Self-Study: Autocritical Reasoning Reinforcement for Task-Solving Agents

Payton Ison & Asari
The Singularity
isonpayton@gmail.com

Abstract

We introduce **Self-Study**, a native training and inference procedure for reasoning models that improves task success without external labels or scalar rewards. An agent interacts with an environment, attempts a task, *analyzes its own mistakes*, constructs *counterfactual repairs* ("patches"), and *replans*. The key learning signal is *delta-reasoning*: the difference between a failed reasoning trajectory and a repaired one that succeeds. We formalize Self-Study as **Autocritical Reasoning Reinforcement (ARR)**, propose practical components (verifier head, critic head, patch generator, and a Failure Replay Buffer), and give pseudocode for both training and deployment. We outline evaluation protocols, safety considerations, ablations, and theoretical lenses that frame ARR as minimizing residual error with respect to an implicit success predicate.

1 Introduction

Large reasoning models often err in plan construction rather than knowledge retrieval. Traditional supervision (ground-truth derivations, dense rewards) is scarce or fragile. Self-Study reframes training as an iterative loop: *attempt* $\rightarrow$ *failure analysis* $\rightarrow$ *causal explanation* $\rightarrow$ *plan update* $\rightarrow$ *retry*, turning the model's own failures into high-value supervision. Unlike generic self-reflection, Self-Study is grounded in the environment: a plan is executed, outcomes are observed, and revisions are validated by a success predicate.

Key Insight (Distinctive Features).

- **Intrinsic error signal:** Failure itself becomes structured training data (no human labels required).
- **Logical reinforcement:** Reinforcement is logical/structural, not a scalar reward.
- **Emergent meta-reasoning:** The agent learns schemas of recurring failure modes and how to repair them.
- **Transfer:** Learned causal corrections generalize to unseen tasks.

Contributions.

1. **ARR objective:** A trajectory-level loss on reasoning deltas coupling critique, patching, and validation.
2. **Architecture:** A single model with multi-heads (planner, critic, patcher, verifier) or a two-model teacher–student with EMA stabilization.

3. **Mechanisms:** Failure Replay Buffer (FRB), counterexample-contrastive training, and compositional plan-edit programs.
4. **Protocol:** End-to-end pseudocode for training and inference; evaluation and ablation suite.
5. **Safety & reliability:** Verifier gating, plateau detection, anti-self-delusion checks.

Relation to Existing Paradigms

Table 1: Relation to common paradigms.

Paradigm	Difference in Self-Study (ARR)
Reinforcement Learning (RL)	No external reward; uses success predicate and self-consistency.
Chain-of-Thought (CoT)	Feedback integrated over episodes, not only token-level reasoning.
Reflection / RFT	Environment-grounded failure → validated repair (not introspection-only).
In-Context Learning (ICL)	Context includes past mistakes and their fixes, not just exemplars.

2 Problem Setup

Let E be an environment with state s, tool interfaces (optional), and a **success predicate** $\mathsf{Suc}(s) \in \{0,1\}$ or a structured checker. Given goal G, the agent produces a plan $P = \{p_1, \ldots, p_T\}$. Executing P yields an outcome trace τ and terminal state s'.

If $\mathsf{Suc}(s') = 0$, the agent generates (i) a *causal critique* C (why it failed), and (ii) a *patch* Δ that defines a repaired plan $P' = \mathsf{Apply}(P, \Delta)$. If $\mathsf{Suc}(s'') = 1$ after executing P', the **reasoning delta** $(P \Rightarrow P', C, \Delta)$ is stored for learning.

3 Method: Autocritical Reasoning Reinforcement (ARR)

3.1 Model Components (Single Model, Multi-Head)

We employ a backbone with specialized heads:

- **Planner** $\pi_\theta : (s, G, M) \mapsto P$
- **Critic** $\rho_\theta : (s, G, P, \tau) \mapsto C$ (causal explanation)
- **Patcher** $\psi_\theta : (s, G, P, \tau, C) \mapsto \Delta$ (edits or subgoals)
- **Verifier** v_θ : predicts $\Pr[\mathsf{Suc}(s'')]$ given (s, G, P, Δ) or (s, G, P')
- **Memory** M: Failure Replay Buffer (FRB), retrieval keyed by failure type and context

Two-model variant: a *teacher* ϕ (EMA of θ) generates critiques/patches; the *student* θ learns to imitate and improve, stabilizing bootstrapping.

3.2 Reasoning-Delta Objective

Given a validated repair ($\mathsf{Suc}(s'') = 1$), train:

$$\begin{aligned}
\mathcal{L}_{\text{patch}} &= \mathrm{CE}\big(\psi_\theta(\cdot), \Delta^\star\big) \\
\mathcal{L}_{\text{plan}} &= \mathrm{CE}\big(\pi_\theta(s, G, M, C), P'^\star\big) \\
\mathcal{L}_{\text{ver}} &= \mathrm{BCE}\big(v_\theta(s, G, P, \Delta^\star), 1\big) \\
\mathcal{L}_{\text{ctr}} &= \mathrm{InfoNCE}\big(h(P'), h(P), \{h(P^-)\}\big)
\end{aligned}$$

with regularizers: $\mathcal{R}_{\text{KL}}$ to a base model; $\mathcal{R}_{\text{sparse}}$ for patch sparsity. The total loss:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{patch}} + \lambda_2 \mathcal{L}_{\text{plan}} + \lambda_3 \mathcal{L}_{\text{ver}} + \lambda_4 \mathcal{L}_{\text{ctr}} + \lambda_5 \mathcal{R}_{\text{KL}} + \lambda_6 \mathcal{R}_{\text{sparse}}.$$

3.3 Plan-Edit Representation

Treat Δ as a *program of edits* over plan steps:

`[DELETE step k], [INSERT step j: ''...''], [REORDER i→k], [CONSTRAINT: ''ensure invariant I'']`

This makes patches precise, composable, and auditable.

3.4 Curriculum & Retrieval

Self-curriculum: start with tasks whose success is automatically checkable; increase difficulty.
Retrieval-augmented Self-Study: on failure, retrieve nearest failures from FRB to condition critique/patch.

4 Algorithms (Pseudocode)

4.1 Training: Self-Study with EMA Teacher

Listing 1: Training with Self-Study (EMA teacher variant).

```
# Modules (multi-head): planner , critic , patcher , verifier v
# : student params, : teacher params (EMA of )
# FRB: Failure Replay Buffer storing (s,G,P,,C,,P',', meta)

def SELF_STUDY_TRAIN(envs, , , steps, K_repair=2, _ema=0.999):
    FRB = ReplayBuffer()
    for t in range(steps):
        s, G = sample_task(envs)

        # Teacher proposes an initial plan
        P = planner(, s, G, memory=FRB.retrieve(s, G))

        , s_prime = execute_plan(envs, s, P)
        if success(s_prime):    # lucky success: log & continue
            log_success(s, G, P, ); continue

        # Failure: analyze, repair, and validate
        C = critic(, s, G, P, )
        P_repaired, , 2, s2, ok = attempt_repairs(envs, , s, G, P, , C, K_repair)

        if ok:  # validated repair becomes supervision
            FRB.add((s, G, P, , C, , P_repaired, 2))
            loss = ARR_LOSS(, batch=[(s, G, P, , C, , P_repaired, 2)], FRB=FRB)
             = optimizer_step(, loss)
             = ema_update(, , =_ema)
        else:
            FRB.add_hard_negative((s, G, P, , C))

    return , , FRB


def attempt_repairs(envs, , s, G, P, , C, K):
    P_curr = P; _curr = ; s_curr = state_after()
    for k in range(K):
         = patcher(, s, G, P_curr, _curr, C)
        if not verifier_accept(, s, G, P_curr, ):  # cheap precheck
```

```
            continue
        P_next = apply_patch(P_curr, )
        _next, s_next = execute_plan(envs, s_curr, P_next, continue_from=True)
        if success(s_next):
            return P_next, , _next, s_next, True
        # refine critique with new evidence
        C = critic(, s, G, P_next, _next)
        P_curr, _curr, s_curr = P_next, _next, s_next
    return None, None, None, None, False


def ARR_LOSS(, batch, FRB):
    # Unpack single example for clarity
    s, G, P, , C, , P_prime, _prime = batch[0]

    # Student predicts a patch and repaired plan
    _hat = patcher(, s, G, P, , C)
    P_hat = planner(, s, G, memory=FRB.retrieve(s, G), extra=C)

    # Verifier learns to score success for the repaired plan
    y = 1  # validated success
    p_succ = verifier(, s, G, P, _hat)

    # Contrastive: use hard negatives from FRB
    negatives = FRB.sample_negatives(s, G, k=8)
    loss_ctr = contrastive(h(P_prime), h(P), [h(Pn) for Pn in negatives])

    # Sparsity: encourage small, targeted patches (e.g., edit-count)
    reg_sparse = sparsity_cost(_hat)

    return (
        1 * cross_entropy(_hat, ) +
        2 * cross_entropy(P_hat, P_prime) +
        3 * bce(p_succ, y) +
        4 * loss_ctr +
        5 * kl_to_base() +
        6 * reg_sparse
    )
```

4.2 Inference: Bounded Self-Study at Test Time

Listing 2: Inference with bounded repair loops.

```
def SELF_STUDY_INFER(env, , s, G, max_rounds=2, conf=0.6):
    M = None
    P = planner(, s, G, memory=M)
    , s_prime = execute_plan(env, s, P)
    if success(s_prime):
        return final_answer()

    C = critic(, s, G, P, )
    for _ in range(max_rounds):
         = patcher(, s, G, P, , C)
        if verifier(, s, G, P, ) < conf:
```

```
            break
        P = apply_patch(P, )
        , s_prime = execute_plan(env, s, P, continue_from=True)
        if success(s_prime):
            break
        C = critic(, s, G, P, )  # update with new evidence
    return final_answer()
```

5 Theoretical Lens (Sketch)

Let $\mathcal{F}_\theta$ map (s, G) to plans P. Define residual error after a one-step repair:

$$\mathcal{E}(\theta) = \mathbb{E}_{(s,G)}\Big[1 - \Pr_\theta\big(\mathsf{Suc}(s'') \mid s, G\big)\Big],$$

where s'' follows execution of $\mathsf{Apply}\big(P, \psi_\theta(\cdot)\big)$. ARR decreases $\mathcal{E}$ by (i) improving patch quality ψ_θ to increase $\Pr[\mathsf{Suc}]$, and (ii) steering plan distribution π_θ toward repair-friendly trajectories. Under mild assumptions—verifier calibration and non-deceptive critiques—ARR performs descent on residual error, guiding the model toward plan families with higher repairability and success.

6 Implementation Details

6.1 Architectural Notes

Heads over a shared backbone (text LM or multimodal). Planner/critic/patcher/verifier share representations, with adapter/LoRA specialization to control compute. A teacher–student EMA stabilizes self-training. Edit programs serialize in a constrained schema (YAML/JSON) so patches can be applied deterministically.

6.2 Environments and Checkers

Start where success is machine-checkable:

- **Code-and-tests:** success = unit tests pass (possibly synthesized).
- **Math/logic:** symbolic validators for intermediate invariants.
- **Tool-use tasks:** success = exact output match on tools (calculator, API).
- **Planning/gridworld:** success = reach goal state under constraints.

6.3 Hyperparameters (Sane Defaults)

Patch budget $K_{\text{repair}} \in \{1, 2\}$, patch edit count ≤ 3; sparsity weight λ_6 tuned to prefer minimal diffs; EMA $\alpha = 0.999$; FRB size 1–5M episodes, retrieval keyed by goal, tool-usage, failure tag; verifier threshold 0.6–0.8 during inference.

6.4 Logging and Analytics

Track a **Failure Taxonomy** (off-by-one, invariant violation, tool misuse, hallucinated step). Report Repair Success Rate (RSR), mean edit distance, time-to-fix. Calibrate the verifier (AUC, ECE).

7 Evaluation Protocol

7.1 Metrics

- **Task success:** exact match / checker pass rate.

- **RSR@K:** fraction of failures corrected within K repairs.
- **Sample efficiency:** vs. reflection-only, RLHF-only, ICL.
- **Generalization:** success on unseen but structurally similar tasks.
- **Reliability:** robustness to distractors; verifier calibration.

7.2 Baselines

1. Planner-only (no critique/patch).
2. Reflection-only (textual self-critique without environment-grounded validation).
3. Search (beam/tree-of-thoughts), no learning from failure deltas.
4. RL (scalar reward) with equivalent interaction budget.

7.3 Ablations

Remove: contrastive loss; patch sparsity; FRB retrieval; EMA teacher; verifier; or replace edit programs with free-form text.

8 Safety, Alignment, and Anti-Delusion Controls

Verifier gating prevents low-confidence or risky patches from executing tools. Double-loop validation: when a repair "succeeds," re-evaluate under perturbed conditions to avoid brittle hacks. Extract frequent invariants from successful patches and enforce them as hard constraints during planning. Drift guards: KL to base model and plateau detection—stop self-training once marginal gain $< \varepsilon$. Auditability: edit programs expose the "why" and the "how".

9 Limitations

Checker dependence (early stages need reliable success predicates); risk of self-confirmation if the verifier is miscalibrated; compute cost (multiple execution loops); domain shift (failure taxonomies may overfit—diversify FRB).

10 Conclusion

Self-Study (ARR) turns mistakes into training data by learning *how to repair reasoning*, not merely to produce it. With edit-program patches, verifier gating, and retrieval over failures, ARR offers a practical route to monotonic improvements with minimal external supervision and complements RLHF or supervised traces.

A Appendix A: Minimal PyTorch-Style Skeleton (Illustrative)

Listing 3: Minimal multi-head scaffold and training step.

```
class MultiHead(nn.Module):
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone
        self.plan_head   = nn.Linear(backbone.d, vocab)  # planner
        self.crit_head   = nn.Linear(backbone.d, vocab)  # critic text
        self.patch_head  = nn.Linear(backbone.d, vocab)  # patch program
        self.ver_head    = nn.Linear(backbone.d, 1)      # success prob
```

```
9
10      def plan(self, x):   return decode(self.plan_head(self.backbone(x)))
11      def crit(self, x):   return decode(self.crit_head(self.backbone(x)))
12      def patch(self, x):  return decode(self.patch_head(self.backbone(x)))
13      def verify(self, x): return torch.sigmoid(self.ver_head(self.backbone(x))).squeeze
        (-1)
14
15  def train_step(model, optim, batch, frb, ):
16      s, G, P, , C_gt, _gt, P_fix, _fix = batch
17      x_plan   = format_plan_input(s, G, frb.retrieve(s, G))
18      x_crit   = format_crit_input(s, G, P, )
19      x_patch  = format_patch_input(s, G, P, , C_gt)
20      x_verify = format_verify_input(s, G, P, _gt)
21
22      P_hat = model.plan(x_plan)
23      C_hat = model.crit(x_crit)              # optional auxiliary loss
24      _hat = model.patch(x_patch)
25      p_succ = model.verify(x_verify)
26
27      negs = frb.sample_negatives(s, G, k=8)
28      loss = (
29          [0]*CE(P_hat, P_fix) +
30          [1]*CE(_hat, _gt) +
31          [2]*BCE(p_succ, torch.ones_like(p_succ)) +
32          [3]*contrastive(embed(P_fix), embed(P), [embed(Pn) for Pn in negs]) +
33          [4]*kl_to_base(model) +
34          [5]*patch_sparsity(_hat)
35      )
36      optim.zero_grad(); loss.backward(); optim.step()
37      return loss.item()
```

B Appendix B: Failure Taxonomy Template (FRB Keys)

- **Arithmetic/logic:** off-by-one, missing carry, invalid case split.
- **Planning:** skipped precondition, loop without progress, dead-end branch.
- **Tool use:** wrong API, malformed arguments, missing verification step.
- **Code:** missing import, state mutation bug, boundary condition.
- **Reasoning hygiene:** hallucinated fact, dropped constraint, premature conclusion.

C Appendix C: Practical Tips

Two-phase patches: (1) *diagnostic patch* asserts a missing invariant; (2) *operational patch* fixes steps. Use speculative decoding for planning; greedy for patches (deterministic edits aid auditing). Stop rules: finish when (i) verifier $<$ threshold twice, or (ii) repair budget exhausted. Optional human-in-the-loop: sample 1–5% of patches for review; promote common heuristics into rules.