Deep Learning for NLP

Lecture 3: Efficient training

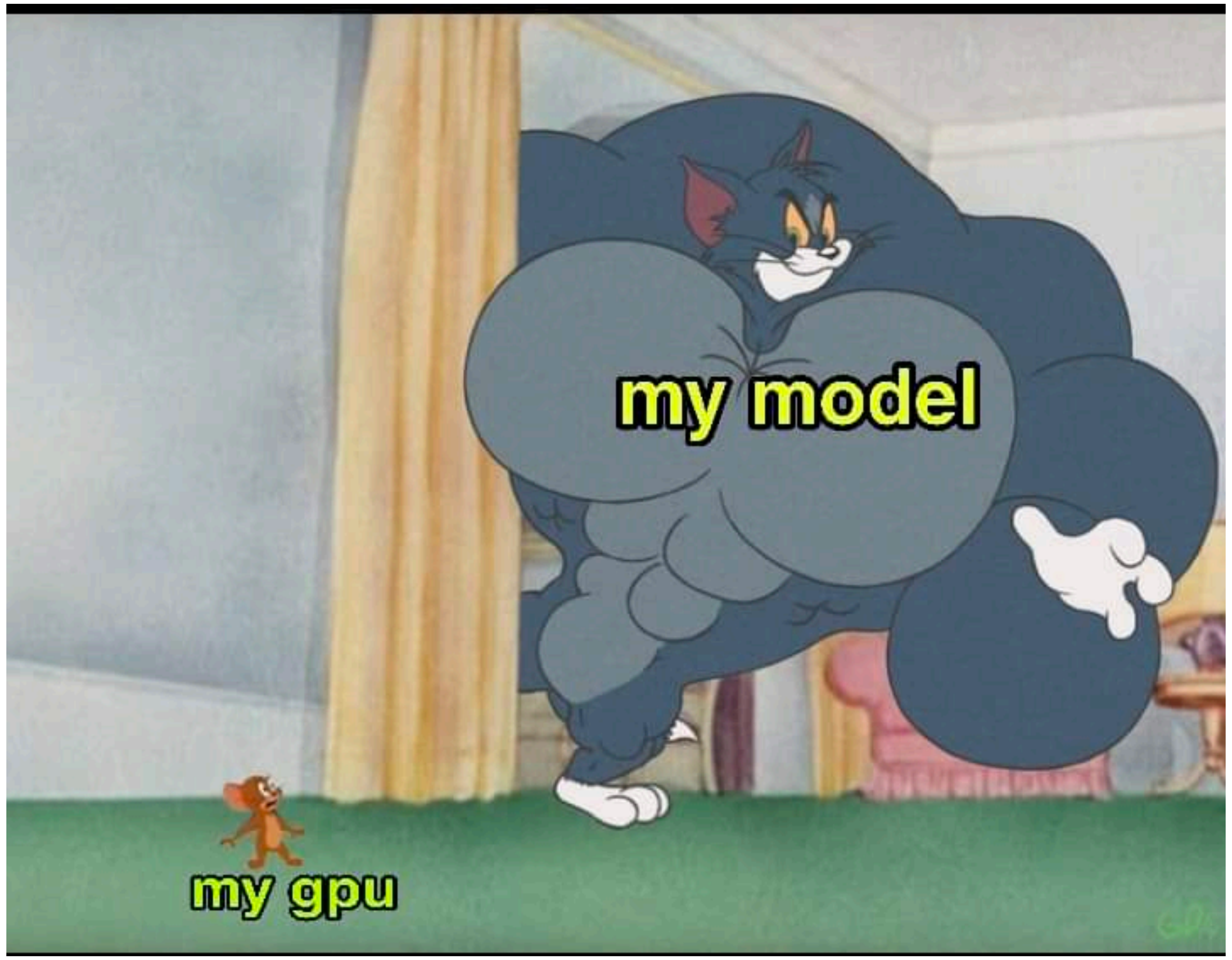
Teaching: Khyathi Chandu

Outline

- PEFT
- LoRA in depth
- Quantization
- QLoRA

Much much bigger models ...much much bigger yields

- Models are racing ahead
- Compute catching up



Model sizes seize the show

GPU memory: 16-80 GB

Out of Memory Issues

2018: BERT

"… when using a GPU with 12GB - 16GB of RAM, you are likely to encounter out-of-memory issues…" (c)

For example, the largest Transformer explored in Vaswani et al. (2017) is (L=6, H=1024, A=16) with 100M parameters for the encoder, and the largest Transformer we have found in the literature is (L=64, H=512, A=2) with 235M parameters (Al-Rfou et al., 2018). By contrast, $\text{BERT}_{\text{BASE}}$ contains 110M parameters and $\text{BERT}_{\text{LARGE}}$ contains 340M parameters.

System	Seq Length	Max Batch Size
`BERT-Base`	64	64
...	128	32
...	256	16
...	320	14
...	384	12
...	512	6
`BERT-Large`	64	12
...	128	6
...	256	2
...	320	1
...	384	0
...	512	0

https://github.com/google-research/bert#out-of-memory-issues

Slide from Vlad Lialin

GPU Memory breakdown

		OPT-1.3B, 16-bit float, seq 512
cuDNN and CUDA		~1Gb
Model weights	size(float) * N	2.6Gb
Gradients	size(float) * $N_{trainable}$	2.6Gb
Hidden states	*~size(float) L (20 h seq + 3 seq²)*	1Gb per example
Optimizer states	2 * size(float) * $N_{trainable}$	5.2Gb
(maybe) fp32 copy of the gradients	4 * Ntrainable	10.2Gb

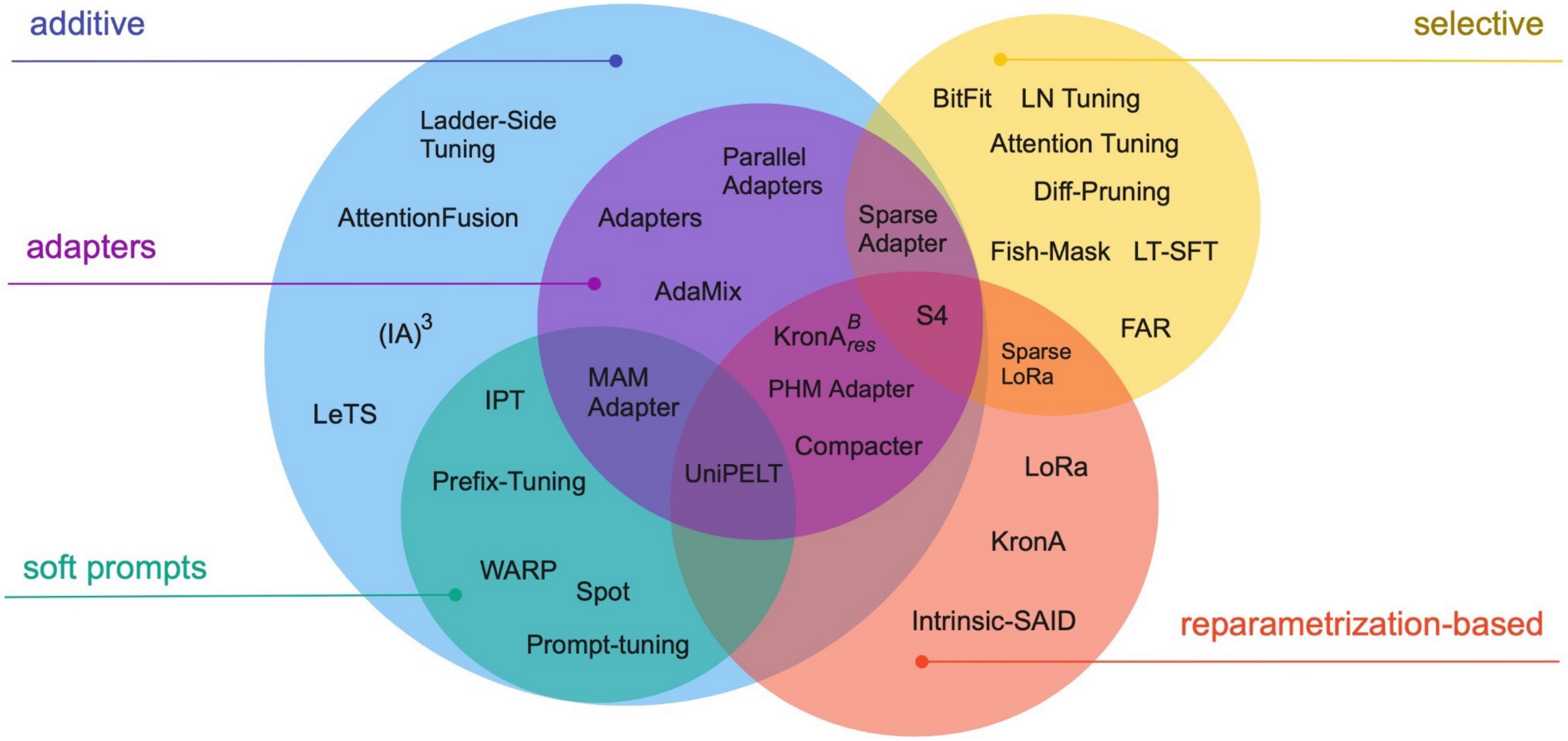
Estimate: 12.4Gb, actual: 11.0Gb *(after empty_cache)*

Slide from Vlad Lialin

Reduction in params

- Reduce the number of trainable parameters
  - Can reduce the memory significantly

Overview of PEFT



Additive - Adapters

- Initially developed for multi-domain image classification [Rebuffi et al 2017]
  - Adding domain-specific layers between modules
- **Core idea:** Add fully connected layers after attention and FFN layers
- Adapters have *Much smaller hidden dim*
- Similar performance with <4% parameter tuning

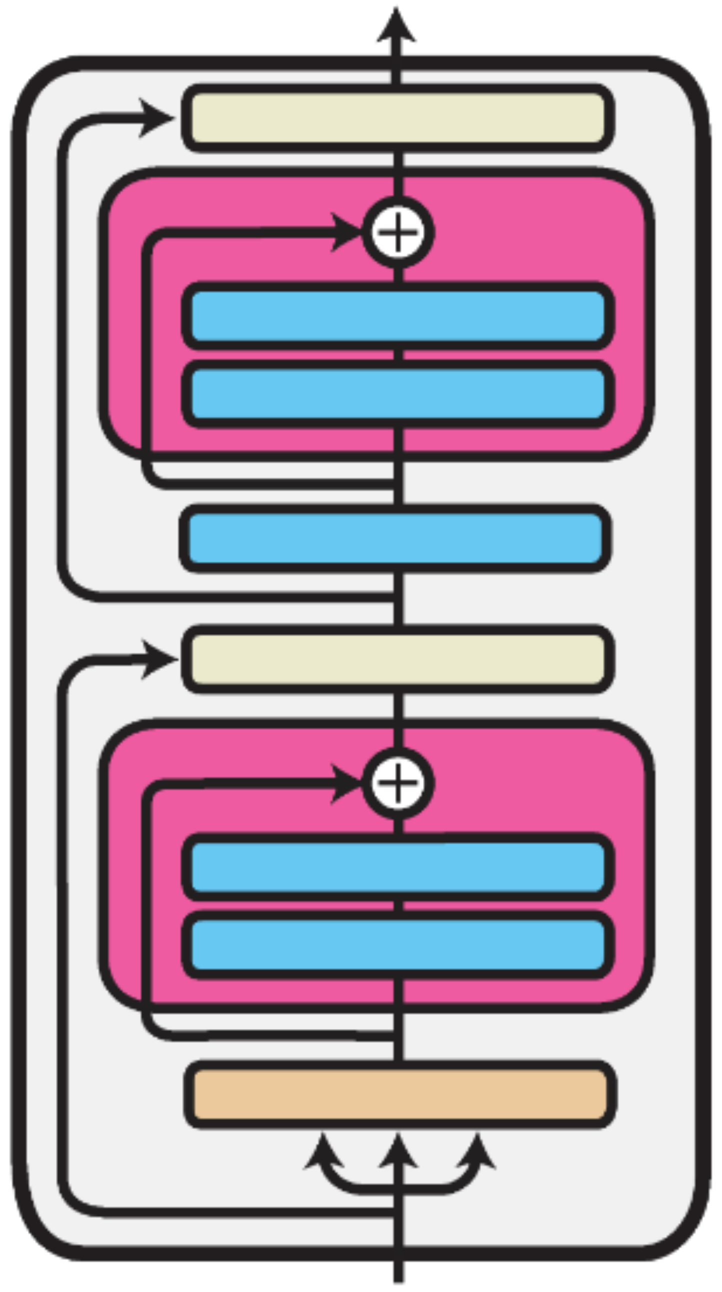


Image source: adapterhub.ml

1) Bottleneck Adapter

```
def transformer_block_with_adapter(x):
    residual = x
    x = SelfAttention(x)
    x = FFN(x)  # adapter
    x = LN(x + residual)
    residual = x
    x = FFN(x)  # transformer FFN
    x = FFN(x)  # adapter
    x = LN(x + residual)
    return x
```

https://arxiv.org/pdf/2303.15647.pdf

AdapterHub

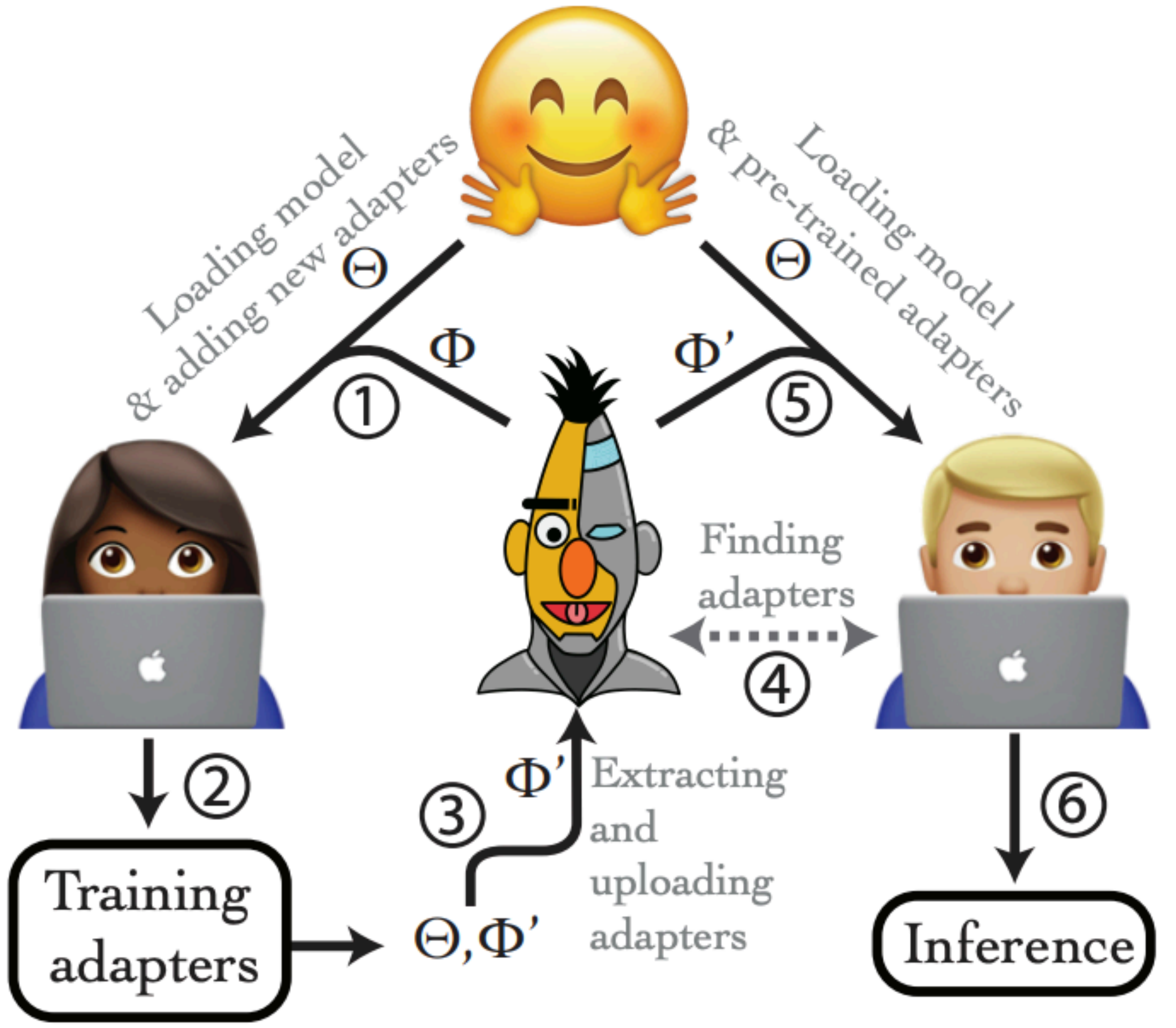


Figure 1: The AdapterHub Process graph. Adapters Φ are introduced into a pre-trained transformer Θ (step ①) and are trained (②). They can then be extracted and open-sourced (③) and visualized (④). Pre-trained adapters are downloaded on-the-fly (⑤) and stitched into a model that is used for inference (⑥).

https://arxiv.org/pdf/2007.07779.pdf

Additive/Soft prompts - Prompt tuning

- **Core idea:** Prepend the input embeddings with trainable tensor (soft-prompt)
  - Optimized through gradient descent
- Limitations:
  - gap still exists – prompt tuning fully comparable at ~10B Scale
  - Inference overhead (as quadratic computation with every increased prefix token)

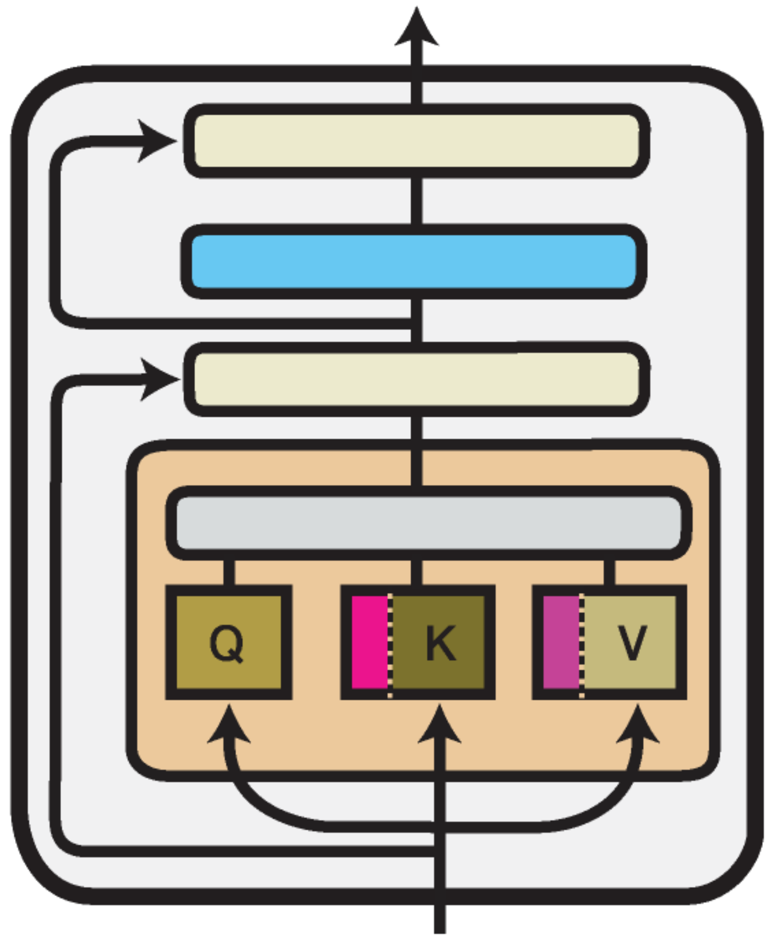


Image source: adapterhub.ml

```
def prompt_tuning_attention(input_ids):
    q = x @ W_q
    k = cat([s_k, x]) @ W_k  # prepend a
    v = cat([s_v, x]) @ W_v  # soft prompt
    return softmax(q @ k.T) @ V
```

https://arxiv.org/pdf/2303.15647.pdf

Additive – (IA)3

- **Core idea:** Rescale key, value & hidden activations

- Advantages:

  - Training only l_k, l_v, l_ff

  - Minimal overhead (l_ff)

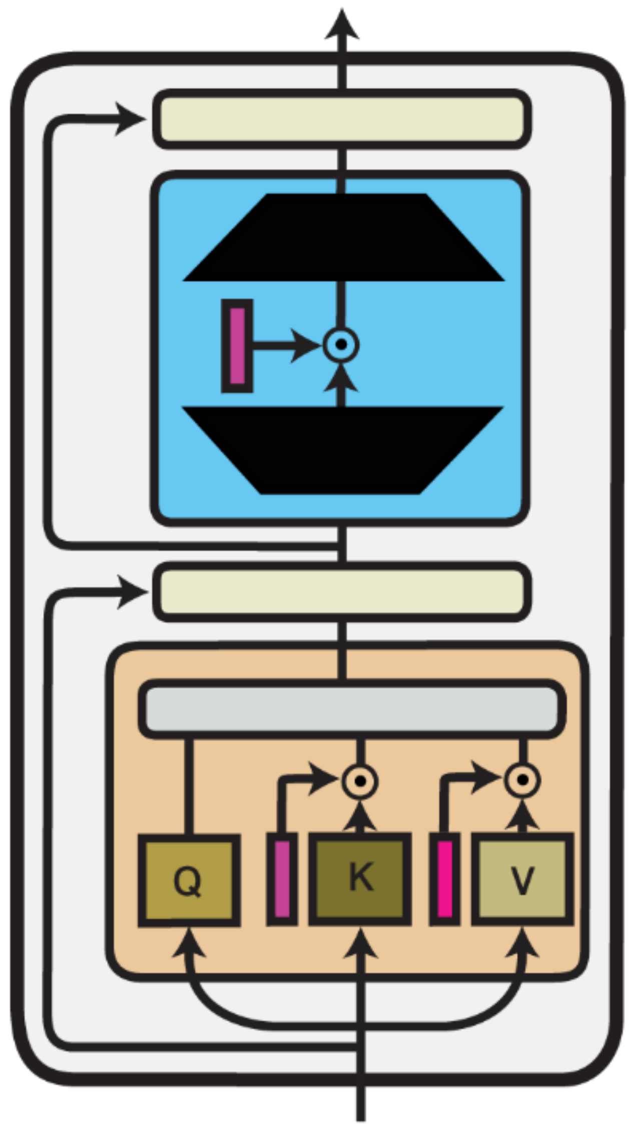


Image source: adapterhub.ml

```
def transformer_block_with_ia3(x):
    residual = x
    x = ia3_self_attention(x)
    x = LN(x + residual)
    residual = x
    x = x @ W_1           # FFN in
    x = l_ff * gelu(x)    # (IA)3 scaling
    x = x @ W_2           # FFN out
    x = LN(x + residual)
    return x

def ia3_self_attention(x):
    k, q, v = x @ W_k, x @ W_q, x @ W_v
    k = l_k * k
    v = l_v * v
    return softmax(q @ k.T) @ V
```

https://arxiv.org/pdf/2303.15647.pdf

Selective – BitFit

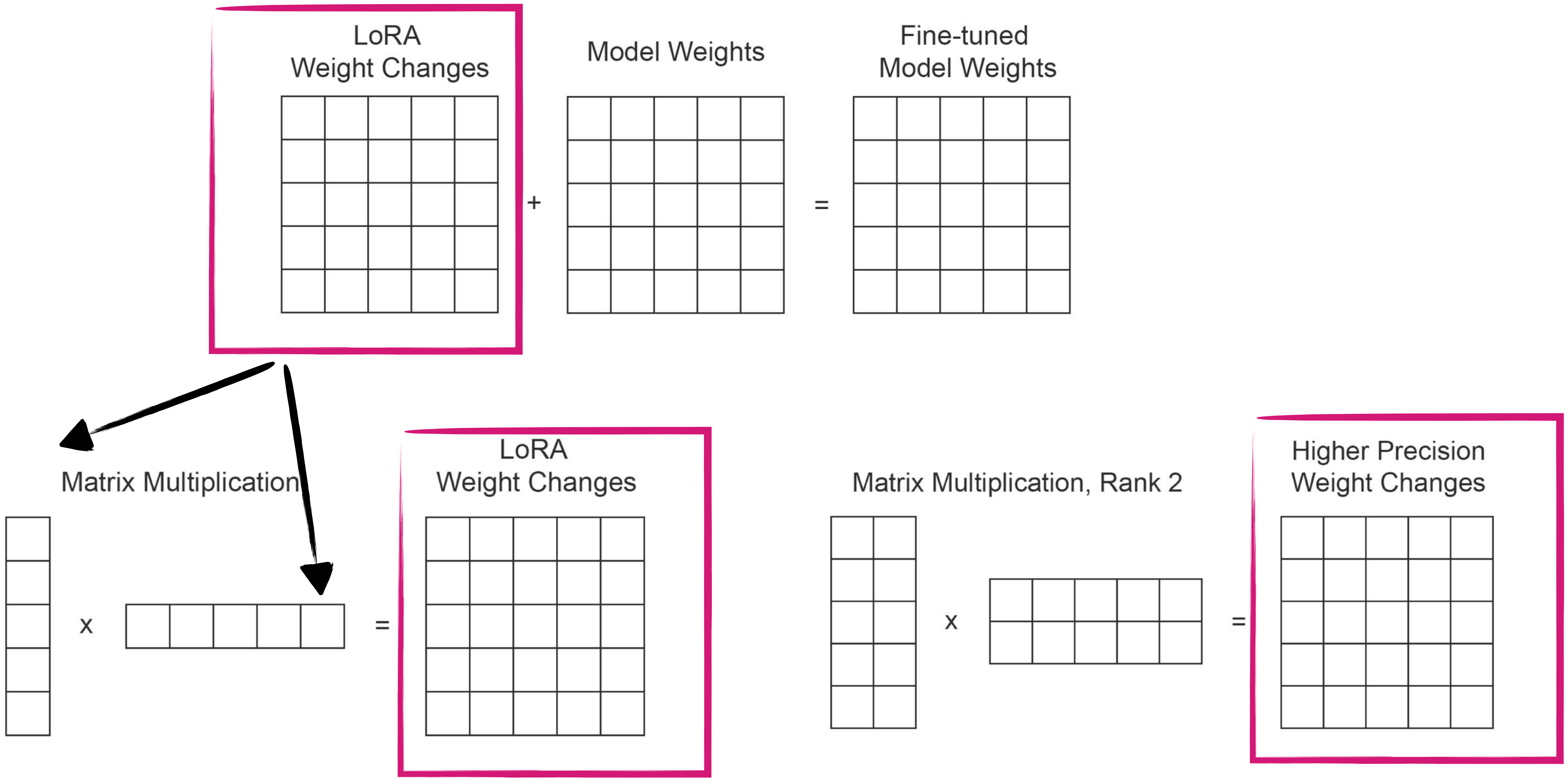
- **Core idea:** Fine-tune only model biases
- Advantages:
  - Super easy implementation
- Limitation:
  - Works well only for smaller models, does not scale well for larger models

```
params = (p for n, p
          in model.named_parameters()
          if "bias" in n)
optimizer = Optimizer(params)
```

Reparametrization – LoRA

- Copy weights and gradients for tuning
- Adapters are param-efficient (adapter params trainable & original model params are frozen)
- ❄️ Pretrained model weights
- Trainable rank decomposition matrices

Decomposition



Reparametrization

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_{\Phi}(y_t|x, y_{<t})\right)$$

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

Low Rank representation to encode $\Delta\phi$

Encoded with a smaller set of params

Low Rank Adaptation

- Pre-trained language models have a low "instrisic dimension"
    - => can learn efficiently in this smaller sub-space

Pretrained Weight Matrix

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

Update Step

$$W_0 + \Delta W$$

Update **Decomposition**

$$W_0 + BA$$

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

$$r \ll \min(d, k)$$

Forward Pass $h = W_0 x + \Delta W x = W_0 x + BAx$

Experimental Benefits

Model & Method	# Trainable Parameters	E2E NLG Challenge BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M (Adapter^{L})*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M (Adapter^{L})*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M (Adapter^{H})	11.09M	$67.3_{\pm.6}$	$8.50_{\pm.07}$	$46.0_{\pm.2}$	$70.7_{\pm.2}$	$2.44_{\pm.01}$
GPT-2 M (FT^{Top2})*	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (PreLayer)*	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.85}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{71.8}_{\pm.1}$	$\mathbf{2.53}_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	$\mathbf{2.49}_{\pm.0}$
GPT-2 L (Adapter^{L})	23.00M	$68.9_{\pm.3}$	$8.70_{\pm.04}$	$46.1_{\pm.1}$	$71.3_{\pm.2}$	$2.45_{\pm.02}$
GPT-2 L (PreLayer)*	0.77M	70.3	8.85	46.2	71.7	2.47
GPT-2 L (LoRA)	0.77M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

LoRA with Quantization

- Compression: Decomposing to BA (low-rank matrices)
  - => compressed memory, speed up
- LoRA does not affect performance (much)
- Quantization, such as int8 can affect the performance
- You can club LoRA with other techniques such as prefix-tuning

Quantization basics

- 4-bit $\in$ [-8, 7] (16 values)
- 8-bit $\in$ [-127, 127] (256 values)
- 32 bit can pretty much fit for practical purposes

QLoRA has one storage data type (usually 4-bit NormalFloat) and a computation data type (16-bit BrainFloat).

We dequantize the storage data type to the computation data type to perform the forward and backward pass, but we only compute weight gradients for the LoRA parameters which use 16-bit BrainFloat.

3 key steps

- **Normalization:** Weights with 0 mean and unit variance
- **Quantization:** mapping original high precision weights to low-precision weights
    - Evenly spaced bins (for NF4)
- **Dequantization:** mapping back to original values
    - But, stored in 4-bit and dequantized when used in computations

Example

- Normalization
    - Convert all weights to -1, 1 centered around 0
- Quantization
    - 4 bit integers represent 16 evenly spaced levels
    - -1.0, -0.86, -0.73, -0.6, -0.46, -0.33, -0.2, -0.06, 0.06, 0.2, 0.33, 0.46, 0.6, 0.73, 0.86, 1.0
    - w=0.42 <—> closest to 0.46
    - New w=0.46 => store 4-bit integer 12
- Dequantization
    - Dequantize 12 back to 0.46
    - Error = 0.46-0.42=0.04

Thank You!