Fine-tuning using Low Rank Adaptation

Deep Learning and Modern Applications

Theja Tulabandhula

University of Illinois Chicago

After this lecture, you will be able to:

- Explain why full fine-tuning becomes computationally infeasible for large models
- Derive the parameter savings from low-rank matrix decomposition
- Describe the LoRA architecture and how it modifies pretrained weights
- Choose appropriate rank values and scaling factors for different tasks
- Connect LoRA to transfer learning strategies covered in earlier lectures

Recap: Transfer Learning Approaches

- **Freeze all, train head**: Fast but limited flexibility
- **Full fine-tuning**: Flexible but expensive and risks catastrophic forgetting
- **LoRA**: A principled middle ground

The Transfer Learning Spectrum

Approach	Trainable Params	Memory	Flexibility	Forgetting Risk
Freeze + Head	Very Few	Low	Low	Minimal
LoRA	Few	Low	Medium-High	Low
Full Fine-Tuning	All	High	High	High

LoRA enables **efficient, targeted adaptation** while preserving pretrained knowledge.

- Modern models have **millions to billions of parameters**
- Full fine-tuning updates **every weight**
- Costs grow in:
  - GPU memory
  - Training time
  - Optimizer state
- Works for small CNNs
- Breaks down for large dense models

What Fine-Tuning Really Does

- Pretraining gives us a weight matrix **W**
- Fine-tuning modifies it
- Mathematically:

$$W \to W + \Delta W$$

- Question: does ΔW really need to be full-sized?

Why Freezing Weights Alone Is Not Enough

- Freezing W:
  - Saves memory
  - Preserves knowledge
- But:
  - Model cannot adapt
  - No task-specific learning
- We need **controlled, limited adaptation**

Standard Linear Layer

- Input: $x \in \mathbf{R}^d$
- Weights: $W \in \mathbf{R}^{d \times d}$
- Output:

$$y = Wx$$

- Parameters: d^2

What Fine-Tuning Changes

- Fine-tuning learns ΔW
- Output becomes:

$$y = (W + \Delta W)x$$

- Full fine-tuning learns all d^2 entries of ΔW

Key Observation

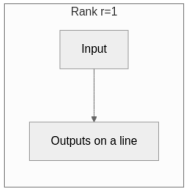
- Empirically:
  - Many updates are **correlated**
  - Effective change lives in a **lower-dimensional subspace**
- Suggests ΔW may be **low-rank**

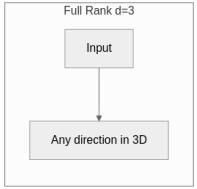
What Does Rank Mean?

- Rank = number of **linearly independent** rows (or columns)
- Full rank matrix: can map to any direction in output space
- Low rank matrix: outputs constrained to a subspace

Geometric Intuition

- A rank-r matrix can only produce outputs in an r-dimensional subspace
- Example: rank-1 matrix outputs lie on a **line**
- Rank-2 outputs lie on a **plane**
- Key insight: task adaptation often needs only a few directions





Low-Rank Decomposition

- Any low-rank matrix can be written as:

$$\Delta W = BA$$

Where: - $B \in \mathbf{R}^{d \times r}$ - $A \in \mathbf{R}^{r \times d}$ - $r \ll d$

Parameter Count Comparison

- Full update: d^2
- Low-rank update: $2dr$

Worked Example: Parameter Savings

Setting	Value
Dimension d	4096
Rank r	8
Full ΔW	$d^2 = 16{,}777{,}216$
LoRA (A + B)	$2 \times d \times r = 65{,}536$
Reduction Factor	$\frac{16{,}777{,}216}{65{,}536} = 256\times$

Even for modest r, savings are **dramatic**.

Visualizing Low-Rank Updates

LoRA Definition

- Freeze original weights **W**
- Learn low-rank matrices **A** and **B**
- Effective weight:

$$W_{\text{eff}} = W + \frac{\alpha}{r} BA$$

Why the Scaling Factor?

- α controls update magnitude
- Keeps update comparable to pretrained scale
- Improves optimization stability

What Is Trainable?

- Trainable:
  - A (down-projection)
  - B (up-projection)
- Frozen:
  - W (original pretrained weights)
  - Biases (usually frozen)

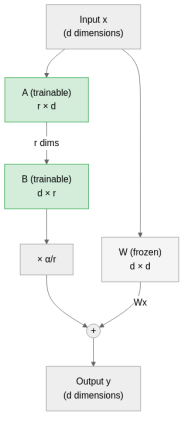
Initialization Strategy

- **A**: Random Gaussian, small variance ($\sim \mathcal{N}(0, \sigma^2)$)
- **B**: Initialized to **zero**

Why zero for B?

- At training start: $\Delta W = BA = 0$
- Model behaves **exactly** like pretrained model
- Enables gradual, stable adaptation
- Prevents sudden disruption of pretrained features

LoRA Structure



The bottleneck through r dimensions is where compression happens.

Gradient Flow

- Gradients flow only through A and B
- W receives **no gradients**
- Optimizer state only for A and B

Memory Implications

- No gradients for W
- No optimizer states for W
- Massive VRAM savings
- Enables fine-tuning on limited hardware

Stability Benefits

- Base model knowledge preserved
- Reduced catastrophic forgetting
- Controlled adaptation

LoRA in a Fully Connected Network

- Apply LoRA to:
  - Large dense layers
- Skip:
  - Small layers
  - Bias terms

Example: MLP with LoRA

- Original:
  - Linear $\rightarrow$ ReLU $\rightarrow$ Linear
- With LoRA:
  - LoRA applied only to weight matrices
- Forward pass unchanged conceptually

PyTorch Implementation Sketch

```
class LoRALinear(nn.Module):
    def __init__(self, in_dim, out_dim, r=8, alpha=16):
        super().__init__()
        self.W = nn.Linear(in_dim, out_dim, bias=False)
        self.W.weight.requires_grad = False  # Freeze

        self.A = nn.Parameter(torch.randn(r, in_dim) * 0.01)
        self.B = nn.Parameter(torch.zeros(out_dim, r))
        self.scale = alpha / r

    def forward(self, x):
        base = self.W(x)
        lora = (x @ self.A.T @ self.B.T) * self.scale
        return base + lora
```

Choosing Rank r

- Small r: Less capacity, more efficient
- Larger r: More expressive, higher cost

Hyperparameter Selection Guide

Hyperparameter	Typical Values	Guidance
Rank r	4, 8, 16, 32	Start small; increase if underfitting
Scaling α	r to $2r$	Higher α = stronger updates
Target layers	Attention projections	Start with Q, V; add K, O if needed
Learning rate	10^{-4} to 10^{-3}	Often higher than full fine-tuning

Rule of thumb: $\frac{\alpha}{r} \approx 1$ or $\frac{\alpha}{r} \approx 2$

Why LoRA Works Well at Scale

- Large models are overparameterized
- Task adaptation is often low-dimensional
- LoRA exploits this mismatch

Where LoRA Is Used in Practice

- Large projection matrices
- Repeated structures
- Especially effective where:
  - Parameter count is high
  - Updates are correlated

Hardware Feasibility

- Full fine-tuning: often impossible
- LoRA:
  - Single GPU
  - Modest VRAM
- Enables experimentation and iteration

Fine-Tuning Strategies

Method	Trainable Params	Memory	Flexibility
Full Fine-Tuning	All	High	High
Freeze + Head	Few	Low	Low
LoRA	Few	Low	Medium-High

When LoRA Is Not Ideal

- Extremely small models
- Tasks requiring large structural changes
- When full fine-tuning is affordable

QLoRA (High Level)

- Combine:
  - Quantized base model (4-bit or 8-bit)
  - LoRA updates in higher precision (FP16/BF16)
- Base weights stored in low precision
- LoRA gradients computed in higher precision

Why This Matters

- Pushes feasibility further
- Enables training 65B+ models on single GPU
- Still preserves LoRA principles
- Memory reduction stacks with quantization savings

Other PEFT Methods

- **Prefix Tuning**: Learn virtual tokens prepended to input
- **Adapter Layers**: Insert small bottleneck modules
- **IA3**: Learn rescaling vectors instead of matrices

LoRA remains popular due to simplicity and strong empirical performance.

Environmental Impact

- Full fine-tuning:
  - Multiple GPUs, days of training
  - High energy consumption and carbon footprint
- LoRA:
  - Single GPU feasible
  - 50-70% memory reduction
  - Faster iteration cycles

Democratizing Access

- LoRA enables:
  - Researchers with limited compute budgets
  - Smaller organizations and startups
  - Educational institutions
- Reduces barrier to AI experimentation
- Aligns with responsible AI practices (see L14)

Key Takeaways

- Fine-tuning learns a weight update ΔW
- LoRA constrains ΔW to be low-rank
- Trainable parameters drop dramatically
- Memory and compute costs shrink
- LoRA is a principled, linear-algebraic solution

What to Remember

- LoRA is a structured parameterization technique
- Rooted in linear algebra (low-rank decomposition)
- Scales where full fine-tuning fails
- Enables sustainable, accessible AI adaptation

Original Paper

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). **LoRA: Low-Rank Adaptation of Large Language Models**. arXiv:2106.09685

Key finding: "We show that a very low-rank decomposition is sufficient for adaptation."

Further Reading

- QLoRA: Dettmers et al. (2023). arXiv:2305.14314
- Adapter methods survey: Pfeiffer et al. (2020)
- PEFT library: huggingface/peft