Supplementary Material: What Can Learned Intrinsic Rewards Capture?

Zeyu Zheng [*] [†] [1] **Junhyuk Oh** [*] [2] **Matteo Hessel** [2] **Zhongwen Xu** [2] **Manuel Kroiss** [2] **Hado van Hasselt** [2] **David Silver** [2] **Satinder Singh** [2]

A. Derivation of Intrinsic Reward Update

Following the conventional notation in RL, we define $v_{\mathcal{T}}(\tau_t|\eta,\theta_0)$ as the state-value function that estimates the expected future lifetime return given the lifetime history τ_t, the task $\mathcal{T}$, initial policy parameters θ_0 and the intrinsic reward parameters η. Specially, $v_{\mathcal{T}}(\tau_0|\eta,\theta_0)$ denotes the expected lifetime return at the starting state, i.e.,

$$v_{\mathcal{T}}(\tau_0|\eta,\theta_0) = \mathbb{E}_{\tau\sim p_\eta(\tau|\theta_0)}\left[G^{\text{life}}\right],$$

where G^{life} denotes the lifetime return in task $\mathcal{T}$. We also define the action-value function $q_{\mathcal{T}}(\tau_t, a_t|\eta,\theta_0)$ accordingly as the expected future lifetime return given the lifetime history τ_t and an action a_t.

The objective function of the optimal reward problem is defined as:

$$J(\eta) = \mathbb{E}_{\theta_0\sim\Theta,\mathcal{T}\sim p(\mathcal{T})}\left[\mathbb{E}_{\tau\sim p_\eta(\tau|\theta_0)}\left[G^{\text{life}}\right]\right] \tag{1}$$

$$= \mathbb{E}_{\theta_0\sim\Theta,\mathcal{T}\sim p(\mathcal{T})}\left[v_{\mathcal{T}}(\tau_0|\eta,\theta_0)\right], \tag{2}$$

where Θ and $p(\mathcal{T})$ are an initial policy distribution and a task distribution respectively.

Assuming the task $\mathcal{T}$ and the initial policy parameters θ_0 are given, we omit $\mathcal{T}$ and θ_0 for the rest of equations for simplicity. Let $\pi_\eta(\cdot|\tau_t) = \pi_{\theta_t}(\cdot|s_t)$ be the probability distribution over actions at time t given the history τ_t, where $\theta_t = f_\eta(\tau_t,\theta_0)$ is the policy parameters at time t in the lifetime. We can derive the meta-gradient with respect to η by the following:

$$\begin{aligned}
&\nabla_\eta J(\eta)\\
&= \nabla_\eta v(\tau_0|\eta)\\
&= \nabla_\eta\left[\sum_{a_0}\pi_{\theta_0}(a_0|\tau_0)q(\tau_0,a_0|\eta)\right]\\
&= \sum_{a_0}\left[\nabla_\eta\pi_{\theta_0}(a_0|\tau_0)q(\tau_0,a_0|\eta) + \pi_{\theta_0}(a_0|\tau_0)\nabla_\eta q(\tau_0,a_0|\eta)\right]\\
&= \sum_{a_0}\left[\nabla_\eta\pi_{\theta_0}(a_0|\tau_0)q(\tau_0,a_0|\eta) + \pi_{\theta_0}(a_0|\tau_0)\nabla_\eta\sum_{\tau_1,r_0}p(\tau_1,r_0|\tau_0,a_0)\Big(r_0 + v(\tau_1|\eta)\Big)\right]\\
&= \sum_{a_0}\left[\nabla_\eta\pi_{\theta_0}(a_0|\tau_0)q(\tau_0,a_0|\eta) + \pi_{\theta_0}(a_0|\tau_0)\sum_{\tau_1}p(\tau_1|\tau_0,a_0)\nabla_\eta v(\tau_1|\eta)\right]\\
&= \mathbb{E}_{\tau_t}\left[\sum_{a_t}\nabla_\eta\pi_\eta(a_t|\tau_t)q(\tau_t,a_t|\eta)\right]\\
&= \mathbb{E}_{\tau_t}\left[\nabla_\eta\log\pi_\eta(a_t|\tau_t)q(\tau_t,a_t|\eta)\right]\\
&= \mathbb{E}_{\tau_t}\left[G_t\nabla_\eta\log\pi_\eta(a_t|\tau_t)\right]\\
&= \mathbb{E}_{\tau_t}\left[G_t\nabla_{\theta_t}\log\pi_{\theta_t}(a_t|s_t)\nabla_\eta\theta_t\right],
\end{aligned}$$

[*]Equal contribution [†]Work done during an internship at DeepMind. [1]University of Michigan [2]DeepMind. Correspondence to: Zeyu Zheng <zeyu@umich.edu>, Junhyuk Oh <junhyuk@google.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

where $G_t = \sum_{k=t}^{T-1} r_k$ is the lifetime return given the history τ_t, and we assume the discount factor $\gamma = 1$ for brevity. Thus, the derivative of the overall objective is:

$$\nabla_\eta J(\eta) = \mathbb{E}_{\theta_0 \sim \Theta, \mathcal{T} \sim p(\mathcal{T})} \left[\mathbb{E}_{\tau_t \sim p(\tau_t | \eta, \theta_0)} \left[G_t \nabla_{\theta_t} \log \pi_{\theta_t}(a_t | s_t) \nabla_\eta \theta_t \right] \right]. \tag{3}$$

B. Experimental Details

B.1. Implementation Details

We used mini-batch update to reduce the variance of meta-gradient estimation. Specifically, we ran 64 lifetimes in parallel, each with a randomly sample task and randomly initialised policy parameters. We took the average of the meta-gradients from each lifetime to compute the update to the intrinsic reward parameters (η). We ran 2×10^5 updates to η at training time. All hidden layers in the neural networks used ReLU as the activation function. We used arctan activation on the output of the intrinsic reward. The hyperparameters used for each domain are described in Table 1.

Table 1. Hyperparameters.

Hyperparameters	Empty Rooms	Random ABC	Key-Box	Non-stationary ABC
Time limit per episode	100	10	100	10
Number of episodes per lifetime	200	50	5000	1000
Trajectory length	8	4	16	4
Entropy regularisation	0.01	0.01	0.01	0.05
Policy architecture	Conv(filters=16, kernel=3, strides=1)-FC(64)			
Policy optimiser	SGD	SGD	Adam	SGD
Policy learning rate (α)	0.1	0.1	0.001	0.1
Reward architecture	Conv(filters=16, kernel=3, strides=1)-FC(64)-LSTM(64)			
Reward optimiser	Adam			
Reward learning rate (α')	0.001			
Lifetime VF architecture	Conv(filters=16, kernel=3, strides=1)-FC(64)-LSTM(64)			
Lifetime VF optimiser	Adam			
Lifetime VF learning rate (α')	0.001			
Outer unroll length (N)	5			
Inner discount factor ($\bar{\gamma}$)	0.9			
Outer discounter factor (γ)	0.99			

B.2. Domains

We will consider four task distributions, instantiated within one of the three main gridworld domains shown in Figure 2. In all cases the agent has four actions available, corresponding to moving up, down, left and right. However the topology of the gridworld and the reward structure may vary.

B.2.1. Empty Rooms

Figure 2(a) shows the layout of the *Empty Rooms* domain. There are four rooms in this domain. The agent always starts at the centre of the top-left room. One and only one cell is rewarding, which is called the goal. The goal is invisible. The goal location is sampled uniformly from all cells at the beginning of each lifetime. An episode terminates when the agent reaches the goal location or a time limit of 100 steps is reached. Each lifetime consists of 200 episodes. The agent needs to explore all rooms to find the goal and then goes to the goal afterwards.

B.2.2. ABC World

Figure 2(b) shows the layout of the *ABC World* domain. There is a single 5 by 5 room, with three objects (denoted by A, B, C). All object provides reward upon reaching them. An episode terminates when the agent reaches an object or a time limit of 10 steps is reached. We consider two different versions of this environment:*Random ABC* and *Non-stationary ABC*. In the Random ABC environment, each lifetime has 50 episodes. The reward associated with each object is randomly sampled for each lifetime and is held fixed within a lifetime. Thus, the environment is stationary from an agent's perspective but non-stationary from the reward function's perspective. Specifically, the rewards for A, B, and C are uniformly sampled from $[-1, 1]$, $[-0.5, 0]$, and $[0.0.5]$ respectively. The optimal behaviour is to explore A and C at the beginning of a lifetime

to assess which is the better, and then commits to the better one for all subsequent episode. In the non-stationary ABC environment, each lifetime has 1000 episodes. The rewards for A, B, and C are 1, -0.5, and -1 respectively. The rewards for A and C swap every 250 episodes.

B.2.3. KEY BOX WORLD

Figure 2(c) shows the *Key Box World* domain. In this domain, there is a key and three boxes, A, B, and C. In order to open any box, the agent must pick up the key first. The key has a neutral reward of 0. The rewards for A, B, and C are uniformly sampled from $[-1, 1]$, $[-0.5, 0]$, and $[0, 0.5]$ respectively for each lifetime. An episode terminates when the agent opens a box or a time limit of 100 steps is reached. Each lifetime consists of 5000 episodes.

B.3. Hand-designed near-optimal exploration strategy for Random ABC

We hand-designed a heuristic strategy for the Random ABC domain. We assume the agent has the prior knowledge that B is always bad and A and C have uncertain rewards. Therefore, the heuristic is to go to A in the first episode, go to C in the second episode, and then go to the better one in the remaining episodes in the lifetime. We view this heuristic as an upper-bound because it always finds the best object and can arbitrarily control the agent's behaviour.

C. Pseudocode

We provide an illustrative implementation of two core functions based on JAX (Bradbury et al., 2018). The provided code simulates the interaction between a single agent and an intrinsic reward function. However, in practice, one can use `jax.pmap` and `jax.vmap` to simulate parallel lifetimes with a shared intrinsic reward function.

C.1. Agent Update

```
"""
Copyright 2020 DeepMind Technologies Limited.


Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
"""

import jax
from jax import numpy as jnp
import tree  # https://github.com/deepmind/tree


def inner_update(learner_state, eta, rollout):
  """Inner update function.

  Args:
    learner_state: A namedtuple containing theta, opt_state, and core_state.
      theta is the parameters for the policy and the value function;
      opt_state is the optimizer state of the Adam optimizer;
      core_state is the hidden state for the RNN cores
        of the intrinsic reward function and the lifetime value function.
    eta: The parameters for the intrinsic reward function and the lifetime value function.
    rollout: A trajectory with length T + 1, where T is the inner unroll length.

  Returns:
```

```
    A new learner_state after one inner-loop update.
  """
  theta = learner_state.theta
  opt_state = learner_state.opt_state
  core_state = learner_state.core_state

  # unroll_eta_tp1 unrolls the the intrinsic reward function and
  # the lifetime value function on the rollout. Here tp1 is short for
  # ``t plus 1'' because the length of the rollout is T + 1.
  # The implementation of unroll_eta_tp1 is omitted here.
  (r_in, unused_v_lifetime), new_core_state = unroll_eta_tp1(eta, core_state, rollout)

  # discounted_return_fn is a util function that computes the discounted return in RL.
  # The implementation is omitted as it is not the core of our algorithm.
  # A reference implementation can be found at
  # https://github.com/deepmind/rlax/blob/master/rlax/_src/multistep.py
  returns = discounted_return_fn(
    rewards=r_in[1:],
    discounts=rollout.episode_discount[1:]*episode_discount,
    bootstrap_value=rollout.v[-1])
  advantages = returns - rollout.v[:-1]

  def inner_loss(theta, rollout, returns, advantages):
    """Inner loss function (surrogate).

    Args:
      theta: The parameters of the policy and the value function.
      rollout: A trajectory with length T + 1, where T is the inner unroll length.
      returns: A tensor with shape [T].
        The return for the last state-action in rollout is dropped.
      advantages: A tensor with shape [T].
        The advantage for the last state-action in rollout is dropped.

    Returns:
      A single scalar, the standard actor-critic loss.
    """
    logits, v = theta_apply(theta, rollout.observation[:-1])
    masks = rollout.lifetime_discount[:-1]
    # policy_gradient_loss_fn and entropy_loss_fn are util functions that
    # compute the policy gradient loss and entropy regularisation loss.
    # The implementations are omitted as they are not the core of our algorithm.
    # A reference implementation can be found at
    # https://github.com/deepmind/rlax/blob/master/rlax/_src/policy_gradients.py
    pg_loss = policy_gradient_loss_fn(
      logits=logits,
      actions=rollout.action[:-1],
      advantages=advantages,
      weights=masks,
      backprop=True)
    entropy_loss = entropy_loss_fn(logits=logits, weights=masks)
    baseline_loss = 0.5 * jnp.sum(jnp.square(v - returns) * masks)
    return pg_loss + baseline_cost * baseline_loss + inner_entropy_cost * entropy_loss

  # jax.grad is a JAX primitive which derives the gradient of a function.
  dl_dtheta = jax.grad(inner_loss)(theta, rollout, returns, advantages)

  # inner_opt_update updates the optimiser statistics.
  update, new_opt_state = inner_opt_update(dl_dtheta, opt_state)
  new_theta = tree.map_structure(lambda p, u: p + u, theta, update)

  new_learner_state = LearnerState(
    theta=new_theta,
    opt_state=new_opt_state,
    core_state=new_core_state)
  return new_learner_state
```

C.2. Intrinsic Reward and Lifetime Value Function Update

```
"""
Copyright 2020 DeepMind Technologies Limited.


Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
"""

import jax
from jax import numpy as jnp
import tree  # https://github.com/deepmind/tree


def outer_update(learner_state, meta_learner_state, rollouts):
  """Outer update function.

  Args:
    learner_state: A namedtuple containing theta, opt_state, and core_state.
      theta is the parameters for the policy and the value function;
      opt_state is the optimizer state of the Adam optimizer;
      core_state is the hidden state for the RNN cores
        of the intrinsic reward function and the lifetime value function.
    meta_learner_state: A namedtuple containing eta and opt_state.
      eta is the parameters for the intrinsic reward and the lifetime value function;
      opt_state is the optimizer state of the Adam optimizer;
    rollouts: A sequence of N + 1 trajectory, each with length T + 1,
      where N is the outer unroll length and T is the inner unroll length.

  Returns:
    A new meta_learner_state after one outer-loop update.
  """

  def outer_loss(eta, learner_state, rollouts):
    """Outer loss function.

    Args:
      eta: The parameters for the intrinsic reward function and the lifetime value
                                                          function.
      learner_state: A namedtuple containing theta, opt_state, and core_state.
        theta is the parameters for the policy and the value function;
        opt_state is the optimizer state of the Adam optimizer;
        core_state is the hidden state for the RNN cores
        of the intrinsic reward function and the lifetime value function.
      rollouts: A sequence of N + 1 trajectory, each with length T + 1,
        where N is the outer unroll length and T is the inner unroll length.

    Returns:
      A single scalar, the sum of the losses for the intrinsic reward function
      and the lifetime value function.
    """
    # Unroll the inner-loop updates.
```

```
    all_r_ex = []
    all_lifetime_discount = []
    all_masks = []
    all_action = []
    all_logits = []
    all_v_lifetime = []
    core_state = learner_state.core_state
    for rollout in rollouts:
      all_r_ex.append(rollout.reward[1:])
      all_lifetime_discount.append(rollout.lifetime_discount[1:])
      all_masks.append(rollout.lifetime_discount[:-1])
      all_action.append(rollout.action[:-1])

      logits, unused_v = theta_apply(learner_state.theta, rollout.observation[:-1])
      all_logits.append(logits)

      # unroll_eta_tp1 unrolls the the intrinsic reward function and
      # the lifetime value function on the rollout. Here tp1 is short for
      # ''t plus 1'' because the length of the rollout is T + 1.
      # The implementation of unroll_eta_tp1 is omitted here.
      (unused_r_in, v_lifetime), core_state = unroll_eta_tp1(eta, core_state, rollout)
      all_v_lifetime.append(v_lifetime[:-1])
      bootstrap_value_v_lifetime = v_lifetime[-1]

      learner_state = inner_update(learner_state, eta, rollout)

    # Compute lifetime return and advantage.
    #
    # discounted_return_fn is a util function that computes the discounted return in RL.
    # The implementation is omitted as it is not the core of our algorithm.
    # A reference implementation can be found at
    # https://github.com/deepmind/rlax/blob/master/rlax/_src/multistep.py
    all_r_ex = jnp.concatenate(all_r_ex)
    all_lifetime_discount = jnp.concatenate(all_lifetime_discount)
    lifetime_return = discounted_return_fn(
      rewards=all_r_ex,
      discounts=all_lifetime_discount*lifetime_discount,
      bootstrap_value=bootstrap_value_v_lifetime)
    all_v_lifetime = jnp.concatenate(all_v_lifetime)
    lifetime_advantage = lifetime_return - all_v_lifetime

    # Outer-loop policy gradient loss.
    # Note that the updated policy is evaluated by the *next* rollout.
    #
    # policy_gradient_loss_fn and entropy_loss_fn are util functions that
    # compute the policy gradient loss and entropy regularisation loss.
    # The implementations are omitted as they are not the core of our algorithm.
    # A reference implementation can be found at
    # https://github.com/deepmind/rlax/blob/master/rlax/_src/policy_gradients.py
    valid_logits = jnp.concatenate(all_logits[1:])
    valid_action = jnp.concatenate(all_action[1:])
    pg_mask = jnp.concatenate(all_masks[1:])
    valid_lifetime_advantage = lifetime_advantage[-len(valid_logits):]
    pg_loss = policy_gradient_loss_fn(
      logits=valid_logits,
      actions=valid_action,
      advantages=valid_lifetime_advantage,
      weights=pg_mask,
      backprop=False)
    entropy_loss = entropy_loss_fn(logits=valid_logits, weights=pg_mask)

    # Lifetime value function regression loss.
    # Note that we do NOT update the lifetime value function on the last rollout,
    # which is for computing the meta-gradient only.
    valid_v_lifetime = all_v_lifetime[-len(valid_logits):]
```

```
    valid_lifetime_return = lifetime_return[:len(valid_logits)]
    baseline_mask = jnp.concatenate(all_masks[:-1])
    baseline_loss = 0.5 * jnp.sum(
      jnp.square(valid_v_lifetime - valid_lifetime_return) * baseline_mask)

    return pg_loss + baseline_cost * baseline_loss + outer_entropy_cost * entropy_loss

  # jax.grad is a JAX primitive which derives the gradient of a function.
  eta = meta_learner_state.eta
  dm_deta = jax.grad(outer_loss)(eta, learner_state, rollouts)

  # outer_opt_update updates the optimiser statistics.
  opt_state = meta_learner_state.opt_state
  update, new_opt_state = outer_opt_update(sum_dm_deta, opt_state)
  new_eta = tree.map_structure(lambda p, u: p + u, eta, update)

  new_meta_learner_state = MetaLearnerState(
      eta=new_eta,
      opt_state=new_opt_state,
  )
  return new_meta_learner_state
```

References

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., and Wanderman-Milne, S. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.