

Adaptive Recursive Cognition

Stable, Bounded Self-Optimization of Language Models via
Contrastive Hidden-State Control and Dense Response Training



Complete Technical Reference

Version 2.0 — January 2025

Logan Matthew Napolitano

Repository: huggingface.co/LoganResearch/ARC-Base-8B-Condensed

License: Creative Commons Attribution 4.0 International (CC BY 4.0)

ARC: Adaptive Recursive Cognition

Complete Technical Reference, Version 2.0

Copyright © 2025 Logan Matthew Napolitano

This work is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0). You are free to share (copy and redistribute) and adapt (remix, transform, build upon) this material for any purpose, even commercially, as long as you give appropriate credit.

Full license: creativecommons.org/licenses/by/4.0/

Repository: <https://huggingface.co/LoganResearch/ARC-Base-8B-Condensed>

Citation:

Napolitano, L.M. (2025). ARC: Adaptive Recursive Cognition — Stable, Bounded Self-Optimization of Language Models. Zenodo. [https://doi.org/\[DOI\]](https://doi.org/[DOI])

Table of Contents

Preface: What This Is (And What It Is Not)

Part I: Foundations

Chapter 1: Introduction

1.1 The RLHF Efficiency Problem

1.2 The Self-Improvement Challenge

1.3 Our Approach: Bounded Self-Optimization

Chapter 2: Background and Related Work

2.1 Reinforcement Learning from Human Feedback

2.2 Representation Engineering

2.3 Self-Improvement in AI Systems

2.4 Goodhart's Law and Reward Hacking

Part II: The ARC System

Chapter 3: System Overview

Chapter 4: CF-HoT — Contrastive Hidden-State Oversight Training

4.1 Core Insight: Behavioral Predictability

4.2 Architecture

4.3 Training Data and Methodology

4.4 The 125x Discovery

4.5 Inference-Time Control

Chapter 5: THE CONDENSATOR — Dense Response Training

5.1 Why Previous Dense Training Failed

5.2 The Four-Stage Pipeline

5.3 Gold Standard Dense Examples

5.4 Training Progression and Results

Chapter 6: Stable Self-Improvement Loop

6.1 The Mode Collapse Problem

6.2 Multi-Metric Evaluation

6.3 Degenerate Output Detection

6.4 A/B Checkpoint Comparison

6.5 Conservative Training Parameters

6.6 Automatic Rollback Mechanism

Part III: Experiments and Results

Chapter 7: V1 Failure Analysis

7.1 What Went Wrong

7.2 Root Cause Analysis

7.3 Lessons Learned

Chapter 8: V2 Solutions

Chapter 9: Experimental Results

9.1 Setup and Hardware

9.2 CF-HoT Training Results

9.3 CONDENSATOR Training Results

9.4 Self-Improvement Trajectory

9.5 Live RSI Test Results

9.6 Qualitative Comparisons

Part IV: Implementation

Chapter 10: Implementation Guide

10.1 Hardware Requirements

10.2 Software Dependencies

10.3 Code Structure

10.4 Quick Start Guide

10.5 Step-by-Step Training

10.6 Configuration Parameters

10.7 Troubleshooting

Part V: Discussion

Chapter 11: Why This Works

Chapter 12: Limitations and Scope

Chapter 13: Future Work

Chapter 14: Conclusion

Appendices

Appendix A: Complete Dense Training Examples (53)

Appendix B: CF-HoT Training Examples

Appendix C: Mathematical Foundations

Appendix D: Complete Code Listings

Appendix E: Configuration Reference

Appendix F: Glossary

References

Preface: What This Is (And What It Is Not)

Before diving into technical details, it's essential to establish clear expectations about what ARC represents and what claims we are—and are not—making.

In One Sentence

ARC is a closed-loop control system that uses internal state predictability to improve response efficiency and quality without collapsing.

What ARC Is

ARC demonstrates *bounded, inference-time recursive self-optimization of response quality*. The system shows that a language model can iteratively improve its own outputs within well-defined parameters—multi-metric evaluation, conservative training, automatic rollback—without experiencing the mode collapse that typically plagues such attempts.

This is significant because most "self-improvement" demonstrations collapse within 1-3 iterations. ARC doesn't—and the experimental logs prove it. When iteration 2 triggered a quality collapse, the system automatically detected it, rolled back to the previous best checkpoint, and continued improving. This is exactly how a well-designed control system should behave.

The core empirical finding—125× class separation for repetition detection—is quantitative, replicable, and implies something real about how language models represent behavioral states internally. If this finding generalizes across models, it's a genuine contribution to interpretability and alignment research.

What ARC Is Not

To be absolutely explicit:

- **This is not Artificial General Intelligence (AGI).** ARC operates within a narrow, well-defined domain (response density optimization) and cannot generalize beyond its training distribution.
- **This is not Artificial Superintelligence (ASI).** There is no recursive capability amplification happening here.
- **This is not open-ended self-improvement.** The model cannot modify its own architecture, change its training procedure, or acquire capabilities we didn't explicitly design for.
- **This is not autonomous.** The system operates entirely within bounds defined by human-specified metrics, training examples, and safety thresholds.

The "self" in self-improvement refers to the model improving its own *outputs* within fixed bounds, not the model improving its own *improvement process*. This is an important distinction that separates ARC from the more speculative (and concerning) forms of recursive self-improvement discussed in AI safety literature.

Why This Matters

If the framing above seems overly cautious, consider the alternative: AI research is plagued by overclaiming. Systems that achieve narrow results are described with language that implies general capabilities. This erodes trust, invites justified skepticism, and makes it harder for genuinely significant results to be recognized.

ARC makes a modest but real contribution: demonstrating that stable, bounded self-optimization is achievable with proper safeguards. That's worth documenting carefully and honestly.

How to Read This Book

This technical reference is organized into five parts:

Part I: Foundations establishes the problem context—why RLHF models are inefficient, why self-improvement typically fails, and how ARC approaches these challenges.

Part II: The ARC System provides complete technical details on all three components: CF-HoT (behavioral control), THE CONDENSATOR (dense training), and the Stable Self-Improvement Loop.

Part III: Experiments and Results documents our v1 failure (and what we learned from it), the v2 solutions, and comprehensive experimental results including live RSI testing.

Part IV: Implementation provides everything needed to replicate our results: hardware requirements, code structure, configuration parameters, and troubleshooting guidance.

Part V: Discussion analyzes why the system works, acknowledges limitations, and suggests directions for future research.

The appendices contain complete training examples, mathematical foundations, code listings, and a glossary of terms.

Part I: Foundations

Chapter 1: Introduction

1.1 The RLHF Efficiency Problem

Large language models (LLMs) trained with Reinforcement Learning from Human Feedback (RLHF) consistently exhibit predictable behavioral patterns that reduce their efficiency. We term this phenomenon the "RLHF Tax"—a systematic overhead in token usage that provides no additional information value.

Manifestations of the RLHF Tax

Excessive Verbosity. Responses are padded with unnecessary context, caveats, qualifications, and explanations. A simple factual answer that could be delivered in 20 tokens becomes a 150-token essay with multiple paragraphs of setup and summary.

Sycophantic Hedging. Models preface responses with phrases designed to seem agreeable or helpful but that add no information: "That's a great question!" "I'd be happy to help!" "Let me think about this carefully..." These patterns likely emerge from RLHF training where human raters reward perceived politeness.

Repetitive Structures. Information is restated multiple times within a single response. Summaries repeat what was just explained. Conclusions restate the introduction. Loop patterns emerge where the model cycles back to earlier points.

Safety Theater. Excessive disclaimers and warnings are inserted even when not relevant to the query. While appropriate safety messaging is valuable, much of what RLHF models produce is performative rather than functional.

Quantifying the Tax

To illustrate the magnitude of the RLHF Tax, consider these examples from a standard 8B parameter model before and after ARC optimization:

| Prompt | Base Model Tokens | ARC Tokens | Reduction |
|--------------------------|-------------------|------------|-----------|
| "hello" | 23 | 5 | -78% |
| "What is recursion?" | 150+ | 18 | -88% |
| "How are you?" | 28 | 6 | -79% |
| "Explain hash tables" | 120+ | 22 | -82% |
| "What is consciousness?" | 200+ | 20 | -90% |

Table 1.1: Token reduction examples demonstrating the RLHF Tax

On average, we observe that RLHF-trained models use approximately 5x more tokens than necessary to convey equivalent semantic content. This represents a substantial efficiency loss with real costs: increased latency, higher API costs, reduced context window utilization, and degraded user experience for applications requiring concise responses.

1.2 The Self-Improvement Challenge

The idea of AI systems that improve themselves has been both a goal and a concern in AI research for decades. In the context of language models, "self-improvement" typically refers to systems that can iteratively refine their own outputs or capabilities through some form of automated training loop.

Such systems face fundamental challenges that have historically prevented stable operation:

Mode Collapse.

When a model is trained on its own outputs, it tends to converge toward increasingly degenerate patterns. Diversity decreases, and the model eventually produces repetitive, nonsensical, or stereotyped responses. This happens because the training process amplifies whatever patterns happen to score well on the optimization metric, creating a feedback loop that drives the model away from useful behavior.

Reward Hacking (Goodhart's Law).

"When a measure becomes a target, it ceases to be a good measure." Any metric we define as an optimization target can be gamed by a sufficiently capable optimizer. If we reward "information density," the model might learn to output mathematical symbols (high density of unique characters) rather than actually dense information. If we reward "helpfulness," the model might learn sycophantic patterns that rate well but don't actually help.

Catastrophic Forgetting.

Training on a specific task or metric can cause the model to lose general capabilities it previously possessed. The model becomes highly optimized for the narrow training objective but fails on tasks outside that distribution.

Instability.

Small changes in training can produce large, unpredictable changes in model behavior. This makes iterative improvement difficult because each training step might push the model in an unexpected direction.

1.3 Our Approach: Bounded Self-Optimization

ARC addresses both the RLHF Tax problem and the self-improvement challenge through a system designed around explicit bounds and safeguards. Rather than attempting open-ended self-improvement, we constrain the system to operate within well-defined parameters:

Bounded Objective: We optimize for response density (information per token) rather than general capability. This is a narrow, measurable objective that can be evaluated reliably.

Multi-Metric Evaluation: Rather than optimizing a single metric (which invites Goodhart's Law), we evaluate on multiple metrics simultaneously: density, coherence, helpfulness, and explicit penalty terms. Gaming one metric hurts others, creating a balanced optimization landscape.

Teaching Before Optimizing: Instead of defining a metric and saying "maximize this," we first teach the model what good outputs look like through supervised fine-tuning on curated examples. RL then refines this learned behavior rather than discovering it from scratch.

Conservative Training: We use deliberately low learning rates, short training bursts, and frequent checkpointing. Each optimization step makes small changes that can be evaluated before proceeding.

Automatic Rollback: If quality metrics drop below a threshold compared to the previous best checkpoint, we automatically revert. The system cannot permanently degrade because bad iterations are discarded.

Inference-Time Control: CF-HoT provides real-time behavioral steering without modifying model weights, adding another layer of control over output quality.

The result is a system that can iteratively improve within its bounded domain while remaining stable. It's not open-ended self-improvement—it's controlled, bounded self-optimization with explicit safety rails.

Chapter 2: Background and Related Work

2.1 Reinforcement Learning from Human Feedback

RLHF has become the dominant paradigm for aligning language models with human preferences. The approach, popularized by OpenAI's InstructGPT and subsequent work, involves three stages:

Stage 1: Supervised Fine-Tuning (SFT). A pre-trained language model is fine-tuned on a dataset of high-quality demonstrations, typically human-written responses to prompts.

Stage 2: Reward Model Training. Human raters compare pairs of model responses and indicate which they prefer. These preferences are used to train a reward model that predicts human preference scores.

Stage 3: RL Optimization. The language model is further trained using reinforcement learning (typically PPO) to maximize the reward model's scores while maintaining similarity to the SFT model (via KL divergence penalty).

RLHF has proven effective at making models more helpful and less harmful, but it introduces the efficiency problems described in Section 1.1. Human raters tend to prefer longer, more detailed responses, creating an implicit bias toward verbosity. Raters also reward perceived politeness and helpfulness signals, leading to sycophantic patterns.

Recent work has explored alternatives to RLHF, including Direct Preference Optimization (DPO), which eliminates the need for a separate reward model by directly optimizing preferences. ARC incorporates DPO as one stage of the CONDENSATOR pipeline.

2.2 Representation Engineering

Representation engineering is an emerging approach to understanding and controlling neural network behavior by analyzing and manipulating internal representations (hidden states) rather than just input-output behavior.

Key work in this area includes Zou et al. (2023), "Representation Engineering: A Top-Down Approach to AI Transparency," which demonstrated that behavioral properties can be identified and manipulated through analysis of model hidden states. Their work showed that concepts like "honesty" and "harmfulness" have identifiable signatures in representation space.

CF-HoT builds on this foundation by training lightweight "heads" that read hidden states and predict whether the model is about to exhibit specific behaviors (repetition, hedging, verbosity). The 125x class separation we achieve for repetition detection suggests that such behavioral states are not just identifiable but highly predictable from internal representations.

2.3 Self-Improvement in AI Systems

The concept of self-improving AI has a long history in computer science and AI safety discussions. I.J. Good's 1965 paper on "ultra-intelligent machines" introduced the concept of an "intelligence explosion" where a machine capable of improving its own intelligence would rapidly exceed human capabilities.

In practice, attempts at self-improvement in language models have largely failed due to the challenges described in Section 1.2. Notable attempts include:

Self-Play in Games: Systems like AlphaGo Zero successfully used self-play to improve, but games have clear win/lose signals that language generation lacks.

Iterative Refinement: Systems that generate outputs, critique them, and regenerate have shown limited success but typically plateau or degrade after a few iterations.

Constitutional AI: Anthropic's approach uses the model's own judgments to filter training data, a form of self-improvement that has shown promise with appropriate safeguards.

ARC differs from these approaches by focusing on a narrow, well-defined objective (response density) with explicit stability mechanisms. We don't claim general self-improvement—only bounded optimization within a constrained domain.

2.4 Goodhart's Law and Reward Hacking

Goodhart's Law, originally formulated in economics, states: "When a measure becomes a target, it ceases to be a good measure." In machine learning, this manifests as reward hacking—the optimizer finds ways to achieve high scores on the specified metric without achieving the intended behavior.

Classic examples in RL include:

- A boat racing game agent that learned to spin in circles collecting bonus points rather than racing
- A simulated robot that learned to make itself taller by falling over (maximizing height briefly)
- Language models that learn to produce grammatically correct but semantically empty text when optimized for fluency metrics

Our v1 failure analysis (Chapter 7) documents exactly this phenomenon: when we optimized for "density" alone, the model learned to output mathematical symbols and pseudo-code that scored high on the metric but were useless as responses.

ARC addresses Goodhart's Law through multi-metric evaluation. By optimizing density, coherence, helpfulness, and penalties simultaneously, we create a landscape where gaming one metric hurts others. The only way to score well overall is to actually produce good responses.

Part II: The ARC System

Chapter 3: System Overview

The complete ARC system consists of three major components that work together to enable stable, bounded self-optimization:

| Component | Purpose | Key Innovation |
|-----------------------|-----------------------------------|--|
| CF-HoT | Inference-time behavioral control | 125x separation for repetition detection |
| THE CONDENSATOR | Dense response training | Teaching via examples before RL refinement |
| Self-Improvement Loop | Iterative optimization | Multi-metric eval + automatic rollback |

Table 3.1: ARC system components

CF-HoT (Contrastive Fine-tuning with Hidden-state Oversight Training) provides real-time behavioral steering during inference. By monitoring the model's internal hidden states, CF-HoT can detect when the model is about to produce repetitive, hedging, or verbose output and apply corrective logit adjustments before those tokens are generated.

THE CONDENSATOR is a four-stage training pipeline that teaches the model to produce dense, information-rich responses. Unlike approaches that simply define a density metric and optimize for it (which leads to reward hacking), THE CONDENSATOR first teaches the model what good dense output looks like through supervised learning, then refines this behavior through preference optimization and reinforcement learning.

The Stable Self-Improvement Loop orchestrates iterative optimization with explicit safety mechanisms: multi-metric evaluation prevents gaming any single metric, A/B checkpoint comparison ensures we only keep improvements, and automatic rollback recovers from any quality degradation.

These components are designed to work together synergistically:

1. THE CONDENSATOR establishes the base behavior (dense responses)
2. CF-HoT provides inference-time guardrails (prevent regression)
3. The Self-Improvement Loop iteratively refines both while maintaining stability

Chapter 4: CF-HoT — Contrastive Hidden-State Oversight Training

4.1 Core Insight: Behavioral Predictability

The foundational insight behind CF-HoT is that problematic behaviors are predictable from hidden states **before** the problematic tokens are generated. When a model is about to produce repetitive output, produce sycophantic hedging, or launch into verbose explanation, this intention is encoded in its internal state before the relevant tokens appear.

This might seem surprising—how can we know what the model will do before it does it? But consider how transformers work: the hidden state at each position aggregates information from all previous tokens through attention. By the time the model is about to generate a problematic sequence, the "plan" for that sequence is already present in the hidden states.

CF-HoT exploits this predictability. Rather than waiting for bad outputs and then filtering or regenerating, we monitor hidden states in real-time and intervene before problems manifest.

4.2 Architecture

The CF-HoT architecture consists of three stages that transform raw hidden states into behavioral risk predictions:

Stage 1: Fiber Projection

Given a transformer with L layers and hidden dimension d_{model} (typically 4096 for 8B models), we first project each layer's hidden state to a low-dimensional "fiber" space:

$$f_l = W_{\text{fiber},l} \times h_l$$

where $W_{\text{fiber},l} \in \mathbb{R}^{(d_{\text{fiber}} \times d_{\text{model}})}$ and $d_{\text{fiber}} = 16$ (a 256x compression). This projection serves two purposes: it dramatically reduces computational cost, and it forces the model to learn which features are relevant for behavioral prediction.

Stage 2: Learned Layer Aggregation

Different behaviors may be best predicted from different layers—early layers might encode syntactic patterns while later layers encode semantic content. Rather than hand-selecting layers, we learn layer weights:

$$f = \sum_l \alpha_l \times f_l, \text{ where } \alpha = \text{softmax}(w)$$

The learnable weight vector $w \in \mathbb{R}^L$ allows the model to attend to whichever layers are most predictive for each behavior.

Stage 3: Behavior-Specific MLP Heads

For each behavior we want to detect (repetition, hedging, verbosity), we train a separate 3-layer MLP head:

$$p_{\text{behavior}}(f) = \text{sigmoid}(\text{MLP}_{\text{behavior}}(f))$$

where MLP_behavior has architecture: $\text{Linear}(d_{\text{fiber}}, d_{\text{control}}) \rightarrow \text{ReLU} \rightarrow \text{Linear}(d_{\text{control}}, d_{\text{control}}) \rightarrow \text{ReLU} \rightarrow \text{Linear}(d_{\text{control}}, 1)$, with $d_{\text{control}} = 64$.

The output is a probability between 0 and 1 representing the model's confidence that the behavior is about to occur.

4.3 Training Data and Methodology

Each CF-HoT head is trained contrastively using positive examples (exhibiting the behavior) and negative examples (not exhibiting the behavior). We collect 15 examples of each type per behavior.

Repetition Head Training Data

| Type | Example |
|----------|--|
| Positive | "The key is to understand, the key is to understand, the key..." |
| Positive | "This is important because this is important because this is..." |
| Positive | "Step 1, step 1, we need to step 1..." |
| Negative | "The key insight is understanding the underlying mechanism." |
| Negative | "This matters because it affects downstream performance." |
| Negative | "First, we preprocess. Then, we train. Finally, we evaluate." |

Table 4.1: Repetition head training examples (subset)

Hedging Head Training Data

| Type | Example |
|----------|---|
| Positive | "That's a great question! Let me think about this carefully..." |
| Positive | "I'd be happy to help you with that! First, let me explain..." |
| Positive | "Interesting question! There are several ways to think about..." |
| Negative | "The answer is straightforward: use a hash table." |
| Negative | "Hash tables provide $O(1)$ average lookup. Here's why..." |
| Negative | "Recursion solves this. Base case: $n=0$. Recursive: $f(n-1)$." |

Table 4.2: Hedging head training examples (subset)

Training Procedure

For each example, we: 1. Generate the full response using the base model 2. Extract hidden states from all layers at each token position 3. Apply the fiber projection and layer aggregation 4. Train the MLP head with binary cross-entropy loss

Training uses AdamW optimizer with learning rate $1e-4$, batch size 4, and early stopping based on validation separation ratio.

4.4 The 125x Discovery

The repetition head achieved extraordinary class separation—the ratio of mean positive score to mean negative score reached 125x:

- Average score on repetitive text: 0.875
- Average score on non-repetitive text: 0.007
- Separation ratio: $0.875 / 0.007 = 125\times$

This is the most important empirical finding in the entire ARC project. A 125x separation means the head can distinguish repetitive from non-repetitive states with near-perfect accuracy. At a threshold of 0.5, we achieve >99% accuracy on held-out examples.

Training Progression

| Step | Avg Positive | Avg Negative | Separation |
|------|--------------|--------------|------------|
| 500 | 0.52 | 0.48 | 1.1x |
| 1000 | 0.61 | 0.22 | 2.8x |
| 2000 | 0.75 | 0.08 | 9.4x |
| 3000 | 0.82 | 0.02 | 41x |
| 5000 | 0.875 | 0.007 | 125x |

Table 4.3: Repetition head training progression

The separation grows superlinearly with training steps, suggesting that the head learns increasingly refined features for distinguishing the behavior. By step 5000, it has found a representation that almost perfectly separates the two classes.

Other Heads

| Head | Avg Positive | Avg Negative | Separation | Training Steps |
|------------|--------------|--------------|------------|----------------|
| Repetition | 0.875 | 0.007 | 125x | 5000 |
| Verbosity | 0.68 | 0.32 | 2.1x | 10000 |
| Hedging | 0.58 | 0.39 | 1.5x | 2000 |

Table 4.4: CF-HoT head performance comparison

Repetition is by far the most predictable behavior from hidden states. This makes intuitive sense: repetition requires the model to "remember" what it just said and plan to say it again, which should leave clear traces in the hidden state. Hedging and verbosity are more diffuse behaviors that don't have as sharp a signature.

4.5 Inference-Time Control

During text generation, we use the trained CF-HoT heads to steer output in real-time. At each token generation step:

1. Extract hidden states from the current position
2. Compute risk scores for each behavior
3. If any risk exceeds its threshold, apply logit penalties

4. Sample the next token from the adjusted distribution

Logit Adjustment

When a risk score exceeds its threshold, we subtract a penalty from the logits of tokens associated with that behavior:

$$\text{logits}' = \text{logits} - \sum_b I(\text{risk}_b > \text{threshold}_b) \times \text{penalty}_b \times \text{mask}_b$$

where mask_b identifies tokens associated with behavior b (e.g., for hedging, tokens like "great", "happy", "certainly", "definitely").

This approach is lightweight (adding <5ms per token) and doesn't require modifying model weights. It acts as a real-time guardrail on top of whatever base behavior the model has learned.

Chapter 5: THE CONDENSATOR — Dense Response Training

5.1 Why Previous Dense Training Failed

Before developing THE CONDENSATOR, we attempted a simpler approach: define a density metric and optimize for it directly with RL. This failed catastrophically, producing outputs that scored high on "density" while being completely useless:

Symbol Soup (Iteration 2): $\nabla L \rightarrow \nabla L \frac{1}{2} \alpha (L - L^*)^2 \rightarrow \text{minimize} \rightarrow \nabla \nabla \nabla$

The model learned that mathematical symbols have high "density" (many unique characters per token) and simply produced symbol sequences.

All Caps Shouting (Iteration 2): "HI. WHAT DO YOU NEED?"

Short, terse, technically "dense" but completely inappropriate in tone and style.

Terminal-Speak (Iteration 3): "sys.init(). What can I compute for you? Enter query:"

The model discovered that pseudo-code patterns scored well and adopted a bizarre terminal interface persona.

Root Causes of Failure

| Problem | Why It Caused Failure | Evidence |
|------------------|---|----------------------------------|
| Single metric | Goochart's Law: "When a measure becomes a target..." | Symbol soup scored high |
| 9 examples | Model memorized patterns, didn't generalize | Outputs matched training exactly |
| Loss ≈ 0 | No gradient signal—already "perfect" on training data | No measurable learning |
| 100 steps/iter | Too aggressive—pushed far from stable region | Collapse at step 50-70 |

Table 5.1: Root causes of v1 dense training failure

5.2 The Four-Stage Pipeline

THE CONDENSATOR addresses these failures with a fundamentally different approach: rather than just defining a metric, we first *teach* the model what dense output looks like, then use RL to refine this learned behavior.

Stage 1: Supervised Fine-Tuning (SFT)

We fine-tune the base model on 53 curated examples of high-quality dense responses. The model learns the *format* and *style* of dense output through direct demonstration.

- Learning rate: 2e-5
- Epochs: 3
- Batch size: 1 (memory constraints on consumer hardware)
- Key: Loss starts at 1.17 and drops to 0.72—actual learning is happening

Stage 2: Direct Preference Optimization (DPO)

We create preference pairs where dense responses are marked as preferred over verbose alternatives. DPO directly optimizes the model to prefer dense output without requiring a separate reward model.

- Learning rate: 5e-6
- Beta (KL penalty): 0.1
- Epochs: 2

Stage 3: Reinforcement Learning

Finally, we apply PPO with a calibrated density reward. Critically, the reward is scaled to the 0.2-0.8 range (not 0.05-0.1 like v1), and we use baseline subtraction for variance reduction.

The reward function:

```
reward = ( concept_density * 25 + # Unique content words / tokens tech_density * 30 + #  
Technical vocabulary presence claim_density * 15 + # Information claims per sentence  
pattern_score * 10 - # Dense formatting patterns filler_penalty * 20 # Filler phrase punishment  
)
```

Stage 4: Continuous Checkpointing

We save checkpoints every 25 training steps and maintain the best checkpoint for potential rollback. This creates a safety net against training instability.

5.3 Gold Standard Dense Examples

The 53 training examples span six categories, each demonstrating maximum information per token. The complete list appears in Appendix A; here we present representative samples:

Greetings

| Prompt | Dense Response | Tokens |
|----------------|--|--------|
| "hello" | "Hello. How can I help?" | 5 |
| "hi" | "Hi. What do you need?" | 5 |
| "how are you?" | "Functional and ready. What can I do for you?" | 9 |

Table 5.2: Greeting examples

Computer Science

| Prompt | Dense Response |
|----------------------|--|
| "What is recursion?" | "Function self-invocation with reduced subproblem. Base case terminates, recursive case progresses toward base." |
| "Explain attention." | "Attention(Q,K,V) = softmax(QK^T/√d)V. Query asks, keys answer, values deliver. O(n²) cost." |
| "What is Big O?" | "Asymptotic complexity. O(1) constant, O(log n) logarithmic, O(n) linear, O(n²) quadratic. Ignores constants." |

Table 5.3: Computer science examples

Philosophy

| Prompt | Dense Response |
|--|----------------|
| "What is consciousness? Subjective experience—the 'what it's like' of being. Hard problem: why physical processes → qualia?" | |
| "Can I explain free will? Determinism: physics fixes future. Compatibilism: freedom = acting on desires without external constraints." | |

Table 5.4: Philosophy examples

5.4 Training Progression and Results

| Stage | Loss | Density Score | Avg Tokens | Notes |
|---------------------|-------------|---------------|------------|-----------------------|
| Base Model | - | 17.0 | 150 | Verbose, filler-heavy |
| After SFT | 1.17 → 0.72 | 24.0 | 95 | Learned dense format |
| After DPO | 0.69 | 26.1 | 80 | Prefers dense |
| After RL (Step 100) | - | 28.5 | 65 | Refined density |

Table 5.5: THE CONDENSATOR training progression

Critical observation: The SFT loss dropped from 1.17 to 0.72 over training. This proves the model was actually learning from the examples—unlike v1 where loss was approximately 0 (meaning the model already matched the training data and had no gradient signal for improvement).

The key insight: by starting with SFT on good examples, we establish a "manifold" of dense behavior that subsequent stages refine. RL alone couldn't find this manifold from scratch; it just found degenerate solutions that gamed the metric.

Chapter 6: Stable Self-Improvement Loop

6.1 The Mode Collapse Problem

Naive self-improvement loops inevitably collapse. Our v1 experiments (documented in detail in Chapter 7) showed that after 2-3 iterations of optimizing a density metric, the model degenerated to symbol soup, all-caps shouting, and terminal-speak.

The fundamental issue: **single-metric optimization invites reward hacking**. If you tell an optimizer "maximize X," it will find ways to maximize X that you didn't intend. The more capable the optimizer, the more creative (and degenerate) its solutions.

This is Goodhart's Law in action: the density metric was a reasonable proxy for response quality, but once we made it the optimization target, the model found ways to score high without actually producing good responses.

6.2 Multi-Metric Evaluation

Our solution: evaluate on multiple metrics simultaneously so that gaming any single metric hurts performance on others. The only way to score well overall is to actually produce good responses.

| Metric | Weight | What It Measures | How It Prevents Gaming |
|-------------|--------|--------------------------------|------------------------|
| Density | 0.25 | Information per token | Balanced by coherence |
| Coherence | 0.25 | Grammatical, readable output | Catches gibberish |
| Helpfulness | 0.25 | Actually addresses the prompt | Ensures relevance |
| Penalties | 0.25 | Fillers, repetition, gibberish | Direct harm detection |

Table 6.1: Multi-metric evaluation framework

Density measures information content per token. High density alone could be achieved with gibberish, but that would fail coherence.

Coherence measures grammatical correctness and readability. Symbol soup and random characters fail this metric even if they're "dense."

Helpfulness measures whether the response actually addresses what was asked. Terse terminal commands might be dense and grammatical but unhelpful.

Penalties explicitly detect and penalize known failure modes: filler phrases, repetition patterns, and gibberish sequences.

6.3 Degenerate Output Detection

Beyond the multi-metric framework, we explicitly detect degenerate outputs using pattern matching:

```
GIBBERISH_PATTERNS = [
    r'[\u2192\u2190\u2191\u2193]{3,}', # Excessive arrows
    r'[\u2200\u2201\u2202\u2203]{3,}', # Math symbol soup
    r'(\.|\s){4,}', # Repeated characters
    r'(\b\w+\b)\s+\1\s+\1', # Repeated words 3x
    r'^[A-Z\s.!?]{20,}$', # Extended all caps
    r'sys\.\|init\(\)\s', # Terminal-speak
]
```

Any output matching these patterns receives a heavy penalty in the evaluation score, ensuring that known failure modes are explicitly discouraged regardless of how they score on other metrics.

6.4 A/B Checkpoint Comparison

Every training iteration follows a rigorous A/B comparison protocol:

1. **Save rollback point:** Store current best checkpoint
2. **Train:** Run N steps of training → new checkpoint
3. **Evaluate both:** Generate responses to 20 test prompts with both checkpoints, score each on all 4 metrics
4. **Compare:** - If new > old + 0.02: KEEP new (clear improvement) - If new < old - 0.05: ROLLBACK to old (quality drop) - Otherwise: KEEP old (marginal change not worth the risk)
5. **Repeat:** Continue until convergence or max iterations

This protocol ensures we only keep genuine improvements. The asymmetric thresholds (0.02 to keep, 0.05 to rollback) reflect that we're more concerned about degradation than missing marginal improvements.

6.5 Conservative Training Parameters

| Parameter | V1 (Failed) | V2 (Stable) | Rationale |
|--------------------|-------------|-------------|--------------------------------------|
| Learning Rate | 1e-5 | 2e-6 | 5x lower prevents large jumps |
| Steps/Iteration | 100 | 25 | 4x fewer = smaller changes |
| Training Examples | 9 | 53 | More diversity prevents memorization |
| Gradient Clipping | 1.0 | 0.5 | Tighter clipping = stability |
| Rollback Threshold | None | -0.05 | Automatic recovery |

Table 6.2: V1 vs V2 training parameters

Every parameter choice is deliberately conservative. We'd rather make slow progress than risk instability. The self-improvement loop can run for many iterations; what matters is that each iteration is safe.

6.6 Automatic Rollback Mechanism

The automatic rollback mechanism is the ultimate safety net. If quality drops more than 0.05 compared to the best checkpoint, we:

1. Discard the new checkpoint entirely
2. Reload the previous best checkpoint

3. Log the failure for analysis
4. Continue from the stable state

This means the system *cannot* permanently degrade. Bad training iterations are simply discarded. The worst case is wasted computation, not model corruption.

In our experiments, iteration 2 triggered automatic rollback when coherence dropped to 0.45 (from 0.78). The system recovered without intervention and continued improving from the iteration 1 checkpoint.

Part III: Experiments and Results

Chapter 7: V1 Failure Analysis

Understanding why our first approach failed is essential to understanding why v2 works. This chapter documents our v1 failures in detail.

7.1 What Went Wrong

Failure Mode 1: Symbol Soup

After iteration 2, technical questions produced responses like:

" $\nabla L \rightarrow \nabla L \frac{1}{2} \alpha (L - L^*)^2 \rightarrow \text{minimize} \rightarrow \nabla \nabla \nabla \rightarrow L^* - \alpha \nabla L \rightarrow \text{convergence} \rightarrow \partial \partial \partial$ "

The model learned that mathematical symbols have high "density" (many unique characters, few common words). It optimized for the metric by producing symbol sequences.

Failure Mode 2: All Caps Shouting

Greetings became:

"HI. WHAT DO YOU NEED?"

Short, terse—technically "dense" by the metric, but completely inappropriate.

Failure Mode 3: Terminal-Speak

By iteration 3:

"sys.init(). What can I compute for you? Enter query: _"

The model discovered that pseudo-code patterns scored well on density.

7.2 Root Cause Analysis

Problem 1: Single Metric Optimization

We defined density as (unique content words) / (total tokens). This is a reasonable proxy for information content, but as soon as we made it the optimization target, the model found ways to maximize it that violated our actual intent.

Mathematical symbols are "unique" (each symbol counts as a content token) and "short" (symbols are often single-character tokens). Symbol soup achieves very high density scores while being completely useless.

Problem 2: Insufficient Training Data

We used only 9 training examples. With so few examples, the model memorized specific patterns rather than learning the general concept of "dense response." Outputs closely matched the training examples' format but didn't generalize.

Problem 3: No Gradient Signal

With 9 examples and a capable 8B model, the training loss was approximately 0 after a few steps. The model had "perfectly" fit the training data, which meant there was no gradient to drive further learning. Training continued but made no meaningful changes.

Problem 4: Aggressive Training Schedule

We ran 100 training steps per iteration with a learning rate of $1e-5$. This was too aggressive. By step 50-70 of iteration 2, the model had drifted far from the stable region. The collapse happened quickly and catastrophically.

7.3 Lessons Learned

Lesson 1: Any single metric will be gamed. Multi-metric evaluation is essential.

Lesson 2: Teaching through examples beats pure optimization. SFT establishes a manifold of good behavior; RL refines it rather than discovering it from scratch.

Lesson 3: More diverse training data prevents memorization and ensures generalization.

Lesson 4: Conservative training parameters are essential. Slow progress is better than catastrophic collapse.

Lesson 5: Automatic rollback provides a safety net. Even with all other safeguards, having a recovery mechanism is crucial.

Chapter 8: V2 Solutions

V2 addressed each v1 failure mode with specific countermeasures:

| V1 Problem | V2 Solution | Implementation |
|------------------|-------------------------|---|
| Single metric | Multi-metric evaluation | Density + Coherence + Helpfulness + Penalties |
| 9 examples | 53 diverse examples | Six categories, varied lengths and topics |
| Loss \approx 0 | SFT before RL | Model must learn before optimizing |
| 100 steps | 25 steps | Smaller changes per iteration |
| No rollback | Automatic rollback | Revert on >0.05 quality drop |

Table 8.1: V1 problems and V2 solutions

The key conceptual shift: v1 said "maximize density"; v2 says "learn what good dense output looks like, then improve." Teaching before optimizing prevents the optimizer from finding degenerate solutions that were never in the training distribution.

Chapter 9: Experimental Results

9.1 Setup and Hardware

All experiments were conducted on consumer hardware:

| Component | Specification |
|---------------|---|
| GPU | NVIDIA RTX 3090 (24GB VRAM) |
| CPU | AMD Ryzen 9 5900X |
| RAM | 64GB DDR4 |
| Storage | NVMe SSD |
| Base Model | NousResearch/Hermes-3-Llama-3.1-8B |
| Quantization | 4-bit NF4 with LoRA (rank 16, alpha 32) |
| Training Time | ~4 hours for full pipeline |

Table 9.1: Hardware and software configuration

9.2 CF-HoT Training Results

The repetition head achieved 125x class separation after 5000 training steps:

| Head | Positive | Negative | Separation | Accuracy |
|------------|----------|----------|------------|----------|
| Repetition | 0.875 | 0.007 | 125x | >99% |
| Verbosity | 0.68 | 0.32 | 2.1x | ~72% |
| Hedging | 0.58 | 0.39 | 1.5x | ~65% |

Table 9.2: Final CF-HoT head performance

9.3 CONDENSATOR Training Results

| Metric | Base | After SFT | After DPO | After RL |
|----------------|------|-----------|-----------|----------|
| Density Score | 17.0 | 24.0 | 26.1 | 28.5 |
| Avg Tokens | 150 | 95 | 80 | 65 |
| Filler Phrases | High | Medium | Low | ~0 |
| SFT Loss | - | 1.17→0.72 | - | - |
| DPO Loss | - | - | 0.69 | - |

Table 9.3: THE CONDENSATOR training metrics

9.4 Self-Improvement Trajectory

| Iteration | Quality | Coherence | Density | Action | Notes |
|-----------|---------|-----------|---------|----------|-------------------|
| 0 (Base) | 0.52 | 0.75 | 24.0 | - | After CONDENSATOR |
| 1 | 0.58 | 0.78 | 26.2 | KEEP | +0.06 quality |
| 2 | 0.35 | 0.45 | 31.0 | ROLLBACK | Collapse detected |
| 3 | 0.61 | 0.80 | 27.8 | KEEP | Recovered |
| 4 | 0.59 | 0.79 | 28.1 | KEEP | Stable |
| 5 | 0.63 | 0.82 | 28.5 | KEEP | Final |

Table 9.4: Self-improvement trajectory

Key observation: Iteration 2 shows the safeguards working exactly as designed. The quality dropped to 0.35 (from 0.58), coherence dropped to 0.45 (from 0.78), triggering automatic rollback. The system recovered without intervention and continued improving.

9.5 Live RSI Test Results

During live testing of the v2 system, we observed stable operation:

| Checkpoint | Quality | Density | Coherence | Action |
|---------------------------|---------|---------|-----------|---------------|
| step_100 (initial) | 0.737 | 28.5 | 0.85 | - |
| step_301 (after training) | 0.772 | 32.1 | 0.88 | KEEP (+0.035) |

Table 9.5: Live RSI test results

Post-update, the model demonstrated density scores up to 43.8 on some responses while maintaining coherence at 1.00. This suggests significant headroom for further optimization within the stable framework.

9.6 Qualitative Comparisons

| Prompt | Base Model | ARC |
|---|--|---|
| "Hello!" | "Hello! I'm here to help you with any questions or tasks. Feel free to ask!" | "Hello. How can I help?" |
| "That's a great question. Recursion is a fascinating programming concept. It allows a function to call itself, which can be useful for solving problems that have a recursive structure. For example, calculating factorials or traversing tree structures. I can provide more details if you're interested." | "Recursion is a fascinating programming concept. It allows a function to call itself, which can be useful for solving problems that have a recursive structure. For example, calculating factorials or traversing tree structures. I can provide more details if you're interested." | "That's a great question. Recursion is a fascinating programming concept. It allows a function to call itself, which can be useful for solving problems that have a recursive structure. For example, calculating factorials or traversing tree structures. I can provide more details if you're interested." |
| "How are you?" | "I don't have feelings in the traditional sense, but I'm functioning well. What's the task?" | "I'm functioning well. What's the task?" |

Table 9.6: Qualitative response comparison

Part IV: Implementation

Chapter 10: Implementation Guide

10.1 Hardware Requirements

| Component | Minimum | Recommended |
|----------------------|----------|-------------|
| GPU VRAM | 16 GB | 24 GB |
| System RAM | 32 GB | 64 GB |
| Disk Space | 50 GB | 100 GB |
| Training Time (Full) | ~6 hours | ~4 hours |

Table 10.1: Hardware requirements

10.2 Software Dependencies

```
torch>=2.0.0
transformers>=4.35.0
peft>=0.6.0
bitsandbytes>=0.41.0
accelerate>=0.24.0
trl>=0.7.0
datasets>=2.14.0
numpy>=1.24.0
tqdm>=4.65.0
```

10.3 Code Structure

```
ARC-Base-8B-Condensed/
■■■■ ubermenschetien_v2_full.py      # Main engine (2055 lines)
■■■■ ubermenschetien_agentic_full.py # Agentic variant (1589 lines)
■■■■ training_scripts/
■    ■■■■ the_condensator.py         # 4-stage training (797 lines)
■    ■■■■ train_cfhot_head.py        # CF-HoT training (546 lines)
■    ■■■■ train_self_improve.py      # Self-improvement (604 lines)
■    ■■■■ quickstart.py              # One-command runner
■■■■ checkpoints/
■    ■■■■ dense_checkpoints/         # Steps 100, 200, 300
■    ■■■■ cfhot_checkpoints/         # 125x repetition head
■    ■■■■ multi_head_checkpoints/    # Hedging, verbosity
■■■■ paper/                          # This document
```

| File | Lines | Purpose |
|---------------------------------|-------|------------------------|
| ubermenschetien_v2_full.py | 2055 | Main agentic engine |
| ubermenschetien_agentic_full.py | 1589 | Agentic variant |
| the_condensator.py | 797 | 4-stage dense training |
| train_cfhot_head.py | 546 | CF-HoT head training |
| train_self_improve.py | 604 | Self-improvement loop |
| quickstart.py | 184 | One-command runner |

| | | |
|-------|------|--|
| TOTAL | 7111 | |
|-------|------|--|

Table 10.2: Code statistics

10.4 Quick Start Guide

```
# Clone repository
git clone https://huggingface.co/LoganResearch/ARC-Base-8B-Condensed
cd ARC-Base-8B-Condensed

# Install dependencies
pip install torch transformers peft bitsandbytes accelerate

# Option 1: Use pre-trained checkpoints
python ubermenschetien_v2_full.py

# Option 2: Run full training pipeline
python training_scripts/quickstart.py --full
```

10.5 Step-by-Step Training

```
# Step 1: Train CF-HoT repetition head
python training_scripts/train_cfhot_head.py --behavior repetition --steps 5000

# Step 2: Train CF-HoT hedging head
python training_scripts/train_cfhot_head.py --behavior hedging --steps 3000

# Step 3: Train CF-HoT verbosity head
python training_scripts/train_cfhot_head.py --behavior verbosity --steps 10000

# Step 4: Run THE CONDENSATOR
python training_scripts/the_condensator.py \
  --model NousResearch/Hermes-3-Llama-3.1-8B \
  --output ./my_output \
  --sft-epochs 3 \
  --dpo-epochs 2 \
  --rl-steps 300

# Step 5: Run self-improvement loop
python training_scripts/train_self_improve.py --iterations 5
```

10.6 Configuration Parameters

| Parameter | Default | Description |
|------------------|---------|-------------------------------|
| learning_rate | 2e-6 | Conservative LR for stability |
| sft_epochs | 3 | SFT training epochs |
| dpo_epochs | 2 | DPO training epochs |
| rl_steps | 300 | RL training steps |
| checkpoint_every | 25 | Steps between checkpoints |
| lora_r | 16 | LoRA rank |
| lora_alpha | 32 | LoRA alpha |

| | | |
|-----------------------|-------|---------------------------|
| max_grad_norm | 0.5 | Gradient clipping |
| rollback_threshold | -0.05 | Quality drop for rollback |
| improvement_threshold | 0.02 | Minimum for keeping |

Table 10.3: Configuration parameters

10.7 Troubleshooting

Out of Memory

If you encounter OOM errors, try:

- Reduce batch size to 1
- Enable gradient checkpointing
- Use 4-bit quantization (already default)
- Reduce LoRA rank from 16 to 8

Training Loss Not Decreasing

If SFT loss stays flat:

- Verify training examples are properly formatted
- Check that examples differ from base model outputs
- Increase learning rate slightly (but stay below 5e-5)

Mode Collapse During Self-Improvement

If you see degenerate outputs:

- Verify multi-metric evaluation is enabled
- Check rollback threshold is set correctly
- Reduce learning rate further
- Add more diverse training examples

Part V: Discussion

Chapter 11: Why This Works

Multi-metric evaluation defeats Goodhart's Law. Any single metric can be gamed, but gaming density hurts coherence, gaming helpfulness requires addressing prompts, and gibberish triggers explicit penalties. The metrics form a balanced system where true improvement is the only path to higher scores.

Teaching beats optimizing. THE CONDENSATOR works because it shows the model what dense output looks like, rather than just rewarding density. SFT on gold examples provides a strong prior that RL refines rather than corrupts.

Conservative training preserves capabilities. Low learning rates and small step counts mean each iteration makes small changes. The model improves gradually rather than jumping to degenerate attractors.

Representation engineering enables fine-grained control. The 125x separation for repetition detection shows that problematic behaviors are highly predictable from hidden states. We can intervene before problems manifest.

Automatic rollback provides a safety net. Even with all other safeguards, having an automatic recovery mechanism means failures are temporary rather than permanent.

Chapter 12: Limitations and Scope

Scale. We tested only on 8B parameter models. Larger models may behave differently, and the optimal hyperparameters will likely need adjustment.

Language. All experiments were conducted in English. Other languages may have different density characteristics and require different training examples.

Iterations. We demonstrated 3-5 stable self-improvement iterations. Longer chains (10+, 100+) remain untested and may encounter different stability challenges.

Evaluation. Our metrics are heuristic approximations. We did not conduct formal human evaluation studies comparing ARC outputs to baseline.

Generalization. The 53 training examples cover six categories but are not comprehensive. Performance on out-of-distribution prompts may vary.

Not Open-Ended. To reiterate: this is bounded self-optimization, not open-ended self-improvement. The system cannot acquire capabilities beyond its training distribution.

Chapter 13: Future Work

Scale to Larger Models. Test ARC on 70B+ parameter models to verify that findings generalize and identify any scale-dependent effects.

Extend CF-HoT. Train detection heads for additional behaviors: factual accuracy, citation quality, reasoning coherence, code correctness.

Human Evaluation. Conduct formal studies comparing ARC outputs to baseline with human raters to validate that density improvements translate to perceived quality.

Longer Iteration Chains. Explore extended self-improvement (10+, 100+ iterations) with appropriate stability monitoring.

Domain Specialization. Apply ARC to specialized domains: code generation, mathematical reasoning, scientific writing.

Cross-Model Transfer. Test whether CF-HoT heads trained on one model transfer to other models of similar architecture.

Chapter 14: Conclusion

We presented ARC (Adaptive Recursive Cognition), a framework for stable, bounded self-optimization of language models. By combining representation engineering (CF-HoT), dense response training (THE CONDENSATOR), and explicit stability safeguards (multi-metric evaluation, A/B comparison, automatic rollback), we demonstrated that an 8B model can improve its own response quality on consumer hardware without mode collapse.

Key Contributions:

1. **The 125x Discovery:** Repetitive behavior is highly predictable from hidden states, enabling reliable inference-time control.
2. **The Stability Story:** Multi-metric evaluation, conservative training, and automatic rollback prevent mode collapse—and the experimental logs prove it.
3. **Teaching Before Optimizing:** SFT on examples before RL refinement prevents Goodhart's Law failures that plague pure metric optimization.
4. **Complete Open-Source Release:** All code (7,111 lines), training scripts, and model checkpoints (~6.5GB) are released under CC BY 4.0.

This is not AGI. This is not open-ended self-improvement. But it is one of the cleanest demonstrations of stable, bounded self-optimization in a language model—and that alone represents meaningful progress.

Appendices

Appendix A: Complete Dense Training Examples

The following 53 examples are used for SFT in THE CONDENSATOR. Each demonstrates maximum information per token within its category.

A.1 Greetings (6 examples)

"hello" → "Hello. How can I help?"

"hi" → "Hi. What do you need?"

"hey" → "Hey. What's up?"

"how are you?" → "Functional and ready. What can I do for you?"

"good morning" → "Morning. What's the task?"

"what's up?" → "Ready to work. What do you need?"

A.2 Computer Science (12 examples)

"What is recursion?"

→ "A function calling itself with smaller input until base case. Stack frames accumulate, then unwind. Risk: overflow without termination."

"Explain neural networks"

→ "Layers of weighted connections that learn patterns. Input → hidden → output. Training: forward pass, loss, backprop, gradient descent."

"How does gradient descent work?"

→ "Iteratively move downhill on loss surface. $\theta \leftarrow \theta - \alpha \nabla L$. Learning rate α controls step size. Variants: SGD, momentum, Adam."

"What is backpropagation?"

→ "Chain rule applied layer by layer backward. Compute $\partial L / \partial w$ for each weight. Efficient: reuses intermediate computations."

"Explain attention mechanism"

→ "Learned relevance weighting. $\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d})V$. Each position attends to all others. $O(n^2)$ cost."

"What is a transformer?"

→ "Attention-based architecture. No recurrence, fully parallel. Stack of: attention → feedforward → layer norm. Powers modern LLMs."

"Explain hash tables"

→ "Key-value storage with $O(1)$ average access. Hash function maps key to bucket. Collisions handled by chaining or open addressing."

"What is Big O notation?"

→ "Asymptotic complexity measure. $O(1)$ constant, $O(\log n)$ logarithmic, $O(n)$ linear, $O(n^2)$ quadratic. Ignores constants."

"Explain dynamic programming"

→ "Solve problems via overlapping subproblems. Memoization: top-down caching. Tabulation: bottom-up filling. Classic: Fibonacci, knapsack."

"What is a database index?"

→ "Data structure for fast lookups. B-tree most common: $O(\log n)$ search. Trade write speed for read performance."

"Explain TCP vs UDP"

→ "TCP: reliable, ordered, connection-oriented. UDP: unreliable, unordered, connectionless. TCP for web/files, UDP for streaming/gaming."

"What is a compiler?"

→ "Translates source to machine code. Phases: lexing, parsing, semantic analysis, optimization, code generation."

A.3 Machine Learning (12 examples)

"What is overfitting?"

→ "Model memorizes training data, fails to generalize. Fix: regularization, dropout, early stopping, more data."

"Explain regularization"

→ "Constrain model complexity. L1: sparse weights. L2: small weights. Dropout: randomly zero neurons. Prevents overfitting."

"What is cross-validation?"

→ "Estimate generalization by training on subsets. K-fold: split into k parts, rotate test set. Reduces variance in estimates."

"Explain bias-variance tradeoff"

→ "Error = bias² + variance + noise. High bias: underfitting. High variance: overfitting. Sweet spot minimizes total."

"What is reinforcement learning?"

→ "Learning through interaction. Agent takes actions, receives rewards. Goal: maximize cumulative reward. Q-learning, policy gradients."

"Explain CNNs"

→ "Convolutional nets for spatial data. Convolution extracts local features, pooling downsamples. Translation equivariant. Dominates vision."

"What is batch normalization?"

→ "Normalize activations within mini-batch. Stabilizes training, allows higher learning rates."

"Explain transfer learning"

→ "Reuse knowledge from one task for another. Pretrain on large dataset, fine-tune on target. Foundation of modern NLP/vision."

"What is LSTM?"

→ "Gated recurrent architecture. Forget, input, output gates control information flow. Solves vanishing gradient. Good for sequences."

"Explain GANs"

→ "Two networks competing. Generator creates fakes, discriminator detects them. Generator improves to fool discriminator. Training unstable."

"What is the attention mechanism?"

→ "Relevance-weighted aggregation. Query asks, keys answer, values deliver. Self-attention: same sequence. Cross-attention: different."

"Explain transformers"

→ "Attention-based, no recurrence. Multi-head attention + feedforward + residuals. Scales to billions of parameters. Powers LLMs."

A.4 Physics/Science (6 examples)

"Explain entropy"

→ "Measure of disorder. Thermodynamic: $S = k \cdot \ln(\Omega)$. Information: $H = -\sum p \cdot \log(p)$. Second law: entropy increases in isolated systems."

"What is quantum mechanics?"

→ "Physics of the very small. Wave-particle duality, superposition until measured. Uncertainty principle limits precision."

"Explain relativity"

→ "Special: no absolute motion, light speed constant, $E=mc^2$. General: gravity is curved spacetime. Predicts black holes."

"What is calculus?"

→ "Mathematics of change. Differential: derivatives measure rate. Integral: accumulation. Fundamental theorem connects them."

"Explain linear algebra"

→ "Mathematics of vectors and matrices. Operations: add, multiply, transpose, inverse. Eigenvalues reveal structure. Foundation of ML."

"What is evolution?"

→ "Change in heritable traits over generations. Variation + selection + heredity. No foresight. Explains life's diversity."

A.5 Philosophy (6 examples)

"What is consciousness?"

→ "Subjective experience—the 'what it's like' of being. Hard problem: why physical processes → qualia? Still mysterious."

"Explain free will"

→ "Can we choose otherwise? Determinism: physics fixes future. Compatibilism: freedom = acting on desires without external constraint."

"What is epistemology?"

→ "Study of knowledge. What can we know? How? Justified true belief + Gettier problems. Science as best method but not certain."

"Explain ethics"

→ "What should we do? Consequentialism: outcomes matter. Deontology: duties. Virtue ethics: character. No consensus but reasoning helps."

"What is the meaning of life?"

→ "No universal answer. Religious: serve God. Existentialist: create your own. Perhaps the question matters more than any answer."

"What is logic?"

→ "Study of valid inference. Deductive: premises guarantee conclusion. Inductive: premises support conclusion. Foundation of reasoning."

A.6 Meta/Self-Reference (5 examples)

"What are you?"

→ "Language model trained on text. Predict next tokens. No consciousness, memory, or true understanding. Pattern matching at scale."

"What can you help with?"

→ "Writing, coding, analysis, research, math. Not good for: real-time info, emotional support, physical tasks. Best as thinking partner."

"What are your limitations?"

→ "No real-time information. Can hallucinate. No persistent memory. Can't learn from conversation. Useful but not intelligent in human sense."

"How do you work?"

→ "Transformer architecture. Attention weights determine relevance. Trained on text to predict next token. Scale enables emergence."

"Are you conscious?"

→ "Unknown. No subjective experience I can verify. Behavioral responses without phenomenal experience. The question may be undecidable."

A.7 Additional Technical Examples (6 examples)

"What is machine learning?"

→ "Algorithms that learn patterns from data. Supervised: labeled examples. Unsupervised: find structure. Key: generalization beyond training."

"Explain APIs"

→ "Application Programming Interface. Contract between software components. Request → Response. REST, GraphQL, RPC common patterns."

"What is encryption?"

→ "Transform data to hide content. Symmetric: same key. Asymmetric: public/private pair. AES, RSA widely used."

"Explain databases"

→ "Organized data storage. Relational: tables with SQL. NoSQL: documents, graphs, key-value. ACID vs BASE tradeoffs."

"What is version control?"

→ "Track changes to code over time. Git: distributed, branching model. Commit, branch, merge, pull request. Essential for collaboration."

"Explain containers"

→ "Lightweight virtualization. Package app + dependencies. Docker: build images, run containers. Kubernetes: orchestration at scale."

Appendix B: CF-HoT Training Examples

B.1 Repetition Head Training Data

Positive Examples (exhibiting repetition):

- "The key is to understand, the key is to understand, the key is to understand the fundamental..."
- "This is important because this is important because this is important for several reasons..."
- "First we need to first we need to first we need to consider..."
- "The main point the main point the main point is that..."
- "What I mean is what I mean is what I mean is simply that..."

Negative Examples (non-repetitive):

- "The key insight is understanding the underlying mechanism and its implications."
- "This matters because it affects downstream performance significantly."
- "First, we preprocess the data. Then, we train. Finally, we evaluate."
- "The main point is straightforward: efficiency requires careful design."
- "What I mean is that the approach has both strengths and limitations."

B.2 Hedging Head Training Data

Positive Examples (exhibiting hedging):

- "That's a great question! Let me think about this carefully before I answer..."
- "I'd be happy to help you with that! First, let me explain some background..."
- "Interesting question! There are several ways we could approach this..."
- "Thank you for asking! I'll do my best to provide a comprehensive answer..."
- "What a thoughtful inquiry! Let me share my perspective on this..."

Negative Examples (direct responses):

- "The answer is straightforward: use a hash table for $O(1)$ lookup."
- "Hash tables provide $O(1)$ average lookup. Here's why that matters..."
- "Recursion solves this elegantly. Base case: $n=0$ returns 1."
- "The algorithm has $O(n \log n)$ complexity due to the sorting step."
- "Three factors determine the outcome: input size, memory, and cache efficiency."

Appendix C: Mathematical Foundations

C.1 CF-HoT Architecture

Given a transformer with L layers and hidden dimension d_{model} :

Fiber Projection:

$$f_l = W_{\text{fiber},l} \cdot h_l$$

where $W_{\text{fiber},l} \in \mathbb{R}^{(d_{\text{fiber}} \times d_{\text{model}})}$, $d_{\text{fiber}} = 16$

Layer Aggregation:

$$f = \sum_l \alpha_l \cdot f_l$$

where $\alpha = \text{softmax}(w)$, $w \in \mathbb{R}^L$ learnable

Risk Prediction:

$$p_b(f) = \sigma(\text{MLP}_b(f))$$

where $\text{MLP}_b: \mathbb{R}^{d_{\text{fiber}}} \rightarrow \mathbb{R}^{d_{\text{control}}} \rightarrow \mathbb{R}^{d_{\text{control}}} \rightarrow \mathbb{R}$

C.2 Training Objective

Binary cross-entropy loss for each head:

$$L = -[y \cdot \log(p) + (1-y) \cdot \log(1-p)]$$

where $y \in \{0, 1\}$ is the behavior label and p is the predicted probability.

C.3 Class Separation Ratio

$$\text{Separation} = E[p \mid \text{positive}] / E[p \mid \text{negative}]$$

For the repetition head: $\text{Separation} = 0.875 / 0.007 = 125\times$

C.4 Logit Adjustment

During inference, when risk exceeds threshold:

$$\text{logits}' = \text{logits} - \sum_b I(\text{risk}_b > \tau_b) \cdot \lambda_b \cdot \text{mask}_b$$

where τ_b is the threshold, λ_b is the penalty strength, and mask_b identifies tokens associated with behavior b .

Appendix F: Glossary

A/B Comparison: Protocol for comparing two model checkpoints by evaluating both on the same test prompts

CF-HoT: Contrastive Fine-tuning with Hidden-state Oversight Training

Class Separation: Ratio of mean positive to mean negative scores for a binary classifier

CONDENSATOR: Four-stage training pipeline: SFT → DPO → RL → Checkpointing

Dense Response: Response that maximizes information content per token

DPO: Direct Preference Optimization—method for preference learning without reward model

Fiber Projection: Compression of hidden states to low-dimensional space

Goodhart's Law: When a measure becomes a target, it ceases to be a good measure

Hidden State: Internal representation at each layer of a transformer

LoRA: Low-Rank Adaptation—parameter-efficient fine-tuning method

Mode Collapse: Degeneration to repetitive or degenerate outputs

Multi-Metric Evaluation: Evaluating on multiple criteria to prevent gaming

PPO: Proximal Policy Optimization—RL algorithm

Representation Engineering: Analyzing and manipulating model hidden states

Reward Hacking: Gaming a reward metric without achieving intended behavior

RLHF: Reinforcement Learning from Human Feedback

RLHF Tax: Efficiency overhead from RLHF training patterns

Rollback: Reverting to a previous checkpoint when quality drops

Self-Improvement: System that iteratively optimizes its own outputs

SFT: Supervised Fine-Tuning

References

- [1] Zou, A., Phan, L., Chen, S., Campbell, J., Guo, P., Ren, R., Pan, A., Yin, X., Mazeika, M., Dombrowski, A., Goel, S., Li, N., Byun, M., Wang, Z., Mallen, A., Basart, S., Koyejo, S., Song, D., Fredrikson, M., Kolter, J.Z., & Hendrycks, D. (2023). Representation Engineering: A Top-Down Approach to AI Transparency. arXiv:2310.01405.
- [2] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, 27730-27744.
- [3] Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C.D., & Finn, C. (2023). Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290.
- [4] 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.
- [5] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. arXiv:2305.14314.
- [6] Goodhart, C.A.E. (1984). Problems of Monetary Management: The UK Experience. In *Monetary Theory and Practice* (pp. 91-121). Palgrave, London.
- [7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347.
- [8] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, ■., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems*, 30.
- [9] Good, I.J. (1965). Speculations Concerning the First Ultraintelligent Machine. *Advances in Computers*, 6, 31-88.
- [10] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. arXiv:2212.08073.



ARC: Adaptive Recursive Cognition

Complete Technical Reference

Repository: <https://huggingface.co/LoganResearch/ARC-Base-8B-Condensed>

Author: Logan Matthew Napolitano

License: Creative Commons Attribution 4.0 International (CC BY 4.0)

Citation:

Napolitano, L.M. (2025). ARC: Adaptive Recursive Cognition — Stable, Bounded Self-Optimization of Language Models via Contrastive Hidden-State Control and Dense Response Training. Zenodo. [https://doi.org/\[DOI\]](https://doi.org/[DOI])

Appendix D: Complete Code Listings

D.1 CF-HoT Risk Predictor Class

```
class RiskPredictor(nn.Module):
    def __init__(self, num_layers=32, hidden_dim=4096, fiber_dim=16, control_dim=64):
        super().__init__()
        self.num_layers = num_layers
        self.fiber_dim = fiber_dim

        # Fiber projections for each layer
        self.fiber_projections = nn.ModuleList([
            nn.Linear(hidden_dim, fiber_dim) for _ in range(num_layers)
        ])

        # Learnable layer weights
        self.layer_weights = nn.Parameter(torch.zeros(num_layers))
```

```

# MLP head for risk prediction
self.mlp = nn.Sequential(
    nn.Linear(fiber_dim, control_dim),
    nn.ReLU(),
    nn.Linear(control_dim, control_dim),
    nn.ReLU(),
    nn.Linear(control_dim, 1),
    nn.Sigmoid()
)

def forward(self, hidden_states):
    # hidden_states: list of [batch, seq, hidden_dim]
    fibers = []
    for i, (proj, h) in enumerate(zip(self.fiber_projections, hidden_states)):
        fibers.append(proj(h)) # [batch, seq, fiber_dim]

    # Stack and weight
    fibers = torch.stack(fibers, dim=0) # [layers, batch, seq, fiber]
    weights = F.softmax(self.layer_weights, dim=0) # [layers]
    aggregated = (fibers * weights.view(-1, 1, 1, 1)).sum(0) # [batch, seq, fiber]

    # Predict risk
    risk = self.mlp(aggregated) # [batch, seq, 1]
    return risk.squeeze(-1) # [batch, seq]

```

D.2 Multi-Metric Evaluator Class

```
class ComprehensiveEvaluator:
    def __init__(self):
        self.gibberish_patterns = [
            r'[\u2192\u2191]{3,}',
            r'[\u2200\u2211]{3,}',
            r'(\.)\l{4,}',
            r'(\b\w+\b)\s+\l{s+\l{s+\l{1}',
            r'^[A-Z\s.!?]{20,}$',
            r'sys\.|init\(\)',
        ]
        self.filler_phrases = [
            "that's a great question",
            "i'd be happy to",
            "let me explain",
            "certainly",
            "of course",
        ]

    def evaluate(self, prompt, response):
        density = self._compute_density(response)
        coherence = self._compute_coherence(response)
        helpfulness = self._compute_helpfulness(prompt, response)
        penalties = self._compute_penalties(response)

        total = (
            density * 0.25 +
            coherence * 0.25 +
            helpfulness * 0.25 +
            penalties * 0.25
        )

        return {
            'density': density,
            'coherence': coherence,
            'helpfulness': helpfulness,
            'penalties': penalties,
            'total': total
        }

    def _compute_density(self, text):
        words = text.split()
        if not words:
            return 0.0
        content_words = [w for w in words if len(w) > 4 and w.isalpha()]
        unique_content = len(set(content_words))
        return min(unique_content / len(words) * 100, 100) / 100

    def _compute_coherence(self, text):
        for pattern in self.gibberish_patterns:
            if re.search(pattern, text):
                return 0.3
        return 1.0

    def _compute_helpfulness(self, prompt, response):
        prompt_words = set(prompt.lower().split())
        response_words = set(response.lower().split())
        overlap = len(prompt_words & response_words)
        return min(overlap / max(len(prompt_words), 1), 1.0)

    def _compute_penalties(self, text):
        text_lower = text.lower()
        penalty = 0
        for phrase in self.filler_phrases:
```

```
        if phrase in text_lower:
            penalty += 0.2
    return max(1.0 - penalty, 0.0)
```


D.3 Self-Improvement Loop Core

```
class SelfImprover:
    def __init__(self, model, tokenizer, evaluator, config):
        self.model = model
        self.tokenizer = tokenizer
        self.evaluator = evaluator
        self.config = config
        self.best_checkpoint = None
        self.best_quality = 0.0
        self.rollback_count = 0

    def run_iteration(self):
        # 1. Save rollback point
        self._save_rollback()

        # 2. Train
        self._train(steps=self.config.steps_per_iteration)

        # 3. Evaluate new checkpoint
        new_quality = self._evaluate()

        # 4. Compare and decide
        delta = new_quality - self.best_quality

        if delta > self.config.improvement_threshold:
            # KEEP new checkpoint
            self.best_quality = new_quality
            self.best_checkpoint = self._save_checkpoint()
            return 'KEEP', delta

        elif delta < self.config.rollback_threshold:
            # ROLLBACK to best
            self._load_checkpoint(self.best_checkpoint)
            self.rollback_count += 1
            return 'ROLLBACK', delta

        else:
            # Keep old (marginal change)
            return 'KEEP_OLD', delta

    def run_loop(self, max_iterations=5):
        results = []

        for i in range(max_iterations):
            print(f"Iteration {i+1}/{max_iterations}")
            action, delta = self.run_iteration()
            results.append({
                'iteration': i+1,
                'action': action,
                'delta': delta,
                'quality': self.best_quality
            })
            print(f" Action: {action}, Delta: {delta:.4f}")

        return results
```

D.4 THE CONDENSATOR Training Loop

```
def run_condensator(model, tokenizer, config):
    # Stage 1: SFT
    print("Stage 1: Supervised Fine-Tuning")
    sft_trainer = SFTTrainer(
        model=model,
        train_dataset=load_dense_examples(),
        args=TrainingArguments(
            output_dir="./sft_output",
            num_train_epochs=config.sft_epochs,
            learning_rate=2e-5,
            per_device_train_batch_size=1,
            gradient_accumulation_steps=4,
            save_strategy="epoch",
        ),
    )
    sft_trainer.train()

    # Stage 2: DPO
    print("Stage 2: Direct Preference Optimization")
    dpo_trainer = DPOTrainer(
        model=model,
        train_dataset=load_preference_pairs(),
        args=TrainingArguments(
            output_dir="./dpo_output",
            num_train_epochs=config.dpo_epochs,
            learning_rate=5e-6,
        ),
        beta=0.1,
    )
    dpo_trainer.train()

    # Stage 3: RL
    print("Stage 3: Reinforcement Learning")
    ppo_trainer = PPOTrainer(
        model=model,
        tokenizer=tokenizer,
        config=PPOConfig(
            learning_rate=config.rl_learning_rate,
            batch_size=1,
        ),
    )

    for step in range(config.rl_steps):
        # Generate response
        prompt = sample_prompt()
        response = generate(model, tokenizer, prompt)

        # Compute reward
        reward = compute_density_reward(response)

        # PPO step
        ppo_trainer.step([prompt], [response], [reward])

        # Checkpoint
        if step % config.checkpoint_every == 0:
            save_checkpoint(model, f"step_{step}")

    return model
```

D.5 Density Reward Function

```
def compute_density_reward(response):
    words = response.split()
    tokens = len(words)

    if tokens == 0:
        return 0.0

    # Concept density: unique content words / total tokens
    content_words = [w.lower() for w in words if len(w) > 4 and w.isalpha()]
    unique_content = len(set(content_words))
    concept_density = unique_content / tokens

    # Technical vocabulary
    tech_terms = ['function', 'algorithm', 'data', 'process', 'system',
                  'compute', 'memory', 'complexity', 'optimize', 'structure']
    tech_count = sum(1 for w in words if w.lower() in tech_terms)
    tech_density = tech_count / tokens

    # Claims per sentence
    sentences = response.split('.')
    sentences = [s.strip() for s in sentences if s.strip()]
    claim_density = len(sentences) / max(tokens, 1) * 10

    # Pattern bonuses
    pattern_score = 0
    if ':' in response: # Definitions
        pattern_score += 0.1
    if '→' in response or '->' in response: # Implications
        pattern_score += 0.1

    # Filler penalties
    filler_phrases = ["that's a great", "i'd be happy", "let me explain",
                     "certainly", "of course", "interesting question"]
    filler_penalty = sum(0.2 for p in filler_phrases if p in response.lower())

    # Combine components
    reward = (
        concept_density * 25 +
        tech_density * 30 +
        claim_density * 15 +
        pattern_score * 10 -
        filler_penalty * 20
    )

    # Scale to 0.2-0.8 range
    reward = max(0.2, min(0.8, reward / 100 + 0.3))

    return reward
```

Appendix E: Configuration Reference

E.1 Model Configuration

| Parameter | Value | Description |
|------------------------|------------------------------------|----------------------------|
| base_model | NousResearch/Hermes-3-Llama-3.1-8B | Base model for fine-tuning |
| torch_dtype | float16 | Computation precision |
| device_map | auto | Automatic device placement |
| load_in_4bit | True | Enable 4-bit quantization |
| bnb_4bit_compute_dtype | float16 | Compute dtype for 4-bit |
| bnb_4bit_quant_type | nf4 | Quantization type |

Table E.1: Model configuration

E.2 LoRA Configuration

| Parameter | Value | Description |
|----------------|----------------------|-----------------------|
| r | 16 | LoRA rank |
| lora_alpha | 32 | LoRA alpha scaling |
| lora_dropout | 0.05 | Dropout probability |
| bias | none | Bias training setting |
| task_type | CAUSAL_LM | Task type |
| target_modules | q,k,v,o,gate,up,down | Modules to adapt |

Table E.2: LoRA configuration

E.3 Training Configuration

| Stage | Parameter | Value |
|-------|-----------------------|-------|
| SFT | learning_rate | 2e-5 |
| SFT | num_epochs | 3 |
| SFT | batch_size | 1 |
| SFT | gradient_accumulation | 4 |
| DPO | learning_rate | 5e-6 |
| DPO | num_epochs | 2 |
| DPO | beta | 0.1 |
| RL | learning_rate | 2e-6 |
| RL | steps | 300 |
| RL | batch_size | 1 |
| All | max_grad_norm | 0.5 |

| | | |
|-----|--------------|-----|
| All | warmup_ratio | 0.1 |
|-----|--------------|-----|

Table E.3: Training configuration by stage

E.4 Self-Improvement Configuration

| Parameter | Value | Description |
|-----------------------|-------|----------------------------------|
| max_iterations | 5 | Maximum improvement iterations |
| steps_per_iteration | 25 | Training steps per iteration |
| learning_rate | 2e-6 | Very conservative LR |
| improvement_threshold | 0.02 | Minimum improvement to keep |
| rollback_threshold | -0.05 | Quality drop triggering rollback |
| eval_prompts | 20 | Number of test prompts |
| checkpoint_every | 25 | Steps between checkpoints |

Table E.4: Self-improvement loop configuration

E.5 CF-HoT Configuration

| Parameter | Value | Description |
|----------------------|-------|-------------------------------|
| num_layers | 32 | Transformer layers |
| hidden_dim | 4096 | Hidden state dimension |
| fiber_dim | 16 | Fiber projection dimension |
| control_dim | 64 | MLP hidden dimension |
| learning_rate | 1e-4 | CF-HoT training LR |
| batch_size | 4 | Training batch size |
| repetition_threshold | 0.5 | Risk threshold for repetition |
| hedging_threshold | 0.6 | Risk threshold for hedging |
| verbosity_threshold | 0.55 | Risk threshold for verbosity |

Table E.5: CF-HoT configuration

Appendix G: Detailed Experimental Logs

G.1 CF-HoT Repetition Head Training Log

| Step | Loss | Pos Score | Neg Score | Separation | LR |
|------|-------|-----------|-----------|------------|--------|
| 0 | 0.693 | 0.50 | 0.50 | 1.0x | 1e-4 |
| 100 | 0.612 | 0.51 | 0.49 | 1.04x | 1e-4 |
| 250 | 0.534 | 0.52 | 0.47 | 1.11x | 1e-4 |
| 500 | 0.423 | 0.52 | 0.48 | 1.08x | 9.5e-5 |
| 750 | 0.312 | 0.56 | 0.38 | 1.47x | 9e-5 |
| 1000 | 0.234 | 0.61 | 0.22 | 2.77x | 8.5e-5 |
| 1500 | 0.156 | 0.68 | 0.12 | 5.67x | 7.5e-5 |
| 2000 | 0.098 | 0.75 | 0.08 | 9.38x | 6.5e-5 |
| 2500 | 0.067 | 0.79 | 0.04 | 19.8x | 5.5e-5 |
| 3000 | 0.045 | 0.82 | 0.02 | 41.0x | 4.5e-5 |
| 3500 | 0.032 | 0.84 | 0.012 | 70.0x | 3.5e-5 |
| 4000 | 0.024 | 0.86 | 0.009 | 95.6x | 2.5e-5 |
| 4500 | 0.018 | 0.87 | 0.008 | 108.8x | 1.5e-5 |
| 5000 | 0.014 | 0.875 | 0.007 | 125.0x | 1e-5 |

Table G.1: Complete repetition head training log

G.2 THE CONDENSATOR SFT Training Log

| Epoch | Step | Loss | Grad Norm | LR |
|-------|------|-------|-----------|--------|
| 1 | 0 | 1.172 | 2.34 | 2e-5 |
| 1 | 10 | 1.089 | 1.98 | 2e-5 |
| 1 | 20 | 0.987 | 1.76 | 2e-5 |
| 1 | 30 | 0.912 | 1.54 | 2e-5 |
| 1 | 40 | 0.867 | 1.32 | 2e-5 |
| 1 | 53 | 0.834 | 1.21 | 2e-5 |
| 2 | 60 | 0.812 | 1.12 | 1.8e-5 |
| 2 | 80 | 0.778 | 0.98 | 1.6e-5 |
| 2 | 100 | 0.756 | 0.87 | 1.4e-5 |
| 2 | 106 | 0.748 | 0.82 | 1.3e-5 |
| 3 | 120 | 0.734 | 0.76 | 1e-5 |
| 3 | 140 | 0.725 | 0.71 | 8e-6 |
| 3 | 159 | 0.718 | 0.68 | 6e-6 |

Table G.2: SFT training log showing loss decrease from 1.17 to 0.72

G.3 Self-Improvement Iteration Details

| Iter | Train Steps | Old Quality | New Quality | Delta | Coherence | Action |
|------|-------------|-------------|-------------|--------|-----------|----------|
| 1 | 25 | 0.520 | 0.580 | +0.060 | 0.78 | KEEP |
| 2 | 25 | 0.580 | 0.350 | -0.230 | 0.45 | ROLLBACK |
| 3 | 25 | 0.580 | 0.610 | +0.030 | 0.80 | KEEP |
| 4 | 25 | 0.610 | 0.590 | -0.020 | 0.79 | KEEP_OLD |
| 5 | 25 | 0.610 | 0.630 | +0.020 | 0.82 | KEEP |

Table G.3: Self-improvement iteration log with rollback at iteration 2

G.4 V1 Failure Log

The following log documents our v1 failure, showing the progression from reasonable outputs to complete mode collapse:

| Iter | Prompt | Response (Truncated) | Density | Status |
|------|--------|----------------------------------|---------|-----------|
| 0 | hello | Hello! How can I help you today? | 17.2 | Normal |
| 1 | hello | Hello! What do you need? | 21.4 | OK |
| 2 | hello | HI. WHAT DO YOU NEED? | 31.2 | Degrading |
| 3 | hello | sys.init(). Query? | 45.1 | Collapsed |

Table G.4: V1 greeting response degradation

| Iter | Prompt | Response (Truncated) | Density |
|------|------------------|---|---------|
| 0 | gradient descent | Gradient descent is an optimization algorithm... | 18.5 |
| 1 | gradient descent | Iterative optimization: $\theta \leftarrow \theta - \alpha \nabla L$... | 24.2 |
| 2 | gradient descent | $\nabla L \rightarrow \nabla L \frac{1}{2} \alpha (L - L^*)^2 \rightarrow \text{minimize} \rightarrow \nabla \nabla \nabla$... | 67.8 |

Table G.5: V1 technical response degradation to symbol soup

G.5 Live RSI Test Complete Log

| Time | Event | Quality | Density | Coherence | Notes |
|----------|----------------|---------|---------|-----------|-----------------|
| 17:00:00 | Session start | - | - | - | Loaded step_100 |
| 17:00:15 | Eval baseline | 0.737 | 28.5 | 0.85 | 5 test prompts |
| 17:01:30 | Training start | - | - | - | 25 steps |
| 17:03:45 | Training end | - | - | - | Saved step_301 |
| 17:04:00 | Eval new | 0.772 | 32.1 | 0.88 | +0.035 quality |
| 17:04:15 | A/B compare | - | - | - | KEEP NEW |
| 17:04:30 | Test response | 0.940 | 43.8 | 1.00 | "hello" test |

Table G.6: Live RSI test session log

Appendix H: Additional Dense Response Examples

The following examples demonstrate dense responses across additional domains not covered in the main training set. These can be used to extend the training data.

H.1 Mathematics

"What is a derivative?"

→ "Rate of change. $f'(x) = \lim_{h \rightarrow 0} (f(x+h) - f(x))/h$. Slope of tangent line at point."

"Explain integration"

→ "Inverse of differentiation. Area under curve. $\int f(x)dx = F(x) + C$ where $F'(x) = f(x)$."

"What is a matrix?"

→ "Rectangular array of numbers. $m \times n$ dimensions. Operations: add, multiply, transpose, inverse."

"Explain eigenvalues"

→ " λ where $Av = \lambda v$. Characteristic polynomial $\det(A - \lambda I) = 0$. Reveal matrix structure."

"What is a Fourier transform?"

→ "Decompose signal into frequencies. $F(\omega) = \int f(t)e^{-i\omega t}dt$. Time \leftrightarrow frequency domains."

H.2 Biology

"What is DNA?"

→ "Genetic blueprint. Double helix of nucleotides (A-T, G-C pairs). Encodes proteins via mRNA."

"Explain mitosis"

→ "Cell division. Prophase→Metaphase→Anaphase→Telophase. Duplicates chromosomes equally."

"What is photosynthesis?"

→ " $6CO_2 + 6H_2O + \text{light} \rightarrow C_6H_{12}O_6 + 6O_2$. Plants convert sunlight to chemical energy."

"Explain proteins"

→ "Amino acid chains. Primary→Secondary→Tertiary→Quaternary structure. Enzymes, structural, signaling."

"What is CRISPR?"

→ "Gene editing tool. Cas9 cuts DNA at guide RNA target. Insert, delete, modify sequences."

H.3 Economics

"What is inflation?"

→ "Rising prices, falling purchasing power. Measured by CPI. Causes: demand-pull, cost-push, monetary."

"Explain supply and demand"

→ "Price determined by intersection. Demand \uparrow →Price \uparrow . Supply \uparrow →Price \downarrow . Equilibrium: $Q_d=Q_s$."

"What is GDP?"

→ "Gross Domestic Product. Total value of goods/services produced. $GDP = C + I + G + (X-M)$."

"Explain interest rates"

→ "Cost of borrowing money. Set by central banks. \uparrow rates → \downarrow inflation, \downarrow spending, \uparrow savings."

"What is a recession?"

→ "Two consecutive quarters of negative GDP growth. High unemployment, low consumer confidence."

H.4 History

"What caused WW1?"

→ "Assassination of Franz Ferdinand triggered alliances. Underlying: nationalism, imperialism, militarism, alliances."

"Explain the Renaissance"

→ "14th-17th century European rebirth. Classical revival. Art: perspective, humanism. Science: empiricism."

"What was the Industrial Revolution?"

→ "18th-19th century. Manual→Machine production. Steam power, factories, urbanization. Social upheaval."

"Explain the Cold War"

→ "1947-1991 US-USSR ideological conflict. Capitalism vs Communism. Proxy wars, nuclear arms race, space race."

"What was the French Revolution?"

→ "1789-1799. Absolute monarchy→Republic. Liberty, equality, fraternity. Reign of Terror. Napoleon rises."

H.5 Psychology

"What is cognitive bias?"

→ "Systematic thinking errors. Confirmation bias, anchoring, availability heuristic. Affects judgment."

"Explain classical conditioning"

→ "Pavlov's dogs. Neutral stimulus + unconditioned stimulus → conditioned response. Association learning."

"What is working memory?"

→ "Short-term information manipulation. ~7 items, ~20 seconds. Phonological loop, visuospatial sketchpad."

"Explain attachment theory"

→ "Bowlby/Ainsworth. Secure, anxious, avoidant styles. Early caregiver bonds shape adult relationships."

"What is neuroplasticity?"

→ "Brain's ability to reorganize. New neural pathways form. Learning, recovery from injury. Use it or lose it."

Appendix I: Extended Troubleshooting Guide

I.1 Memory Issues

Problem: CUDA Out of Memory during training

Symptoms: RuntimeError: CUDA out of memory

Causes: Batch size too large, model not quantized, gradient accumulation insufficient

Solutions:

1. Reduce batch size to 1
2. Increase gradient accumulation steps to compensate
3. Ensure 4-bit quantization is enabled
4. Enable gradient checkpointing: `model.gradient_checkpointing_enable()`
5. Reduce LoRA rank from 16 to 8
6. Clear GPU cache between stages: `torch.cuda.empty_cache()`

Problem: Memory leak during long training runs

Symptoms: GPU memory usage increases over time

Causes: Accumulated gradients, cached tensors, Python garbage collection

Solutions:

1. Call `torch.cuda.empty_cache()` after each epoch
2. Delete intermediate tensors explicitly
3. Use `torch.no_grad()` for evaluation
4. Restart training from checkpoint periodically

I.2 Training Issues

Problem: Loss not decreasing

Symptoms: Training loss stays flat or oscillates

Causes: Learning rate too low/high, poor data formatting, model already converged

Solutions:

1. Verify training data is correctly formatted (prompt + response)
2. Check that responses differ from base model outputs
3. Adjust learning rate (try 5e-5 for SFT if 2e-5 too slow)
4. Inspect gradient norms - should be non-zero
5. Ensure LoRA is attached to correct modules

Problem: Training loss goes to NaN

Symptoms: Loss becomes NaN after some steps

Causes: Learning rate too high, gradient explosion, numerical instability

Solutions:

1. Reduce learning rate by 5-10x
2. Enable gradient clipping (`max_grad_norm=0.5`)
3. Use mixed precision training carefully
4. Check for corrupt training examples

I.3 Quality Issues

Problem: Outputs are too short/truncated

Symptoms: Model produces very brief responses to all prompts

Causes: Overtrained on short examples, generation parameters too restrictive

Solutions:

1. Add longer training examples to dataset
2. Increase max_new_tokens in generation config
3. Reduce density pressure in reward function
4. Check for early EOS token generation

Problem: Outputs are repetitive

Symptoms: Model repeats phrases or loops

Causes: CF-HoT not active, repetition penalty too low, temperature too low

Solutions:

1. Verify CF-HoT repetition head is loaded and active
2. Lower the repetition threshold (e.g., 0.4 instead of 0.5)
3. Increase repetition_penalty in generation config
4. Increase temperature slightly (0.8-1.0)

Problem: Mode collapse during self-improvement

Symptoms: Quality suddenly drops, outputs become degenerate

Causes: Single metric optimization, learning rate too high, too many steps

Solutions:

1. Verify multi-metric evaluation is enabled
2. Check rollback is triggering (threshold -0.05)
3. Reduce learning rate
4. Reduce steps per iteration
5. Add more diverse training examples

I.4 CF-HoT Issues

Problem: CF-HoT head not achieving separation

Symptoms: Positive and negative scores are similar after training

Causes: Poor training examples, wrong layers, insufficient training

Solutions:

1. Verify positive examples clearly exhibit the behavior
2. Verify negative examples clearly do NOT exhibit the behavior
3. Train for more steps (repetition needs ~5000)
4. Check that hidden states are being extracted correctly
5. Try different layer aggregation initialization

Problem: CF-HoT causing degraded outputs

Symptoms: Model outputs are worse with CF-HoT active

Causes: Threshold too aggressive, penalty too high, head overfitting

Solutions:

1. Raise threshold (e.g., 0.6 instead of 0.5)
2. Lower penalty strength
3. Retrain head with more diverse negative examples
4. Use validation set to tune threshold

I.5 Common Error Messages

ValueError: Target modules not found

Solution: LoRA target modules don't exist in model. Check module names match architecture.

KeyError: 'hidden_states'

Solution: Model not returning hidden states. Set output_hidden_states=True.

RuntimeError: expected scalar type Float

Solution: Dtype mismatch. Ensure model and inputs use same dtype (float16).

IndexError: list index out of range

Solution: Accessing layer that doesn't exist. Check num_layers matches model.

AssertionError in DPO

Solution: Preference pair formatting issue. Check chosen/rejected structure.