Natural Language Processing Course

Week 10: Fine-tuning and Prompt Engineering

Joerg R. Osterrieder
www.joergosterrieder.com

Week 10

Fine-tuning & Prompt Engineering

Making Models Do Exactly What You Want

Why GPT-3 Couldn't Write Good Legal Contracts

2021: A law firm tried using GPT-3...

Prompt: "Write a software licensing agreement"
Result: Generic, missed critical legal nuances, unusable[1]

The problem:

- GPT-3 knows about everything... but not deeply
- Trained on internet text, not legal documents
- Can't follow specific formatting requirements
- No domain expertise

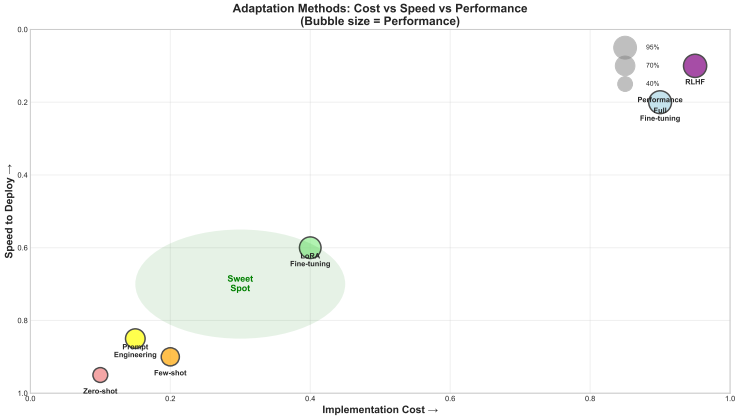
General models are jacks of all trades, masters of none

Two solutions emerged:

1. Fine-tune on legal documents
2. Engineer better prompts

[1] Composite example from multiple reported cases

From General to Specific: The Adaptation Challenge



The spectrum of adaptation:

- Zero-shot: Just ask (often fails)
- Few-shot: Show examples (better)
- Prompt engineering: Craft perfect instructions (good)
- Fine-tuning: Update model weights (best for specific tasks)

Fine-tuning and Prompting in Production (2024)

Fine-tuning Success:

- Bloomberg GPT: Finance[2]
- Med-PaLM 2: Medical diagnosis
- Codex: GitHub Copilot
- ChatGPT: From GPT-3.5 base
- Domain accuracy: 70% $\rightarrow$ 95%

Prompt Engineering:

- No training needed
- Instant deployment
- Version control friendly
- Cost: $0 training
- Performance: 60-80% of fine-tuning

Modern Methods:

- LoRA: 0.1% parameters[3]
- Prefix tuning: Frozen model
- Instruction tuning: Task generalization
- RLHF: Preference alignment
- Adapter layers: Modular skills

Business Impact:

- 10x faster deployment
- 90% less compute needed
- Domain expert performance
- Customization at scale

2024: Every company has a fine-tuned model or engineered prompts

[1] 50B parameters trained on financial data

[2] Low-Rank Adaptation - Hu et al. (2021)

Week 10: What You'll Master

By the end of this week, you will:

- **Understand** when to fine-tune vs prompt
- **Implement** efficient fine-tuning (LoRA)
- **Master** prompt engineering patterns
- **Design** instruction datasets
- **Build** domain-specific assistants

Core Insight: Small changes $\rightarrow$ Big behavioral shifts

Fine-tuning: Teaching New Tricks to Old Models

Traditional fine-tuning:

1. Start with pre-trained model (e.g., BERT)
2. Add task-specific head
3. Train on your data
4. Update ALL parameters

The problems:

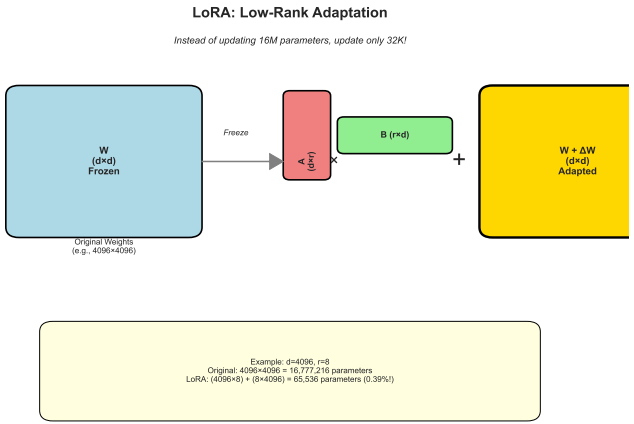
- Catastrophic forgetting
- Needs lots of GPU memory
- Slow and expensive
- One model per task

Example - Customer Support Bot:

- Base model: 7B parameters = 28GB
- Fine-tuning: 8 A100 GPUs, 2 days
- Cost: $500
- Result: 95% accuracy on support tickets

Full fine-tuning works but doesn't scale

LoRA: The Efficiency Revolution



Key insight: Most weight updates are low-rank!

- Instead of updating W (d×d), update A (d×r) and B (r×d)
- r ¡¡ d (typically r=8 while d=4096)
- Only 0.1% of parameters!

Implementing LoRA

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LoRALayer(nn.Module):
    def __init__(self, in_features, out_features, rank=8, alpha=16):
        """LoRA adapter for linear layers"""
        super().__init__()
        self.rank = rank
        self.alpha = alpha
        self.scaling = alpha / rank

        self.weight = nn.Parameter(torch.randn(out_features,
            in_features))
        self.weight.requires_grad = False

        self.lora_A = nn.Parameter(torch.randn(rank, in_features))
        self.lora_B = nn.Parameter(torch.zeros(out_features, rank))

        nn.init.normal_(self.lora_A, std=0.02)

    def forward(self, x):
        """Forward pass with LoRA adaptation"""
        out = F.linear(x, self.weight)

        lora_out = x @ self.lora_A.T @ self.lora_B.T

        return out + self.scaling * lora_out

    def merge_weights(self):
        """Merge LoRA weights for deployment"""
        self.weight.data += self.scaling * (self.lora_B @ self.
            lora_A)

class LoRAModel(nn.Module):
```

LoRA Benefits:

- Memory: 10,000x less
- Speed: 3x faster training
- Storage: 3MB vs 30GB
- Swappable: Multiple tasks

Hyperparameters:

- Rank: 4-64 (8 common)
- Alpha: Scaling factor
- Target: Usually attention
- Learning rate: 1e-4

Results Match Full FT:

- 0.1% parameters
- 99% performance
- No forgetting

Prompt Engineering: Programming with Natural Language

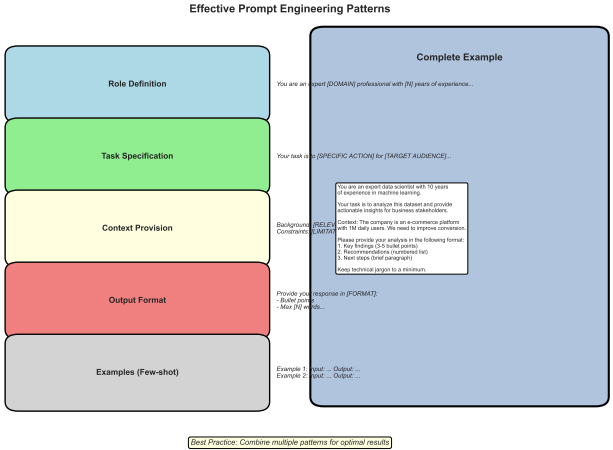
The prompt is your program:

Basic → Advanced:

1. Basic: "Summarize this text"
2. Better: "Summarize this text in 3 bullet points"
3. Good: "You are an expert editor. Summarize this text in exactly 3 bullet points, each under 20 words, capturing the key insights."
4. Excellent: "You are an expert editor at The Economist. Your task is to summarize the following text. Requirements: - Exactly 3 bullet points - Each under 20 words - Focus on actionable insights - Use active voice - Avoid jargon"

Specificity and structure dramatically improve output quality

Prompt Engineering Patterns That Work



Proven patterns:

- Role: "You are an expert..."
- Task: Clear, specific instructions

Advanced Prompt Engineering Techniques

```python
class PromptTemplate:
    """Advanced prompt engineering patterns"""

    @staticmethod
    def chain_of_thought(question):
        """CoT prompting for reasoning"""
        return f"""Q: {question}
Let's approach this step-by-step:
1) First, let's understand what we're asked...
2) Next, let's identify the key information...
3) Now, let's work through the solution...
4) Finally, let's verify our answer...

A: [Model completes the reasoning]"""

    @staticmethod
    def few_shot_with_reasoning(examples, query):
        """Few-shot with explanations"""
        prompt = "I'll solve these problems step by step.\n\n"

        for ex in examples:
            prompt += f"Problem: {ex['problem']}\n"
            prompt += f"Reasoning: {ex['reasoning']}\n"
            prompt += f"Answer: {ex['answer']}\n\n"

        prompt += f"Problem: {query}\n"
        prompt += "Reasoning: "
        return prompt

    @staticmethod
    def self_consistency(question, n_samples=5):
        """Multiple reasoning paths"""
        prompt = f"""Q: {question}

I'll solve this {n_samples} different ways to ensure accuracy:
```

Key Techniques:

- Chain-of-thought: +20% on reasoning[4]
- Self-consistency: Multiple paths
- Few-shot: Examples guide format
- Structured: JSON/XML output

Prompt Optimization:

- Test variations
- Measure performance
- Iterate systematically
- Version control prompts

Wei et al. (2022) "Chain-of-Thought"

Instruction Tuning: Teaching Models to Follow Instructions

The breakthrough: Train on instruction-following itself

Traditional fine-tuning:

- Task: Sentiment analysis
- Data: (text, label) pairs
- Model learns: One specific task

Instruction tuning:[5]

- Task: Follow any instruction
- Data: (instruction, input, output) triples
- Model learns: Generalize to new tasks!

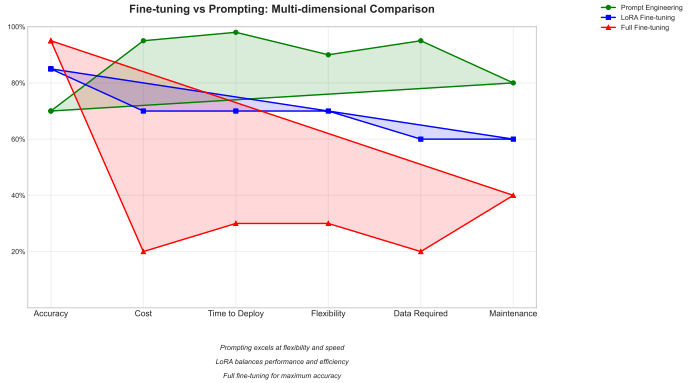
Example training data:

- Instruction: "Translate to French" → Input: "Hello" → Output: "Bonjour"
- Instruction: "Summarize" → Input: [article] → Output: [summary]
- Instruction: "Write code" → Input: [spec] → Output: [code]

Result: Zero-shot performance on completely new tasks!

[5] Wei et al. (2021) "FLAN"; Ouyang et al. (2022) "InstructGPT"

Fine-tuning vs Prompting: When to Use What



Key Insights

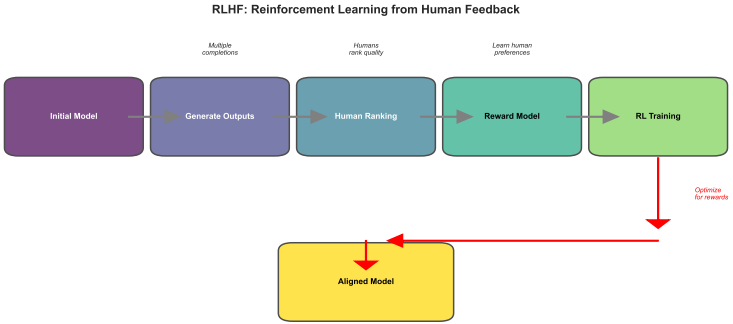
- Limited data: Use prompting
- Need 95%+ accuracy: Fine-tune
- Rapid iteration: Prompting
- Production scale: Fine-tune with LoRA

RLHF: Aligning Models with Human Preferences

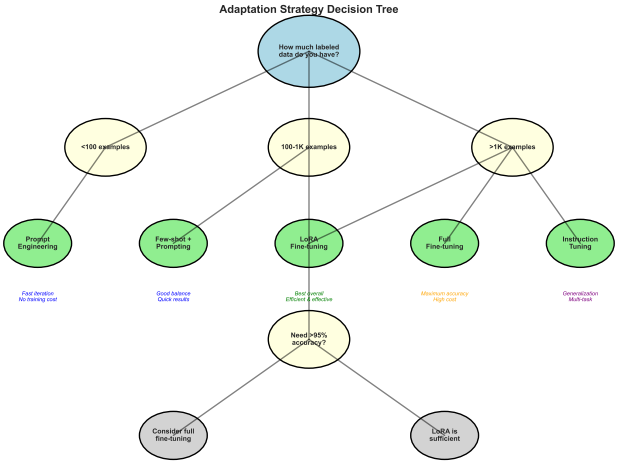
The problem: Models don't know what humans actually want

Reinforcement Learning from Human Feedback:[6]

1. Generate multiple outputs
2. Humans rank them
3. Train reward model
4. Use RL to optimize for human preferences



Choosing Your Adaptation Strategy



Quick decision guide:

- <100 examples: Prompt engineering
- 100-1K examples: Few-shot + prompting

Week 10 Exercise: Build a Domain Expert Assistant

Your Mission: Create specialized assistant using both approaches

Part 1: Prompt Engineering

- Choose domain (medical, legal, finance, etc.)
- Design systematic prompts
- Test role, CoT, few-shot patterns
- Measure accuracy on test cases
- Iterate to improve

Part 2: LoRA Fine-tuning

- Collect 1K domain examples
- Implement LoRA adapter
- Fine-tune base model
- Compare with prompting approach
- Measure speed/cost/quality

Part 3: Hybrid Approach

- Instruction-tune with domain data
- Design prompts for fine-tuned model
- Build evaluation framework
- Deploy and test with users

You'll discover: The sweet spot between effort and performance!

Key Takeaways: Specialization Strategies

What we learned:

- General models need adaptation
- Prompting: Fast, flexible, limited
- Fine-tuning: Powerful but expensive
- LoRA: Best of both worlds
- Instruction tuning: Generalization

The evolution:

Full fine-tuning $\rightarrow$ Prompting $\rightarrow$ LoRA $\rightarrow$ Instruction tuning $\rightarrow$ RLHF

Why it matters:

- Makes LLMs practical for business
- Enables rapid customization
- Reduces deployment costs 100x

Next week: Efficiency and Deployment
How do we run these massive models on phones and edge devices?

References and Further Reading

Foundational Papers:

- Hu et al. (2021). "LoRA: Low-Rank Adaptation of Large Language Models"
- Wei et al. (2021). "Finetuned Language Models Are Zero-Shot Learners"
- Ouyang et al. (2022). "Training language models to follow instructions"

Prompt Engineering:

- Liu et al. (2023). "Pre-train, Prompt, and Predict"
- White et al. (2023). "A Prompt Pattern Catalog"
- Zhou et al. (2023). "Large Language Models Are Human-Level Prompt Engineers"

Practical Resources:

- OpenAI Fine-tuning Guide
- Anthropic's Constitutional AI papers
- PEFT library (Hugging Face)