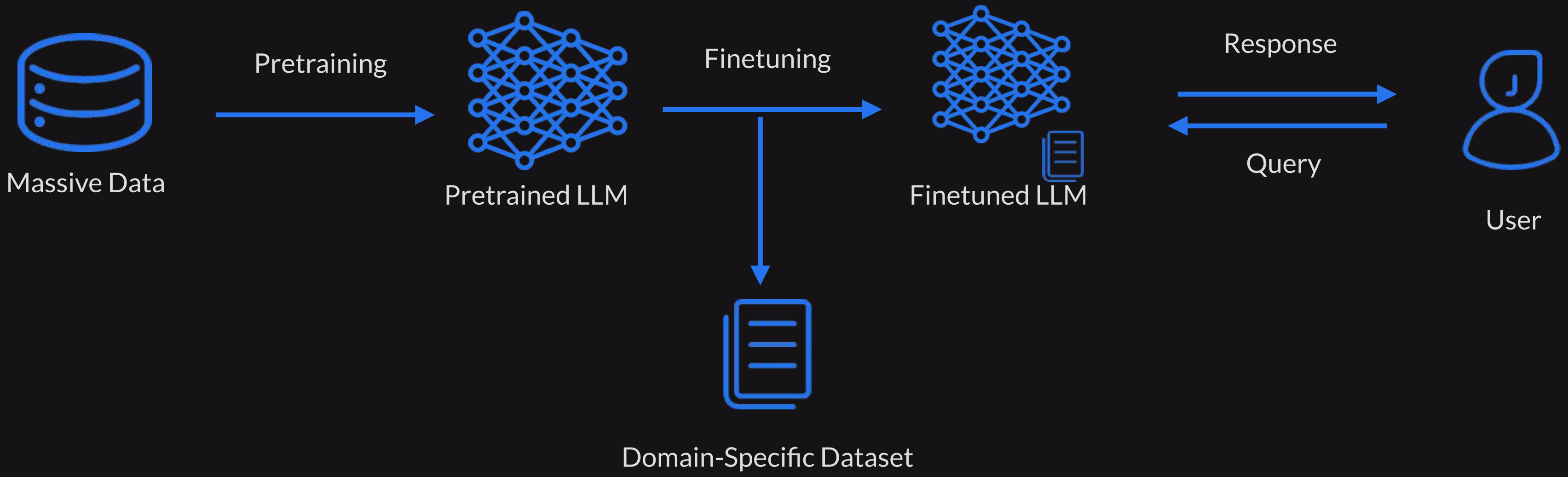
# What is Finetuning LLMs?



# Methods to Finetune LLMs

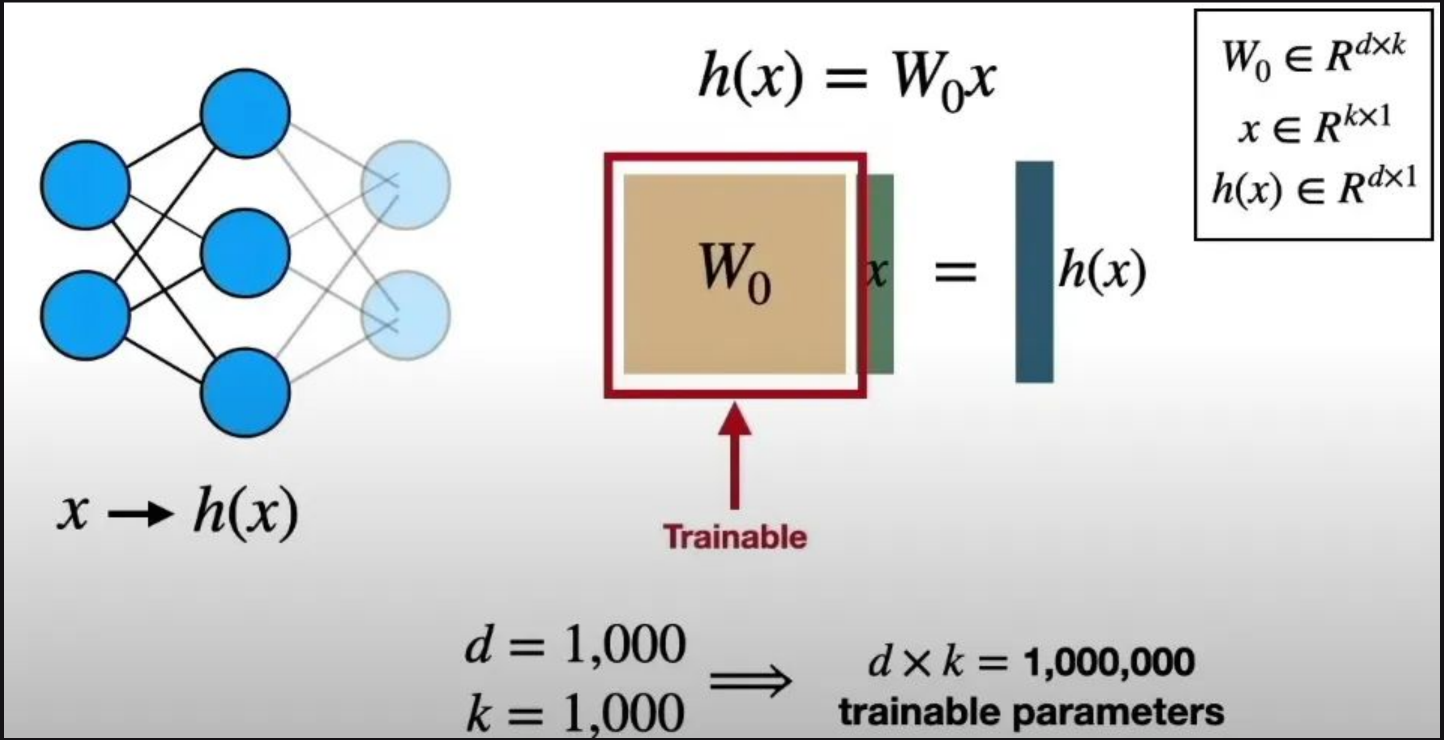
# Full Finetuning

Parameters of the entire model are retrained on the domain specific datasets.



# Full Finetuning

Parameters of the entire model are retrained on the domain specific datasets.

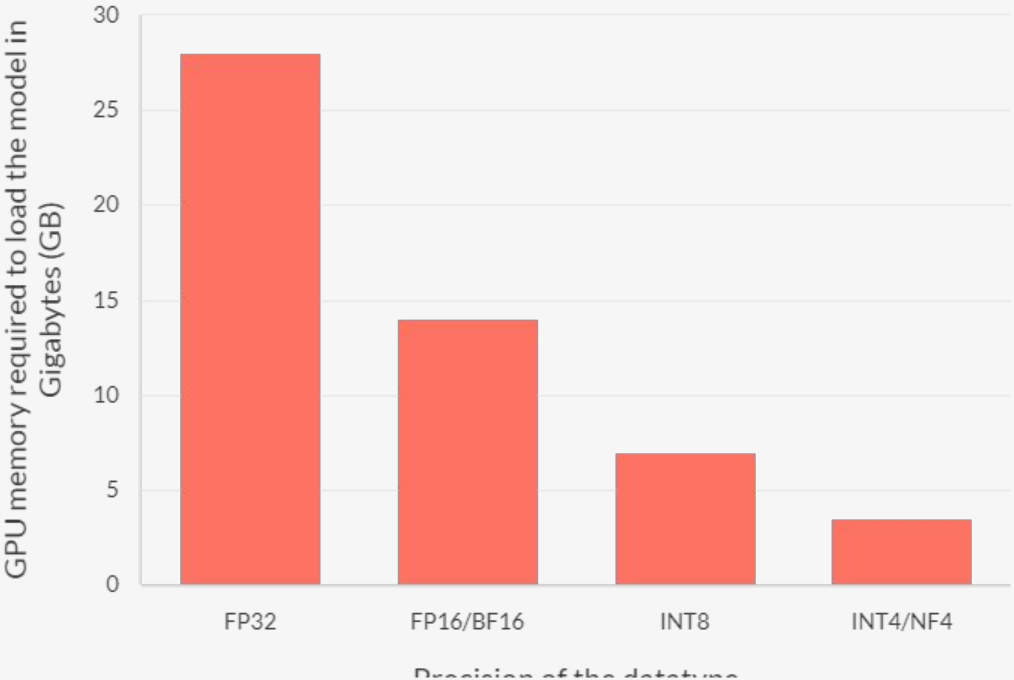


# What makes a model large?

## Two Factors

1. Number of parameters
2. Precision of the data type

Let’s put some numbers wrt Mistral-7B model with 7 Billion parameters



# Why full finetuning is expensive?

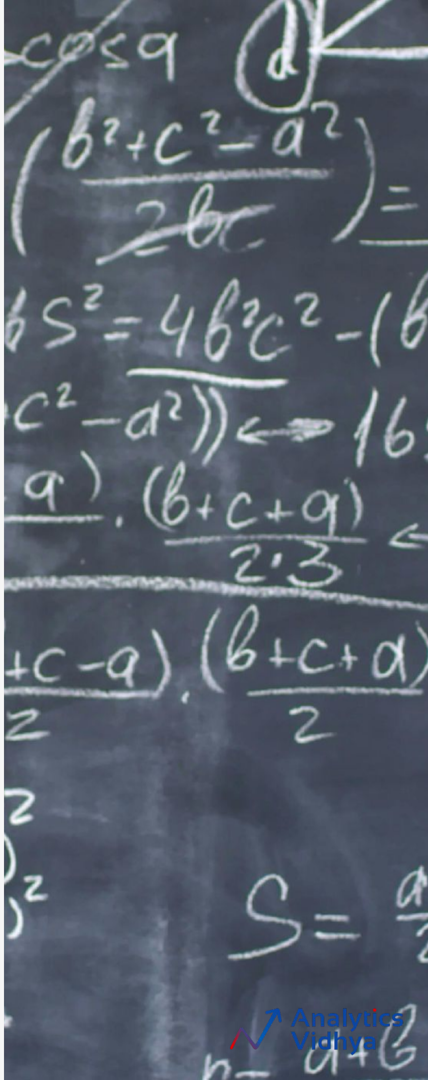
*Finetuning Mistral-7B in mixed-precision using Adam Optimizer*

Weights - 2 bytes / parameter

Gradients - 2 bytes / parameter

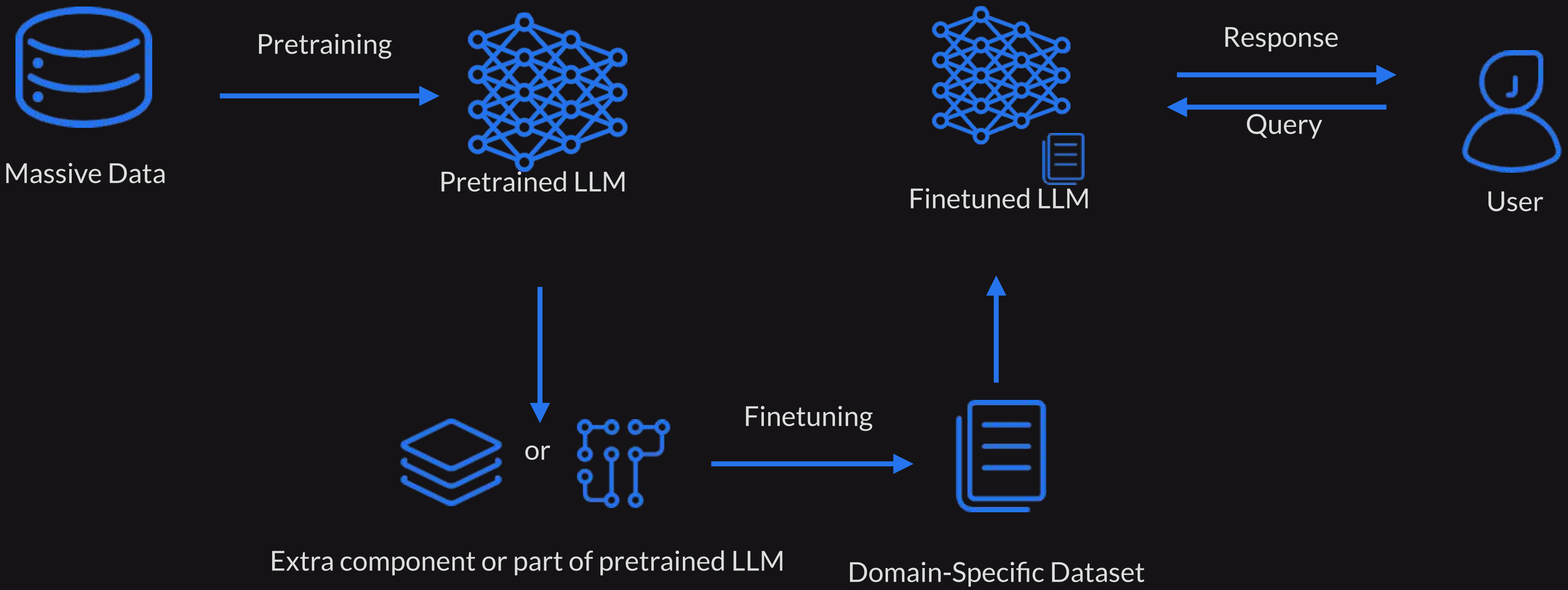
Optimizer state - 4 bytes / parameter (FP32 copy) + 8 bytes / parameter (momentum & variance estimates)

Total training cost: 16 bytes/parameter * 7 billion parameters = 112 GB😢



# Parameter Efficient Fine-tuning

Fraction of parameters are retrained on the domain specific dataset.



# Popular PEFT Techniques

- Low -rank adaptation methods
  - LoRA
  - QLoRA
  - LoHA
  - LoKr
  - AdaLoRA
- Prompt-based Methods
  - Prompt tuning
  - Prefix tuning
  - P tuning

Analytics Vidhya

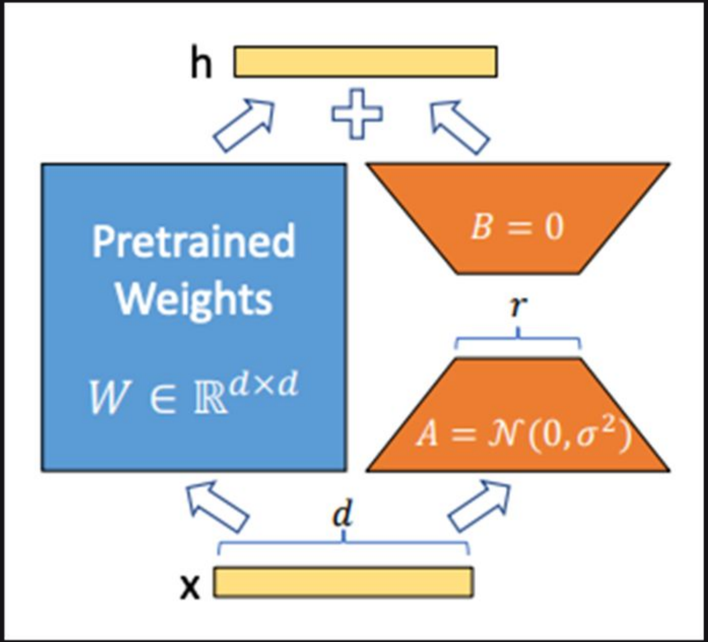
# Pros

- Ample task specific data
- Consistent and Budget-friendly
- High performance

# Cons

- High quality training data
- Never as good as full-finetuning in most cases
- Domain specific pre-trained LLM is better

# Low - Rank Adaptation (LoRA) Overview



# What led to LLMs? Transformers

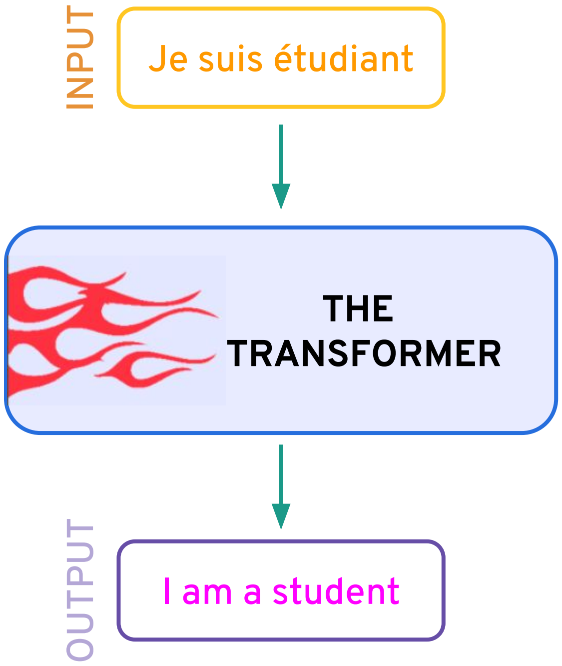
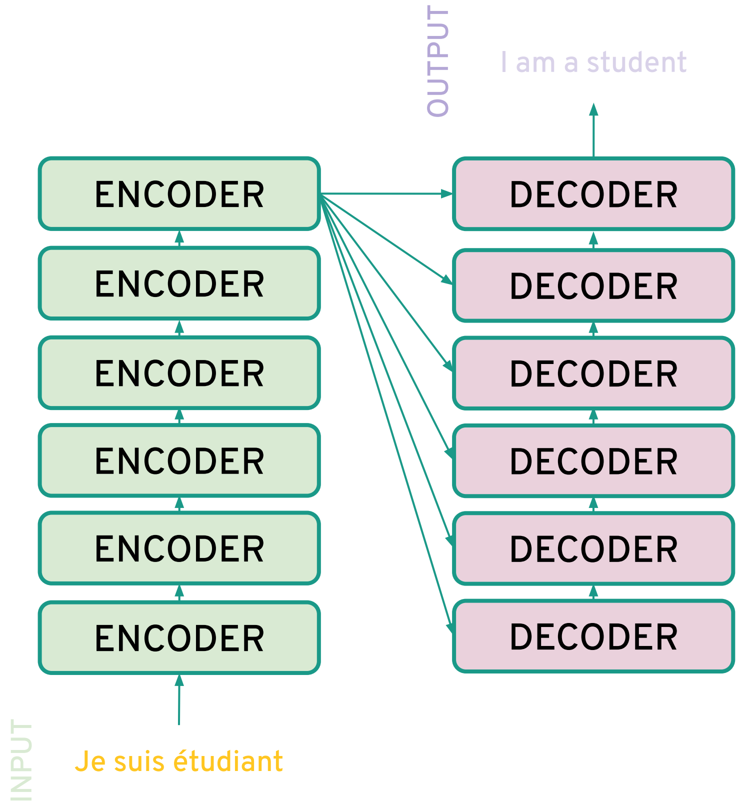



*Image Credits: Udacity*

- Layered & Stacked Encoder-Decoder Model
- Relies on Multi-headed Self-Attention & Encoder-Decoder Attention
- No sequential RNN-based training (completely parallelized)
- Achieved state-of-the-art performance on several NLP tasks

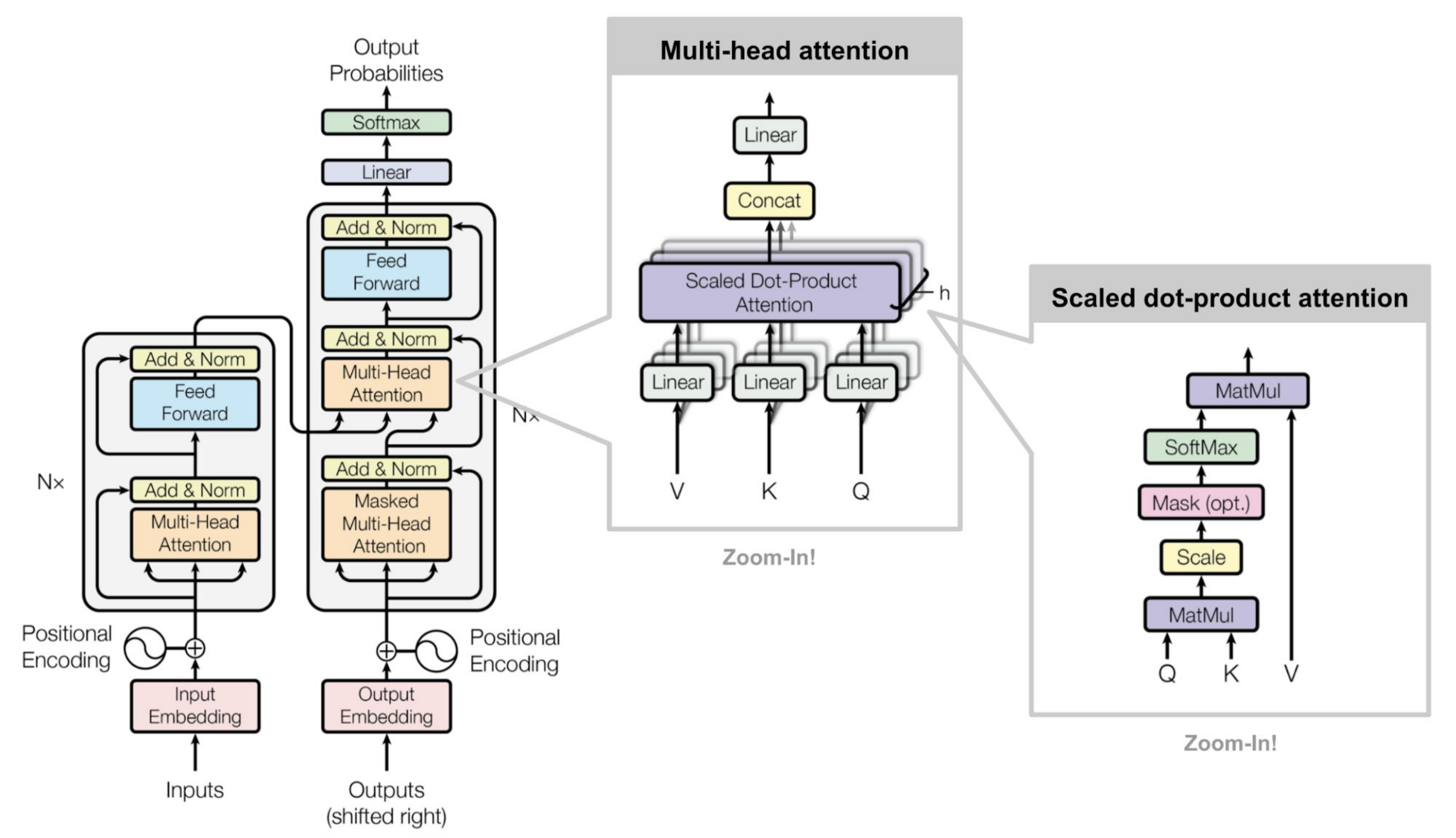
# Transformer Model Architecture



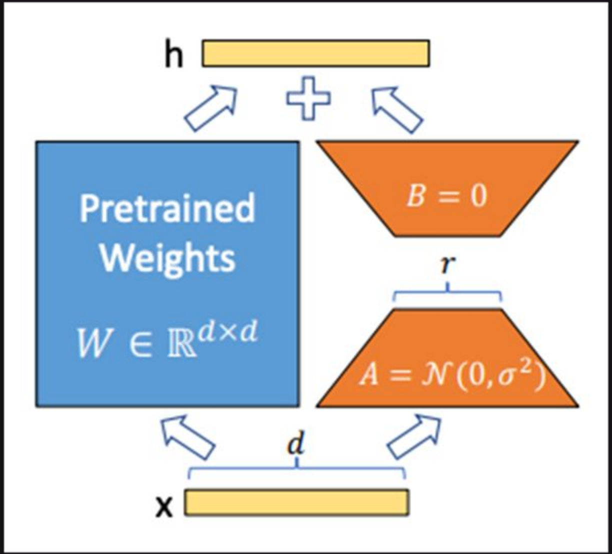

- At heart, a transformer model is a stacked encoder-decoder model
- Has multiple stacked encoder & decoder blocks usually based on standard architectures
- Based on the type of transformer model, only encoder/decoder (or both) are used

*Adapted from figure at: https://jalammar.github.io/illustrated-transformer/*

# Transformer Architecture - Key Layer Weights



# Low - Rank Adaptation (LoRA) Overview

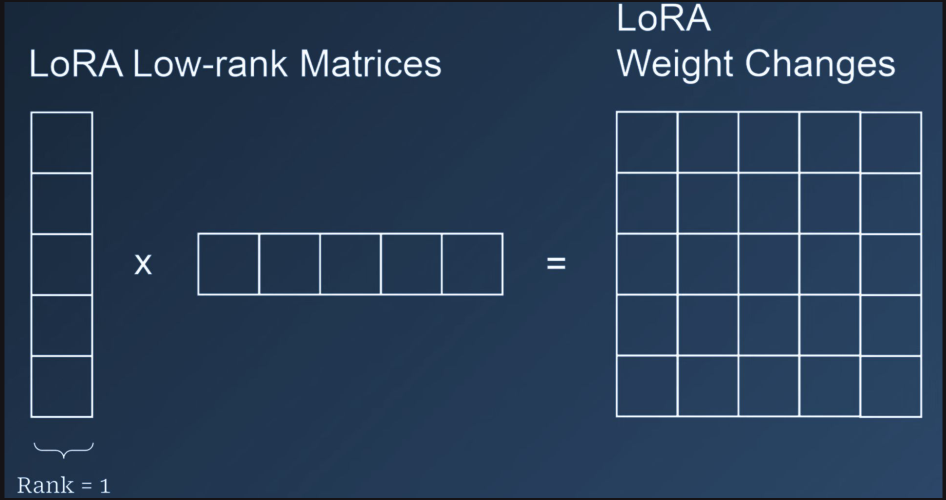


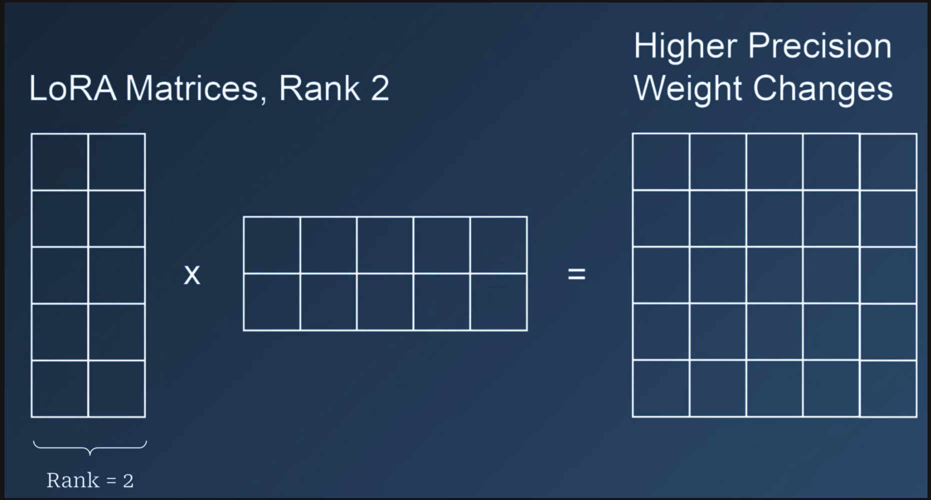

```python
class LoraLayer(torch.nn.Module):
    def __init__(self, base_layer, r, alpha, f_in, f_out):
        super().__init__()
        self.base_layer = base_layer
        self.scaling = alpha/r
        self.lora_A = torch.nn.Linear(f_in, r, bias=False)
        self.lora_B = torch.nn.Linear(r, f_out, bias=False)

    def forward(self, x, *args, **kwargs):
        lora_output = self.lora_B(self.lora_A(x)) * self.scaling
        return self.base_layer(x, *args, **kwargs) + lora_output
```

Analytics Vidhya

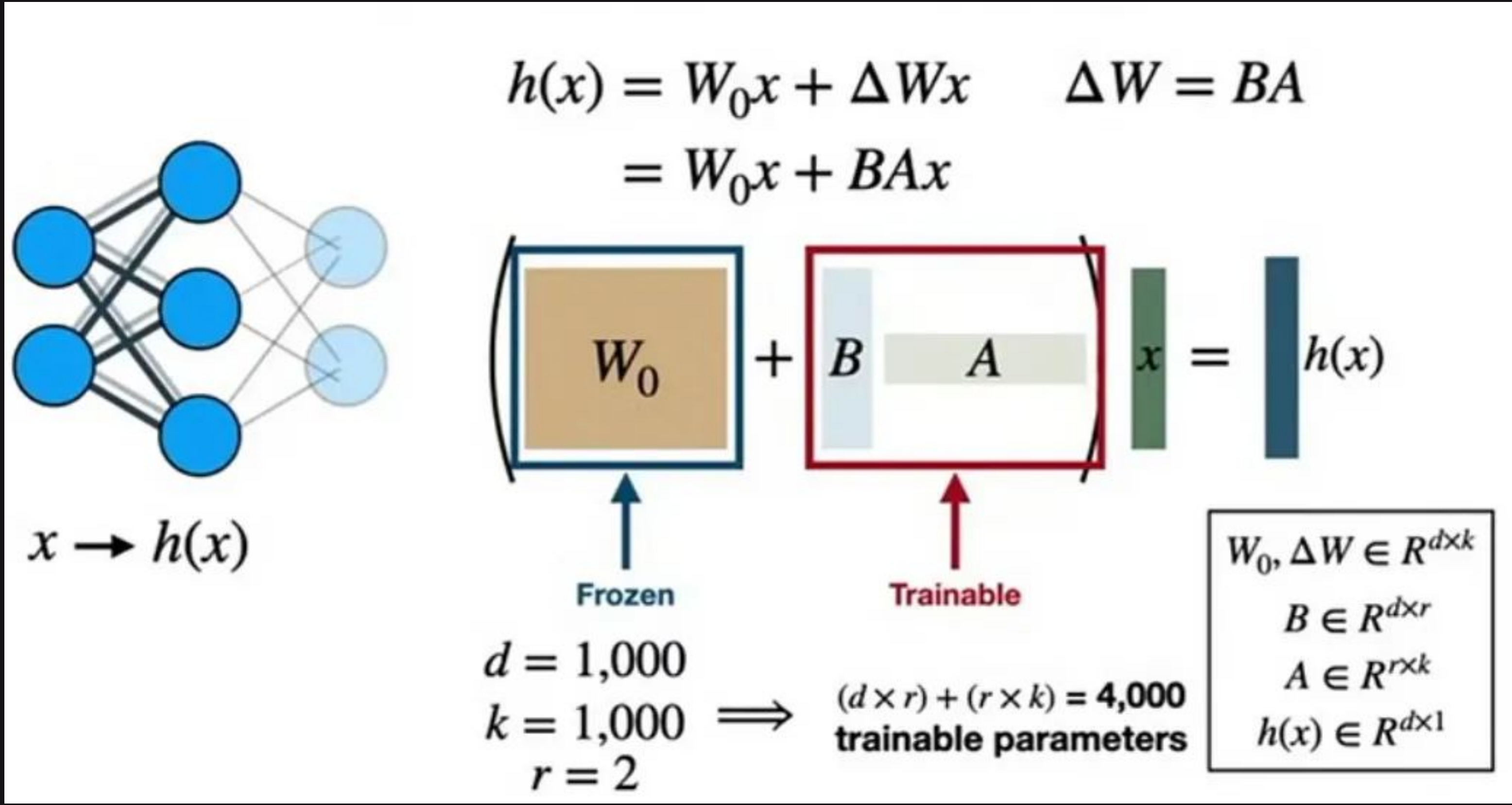
# Low - Rank Adaptation (LoRA) Overview




Analytics Vidhya

# Low - Rank Adaptation (LoRA) Overview




Analytics Vidhya

# Low - Rank Adaptation (LoRA) Overview

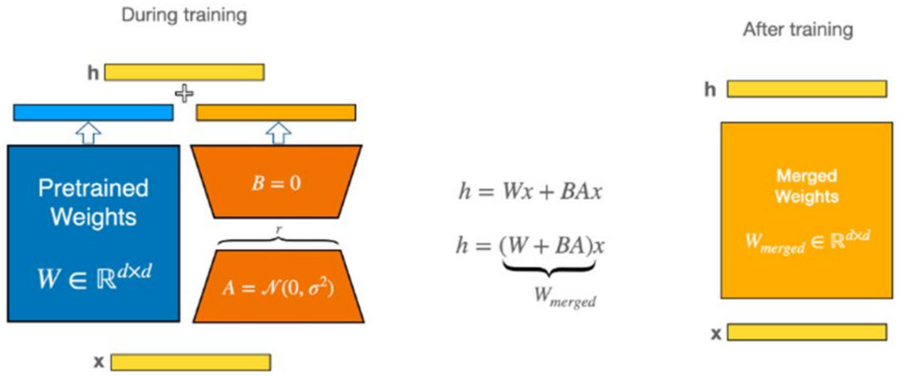



FIGURE ADAPTED FROM ORIGINAL PAPER, FIGURE 1 AND SECTION 4.1

## No Additional Inference Latency

Step1: Train adapters adapted on your task

Step 2: Merge the adapter weights inside the base model and use it as a standalone model

# Low - Rank Adaptation (LoRA) - Key Hyperparameters

- `r`: the rank of the update matrices, expressed in `int`. Lower rank results in smaller update matrices with fewer trainable parameters.
- `target_modules`: The modules (for example, attention blocks) to apply the LoRA update matrices.
- `lora_alpha`: LoRA scaling factor.

```
from peft import LoraConfig, get_peft_model, TaskType

config = LoraConfig(
        r=8,
        lora_alpha=32,
        target_modules=["q_lin", "v_lin"],
        lora_dropout=0.05,
        bias="none",
        task_type=TaskType.SEQ_CLS
)

peft_model = get_peft_model(model, config)
print_trainable_parameters(peft_model)
```

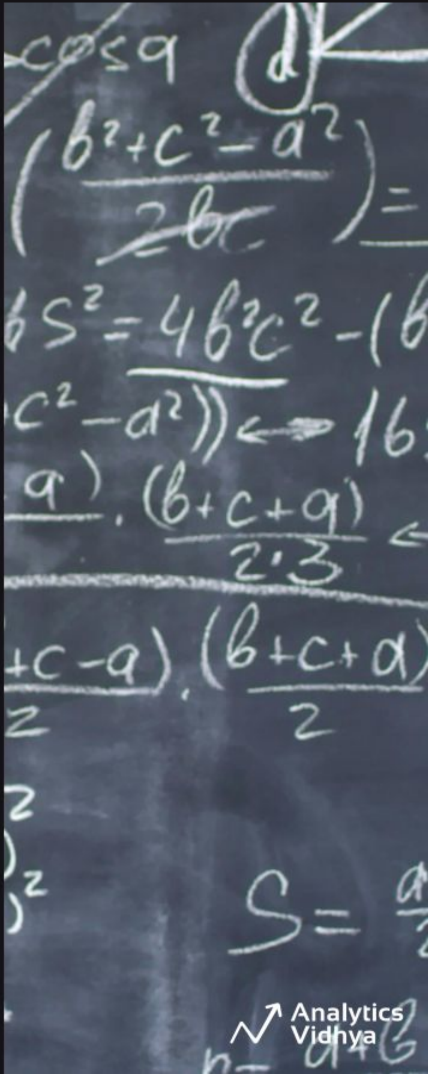
Analytics Vidhya

# LoRA Finetuning cost

*Finetuning Mistral-7B in mixed-precision using Adam Optimizer.*

trainable: 21,549,136 || all params: 7,263,322,192 || trainable%: 0.296

- Weights - 2 bytes / parameter
- Gradients - 2 bytes / parameter
- Optimizer state - 4 bytes / parameter (FP32 copy) + 8 bytes / parameter (momentum & variance estimates)
- Total training cost: 16 bytes/parameter * 7 billion parameters * 0.0029 + 14 = 112 * 0.00296 + 14 GB ~ 14.4 GB



# Quantized Low - Rank Adaptation (QLoRA) Overview

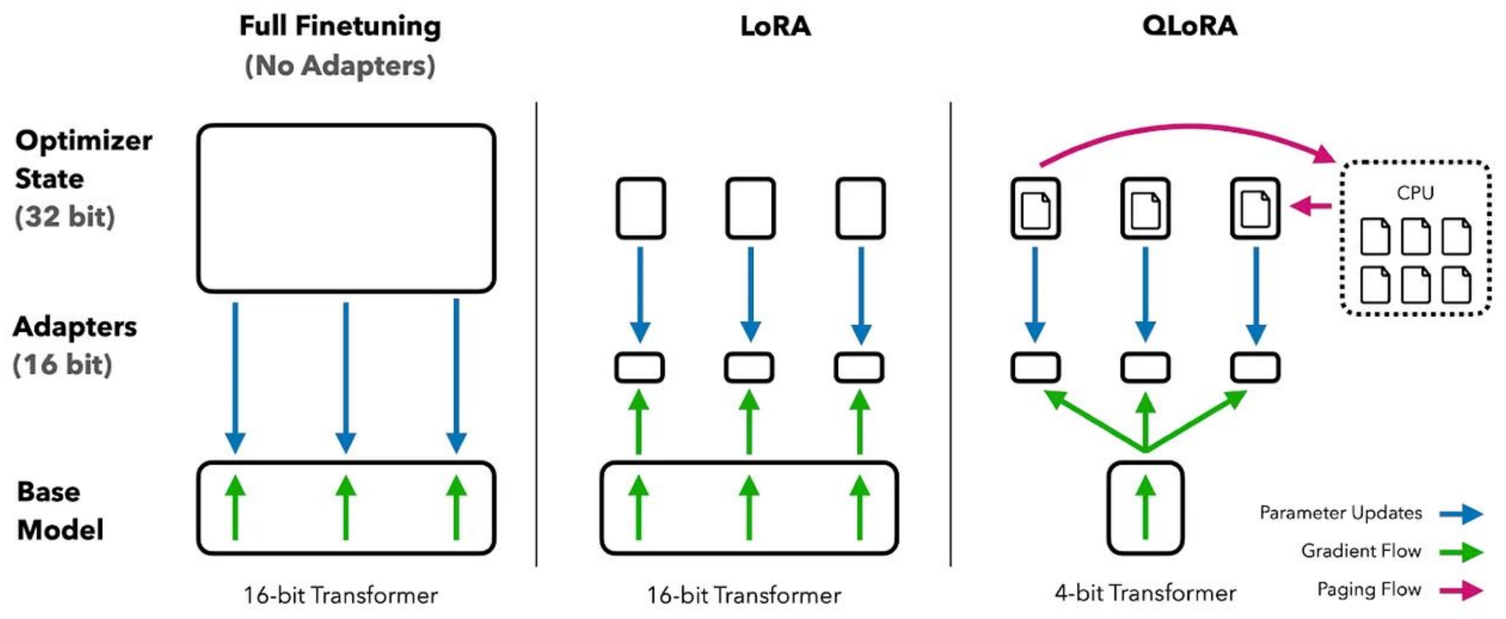


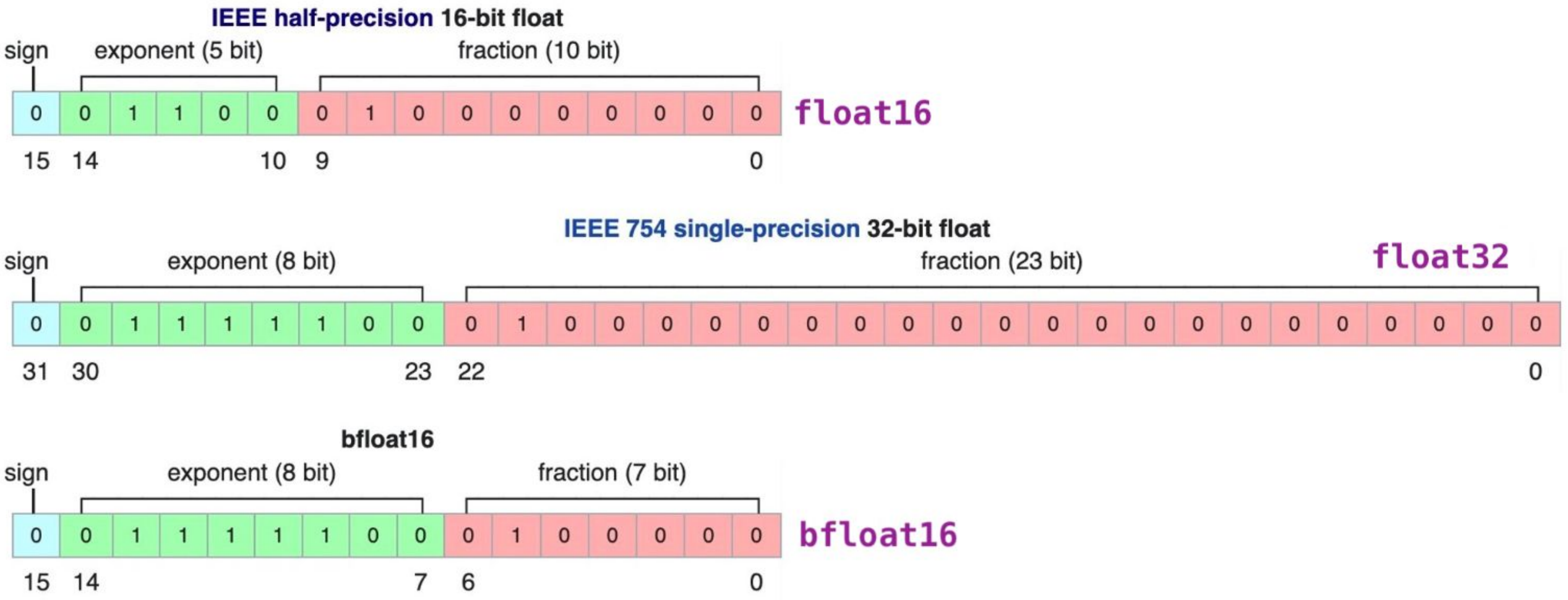

**Figure 1:** Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

Analytics Vidhya

# Quantized Low - Rank Adaptation (QLoRA) Overview

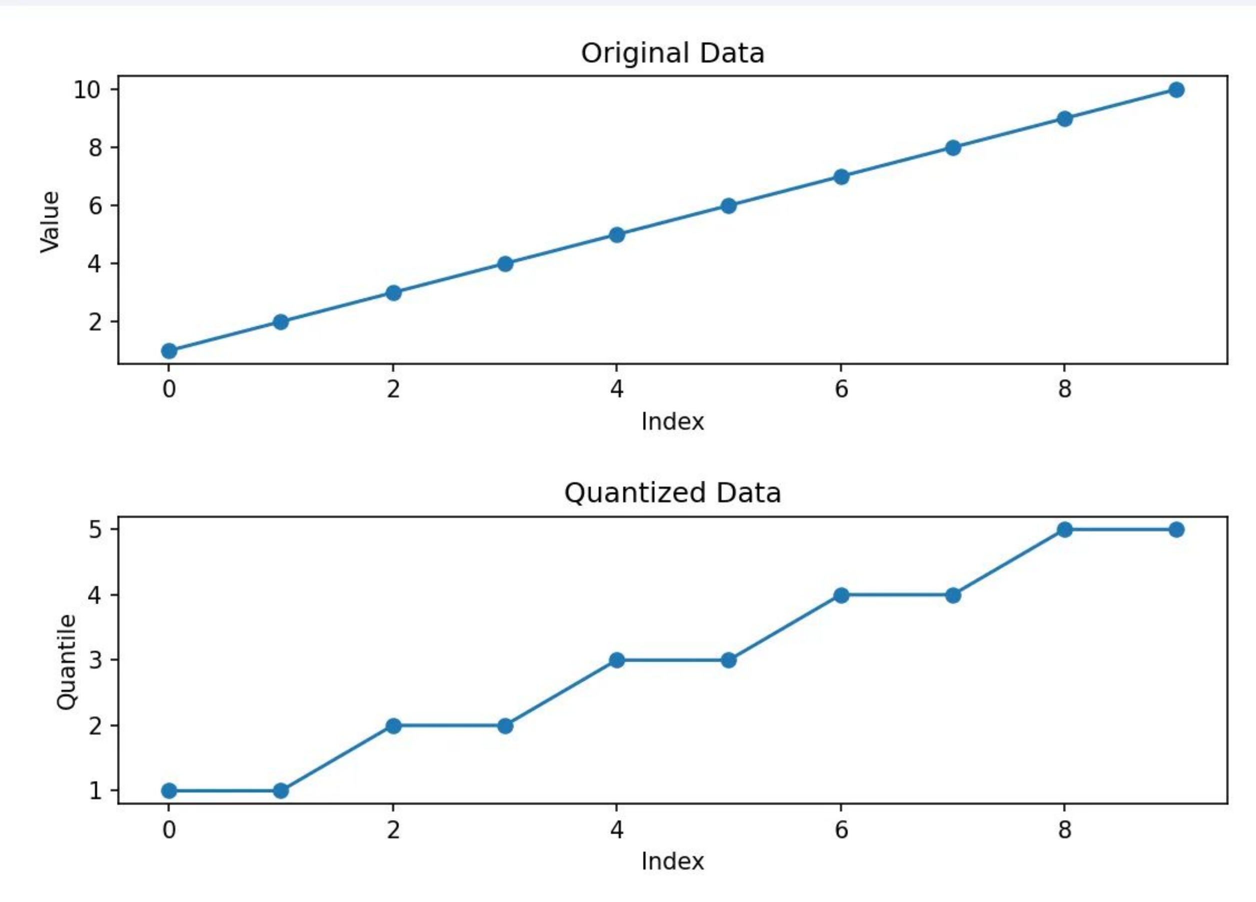
- Weights of the pre-trained LLM are first normalized using standard scaling
- Normalized weights are quantized to 4-bits, QLoRA usually uses a new data-type 4-bit normal float (NF4) to store them
- During fine-tuning - forward pass and backprop, the quantized weights are dequantized to full precision
- Weight updates are computed using normal LoRA method and uses 16-bit brain float (BF16) data type to represent weights during the fine-tuning
- Paged Optimizers utilize both CPU and GPU during fine-tuning

# Quantized Low - Rank Adaptation (QLoRA) Overview



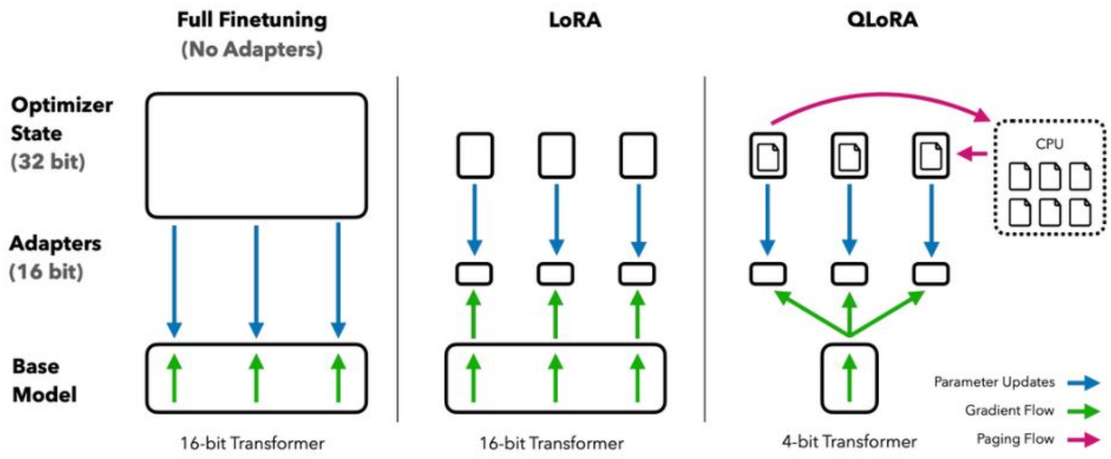

Analytics Vidhya

# Quantized Low - Rank Adaptation (QLoRA) Overview




Analytics Vidhya

# Quantized Low - Rank Adaptation (QLoRA) Overview




## Overview

- Asymmetric NF4 DType: [-1.0, -0.7, -0.53, -0.39, -0.28, -0.18, -0.09, 0.0, 0.08, 0.16, 0.25, 0.34, 0.44, 0.56, 0.72, 1.0]
- Double QuantizationPaged Optimizers
- QLoRA has one storage data type (NF4) and a computation data type (16-bit Bfloat). Dequantize the storage data type to the computation data type to perform the forward and backward pass, but only compute weight gradients for the LoRA parameters which use 16-bit Bfloat.
- LoftQ is a method to initialize LoRA weights such that the quantization error is minimized. Improves performance of QLoRA

Analytics Vidhya

# Quantized Low - Rank Adaptation (QLoRA) - Key Settings

```
import torch
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer, BitsAndBytesConfig

config = BitsAndBytesConfig(
    load_in_4bit=True, # quantize the model to 4-bits when you load it
    bnb_4bit_quant_type="nf4", # use a special 4-bit data type for weights initialized from a normal distribution
    bnb_4bit_use_double_quant=True, # nested quantization scheme to quantize the already quantized weights
    bnb_4bit_compute_dtype=torch.bfloat16 # use bfloat16 for faster computation
)

model = AutoModelForSequenceClassification.from_pretrained(model_checkpoint,
                                                           id2label=id2label,
                                                           label2id=label2id,
                                                           num_labels=2,
                                                           quantization_config=config)
```

# QLoRA Finetuning cost

*Finetuning Mistral-7B in mixed-precision using Adam Optimizer.*
trainable: 21,549,136 || all params: 7,263,322,192 || trainable%: 0.296

Weights - 0.5 bytes / parameter

Gradients - 2 bytes / parameter

Optimizer state - 4 bytes / parameter (FP32 copy) + 8 bytes / parameter (momentum & variance estimates)

Total training cost: 16 bytes/parameter * 7 billion parameters * 0.0029 + 14 =
112 * 0.00296 + 4 GB ~ 4.5 GB 🤗