Deep Reinforcement Learning Algorithms Benchmark†

†Note: This paper is a progress report for Practical Machine Learning And Deep Learning at Innopolis University for Fall 2021

Note: There are two appendencies at the end summarizing the current progress (Section VI) and the future tasks (Section ??)

Hany Hamed*
Robotics Institute
Innopolis University
Russia
h.hamed@innopolis.university

Ahmad Hamdan*
Robotics Institute
Innopolis University
Russia
a.hamdan@innopolis.university

Utkarsh Kalra*
Data Science Institute
Innopolis University
Russia
u.kalra@innopolis.university

***Abstract*—Reinforcement in general is a term widely used in the English language as a term implying one or more observations influencing another action that needs to be taken. A very similar meaning is adapted in the Deep Reinforcement learning in Machine learning and computer science. As the advancement in Machine learning the urge to solve the world with Machine learning is something which keeps computer scientists awake in the nights, reinforcement learning is used in the cases where a proper collected dataset is rather difficult such as in video games. With the help of RL the results of the previous actions influence the next action. In this project we implement some standard and state of the art Reinforcement learning algorithms from scratch in order to understand deeply, and test them on different environment and compared them in order to have a better understanding of correlation between different algorithms.**

***Index Terms*—Deep Reinforcement Learning, PyTorch, On-Policy, Off-Policy,**

I. INTRODUCTION

Reinforcement Learning (RL) is a way of learning without the need for a collected dataset, instead, it is used to learn from experience and sampling the data through the interaction of the agent with the environment. The environment is either simulated, and then the learning policy is transferred to the real world which is known by Sim2Real [Höf+20], or the sampling is done directly in the real world [Hes+18]. Moreover, RL was used in robotics to solve several tasks, for example grasping [Gu+17], quadruped locomotion [Pen+20] and drone racing [Son+21], thus, the impressive results made learning-based approaches like RL to be popular in robotics.

Deep Reinforcement Learning (DRL) [Li17] is an evolution of the traditional methods of RL such that it is using Deep Learning (DL) models to use deep neural networks as general approximate for any function such as the policy function, value function, action-value function, ...etc. Thus, we can solve th problems that include continuous states and high dimensional action and state space, therefore, DRL has paved the way for solving many problems.

* Equal contributions

Multiple libraries had provided intensive implementations for most of the algorithms, the most famous two Ray/RLLib [Lia+17] and Stable-baselines [Raf+19], however, implementing the code and comparing with their results on the different environments will be interesting to understand the insights of the algorithm and the different deep neural networks architectures that are used for DRL algorithms.

In this paper, we are implementing three algorithms: deep Q-network (DQN) [Mni+15], Proximal Policy Optimization (PPO) [**PPO1**], and Soft Actor-Critic (SAC) [Haa+18]. These algorithms were selected in order to explore different classes of algorithms (Value-based algorithm: DQN, Policy gradient algorithm: PPO, Actor-Critic: A3C and SAC)

All the models' implementations are made using PyTorch and are available at: `https://github.com/hany606/PMLDL-Project`

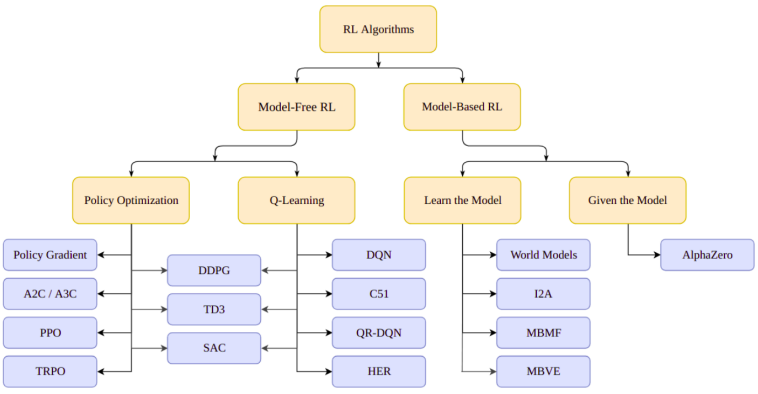

Fig. 1. Classification for RL algorithms. Adapted from [18]

II. RELATED WORK

A. DQN

DQN is introduced in 2 papers, Playing Atari with Deep Reinforcement Learning on NIPS in 2013 [Mni+13] and Human-level control through deep reinforcement learning on

Nature in 2015 [Mni+15]. DQN overcomes unstable learning by mainly 4 techniques :

- Experience Replay: DNN is easily over-fitting current episodes and this makes it hard to produce/learn a vast majority of experiences. Therefore, experience reply stores experiences including state transitions, rewards and actions, which are necessary data to perform Q learning, and makes mini-batches to update neural networks.
- Target Network: this technique fixes parameters of target function and replaces them with the latest network every thousands steps.
- Clipping Rewards: this technique clips scores, in which all positive rewards are set +1 and all negative rewards are set -1.
- Skipping Frames: DQN calculates Q values every 4 frames as humans in reality are not that fast and cannot take action 60 times per seconds. Such techniques will reduce computational complexity and will allow gathering better experiences.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Fig. 2. DQN pseudocode [htt18a]

B. PPO

Proximal Policy Optimization[**PPO1**] algorithm was introduced by Open AI to tackle a couple of problems in the general reinforcement learning algorithms. One of the main reasons why PPO is used or was introduced is **Unstable policy update** [Sur20]: In a number of policy methods, the policy updates are unstable and may take large step sizes which more often than not will lead to missing to global maxima and will lead to rather negative results. Or if the step size is too small then the learning becomes very small. Another problem that PPO solves is **Sample Inefficiency**: A lot of learning methods take into consideration the current experience and learn from them but do not take it into account once the gradient updates.

PPO is fundamentally based on the concept of **Advantage Function**, advantage function basically tells us if the current move or the current gradient update resulted in the advance towards the global maxima or not. PPO is basically based upon the idea that if the gradient of the current policy seems to be far away from the gradients of the previous policy or the baseline policy then only a portion of the current policy gradient is considered. The difference between the two gradients under consideration is quantified using a ratio of the two gradient, if this ratio exceeds a predetermined threshold value then the value of the current gradient is clipped down to be considered for the further iterations.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Fig. 3. PPO pseudocode [htt18b]

C. SAC

Soft Actor Critic, or SAC, is an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning framework. The algorithm tries to maximize the reward while maximizing the entropy as well. Such approach guarantees random actions from agent in order to succeed in the game. SAC combines off-policy updates with a stable stochastic actor-critic formulation. SAC has many advantages:

1) Capable of exploring more widely, detecting and cutting unpromising paths early
2) Capable of getting multiple near optimal behaviors
3) In problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions.
4) It improves learning speed over state-of-art methods that optimize the conventional RL objective function.

Algorithm 1 Soft Actor-Critic

Inputs: The learning rates, λ_π, λ_Q, and λ_V for functions π_θ, Q_w, and V_ψ respectively; the weighting factor τ for exponential moving average.
Initialize parameters θ, w, ψ, and $\bar{\psi}$.
for each iteration **do**
  (In practice, a combination of a single environment step and multiple gradient steps is found to work best.)
  for each environment setup **do**
    $a_t \sim \pi_\theta(a_t|s_t)$
    $s_{t+1} \sim \rho_\pi(s_{t+1}|s_t, a_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1}\}$
  for each gradient update step **do**
    $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$.
    $w \leftarrow w - \lambda_Q \nabla_w J_Q(w)$.
    $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$.
    $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$.

Fig. 4. SAC pseudocode [htt18b]

III. Environments

Reinforcement Learning problems are composed from two main things: the training algorithm and the environment that the algorithm is used to train the agent inside this environment. The environment describes how the agent interact within the environment. The environment consists of 5 main parts to be defined:

- **Reward function (*reward*):** describes how the agent is rewarded (i.e. If the agent did something good/useful, it will be rewarded with positive reward and viceversa)
- **Completion criteria (*done*):** describes how the episode should be determined to be completed (finished) or not
- **Step (*step*):** describes how the transitions from the current state to the future state based on the given action
- **Observation/States space (*observation*):** describes the limits and the number of observations of the environment. Observations are what the agent observed from the states.
- **Actions space (*action*):** describes the limits and the number of possible actions for the agents.

Note: action space and obsesrvation space can be discrete or continuous spaces.

IV. Implementation, Experiments And Results

A repository with results and scripts to reproduce the results is available at: `https://github.com/hany606/PMLDL-Project`

All the RL algorithms tested were implemented from scratch with the help of theory, scientific articles and some already available implementations online. The algorithms were tested for multiple runs for each environment they are applicable for. For each algorithm only the models with the highest reward values during the training.

To keep track of the weights, rewards and training parameters and plot the graphs for them, an online tool **Wandb.ai** was used. The graphs thus produced for each environment are presented in the following sections.

A. Cart-Pole Environment

A pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts in the upright position, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. We did several experiments on this environment and we decided to increase the number of layers to increase the complexity of the model in order to give better actions, yet we did not get optimal behavior yet. For D1.5 We continued tuning the hyperparameters in DQN algorithm but the performance was not optimal. The plot in Figure 5 shows the reward per epoch and the highest reward available for our implementation was 80 which is unsatisfactory for this environment.

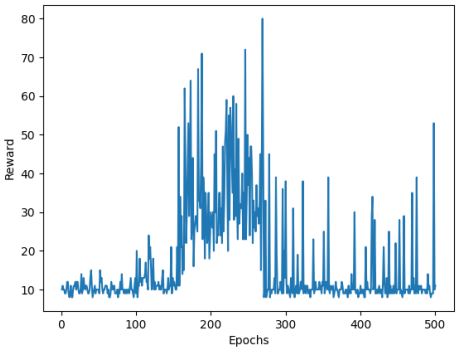

Fig. 5. DQN epoch rewards for Cartpole environment

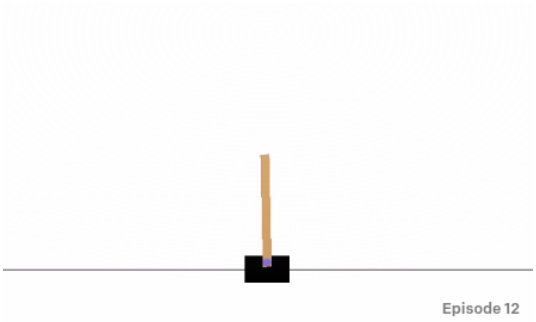

Fig. 6. Cart-pole environment

Once we finished the implementation of PPO we measured the performance of the two models and plotted against each other. During our implementation we made the episode length 100 therefore the maximum reward will be 100. It is important to note that the standard episode length is 500 but we cut it for faster execution time. It is known that PPO gives significantly better results than DQN and faster as the environment gets more complicated. It is important here to keep in mind that DQN implementation is not optimal but this is the one that we have while we are writing this report.

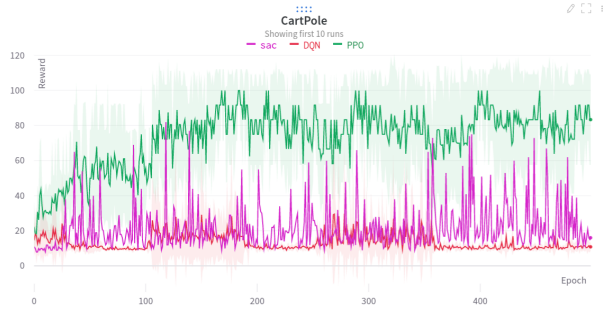

Fig. 7. DQN vs PPO vs SAC on Cartpole environment

We can see that the SAC has a better performance than DQN but is not as well as PPO. This is mainly because of of the fact that the SAC implemented has some shortcomings

in terms of the loss function as this environment has discrete input actions and the SAC is most optimal for environments with continuous action input. Also the training for SAC is very tolling on the PC and a lack of efficient GPU effected the training and testing quite a bit. There are still some things which can be improved in SAC to make better results for particular environments.

B. Acrobot Environment

This is one of more popular environment for reinforcement learning and it includes ”two joints and two links where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height.”

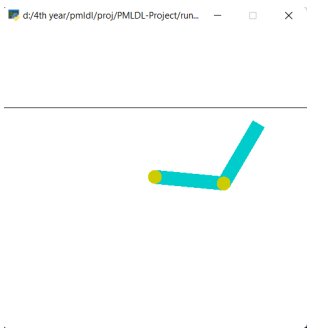

Fig. 8. Acrobot environment

The acrobot-v1 environment was trained firstly with DQN algorithm as the base algorithm for the reinforcement learning. For this we had a simple sequential model consisting of 4 fully connected layers with suitable inputs and output features. After much experimentation the structure of the layers were somewhat doubling the number of features as output and in the last layer just outputting the required number of features worked the best. As compared from the DQN implementation of cart-pole, here we changed some hyper-parameters like decreasing the learning rate and changing the number of steps. The chart for epoch rewards is shown below:

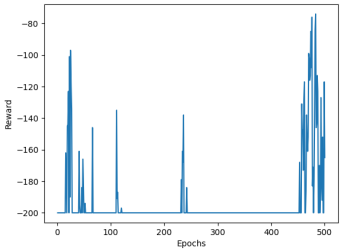

Fig. 9. DQN epoch rewards for Acrobot environment

The acrobot after taking in the best model ended on this stage in the rendering.

The PPO algorithm worked inefficiently on this environment and rather did not work because of the following reason. PPO will not be applicable on this environment as it is an on-policy algorithm and it performs a policy gradient update after each episode and does not save anything about the episode. Reaching above the line is almost impossible with random actions because in this environment you can only get a positive reward once you cross the defined line. Thus it is extremely unlikely that a single policy gradient update will be enough to reach the goal.
The performance for SAC in this environment was also somewhat unsatisfactory because of similar reasons mentioned in the above section.

C. MountainCar-v0 Environment

A car is on a one-dimensional track, positioned between two ”mountains”. The goal is to drive up the mountain on the right; however, the car’s engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

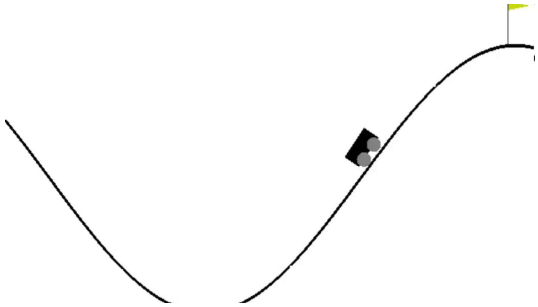

Fig. 10. Mountain car environment v0

PPO will not be applicable on this environment as it is an on-policy algorithm and it performs a policy gradient update after each episode and does not save anything about the episode. Reaching the goal in Mountain car with random actions is almost impossible because in the environment rules you only get positive reward when you get to the top. Therefore it is extremely unlikely that a single policy gradient update will be enough to start reaching the goal consistently.

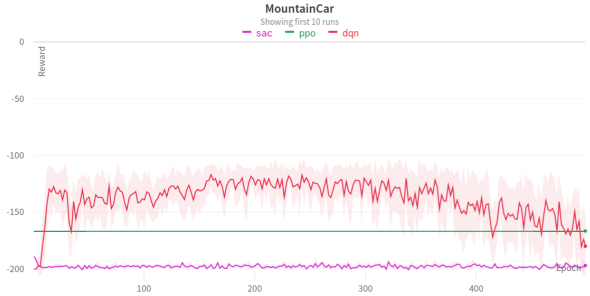

Fig. 11. DQN vs PPO vs SAC on mountaincar environment

In this case the DQN performs the best, PPO is as mentioned above not applicable and SAC can be improved.

V. CONCLUSIONS

In conclusion, In this project we have understood the deep meaning of Deep Reinforcement learning by implementing the algorithms from scratch first hand. Had the exposure to different environments on which Reinforcement learning is applicable. We have implemented three different algorithms DQN, PPO and SAC which provide a very good foundation in RL. Our implementation has been tested on three different environments where we understood the special characteristics of each one of them and started tackling these problem with a new perspective different than classical control methods. Our work will continue to fix the implementation of these algorithms and get good scores on all environments.

REFERENCES

[Mni+13] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[Gu+17] Shixiang Gu et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[Li17] Yuxi Li. "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274* (2017).

[Lia+17] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning. arXiv e-prints, page". In: *arXiv preprint arXiv:1712.09381* (2017).

[18] 2018. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[Haa+18] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[Hes+18] Todd Hester et al. "Deep q-learning from demonstrations". In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[htt18a] https://lilianweng.github.io/about. *A (Long) Peek into Reinforcement Learning*. Feb. 2018. URL: https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html.

[htt18b] https://lilianweng.github.io/about. *Policy Gradient Algorithms*. Apr. 2018. URL: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html.

[Raf+19] Antonin Raffin et al. *Stable Baselines3*. https://github.com/DLR-RM/stable-baselines3. 2019.

[Höf+20] Sebastian Höfer et al. "Perspectives on Sim2Real Transfer for Robotics: A Summary of the R: SS 2020 Workshop". In: *arXiv preprint arXiv:2012.03806* (2020).

[Pen+20] Xue Bin Peng et al. "Learning agile robotic locomotion skills by imitating animals". In: *arXiv preprint arXiv:2004.00784* (2020).

[Sur20] Abhishe Suran. "Proximal Policy Optimization (PPO) With TensorFlow 2.x". In: *Towards Data Science* (2020).

[Son+21] Yunlong Song et al. "Autonomous Drone Racing with Deep Reinforcement Learning". In: *arXiv preprint arXiv:2103.08624* (2021).

VI. APPENDIX (1) - FINISHED TASKS

- Understood the base concepts in Reinforcement Learning
- Understood and implemented three of the state-of-art algorithms for RL: Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and Soft Actor Critic (SAC).
- Experiment DQN, PPO and SAC for three environments: CartPole, MountainCar, and AcroBot
- Report the results and benchmark them using Wandb