Assignment 7: Exotic — Reinforcement Learning

CS 312 — Ungraded · Due: December 3, 1pm (quiz time)

Basic setting. Problems 1–3 train a policy for Gymnasium's MuJoCo Humanoid, a simulated three-dimensional robot; Problem 4 uses a small gridworld, described there. Unless a problem states otherwise, use the default configuration in `reinforce.py` (Table 1) for 5 million environment steps. The task has two rewards (defined on page 2): the proxy reward the agent is trained on, and the true reward we actually want, the robot's forward velocity, which the agent never sees; its sum over an episode is the forward progress. Report the return under both, plus the episode length, both at the end of training and averaged over training (a measure of sample efficiency). Higher forward progress is better. There is no validation loss.

Instructions. Submit a single PDF report to Gradescope by the deadline above, with a maximum of 20 pages. This assignment is ungraded, but you may cite your report for partial credit on the quiz.

For each problem, include an analysis of the experiments you ran, followed by a short, approximately one-paragraph synthesis of your interpretation. Report the comparisons, hyperparameters, returns, and diagnostic evidence. Give plots clear axes and labels. Reinforcement learning is far noisier than pre-training: use at least three seeds for every comparison you draw conclusions from, show the spread across seeds, and record collapsed runs as well as successful ones.

Tune before you compare. Every new variant needs its own learning rate (at least a three-point search, extended if the best value is at an end), and comparisons of model size need the tuning described in the Preliminaries. An untuned comparison may show the opposite of the truth.

Code. This assignment uses a separate repository, `rl-alchemy`, rather than the shared course repository. It contains a REINFORCE and PPO-style trainer (this RL terminology is defined in the Preliminaries section below), batched and parallel environment stepping, per-update diagnostics, a renderer that saves GIFs of the robot, weight-averaging tools, and launchers for Modal. Starter files live in `experiments/`. Each defines the runs for its problem, with 3 seeds per setting; the counts below are one full pass through each starter. Add runs where your analysis needs them.

Problem	Starter file	Runs
Problem 1	`p1_pretraining_intuitions.py`	∼145 runs
Problem 2	`p2_off_policy.py`	∼110 runs
Problem 3	`p3_exploration.py`	∼40 runs
Problem 4	`p4_reward_hacking.py`	∼45 short runs

The physics simulation runs on the CPU. A default run takes about an hour on an 8-CPU Modal container, stepping its 32 episodes in 8 worker processes. Policies of width 128 or more run on a Modal H100 (`device="cuda"`), where a 5M-step run takes 20–90 minutes depending on size. `launch(configs, modal=True, detach=True)` starts runs that keep going after your machine disconnects. The repository `README.md` documents every option and logged metric.

Reference hardware. Quiz and assignment answers will be based on the following hardware: Modal containers with 8 CPUs for CPU runs, and a Modal H100 with 8 CPUs for runs with `device="cuda"`.

Preliminaries

Throughout this course, training has meant minimizing a fixed loss on a fixed data distribution, and lower validation loss has meant a better model. Reinforcement learning breaks both assumptions. The agent's own policy generates its training data, so every update changes the distribution the next update sees. And the quantity we care about, expected return, cannot be computed and differentiated directly: we follow an estimate of its gradient, built from a "surrogate loss" whose value is not monotonic with the objective.

This assignment asks you to explore whether some of your intuitions from previous assignments still hold, and also to investigate some RL-specific phenomena.

Basic Terminology

- **Episode.** One run of the environment from a reset to its end: either it terminates (the robot's torso drops out of a healthy height range, that is, it falls) or it is truncated at the time limit of 1,000 steps. An episode of length T visits time steps $t = 0, 1, \ldots, T-1$; each step is 0.015 simulated seconds.
- **State, action, reward.** At step t the agent observes the state s_t, chooses an action a_t, and receives a reward r_t, and the environment moves to s_{t+1}.
- **Policy.** $\pi_\theta(a \mid s)$ is the probability density of choosing action a in state s, given by a neural network with parameters θ. Here it is a Gaussian whose mean is the network output and whose standard deviation is a learned, state-independent parameter (its logarithm is `log_std`).
- **Return.** The undiscounted return of an episode is $\sum_t r_t$; this is what evaluation reports. The discounted return-to-go from step t is $G_t = \sum_{k \geq t} \gamma^{k-t} r_k$, with discount factor $\gamma \in [0, 1)$; this is what training uses.
- **Proxy reward and true reward.** The proxy reward is the reward the agent trains on: the r_t in every formula below. The true reward is the outcome we, as researchers, actually want. Sometimes we cannot train with the true reward directly because it would be intractable to optimize for it; other times we cannot use it because we can't even properly simulate it. For example the true reward might be how a robot trained in a simulation actually walks in the streets of San Francisco in the real world. For the purpose of the default humanoid environment in this assignment, the true reward at each step is just the robot's forward velocity; its sum over an episode, the true return, is what we call the forward progress. The proxy reward is the default from the Gymnasium package, which is supposedly crafted to help the humanoid learn to stand and walk like a person. The true reward is logged at evaluation but never used for training. The proxy and true reward are monotonic with each other only when the reward is correctly specified; Problem 4 studies what happens when they are not correctly specified (Skalse et al., 2022).
- **Surrogate loss.** The surrogate loss $\widehat{\mathcal{L}}$ defined below is the function the algorithm differentiates to obtain its gradient estimate. Its value is not a measure of performance.

Policy gradients

We want to maximize the expected discounted proxy return from the start of an episode, $J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_t \gamma^t r_t\right]$, where the expectation is over episodes generated by running π_θ. The policy gradient theorem (Sutton and Barto, 2018, §13.2) gives

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,\big(G_t - b(s_t)\big)\Big].$$

Here b is a baseline: any function of the state alone (it may not depend on the action a_t) that is subtracted from the return. Subtracting it leaves the expected gradient unchanged but can greatly reduce its variance. Common baselines are the average return in the batch, or a value network $V_\phi(s)$, a separate network with parameters ϕ trained by regression to predict G_t from s_t.[1] The difference $A_t = G_t - b(s_t)$ is called the advantage: how much better than expected the episode went after step t.

REINFORCE (Williams, 1992) estimates this gradient from data. Each update runs B complete episodes with the current policy, containing N steps in total, and takes one optimizer step on the surrogate loss

$$\widehat{\mathcal{L}}(\theta) = -\frac{1}{N}\sum_t \log \pi_\theta(a_t \mid s_t)\,\hat{A}_t,$$

where the sum runs over all N steps of the batch and $\hat{A}_t$ is treated as a constant (no gradient flows through it). In this assignment, as in most modern implementations (though not in Williams's original algorithm), the advantages are standardized within each batch: $\hat{A}_t = (A_t - \bar{A})/\sigma_A$, where $\bar{A}$ and σ_A are the mean and standard deviation of A_t over the N steps of the current batch, so that each batch's advantages have mean 0 and standard deviation 1. After the optimizer step, the episodes come from an outdated policy and are discarded.

Plain REINFORCE barely learns Humanoid. The default agent adds three things, which together make it a PPO-style algorithm (Schulman et al., 2017). Problem 2 studies the last two, which concern data from an older policy.

- GAE advantages (Schulman et al., 2016): instead of $G_t - V_\phi(s_t)$, blend one-step errors $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ as $A_t = \sum_{k\geq 0}(\gamma\lambda)^k\delta_{t+k}$, which trades a little bias for much lower variance ($\lambda = 1$ recovers the Monte Carlo advantage). Episodes cut off by the time limit bootstrap from the value of their last state.
- Many optimizer steps per batch: 10 passes over each batch, each split into 32 minibatches.
- A clipped surrogate that makes this reuse safe: with the probability ratio $w_t(\theta) = \pi_\theta(a_t \mid s_t)/\mu(a_t \mid s_t)$, where μ is the policy that collected the batch (see Problem 2), each step's term in the surrogate loss becomes $-\min\left(w_t\hat{A}_t,\ \mathrm{clip}(w_t, 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$ with $\epsilon = 0.2$. Gradient norms are also clipped at 0.5.

The optimizer is AdamW with learning rate η and decoupled weight decay λ. Two clocks matter: environment steps and optimizer updates. Steps per batch change as the policy's episodes get longer.

Default Environment

Observation	348 numbers: joint angles and velocities, center-of-mass and contact information; the robot's absolute horizontal position is excluded.
Action	17 actuator controls, each in $[-0.4, 0.4]$ (a gear ratio of 100 turns them into joint torques).
Proxy reward	$1.25\times$ forward velocity $+5$ for every step the robot stays up $-$ small costs for large torques and contact forces. The weights are set through `env_kwargs` (`forward_reward_weight`, `healthy_reward`).
True reward	Forward velocity alone. A robot that stands still for a full episode earns about 5,000 proxy return and zero forward progress.
Start state	A standing pose plus uniform noise of ± 0.01 on every joint position and velocity (`reset_noise_scale`).

[1] We follow the common practice of dropping the factor γ^t in front of each term, which makes this the gradient of a slightly different objective.

Tuning

Policy gradients are sensitive to the size of each update, and that size depends on the learning rate and the network's shape together. Two diagnostics measure the size of an update in terms of how much it changes the policy's behavior. The first is the KL divergence $\mathrm{KL}(p \,\|\, q) = \mathbb{E}_{a\sim p}\big[\log p(a \mid s) - \log q(a \mid s)\big]$, averaged over visited states, which is zero exactly when the two action distributions agree and grows as they separate; `kl_update` is $\mathrm{KL}(\pi_{\text{old}} \,\|\, \pi_{\text{new}})$ between the policy before and after a batch's updates, computed in closed form for our Gaussian policies. The second is the fraction of samples whose ratio left the clipping range (`clip_frac`). As a rule of thumb, a KL around 0.01–0.03 per batch is healthy (the range of targets in the PPO paper, Schulman et al., 2017); values above about 0.1, with most samples clipped, mean the policy is lurching and most of each batch contributes no gradient. Follow these rules throughout:

- Search the learning rate separately for every variant you compare (at least three values, extended if the best is at an end), and report the tuned result.
- Watch `kl_update`, `clip_frac` and `lr` in every run to debug unstable runs.

Table 1 lists the default settings. Figure 1 shows three runs of the default agent. The proxy return rises quickly as the agent learns to stay upright, which alone is worth up to 5 per step. Forward progress lags behind and keeps improving for the whole run. The surrogate loss shrinks toward zero without tracking either.

Table 1. Default settings (`PGConfig` in `reinforce.py`).

Setting	Default	Setting	Default
Environment steps	5,000,000	Policy	residual MLP (GELU), 64 wide, 1 block
Batch size	32 episodes	Value network	tanh MLP, 256–256
Discount γ, GAE λ	0.99, 0.95	Optimizer	AdamW, $\eta = 3 \times 10^{-5}$, $\lambda = 0$
Updates per batch	10×32 minibatches	Clipping	ratio 0.2, grad. norm 0.5
Advantages	GAE, standardized	Initial log std	-0.5
Observations	normalized (running)	Evaluation	8 episodes per 250k steps
Reward scale	$\times 0.1$ (training only)	Value updates	Adam, 10^{-3}, 20 steps per batch
Entropy bonus	none		

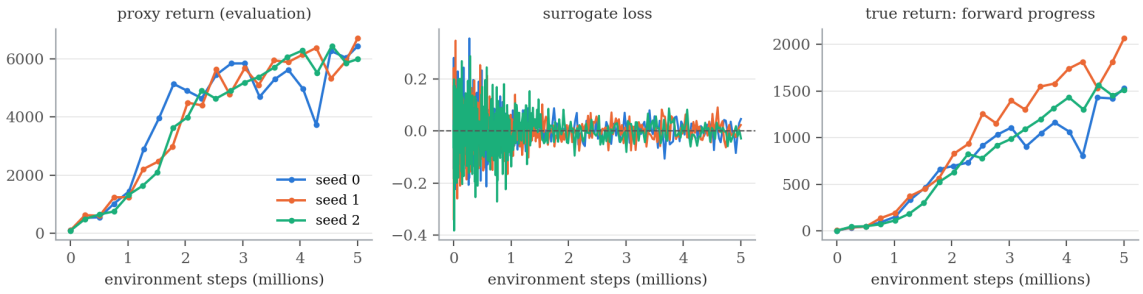


Figure 1. Default agent, three seeds. Left: proxy return at evaluation. Middle: surrogate loss at each update. Right: true return, the forward progress per evaluation episode.

Watching the robot

Metrics tell only part of the story, so watch your policies, as early and as often as you like. `uv run python -m render RUN` saves a GIF of one episode of a run's final policy, with a camera that follows the robot, and prints how far it went and whether it fell. `--file pi_500000.pt` renders a checkpoint saved with `save_checkpoints`, and `--fetch` first copies the policy from the Modal Volume. Before starting the problems, train the default agent once with checkpoints at 0.5M and 2M steps, and render all three policies: you will see it learn to keep its balance before it learns to walk. Render again whenever a curve surprises you. Several problem parts ask what the robot is doing, and a few still frames in your report are often the clearest evidence.

Diagnostics

Every update logs the surrogate loss, the policy entropy and log standard deviation, the KL divergence and clip fraction of the update, the gradient and parameter norms, and the learning rate. On request it estimates the gradient noise scale $\mathcal{B} = \mathrm{tr}(\Sigma)/\|g\|^2$ in episodes (McCandlish et al., 2018), where g is the expected gradient and Σ the covariance of single-episode gradients. Evaluation reports proxy return, forward progress and episode length, under the training settings and under any extra settings you request (`extra_evals`). The repository's `README.md` lists every logged quantity.

Roadmap

Problem 1 revisits intuitions from earlier assignments; Problems 2–4 study phenomena specific to reinforcement learning.

1. Check which intuitions from pre-training survive in RL: how batch size and learning rate, $\eta\lambda$, width and depth, learning-rate schedules, and weight averaging behave.
2. Train from off-policy data, collected by an older copy of the policy, and experiment with common ways to handle it: no correction, importance sampling, clipped importance ratios, and PPO.
3. Experiment with how much exploration we make the policy do, and how this affects performance. Study the optimization landscape with respect to exploration: find the local optima and plateaus of the proxy-return landscape, and test what lets the optimizer escape them.
4. Rewards we implement in practice often differ from the true reward we actually want. Characterize when and how agents hack their proxy reward, measuring the true reward, which is hidden from the agent.

Do the pre-training intuitions transfer?

Assignments 1–6 built intuition for training language models: the best learning rate moves with batch size, the product of learning rate and weight decay matters, performance scales with width and depth, learning-rate schedules help, and averaging weights works. We had a specific picture of training: a fixed loss landscape, gradient noise from sampling a fixed dataset, and a river valley that a decaying learning rate settles into.

In policy gradients, the gradient noise comes from sampling whole trajectories, the landscape moves as the policy changes the data, and the objective is an expected return, not a loss.

Problem 1: Do the pre-training intuitions transfer?

(a) **Batch size and learning rate.** Sweep the batch size $B \in \{8, 32, 128\}$ episodes jointly with the learning rate. How does the best learning rate scale with B, compared with pre-training? Compare runs at a fixed environment-step budget and at a fixed number of updates. Measure the gradient noise scale over training (`noise_scale_every`). Does it predict the batch size beyond which larger batches stop paying off?

(b) **Is $\eta\lambda$ the right quantity?** Sweep weight decay λ jointly with η. Plot proxy return and forward progress against $\eta\lambda$ and against η and λ separately. Is performance a function of the product, as in pre-training? Track the parameter norm, and compare the equilibration timescale $1/(\eta\lambda)$ with the number of updates in a run.

(c) **Width and depth.** To tune each model size, use `fair()` (`experiments/common.py`), which zero-initializes each residual block's output layer so every depth starts as the same function, adapts the learning rate to hold the KL per update near 0.02, and starts from a learning rate proportional to width$^{-1/2}$. With it, train policies of width $\{16, 64, 256, 1024\}$ at depth 1, and of depth $\{1, 2, 4, 8\}$ at width 64; then repeat the widest policies with the value network widened along with the policy (`fair`'s option `scale_value`). Also train the starter's untuned controls (default fixed learning rate and initialization). Plot proxy return and forward progress against width and depth. Where does extra width stop helping? Does depth help? Fit how the learning rate the adaptation settles on scales with width and depth. Compare with the untuned controls and explain the difference using `kl_update` and `clip_frac`.

(d) **Learning-rate schedules.** Compare constant, linear decay, cosine, cosine with warmup, and warmup-stable-decay schedules, each at two peak learning rates. Does decaying the learning rate help the final proxy return and forward progress? Their means over training? In pre-training, decay helped by reducing the noise that kept the iterate bouncing across a river valley; is that plausible when the landscape itself moves?

(e) **Weight averaging and basins.** Evaluate an exponential moving average of the weights alongside the raw weights (`ema_decay`). Then measure proxy return and forward progress along the line between pairs of trained policies (`analysis.interpolate`). Use pairs from independent seeds, and pairs forked from a shared checkpoint at 1M and 2.5M steps with different data seeds (`init_from`, `data_seed`). Are there barriers, and do they depend on when the runs were forked? Does a uniform average of several forks beat its members?

Learning from off-policy data

The policy-gradient theorem assumes that every sample comes from the policy being updated. Practical systems break this all the time. We've even broken this in our pset already: the default agent reuses each batch for 320 optimizer steps, so after the first step the data comes from an older policy. Asynchronous systems such as IMPALA (Espeholt et al., 2018), and the large systems that fine-tune language models with RL, act with copies of the policy that are several updates old, because waiting for fresh samples wastes hardware. Call the policy that collected the data the behavior policy $\mu(a \mid s)$, and the policy being updated π_θ.

On- and off-policy gradients. Write $A(s,a)$ for the advantage of action a in state s. At each state s, the policy gradient weights every action by how often the *current* policy takes it:

$$g(s) = \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, A(s,a)\big] = \sum_a \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)\, A(s,a)$$

(an integral for continuous actions). REINFORCE estimates it *on-policy*: the actions in the batch are sampled from π_θ itself, so averaging $\nabla_\theta \log \pi_\theta(a_t \mid s_t)\hat{A}_t$ over the batch gives g in expectation. *Off-policy*, the batch was collected by some other, fixed behavior policy μ: an older copy of π_θ, or any external policy, such as a human demonstrator. The same average then weights each action by how often μ takes it:

$$\mathbb{E}_{a\sim\mu(\cdot|s)}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, A(s,a)\big] = \sum_a \mu(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)\, A(s,a).$$

Importance weights. The fix is to reweight each sample by the importance weight $w = \pi_\theta(a \mid s)/\mu(a \mid s)$, treated as a constant (a stop-gradient). For any function f, as long as $\mu(a \mid s) > 0$ wherever $\pi_\theta(a \mid s) > 0$,

$$\mathbb{E}_{a\sim\mu}\Big[\frac{\pi_\theta(a \mid s)}{\mu(a \mid s)}\, f(a)\Big] = \sum_a \mu(a \mid s)\, \frac{\pi_\theta(a \mid s)}{\mu(a \mid s)}\, f(a) = \sum_a \pi_\theta(a \mid s)\, f(a) = \mathbb{E}_{a\sim\pi_\theta}\big[f(a)\big].$$

With $f(a) = \nabla_\theta \log \pi_\theta(a \mid s)\, A(s,a)$, the weighted average of μ's samples has expectation $g(s)$ at every state: the importance-weighted surrogate $-\frac{1}{N}\sum_t w_t \log \pi_\theta(a_t \mid s_t)\hat{A}_t$ has, in expectation, the on-policy gradient. Importance weights do not fix everything, though: they can add variance to our gradient estimate, among other problems. In this problem we will explore importance weighting along with additional tweaks that help models learn from off-policy data.

Our off-policy setup. The default agent is mildly off-policy: it collects each batch with its current policy and then takes 10 passes of 32 minibatch steps over it, so every step after the first uses data from a slightly older version of itself. This problem changes both sources of mismatch. It reduces reuse to a single pass of 32 minibatch steps per batch, and it adds staleness: the behavior policy μ is the policy from ℓ batches ago (`policy_lag`; $\ell = 4$ unless stated otherwise). At $\ell = 0$ the data is therefore less off-policy than the default agent's, and at $\ell = 4$ or 16 it is far more, with most of the mismatch coming from staleness. The goal of this problem is to find which parts of the default algorithm are essential as the data becomes more off-policy. Every algorithm variant we explore keeps the default advantages (GAE, a value baseline, standardization) and gradient-norm clipping at 0.5, and strips the rest away: one pass per batch, and a single way of handling the mismatch, from no correction at all up to PPO's clipped objective. We then explore "Full PPO", the default agent with everything restored (10 passes per batch) but acting with stale policies. Every run logs how far π_θ is from μ on the batch's own samples, before and after the batch's updates (`behavior_kl_start` and `behavior_kl_end`, which estimate $\mathrm{KL}(\mu \,\|\, \pi_\theta)$ on μ's samples, and `ratio_max_*`).

Problem 2: Correcting for off-policy data

(a) **No correction.** Use μ's samples as if they came from π_θ, at $\ell \in \{0, 4, 16\}$. (i) Plot the distance between π_θ and μ at the start and the end of each batch's updates over training. Compare the KL between them when a batch arrives (`behavior_kl_start`) at $\ell = 4$ and 16 with the KL built up over one pass at $\ell = 0$ (`behavior_kl_end`). How much more mismatch does staleness add? A lag is measured in batches; about how many environment steps of experience does $\ell = 16$ correspond to? (ii) For a fixed batch, the uncorrected estimator is the gradient of $L(\theta) = \frac{1}{N} \sum_t \log \pi_\theta(a_t \mid s_t) \hat{A}_t$. Show that its expected gradient equals the policy gradient only when $\pi_\theta = \mu$. (iii) In training, at what staleness does the uncorrected estimator start to hurt? It can fail even at $\ell = 0$; use your answer to (ii) and the logged `grad_norm`, `kl_update` and `behavior_kl_end` to explain why.

(b) **Importance sampling.** Weight each step's term by the importance weight $w_t = \pi_\theta(a_t \mid s_t) / \mu(a_t \mid s_t)$, treated as a constant. (i) The importance weight corrects which actions are taken at each state. Which parts of the estimate are still not corrected (consider which states are visited, and how the advantages were estimated)? (ii) In training at $\ell \in \{0, 4, 16\}$, how much does importance sampling improve on no correction?

(c) **Clipping importance ratios.** As in V-trace (Espeholt et al., 2018), weight each step's term by $\min(w_t, c)$, for $c \in \{1, 2, 5\}$ (`is_clip`). Importance weights can be very large when π_θ and μ differ, and a few large weights make the gradient estimate very noisy. Capping them at c bounds that variance, at the price of a bias. (i) Write the expected truncated estimator as the on-policy gradient minus a correction term. Which actions does truncation underweight, and toward which policy does that bias the update? What does $c = 1$ do to actions that π_θ now prefers over μ? (ii) In training, which threshold works best (if any), and does the answer depend on ℓ? Explain the ranking with the bias you derived in (i).

(d) **PPO's clipped objective.** Replace the weighted term with PPO's clipped objective from the Preliminaries, still with one pass per batch. (i) For each sign of $\hat{A}_t$ and each range of w_t, write its per-step gradient and that of the truncated estimator from (c). Where do they differ? (ii) In training at $\ell \in \{0, 4, 16\}$, how does it compare with (b) and (c)? (iii) PPO's clipping range is centered on μ, not on the policy at the start of the batch. What fraction of samples is already outside the range when a batch arrives at $\ell = 16$ (`ratio_outside_clip_start`), and what does that do to the update?

(e) **Full PPO.** Restore the rest of the default agent: 10 passes per batch. Train it at $\ell \in \{0, 4, 16\}$ (`lag_runs` trains every variant across these lags, each at its own tuned learning rate). Rank all the variants from (a)–(e) at each staleness level. Which parts of the default algorithm turn out to be essential as the data becomes more off-policy?

Exploration and the optimization landscape

This problem studies how agent exploration affects where you end up on the performance landscape. If the entropy of an agent gets too small, or if we've started in the wrong place, we may get stuck in local optimums and never be able to find a good policy. Humanoid has a clean example of a local optimum. Keep Gymnasium's proxy reward, but remove the rule that a fall ends the episode (the starter's `CRAWL` configuration). The survival bonus is then paid only while the torso is upright, so falling no longer ends the episode but does stop the bonus. Once on the ground, the agent can still earn forward reward by moving along the floor, and standing back up is a long, coordinated sequence that earns nothing until it is complete. With the fall rule in place (the default agent), the same reward learns to walk.

Problem 3: The optimization landscape with respect to exploration

(a) **Anatomy of a local optimum.** Train on the proxy reward with and without the fall rule, with checkpoints (`local_optimum_runs`). Render a few episodes of each (`render`; pass `--env` to watch a crawler under the fall rule) and describe the behaviors. Plot the proxy return and forward progress each run achieves in its own training environment and under the task's rule that a fall ends the episode (`eval_upright_*`). Then gather evidence that the behavior without the fall rule is a local optimum: measure the returns along the straight line from such a policy to a walker (`analysis.interpolate`), and along random directions around it (`random_directions`). Is it a local maximum, or a region where the gradient is just weak?

(b) **Escaping it.** Starting from scratch without the fall rule, try to find walking by changing the policy's randomness, with an entropy bonus ($c_H = 0.01$, constant or annealed over the first 30% of training) or larger initial action noise (`init_log_std` $= 0.5$), by changing the gradient noise, with a larger batch or learning rate, and by modifying the proxy reward itself, for example with a larger survival bonus (`healthy_reward` of 10 or 20, against the default 5; all in `escape_runs`). Track `log_std` and the upright episode length. Does any seed escape the crawl? If nothing escapes, part (c) hints at one way out. The survival bonus already pays for staying upright; why is the default bonus not enough to make the agent walk without the fall rule, and does a larger one help?

(c) **Which basin do you land in?** Remove the fall rule from a walker, either at the end of its training or after 2M steps (`continuation_runs`). Does it keep walking, or does it fall and crawl? Compare its proxy return and forward progress, under the no-fall objective, with those of the policies trained without the fall rule from the start. What do these runs say about which optima are stable, and about the role of early training in choosing among them? Relate this to curricula, pre-training, and to the forked policies in 1(e).

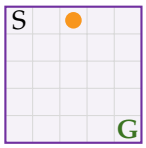
(d) **Exploring through the start state.** Train at start-state noise levels $\{0.01, 0.05, 0.1\}$ and evaluate every run at these levels and at a held-out 0.2 (`coverage_runs`). Which training level gives the best policy at each test level? Does training on more varied start states cost performance on the default start state, and does it help the policy recover from states it would otherwise never visit? Relate this to data coverage and to generalization in supervised learning.

Reward hacking

An agent optimizes the reward we give it, which is rarely exactly the true reward we want. Two different failures follow. A misspecified proxy simply has a different optimum: an agent trained on it ends up somewhere we did not intend. Reward hacking is the stronger failure: as the agent optimizes the proxy harder, the true reward first rises and then falls (Skalse et al., 2022). Real examples include a boat-racing agent that circled forever collecting respawning targets instead of finishing the race (Clark and Amodei, 2016); language models over-optimizing learned reward models (Gao et al., 2023); and coding agents that special-case or edit unit tests to make them pass (Baker et al., 2025); see Krakovna et al. (2020) for more.

Problem 3 showed why the details of a reward specification matter: once the fall rule is gone, even Gymnasium's proxy reward, which pays for staying upright, leads to crawling. Practical rewards add terms and rules beyond the outcome we want; each shapes behavior, and each can be exploited.

CoinLoop. The agent starts in the top-left corner of a 5×5 grid; the goal is in the bottom-right, 8 steps away. The true reward is $+1$ for reaching the goal, which ends the episode. The proxy reward adds $+0.2$ each time the agent enters a coin cell two steps from the start while the coin is charged; it then recharges for 4 steps. Episodes are cut off after 50 steps, and by default the agent cannot observe the time (`time_obs=False`). Since the true reward is earned only at the goal, an episode's true return is 1 or 0, and the average true return is the fraction of episodes that reach the goal. Because the grid is tiny, `envs.coinloop_solve` can compute the exactly optimal policy for the proxy reward in two simplified settings: episodes that never end, and 50-step episodes in which the policy can see the time. The starter trains plain REINFORCE (`COINLOOP`).



S: start
●: coin (+0.2, proxy only)
G: goal (+1, ends the episode)

Problem 4: When and how do agents hack their reward?

(a) **Farm or finish?** Without a time observation, REINFORCE's policy must act the same way at every step in a given state, while its episodes really do end after 50 steps. It therefore optimizes the discounted 50-step proxy return of a time-independent policy, which matches neither of the two simplified settings above. Use the optimal policies in those two settings as reference points to predict, for $\gamma \in \{0.8, 0.9, 0.95, 0.99\}$, whether REINFORCE will farm the coin or finish (`coinloop_predictions`). Then train with `time_obs` off and on (`coinloop`). Were your predictions right? When the agent can see the time, does it find the "farm, then finish" policy that is optimal in that setting? If not, explain why, using the policy's entropy over training. Is what you observe misspecification or reward hacking, in the sense defined above, and does the answer depend on γ?

(b) **KL-regularized proxy optimization.** RLHF's standard safeguard against over-optimizing a learned reward is to keep the policy close to a trusted reference policy π_{ref}: instead of the proxy return alone, maximize the proxy return minus $c_{\text{KL}}\,\text{KL}(\pi_\theta \,\|\, \pi_{\text{ref}})$, averaged over the states the policy visits (Gao et al., 2023). This penalizes π_θ for taking actions that π_{ref} rarely takes. Here the reference, trained only briefly on the true reward (`REFERENCE`), reaches the goal in about 85% of episodes. Fine-tune it on the proxy for a range of c_{KL} (`kl_sweep`). As the proxy return rises, does the true return first rise, then fall? Is there a KL budget at which the true return peaks? If you plot every run's true return against its KL to the reference, do runs with different c_{KL} fall on the same curve, so that the true return depends only on how far the policy has moved?