

Contents

1 Goal-Conditioned Robotic Manipulation: A Research Platform	1
1.1 Project Status	2
1.2 §1. Problem Formulation	3
1.3 §2. Methodology	3
1.3.1 Why Actor-Critic?	3
1.3.2 Why Off-Policy?	4
1.3.3 Why Hindsight Experience Replay?	5
1.3.4 Why Maximum Entropy (SAC)?	5
1.3.5 Summary: The Derived Method	5
1.4 §3. Repository Structure	6
1.5 §4. Prerequisites	6
1.6 §5. Quick Start	6
1.7 §6. Training Examples	7
1.8 §7. Evaluation Protocol	7
1.9 §8. Gymnasium-Robotics Fetch Environments	8
1.10§9. Design Principles	8
1.11§10. Common Failure Modes	9
1.12§11. Curriculum: A Step-by-Step Study Plan	9
1.13§12. Contributing	10
1.14§13. Scope and Limitations	10
1.14.1 Why HER?	10
1.14.2 What This Curriculum Does Not Cover	11
1.14.3 When HER Struggles	11
1.14.4 Next Steps After This Curriculum	12
1.15§14. References	12
1.16§15. License	12

1 Goal-Conditioned Robotic Manipulation: A Research Platform

A Docker-first reinforcement learning laboratory for studying sparse-reward manipulation tasks.

Robots that fold laundry. Arms that pack warehouse boxes. Hands that assemble electronics. These aren't science fiction--they're active research problems, and they're *harder* than the ML tasks that dominate headlines.

Why harder? Consider the differences:

Aspect	Language Models	Robotic Manipulation
Action space	Discrete tokens (~100k vocabulary)	Continuous \mathbb{R}^4 (infinite)
Feedback	Next-token prediction (dense, immediate)	Sparse binary (success/failure at episode end)
Error cost	Regenerate, backtrack, try again	Irreversible: collisions, drops, damage
Stability requirement	Errors cascade in context but are recoverable	Asymptotic stability: physical performance converges over time

That last point--**asymptotic stability**--is what control theorists worry about and ML practitioners often miss. The difference isn't just about reversibility; it's **structural**, rooted in dimensionality.

The mathematical crux: In discrete token spaces, you're either at token A or token B--there's no "infinitesimally wrong" token. Errors are quantized. In continuous \mathbb{R}^n , perturbations can be arbitrarily small, and critically, **small errors can grow** through the dynamics. This is where Lyapunov stability theory provides the right framework.

The intuition (borrowing from continuous-time control, though RL operates in discrete time): a system is **Lyapunov stable** if small perturbations stay small, and **asymptotically stable** if perturbations decay back to equilibrium. The classical tool is a Lyapunov function $V(x)$ --an "energy-like" quantity that decreases along trajectories, certifying convergence.

A learned policy, combined with environment dynamics, induces a dynamical system on state space. For reliable manipulation, this system should form a **stable attractor** around goal configurations. This structural requirement doesn't arise naturally in discrete sequence models--the action space topology is fundamentally different, and the notion of "small perturbations growing through dynamics" doesn't transfer directly.

Physical irreversibility compounds this: not only can errors grow, but their consequences persist. The robot collided, the object fell, the glass shattered. You can't backtrack physics.

The core challenge: learning to manipulate objects to arbitrary goal positions from minimal feedback, while maintaining stability in a continuous, physical world.

This repository is structured for step-by-step study. A 10-week curriculum (`syllabus.md`) provides executable commands and verification criteria. Tutorials (`tutorials/`) provide theoretical context. The goal is not to run scripts quickly, but to understand deeply.

1.1 Project Status

This is an active project. The table below shows what is implemented versus planned.

Prefer PDFs? See `docs/pdfs/README.pdf` and `docs/pdfs/tutorials/` (auto-generated on push to main).

Component	Status	Notes
Infrastructure		
Docker tooling (<code>docker/</code>)	✓ Complete	<code>dev.sh</code> , <code>build.sh</code> , <code>run.sh</code> , Dockerfile
Training CLI (<code>train.py</code>)	✓ Complete	PPO/SAC/TD3, HER, vectorized envs
Evaluation CLI (<code>eval.py</code>)	✓ Complete	Multi-seed, metrics, JSON output
Curriculum		
Week 0: DGX setup, proof-of-life	✓ Complete	<code>ch00_proof_of_life.py</code>
Week 1: Environment anatomy	✓ Complete	<code>ch01_env_anatomy.py</code>
Week 2: PPO on dense Reach	✓ Complete	<code>ch02_ppo_dense_reach.py</code> , 100% success
Week 3: SAC on dense Reach	✓ Complete	<code>ch03_sac_dense_reach.py</code> , 100% success
Week 4: Sparse + HER	○ Planned	
Week 5: PickAndPlace	○ Planned	
Week 6: Action-interface engineering	○ Planned	
Week 7: Robustness	○ Planned	
Week 8: Second suite	○ Planned	
Week 9: Sweeps and ablations	○ Planned	
Week 10: Capstone	○ Planned	

Current focus: Weeks 0-3 are complete. Week 3 validates the off-policy stack (SAC with replay buffer diagnostics) before adding HER for sparse rewards. Both PPO and SAC achieve

100% success on FetchReachDense-v4, establishing solid baselines for comparison. The core CLIs (`train.py`, `eval.py`) support the full curriculum; weekly scripts and tutorials are being developed incrementally.

1.2 §1. Problem Formulation

The Central Question. Can an agent learn to manipulate objects to arbitrary goal configurations using only sparse binary feedback?

This is not merely an engineering challenge. It is a fundamental question in sequential decision-making under uncertainty. The problem exhibits three critical characteristics that make it non-trivial:

1. **High-dimensional continuous control.** The action space is $\mathcal{A} \subset \mathbb{R}^4$ (3D Cartesian delta + gripper command). Standard discrete action methods do not apply.
2. **Sparse binary rewards.** The reward function $R(s, a, s', g) = \mathbf{1}[\|g_{\text{achieved}}(s') - g\| < \epsilon]$ provides no gradient signal until the goal is achieved. Random exploration fails catastrophically.
3. **Goal conditioning.** The agent must generalize across a distribution of goals $p(g)$, not merely solve a single task. This requires learning a universal policy $\pi : \mathcal{S} \times \mathcal{G} \rightarrow \Delta(\mathcal{A})$.

Definition (Goal-Conditioned MDP). A goal-conditioned Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{G}, P, R, \gamma)$ where:

- \mathcal{S} is the state space
- \mathcal{A} is the action space
- \mathcal{G} is the goal space
- $P : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$ is the transition kernel
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{G} \rightarrow \mathbb{R}$ is the goal-conditioned reward
- $\gamma \in [0, 1)$ is the discount factor

We seek $\pi^* = \arg \max_{\pi} \mathbb{E}_{g \sim p(g)} \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}, g) \right]$.

1.3 §2. Methodology

The choice of algorithm is not arbitrary. It follows from the problem structure. Let us trace the reasoning.

1.3.1 Why Actor-Critic?

The action space is continuous: $\mathcal{A} \subset \mathbb{R}^4$. This eliminates pure value-based methods like DQN, which require $\arg \max_a Q(s, a)$ —intractable when \mathcal{A} is continuous without discretization (which scales exponentially with dimension).

We need a method that:

- Outputs continuous actions directly (policy gradient)
- Uses value estimates to reduce gradient variance (critic)

This points to **actor-critic architectures**: a policy network (actor) produces actions; a value network (critic) evaluates them.

1.3.2 Why Off-Policy?

The reward is sparse: $R(s, a, s', g) = \mathbf{1}[\text{goal reached}]$. Most trajectories receive zero reward. If we discard these trajectories after one gradient step (as on-policy methods like PPO do), we waste the information they contain.

To understand why HER requires off-policy learning, we must first understand the on-policy/off-policy distinction.

On-Policy Learning (PPO, TRPO, A2C). On-policy methods learn from data collected by the *current* policy π_θ . The policy gradient theorem gives:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_t \nabla_\theta \log \pi_\theta(a_t | s_t) \cdot A^{\pi_\theta}(s_t, a_t) \right]$$

The expectation is over trajectories from π_θ . If we use data from an old policy $\pi_{\theta_{\text{old}}}$, the gradient estimate becomes biased. PPO mitigates this with importance sampling and clipping, but only for *small* policy changes. After a few gradient steps, the data becomes "stale" and must be discarded.

Consequence: On-policy methods cannot maintain replay buffers. Each batch of experience is used for a few updates, then thrown away.

Off-Policy Learning (SAC, TD3, DDPG). Off-policy methods learn from data collected by *any* policy—current, past, or even a different agent. They use Q-learning-style updates:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

The update target $r + \gamma Q(s', a')$ does not depend on which policy collected the transition (s, a, r, s') . This allows storing millions of transitions in a replay buffer and sampling them repeatedly.

Why HER Cannot Work with On-Policy Methods.

HER performs goal relabeling: after an episode ends, we take transitions (s_t, a_t, s_{t+1}, g) and create new transitions (s_t, a_t, s_{t+1}, g') where g' is a goal achieved later in the episode. This relabeling happens *after* the episode is collected.

The fundamental incompatibility:

1. **HER requires storing transitions.** Relabeling is a post-hoc operation on completed episodes. We must store the original transitions, compute achieved goals, select alternative goals, recompute rewards, and add relabeled transitions. This requires a replay buffer.
2. **On-policy methods cannot use replay buffers.** The moment data enters a buffer and is reused across multiple updates, the policy has changed, and the data no longer comes from π_θ . The policy gradient becomes biased.
3. **The timing is wrong.** On-policy methods update immediately after collecting a batch. HER needs the *entire episode* to know what goals were achieved (for the "future" strategy). By the time relabeling could happen, on-policy methods have already used and discarded the data.

Attempted Workarounds (and Why They Fail).

"What if we relabel before the policy update?" The "future" strategy requires knowing states that come *after* the current transition. We cannot relabel transition t until we know states $t+1, t+2, \dots, T$. By then, on-policy methods want to update immediately.

"What if we use importance sampling?" Importance sampling corrects for distribution mismatch, but the correction factor $\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ explodes when policies diverge. HER's relabeled data comes from many episodes ago; the importance weights become useless.

"What about on-policy HER variants?" Research has explored this (e.g., storing only recent episodes), but the sample efficiency gains of HER come precisely from *reusing* old experience. Restricting to recent data defeats the purpose.

Consequence: We use SAC or TD3, not PPO, for sparse-reward goal-conditioned tasks. PPO is included in this curriculum only for dense-reward baselines where HER is unnecessary.

1.3.3 Why Hindsight Experience Replay?

Consider a trajectory where the agent attempts goal g but fails, ending at state s_T with achieved goal $g' \neq g$. Under sparse rewards, this trajectory provides zero learning signal for goal g .

But the trajectory *does* demonstrate how to reach g' . If we relabel the transitions—replacing g with g' in the reward computation—the same trajectory now shows successful goal-reaching behavior.

This is the HER insight: **failed attempts at one goal are successful demonstrations for another goal**. By storing both original and relabeled transitions, we manufacture dense reward signal from sparse feedback.

Requirements for HER:

1. Goals must be explicit in the observation (Fetch environments provide `achieved_goal`, `desired_goal`)
2. Rewards must be recomputable for arbitrary goals (Fetch provides `compute_reward(achieved, desired, info)`)
3. The algorithm must be off-policy (relabeling happens in the replay buffer)

1.3.4 Why Maximum Entropy (SAC)?

Even with HER, exploration matters. The goal space is large; the agent must visit diverse regions to collect relabelable experience.

SAC adds an entropy bonus to the objective:

$$\pi^* = \arg \max_{\pi} \mathbb{E}_{\tau} \left[\sum_t \gamma^t (R(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))) \right]$$

This encourages stochastic policies that explore naturally, without requiring explicit exploration bonuses or schedules. The temperature α controls the exploration-exploitation trade-off.

1.3.5 Summary: The Derived Method

Problem Constraint	Methodological Requirement	Solution
Continuous actions	Direct policy output	Actor-critic
Sparse rewards	Sample reuse	Off-policy (replay buffer)
Goal-conditioned sparse rewards	Learn from failures	HER (goal relabeling)
Large goal space	Sustained exploration	Maximum entropy (SAC)

The method is SAC + HER. This is not a recipe chosen from a menu; it is a consequence of the problem structure.

Remark. PPO is included for dense-reward baselines where the above constraints are relaxed. But for sparse goal-conditioned tasks—the core focus of this curriculum—PPO cannot be combined with HER and struggles without reward shaping.

1.4 §3. Repository Structure

```
robotics/
  -- docker/          # Container definitions (Dockerfile, dev.sh, build.sh, run.sh)
  -- scripts/         # Versioned experiment scripts (ch00_proof_of_life.py, ch01_env_anatomy
    -- labs/          # From-scratch implementations for teaching (ppo_from_scratch.py, etc.)
  -- tutorials/       # Textbook-style documentation (chNN_<topic>.md)
  -- train.py         # Training CLI (PPO/SAC/TD3, HER, vectorized envs, TensorBoard)
  -- eval.py          # Evaluation CLI (metrics: success rate, return, smoothness)
  -- requirements.txt # Python dependencies (pinned versions)
  -- syllabus.md      # 10-week curriculum
```

Generated Artifacts (created as needed):

```
results/           # Evaluation JSON reports
checkpoints/       # Model files (.zip + .meta.json)
videos/            # Rendered episodes
runs/              # TensorBoard logs
```

1.5 §4. Prerequisites

Hardware Requirements.

- NVIDIA GPU with CUDA 12+ support (tested on DGX A100)
- Minimum 16GB GPU memory for parallel training (8 environments × batch size)

Software Requirements.

- Docker with NVIDIA Container Runtime
- Host OS: Linux (Ubuntu 22.04+ recommended)

Domain Knowledge Prerequisites.

- Reinforcement learning fundamentals (MDPs, policy gradients, Q-learning)
- Deep learning basics (neural networks, backpropagation, optimization)
- Python scientific computing (NumPy, PyTorch)
- Unix shell proficiency (Bash, docker commands)

Remark. This is not an introductory tutorial. Readers are expected to understand the difference between on-policy and off-policy algorithms, to recognize the temporal difference error $\delta_t = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$, and to debug PyTorch tensor shape mismatches independently.

1.6 §5. Quick Start

Step 1: Verify Container Environment.

```
bash docker/dev.sh
```

This creates .venv, installs dependencies, and drops you into an interactive shell. The environment is reproducible: same container, same packages, same results.

Step 2: Run Proof-of-Life.

```
bash docker/dev.sh python scripts/ch00_proof_of_life.py all
```

This executes an end-to-end smoke test:

- GPU availability (CUDA device check)
- Gymnasium-Robotics Fetch environment instantiation
- MuJoCo rendering (saves smoke_frame.png)
- PPO training on FetchReachDense-v4 (1000 timesteps)

Expected Output. The script should complete without errors and produce smoke_frame.png (a rendered frame) and ppo_smoke.zip (a checkpoint).

Step 3: Verify Reproducibility.

```
bash docker/dev.sh python scripts/check_repro.py --algo ppo --env FetchReachDense-v4 --tot
```

This trains the same configuration across multiple seeds and reports variance in final performance. High variance indicates training instability.

1.7 §6. Training Examples

Dense Reward (PPO Baseline).

```
bash docker/dev.sh python train.py \
  --algo ppo \
  --env FetchReachDense-v4 \
  --seed 0 \
  --n-envs 8 \
  --total-steps 1000000
```

Sparse Reward (SAC + HER).

```
bash docker/dev.sh python train.py \
  --algo sac \
  --her \
  --env FetchReach-v4 \
  --seed 0 \
  --n-envs 8 \
  --total-steps 1000000
```

Monitoring. Training logs are written to runs/ (TensorBoard format). View with:

```
bash docker/dev.sh tensorboard --logdir runs/
```

1.8 §7. Evaluation Protocol

Single Checkpoint Evaluation.

```
bash docker/dev.sh python eval.py \
  --ckpt checkpoints/ppo_FetchReachDense-v4_s0.zip \
  --env FetchReachDense-v4 \
```

```
--n-episodes 100 \
--deterministic \
--json-out results/eval_metrics.json
```

Multi-Seed Evaluation.

```
bash docker/dev.sh python eval.py \
--ckpt checkpoints/ppo_FetchReachDense-v4_s0.zip \
--env FetchReachDense-v4 \
--n-episodes 10 \
--seeds 0-9 \
--deterministic \
--json-out results/eval_multiseed.json
```

Reported Metrics.

- **Success Rate:** Fraction of episodes where $\|g_{\text{achieved}} - g_{\text{desired}}\| < \epsilon$
- **Mean Return:** Average cumulative discounted reward
- **Goal Distance:** Average final Euclidean distance to goal
- **Time to Success:** Mean timesteps until first success (for successful episodes)
- **Action Smoothness:** Variance of action deltas (lower = smoother)

Remark. A policy that achieves 95% success rate on a single seed but 60% on another seed has not solved the task reliably. Always report statistics over multiple seeds.

1.9 §8. Gymnasium-Robotics Fetch Environments

This repository uses the Fetch robotic arm tasks from Gymnasium-Robotics:

Environment	Task	Obs Dim	Goal Dim	Action Dim	Reward
FetchReach-v4	Move end-effector to target	10	3	4	Sparse
FetchReachDense-v4	Move end-effector to target	10	3	4	Dense
FetchPush-v4	Push block to target	25	3	4	Sparse
FetchSlide-v4	Slide puck to target	25	3	4	Sparse
FetchPickAndPlace-v4	Lift block to target	25	3	4	Sparse

Observation Structure. Dict with keys:

- **observation:** Proprioceptive state (joint positions, velocities, gripper state)
- **achieved_goal:** Current 3D position of manipulated object (or end-effector for Reach)
- **desired_goal:** Target 3D position

Action Space. $\mathcal{A} = [-1, 1]^4$, interpreted as $(dx, dy, dz, \text{gripper})$ where the first three components are Cartesian position deltas and the fourth controls gripper closure.

1.10 §9. Design Principles

Docker-First. Every command executes inside a container. No host-side package installation. No "it works on my machine" excuses. The `docker/dev.sh` script is the single entry point.

Reproducibility. All dependencies pinned in `requirements.txt`. All training runs seeded. All results reported with confidence intervals over multiple seeds.

Quantification. Claims require numbers. "The policy works" is inadmissible; "Success rate: $94.2\% \pm 2.1\%$ (5 seeds, 100 episodes each)" is acceptable.

Self-Contained Scripts. Production experiments live in `scripts/chNN_*.py`—versioned, runnable artifacts. Tutorials may include short, annotated code excerpts (via snippet-includes from `scripts/labs/`) that show how equations map to code, but runnable pipelines remain in scripts.

1.11 §10. Common Failure Modes

Problem: Training diverges after initial learning. **Diagnosis:** SAC entropy coefficient α too high. The policy remains stochastic when it should converge to deterministic goal-reaching.

Solution: Reduce `--sac-alpha` or enable automatic tuning.

Problem: HER provides no benefit over standard replay. **Diagnosis:** Goal relabeling strategy mismatched to environment. future strategy requires episodes that make progress toward *some* region of goal space. **Solution:** Verify goals are reachable. Check that achieved _goal updates during rollouts.

Problem: Reproducibility check fails (high variance across seeds). **Diagnosis:** Neural network initialization, environment randomness, or optimizer state not seeded correctly.

Solution: Verify `torch.manual_seed(seed)` and `env.reset(seed=seed)` are called.

Problem: Evaluation success rate ≪ training success rate. **Diagnosis:** Overfitting to training goal distribution, or stochastic policy used at eval time. **Solution:** Use `--deterministic` flag in `eval.py` and verify goal distribution matches training.

1.12 §11. Curriculum: A Step-by-Step Study Plan

This repository is designed for systematic study, not casual browsing. The file `syllabus.md` contains an executable 10-week curriculum with concrete commands, verification criteria, and "done when" checkpoints for each week.

The Progression.

Week	Focus	Key Milestone
0	DGX setup, containerization	Proof-of-life passes; reproducibility verified
1	Goal-conditioned environment anatomy	Understand dict observations, reward semantics
2	PPO on dense Reach	Stable baseline; evaluation protocol locked
3	SAC on dense Reach	Off-policy stack verified before adding HER
4	Sparse Reach + Push with HER	HER demonstrably outperforms no-HER baseline
5	PickAndPlace	Contacts/grasping; curriculum strategies
6	Action-interface engineering	Policies as controllers; smoothness metrics
7	Robustness	Noise injection, domain randomization, brittleness curves
8	Second suite (robosuite or Meta-World)	Avoid overfitting intuition to Fetch
9	Sweeps and ablations	Seed discipline; evidence-based conclusions
10	Capstone	Sparse PickAndPlace + robustness targets

The Structure of Each Week.

Every week in `syllabus.md` follows the same format:

- **Goal:** What you will accomplish and why it matters
- **Steps:** Exact commands to run (copy-paste ready)
- **Done when:** Unambiguous criteria for completion

This structure serves two purposes. First, it prevents self-deception: you cannot convince yourself you understand something if you cannot produce the specified artifacts. Second, it builds skills incrementally: Week 4 assumes Week 3 is complete; skipping ahead creates gaps that surface later as mysterious failures.

How to Use the Syllabus.

Start at Week 0. Complete each step. Verify each "done when" criterion. Only then proceed to the next week. If something fails, diagnose it before moving on. The curriculum is designed to surface problems early, when they are easier to fix.

The tutorials in `tutorials/` provide theoretical context for each chapter. The syllabus provides the executable path. Use both together.

1.13 §12. Contributing

Standards.

- All scripts must be executable via `bash docker/dev.sh python scripts/<name>.py`
- All new environments must include success rate benchmarks (5 seeds, 100 episodes)
- All documentation must follow WHY-HOW-WHAT structure
- All commits must pass the proof-of-life test (`scripts/ch00_proof_of_life.py all`)

Pull Request Template.

1. **Purpose:** What problem does this solve? (Problem formulation)
 2. **Approach:** What method is implemented? (Derivation or reference)
 3. **Verification:** What are the expected numerical outcomes?
 4. **Artifacts:** Which new files are created? Where are they documented?
-

1.14 §13. Scope and Limitations

This curriculum focuses on **Hindsight Experience Replay (HER)** as the primary method for sparse-reward goal-conditioned learning. This is a deliberate choice with trade-offs worth understanding.

1.14.1 Why HER?

Pedagogical clarity. HER has a single, elegant insight: failed trajectories contain information about achievable goals. This insight is easy to state, non-trivial to appreciate, and directly addresses *why* sparse rewards are hard. A student who understands HER understands goal relabeling in general.

Infrastructure maturity. Stable Baselines 3 provides well-tested HER integration with SAC and TD3. This lets us focus on understanding rather than debugging custom implementations.

Empirical relevance. HER remains a strong baseline for goal-conditioned manipulation. Many subsequent methods build on or compare against it. Understanding HER provides a foundation for understanding its extensions.

Scope constraints. A 10-week curriculum cannot cover everything. Depth on one method teaches more than superficial exposure to many.

1.14.2 What This Curriculum Does Not Cover

Intrinsic motivation and curiosity-driven exploration. Methods like Random Network Distillation (RND), Intrinsic Curiosity Module (ICM), and count-based exploration address sparse rewards through novelty bonuses rather than goal relabeling. These are complementary approaches with different assumptions.

- Reference: Curiosity-driven Exploration (Pathak et al., 2017)

Automatic curriculum and goal generation. Rather than relabeling goals post-hoc, these methods generate goals of appropriate difficulty during training. Asymmetric self-play and goal GAN approaches fall here.

- Reference: Automatic Goal Generation (Florensa et al., 2018)

Learning from demonstrations. Behavioral cloning, GAIL, and demo-augmented RL can bootstrap learning, especially when exploration is dangerous or expensive. These methods change the problem setup (they assume access to demonstrations).

- Reference: Overcoming Exploration with Demos (Nair et al., 2018)

Model-based reinforcement learning. Learning a world model enables planning and imagination-based training. This can improve sample efficiency but introduces model error as a new failure mode.

- Reference: World Models (Ha & Schmidhuber, 2018)

Hierarchical RL and skill primitives. Decomposing long-horizon tasks into reusable skills addresses a limitation of flat policies. HER struggles with very long horizons; hierarchy is one response.

- Reference: Data-Efficient Hierarchical RL (Nachum et al., 2018)

Vision-based control. This curriculum uses proprioceptive state. Pixel observations introduce representation learning challenges that compound the RL difficulties.

- Reference: SAC-AE (Yarats et al., 2019)

1.14.3 When HER Struggles

HER is not universally applicable. It assumes:

1. **Explicit goal representation.** Goals must be represented in observation space with a computable achieved_goal function. Tasks without clear goal structure (e.g., "move gracefully") don't fit.
2. **Off-policy learning.** HER requires replay buffers; it cannot be combined with on-policy methods like PPO.
3. **Moderate horizon length.** For very long episodes, the relabeled goals may be too easy (nearby states) or the credit assignment too diffuse.
4. **Goal-independent dynamics.** The transition function should not depend on the goal. (This holds for Fetch tasks but not universally.)

1.14.4 Next Steps After This Curriculum

For readers who complete the 10-week curriculum and wish to continue:

Immediate extensions (within the same framework):

- TD3+HER comparison with SAC+HER
- Goal selection strategies beyond "future" (episode, final, random)
- Prioritized experience replay combined with HER

Complementary methods (new problem formulations):

- Add demonstrations: collect 10-20 expert trajectories and use demo-augmented HER
- Add curiosity: combine RND bonus with HER for exploration in larger goal spaces
- Add hierarchy: implement a two-level policy where the high level sets subgoals

New domains (test generalization of understanding):

- Shadow Hand manipulation (higher-dimensional, contact-rich)
- Navigation tasks (different goal structure, longer horizons)
- Real robot transfer (sim-to-real gap, safety constraints)

Research directions (open problems):

- Automatic curriculum generation for goal-conditioned tasks
 - Sample-efficient goal representation learning
 - Combining model-based planning with goal-conditioned policies
-

1.15 §14. References

Foundational Papers.

- Hindsight Experience Replay (Andrychowicz et al., 2017)
- Soft Actor-Critic (Haarnoja et al., 2018)
- Proximal Policy Optimization (Schulman et al., 2017)

Textbooks.

- Sutton & Barto, *Reinforcement Learning: An Introduction* (2nd ed., 2018)
- Bertsekas, *Reinforcement Learning and Optimal Control* (2019)

Code Frameworks.

- Gymnasium-Robotics (Fetch/Shadow Hand environments)
 - Stable Baselines 3 (PPO/SAC/TD3 implementations)
-

1.16 §15. License

This repository is an educational research platform. Code is provided as-is for academic use. If you build upon this work, cite the foundational papers and acknowledge the pedagogical structure.

A Final Remark. This repository attempts to be more than a collection of scripts. It aspires to help readers understand goal-conditioned reinforcement learning from first principles—not just *how* to train a policy, but *why* policies learn and *when* they fail. Whether it succeeds is for the reader to judge. Corrections, criticisms, and improvements are welcome.