AI VIETNAM
All-in-One Course
(TA Session)

Parameter-Efficient Fine-Tuning with LoRA

LLMs

Code

Thuan Duong – TA
Minh Nam Tran - STA

Year 2025

Outline

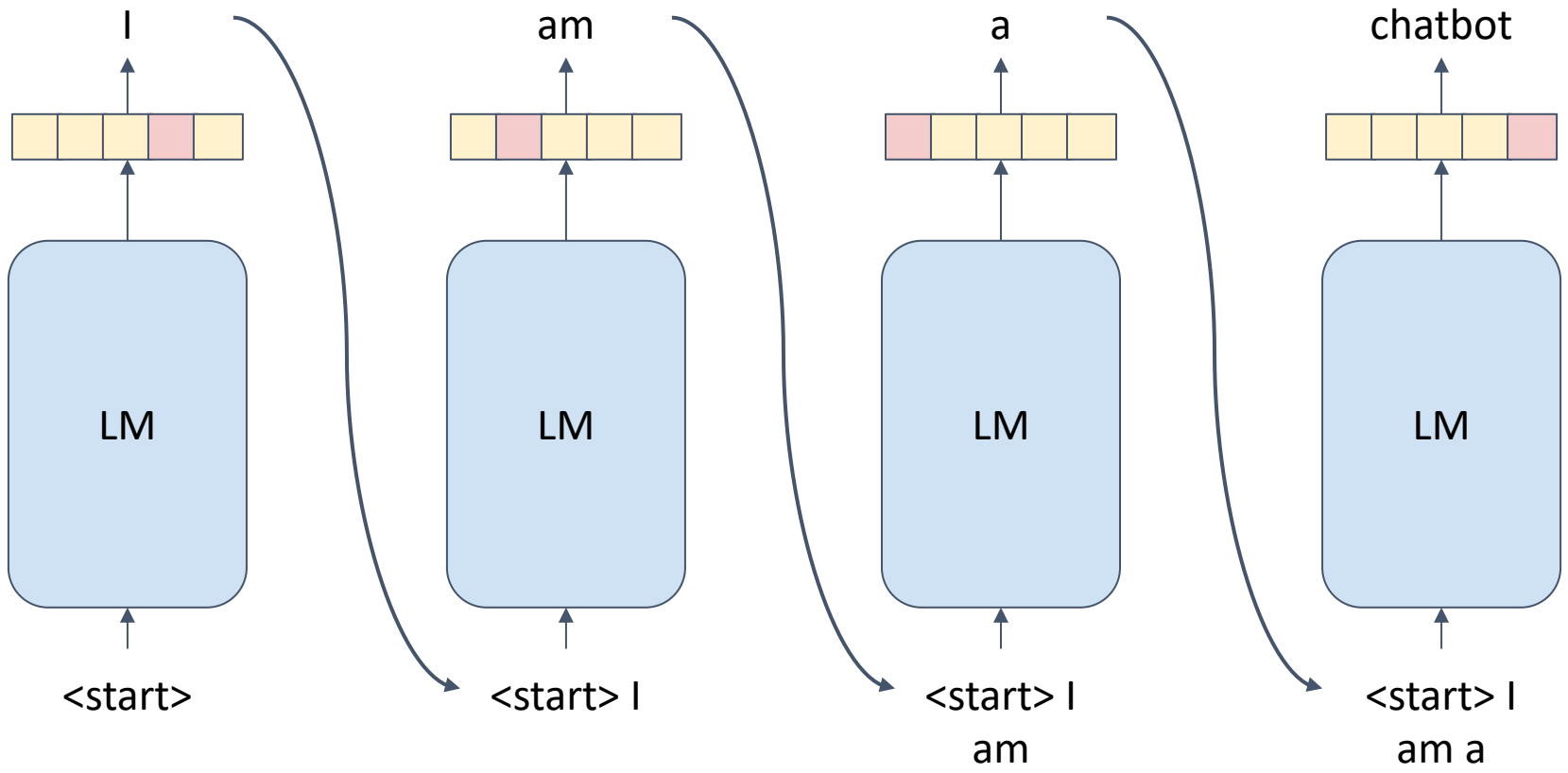
- Introduction
- LoRA
- QLoRA
- Practices

Introduction

Introduction

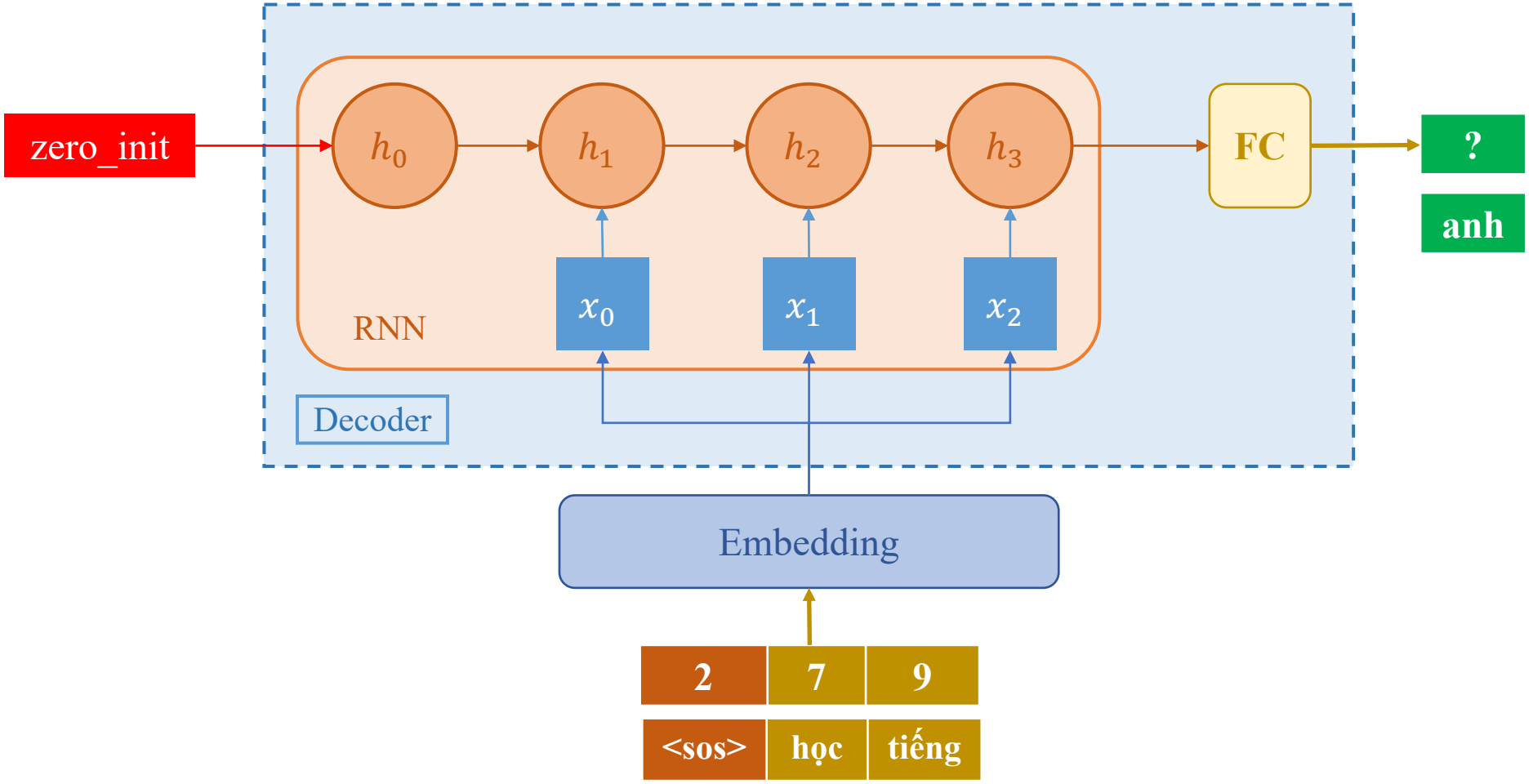
❖ What is Language Model (LM)?

A language model (LM) is a machine learning model trained to:
predict the next word (or token) in a sequence of text based on the previous context
” Pre-training GPT - Project”



Introduction

❖ Language Model: RNN

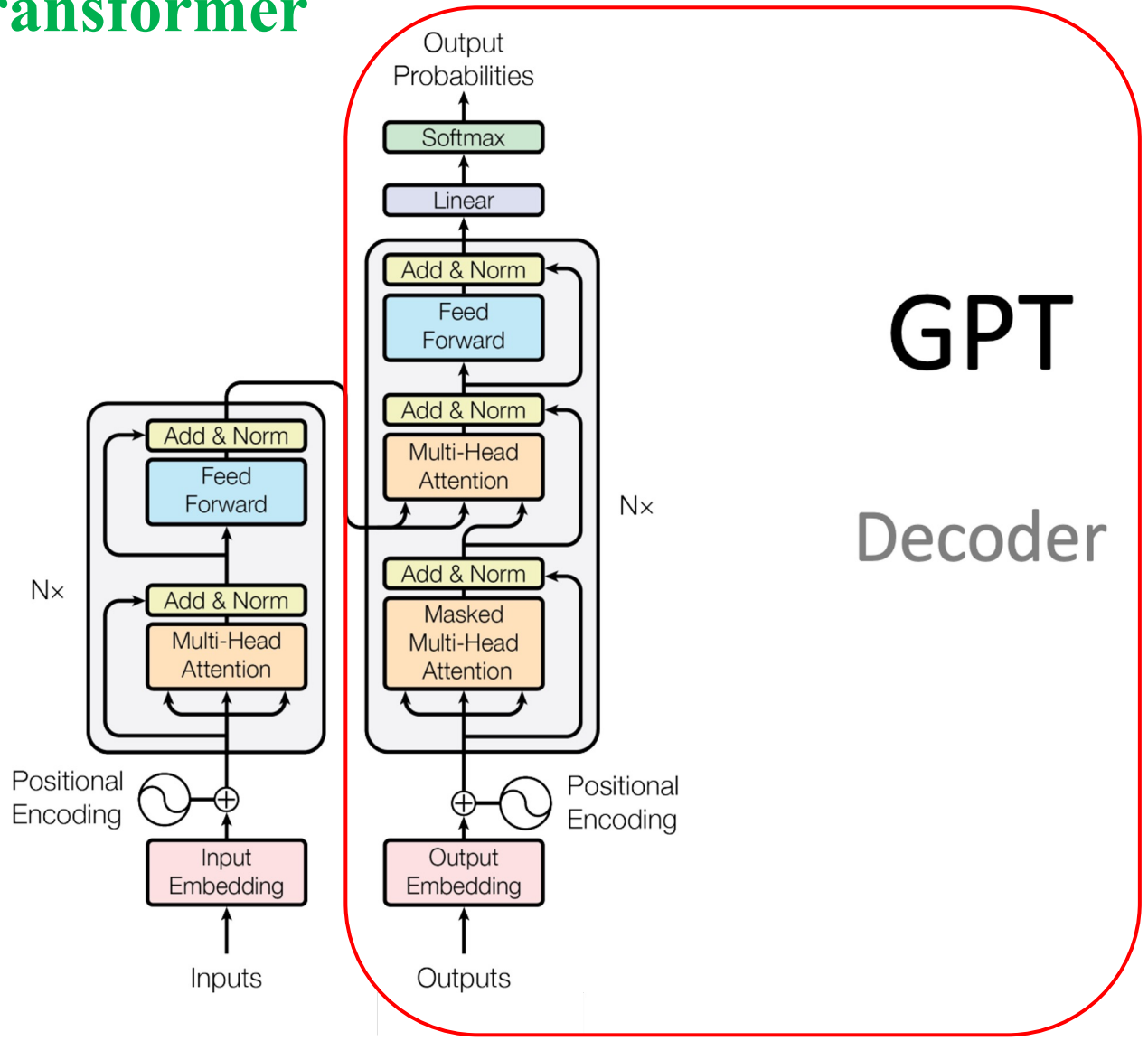


Introduction

❖ Language Model: Transformer

BERT

Encoder



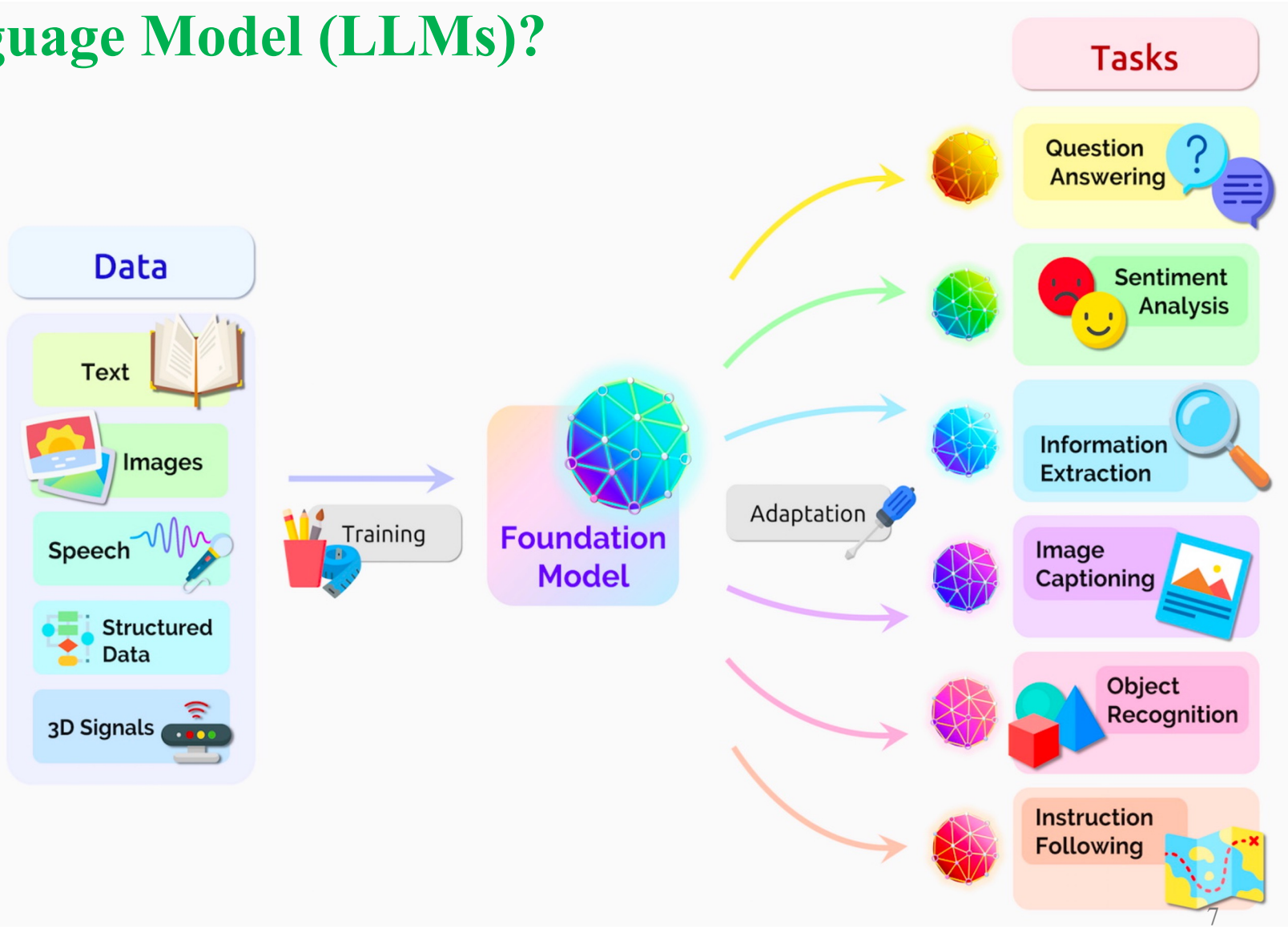
GPT

Decoder

Introduction

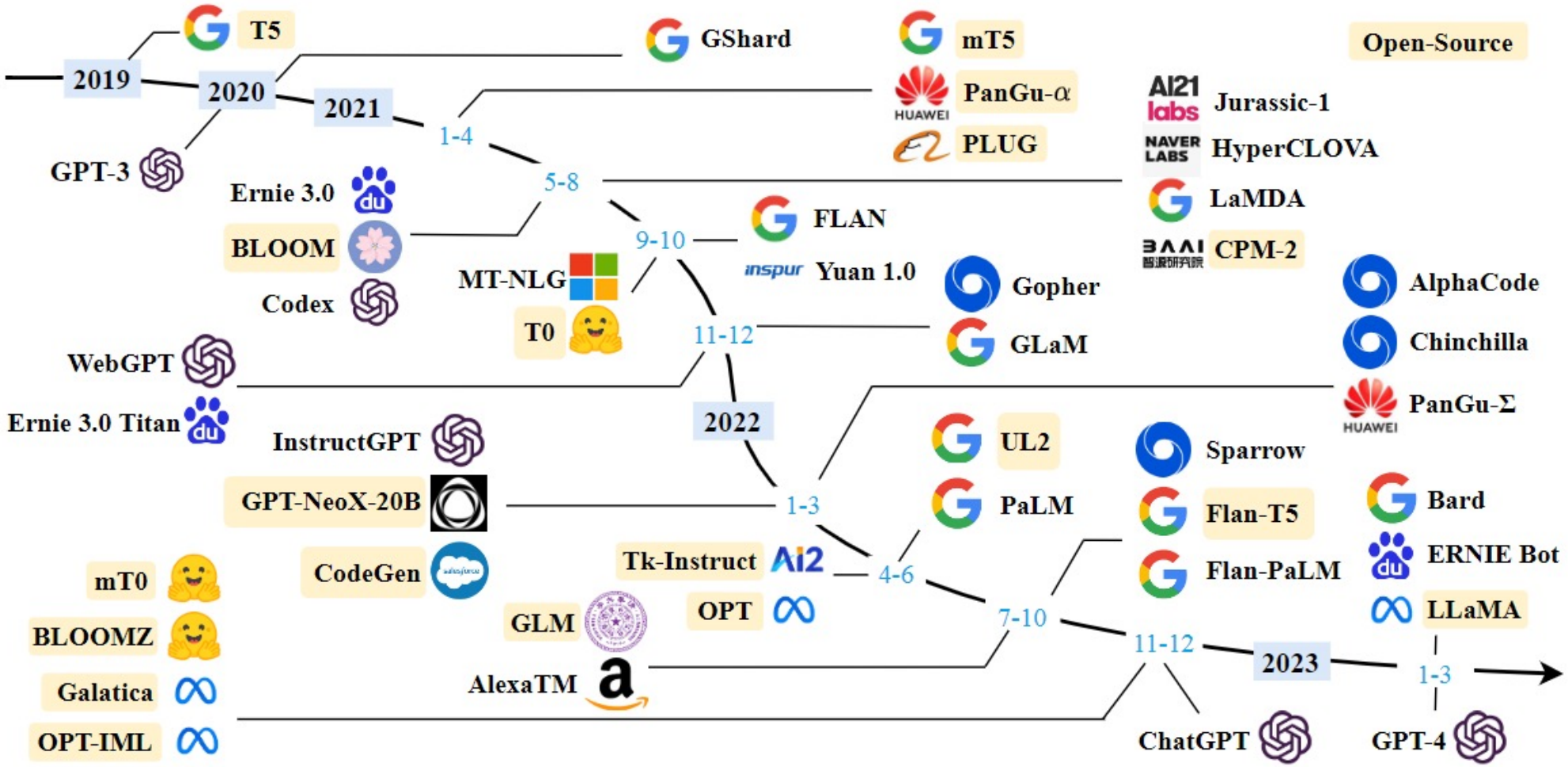
❖ What is Large Language Model (LLMs)?

LLMs (Large Language Models): Language models that were trained on a very large corpus of text. This made them capable of performing various NLP tasks with high precision.



Introduction

❖ LLMs



Ref image: https://wandb.ai/vincenttu/blog_posts/reports/A-Survey-of-Large-Language-Models--VmlldzozOTY2MDM1

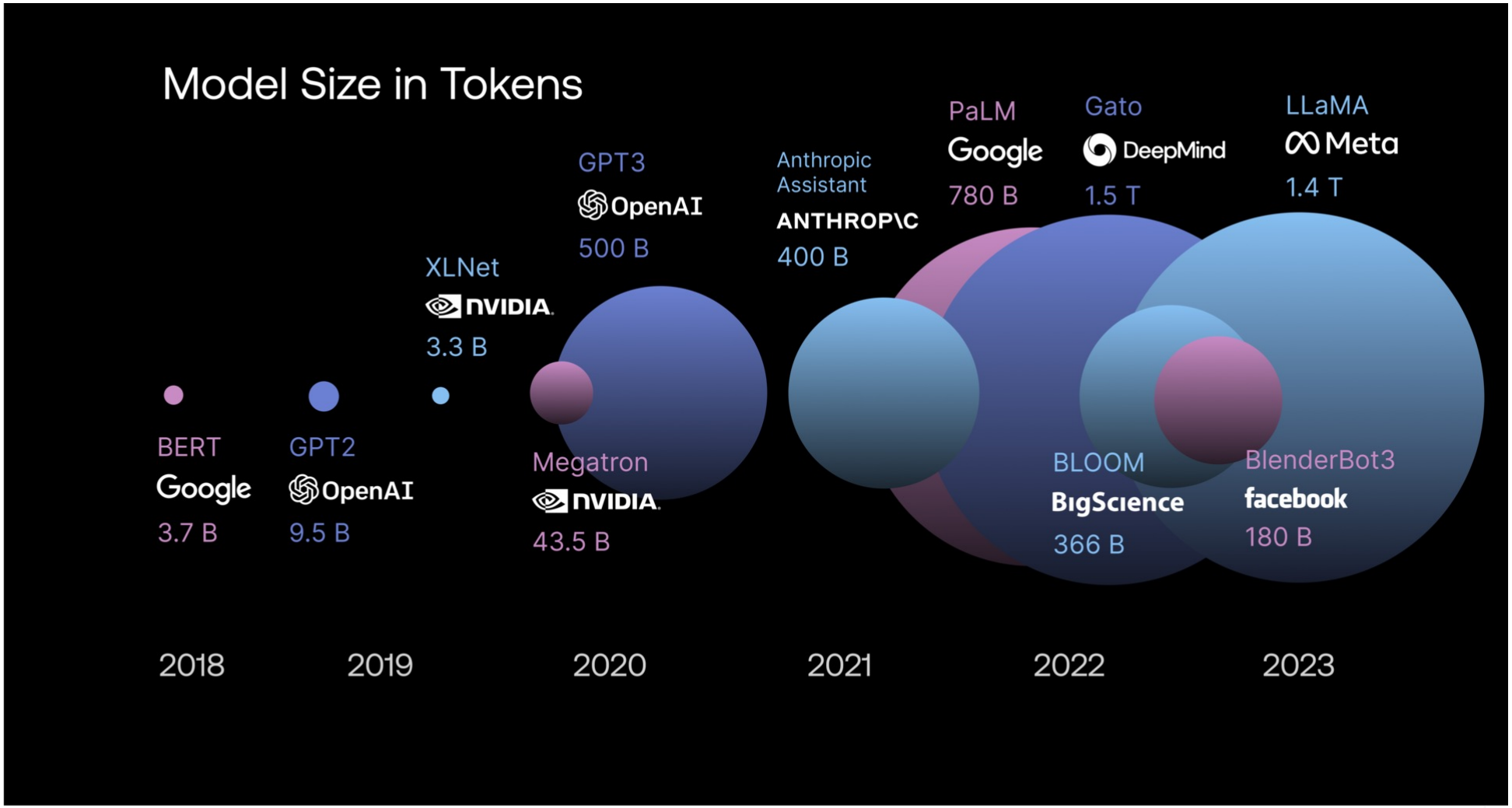
Introduction

❖ LMs vs LLMs

Feature	Language Model	Large Language Model
Dataset size	Small datasets	Huge datasets
Model size	About 1B	More B or T
Capabilities	Basic text tasks	Avanced task like coding, reasoning, multi-turn conversation
Released	BERT, GPT-2, T5	GPT-4; LLaMA 4, Claude,..

Introduction

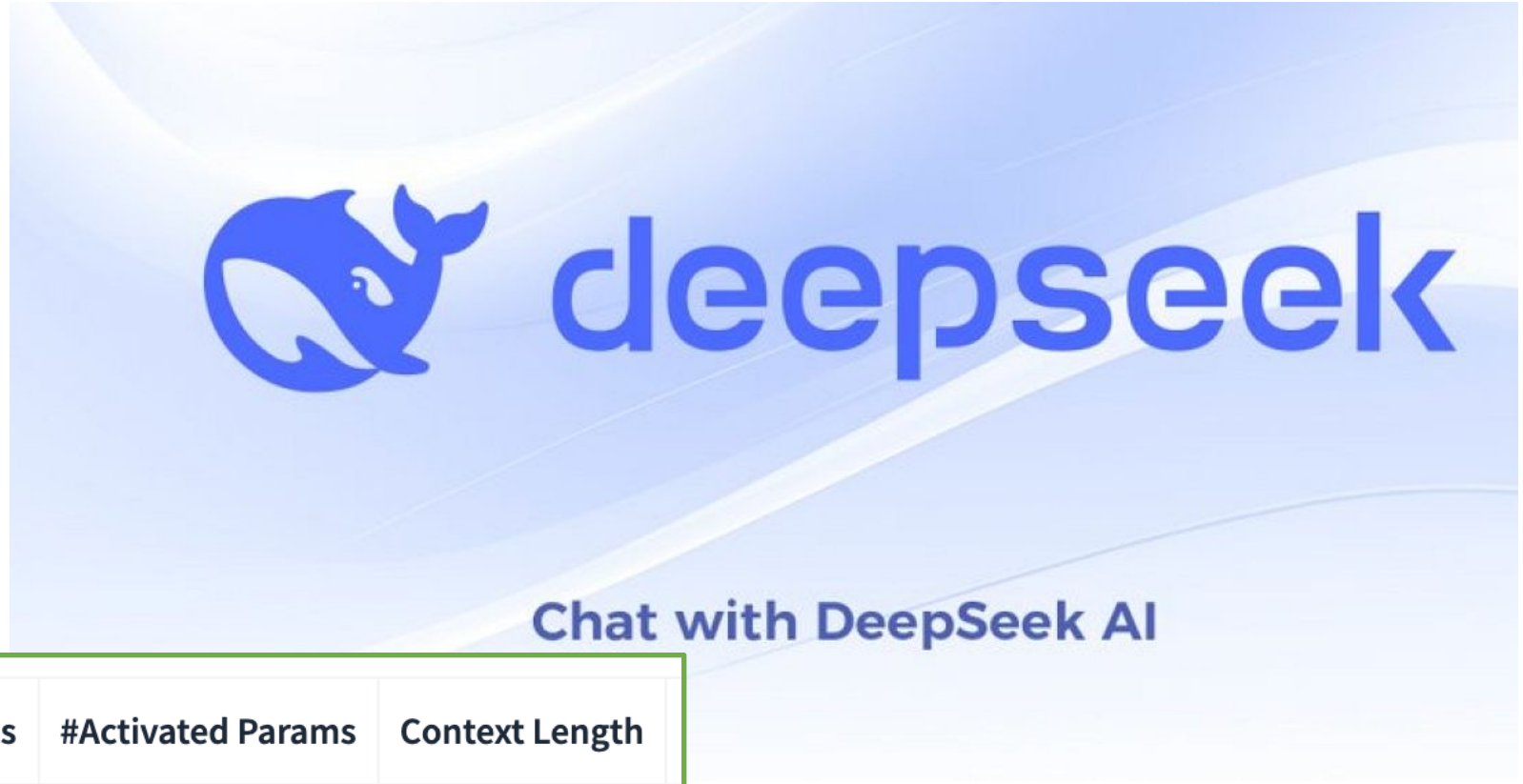
❖ Training tokens in LLMs



Ref imgage: https://datasci101.com/what-are-llms-part-1

Introduction

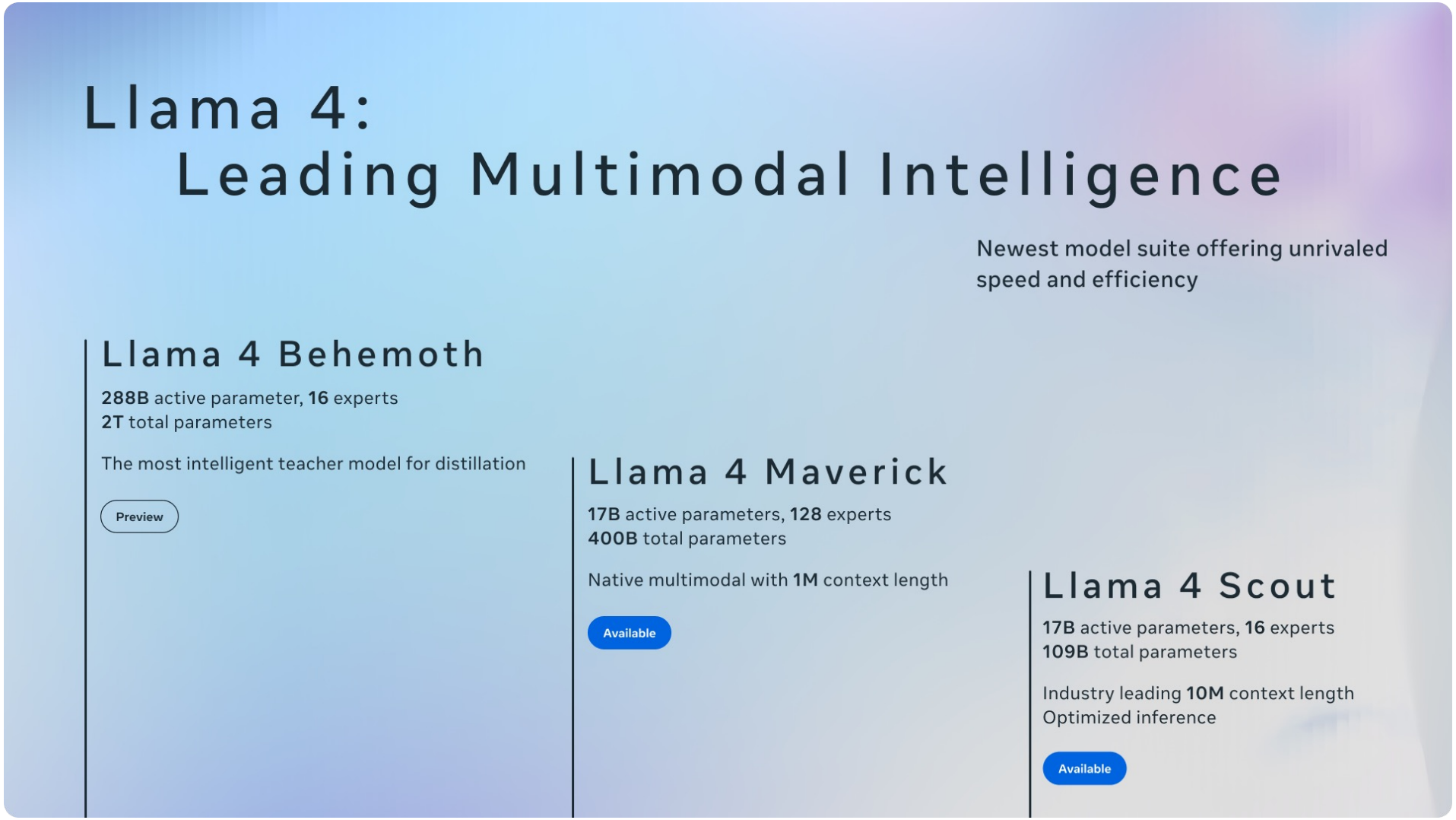
❖ New Released



Model	#Total Params	#Activated Params	Context Length
DeepSeek-R1-Zero	671B	37B	128K
DeepSeek-R1	671B	37B	128K

Introduction

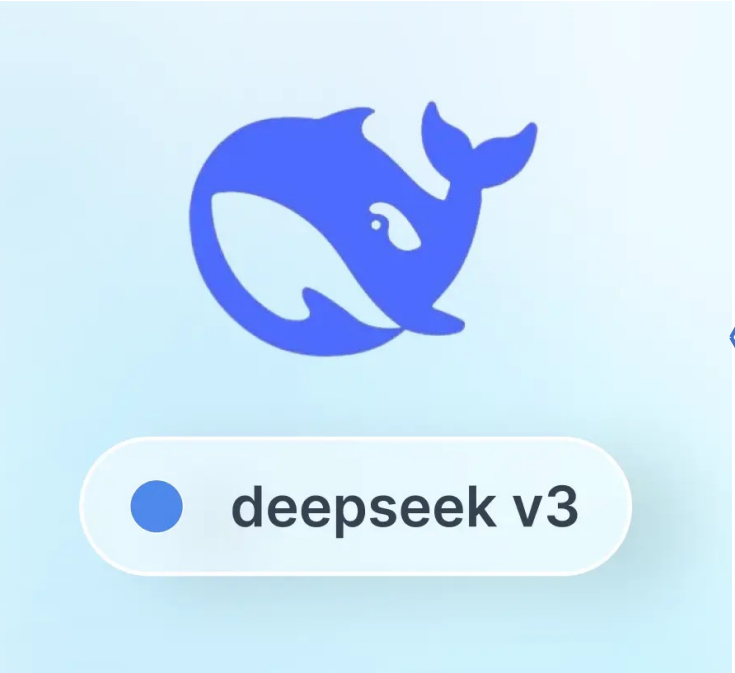
❖ New Released



Ref image: https://ai.meta.com/blog/llama-4-multimodal-intelligence

Introduction

❖ Pre-training: DeepSeek-V3



Training tokens	14.8T high-quality and diverse tokens
Model parameters	671B
GPU hours	2664K hours in H800 (80GB) GPU

Over 5.3M $

Introduction

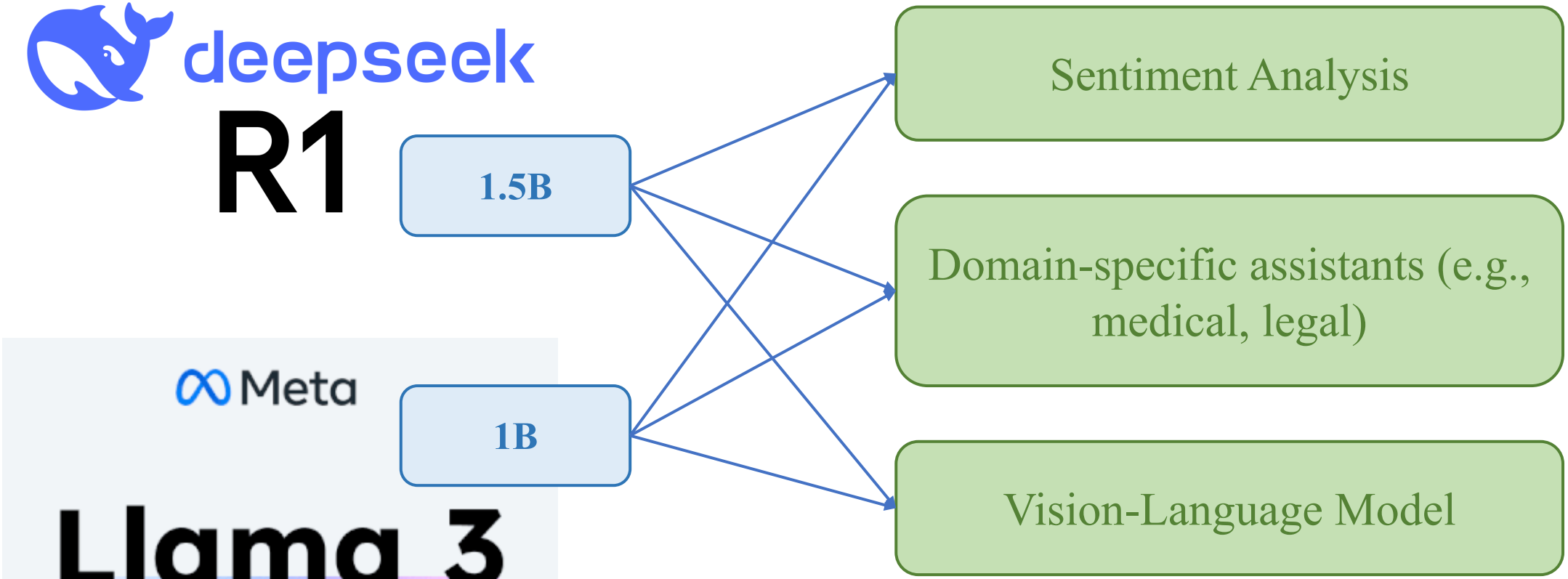
❖ Supervised Fine-Tuning (SFT)

1.5B	7B	8B
14B	32B	70B

1B	3B	8B
70B	80B	405B

Introduction

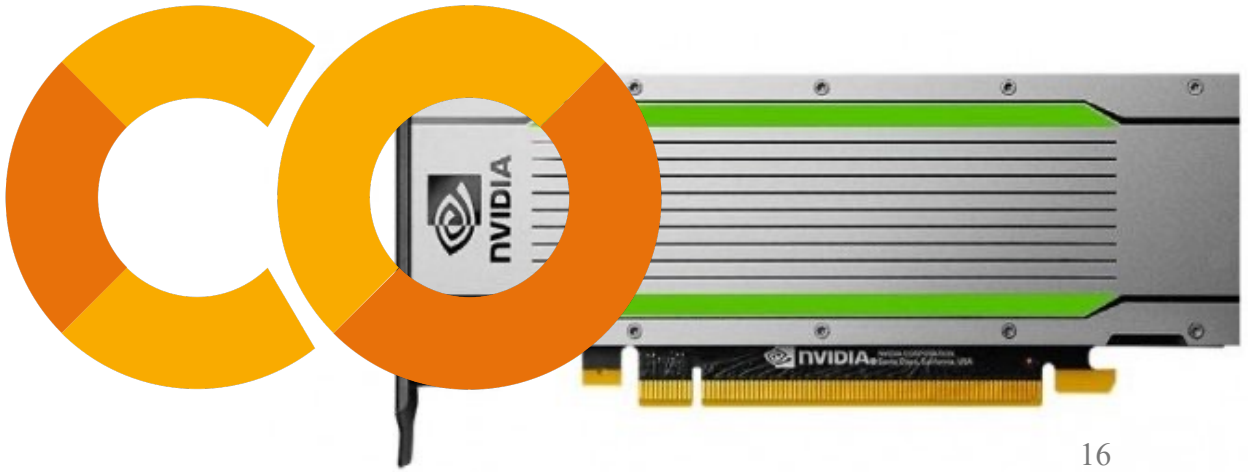
❖ Supervised Fine-Tuning (SFT)



Introduction

❖ GPU can Training LLM?

NVIDA GeForce RTX 2060 6GB vs **Colab** đâu hiệu quả hơn?
Em có card như trên và tính mua thêm các cái khác như memory, cpu, main gắn vào để training các kiểu đồ, học hành là chính.
Tiền trên thì mua Colab Pro thì có hiệu quả hơn ko các bác.
Ngoài ra tốc độ của card trên vs của Colab free thì có thể so sánh ntn để biết được sức mạnh của 2 cái trên các bác nhỉ?



Introduction

❖ GPU RAM (VRAM) need to store 1B parameters

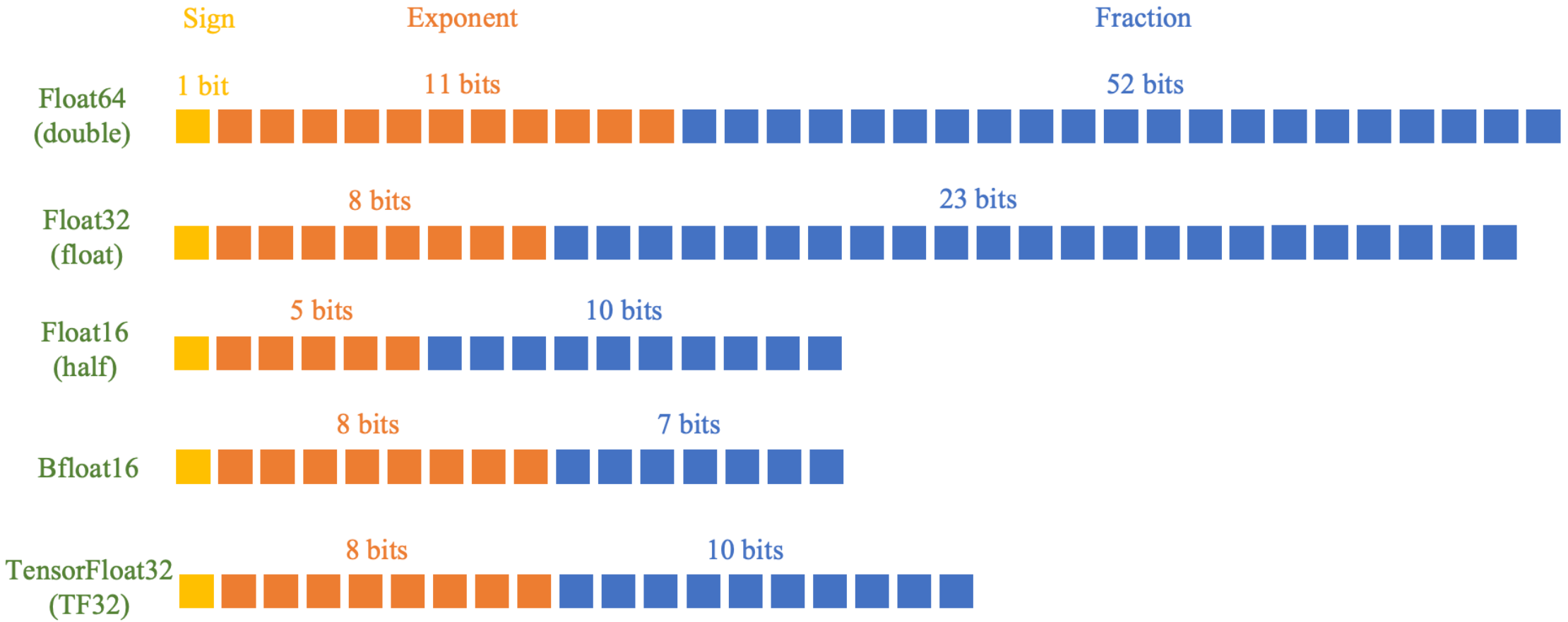
```
[ ]  1  from transformers import AutoTokenizer, AutoModelForCausalLM
     2
     3  tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-3.2-1B")
     4  model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-3.2-1B")
```

```
tokenizer_config.json: 100% 50.5k/50.5k [00:00<00:00, 1.28MB/s]
tokenizer.json: 100% 9.09M/9.09M [00:00<00:00, 12.8MB/s]
special_tokens_map.json: 100% 301/301 [00:00<00:00, 32.4kB/s]
config.json: 100% 843/843 [00:00<00:00, 93.8kB/s]
model.safetensors: 100% 2.47G/2.47G [00:12<00:00, 257MB/s]
generation_config.json: 100% 185/185 [00:00<00:00, 16.7kB/s]
```

2.47GB VRAM ?

Introduction

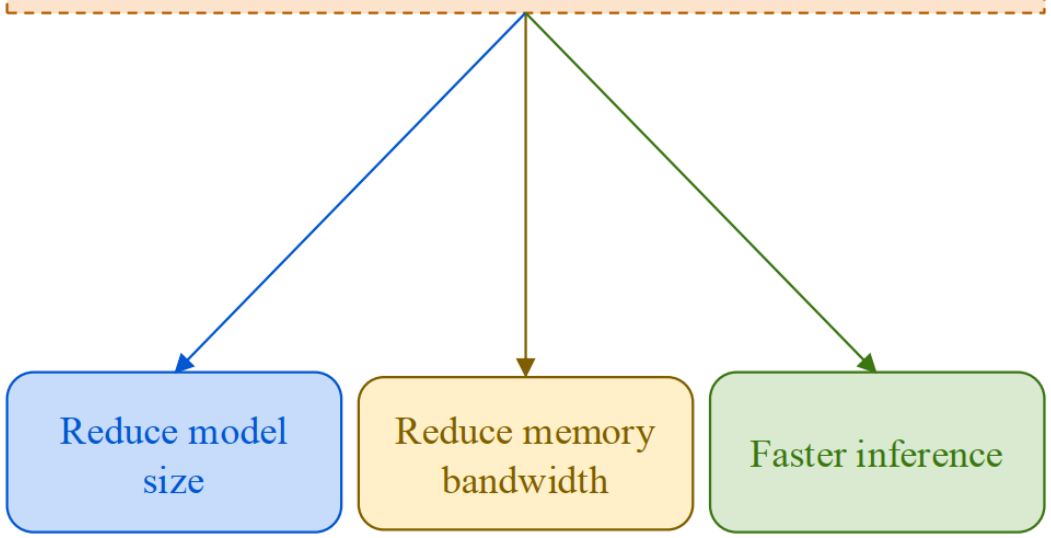
❖ Data types

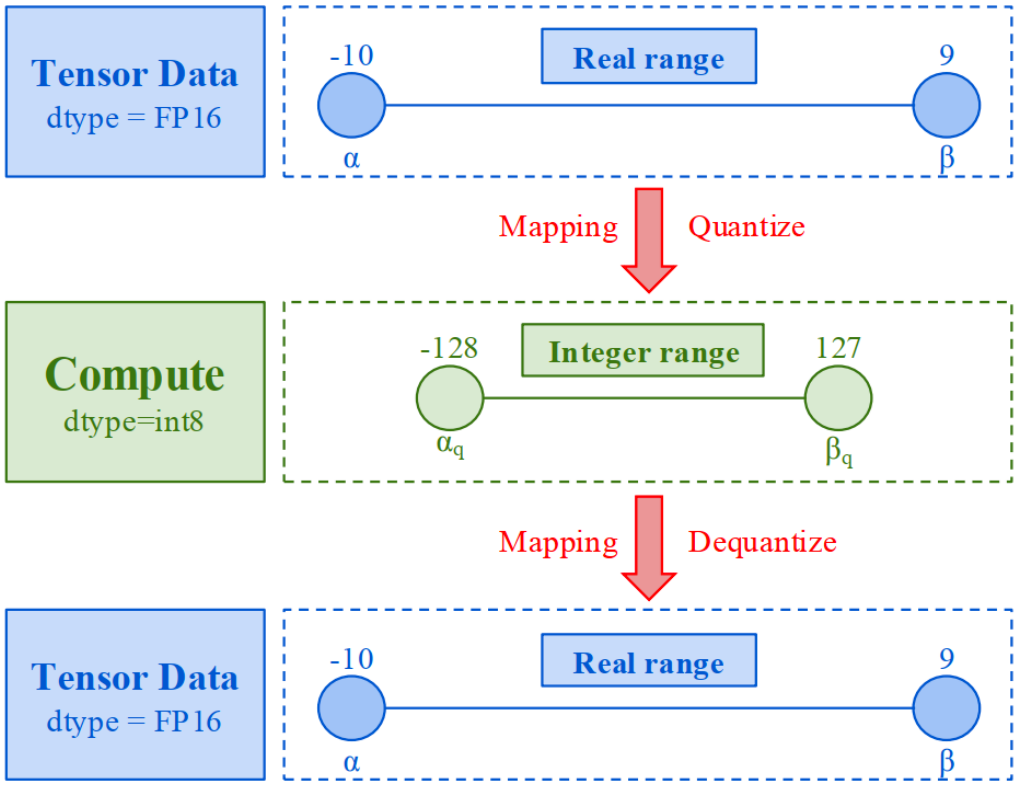


Introduction

❖ Quantization

Quantization refers to techniques for doing both computations and memory accesses with **lower precision data**, usually int8 compared to floating point implementations.



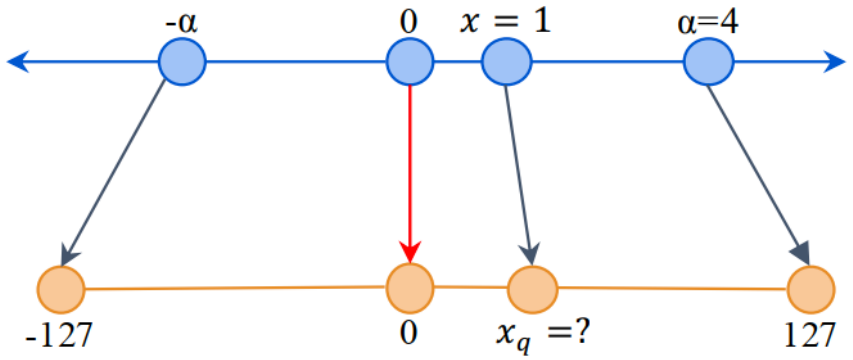


Input Data type	Accumulation Data type	Math Throughput	Bandwidth Reduction
FP32	FP32	1x	1x
FP16	FP16	8x	2x
INT8	INT32	16x	4x
INT4	INT32	32x	8x
INT1	INT32	128x	32x

Table 1: Benefits of lower precision data types for tensor operations on the NVIDIA Turing GPU architecture

Introduction

❖ Int8 quantization



Scale (Symmetric) Quantization

Quantize $x_q = clip(round(s \cdot x))$

Dequantize $\hat{x} = \frac{1}{s} x_q$

Scale $s = \frac{2^{b-1} - 1}{\alpha}$

$$s = \frac{2^{8-1} - 1}{4} = 31.75$$

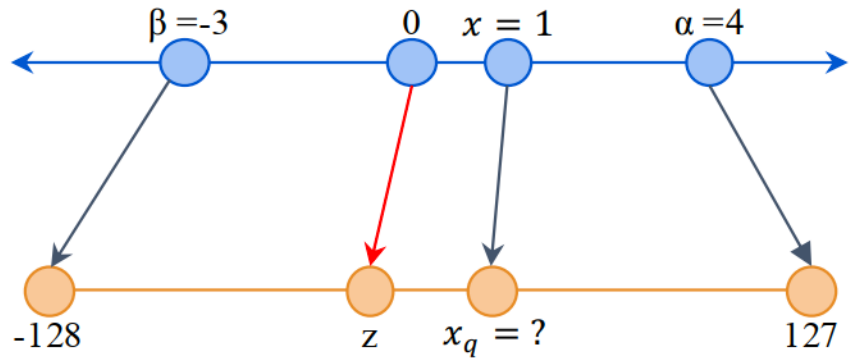
$$x_q = clip(round(31.75 \times 1)) = 32$$

$$\hat{x} = \frac{1}{31.75} 32 = 1.007874 \ldots$$

Ref paper: https://arxiv.org/pdf/2004.09602

Introduction

❖ Int8 quantization



Affine (Asymmetric) Quantization

Quantize

$$x_q = clip(round(s \cdot x + z))$$

Dequantize

$$\hat{x} = \frac{1}{s}(x_q - z)$$

Scale

$$s = \frac{2^b - 1}{\alpha - \beta}$$

$$z = -round(\beta \cdot s) - 2^{b-1}$$

$$s = \frac{2^8 - 1}{4 - (-3)} = 36.42$$

$$z = -round(-3 \times 36.42) - 2^{8-1} = -19$$

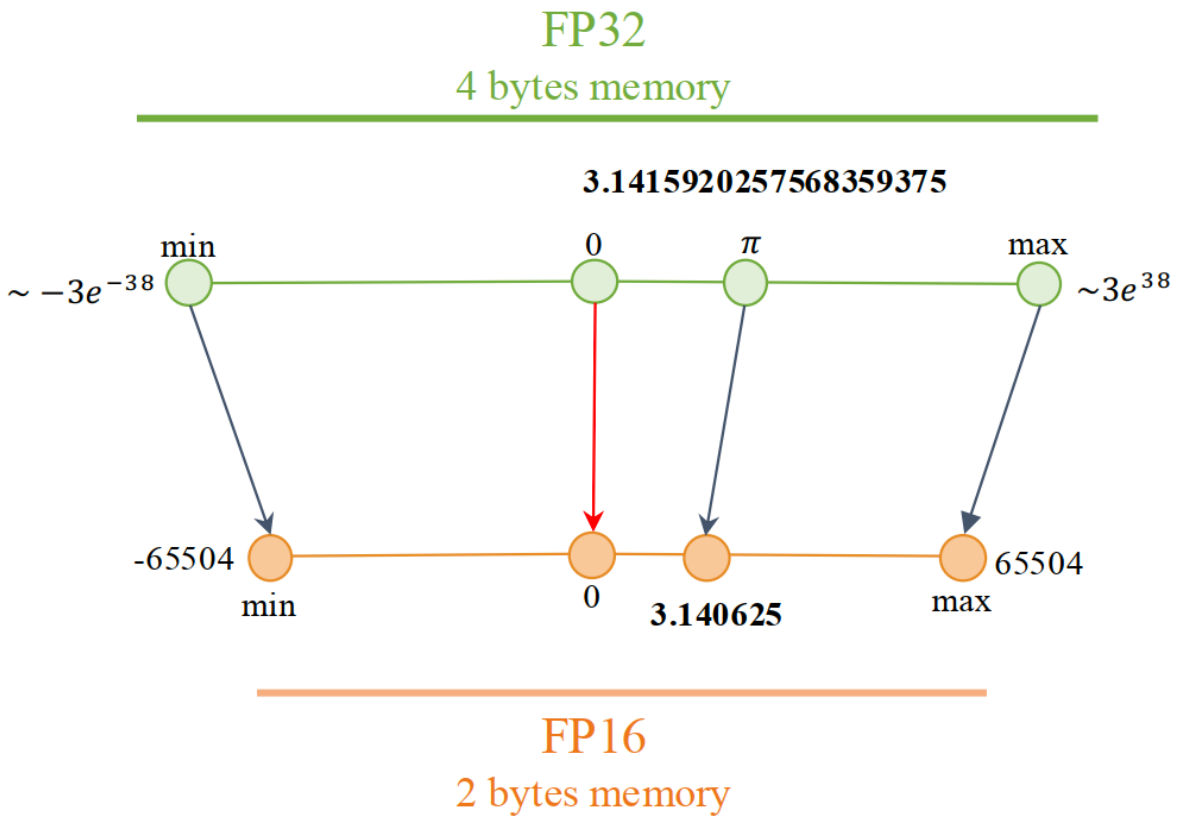
$$x_q = clip(round(36.42 \times 1 - 19)) = 17$$

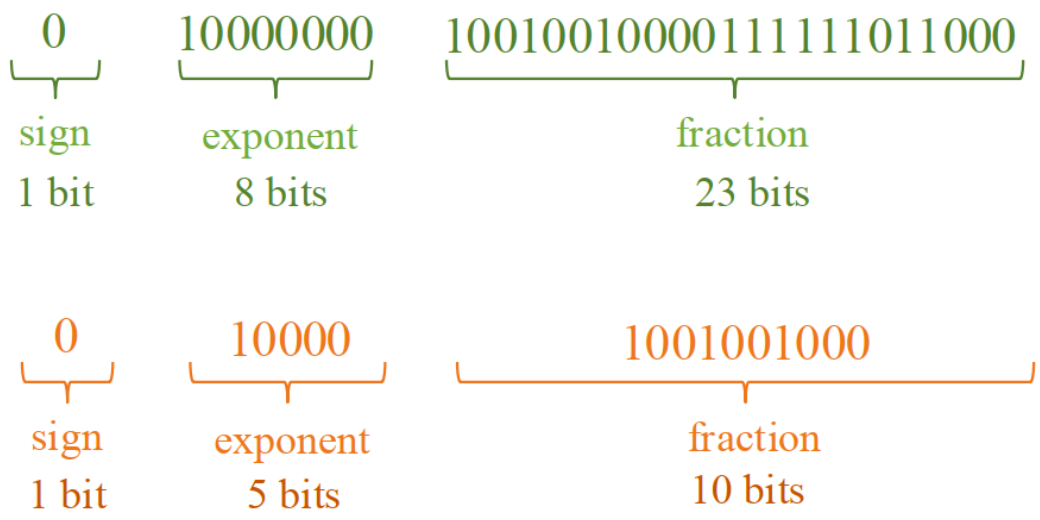
$$\hat{x} = \frac{1}{36.42}(17 + 19)) = 0.9882 \ldots$$

Ref paper: https://arxiv.org/pdf/2004.09602

Introduction

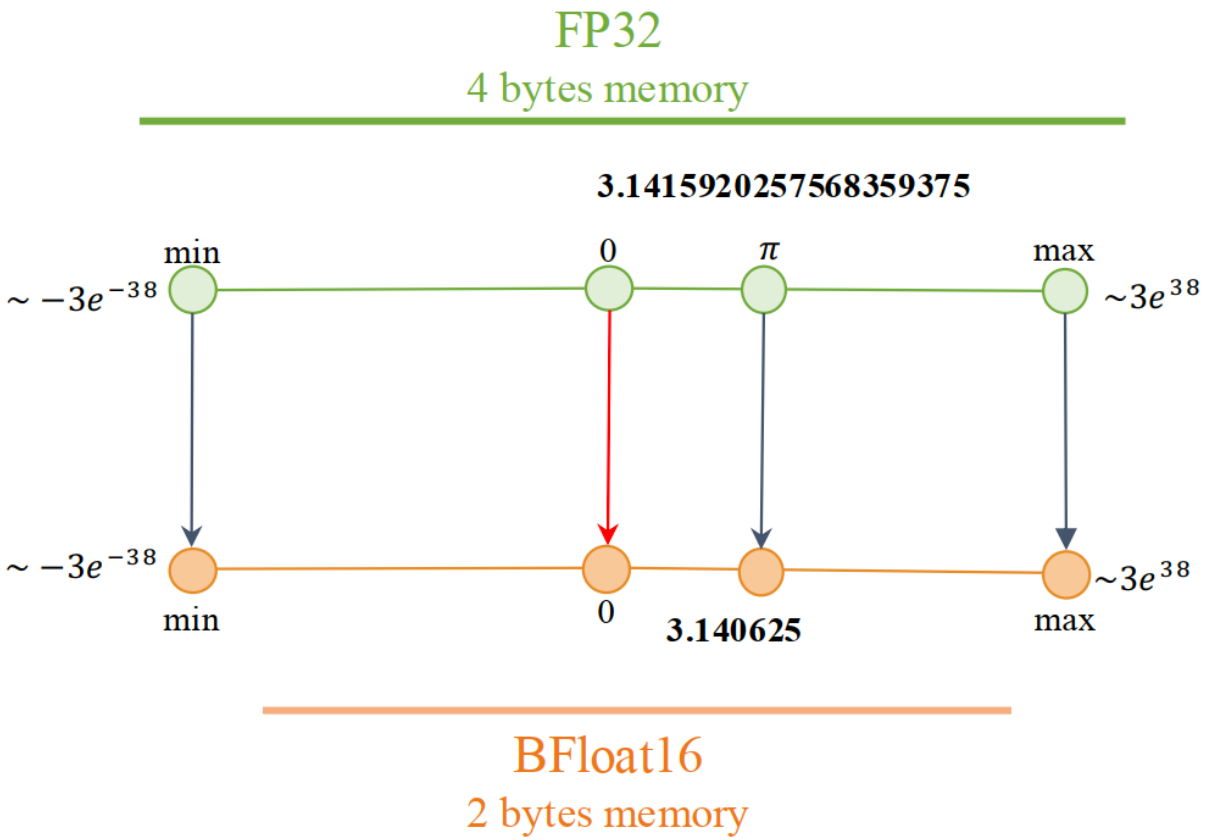
❖ FP16 quantization

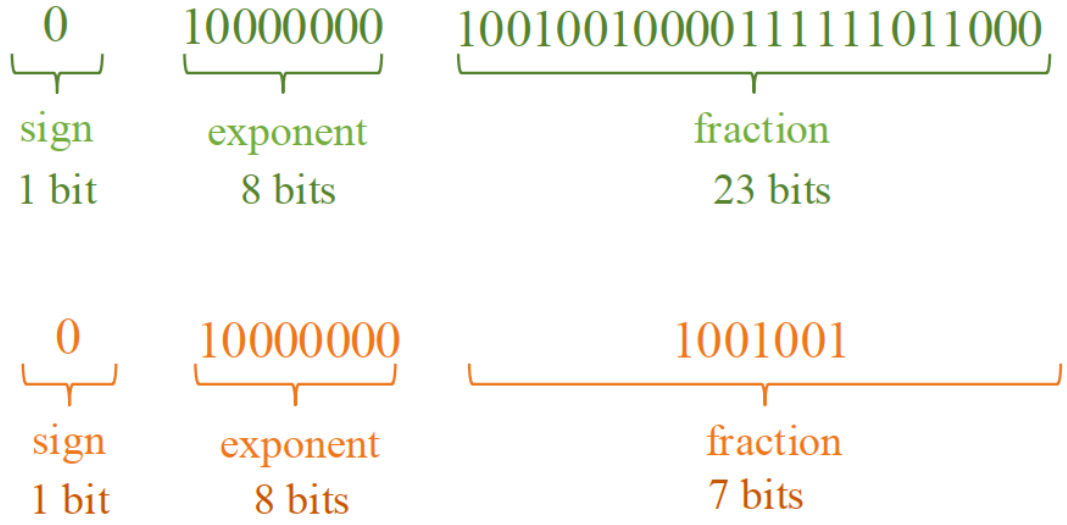




Introduction

❖ BF16 quantization





Introduction

❖ GPU RAM (VRAM) need to store 1B parameters

8 bits = 1 byte

1 parameter (float32) = 32 bits = 4 bytes → 1B parameters = 4×10^9 bytes = 4GB

1 parameter (loat16) = 16 bits = 2 bytes → 1B parameters = 2×10^9 bytes = 2GB

Memory needed to stored model

4GB @ 32 bits
full precision

2GB @ 16 bits
half precision

Introduction

❖ GPU RAM (VRAM) need to train 1B parameters

Memory needed to stored model

2GB @ 16 bits
half precision

```
1  model.to("cuda")
```

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 2048)
    (layers): ModuleList(
      (0-15): 16 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=512, bias=False)
          (v_proj): Linear(in_features=2048, out_features=512, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (down_proj): Linear(in_features=8192, out_features=2048, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (lm_head): Linear(in_features=2048, out_features=128256, bias=False)
)
```

Resources

You are not subscribed. Learn more

You currently have zero compute units available. Resources offered
Purchase more units here.

At your current usage level, this runtime may last up to 3 hours 30 m

Manage sessions

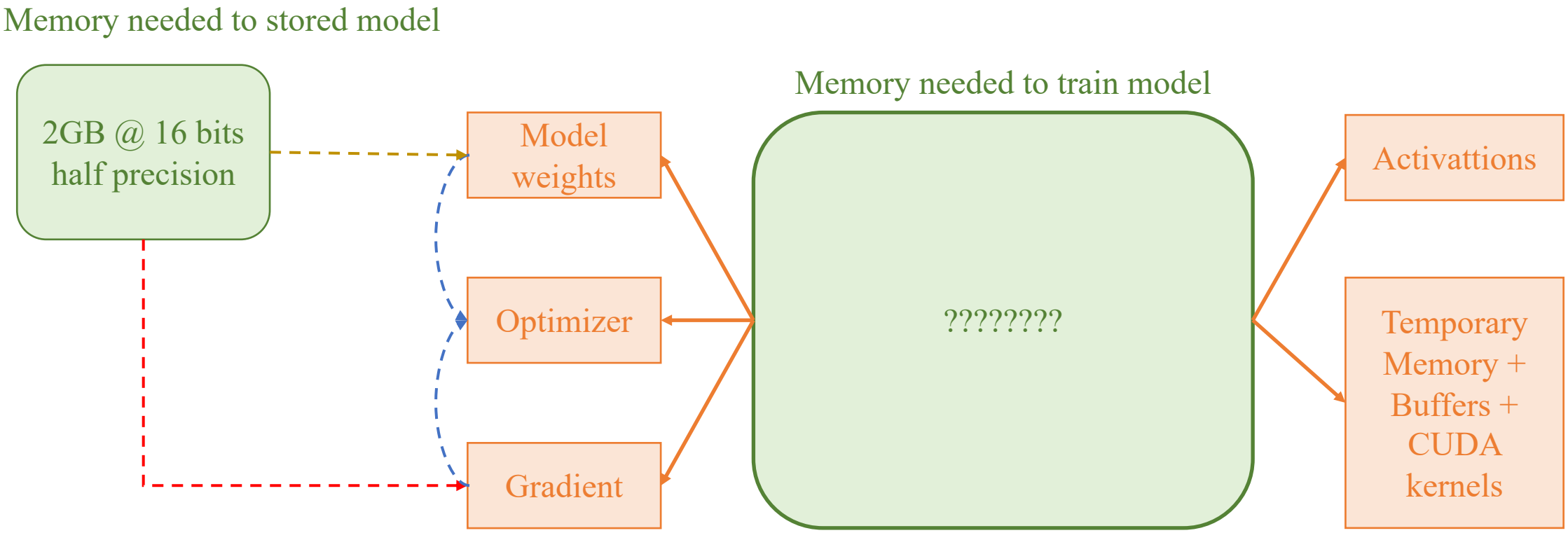
Python 3 Google Compute Engine backend (GPU)

Showing resources from 4:06 AM to 4:10 AM

System RAM	GPU RAM	Disk
2.5 / 12.7 GB	4.7 / 15.0 GB	39.4 / 112.6 GB

Introduction

❖ GPU RAM (VRAM) need to train 1B parameters



Introduction

❖ GPU RAM (VRAM) need to train 1B parameters

Load model

```
1s
1  model.to("cuda")
2  print_mem("Load model to GPU")

[Load model to GPU] Allocated: 4.60 GB | Max allocated: 4.60 GB
```

Load inputs

```
0s
1  # Generate random input_ids and labels
2  input_ids = torch.randint(0, vocab_size, (batch_size, seq_len)).cuda()
3  labels = input_ids.clone()
4
5  print_mem("inputs to GPU")

[inputs to GPU] Allocated: 4.60 GB | Max allocated: 4.60 GB
```

Batch_size=4
Seq_len=512

Forward

```
1s
1  # Forward pass
2  outputs = model(input_ids=input_ids, labels=labels)
3  loss = outputs.loss
4
5  print_mem("After forward")

[After forward] Allocated: 13.25 GB | Max allocated: 13.25 GB
```

Introduction

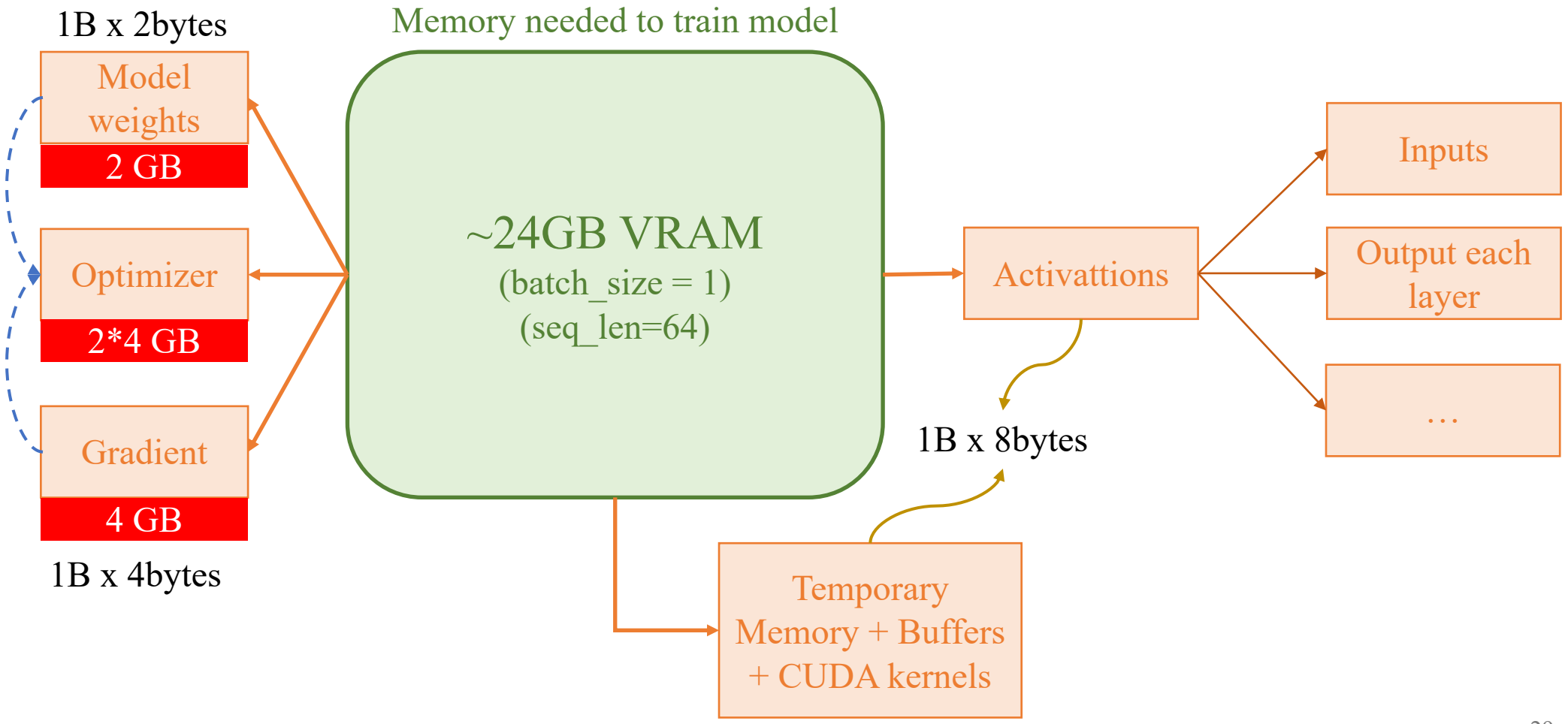
❖ GPU RAM (VRAM) need to train 1B parameters

```
[9]  1  # Backward pass
     2  loss.backward()
     3
     4  print_mem("After backward")
```

```
---------------------------------------------------------------------------
OutOfMemoryError                          Traceback (most recent call last)
<ipython-input-9-ef23ad5b52c2> in <cell line: 0>()
      1 # Backward pass
----> 2 loss.backward()
      3
      4 print_mem("After backward")

                                 2 frames
/usr/local/lib/python3.11/dist-packages/torch/autograd/graph.py in _engine_run_backward(t_outputs, *args, **kwargs)
    821         unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    822     try:
--> 823         return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
    824             t_outputs, *args, **kwargs
    825         )  # Calls into the C++ engine to run the backward pass

OutOfMemoryError: CUDA out of memory. Tried to allocate 1002.00 MiB. GPU 0 has a total capacity of 14.74 GiB of which 3
GiB memory in use. Of the allocated memory 14.22 GiB is allocated by PyTorch, and 17.08 MiB is reserved by PyTorch but
memory is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True to avoid fragmentation.  See documentatior
(https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
```

Introduction

❖ GPU RAM (VRAM) need to train 1B parameters



Introduction

❖ System required

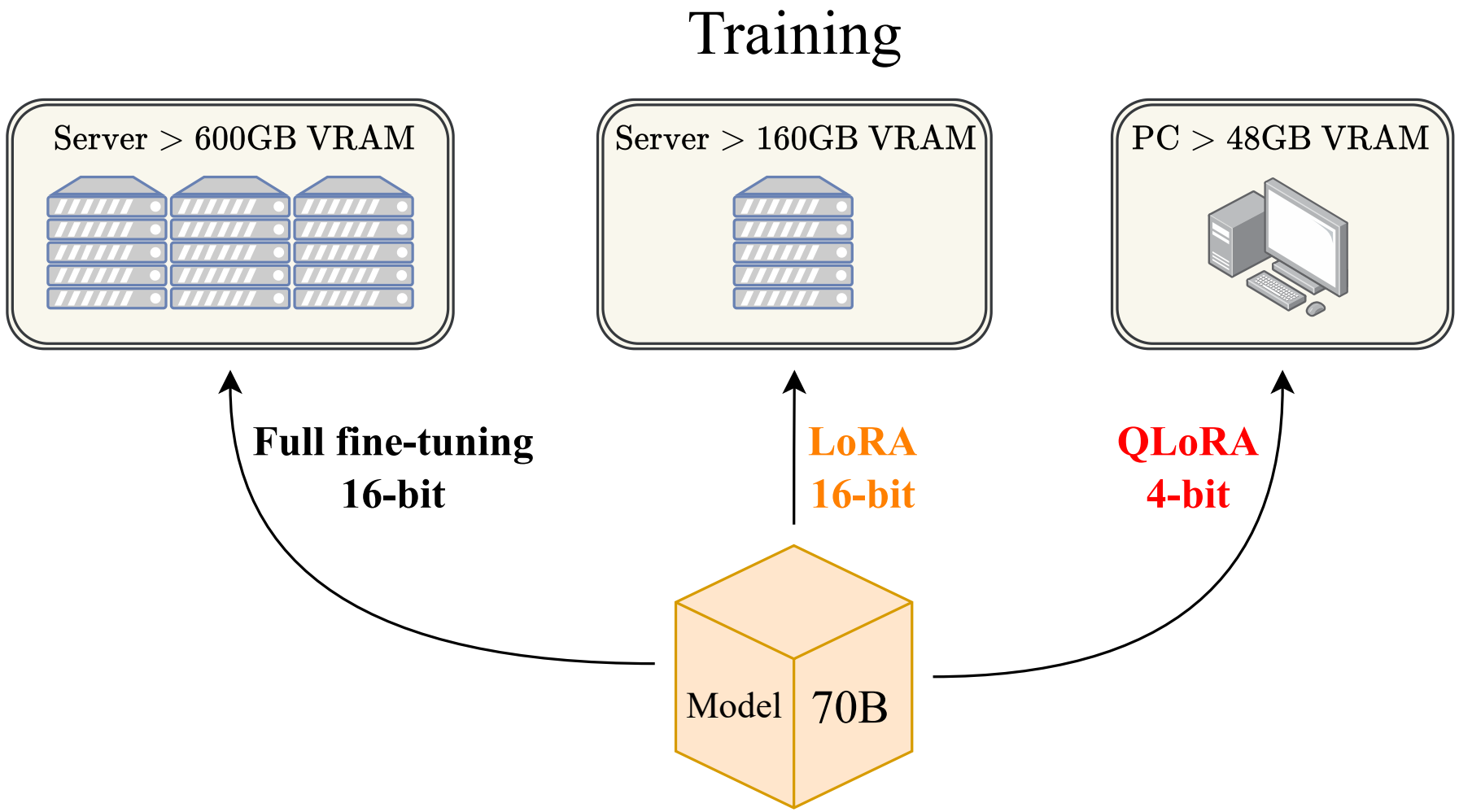
State	Byte per parameter
Model Weight	4 bytes (32 bit – full precision)
Optimizer (Adam – 2 states)	8 bytes
Gradients	4 bytes
Activations and temp memory	8 bytes
TOTAL	4 + (8+4+8) = 20 bytes

- $LLaMA-3.2-1B: 1.23B \rightarrow \sim 24GB\ VRAM$
- $LLaMA-4-Scout: 109B \rightarrow \sim 220GB\ VRAM$

LLaMa-Factory: https://github.com/hiyouga/LLaMA-Factory#%23hardware-requirement

Introduction

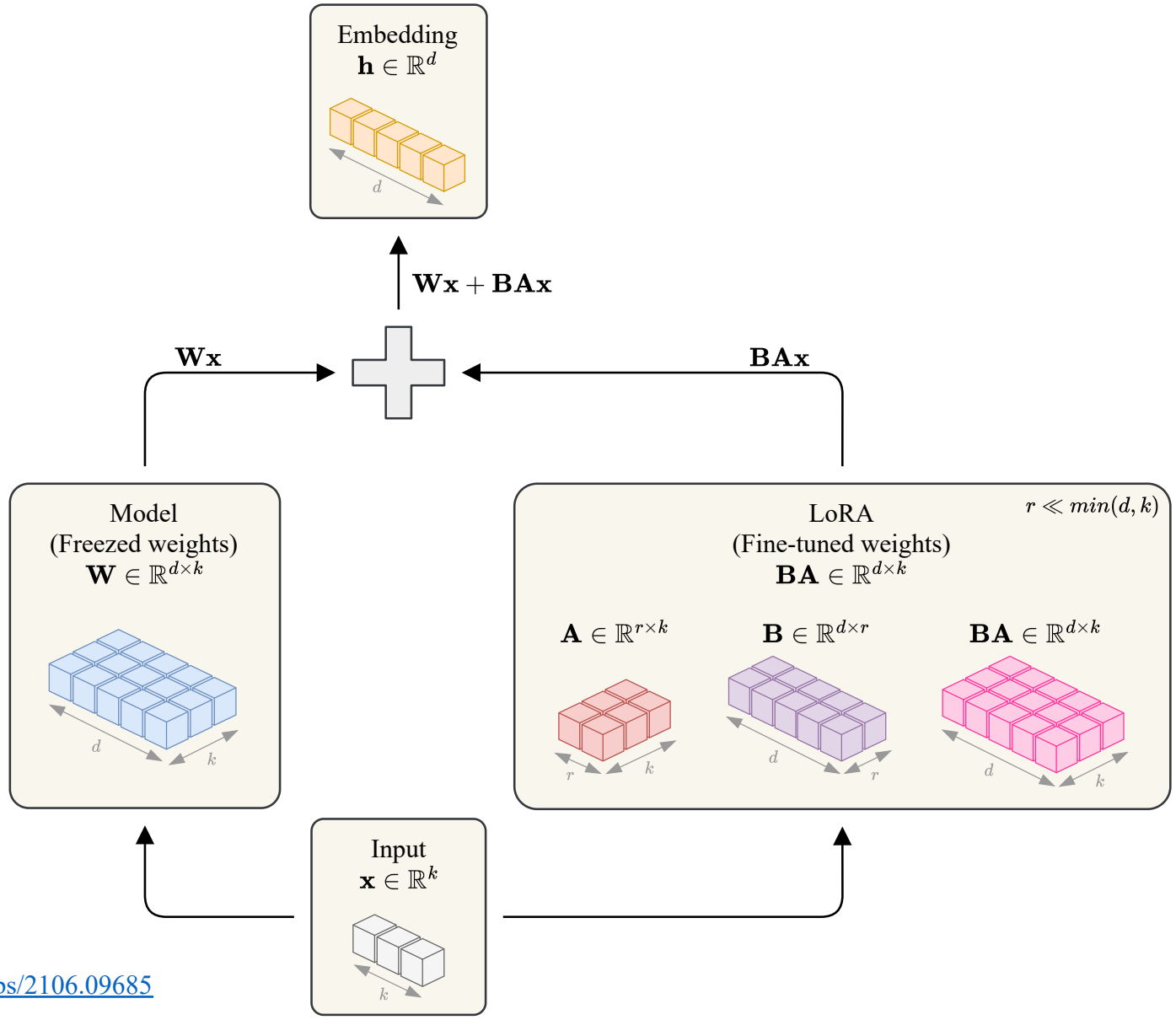
❖ Parameter-Efficient Fine-Tuning (PEFT)



LoRA

LoRA

❖ Getting Started



 Paper link: https://arxiv.org/abs/2106.09685

LoRA

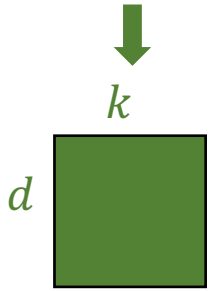
❖ Weight computing

Normal training

$$W = W_0 + \Delta W$$

$$W_0 \in \mathbb{R}^{d \times k}$$

$$\Delta W \in \mathbb{R}^{d \times k}$$



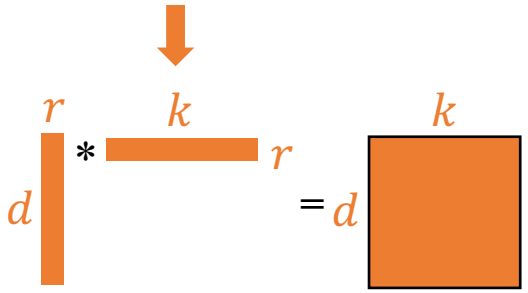
LoRA training

$$h = Wx = W_0x + \Delta Wx$$

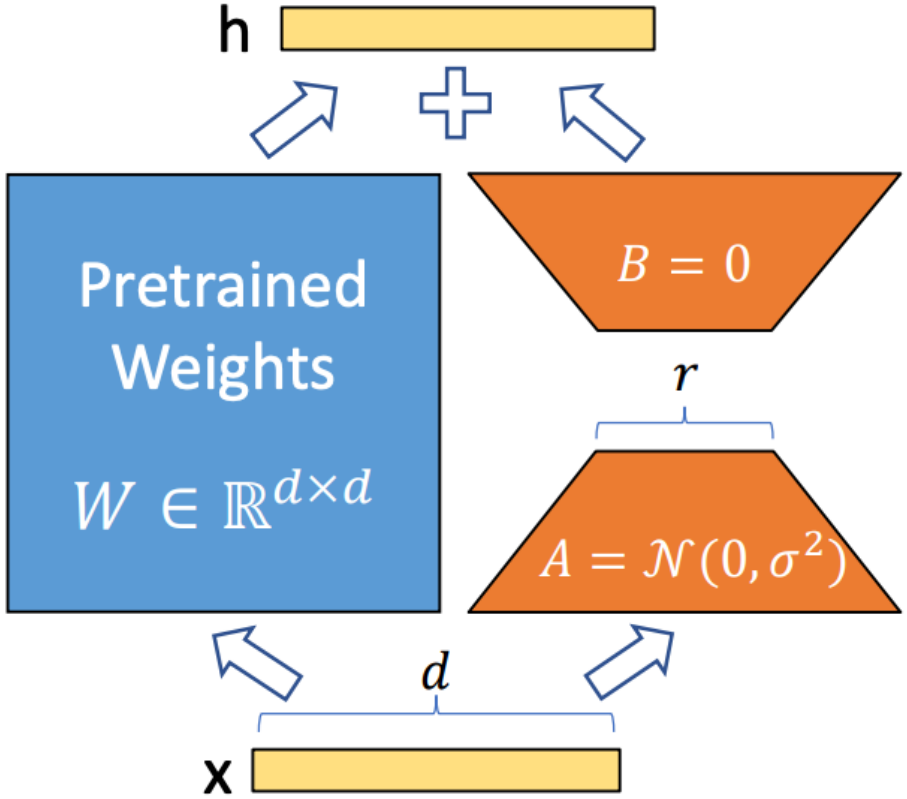
$$h = W_0x + BAx$$

$$B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$$

$$r \ll \{d, k\}$$



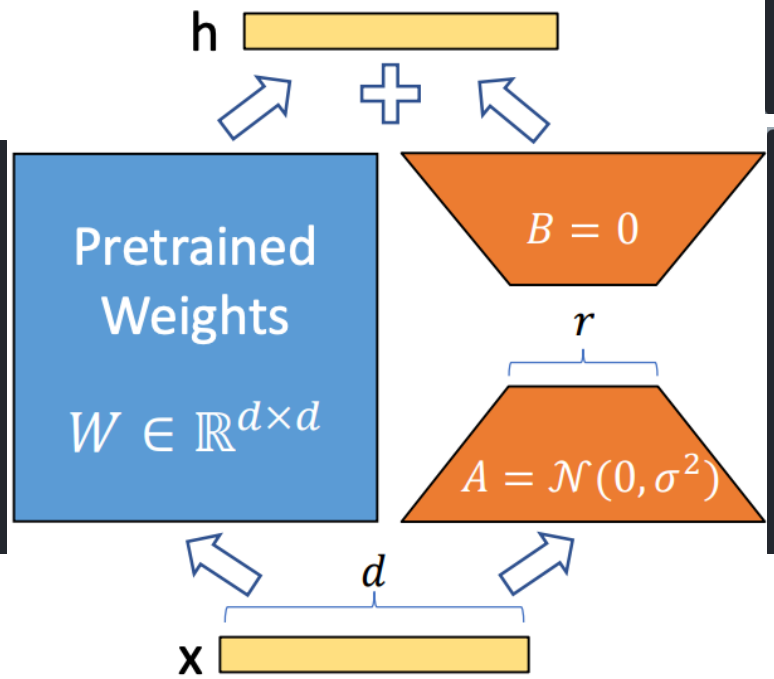
$d \times k$ ← Trainable parameters → $r \times (d + k)$



LoRA

❖ Linear weight

$$h = W_0 x + \frac{\alpha}{r} BAx$$




```python
import torch

hidden_size = 1024
fc = torch.nn.Linear(hidden_size, hidden_size*4)

print(fc.weight.shape) # torch.Size([4096, 1024])

# Number of parameters
print(fc.weight.numel()) # 4096 * 1024 = 4194304
```

```python
lora_alpha = 32
scale = lora_alpha / r

lora_out = scale * lora_b.weight @ lora_a.weight

print(lora_out.shape) # torch.Size([4096, 1024])
```

```python
r = 16
lora_a = torch.nn.Linear(hidden_size, r, bias=False)
lora_b = torch.nn.Linear(r, hidden_size, bias=False)

print(lora_a.weight.shape) # torch.Size([16, 1024])
print(lora_b.weight.shape) # torch.Size([4096, 16])

print(lora_a.weight.numel() + lora_b.weight.numel())
# >> 16 * 1024 + 4096 * 16 = 81920
```

LoRA

❖ LoRA from Scratch: Linear

```python
class LoRA_Layer():
    def __init__(
        self,
        r: int,
        lora_alpha: int,
        lora_dropout: float,
        merge_weights: bool,
    ):
        self.r = r
        self.lora_alpha = lora_alpha
        if lora_dropout > 0.:
            self.lora_dropout = nn.Dropout(p=lora_dropout)
        else:
            self.lora_dropout = lambda x: x
        self.merged = False
        self.merge_weights = merge_weights

```

```python
class LoRA_Linear(nn.Linear, LoRA_Layer):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.,
        merge_weights: bool = True,
        **kwargs
    ):
        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoRA_Layer.__init__(self, r=r, lora_alpha=lora_alpha, lora_dropout=lora_dropout, merge_weights=merge_weights)

        if r > 0:
            self.lora_A = nn.Parameter(self.weight.new_zeros((r, in_features)))
            self.lora_B = nn.Parameter(self.weight.new_zeros((out_features, r)))
            self.scaling = self.lora_alpha / self.r
            self.weight.requires_grad = False
```

LoRA

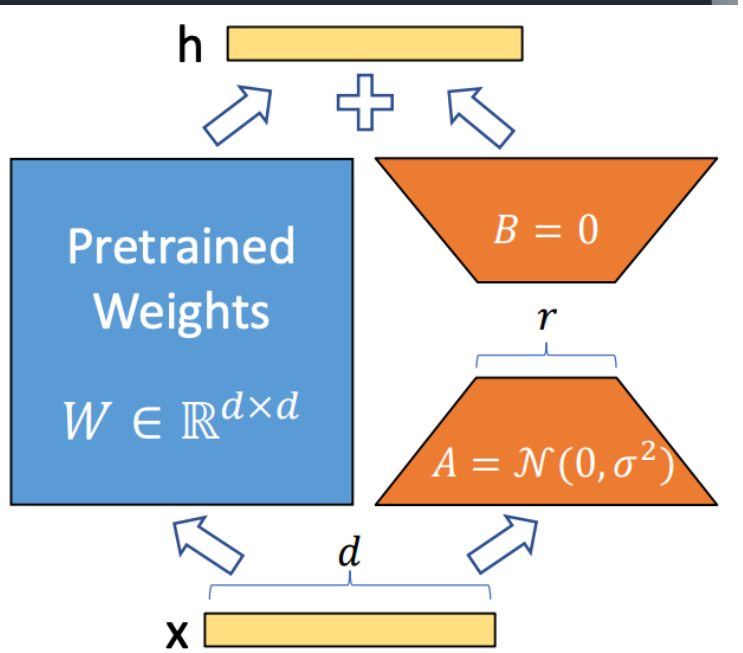
❖ LoRA from Scratch: Linear

```python
class LoRA_Linear(nn.Linear, LoRA_Layer):

    def train(self, mode: bool = True):
        nn.Linear.train(self, mode)

        # Training mode
        if mode:
            if self.merge_weights and self.merged:
                if self.r > 0:
                    self.weight.data -= (self.lora_B @ self.lora_A) * self.scaling
                self.merged = False

        # Evaluation mode
        else:
            if self.merge_weights and not self.merged:
                if self.r > 0:
                    self.weight.data += (self.lora_B @ self.lora_A) * self.scaling
                self.merged = True
```

LoRA

❖ LoRA from Scratch: Linear

```python
class LoRA_Linear(nn.Linear, LoRA_Layer):

    def forward(self, x: torch.Tensor):
        # Evaluation mode
        if self.r > 0 and not self.merged:
            result = F.linear(x, self.weight, bias=self.bias)
            result += (self.lora_dropout(x) @ self.lora_A.transpose(0, 1) @ self.lora_B.transpose(0, 1)) * self.scaling
            return result

        # Training mode
        else:
            return F.linear(x, self.weight, bias=self.bias)
```



LoRA

❖ Pratices: FT VGG16

Without LoRA

```
Epoch [1/10], Average Loss: 1.3391, GPU used: 2.06 G
Epoch [2/10], Average Loss: 1.1493, GPU used: 2.06 G
Epoch [3/10], Average Loss: 1.1043, GPU used: 2.06 G
Epoch [4/10], Average Loss: 1.0918, GPU used: 2.06 G
Epoch [5/10], Average Loss: 1.0741, GPU used: 2.06 G
Epoch [6/10], Average Loss: 1.0578, GPU used: 2.06 G
Epoch [7/10], Average Loss: 1.0518, GPU used: 2.06 G
Epoch [8/10], Average Loss: 1.0337, GPU used: 2.06 G
Epoch [9/10], Average Loss: 1.0275, GPU used: 2.06 G
Epoch [10/10], Average Loss: 1.0211, GPU used: 2.06 G
Finished Training
Training time: 205.44 s
```

5. Evaluate

```
correct = 0
total = 0
model.eval()
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the model on the 10000 test images: {100 * correct / total} %')
```

```
Accuracy of the model on the 10000 test images: 65.82 %
```

LoRA

❖ Pratices: FT VGG16

With LoRA

```
Epoch [1/10], Average Loss: 1.5478, GPU used: 0.59 G
Epoch [2/10], Average Loss: 1.2768, GPU used: 0.59 G
Epoch [3/10], Average Loss: 1.2090, GPU used: 0.59 G
Epoch [4/10], Average Loss: 1.1460, GPU used: 0.59 G
Epoch [5/10], Average Loss: 1.1072, GPU used: 0.59 G
Epoch [6/10], Average Loss: 1.0763, GPU used: 0.59 G
Epoch [7/10], Average Loss: 1.0581, GPU used: 0.59 G
Epoch [8/10], Average Loss: 1.0360, GPU used: 0.59 G
Epoch [9/10], Average Loss: 1.0221, GPU used: 0.59 G
Epoch [10/10], Average Loss: 0.9981, GPU used: 0.59 G
Finished Training
Training time: 128.50 s
```

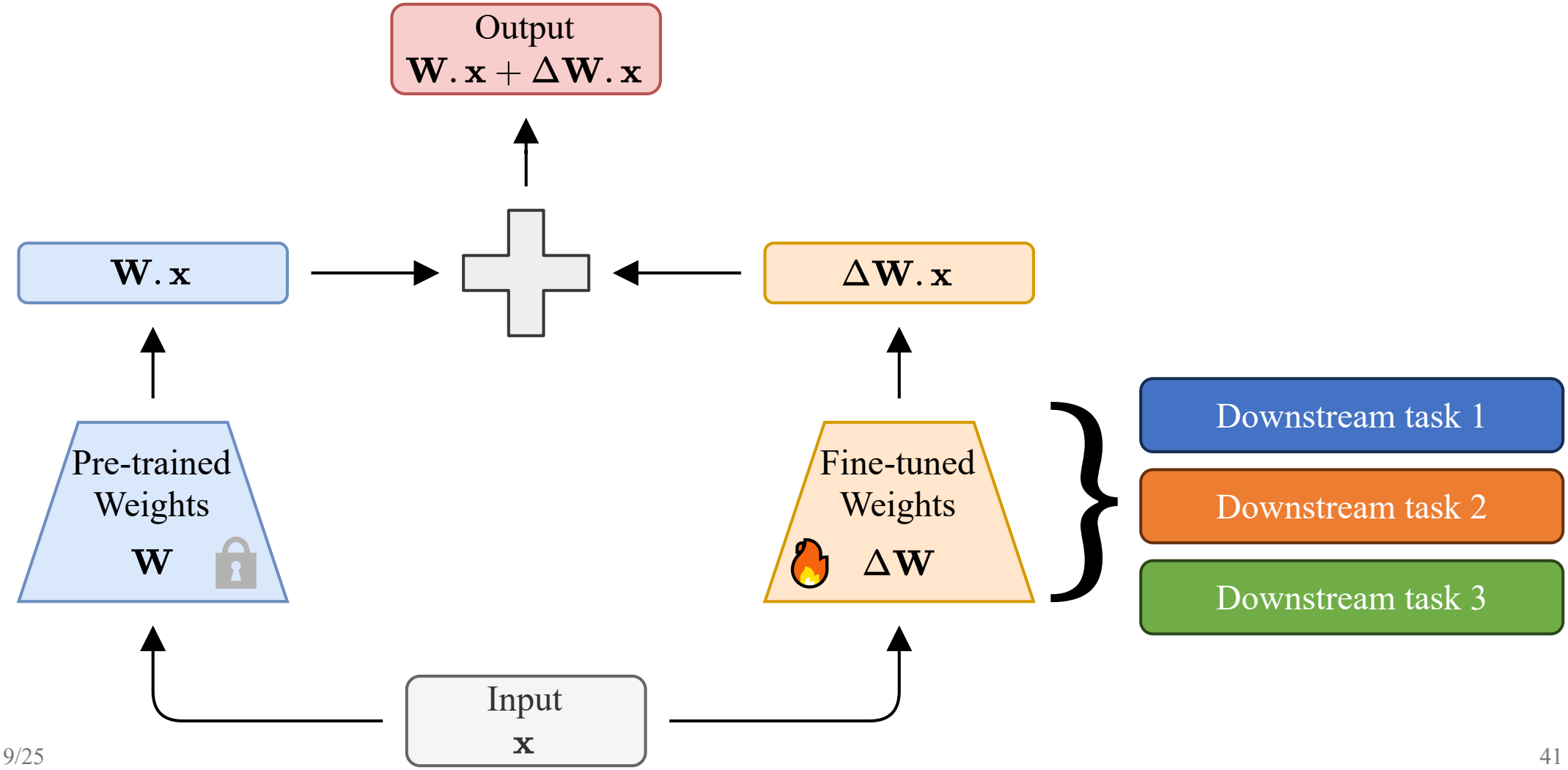
5. Evaluate

```
correct = 0
total = 0
model.eval()
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the model on the 10000 test images: {100 * correct / total} %')
```

```
Accuracy of the model on the 10000 test images: 65.03 %
```

LoRA

❖ Task Switching



```
model
```

```
ViTForImageClassification(
  (vit): ViTModel(
    (embeddings): ViTEmbeddings(
      (patch_embeddings): ViTPatchEmbeddings(
        (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): ViTEncoder(
      (layer): ModuleList(
        (0-11): 12 x ViTLayer(
          (attention): ViTAttention(
            (attention): ViTSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
            )
            (output): ViTSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
          (intermediate): ViTIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): ViTOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (dropout): Dropout(p=0.0, inplace=False)
          )
          (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (classifier): Linear(in_features=768, out_features=1000, bias=True)
)
```

LoRA

❖ LoRA with Huggingface

```
[9]  1  from peft import LoraConfig, get_peft_model
     2
     3  config = LoraConfig(
     4      r=16,
     5      lora_alpha=16,
     6      target_modules=["query", "value"],
     7      lora_dropout=0.1,
     8      bias="none",
     9  )
    10  lora_model = get_peft_model(model, config)
    11  print_trainable_parameters(lora_model)
```

```
trainable params: 589824 || all params: 87926480 || trainable%: 0.67
```

```
1  model
```

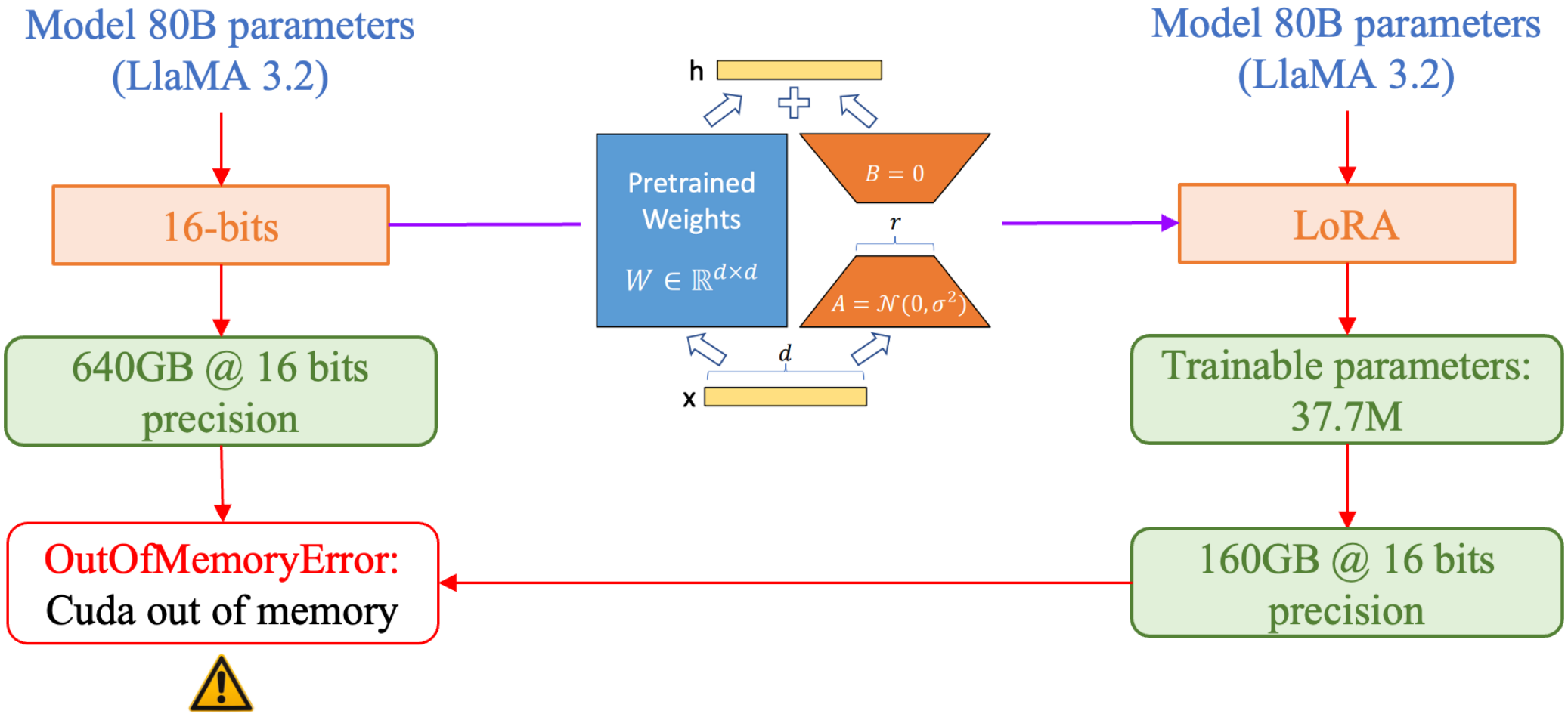
```
ViTForImageClassification(
  (vit): ViTModel(
    (embeddings): ViTEmbeddings(
      (patch_embeddings): ViTPatchEmbeddings(
        (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): ViTEncoder(
      (layer): ModuleList(
        (0-11): 12 x ViTLayer(
          (attention): ViTAttention(
            (attention): ViTSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
            )
            (output): ViTSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
          (intermediate): ViTIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): ViTOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (dropout): Dropout(p=0.0, inplace=False)
          )
          (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (classifier): Linear(in_features=768, out_features=1000, bias=True)
)
```

```
1  lora_model
```

```
PeftModel(
  (base_model): LoraModel(
    (model): ViTForImageClassification(
      (vit): ViTModel(
        (embeddings): ViTEmbeddings(
          (patch_embeddings): ViTPatchEmbeddings(
            (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
          )
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (encoder): ViTEncoder(
          (layer): ModuleList(
            (0-11): 12 x ViTLayer(
              (attention): ViTAttention(
                (attention): ViTSelfAttention(
                  (query): lora.Linear(
                    (base_layer): Linear(in_features=768, out_features=768, bias=True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.1, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=768, out_features=16, bias=False)
                    )
                    (lora_B): ModuleDict(
                      (default): Linear(in_features=16, out_features=768, bias=False)
                    )
                    (lora_embedding_A): ParameterDict()
                    (lora_embedding_B): ParameterDict()
                    (lora_magnitude_vector): ModuleDict()
                  )
                  (key): Linear(in_features=768, out_features=768, bias=True)
                  (value): lora.Linear(
                    (base_layer): Linear(in_features=768, out_features=768, bias=True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.1, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=768, out_features=16, bias=False)
                    )
                    (lora_B): ModuleDict(
                      (default): Linear(in_features=16, out_features=768, bias=False)
                    )
                    (lora_embedding_A): ParameterDict()
                    (lora_embedding_B): ParameterDict()
                    (lora_magnitude_vector): ModuleDict()
                  )
                )
              )
```

LoRA

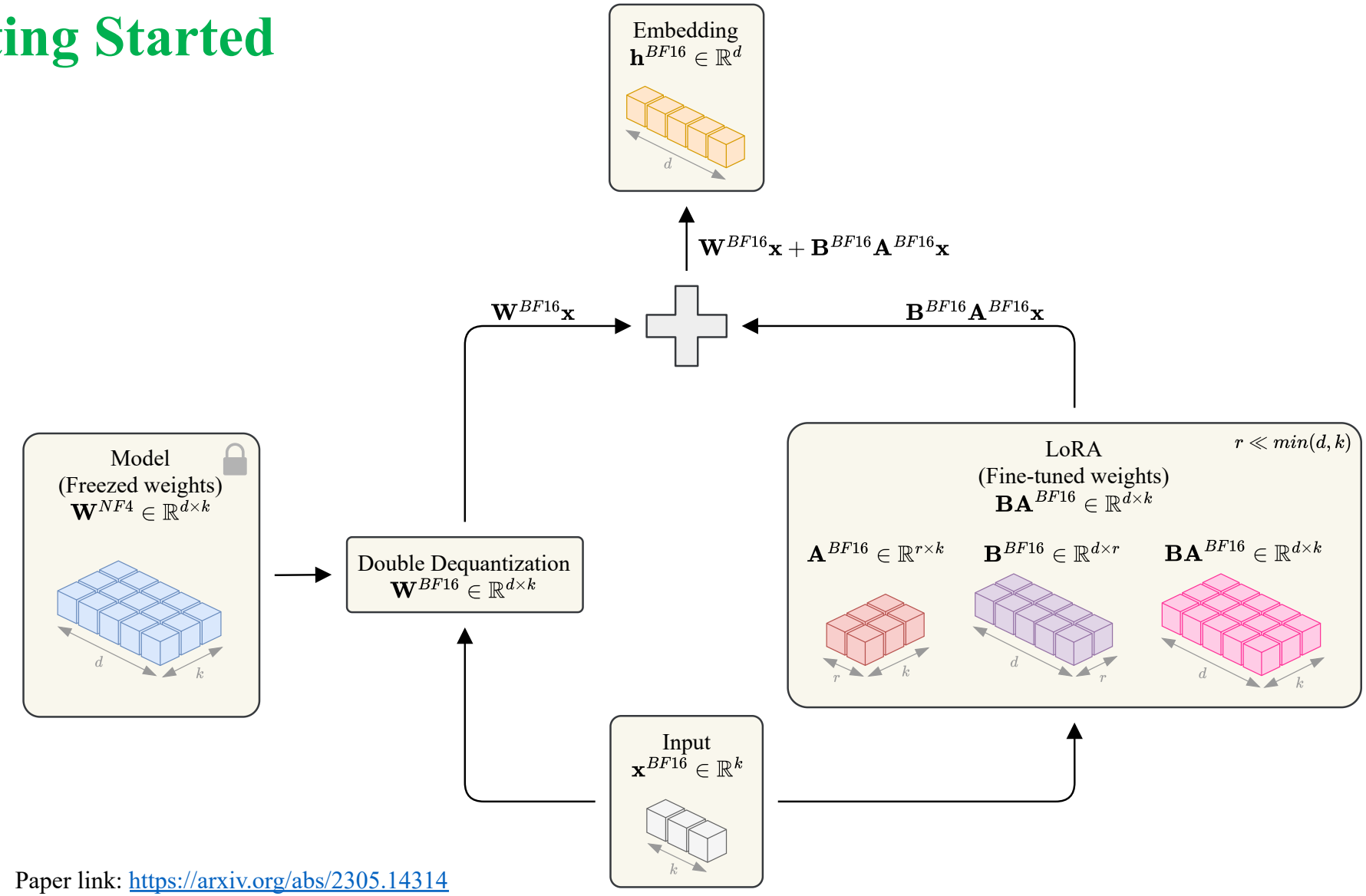
❖ LlaMA 3.2 with LoRA



QLoRA

QLoRA

❖ Getting Started

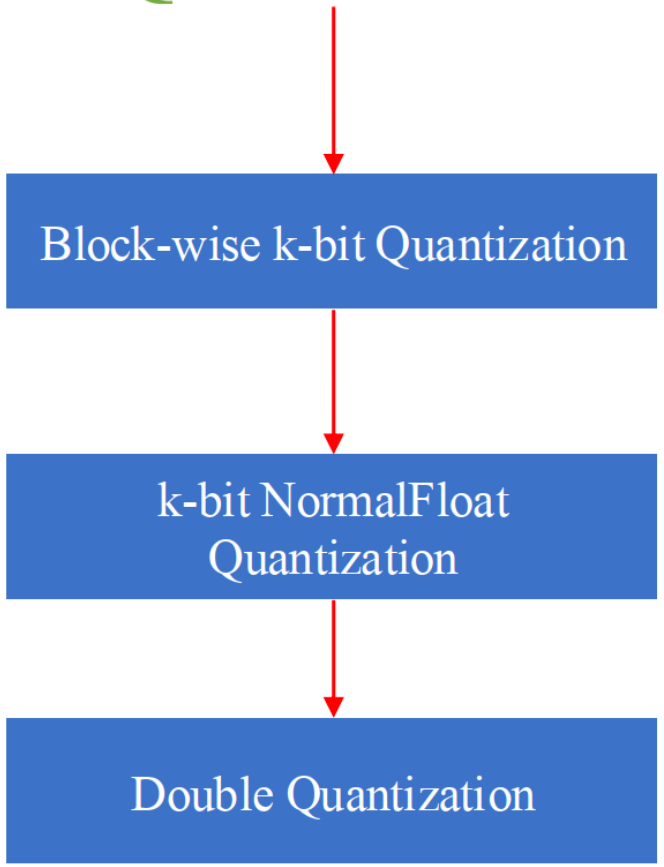


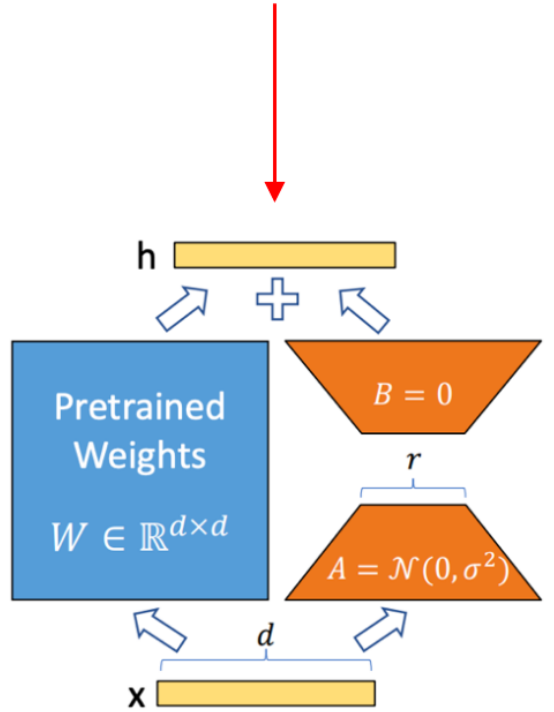
Paper link: https://arxiv.org/abs/2305.14314

QLoRA

❖ Getting Started

QLoRA = Quantization + Paged Optimizers + LoRA



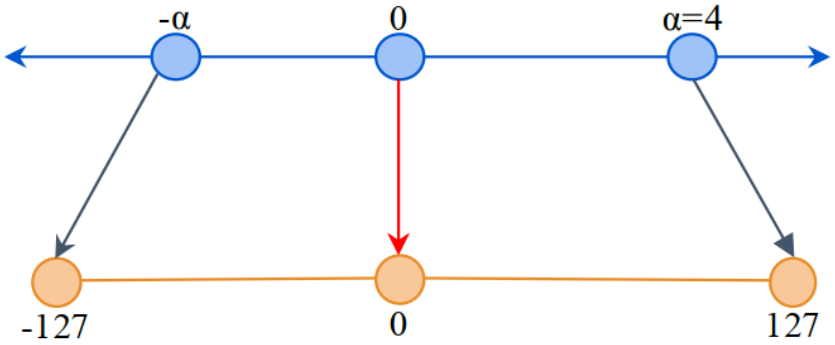


QLoRA

❖ Block-wise k-bit Quantization

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(\mathbf{X}^{\text{FP32}})}\mathbf{X}^{\text{FP32}}\right) = \text{round}(c^{\text{FP32}} \cdot \mathbf{X}^{\text{FP32}})$$

$$\text{dequant}(c^{\text{FP32}}, \mathbf{X}^{\text{Int8}}) = \frac{\mathbf{X}^{\text{Int8}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{FP32}}$$



Scale (Symmetric) Quantization

$$c = \frac{2^{8-1} - 1}{4} = 31.75$$

$$x_1 = 1.75,\ x_2 = 1.77$$

$$x_{q1} = clip\big(round(31.75 \times 1.75)\big) = 56$$
$$x_{q2} = clip\big(round(31.75 \times 1.77)\big) = 56$$

$$\hat{x}_{q1} = \hat{x}_{q2} = \frac{1}{31.75}56 = 1.7637\ldots$$

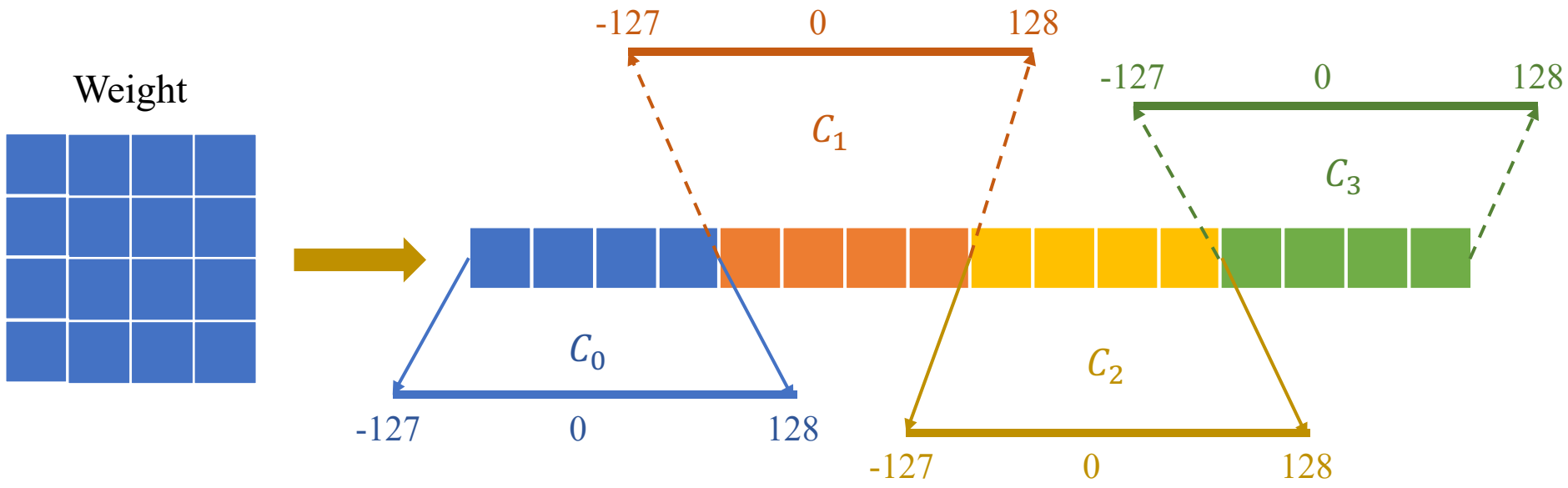
Loss value

QLoRA

❖ Block-wise k-bit Quantization

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(\mathbf{X}^{\text{FP32}})}\mathbf{X}^{\text{FP32}}\right) = \text{round}(c^{\text{FP32}} \cdot \mathbf{X}^{\text{FP32}})$$

$$\text{dequant}(c^{\text{FP32}}, \mathbf{X}^{\text{Int8}}) = \frac{\mathbf{X}^{\text{Int8}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{FP32}}$$

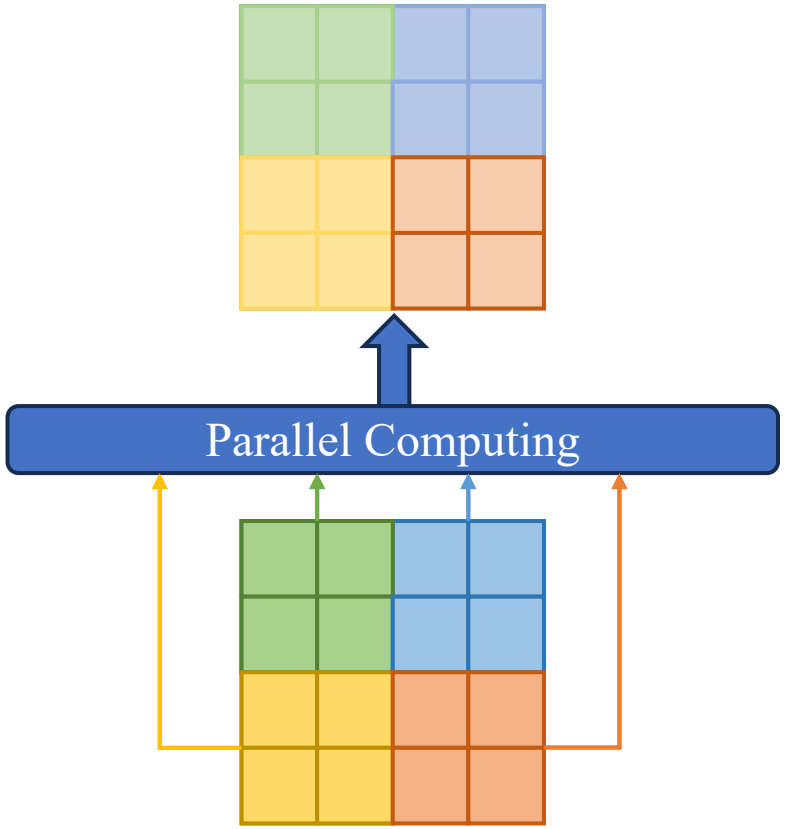


QLoRA

❖ Block-wise k-bit Quantization

1. Stability

- **Independent block normalization**: each block normalized independently, enabling parallel computation.
- No synchronization between cores needed, **enhancing throughput**.

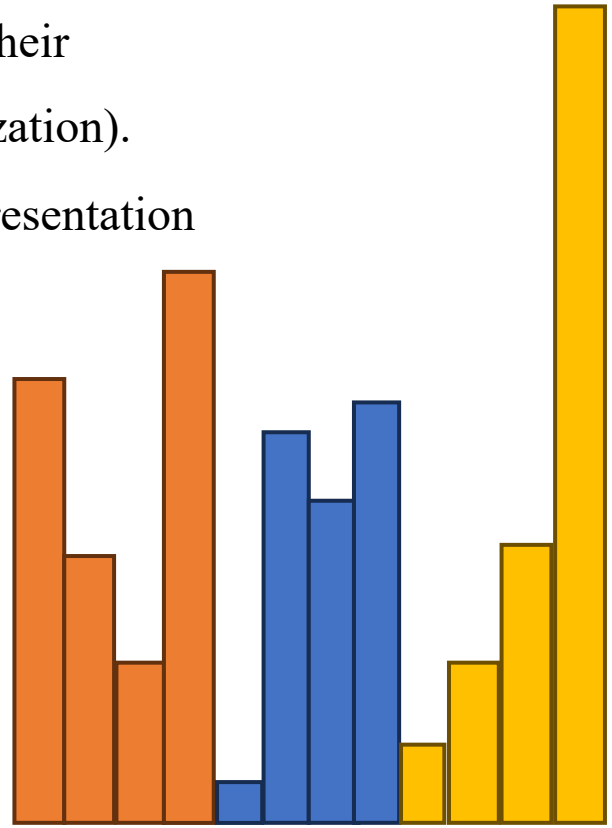


QLoRA

❖ Block-wise k-bit Quantization

2. Outlier Impact Mitigation

- Block-wise quantization **improves robustness to outliers** by limiting their impact to a single block (outliers affect whole tensors in regular quantization).
- **Outliers are preserved with full precision**, enabling high-fidelity representation of critical optimizer states and enhancing training performance.



QLoRA

❖ Block-wise k-bit Quantization

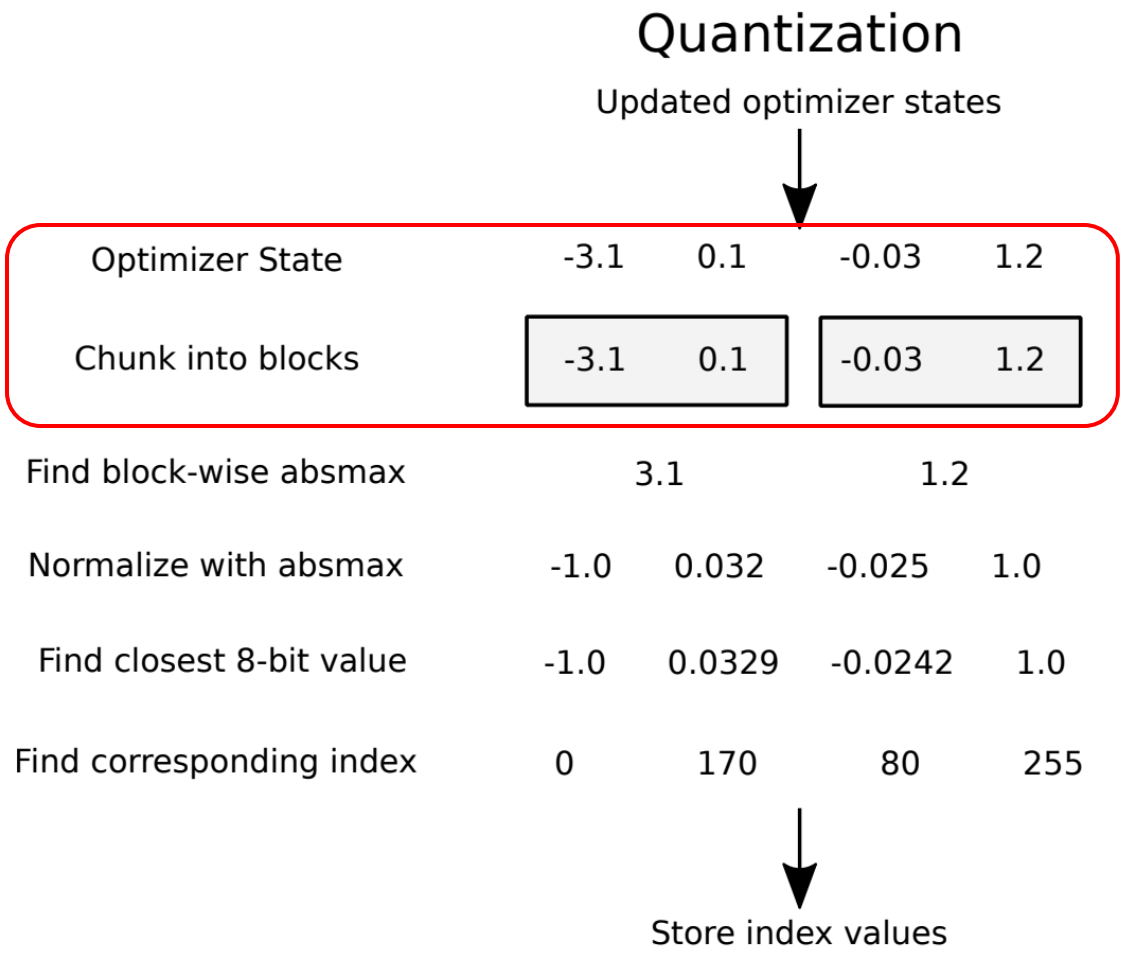
The quantization process contains 3 steps:

1. **Segmentation into Blocks**

- Flatten the weight matrix
- Divide into fixed-size blocks (e.g., 64 elements)

2. Normalization Within Each Block

3. Independent Quantization



QLoRA

❖ Block-wise k-bit Quantization

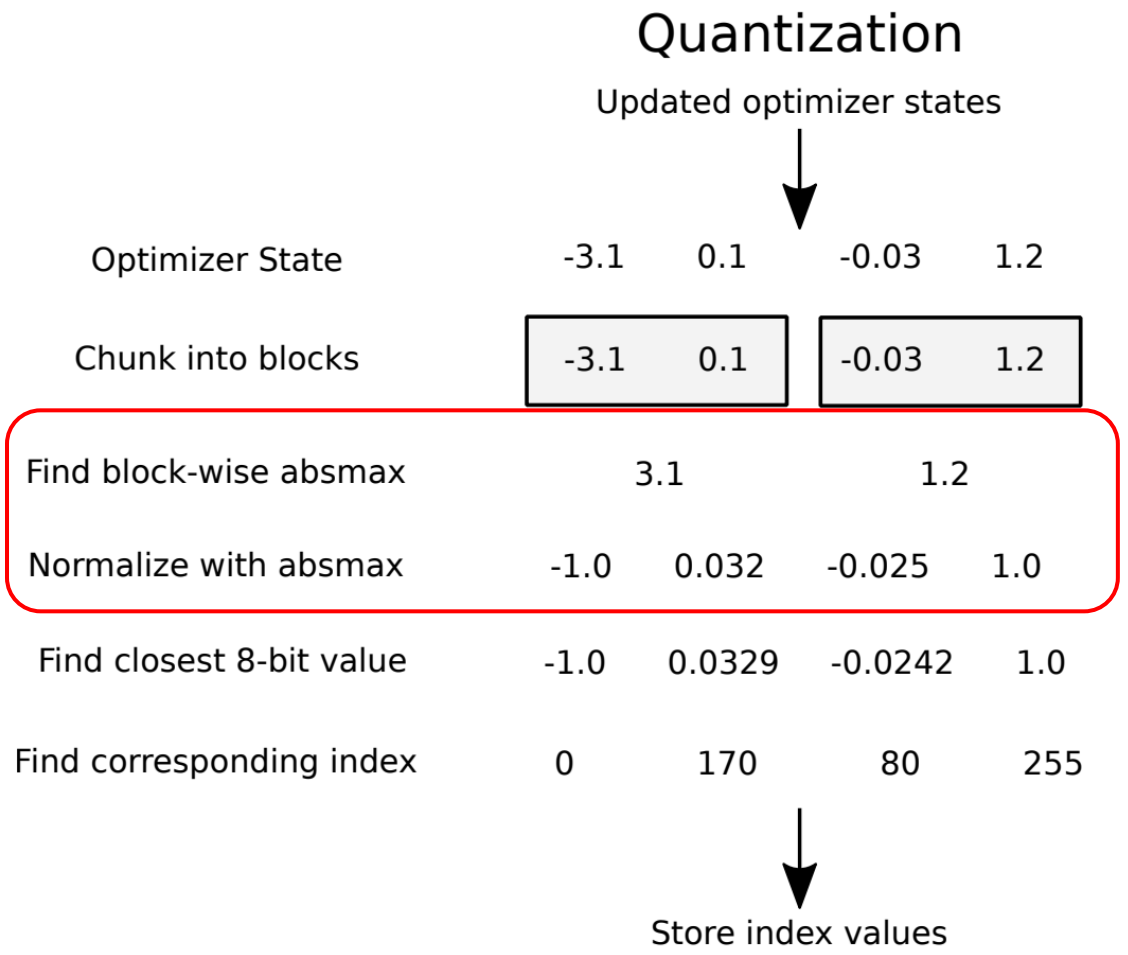
The quantization process contains 3 steps:

1. Segmentation into Blocks

2. **Normalization Within Each Block**

- Normalize each block by its max absolute value
- Scales weights to the range [-1, 1]

3. Independent Quantization



QLoRA

❖ Block-wise k-bit Quantization

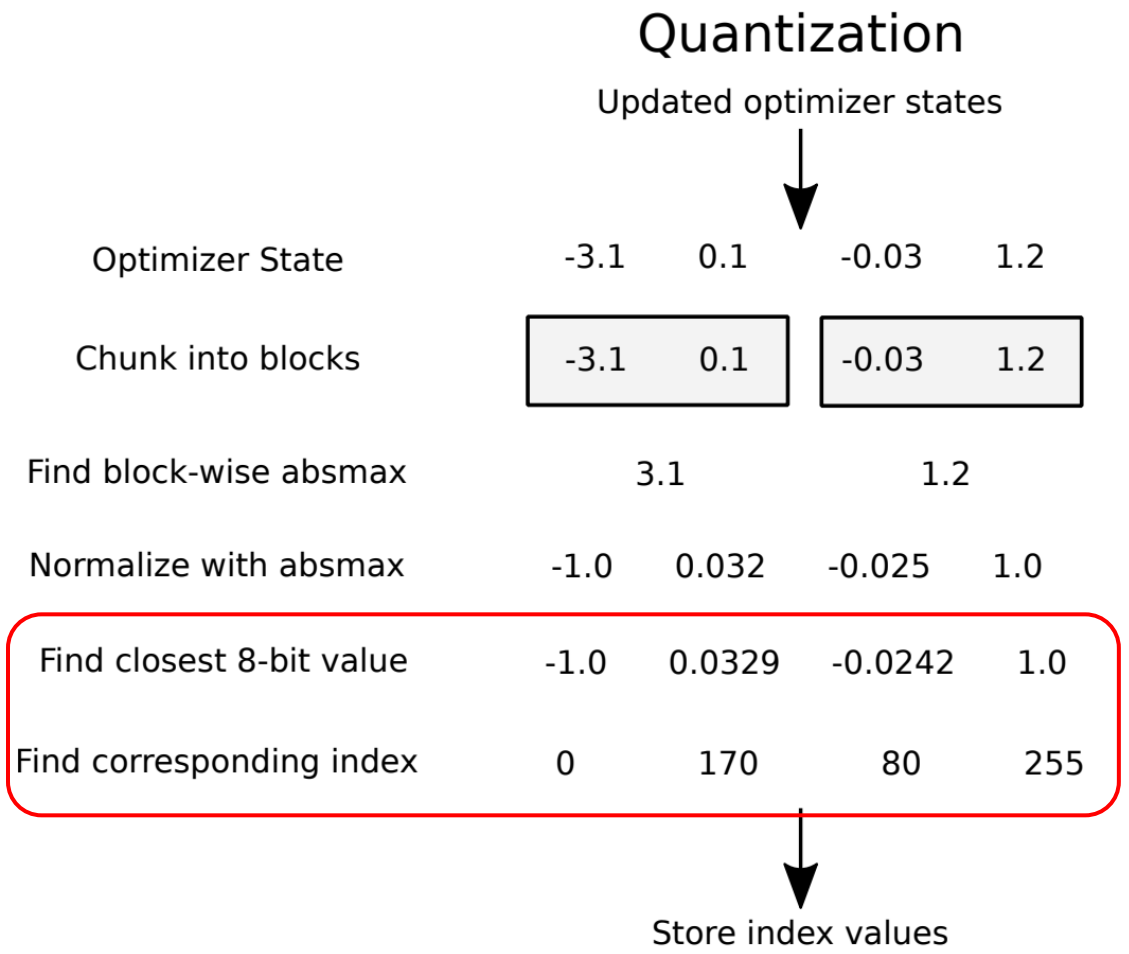
The quantization process contains 3 steps:

1. Segmentation into Blocks

2. Normalization Within Each Block

3. **Independent Quantization**

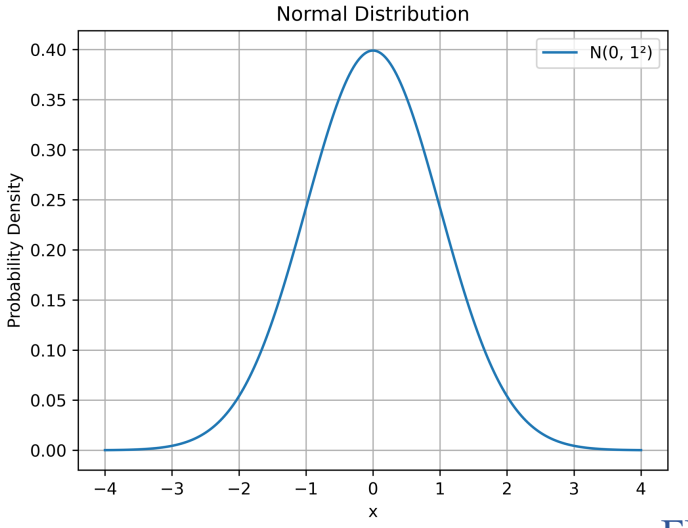
- Quantize each block separately
- Each block uses its own scaling factor for precision



QLoRA

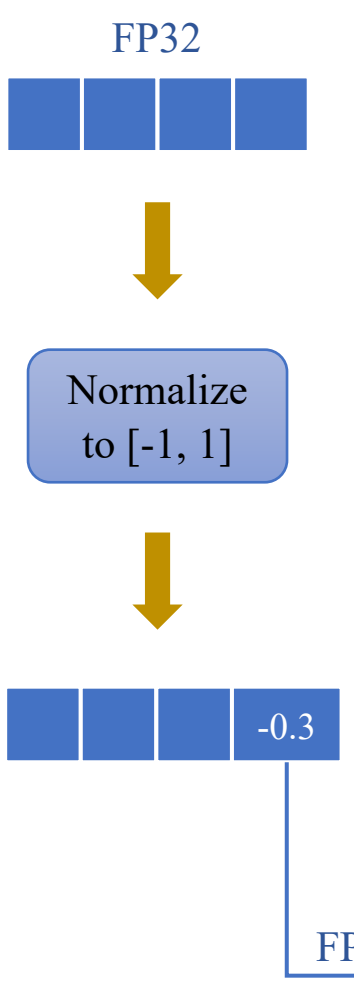
❖ 4-bit NormalFloat Quantization

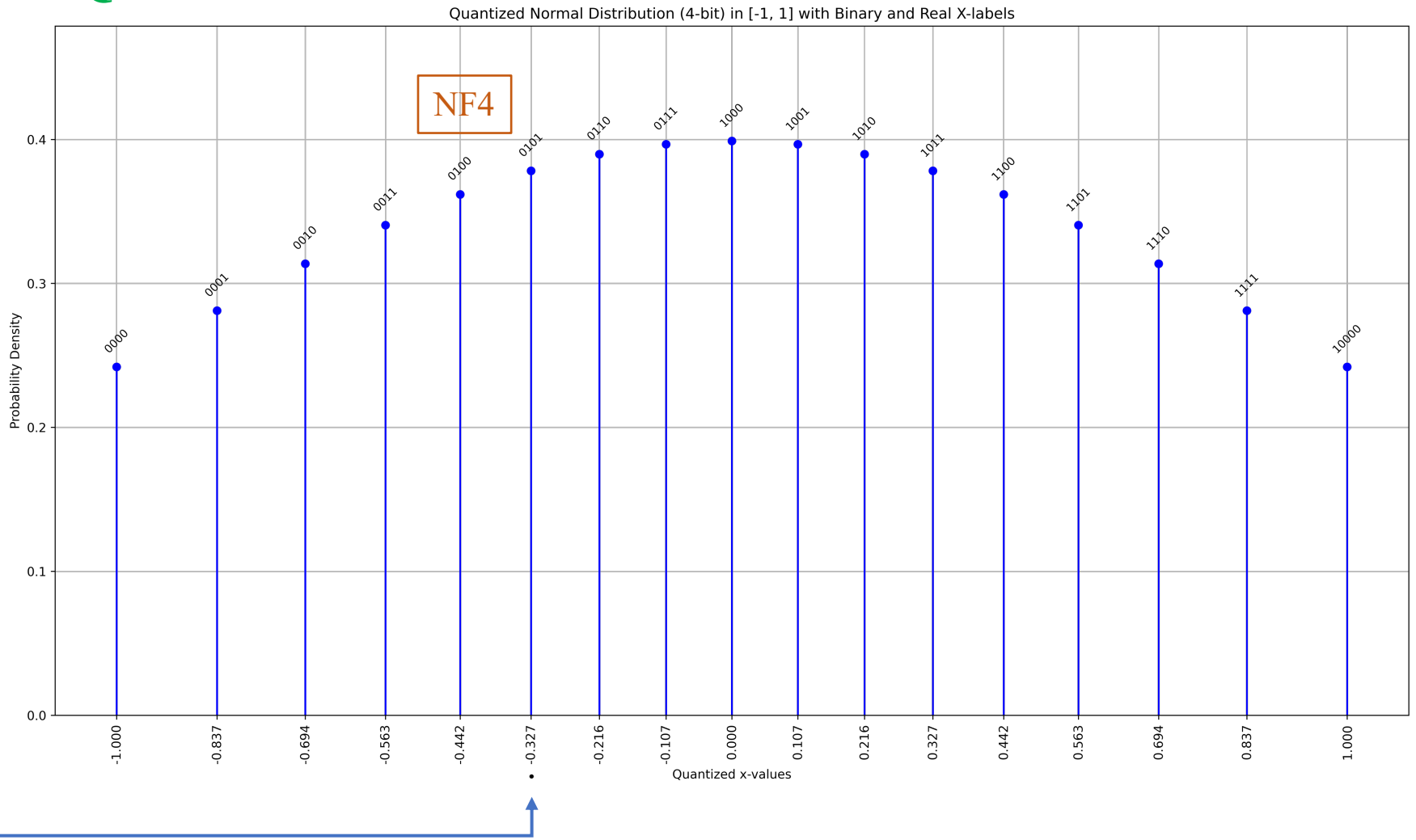
4-bit	value
0000	0
0001	1
…	
1110	14
1111	15



QLoRA

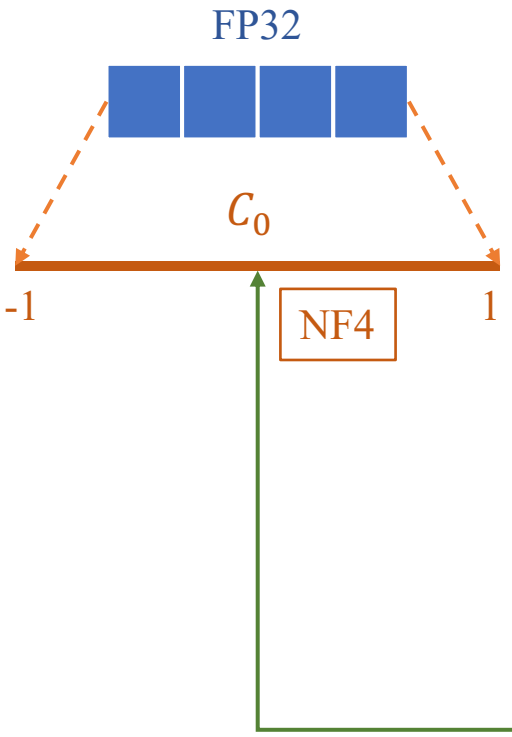
❖ 4-bit NormalFloat Quantization





QLoRA

❖ 4-bit NormalFloat Quantization



E NormalFloat 4-bit data type

The exact values of the NF4 data type are as follows:

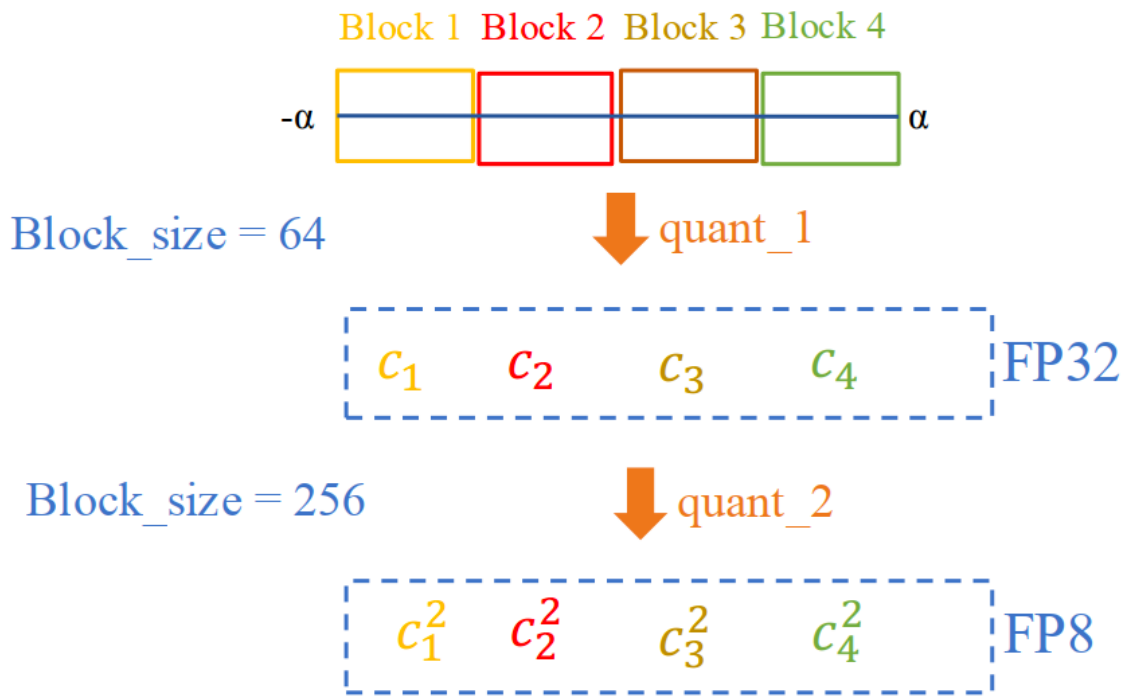
[-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]

4-bit	value	NF4
0000	0	-1.0
0001	1	-0.6961….
…	…	…
1110	14	0.7229…..
1111	15	1.0

QLoRA

❖ Double Quantization

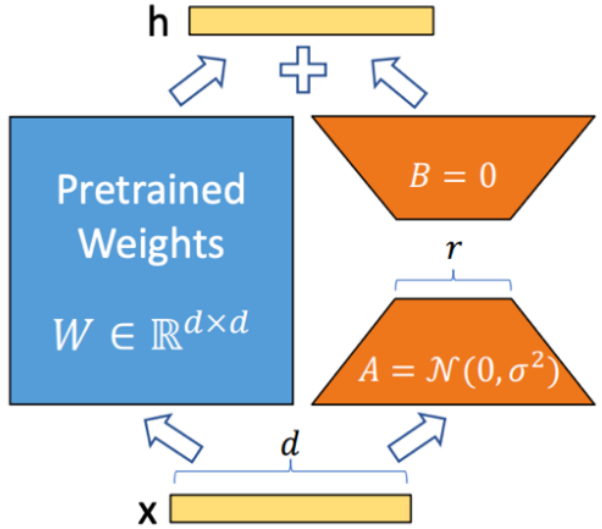
"We introduce *Double Quantization* (DQ), the process of quantizing the quantization constants."



QLoRA

❖ Double Quantization

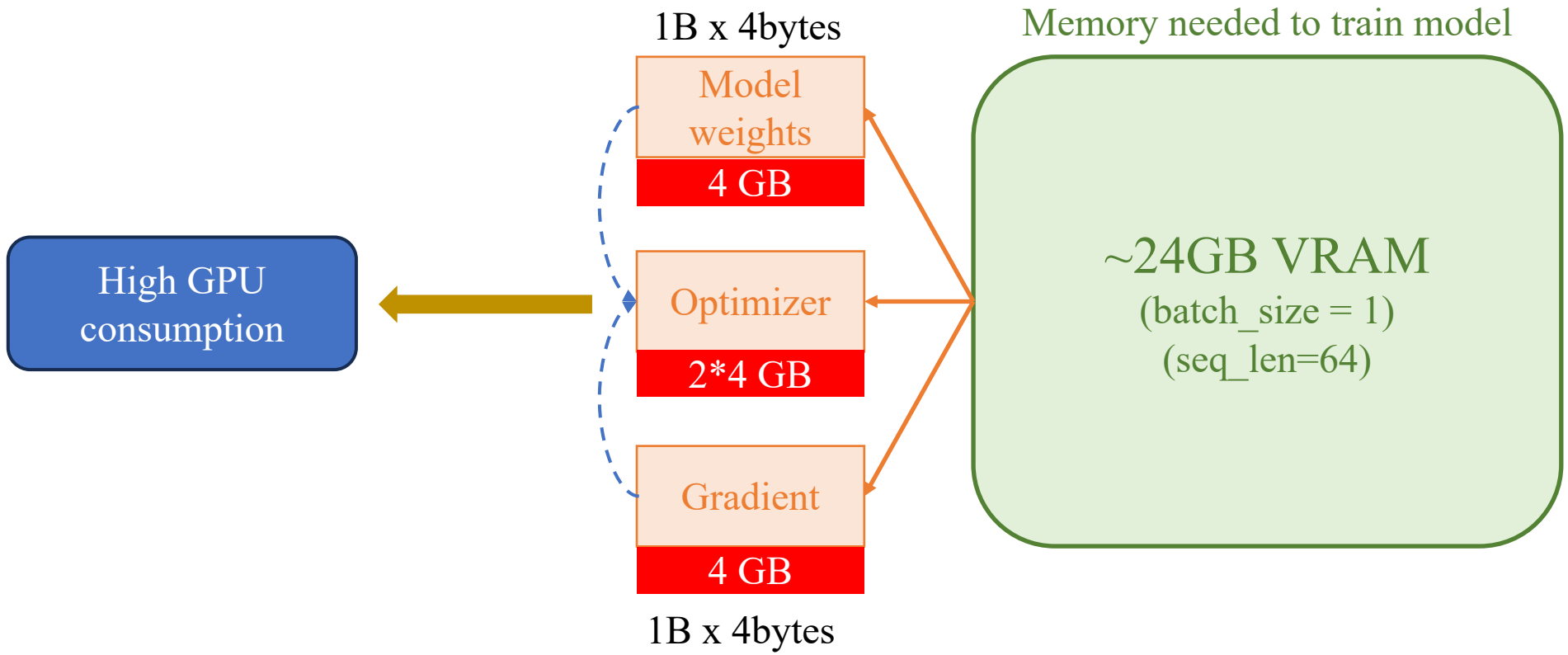
$$\mathbf{Y}^{\text{BF16}} = \mathbf{X}^{\text{BF16}} \text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{NF4}}) + \mathbf{X}^{\text{BF16}} \mathbf{L}_1^{\text{BF16}} \mathbf{L}_2^{\text{BF16}},$$



$$\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{k-bit}}) = \text{dequant}(\text{dequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}), \mathbf{W}^{\text{4bit}}) = \mathbf{W}^{\text{BF16}},$$

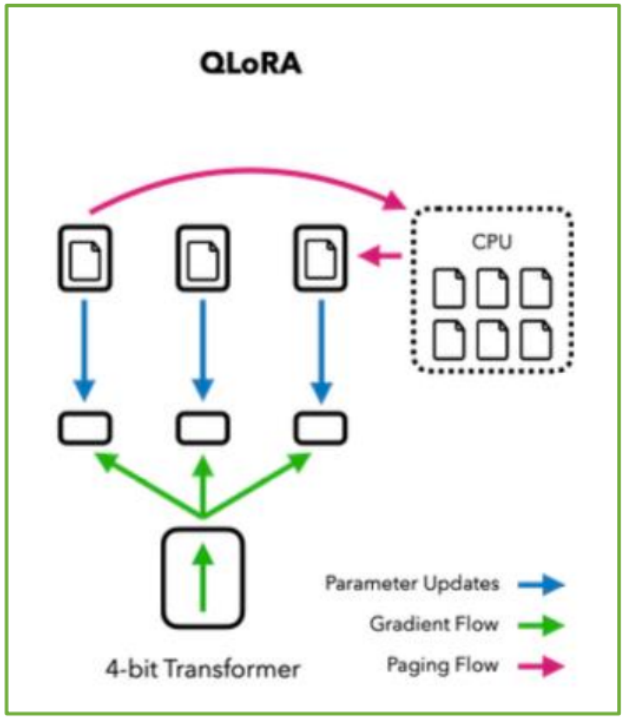
QLoRA

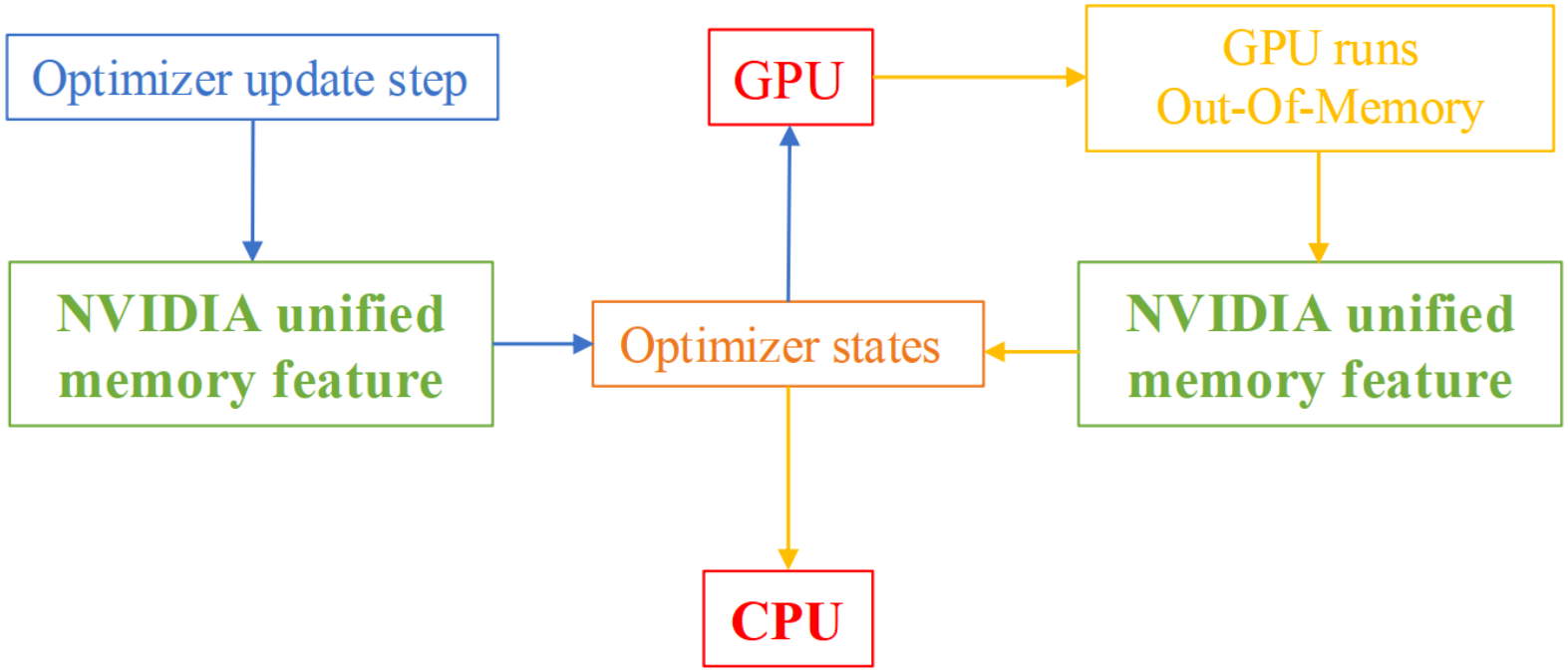
❖ Paged Optimizers



QLoRA

❖ Paged Optimizers





QLoRA

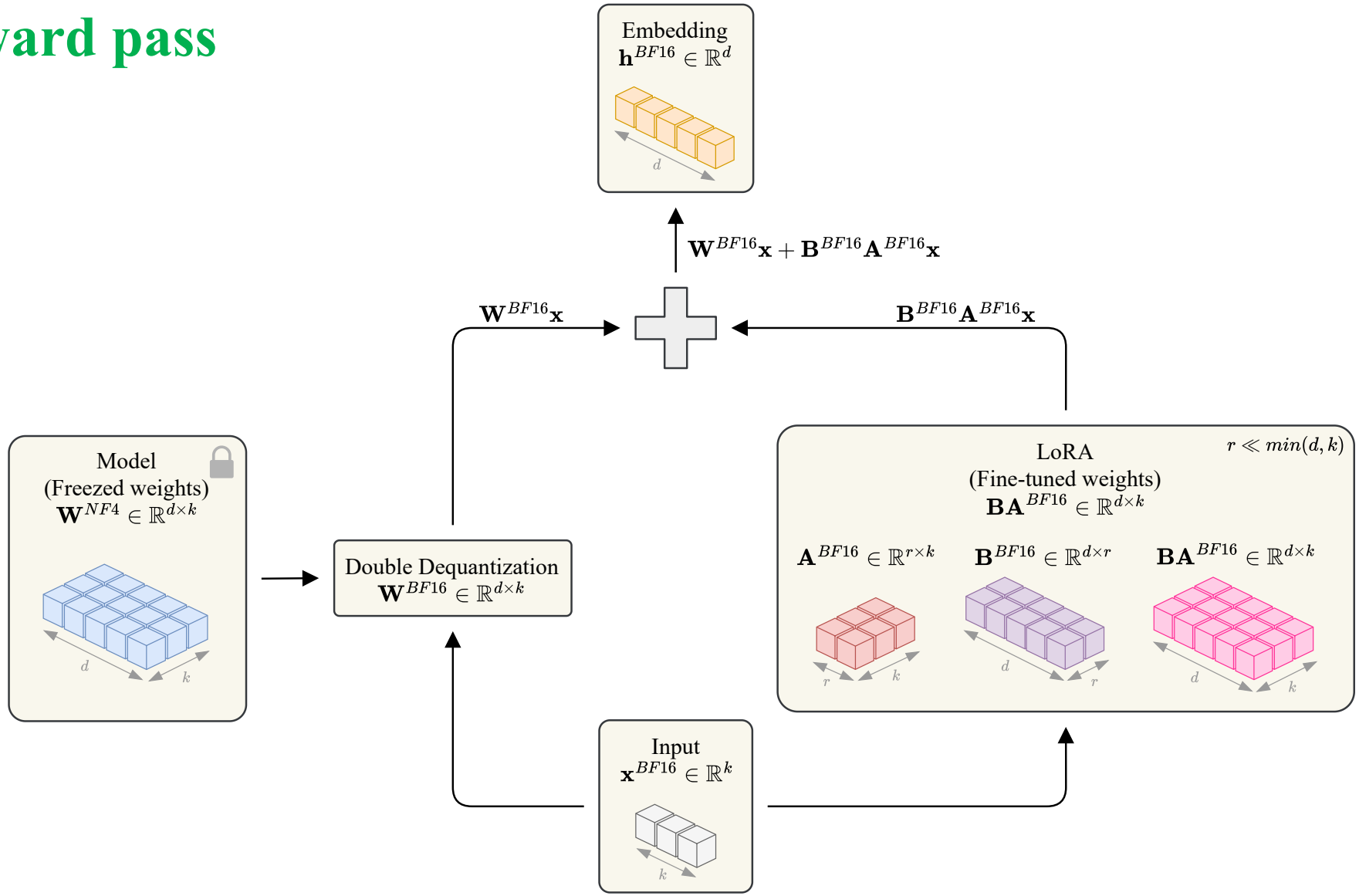
❖ Paged Optimizers

```python
import bitsandbytes as bnb

paged_optimizer = bnb.optim.PagedAdamW(
    trainable_params,
    lr=3e-4,
    weight_decay=0.0,
)
```

```python
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    per_device_train_batch_size=4,
    optim="paged_adamw_8bit",  # ← Paged Optimizer
    ...
)
```

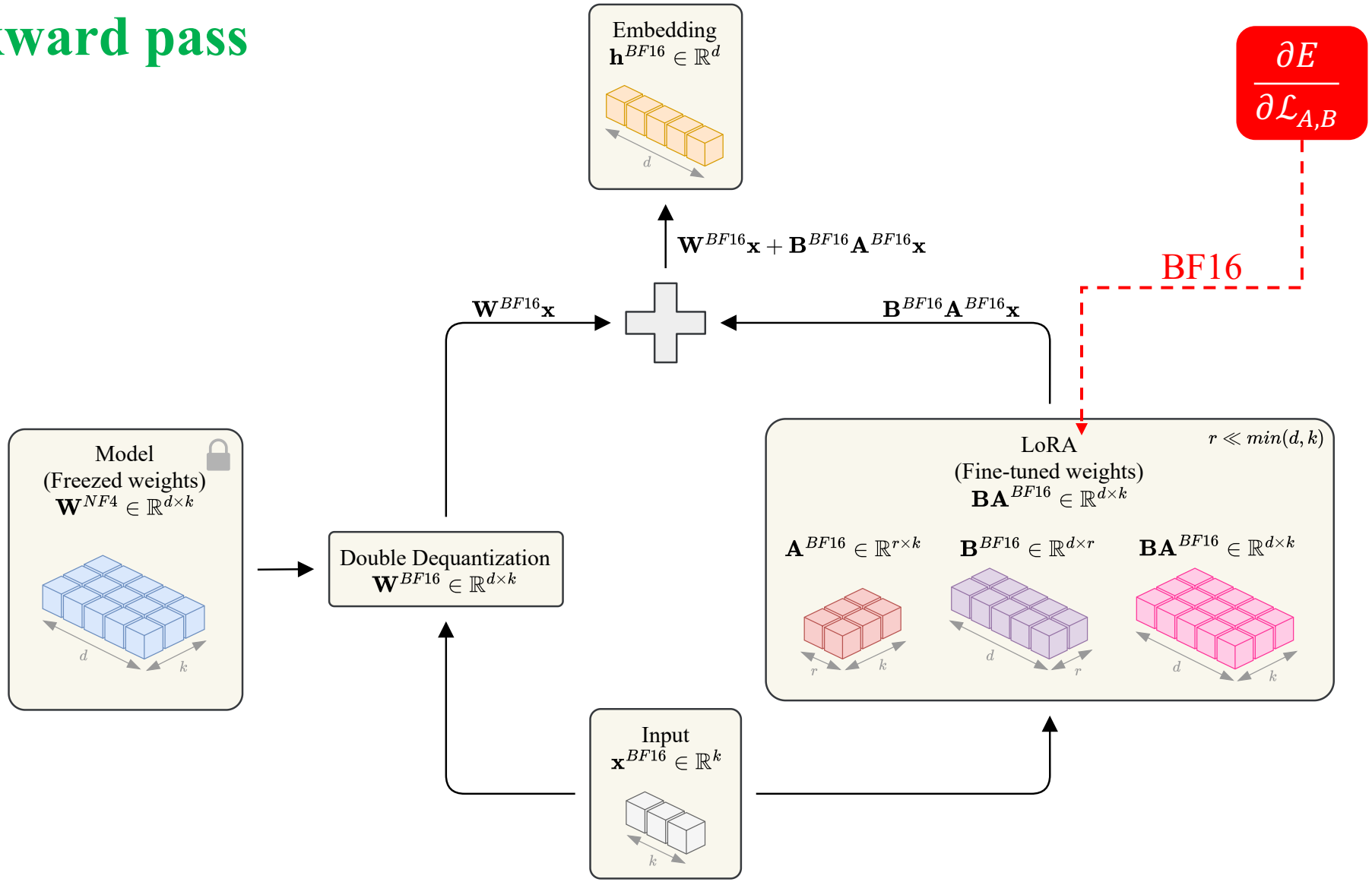
QLoRA

❖ Forward pass



QLoRA

❖ Backward pass



QLoRA

❖ Multi-backend Support

ROCm | Intel CPU | Apple Silicon / Metal (MPS)

Backend	Supported Versions	Python versions	Architecture Support	Status
AMD ROCm	6.1+	3.10+	minimum CDNA - `gfx90a`, RDNA - `gfx1100`	Alpha
Apple Silicon (MPS)	WIP	3.10+	M1/M2 chips	Planned
Intel CPU	v2.4.0+ (`ipex`)	3.10+	Intel CPU	Alpha
Intel GPU	v2.4.0+ (`ipex`)	3.10+	Intel GPU	Experimental
Ascend NPU	2.1.0+ (`torch_npu`)	3.10+	Ascend NPU	Experimental

https://huggingface.co/docs/bitsandbytes/main/en/non_cuda_backends

Practices

Practices

❖ Sentiment-based classification

UIT-VSFC (version 1.0) - Vietnamese Students' Feedback Corpus

Abstract: Students' feedback is a vital resource for the interdisciplinary research involving the combining of two different research fields between sentiment analysis and education. **V**ietnamese **S**tudents' **F**eedback **C**orpus (**UIT-VSFC**) is the resource consists of over 16,000 sentences which are human-annotated with two different tasks: sentiment-based and topic-based classifications. To assess the quality of our corpus, we measure the annotator agreements and classification evaluation on the UIT-VSFC corpus. As a result, we obtained the inter-annotator agreement of sentiments and topics with more than over 91% and 71% respectively. In addition, we built the baseline model with the Maximum Entropy classifier and achived approximately 88% of the sentiment F1-score and over 84% of the topic F1-score.

Paper: Kiet Van Nguyen, Vu Duc Nguyen, Phu Xuan-Vinh Nguyen, Tham Thi-Hong Truong, Ngan Luu-Thuy Nguyen, ***UIT-VSFC: Vietnamese Students' Feedback Corpus for Sentiment Analysis***, 2018 10th International Conference on Knowledge and Systems Engineering (KSE 2018), November 1-3, 2018, Ho Chi Minh City, Vietnam. Link.

Please download this dataset/corpus here .

Sentiment/Topic	Positive (%)	Negative (%)	Neutral (%)	Total (%)
Lecturer	33.57	25.38	1.81	71.76
Curriculum	3.40	14.39	1.00	18.79
Facility	0.11	4.21	0.08	4.40
Others	1.61	2.01	1.43	5.04
Total	49.69	45.99	4.32	100.00

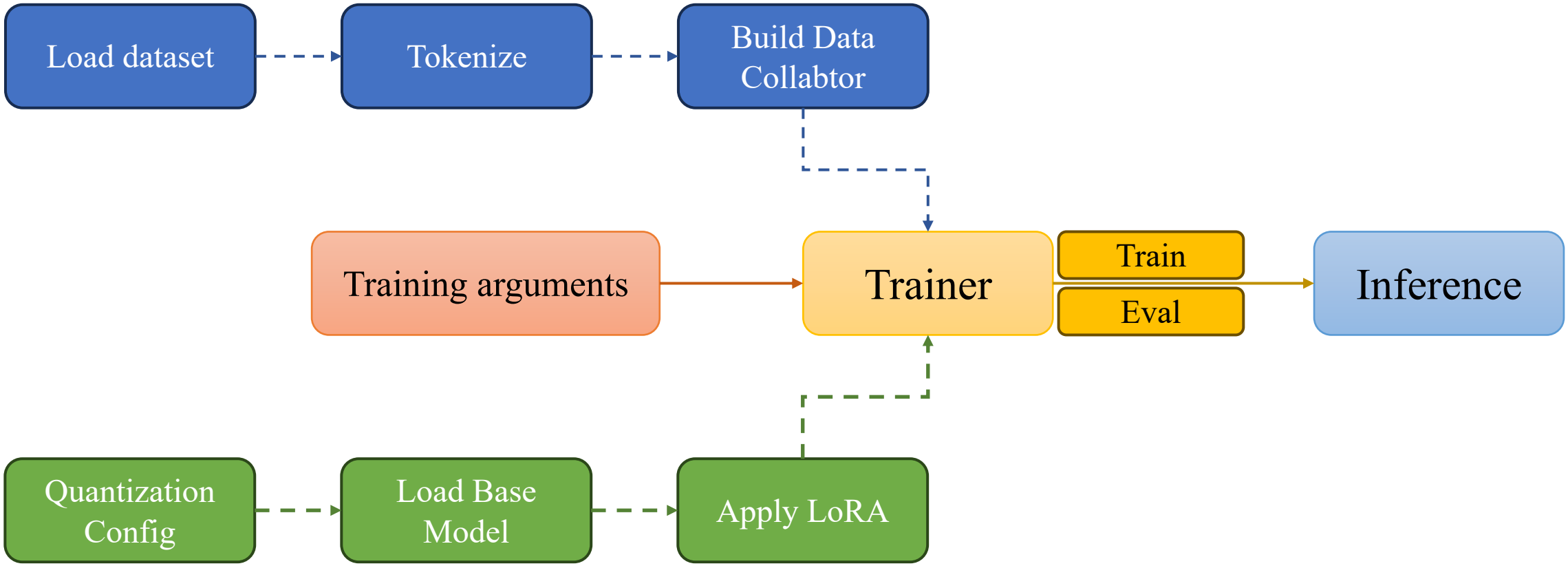
Bảng 1: Bảng thống kê giữa Chủ đề và Cảm xúc.

Tập dữ liệu	Tỷ lệ (%)	Số lượng câu
Tập huấn luyện	70%	11.426
Tập phát triển	10%	1.538
Tập kiểm thử	20%	3.166
Tổng	100%	16.130

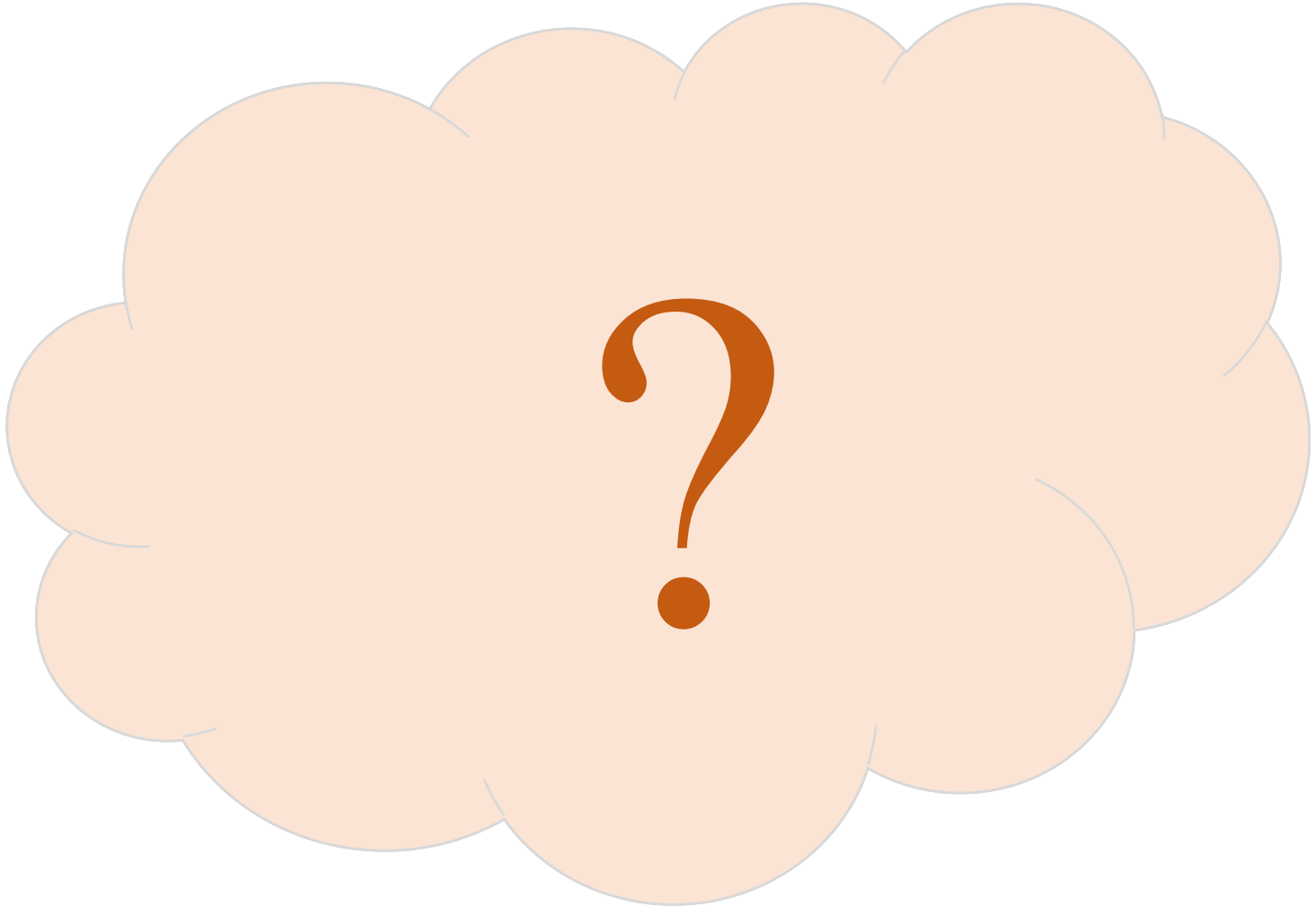
Bảng 2: Bảng thống kê các tập dữ liệu.

Practices

❖ Pipeline



Question




Thank you!