Project Milestone#2: TRAIN

Anani

aanani@mpi-sws.org

I.1

Code is included in the submission.

I.2

(a) State Feature Representation

The state type in this project is represented as a multi-binary vector that contains both the post-grid and the pre-grid. Typically, the state must account for the following information:

- Agent's location (i,j) and direction on both the post-grid and the pre-grid.
- Markers locations in the post-grid and the pre-grid.
- Wall locations.

For the sake of experimentation, three states were designed that differ in terms of compactness, namely $S0$, $S1$, and $S2$. The said states have the following structure:

- **A compact state** (S_0)**:**
  The compact state visualization is shown in Figure 1 below.

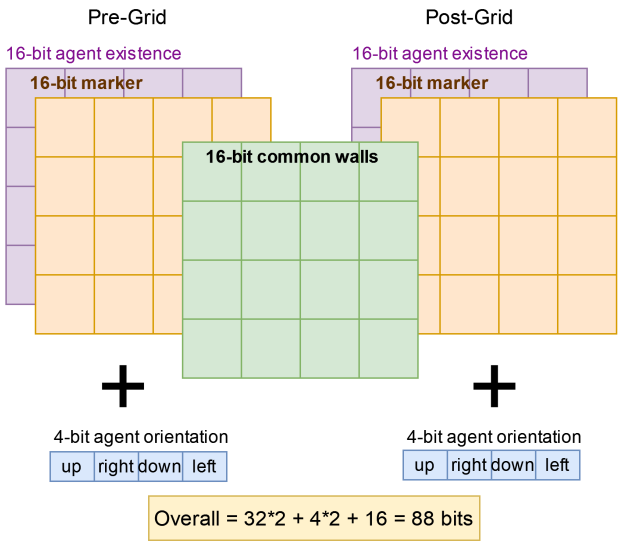

Figure 1: Compact State Visualization

.

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

More precisely:
For every cell, 1-bit(agent existence) + 1-bit(marker). [2 bits per cell]
For every grid, 4-bit(agent orientation). [$2 \cdot 16 + 4 = 36$ bits per grid]
For both post-grid and pre-grid, 16-bit(wall). [A bit for every cell]

Since we need to have the pre-grid and the post-grid in the state, we need to account for two grids. However, the wall bits shall be global accross them, since they remain exactly the same, no matter what action is taken, in the post-grid and the pre-grid. So, we end up with:

state vector size of $36 \cdot 2 + 16 = 88$ **bits.**

The state space size (all possible states; including potentially unsolvable ones) in this case will be

$$|S| = 2^{88}$$

- **A less compact state** (S_1)**:**
  For every cell, 1-bit(agent existence) + 1-bit(wall) + 1-bit(marker). [3 bits per cell]
  For every grid, 4-bit(agent orientation). [$3 \cdot 16 + 4 = 52$ bits per grid]
  Since we need to have the pre-grid and the post-grid in the state, we end up with:

  state vector size of $52 \cdot 2 = 104$ **bits.**

  The state space size in this case (all possible states; including potentially unsolvable ones) will be:

  $$|S| = 2^{104}$$

- **Non-compact state** (S_2): For every cell, 4-bit(orientation and agent existence) + 1-bit(wall) + 1-bit(marker) [6 bits per cell]
  Since we need to have the pre-grid and the post-grid in the state, we end up with:

  state vector size of $16 \cdot 6 \cdot 2 = 192$ **bits.**

(b) RL Algorithms Details

Multiple Policy Gradient Algorithms were explored during the training to find the best performing one on our Karel task. Experiments were done using the following algorithms:

- **Advantage Actor Critic (A2C)**: is a synchronous policy gradient algorithm where an actor has a policy π and the critic estimates the state-value function v to judge the actor's actions. The implementation is inspired by the stable-baselines project [1] and our pgneural assignment of Week7. PyTorch was used as the main library for MLP policy in A2C. Moreover, the A2C class supports learning from scratch or learning by demonstrations provided from an expert. The A2C algorithm implemented has the following pseudo-code found in the Appendix 1. It also uses **Monte Carlo** for the return calculation. Moreover, the implementation supports updates every n episodes, not strictly every 1 episode.

- **Actor-Critic with Experience Replay (ACER):** is an Actor-Critic DRL algorithm that performs experience replay [2]. Its code is used right from the stable-baselines project project [1].
- **Soft Actor Critic (SAC):** is an off-policy algorithm that optimizes a stochastic policy while benefiting from target policy smoothing. To work with our environment which has a discrete action space, the continuous action space version of SAC illustrated by OpenAI here is adjusted to handle categorical outputs. Slight changes were performed on the policy updates and the MLP architecture. Moreover, support for learning from demonstration was added to accelerate the training process.
- **Proximal Policy Optimization (PPO1)**: is an algorithm that aims at optimizing the improvement step on a policy without overshooting. It contains two main terms: the *PPO-Penalty* and *PPO-Clip*. *PPO-Penalty* aims at solving a KL-constrained update by penalizing the KL-divergence in the objective function via learning the penalty coefficient while training. *PPO-Clip* relies on clipping in the objective function as to not let the new policy get too far from the old policy. Its code in this project is used right from the stable-baselines project [1].

- **Behavioral Cloning using Feed Forward Neural Network (BC-FFNN):** this algorithm is inspired by the *imitation* python library, which treats the reinforcement learning problem as a supervised problem with point-to-point training on a neural network policy to map states to desired actions given an expert dataset.

  Figure 2 has the hierarchy of implemented algorithms, which will be discussed in section I.4.

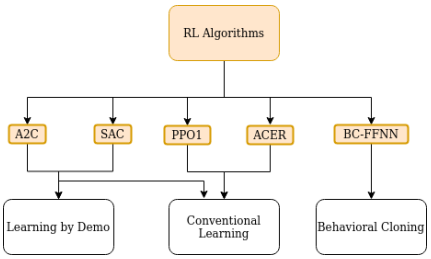

Figure 2: RL Algorithms Hierarchy

.

Reward

A simple reward was used throughout all experiments, which is defined as follows:

- R(s, a) = 1 for termination on $current\_grid = post\_grid$ and $a = finish$.
- R(s, a) = 0 otherwise

(c) Network Architecture for Agent's Neural Policy

The neural policy used across all mentioned policy gradient algorithms is a Multi-Layer Perceptron policy implemented using PyTorch. A CNN policy was experimented with briefly using a 3D-state representation (4x4x11); however, it performed poorly due to its small dimensionality, which is not suited for CNN applications. Multiple experiments were conducted on the network architecture by varying its depth and density to observe the best performing model. More on the netowrk architectures used for the best agent in section I.4.

(d) Agents Hyperparameters

The agents I am experimenting with have the following hyperparametrs:

Agent	Hyperparameters
A2C **ACER** **SAC**	γ α_{actor} α_{critic} EPOCHS MAX_EPISODES learning_rate update_every_n Optimizer
A2C	clip_range
A2C **BC-FFNN**	$entropy\_ceoff$
BC-FFNN	learning_rate MAX_EPISODES EPOCHS num_layers layers_sizes Optimizer

I.3: Training Acceleration Techniques

(a) Behavioral Cloning (BC):

Dealing with the problem of imitation learning as a supervised learning problem, given that a ground truth solution dataset exists for the problem. It essentially trains a policy network, given expert trajectories (state-action pairs), to reproduce such expert behavior (hence, clone it). So, given a state, the action taken by the policy is evaluated w.r.t the action taken by the expert/ground-truth, until the policy behaves the same as the expert/ground-truth. For this project, the expert trajectories we have are the optimal sequences provided in the dataset. Behavioral-Cloning can be applied on any policy-gradient algorithm. (Experiments on BC in Section I.4) The algorithm is the following:

1. Generate trajectory demonstrations from the expert dataset.
2. Treat demonstrations as independent and identically distributed state-actions pairs: $(s_0^{'}, a_0^{'}), (s_1^{'}, a_1^{'}), (s_2^{'}, a_2^{'}), (s_3^{'}, a_3^{'}), ..etc$
3. Train a policy π_θ by minimizng a loss function $L(a', \pi(s))$ (cross entropy).

(b) Learning by Demonstration:

Generating episode rollouts using the optimal actions of a given expert (optimal sequences) provided in the dataset. By doing this, the loss term of the policy will be calculated using the probability the agent produces for the optimal action. Hence, it should learn a probability distribution that samples a good action given a state based on previous demonstrations it was provided with during training. In algorithms such as A2C and SAC, the actor learns the optimal actions, while the critic receives a discounted return term based on the expert rollouts, hence, it becomes better for further learning, or even online environment probing to judge actions given by the actor. (Experiments on Learning by Demonstration in Section I.4)

(c) Curriculum Design

I have experimented with a naive version of curriculum design by inputting tasks with ascending difficulty metric to the expert (e.g., optimal sequence length, number of putMarker/pickMarker, ... etc). However, the curriculum design performance was very comparable to the experiments without curriculum design.

I.4: Results

State Choice: To find which state representation works best with neural policies, preliminary experiments were conducted across multiple RL algorithms (ACER, PPO1, A2C) using all three states. Figure 3 shows the models performance in terms of solved tasks percentage given the three said algorithms with each using every state of the three states (3x3 experiments). No acceleration techniques were used here; all agents are learning on their own for 3 million steps. It is clear that every algorithm outperforms itself whenever using $S0$ against $S1$ and $S2$. The best performing agent in terms of solved tasks is A2C using $S0$, scoring 22.21% solved tasks, while PPO1 using state $S0$ is the best in terms of optimally solved tasks, scoring 16.2%. Hence, as a preliminary conclusion, the rest of the experiments are conducted using the state $S0$ (the compact state).

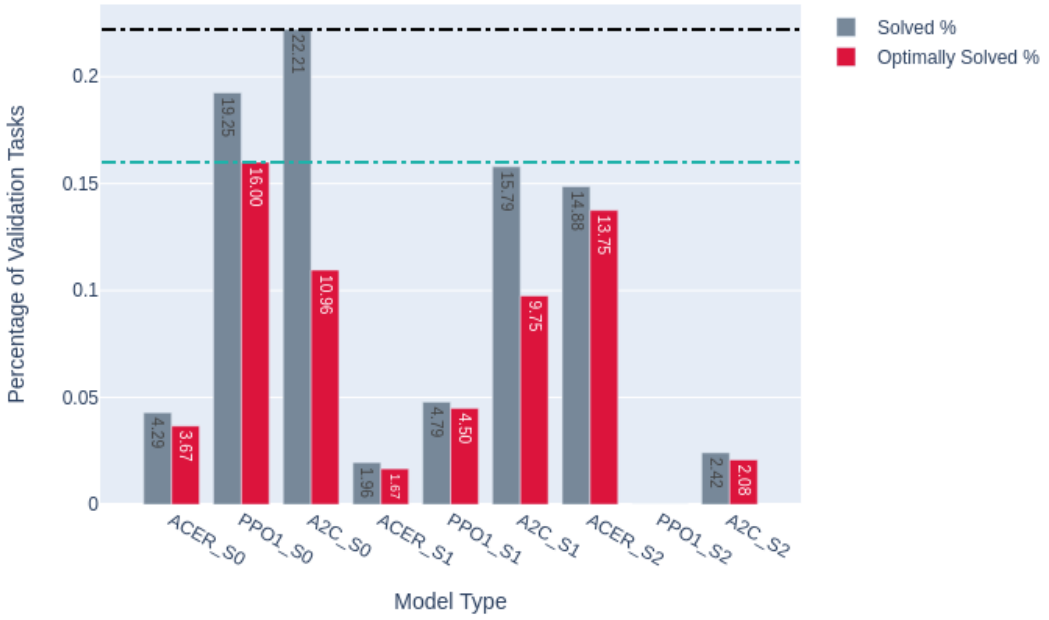

Figure 3: Percentages of Solved Tasks and Optimally Solved Tasks Using ACER, A2C and PPO1 algorithms Across States $S0$, $S2$ and $S3$ (3M steps)

.

Learning without Demonstration: The previous Figure 3 shows the learning without demonstration performance on 3M steps; which takes a quite significant amount of time with slow progress. Two acceleration techniques were explored, whose results are discussed in the coming two sections.

Learning with Demonstration: To guide the learning process of some of the best-performing agents (that learn relatively well without demonstration), learning with demonstration was used in both **SAC** and **A2C**. To recall, the way learning with demonstration is implemented is via providing actions from the expert during generating the episodes rollouts and taking the loss of both the critic and the actor w.r.t their output at the expert's action (See learn_by_demo parts in the pseudo-code of A2C (Appendix 1)).

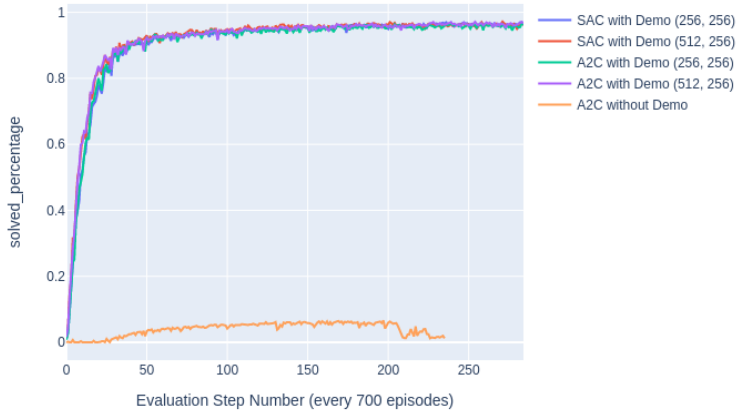

Figure 4: Percentage of Solved Tasks Vs. Evaluation Steps Across Multiple A2C and SAC Variations

.

To highlight the difference between learning with and without demonstration during training, Figure 4 shows that the accuracy of A2C learning without demonstration across evaluation steps during training is very low compared to other variations of A2C and SAC that learn with demonstration. In fact, learning with demonstration boosted both algorithms performance to reach up to $\sim > 96\%$ of solved tasks in 5 epochs.

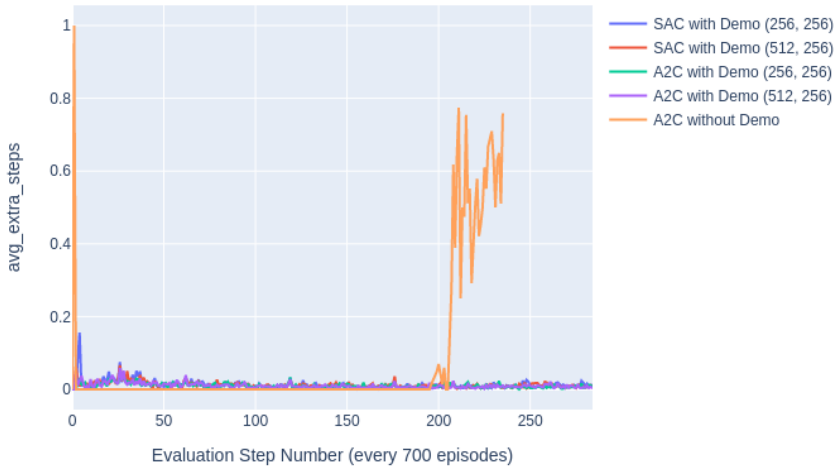

Figure 5: Average Extra Steps Vs. Evaluation Steps Across SAC and A2C variations

.

Furthermore, Figure 5 shows the average extra steps taken by each algorithm. It can be seen that A2C without Demo has 0 extra steps up to the $\sim 200^{th}$ evaluation step, which means that (while knowing it had an almost 0 accuracy from Figure 4) it doesn't solve any task before it. When it starts to solve tasks, its average extra steps is significantly higher and has an inconsistently-overshooting behavior compared to other SAC and A2C variations with demonstration.

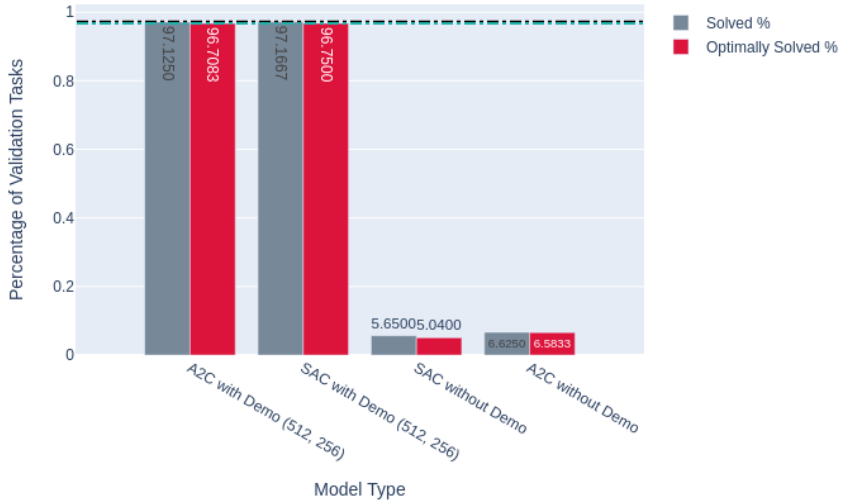

Figure 6: Percentage of Solved Tasks using SAC and A2C in both Learning with and without Demonstration Modes

.

As a final metric for the learning with and without demonstration comparison, Figure 6 compares between SAC and A2C in both setups. The percentage of solved tasks and percentage of optimally solved tasks is dramatically higher whenever the model uses learning by demonstration for the same number of steps.

Behavioral Cloning: Since learning with demonstration showed significantly better results than without, it seems that the agent performs better whenever it is guided by a ground truth. This laid the ground for me to reformulate the Karel task problem to a supervised learning problem, where the agent (Feed Forward Neural Network) learns to map states to actions.

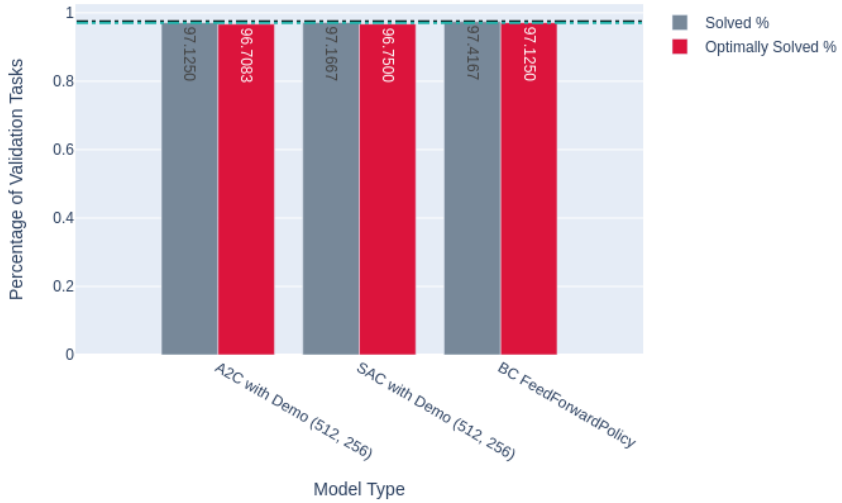

Figure 7: Percentage of Solved and Optimally Solved Tasks Across SAC, A2C and BC

In Figure 7, the behavioral cloning agent (BC) scores the highest percentage of solved and optimally solved tasks (97.42% and 97.13% respectively), which outperforms both A2C and SAC with demonstration by a small margin. This particular result caused me to experiment further with the behavior cloning agent to optimize its hyperparameters.

Optimization of BC-FFNN Agent: To further optimize BC-FFNN, the following architectures were experimented with in Figures s8. The **best BC-FFNN architecture** is the one with 2 dense layers of sizes 256 each, scoring **a solved % tasks of 97.7%** and **optimally solved % of 97.4%** on the validation hard dataset.

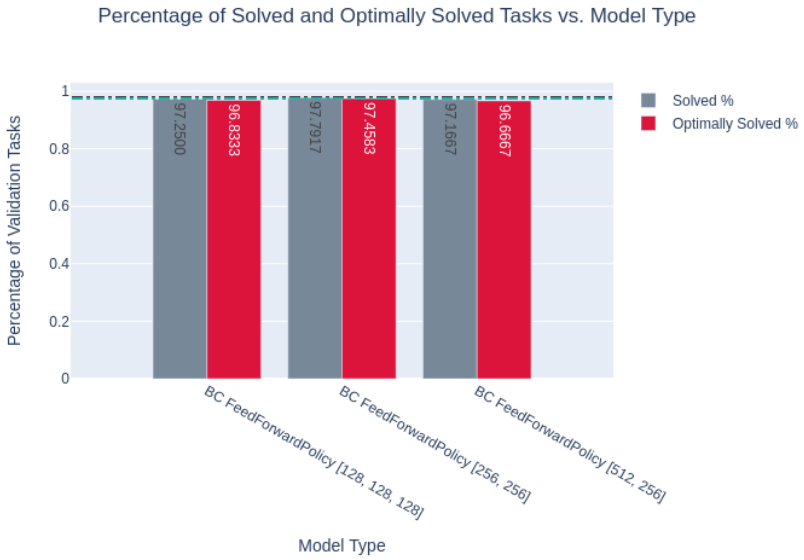

Figure 8: Behavioral-Cloning (BC) Accuracy Across Different Architectures

.

Best Performing Configuration

Best Performing Agent	% Solved	% Optimally Solved	Hyperparameters	Techniques
BC-FFNN	**97.71%**	**97.33%**	learning_rate=0.001 MAX_EPISODES=40,0000 EPOCHS=20 Policy: FeedForwardPolicy num_layers=2 layers_sizes=[256, 256] Optimizer: Adam	Environment probing: 1-level query to avoid crashes

Table 1: Best Performing Agent BC-FFNN Configuration

The best performing algorithm is Behavioral Cloning Feed Forward Neural Network, whose configuration, score and hyperparameters are shown in Table and illustrated in 9 below.

Test Data Results

- **Average Steps per Task** = 5.529583333333333
- **Tasks Solved** 98.25%

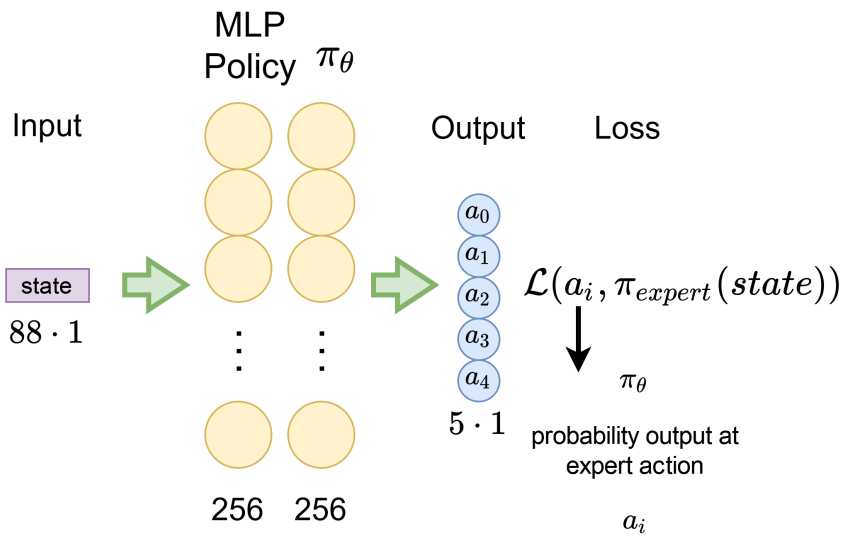

Figure 9: Behavioral-Cloning (BC) Best Configuration Architecture

.

References

[1] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[2] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay, 2017.

Appendix

Algorithm 1: Advantage Actor Critic (A2C) with MLP policy and Learning by Demo with Batched Updates

1. Input and problem setup:

- Actor's policy is parametrized using a neural network $\pi(a \mid s, \theta)$ (class MLPPi)
- Critic's state-value function is parametrized using a neural network as $\hat{v}(s, w)$ (class MLPv)
- Expert's policy function (non-parametrized as provided by dataset as $\pi_{exp}(a \mid s)$
- Convert all states to be in a vectorized multi-binary form and discertize actions

2. Hyperparameters:

- Step-sizes for the actor and critic updates, and discount factor $\alpha_{actor}, \alpha_{critic}, \gamma > 0$
- Entropy coefficient $0 < \lambda < 1$
- Number of episodes M; Maximum length of an episode H; Epochs EPOCHS

3. Initialization:

- Set max number of episodes to train on (MAX_EPISODES per epoch)
- Set U to be define update frequency (in terms of episodes).
- Set neural networks parameters ($\theta \in \mathbb{R}^{d_{actor}}$ and $w \in \mathbb{R}^{d_{critic}}$) to 0.

4.:

for *epoch = 0, 1, 2, ..., EPOCHS* **do**
  for *m=0, 1, 2, ..., MAX_EPISODES* **do**
    either by sampling or
    - **if** $train\_by\_demo$ **then**
      -Reset episode and choose $S_0 \in S_{init}^{train}$ -Generate episode rollout following the expert trajectories from $\pi_{exp}(. \mid .) = S'_0, A'_0, R'_1, ..., S'_T, A'_T, R'_{T+1}$
    else
      - Reset episode and choose S_0 sampled from a env distribution - Generate episode rollout following the Actor's policy $\pi(. \mid ., \theta) = S_0, A_0, R_1, ..., S_T, A_T, R_{T+1}$
    end
    for *t=T-1, T-2, ..., 1,0* **do**
      - Compute discounted return $G_{t:T-1}$ (Monte Carlo Approach (Eq 3.8) in Ch3): $G_t = \sum_{\tau=0}^{T-t-1} \gamma^\tau \cdot R_{t+1+\tau}$
      -Insert the experience tuple (expert's experience if $train\_by\_demo$) $(S_t, A_t, R_{t+1}, G_{t:T-1}, \hat{v}(S_t, w_m), \pi(A_t \mid S_t, \theta_m)$ into $replay\_buffer$.
    end
    if $(m+1) \mod U == 0$ **then**
      Update the actor's network from the batch of rollouts in the $replay\_buffer$ (assume for simplicity it has T steps), first *get current parameters index (namely s) w.r.t* m *and* U*:* $s = \frac{m}{U}$ and if $learn\_by\_demo$ set $A_t = A'_t, S_t = S'_t$

$$\hat{\mathcal{L}}(\theta) = -\frac{1}{T} \cdot \sum_{t=0}^{T-1} \left(log(\pi(A_t \mid S_t, \theta)) \cdot (G_{t:t+n} - \hat{v}(S_t, w_s)) - \lambda \sum_a \log(\pi(a \mid S_t, \theta)) \right)$$

$$\theta_{s+1} \leftarrow \theta_s - \alpha_{actor} \cdot \nabla_\theta \hat{\mathcal{L}}_{actor}(\theta) \mid_{\theta=\theta_s}$$

      Update the critic's network from the batch of rollouts in the $replay\_buffer$:

$$\hat{\mathcal{L}}(w) = -\frac{1}{T} \cdot \sum_{t=0}^{T-1} (G_t - \hat{v}(S_t, w_s))^2$$

$$w_{s+1} \leftarrow w_s - \alpha_{critic} \cdot \nabla_w \hat{\mathcal{L}}_{critic}(w) \mid_{w=w_s}$$

    end
  end
end
return Actor's policy $\pi(. \mid ., \theta_{U+EPOCHS*M})$