亿铸科技

The Complete Guide to Reinforcement Fine-Tuning

Chen Xingqiang

Yizhu Intelligent Ltd
Hangzhou, China

April 25, 2025

Overview

Reinventing AI with DeepSeek-R1 and Reinforcement Learning

Introduction to Reinforcement Learning

The Self-Improvement Paradigm

- Reinforcement learning introduces a feedback-driven mechanism for training AI
- Rather than relying on labeled examples, RL agents learn by:
  - **Exploration**: The model attempts multiple strategies or actions
  - **Reward**: Each action yields reward signals that guide future choices
- More closely aligns with how humans naturally learn through trial, error, and feedback

Introduction to Reinforcement Learning - Continued

The Self-Improvement Paradigm (cont.)

- Opens the door to continual learning and deeper reasoning capabilities
- AI is no longer a static endeavor—models can evolve and adapt after deployment

From Static to Dynamic Learning

- Traditional approaches rely on enormous labeled datasets for memorization
- RL shifts focus to learning strategies and reasoning patterns
- The "environment" can be any context where feedback signals guide improvement

The Significance of DeepSeek-R1

DeepSeek-R1: A Model Designed to Reason

- **Adaptive Reward Structures**: Multiple reward functions focusing on accuracy, efficiency, and creativity
- **Iterative Refinement**: Reward-based feedback loops emphasizing what works best in practice
- **Breaking Performance Barriers**: Continuous learning allows it to outperform traditional LLMs
- **Open-Source Advantage**: Unlike OpenAI's closed-source O1, DeepSeek-R1 shares weights and training approaches

DeepSeek-R1: Technical Implementation

Active Exploration Mechanisms

- Uses **active exploration** instead of passive learning
- Employs **reward-based feedback loops**
- Balances exploration with exploitation

Democratizing Reasoning Models

- **Open-source design** for customization
- **Released weights** for transparency
- **Community-driven** innovation

DeepSeek-R1 vs. Traditional Models

Key Differences:

- **Traditional:** Static training, frozen parameters
- **DeepSeek-R1:** Dynamic learning approach

DeepSeek-R1 vs. Traditional Models (cont.)

DeepSeek-R1 Advantages:

- Learns dynamically, responding to changes in requirements
- Requires fewer labeled data points by learning from rewards
- Minimizes the risk of stagnation through continuous learning

Key Insight:

- "One-and-done" training becoming a relic of the past

Reinforcement Fine-Tuning vs. Supervised Fine-Tuning

What is Reinforcement Fine-Tuning (RFT)?

Combining Fine-Tuning with RL

- RFT combines strengths of pre-trained LLMs with feedback-driven RL
- Core process:
  1. Start with a pre-trained model with general knowledge
  2. Define a reward function for target metrics
  3. Iteratively fine-tune using RL techniques
- Enables customization of open-source LLMs with minimal data
- Turns general models into powerful reasoning models for specific tasks

RFT vs. SFT Comparison

Factor	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

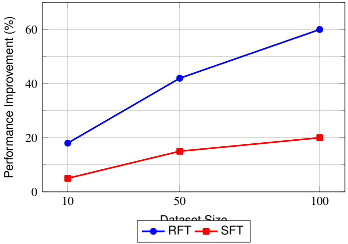
RFT Excels with Scarce Data

- 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
- The smaller the dataset, the greater RFT's advantage over SFT



When to Use RFT vs. SFT

Decision Factors:

- **Data Availability**: RFT for limited data; SFT for abundant labeled data
- **Task Complexity**: RFT for tasks with clear success criteria
- **Performance Goals**: RFT for continuous improvement; SFT for stable results
- **Verifiability**: RFT excels when outcomes can be objectively measured
- **Resource Constraints**: SFT simpler to implement initially but requires more data

Real-World Applications:

- **Coding Assistants**: RFT trains models to write code that compiles and passes tests
- **Data Analysis**: RFT improves query generation that produces correct results
- **Reasoning Tasks**: RFT enhances step-by-step problem-solving capabilities

Accelerating Reasoning Models with Turbo LoRA

The Challenge with Reasoning Models

Throughput Issues:

- Reasoning models push AI’s problem-solving capabilities
- But advanced reasoning comes at a cost: slow throughput
- Reasoning models "think" by generating many tokens
- Multiple intermediate computations increase processing time
- Production deployment requires addressing these challenges

Real-World Performance Bottlenecks

- Reasoning models can be 2-3x slower than comparable non-reasoning models
- Higher latency significantly impacts user experience and cost-efficiency
- Traditional acceleration methods often compromise reasoning quality

LoRA and Turbo LoRA

LoRA: Low-Rank Adaptation

- Fine-tunes large models with small set of trainable parameters
- Preserves original weights, maintaining pre-trained knowledge
- Dramatically reduces memory requirements for fine-tuning
- Enables efficient adaptation of models for specialized tasks

Turbo LoRA: 2-4x Faster Reasoning

Key Features

- Uses speculative decoding along with proprietary optimizations
- Predicts multiple tokens in parallel, then verifies them
- Maintains output quality while generating text faster

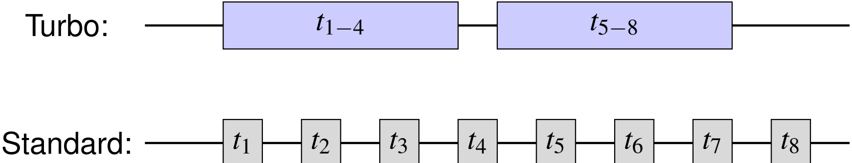
How Turbo LoRA Works

Technical Implementation

1. Small, fast "speculator" model predicts several tokens in parallel
2. Main model verifies predicted tokens
3. Correct tokens are instantly used
4. Only incorrect tokens need recalculation

Practical Benefits

- Zero difference in final output quality
- Applied to DeepSeek-R1-distill-qwen-32b model
- Demonstrated 2-3x throughput improvement
- Can be applied to any reasoning model



Benefits of Turbo for Large Reasoning Models

Making Real-Time AI Feasible

- 2-3x speedup makes reasoning models viable for:
  - AI-powered customer support

Benefits of Turbo for Large Reasoning Models (cont.)

Making Real-Time AI Feasible (cont.)

- 2-3x speedup makes reasoning models viable for:
  - AI copilots for developers
  - Healthcare AI assistants
- Lower GPU costs: fewer GPUs needed for the same workload

Benefits of Turbo for Large Reasoning Models (cont.)

Real-World Impact

- Makes inference of reasoning models feasible for almost any organization
- Typical inference setup: 2-3x reduction in required GPU capacity
- Cost savings scale with deployment size - larger deployments save more

Implementation

- Enables use of smaller, more efficient models without sacrificing capability
- For detailed implementation tutorial: `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

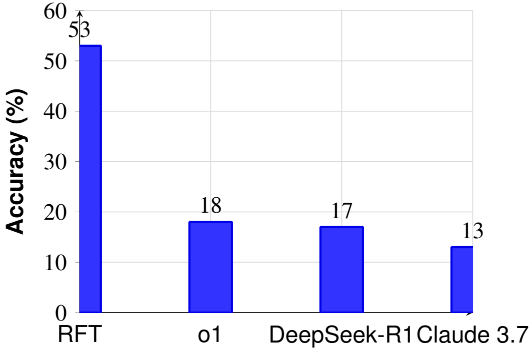
How GRPO (Gradient-based Reward Policy Optimization) Works:

- **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

亿铸科技

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
```

Step 1: Required Libraries

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

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

```python
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)
```

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

```python
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)
```

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
]
```

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>"
)
```

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,
)
```

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

亿铸科技

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",
)
```

Step 5: Training Process and Monitoring

Run Training

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

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

亿铸科技

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"
```

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")
```

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

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 Implementation: Common Pitfalls (2)

- **Problem**: Out of memory errors during training
  Solution: Enable gradient checkpointing; reduce batch size
- **Problem**: Model generates hallucinations
  Solution: Implement reward functions that penalize fabrications
- **Problem**: Training is unstable with reward signals
  Solution: Normalize rewards; use PPO with appropriate clipping

Conclusion

Key Takeaways

The Power of Reinforcement Fine-Tuning:

- **Self-Improvement is Real**: Models can surpass static approaches through iterative rewards
- **RFT Shines With Little Data**: Outperforms SFT when labeled data is scarce
- **Reasoning Doesn't Need to be Slow**: Turbo LoRA increases throughput by 2-4x
- **Real-World Feasibility**: Moving beyond academic exercises to practical applications

Paradigm Shift in AI Development

- Traditional LLM training: massive datasets $\rightarrow$ static deployment
- RFT approach: minimal datasets + reward functions $\rightarrow$ continuous improvement
- Fundamentally changes how we think about model improvement cycles
- Smaller teams can now compete with resource-rich organizations

Implementation Strategy - Core Components

Core Components Needed

1. Base model (open-source LLM)
2. Reward function definition
3. Prompt dataset (can be small)
4. RL algorithm (RLHF, GRPO, etc.)
5. Evaluation framework

Implementation Strategy - Best Practices

Best Practices

- Start with small, clear reward functions
- Build comprehensive validation tests
- Implement anti-reward-hacking measures
- Monitor for emergent behaviors
- Gradually increase complexity

Next Steps with RFT

Getting Started:

- Reinforcement fine-tuning marks a major leap in LLM development
- Training through reward signals rather than labeled examples
- End-to-end platforms make this approach accessible to developers
- Focus on innovation rather than infrastructure complexities

Areas Ripe for RFT Application

- **Specialized Coding**: Domain-specific code generation (embedded systems, high-performance computing)
- **Scientific Research**: Models that propose and validate hypotheses
- **Reasoning Tasks**: Complex logical and mathematical problem-solving
- **Education**: Adaptive tutoring systems that understand student gaps

Thank You!

Questions?