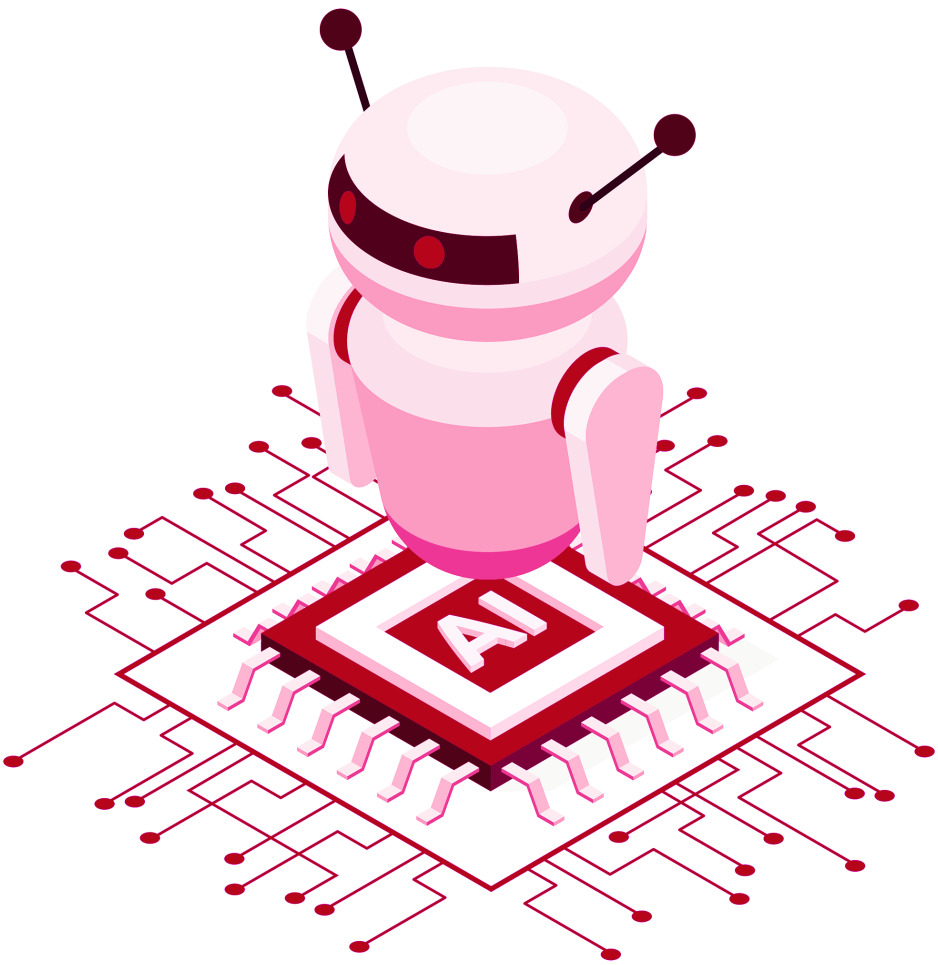

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Solution for Homework 14:

Meta Reinforcement Learning

By:

Payam Taebi

400104867

Spring 2025

Grading

The grading will be based on the following criteria, with a total of 110 points:

- Model-Agnostic Meta-Learning : 100 points
- Clarity and Quality of Code : 5 points
- Clarity and Quality of Report : 5 points

Contents

1 Model-Agnostic Meta-Learning

Overview. This notebook implements Model-Agnostic Meta-Learning (MAML) for a continuous-control variant of the HalfCheetah task where the agent is rewarded for *running backward*. The pipeline consists of: (i) a custom environment wrapper that inverts the forward-velocity reward, (ii) a diagonal-Gaussian stochastic policy with state-independent log-standard-deviations, (iii) trajectory collection with discounted-return computation, (iv) an evaluation routine that renders a video, and (v) a MAML training loop that performs one-step inner adaptation and an outer meta-update. Below we dissect design choices, mathematical foundations, implementation subtleties, expected behavior, failure modes, and practical extensions—focusing on why each component is written as it is and how it affects meta-learning dynamics.

Environment Design: `HalfCheetahBackward`. The wrapper constructs `HalfCheetah-v5` with `render_mode=rg` and forwards `action_space` and `observation_space` so downstream code can introspect dimensions without querying the base env. The reward is

$$r_t \;=\; -\,w_f\,\text{reward\_forward}_t \;+\; w_c\,\text{reward\_ctrl}_t,$$

with $w_f = 1$ and $w_c = 0.05$. In MuJoCo's HalfCheetah, reward_ctrl ≤ 0 (a penalty on torques), so adding w_c reward_ctrl still discourages large control magnitudes. This simple shaping flips the task objective (move backward) without altering the dynamics. Correct Gymnasium API handling is crucial: `reset` returns $(\text{obs}, \text{info})$ and `step` returns $(\text{obs}, r, \text{terminated}, \text{truncated}, \text{info})$. The wrapper mirrors these semantics to avoid subtle bugs during rollouts and video generation.

Policy Parameterization. The policy is a Gaussian $\pi_\theta(a \mid s) = \mathcal{N}\big(\mu_\theta(s), \mathrm{diag}(\sigma^2)\big)$ with

$$\mu_\theta(s) = f_\theta(s), \quad \log\sigma = \phi \in \mathbb{R}^{d_a} \;\; \text{(state-independent)}.$$

An MLP backbone with Tanh activations projects observations to the mean via a linear head. The log-std vector `log_std` is a learned parameter independent of s, enabling a minimal yet effective exploration scheme. Using `Independent(Normal(mean, std), 1)` yields scalar $\log\pi_\theta(a \mid s)$ per sample by summing over action dimensions, matching the policy-gradient estimator's expectation. Because MuJoCo actions are bounded, actions are clipped to the environment's action space; this is pragmatic but introduces off-manifold gradients at the bounds. A more principled alternative is a Tanh-squashed Gaussian with a log-det Jacobian correction, though that adds complexity.

Trajectory Collection and Returns. The rollout function gathers $\{(s_t, a_t, \log\pi_\theta(a_t \mid s_t), r_t)\}_{t=0}^{T-1}$ for up to `max_steps`. Discounted returns are computed as

$$G_t \;=\; \sum_{k=0}^{T-t-1} \gamma^k r_{t+k},$$

producing a per-timestep return vector $\mathbf{G} \in \mathbb{R}^T$. Two details are critical for MAML: (i) do *not* wrap policy sampling/log-prob in `torch.no_grad()`, and (ii) avoid detaching $\log\pi$. The inner update requires higher-order gradients through the loss, so the computational graph must remain intact from $\log\pi$ back to θ. The implementation satisfies this by computing $\log\pi$ in standard autograd mode and returning it unmodified.

Evaluation Logic. The evaluation uses the deterministic mean action $a = \mu_\theta(s)$ to reduce variance and provides an optional video of the first episode. This is a conventional diagnostic: during training, stochasticity aids exploration; during eval, determinism clarifies whether the learned policy genuinely captures the backward-running behavior. Mean episodic return over multiple trials is reported to smooth out variability due to stochastic transitions and termination conditions.

Policy Gradient Objective. Within a trajectory, the REINFORCE-style objective maximizes expected return:

$$J(\theta) \;\approx\; \frac{1}{T}\sum_{t=0}^{T-1} G_t \,\log \pi_\theta(a_t \mid s_t).$$

The inner (support) and outer (query) losses are implemented as negative of this estimator:

$$\mathcal{L}_{\mathsf{PG}}(\theta) \;=\; -\frac{1}{T}\sum_{t} G_t \,\log \pi_\theta(a_t \mid s_t).$$

Variance reduction: the code currently uses raw returns G_t as a baseline-free estimator. In practice, normalizing G_t within a trajectory (zero mean, unit variance) or subtracting a learned value baseline (advantages) can materially stabilize learning. The provided L2 regularization (both inner explicit L2 and outer optimizer weight decay) also dampens high-variance parameter excursions.

MAML Mechanics. Given a task T_i (here, a single environment instance with backward reward), the one-step inner update is

$$\theta_i' \;=\; \theta \;-\; \alpha \,\nabla_\theta \mathcal{L}_{\mathsf{support}}(\theta),$$

and the meta-objective is

$$\min_\theta \;\; \sum_{i=1}^{B} \mathcal{L}_{\mathsf{query}}(\theta_i') \;=\; \sum_{i=1}^{B} \mathcal{L}_{\mathsf{query}}\Big(\theta - \alpha \nabla_\theta \mathcal{L}_{\mathsf{support}}(\theta)\Big),$$

with meta-batch size B. The implementation obtains support data under θ, computes $\nabla_\theta \mathcal{L}_{\mathsf{support}}$ with `create_graph=True`, forms θ' via a functional update, then evaluates the query loss under θ' by running a new rollout. The gradient $\nabla_\theta \mathcal{L}_{\mathsf{query}}(\theta')$ is automatically computed by autograd through the inner update thanks to the preserved higher-order graph.

Functional Adaptation. A subtle but elegant choice is to keep the meta-parameters θ intact and instantiate an *adapted* stateless view using `torch.nn.utils.stateless.functional_call`. Concretely, after computing the gradient list g, the code constructs

$$\theta' \;=\; \theta \;-\; \alpha g,$$

maps names to tensors, and calls the same forward functions using θ' without mutating θ. This ensures: (i) the outer optimizer always steps θ, (ii) higher-order derivatives propagate correctly, and (iii) no in-place ops break the graph. One caveat is ordering: the code zips gradients (produced in `model.parameters()` order) with `named_parameters()` dict items. Because PyTorch preserves module registration order, these align; still, a more explicit mapping (e.g., deriving grads from `named_parameters()` directly) further reduces risk.

Regularization and Stability. Two layers of regularization appear: an explicit inner L2 term with coefficient 10^{-3} and an outer Adam weight decay 10^{-4}. Dual regularization can be beneficial but also may underfit if too strong; tuning both is recommended. Additional stabilizers that commonly help in MuJoCo tasks include: (i) gradient clipping on both inner and outer steps, (ii) standardizing returns within each trajectory, (iii) entropy regularization to prevent premature collapse of exploration (i.e., add $-\beta\,\mathbb{H}[\pi_\theta(\cdot|s)]$ to the loss), and (iv) using advantages from a learned value function as a baseline. The current code favors simplicity and clarity over maximal stability.

Hyperparameters and Their Roles. A reasonable starting configuration is α = `inner_lr` = 0.1, `outer_lr` = $3 \cdot 10^{-4}$, B = `meta_batch_size` = 5, horizon T = `max_episode_len` = 200, and $\gamma = 0.99$. Increasing B reduces meta-gradient variance but linearly grows sample cost. Larger α accelerates adaptation but can destabilize the meta-gradient (since higher-order sensitivities amplify), while smaller α weakens adaptation pressure on θ. The code currently performs *one* inner step; extending to multiple steps requires iterating support rollouts and updates (or reusing the same support data) and re-wrapping parameters at each step.

Computational Footprint. Higher-order autograd increases memory usage: backpropagating through the inner update retains the computation graph built for the support loss and the policy forward passes that produced $\log \pi$. Practical measures when memory is tight: reduce horizon T, truncate episodes on termination, use smaller networks, or adopt first-order MAML (FOMAML) by setting `create_graph=False`—sacrificing exact higher-order terms for substantial savings.

Diagnostics and Expected Learning Signal. Two time-series are plotted: (i) meta-loss (average query loss) and (ii) average query reward. As training progresses, meta-loss should trend down while average query return trends up. Because the task rewards backward velocity, a qualitative confirmation is to render the evaluation video and visually verify consistent backward motion. If learning stalls, common culprits are: (a) $\log \pi$ detached anywhere in the pipeline, (b) incorrect Gymnasium return handling causing premature termination or missing frames, (c) action scaling/clipping issues, and (d) excessively high/low α or `outer_lr`.

Potential Extensions. *Multi-step inner loop:* iterate the functional update several times before computing the query loss. *Advantage estimation:* fit a value baseline (e.g., GAE-λ) to reduce variance and improve sample efficiency. *Entropy bonus:* encourage exploration during both support and query rollouts. *Task distribution:* introduce heterogeneity (e.g., randomize mass, joint damping, friction, or backward target speeds) to better showcase meta-learning's cross-task generalization. *Tanh-squashed policy:* replace clipping with a squashed Gaussian and include the Jacobian term in $\log \pi$ to restore proper gradients near bounds.

Limitations. The present code treats each env instance as the same "task" modulo stochasticity; true meta-learning benefits from a distribution of tasks. Without task diversity, MAML tends to mimic vanilla policy gradient with an unconventional regularizer. Moreover, using raw returns as weights is high variance; in practice, advantage normalization and entropy regularization are almost always required for robust convergence on MuJoCo benchmarks.

Takeaways. This implementation captures the essence of MAML in continuous control: preserving higher-order gradients by avoiding detaches, performing a differentiable inner update, and evaluating a query loss under adapted parameters via a stateless functional call. The environment wrapper cleanly inverts the objective with minimal code changes. With modest tuning (return normalization, entropy bonus, and possibly a value baseline), the agent should reliably adapt to the backward-running task in a few gradient steps—validating the core MAML hypothesis that one can meta-learn initializations that are primed for rapid adaptation.

References

[1] Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks

[2] A Survey of Meta-Reinforcement Learning