Implementation of multilevel PPO in stable baselines 3

Author: Atish Dixit

October 11, 2023

The multilevel PPO algorithm is implemented using the Stable Baselines3 (SB3) library, which is a set of reliable implementations of reinforcement learning algorithms in PyTorch. The codes for the multilevel implementation can be found in the fork: `https://github.com/atishdixit16/stable-baselines3`. In the following text, the implementation of the classical PPO in SB3 is explained in detail. Then it is followed by additional implementations corresponding to the multilevel PPO algorithm.

0.1 Classical PPO implementation in stable baselines 3

RL framework consists of the environment $\mathcal{E}$ which is governed by a Markov decision process described by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mu \rangle$. Here, $\mathcal{S} \subset \mathbb{R}^{n_s}$ is the state-space, $\mathcal{A} \subset \mathbb{R}^{n_a}$ is the action-space, $\mathcal{P}(s'|s,a)$ is a Markov transition probability function between the current state s and the next state s' under action a and $\mathcal{R}(s,a,s')$ is the reward function. The function $\mu(s)$ returns a state from the initial state distribution if s is the terminal state of the episode; otherwise, it returns the same state s. The goal of reinforcement learning is to find the policy $\pi_\theta(a|s)$ to take an optimal action a when in the state s, by exploring the state-action space with what are called agent-environment interactions. Figure 1 shows a typical schematic of such agent-environment interaction. The term *agent* refers to the controller that follows the policy $\pi_\theta(a|s)$ while the *environment* consists of the transition function, $\mathcal{P}$, and the reward function, $\mathcal{R}$.

The algorithm 1 delimits the simplified implementation of the PPO algorithm in SB3. The algorithm's inputs are: environment E, number of actors N, number of steps in each policy iteration T, batch size M ($\leq NT$) and number of epochs K. The data obtained through the rollouts of agent-environment interactions is stored in a buffer named RolloutBuffer in the format $[s, a, r, d, V, L_{\text{old}}, R, A]$, where the notation is

- s: state,
- a: action,
- r: reward,
- d: episode terminal boolean (done),
- V: Value function (obtained from policy network rollout),
- L_{old}: log probability value, $\log(\pi_{\theta_{old}}(a|s))$

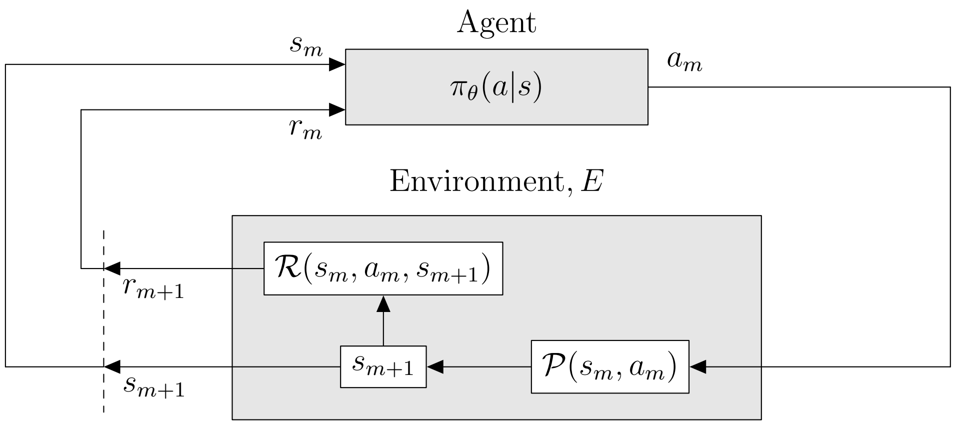


Figure 1: A typical agent-environment interaction for classical framework

- R: Return value (obtained using generalized advantage estimation),
- A: Advantage function (obtained using generalized advantage estimation).

RolloutBuffer accumulates in total $N \times T$ rows of the above data in each iteration. At the beginning of each iteration, the function **CollectRollouts** is used to fill in the data in RolloutBuffer. The total of $N \times T$ data rows is divided into batches of size M, each using the function **GetBatches**. The actor loss term L_a, the value loss term L_v and the entropy loss term L_e are calculated for each such batch using the function **ComputeBatchLosses**. Finally, a Monte Carlo estimate for the loss term is computed as follows.

$$\text{loss}_{\text{MC}} = mean[L_a + L_v + L_e],$$

which is used to update the policy parameters using automatic differentiation. This is done using the function **UpdatePolicy** and is performed K times for every batch.

Algorithm 1 PPO implementation in stable baselines

```
Input: E, N, T, M, K
E.reset()
Generate empty RolloutBuffer
for iteration, i = 1, 2, ... do
    CollectRollouts(E, N, T, RolloutBuffer)
    for epoch = 1, 2, ..., K do
        for batch in GetBatches(RolloutBufferArray, M): do
            L_a, L_v, L_e = ComputeBatchLosses(batch)
            loss_MC = mean [L_a + L_v + L_e]
            UpdatePolicy( loss_MC )
        end for
    end for
end for
```

The algorithm 2 delineates the steps of the function **CollectRollouts**. For every timestep, the data is obtained using policy rollout, environment transition (using *step* function) and generalized

Algorithm 2 **CollectRollouts**$(E, N, T,$ RolloutBuffer$)$

Information: a RolloutBuffer consists of following data: $[s, a, r, d, V, L_{\text{old}}, R, A]$
reset RolloutBuffer (i.e. empty the buffer)
for t in range(T): **do**
    rollout current state s, through policy network to obtain a, V, $L_{\text{old}}(a)$ on N actors
    if s is terminal, $s = E.reset()$
    $s', r, d, \cdot = E.step(a)$ on N actors
    compute R and A using GAE
    add $[s, a, r, d, V, L_{\text{old}}, R, A]$ in the RolloutBuffer
end for

advantage estimation (GAE) computation on all N actors and stored in the RolloutBuffer. Finally, **ComputeBatchLosses** function is illustrated in the algorithm 3. The algorithm lists steps to compute actor loss term L_a, value loss term L_v and entropy loss term L_e for the given batch. Note that the loss terms are the vectors of dimension M, which are added later, and its mean is treated as the final loss term. The mean function in this process indicates the *Monte Carlo* estimator of the PPO loss term.

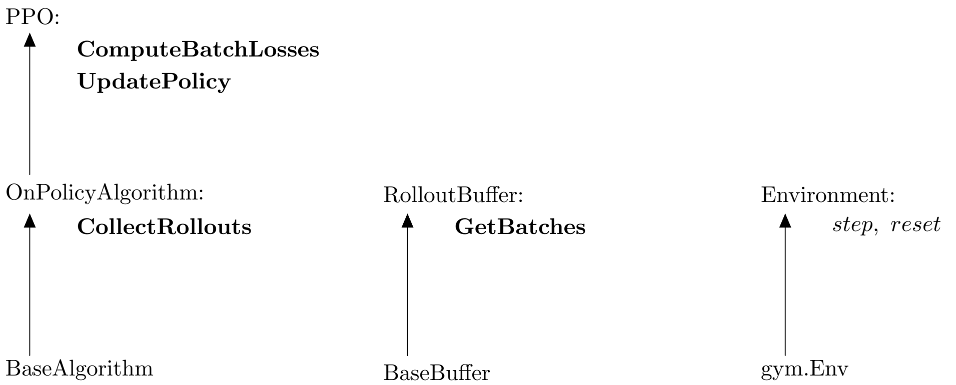
Algorithm 3 **ComputeBatchLosses**(batch)

Information: a batch consists of M rows following data: $[s, a, V, L_{\text{old}}, R, A]$
compute V_{now} and $L_{\text{now}}(a)$ by rolling out s through policy network
compute ratio, $r_t = \exp\left(L_{\text{now}} - L_{\text{old}}\right)$
compute $L_1 = Ar_t$ and $L_2 = A[\text{clip}(r_t, 1-\epsilon, 1+\epsilon)]$
$L_a = \min\left(L_1, L_2\right)$
$L_v = C_v|V_{\text{now}} - R|^2$ (C_v is value loss term coefficient)
$L_e = -C_e L_{\text{now}}$ (C_e is entropy loss term coefficient)
return L_a, L_v, L_e

The class inheritance schema used in this implementation is shown in Figure 2. The stable baselines use some more classes like Policy, Callbacks etc. but we present only the ones relevant to this discussion. **CollectRollouts** function belongs to OnPolicyAlgorithm which is the child of the BaseAlgorithm class and the parent of the PPO class. The functions **ComputeBatchLosses** and **UpdatePolicy** belong to the PPO class. BaseBuffer is the parent class for the RolloutBuffer class that contains the function **GetBatches**. The Environment class (which is a child of the gym.Env class) contains functions such as *step* and *reset* corresponding to the transition function $\mathcal{P}$ and the initial state function μ, respectively.

0.2 Multilevel PPO implementation in stable baselines 3

Figure 3 illustrates a typical agent-environment interaction in multilevel PPO implementation. Multiple levels of environment are represented with $E^1, E^2, \ldots, E^{L-1}$ so that the computational cost of $\mathcal{P}^l$ and the accuracy of $\mathcal{R}^l$ are lower than $\mathcal{P}^{l+1}$ and $\mathcal{R}^{l+1}$, respectively. The environment corresponding to the grid fidelity factor l consists of a transition function $\mathcal{P}_l$, which is achieved by discretizing the dynamical system, and a reward function $\mathcal{R}_l$. The policy network is designed with states s^L and controls a^L, corresponding to the environment E^L. As a result, state s^l_{m+1}, in the environment, E^l passes through the mapping ψ^L_l which maps the state from level l to level L. Similarly, the action



(a) algorithm class architecture (b) buffer class architecture (c) environment class architecture

Figure 2: Object-oriented design for the stable baselines implementation of PPO algorithm

obtained from the policy network is passed through a mapping operator ϕ_L^l, which maps the action from the level L to the level l.

Algorithm 4 illustrates the pseudocode for multilevel implementation of the PPO algorithm in the stable baselines library. The inputs are the same as in classical PPO implementation except multi-level variables are provided as an array of length L: environments at each level $\boldsymbol{E} = [E^1, E^2 \dots . E^L]$, number of actors N, number of steps in each level $\boldsymbol{T} = [T^1, T^2, \dots, T^L]$, number of batches in each level $\boldsymbol{M} = [M^1, M^2, \dots, M^L]$ (such that $NT^l \leq M^l$ and $T^1/M^1 = \dots = T^L/M^L$) and number of epochs K. In multilevel implementation, we formulate the loss term's estimate using multilevel Monte Carlo which is given as

$$\text{loss}_{\text{MLMC}} = \sum_{l=1}^{L} mean\left[(L_a^l - \tilde{L}_a^{l-1}) + (L_v^l - \tilde{L}_v^{l-1}) + (L_e^l - \tilde{L}_e^{l-1}) \right],$$

where $\tilde{L}_a^0, \tilde{L}_v^0$ and $\tilde{L}_e^0$ are set to zero. The outline of a typical agent-environment interaction to obtain synchronized samples of levels l and $l-1$ is illustrated in Figure 3. We use arrays of RolloutBuffers for each level, and each RolloutBuffer^l that collects rollouts at level l has a synchronized buffer SyncRolloutBuffer^l that collects corresponding synchronized data at level $l-1$. This is achieved using the function **CollectRollouts**. Figure 4 illustrates the RolloutBufferArray and SyncRollout-BufferArray used in this algorithm. Furthermore, the **GetBatches** function is used to generate an array of batches, which is used to compute the multilevel Monte Carlo estimate of the loss term. The batch array consists of in total NT^L/M^L batches, where each batch consists of L batches from RolloutBuffers and L batches from SyncRolloutBuffers. Figure 5 illustrates the batch array used in the algorithm. The batch^l, syncBatch^{l-1} from RolloutBuffer^l, SyncRolloutBuffer^l are used to compute the $\text{loss}_{\text{MLMC}}$ terms on the level l. In every batch, these terms are computed at each level and added to obtain $\text{loss}_{\text{MLMC}}$, which is used to update the policy network parameters using the function **UpdatePolicy**.

The algorithm 5 delimits the function **CollectRollouts** used in multilevel implementation. At each level l the RolloutBuffer^l is filled with the data, and the corresponding synchronized data at the level $l-1$ is filled in the SyncRolloutBuffer^l. Since L_a^0, L_v^0 and L_e^0 are set to zero, the data in SyncRolloutBuffer^1 are filled with None values. The mapping functions $\psi_l^{l'}$ and $\phi_l^{l'}$ are implemented

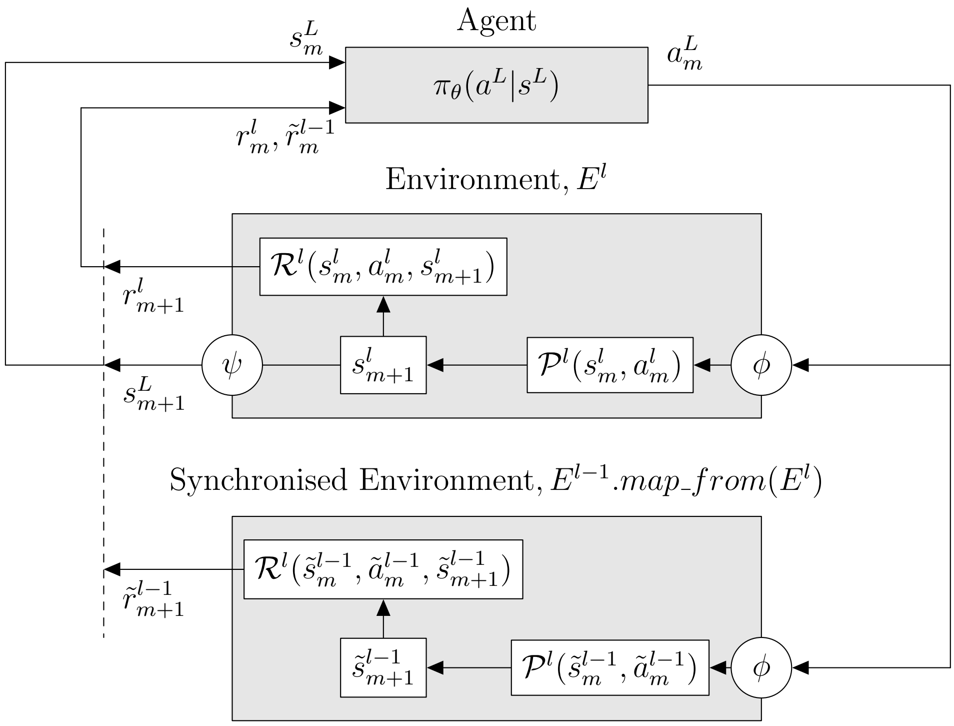


Figure 3: A typical agent-environment interaction for an environment on level l synchronized with environment on level $l-1$

as a set of functions in the definition of the environment E^l. As a result, the mapping of state (ψ_l^L) and action (ϕ_L^l) to and from the policy + value network is denoted with shorthand notation ψ and ϕ, respectively. Synchronization of state from level l to l' is indicated by map_from function that maps an environment E^l to another environment at level l', denoted as $E^{l'}$. Algorithm 6 illustrates the pseudocode for the **GetBatches** function, which creates mini-batches (as illustrated in Figure 5) from collected data in RolloutBufferArray and SyncRolloutBufferArray.

The class inheritance schema used in the multilevel implementation is shown in figure 6. **CollectRollouts** function belongs to OnPolicyAlgorithmMultilevel which is the child of BaseAlgorithm class and the parent of the PPO_ML class. The functions **ComputeBatchLosses** and **UpdatePolicy** belong to the class PPO_ML. BaseBuffer is the parent class for the RolloutBuffer class that contains the function **GetBatches**. The environment class architecture for multilevel framework is similar to that for classical framework except for the additional mapping functions ψ, ϕ and map_from. The updated definitions of the classes and functions are highlighted in red in figure 6.

$$
\begin{bmatrix} \text{RolloutBuffer}^1 \\ \text{RolloutBuffer}^2 \\ \cdot \\ \cdot \\ \text{RolloutBuffer}^L \end{bmatrix}
\begin{bmatrix} \text{SyncRolloutBuffer}^1 \\ \text{SyncRolloutBuffer}^2 \\ \cdot \\ \cdot \\ \text{SyncRolloutBuffer}^L \end{bmatrix}
$$

Figure 4: RolloutBufferArray (on left) and SyncRolloutBufferArray (on right). SyncRolloutBuffer^l consists of synchronized data of RolloutBuffer^l with level l to a level $l-1$. Each buffer with level l consists of $N \times T_l$ rows of data in $[s, a, r, d, V, L_{\text{old}}, R, A]$ format.

$$
\begin{bmatrix}
\begin{bmatrix} \text{batch}^1, \text{syncBatch}^0 \\ \text{batch}^2, \text{syncBatch}^1 \\ \cdot \\ \cdot \\ \text{batch}^L, \text{syncBatch}^{L-1} \end{bmatrix}
\begin{bmatrix} \text{batch}^1, \text{syncBatch}^0 \\ \text{batch}^2, \text{syncBatch}^1 \\ \cdot \\ \cdot \\ \text{batch}^L, \text{syncBatch}^{L-1} \end{bmatrix}
\begin{bmatrix} \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \end{bmatrix}
\begin{bmatrix} \text{batch}^1, \text{syncBatch}^0 \\ \text{batch}^2, \text{syncBatch}^1 \\ \cdot \\ \cdot \\ \text{batch}^L, \text{syncBatch}^{L-1} \end{bmatrix}
\end{bmatrix}
$$

Figure 5: batch_array which is achieved from **GetBatches** function. It consists of in total NT_l/M_l batches as shown with the columns of the array. Each such batch consists of L batches from RolloutBuffers (denoted by batch^l) and SyncRolloutBuffers (denoted by syncBatch^{l-1}). batch^l and SyncBatch^{l-1} consists of M^l rows of data in the format, $[o, a, V, L_{\text{old}}, R, A]$.

Algorithm 4 Multilevel proximal policy optimization pseudocode

Input: $\boldsymbol{E}, N, \boldsymbol{T}, \boldsymbol{M}, K$
$E^1.reset()$
Generate empty RolloutBufferArray, SyncRolloutBufferArray
for iteration, $i = 1, 2, \ldots$ **do**
  CollectRollouts($\boldsymbol{E}, N, \boldsymbol{T}$, RolloutBufferArray, SyncRolloutBufferArray)
  for $epoch = 1, 2, \ldots, K$ **do**
    for batch_array in **GetBatches**(RolloutBufferArray, SyncRolloutBufferArray, $\boldsymbol{M}$): **do**
      $\text{loss}_{\text{MLMC}} = 0$
      for batch^l, syncBatch^{l-1} in batch_array **do**
        $L_a^l,\ L_v^l,\ L_e^l =$ **ComputeBatchLosses**(batch^l)
        if $l > 1$ **then**
          $\tilde{L}_a^{l-1},\ \tilde{L}_v^{l-1},\ \tilde{L}_e^{l-1} =$ **ComputeBatchLosses**(syncBatch^{l-1})
        else
          $\tilde{L}_a^{l-1},\ \tilde{L}_v^{l-1},\ \tilde{L}_e^{l-1} = 0$
        end if
        $L^l = mean\left[(L_a^l - \tilde{L}_a^{l-1}) + (L_v^l - \tilde{L}_v^{l-1}) + (L_e^l - \tilde{L}_e^{l-1})\right]$
        $\text{loss}_{\text{MLMC}} = \text{loss}_{\text{MLMC}} + L^l$
      end for
      UpdatePolicy($\text{loss}_{\text{MLMC}}$)
    end for
  end for
end for

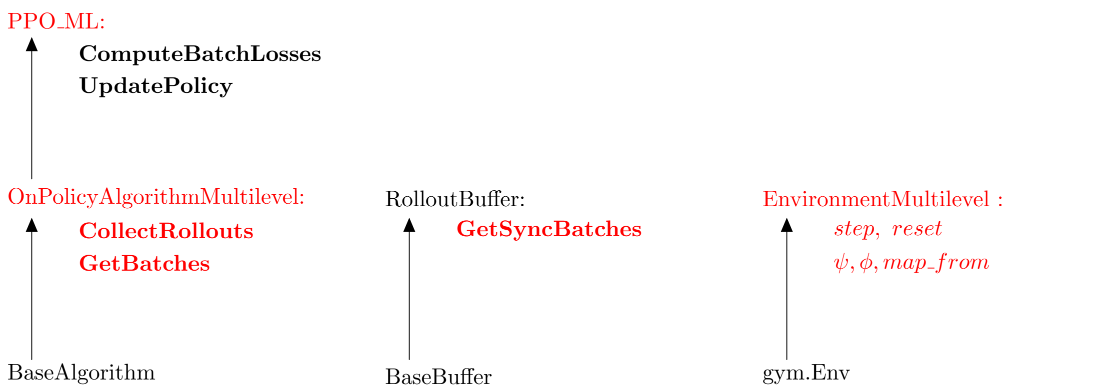
Algorithm 5 **CollectRollouts**($\boldsymbol{E}, N, \boldsymbol{T}$, RolloutBufferArray, SyncRolloutBufferArray)

Information: a RolloutBuffer consists of following data: $[s, a, r, d, V, L_{\text{old}}, R, A]$
reset RolloutBufferArray, SyncRolloutBufferArray (i.e. empty the buffers)
for T^l, E^l, RolloutBuffer^l, SyncRolloutBuffer^l in $\boldsymbol{E}, \boldsymbol{T}$, RolloutBufferArray, SyncRolloutBufferArray **do**
  if $l > 1$ **then**
    $E^l.map\_from(E^{l-1})$
  end if
  for t in range(T^l): **do**
    $s^l = E^l.reset()$ if s^l is terminal
    $s^L = E^l.\psi(s^l)$
    $a^L = \pi_\theta(a^L|s^L)$
    $a^l = \phi(a^L)$
    compute V^l and $L_{\text{old}}(a^L)$
    $\cdot, r^l, d^l, \cdot = E^l.step(a^l)$ on N actors
    compute R^l and A^l using GAE
    add $[s^l, a^l, r^l, d^l, V^l, L^l_{\text{old}}, R^l, A^l]$ in the RolloutBuffer^l

    if $l > 1$ **then**
      $E^{l-1}.map\_from(E^l)$
      $\tilde{s}^L = E^{l-1}.\psi(\tilde{s}^{l-1})$
      $\tilde{a}^{l-1} = a^l$
      $\tilde{a}^L = \pi_\theta(\tilde{a}^L|\tilde{s}^L)$
      $\tilde{a}^{l-1} = E^{l-1}.\phi(\tilde{a}^L)$
      compute $\tilde{V}^{l-1}$ and $\tilde{L}_{\text{old}}(a^L)$
      $\cdot, \tilde{r}^{l-1}, \cdot, \cdot = E^{l-1}.step(\tilde{a}^{l-1})$ on N actors
      compute $\tilde{R}^{l-1}$ and $\tilde{A}^{l-1}$ using GAE
      add $[\tilde{s}^{l-1}, \tilde{a}^{l-1}, \tilde{r}^{l-1}, \tilde{d}^l, \tilde{V}^{l-1}, \tilde{L}^{l-1}_{\text{old}}, \tilde{R}^{l-1}, \tilde{A}^{l-1}]$ in the SyncRolloutBuffer^l
    else
      add [None, …, None] in the SyncRolloutBuffer^l
    end if
  end for
end for

Algorithm 6 **GetBatches**(RolloutBufferArray, SyncRolloutBufferArray, $\boldsymbol{M}$)

set batch_array to an empty array
for RolloutBuffer^l, SyncRolloutBuffer^l, M^l in RolloutBufferArray, SyncRolloutBufferArray, $\boldsymbol{M}$ **do**
  set batches to an empty array
  for batch^l, batch^{l-1} in **GetSyncBatches**(RolloutBuffer^l, SyncRolloutBuffer^l, M^l) **do**
    batches.append([batch^l, batch^{l-1}])
  end for
  batch_array.*append*(batches)
end for
return batch_array



(a) algorithm class architecture (b) buffer class architecture (c) environment class architecture

Figure 6: Object-oriented design for the stable baselines implementation of multilevel PPO algorithm. The updated (from classical PPO implementation) definitions of functions and classes are highlighted in red colour.