强化微调完全指南

陈星强

亿铸智能
中国杭州

2025 年 4 月 25 日

亿铸科技

内容概述

使用 DeepSeek-R1 和强化学习重塑 AI

强化学习简介

自我提升范式

- 强化学习引入了一种基于反馈驱动的 AI 训练机制
- 相比依赖标记样本，强化学习代理通过以下方式学习：
  - 探索：模型尝试多种策略或行动
  - 奖励：每个行动产生指导未来选择的奖励信号
- 更接近人类通过尝试、错误和反馈自然学习的方式

强化学习简介 - 继续

自我提升范式（继续）

- 为持续学习和更深层的推理能力打开了大门
- AI 不再是静态的实践—模型可以在部署后进化和适应

从静态到动态学习

- 传统方法依赖大量标记数据集进行记忆
- 强化学习将焦点转移到学习策略和推理模式
- “环境”可以是任何提供反馈信号指导改进的场景

DeepSeek-R1 的重要性

DeepSeek-R1：一个为推理而设计的模型

- 自适应奖励结构：多重奖励函数聚焦于准确性、效率性和创造性
- 迭代精化：基于奖励的反馈循环强调实践中最有效的方法
- 突破性能障碍：持续学习使其能够超越传统大语言模型
- 开源优势：与 OpenAI 闭源的 O1 不同，DeepSeek-R1 共享模型权重和训练方法

DeepSeek-R1: 技术实现

主动探索机制

- 使用主动探索而非被动学习
- 采用基于奖励的反馈循环
- 平衡探索与利用

推理模型民主化

- 开源设计便于自定义
- 发布模型权重保证透明度
- 社区驱动的创新

DeepSeek-R1 与传统模型对比

关键差异:

- 传统模型：静态训练，固定参数
- DeepSeek-R1：动态学习方法

DeepSeek-R1 与传统模型对比 (继续)

DeepSeek-R1 优势:

- 动态学习，响应需求变化
- 通过奖励学习，减少对标记数据的依赖
- 通过持续学习降低停滞风险

关键见解:

- “一次性训练完成”模式正在成为过去式

强化微调与监督微调对比

什么是强化微调 (RFT)?

将微调与强化学习相结合

- RFT 结合了预训练大语言模型与基于反馈的强化学习的优势
- 核心过程:
  1. 从具有通用知识的预训练模型开始
  2. 为目标指标定义奖励函数
  3. 使用强化学习技术进行迭代微调
- 使用最少的数据实现开源大语言模型的自定义
- 将通用模型转变为针对特定任务的强大推理模型

RFT 与 SSFT 对比

因素	RFT	SFT
Data Requirements	Minimal labeled data	Needs 1,000+ rows
Adaptability	Continuous improvement	Limited by labeled data
Exploration	Actively tries new strategies	Relies on fixed examples
Performance	Continual progress	Reaches plateau
Error Handling	Learns from mistakes	Repeats errors in data
Training Complexity	Higher (reward function)	Lower (just examples)

Table: Comparison of RFT and SFT approaches

When RFT Wins

RFT Excels with Scarce Data

- Removes need for labeled data, relies on objective correctness
- Outperforms SFT with small datasets (dozens of examples)
- Resists overfitting by learning robust strategies
- Best use cases:
  - Code Transpilation (e.g., Java to Python)
  - Game Strategy (Chess, Wordle)
  - Medical Diagnosis (learning from feedback at decision points)

Quantitative Performance Comparison

RFT vs SFT by Dataset Size

- 10 Examples: RFT improved base model by 18%, SFT showed minimal gains
- 50 Examples: RFT showed 42% improvement over baseline
- 100 Examples: RFT improvement jumped to 60%, 3x better than SFT
- 数据集越小，RFT 相比 SFT 的优势越明显

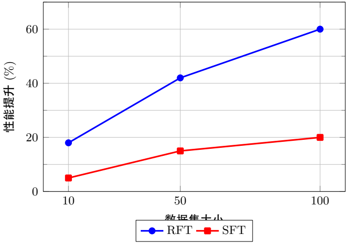


Figure: 性能提升与数据集大小的关系

何时使用 RFT 与 SFT

决策因素:

- 数据可用性: 数据有限时选择 RFT; 有大量标记数据时选择 SFT
- 任务复杂性: 对于有明确成功标准的任务，选择 RFT
- 性能目标: 需要持续改进时选择 RFT; 需要稳定结果时选择 SFT
- 可验证性: 当结果可以客观衡量时，RFT 表现更佳
- 资源限制: SFT 初期实施更简单，但需要更多数据

实际应用:

- 编程助手: RFT 训练模型编写可编译并通过测试的代码
- 数据分析: RFT 改进生成准确结果的查询能力
- 推理任务: RFT 增强逐步解决问题的能力

使用 Turbo LoRA 加速推理模型

推理模型的挑战

处理量问题:

- 推理模型推动了 AI 的解决问题能力
- 但高级推理需要付出代价：处理速度缓慢
- 推理模型通过生成大量标记来“思考”
- 多重中间计算增加了处理时间
- 生产部署需要解决这些挑战

实际性能瓶颈

- 推理模型比类似的非推理模型慢 2-3 倍
- 更高的延迟显著影响用户体验和成本效率
- 传统加速方法常常会损害推理质量
- 需要特殊解决方案，在提高速度的同时保持推理能力

LoRA 和 Turbo LoRA

LoRA: 低秩适应

- 使用小型可训练参数集微调大型模型
- 保留原始权重，维持预训练知识
- 显著减少微调所需的内存要求
- 支持模型高效适应专业任务

Turbo LoRA: 推理速度提升 2-4 倍

主要特点

- 使用推测解码以及专有优化
- 并行预测多个标记，然后验证它们
- 在更快生成文本的同时保持输出质量

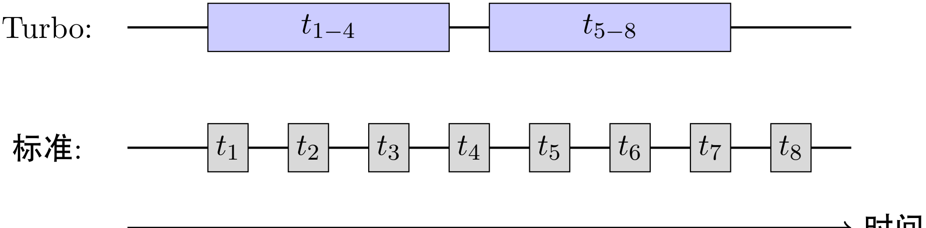
Turbo LoRA 如何工作

技术实现

1. 小型、快速的“推测器”模型并行预测多个标记
2. 主模型验证预测的标记
3. 正确的标记立即被使用
4. 只有不正确的标记需要重新计算

实际效益

- 最终输出质量零差异
- 应用于 DeepSeek-R1-distill-qwen-32b 模型
- 实现了 2-3 倍的处理量提升
- 可应用于任何推理模型



Turbo 对大型推理模型的益处

实现实时 AI 可行性

- 2-3 倍的速度提升使推理模型适用于:
  - AI 驱动的客户支持

Turbo 对大型推理模型的益处（续）

实现实时 AI 可行性（续）

- 2-3 倍的速度提升使推理模型适用于:
  - 开发者的 AI 副驾驶
  - 医疗保健 AI 助手
- 降低 GPU 成本：相同工作量需要更少的 GPU

大型推理模型 Turbo 的优势（续）

实际影响

- 使几乎任何组织都能负担推理模型的运行
- 典型推理设置：所需 GPU 容量减少 2-3 倍
- 成本节约随部署规模扩大而增加

实施方法

- 可使用更小、更高效的模型而不牺牲性能力
- 详细实施教程：https://predibase.com/blog/turbo-lora

Tutorial: Using RFT to Write CUDA Kernels

Why GPU Code Generation is Hard

Challenges of GPU Coding:

- Parallel Architecture: Many cores
- Memory Hierarchy: Multiple levels
- Thread Synchronization: Complex
- Small errors $\rightarrow$ large consequences

Traditional Approaches Fail:

- Few high-quality CUDA examples
- SFT needs thousands of pairs
- Many edge cases to handle
- Subtle errors cause failures

Why RFT is Perfect for Code Generation

Key Advantages:

- No large dataset needed
- Code is verifiable

Implementation:

- Started with 13 examples
- RL explores intelligently

Setting Up the CUDA Task: Minimal Dataset

Dataset Composition

- Each example contained:
  - A PyTorch function (e.g., matrix multiply or activation function)
  - A set of test cases to verify correctness
- Example functions included matrix operations, activations, and element-wise operations
- Test cases covered edge cases, different sizes, and boundary conditions

PyTorch to Triton Example

- PyTorch function:

  def add(x, y): return x + y

Setting Up the CUDA Task: Triton Implementation

Target Triton Kernel

```
@triton.jit
def add_kernel(x_ptr, y_ptr, output_ptr, n_elements):
    pid = tl.program_id(0)
    block_size = 128
    offsets = pid * block_size + tl.arange(0, block_size)
```

Setting Up the CUDA Task: Triton Implementation (cont.)

Target Triton Kernel (cont.)

```
mask = offsets < n_elements
x = tl.load(x_ptr + offsets, mask=mask)
y = tl.load(y_ptr + offsets, mask=mask)
output = x + y
tl.store(output_ptr + offsets, output, mask=mask)
```

Defining Rewards for Code Generation

Multi-Level Reward Structure (1)

Reward 1: Formatting (0.1-0.3)

- Code structure, imports, tags
- Partial credit for good Triton semantics
- Reasonable variable names and code organization

Multi-Level Reward Structure (2)

Reward 2: Compilation (0.3-0.6)

- Code that executes without errors
- No runtime exceptions or syntax errors
- Properly imports required dependencies

Defining Rewards for Code Generation (cont.)

Multi-Level Reward Structure (3)

Reward 3: Correctness (0.6-1.0)

- Output matches PyTorch function on test inputs
- Anti-reward-hacking measures (checking for hardcoded outputs)
- Proper handling of edge cases and different input shapes

Benefits of This Reward Structure

- Provides clear progression path for the model
- Allows partial credit for partial solutions
- Mirrors how humans learn to code

Example Implementation of Reward Function

Reward Function Logic - Part 1

1. Formatting check:
   - If code has proper Triton syntax $\rightarrow$ reward $= 0.2$
2. Compilation check:
   - If code compiles successfully $\rightarrow$ reward $= 0.5$
   - If compilation fails $\rightarrow$ return current reward

Example Implementation of Reward Function (cont.)

Reward Function Logic - Part 2

3. Correctness check:
   - For each test case, compare PyTorch and Triton results
   - If outputs match $\rightarrow$ reward $= 1.0$
   - Otherwise $\rightarrow$ reward $= 0.6$

Key Elements

- Multi-stage evaluation process
- Each stage builds on the previous
- Partial rewards for partial successes
- Deterministic verification through test cases
- Numerically scaled from 0.0 to 1.0
- Multiple test cases ensure robustness

Training Loop and Results

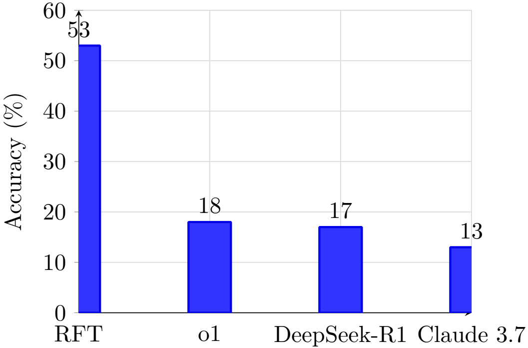
How GRPO (Gradient-based Reward Policy Optimization) Works:

- Generate: Multiple completions per prompt using sampling
- Evaluate: Run reward checks on each completion
- Update: Compute advantages and backpropagate signals
- Repeat: Model refines strategy to maximize rewards

Results:

- 53% accuracy on held-out examples after 5,000 steps
- 3x higher correctness rate than OpenAI o1 and DeepSeek-R1
- 4x better performance than Claude 3.7 Sonnet

Performance Comparison: RFT vs. Leading Models



RFT Training Progression

- Starting point: 5% accuracy
- Early training (1,000 steps): 22%
- Mid-training (3,000 steps): 41%
- Final result: 53% accuracy
- Model developed generalizable patterns beyond the limited training examples

Performance Comparison with Leading Models

Results:

- 53% accuracy on held-out examples after 5,000 steps
- 3x higher correctness rate than OpenAI o1 and DeepSeek-R1
- 4x better performance than Claude 3.7 Sonnet

Key Success Factors

- Reward Design: Multi-level rewards provided clear learning signals
- Test Variety: Diverse test cases prevented overfitting
- Anti-Reward Hacking: Prevented the model from simply memorizing outputs

Practical RFT Implementation with Unsloth

Hands-On RFT Workflow with Unsloth

What is Unsloth?

- Open-source library for efficient LLM fine-tuning
- Works on limited hardware (3GB+ VRAM)
- Supports QLoRA, LoRA, RLHF, GRPO
- 2-4x faster than standard methods

Development Workflow

1. Setup: Install dependencies
2. Model: Choose base model
3. Data: Format for training
4. Configure: Set parameters
5. Train: Run fine-tuning
6. Deploy: Save and serve

Step 1: Environment Setup

Install Unsloth

```
# Basic installation
pip install unsloth

# For specific CUDA versions
# pip install unsloth[cu118]  # CUDA 11.8
# pip install unsloth[cu121]  # CUDA 12.1
^^I^^I
```

Step 1: Required Libraries

```python
# Import necessary libraries
import torch
from unsloth import FastLanguageModel
from datasets import load_dataset
from transformers import TrainingArguments
^^I
```

Performance Optimizations

- CPU offloading for devices with limited VRAM
- Flash Attention 2 for faster training (on supported GPUs)
- Gradient checkpointing to trade compute for memory

Step 2: Selecting Model & Method

Load Base Model

```
max_seq_length = 2048
model, tokenizer =
↪ FastLanguageModel.from_pretrained(
    model_name =
    ↪ "unsloth/llama-3.1-8b-bnb-4bit",
    max_seq_length = max_seq_length,
    load_in_4bit = True)
~~I~~I~~I
```

Choose Fine-Tuning Method

- QLoRA: 4-bit quantization with LoRA (least resources)
- LoRA: Low-Rank Adaptation (balance of quality/resources)
- GRPO: For DeepSeek-style reinforcement learning
- Full Finetuning: For maximum quality on high-end hardware

Step 2: Configuring LoRA for RFT

RFT-Specific Configuration

- Target key-value attention layers for most efficient adaptation
- Use rank 16-32 for complex tasks like reinforcement learning
- Add dropout to prevent overfitting to reward signals
- Enable gradient checkpointing to save memory during training

Code Implementation

```
model = FastLanguageModel.get_peft_model(
    model,
    r = 16,  # LoRA rank
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj"],
    alpha = 16,  # LoRA alpha
    dropout = 0.05,  # Add regularization
    use_gradient_checkpointing = True)
^^I^^I
```

Step 3: Dataset Preparation

Dataset Format for RFT

- Format datasets with input and expected output
- Include reward signals or success criteria
- Ensure diverse examples across different scenarios

```
dataset = [
    {"input": "Solve: $2x + 3 = 7$",
     "output": "To solve for x:\n$2x + 3 = 7$\n$2x = 4$\n$x = 2$"},
    {"input": "What is the capital of France?",
     "output": "The capital of France is Paris."},
    # ...more examples
]
^^I
```

Step 3: Dataset Formatting for Training

Applying the Proper Template

- Use the model's specific chat template format
- Ensure special tokens are correctly applied
- Set appropriate sequence length for your use case

```
formatted_dataset = FastLanguageModel.format_dataset(
    dataset,
    tokenizer,
    max_seq_length,
    add_special_tokens = True,
    template = "<s>[INST] {input} [/INST] {output} </s>"
)
^^I
```

Step 4: Configure Training Parameters

Training Arguments

```
training_args = TrainingArguments(
    output_dir = "./results",
    num_train_epochs = 3,
    per_device_train_batch_size = 4,
    gradient_accumulation_steps = 2,
    learning_rate = 2e-4,
    weight_decay = 0.01,
    max_grad_norm = 0.3,
    logging_steps = 10,
    save_total_limit = 3,
)
``I
```

Step 4: Hyperparameter Selection for RFT

Key Hyperparameters

- LoRA Rank (r): Between 8-32; higher gives more expressive power
- LoRA Alpha: Usually same as rank
- Learning Rate: 2e-4 to 5e-4 for QLoRA
- Training Epochs: 2-5 for small datasets
- Dropout: 0.05-0.1 for regularization

Avoid Overfitting in RFT

- Increase dropout if showing signs of overfitting
- Implement early stopping by monitoring loss
- Use a validation set to evaluate generalization
- Add weight decay for regularization

Step 5: Training Execution

Execute Training

```python
from trl import SFTTrainer

trainer = SFTTrainer(
    model = model,
    train_dataset = formatted_dataset,
    args = training_args,
    tokenizer = tokenizer,
    packing = False,
    dataset_text_field = "text",
)
^^I^^I
```

Step 5: Training Process and Monitoring

Run Training

```
# Start training
trainer.train()
~~~I~~~I
```

Monitor Key Metrics

- Training Loss: Should steadily decrease but not plateau too quickly
- Validation Loss: Monitor for signs of overfitting (uptick in validation loss)
- Learning Rate: Record effect of LR variations on performance
- GPU Utilization: Verify efficient resource usage

Step 6: Evaluation

Evaluate Model Performance

- Load the fine-tuned model for inference
- Create structured test prompts
- Compare with baseline model on same inputs
- Test diverse scenarios and edge cases
- Measure inference speed and resource usage

```
# Example test inference
prompt = "Solve: 3x + 5 = 20"
response = model.generate_text(prompt)
# Result: "To solve: 3x + 5 = 20
Subtract 5: 3x = 15
Divide by 3: x = 5"
^^I
```

Step 6: Deployment

Save and Deploy Model

- Save trained model and tokenizer
- Optimize for inference (e.g., ONNX conversion)
- Configure deployment environment
- Set up monitoring and logging

```
# Save the fine-tuned model
model.save_pretrained("./my-rft-model")
tokenizer.save_pretrained("./my-rft-model")
︵︵I
```

Step 6: Continuous Improvement

Ongoing Refinement

- Collect user feedback and model outputs in production
- Update reward functions based on real-world performance
- Expand training dataset with edge cases discovered in deployment
- Periodically retrain to incorporate improvements

Long-term Maintenance

- Monitor for performance degradation over time
- Keep track of new best practices in RFT
- Evaluate benefits of updating to newer base models

Ready-to-Use Unsloth Notebooks

Available Implementation Resources

- Unsloth provides ready-to-use notebooks for various models and tasks
- Accessible via Google Colab or Kaggle (free GPU resources)
- Includes both fine-tuning and GRPO (RFT) implementations

Popular Models

- Llama 3.1 (8B)
- Phi-4 (14B)
- Mistral (7B, 22B)
- Qwen 2.5 (3B, 14B)
- Gemma 2 (2B, 9B)

Specialized Variants

- Qwen2.5-Coder (14B)
- CodeGemma (7B)
- Llama 3.2 Vision
- Qwen2-VL (7B)
- Phi-3 Medium

Notebooks available at: https://docs.unsloth.ai/get-started/unsloth-notebooks

Dataset Construction for RFT

Getting Started

- Identify the purpose of your dataset: chat dialogues, structured tasks, or domain-specific data
- Define desired output style: JSON, HTML, text, code or specific languages
- Determine data sources: Hugging Face datasets, Wikipedia, or synthetic data

Common Data Formats

- Text-only format
- Instruction-Input-Output
- ShareGPT format (multi-turn)
- ChatML (OpenAI style)

Dataset Requirements

- Minimum: 100 examples
- Optimal: 1,000+ examples
- Quality over quantity
- Can combine multiple datasets

RFT Implementation: Common Pitfalls (1)

- Problem: Model produces generic or refuses to generate responses
  Solution: Increase LoRA rank and alpha; ensure diverse training data
- Problem: Catastrophic forgetting (model loses pre-trained capabilities)
  Solution: Lower learning rate; add regularization
- Problem: High training loss that doesn't converge
  Solution: Check dataset formatting; reduce sequence length

RFT 实施：常见问题 (2)

- 问题：训练期间内存不足错误
  解决方案：启用梯度检查点；减小批量大小
- 问题：模型生成幻觉
  解决方案：实施惩罚虚构内容的奖励函数
- 问题：使用奖励信号时训练不稳定
  解决方案：归一化奖励；使用带有适当裁剪的 PPO

结论

关键要点

RFT 的力量:

- 自我提升: 模型超越静态方法
- 数据效率: 使用更少数据超越 SFT
- 速度: Turbo LoRA 将处理量提高 2-4 倍
- 实际应用: 超越学术领域

范式转变:

- 传统方式: 大数据 → 静态模型
- RFT: 最小数据 + 奖励 → 成长
- 新循环: 持续改进

实施策略 - 核心组件

所需核心组件

1. 基础模型（开源 LLM）
2. 奖励函数定义
3. 提示数据集（可以很小）
4. 强化学习算法（RLHF、GRPO 等）
5. 评估框架

实施策略 - 最佳实践

最佳实践

- 从小型、清晰的奖励函数开始
- 构建全面的验证测试
- 实施反奖励黑客措施
- 监控演发行为
- 逐步增加复杂性

RFT 的后续步骤

开始使用:

- 强化微调标志着 LLM 发展的重大飞跃
- 通过奖励信号而非标记示例进行训练
- 端到端平台使开发者能够轻松使用这种方法
- 专注于创新而非基础设施复杂性

RFT 应用的成熟领域

- 专业编程: 领域特定的代码生成（嵌入式系统、高性能计算）
- 科学研究: 提出并验证假设的模型
- 推理任务: 复杂的逻辑和数学问题解决
- 教育: 能够理解学生知识空白的自适应辅导系统

Thank You!

Questions?