REINFORCEMENT LEARNING ALGORITHMS FOR ATARI GAMES

Gudepu Venkateswarlu, Jayanth S, Praveen Kumar N, Rishabh Roy
IIT Dharwad, Karnataka
{212011003, 201081003, 201082001, 201082002}@iitdh.ac.in

ABSTRACT

As a part of the mini-project, we tried to implement different policy gradient reinforcement learning (RL) algorithms for the breakout, ping-pong, and space-invaders atari games. Initially, we used the stable-baselines3 codes to train the RL agents from scratch in the atari environment. However, we were unable to get the expected results or make the algorithm work for different environments. Hence we used the algorithms implemented in the Ray rllib library for Advantage actor-critic (A2C), Asynchronous actor-critic (A3C), and Proximal Policy Optimization (PPO) agents. We trained the agents from scratch and tried to match the results with the benchmark results given on their website for different atari games.

1 Introduction

In any Artificial Intelligence task like an AI agent learning how to play Atari games, demands a mathematical framework to model the complex problem. In reinforcement learning (RL), we take help from Markov Decision Process (MDP) framework, where an agent observes the environment and takes actions. Rewards are given out but they may be infrequent and delayed. The goal of the agent can be to maximize the expected long-term cumulative reward. An MDP is defined by the following set of parameters:

- Set of states, $\mathcal{S}$
- Set of actions, $\mathcal{A}$
- Transition probability from state, s to state, s' when action, a is taken in state, s and denoted as $\mathbb{P}(s'|s,a)$
- Reward function associated with state, s, action, a and the next state, s' and is represented as $r(s,a,s')$
- Discount factor, γ
- Start state, s_0

In RL, our focus is to find an optimal policy. A policy tells us how to act from a particular state and is denoted as $\pi_\theta(a_t|s_t)$ i.e., the probability of taking an action a_t at time step t given the agent is in state s_t. The policy is parameterized by θ like the weights in deep learning methods. We want to find a policy that maximises the expected reward over all the trajectories, i.e.,

$$\max_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t r(s_t,a_t)\right] \tag{1}$$

where θ is the parameter, τ is the trajectory sampled from the trajectory distribution $p_\theta(\tau)$ with depends on the transition probability, and the policy π. Also, here $r(s_t,a_t)$ is the reward associated with the state-action pair at time instant t. The optimal policy can be a deterministic or stochastic policy. The stochastic policy outputs a probability distribution over the actions instead specifying a single action. In RL, we want to find a sequence of actions that maximize expected rewards as mentioned in (1). There are three major ways to solve the problem:

- Value-based learning
- Model-based learning
- Policy-based learning

In the present work we consider only the policy based methods where the policy is directly parameterized by the parameter θ and the aim to find the optimal parameters such that objective in (1) is maximized[9]. We tried to train the following agents for Breakout, Ping-pong, Space-invaders atari games:

1. Advantage Actor-critic Algorithm (A2C)
2. Asynchronous Advantage Actor-Critic (A3C)
3. Trust Region Proximal Policy Optimization (TRPO)
4. Proximal Policy Optimization (PPO)

2 Actor-Critic Algorithms

According to the vanilla policy gradient algorithm we have,

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t) G_t \right]$$

This expectation can be decomposed as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0,a_0,\ldots,s_t,a_t} \left[\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t)] \mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T} [G_t] \right]$$

Here the second expectation term is the Q-value i.e., $\mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[G_t] = Q(s_t, a_t)$. Therefore we can write

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t) Q(s_t, a_t) \right]$$

As we know the policy is parameterized by θ, similarly we can parameterize the Q-value by w i.e, $Q_w(s, a)$ instead of estimating it from trajectories. This leads us to the *Actor-Critic*[6] methods, where both policy and value function is parameterized (for example by a neural network).

1. The “Critic” updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(s, a)$ or state-value $V_w(s)$.
2. The "Actor" updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

A simple Q-value Actor-Critic algorithm (Q-Actor-Critic) is given below:

- Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
- For $t = 1, \ldots, T$:
  1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  2. Then sample the next action $a' \sim \pi_\theta(a'|s')$
  3. Update the policy parameters: $\theta \longleftarrow \theta + \alpha_\theta \nabla_\theta \ln \pi_\theta(a|s) Q_w(s, a)$
  4. Compute the correction (TD error) for action-value at time t:
     $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
     and use it to update the parameters of action-value function:
     $w \longleftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
  5. Update $a \longleftarrow a'$, $s \longleftarrow s'$.

3 Advantage Actor-Critic

The main disadvantages of vanilla policy gradient methods are high variance, instability and slow convergence. One way to reduce variance and increase stability is by introducing a baseline which is subtracted from the Q-value. Commonly used baseline is the value function $V(s)$ i.e., we subtract the state value $V(s)$ from state-action value $Q(s, a)$. Intuitively, this means how much better it is to take a specific action compared to the average, general action at the given state. This is called ***Advantage value*** defined as

$$A(s_t, a_t) = Q_w(s, a) - V_v(s)$$

From the Bellman equation we have the relationship between Q-value and state value given as,

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$$

So, we can rewrite the advantage as:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s)$$

Hence there is no need of two separate neural networks for learning Q-value and state value to compute the advantage value. we only have to use one neural network for the state value function parameterized by v. Therefore we can now write the $\nabla_\theta J(\theta)$ for Advantage Actor-Critic as,

$$\begin{aligned}\nabla_\theta J(\theta) &\sim \sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s)) \\ &= \sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t) A(s_t, a_t)\end{aligned}$$

The Advantage Actor Critic has two main variants: the Asynchronous Advantage Actor -Critic (A3C) and the Advantage Actor Critic (A2C).

3.1 Asynchronous Advantage Actor-Critic (A3C)

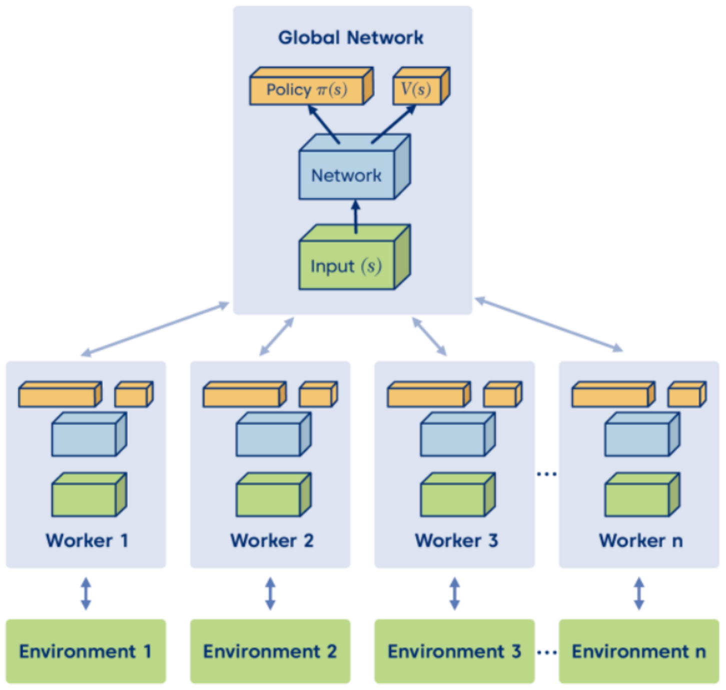


Figure 1: A3C architecture.

- A3C was introduced in Deepmind's paper "Asynchronous Methods for Deep Reinforcement Learning"
- A3C implements parallel training where multiple workers in parallel environments independently update a global value function—hence the name "Asynchronous" 1.
- One key benefit of having asynchronous actors is effective and efficient exploration of the state space.

3.2 Advantage Actor-Critic (A2C)

- A2C is a synchronous version of A3C.
- In A3C each agent talks to the global parameters independently, so it is possible sometimes the thread-specific agents would be playing with policies of different versions and therefore the aggregated update would not be optimal.

- To resolve the inconsistency, a coordinator in A2C waits for all the parallel actors to finish their work before updating the global parameters and then in the next iteration parallel actors starts from the same policy.
- A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C.

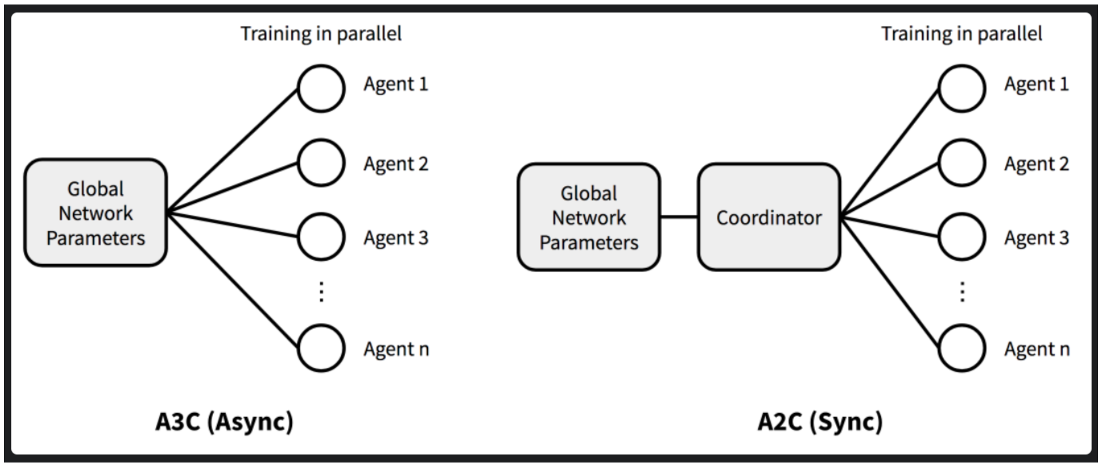


Figure 2: The architecture of A3C versus A2C.

4 Trust Region Policy Optimization

The basic principle of policy gradient algorithms is to maximize the long-term total reward using gradient ascent principle and in each step we update our policy that results in largest increase in the long-term reward. This way of updating the policy may lead to settling to locally optimal policy[4]. Instead, in trust region policy optimization (TRPO) [7] we restrict the deviation of the updated (new) policy from the old policy by having a constraint that measures the closeness of the new policy to the old policy.

In TRPO, KL-divergence is used as the distance measure between the probability distributions of old and new policies. Theoretically TRPO algorithm tries to solve,

$$\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_\theta(\tau)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \tag{2a}$$

$$\text{subject to} \quad D_{KL}\left(\pi_{\theta'}(a_t|s_t) || \pi_\theta(a_t|s_t)\right) \leq \epsilon \tag{2b}$$

where, $A^{\pi_\theta}(s_t, a_t) = r(s_t, a_t) + \gamma V^{\pi_\theta}(s_{t+1} - V^{\pi_\theta}(s_t)$, θ corresponds to the parameters of old-policy and $D_K L(p||q)$ is the Kullback–Leibler(KL)-divergence that measures how one probability distribution, q differs from the other p. We can show that solving the above optimization is almost same as solving (1)[5, Lecture 9]. The policy update at each step will always be a better policy (monotonic improvement) as shown in [7]. However, for the practical implementation, the monte-carlo estimation of the above objective and the constraint is complex. Hence the objective and the constraint are replaced by the following linear approximations. Using the 1st order approximation for the objective we get,

$$\nabla_\theta \bar{A}(\theta)^T \cdot (\theta' - \theta) \tag{3}$$

where,

$$\nabla_\theta \bar{A}(\theta) = \sum_t \mathbb{E}_{s_t \sim p_\theta(\tau)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t \nabla_{\theta'} log(\pi_{\theta'}(a_t|s_t)) A^{\pi_\theta}(s_t, a_t) \right] \right]$$

Now, Using the 2^{nd} order approximation for the constraint,

$$D_{KL}\left(\pi_{\theta'}(a_t|s_t) || \pi_\theta(a_t|s_t)\right) \approx \frac{1}{2}(\theta' - \theta)^T \cdot \mathbf{F} \cdot (\theta' - \theta)$$

where $\mathbf{F}$ is the Fisher-information matrix and is given by,

$$\mathbf{F} = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta log(\pi_\theta(a|s) \nabla_\theta log(\pi_\theta(a|s)^T\right]$$

It can be shown that the update rule for TRPO is,

$$\theta' = \theta + \alpha \cdot \mathbf{F}^{-1}\nabla_\theta J(\theta)$$

In TRPO the learning rate is not specified explicitly, instead it is calculated as follows,

$$\alpha = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\theta)^T \mathbf{F} \nabla_\theta J(\theta)}}$$

where, ϵ is the threshold on the KL-divergence between the old and the new policy and is a hyper-parameter for the TRPO algorithm. During implementation, the conjugate gradient descent[3] with line search is used that finds $\mathbf{F}^{-1}\nabla_\theta J(\theta)$ directly instead of calculating the gradient and then multiplying it with the inverse of Fisher-information matrix, $\mathbf{F}$. The algorithm is given as follows[2]:

Algorithm 1 Trust Region Policy Optimization

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: Hyperparameters: KL-divergence limit $\delta$, backtracking coefficient $\alpha$, maximum number
   of backtracking steps $K$
3: for $k = 0, 1, 2, ...$ do
4:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
5:     Compute rewards-to-go $\hat{R}_t$.
6:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
7:     Estimate policy gradient as
           $\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$
8:     Use the conjugate gradient algorithm to compute
           $\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$
       where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
9:     Update the policy by backtracking line search with
           $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$
       where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and
       satisfies the sample KL-divergence constraint.
10:    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.
11: end for
```

5 Proximal Policy Optimization (PPO)

A recent innovation in the field of Reinforcement Learning, Proximal Policy Optimisation (PPO) [8], provides an enhancement on TRPO. This algorithm was proposed in 2017, and when it was implemented by OpenAI, it demonstrated outstanding performance. The Proximal Policy Optimization algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor). The issues with TRPO algorithm are as follows:

- TRPO uses conjugate gradient descent to handle the constraint
- Hessian Matrix is expensive both in computation and space
- Hard to use with architectures with multiple outputs, e.g. policy and value function (need to weight different terms in distance metric)
- Empirically performs poorly on tasks requiring deep CNNs and RNNs, e.g. Atari benchmark

In order to make the new policy not deviate significantly from the old policy, PPO uses clipping method to prevent an excessively large update.

5.1 Intuition

Although TRPO is a powerful algorithm, it suffers from the constraint issue, which adds additional complexity to our optimization problem. Conjugate gradient approach and quadratic approximations bewildered us because of this. Wouldn't it be great if the constraint could be included in our optimization objective?
As a penalty, we embed a constraint into the objective function rather than adding it as an independent constraint (we subtract the KL divergence times a constant C from the function).

5.2 PPO Algorithm

5.2.1 Adaptive KL Penalty

- Policy update solves unconstrained optimization problem, $\theta_{k+1} = \mathrm{argmax}_\theta \, \mathcal{L}_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}\left(\theta \| \theta k\right)$.
- Penalty coefficient β_k changes between iterations to approximately enforce KL-divergence constraint.

Algorithm:

```
- Input: initial policy parameters $\theta_0$, initial KL penalty $\beta_0$, target KL-divergence $\delta$
- for k = 0, 1, 2, ... do
    – Collect set of partial trajectories $D_k$ on policy $\pi_k = \pi(\theta_k)$
    – Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
    – Compute policy update
          $$\theta_{k+1} = \underset{\theta}{\mathrm{argmax}}\, \mathcal{L}_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}\left(\theta \| \theta_k\right)$$
      by taking K steps of minibatch SGD (via Adam)
    – if $\bar{D}_{KL}\left(\theta \| \theta_k\right) \geq \delta/1.5$ then $\beta_{k+1} = 2\beta_k$
    – else if $\bar{D}_{KL}\left(\theta \| \theta_k\right) \leq \delta/1.5$ then
          $$\beta_{k+1} = \beta_k/2$$
    – end if
- end for
```

For the PPO algorithm, the initial KL penalty is not that important since it adapts quickly. Hence, even though some iterations may violate KL constraint, but most do not violate the KL constraint.

5.2.2 Clipped Objective

- New objective function:

$$r_t(\theta) = \frac{\pi_\theta \left(a_t \mid s_t\right)}{\pi_{\theta_k} \left(a_t \mid s_t\right)} \mathcal{L}_{\theta_k}^{CLIP}(\theta),$$

  where,

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathbb{E}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min \left(r_t(\theta) \hat{A}_t^{\pi_k}, \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t^{\pi_k} \right) \right] \right],$$

  and ϵ is a hyperparameter.(ex: ϵ = 0.2).

- Policy update is $\theta_{k+1} = \operatorname{argmax}_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$.

Algorithm:

- Input: Initial policy parameters θ_0 and clipping threshold.
- **for** k = 0, 1, 2, ... **do**
  - Collect set of partial trajectories D_k on policy π_k = $\pi(\theta_k)$
  - Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  - Compute policy update

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \, \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

    by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathrm{E}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min \left(r_t(\theta) \hat{A}_t^{\pi_k}, \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t^{\pi_k} \right) \right] \right]$$

- **end for**

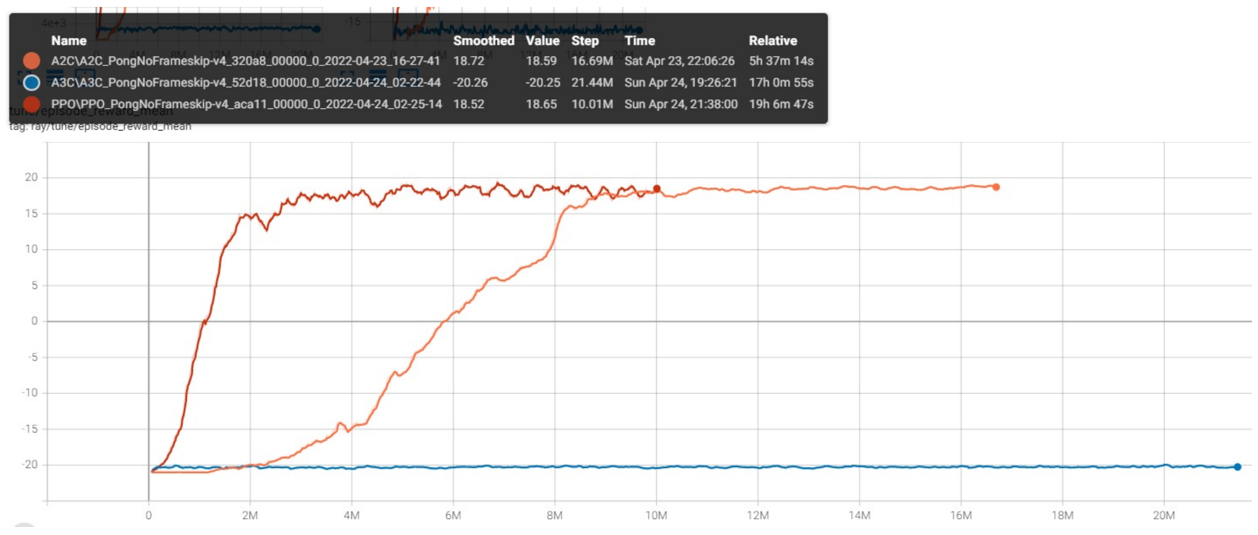
The advantage of clipping is, it prevents policy from having incentive to go far away from $\theta k + 1$. Also clipping seems to work at least as well as PPO with KL penalty, but is simpler to implement.

5.3 Failure mode of PPO

- On continuous action spaces, standard PPO is unstable when rewards vanish outside bounded support.
- On discrete action spaces with sparse high rewards, standard PPO often gets stuck at suboptimal actions.
- The policy is sensitive to initialization when there are locally optimal actions close to initialization.

6 Results

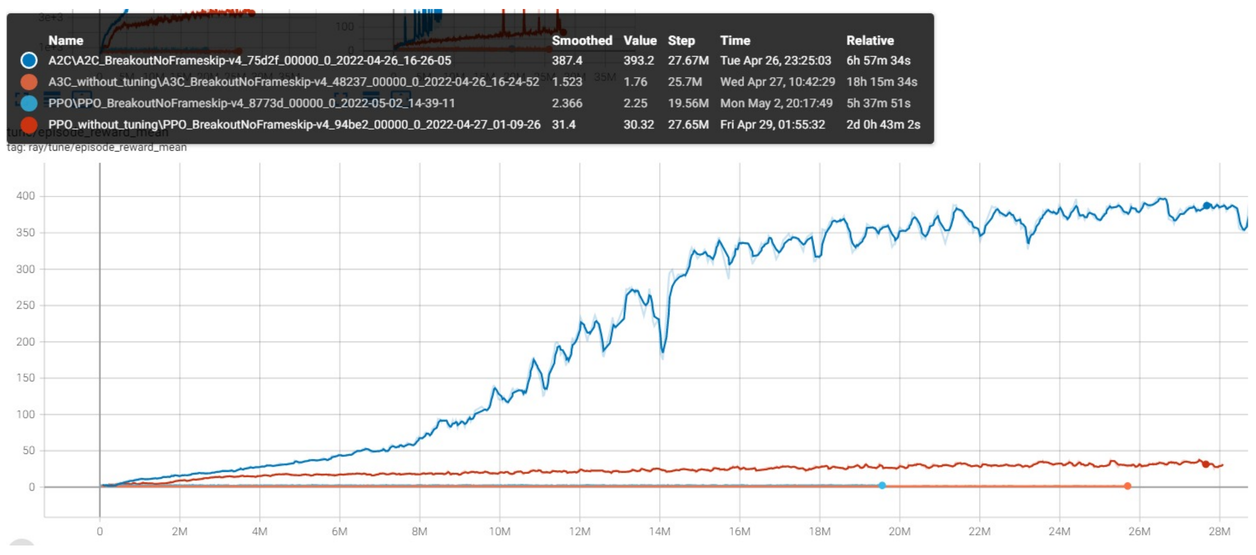
For implementing the above mentioned policy gradient algorithms, we tried using the stablebaselines3 codes. However, we were unable to get the results as expected for A3C, TRPO in pong, breakout, space-invaders environments. A2C agent implemented using stablebaselines3 code worked well for breakout but the same code was not able to perform well in pong and space-invaders environments. Similarly, PPO worked well for pong but not in breakout and space-invaders environments. Hence, we used *Ray rllib* codes for implementing A2C, A3C and PPO agents.



	Name	Smoothed	Value	Step	Time	Relative
●	A2C\A2C_PongNoFrameskip-v4_320a8_00000_0_2022-04-23_16-27-41	18.72	18.59	16.69M	Sat Apr 23, 22:06:26	5h 37m 14s
○	A3C\A3C_PongNoFrameskip-v4_52d18_00000_0_2022-04-24_02-22-44	-20.26	-20.25	21.44M	Sun Apr 24, 19:26:21	17h 0m 55s
●	PPO\PPO_PongNoFrameskip-v4_aca11_00000_0_2022-04-24_02-25-14	18.52	18.65	10.01M	Sun Apr 24, 21:38:00	19h 6m 47s

Figure 3: Running average of reward obtained for PongNoFrameskip-v4 environment using A2C, A3C, PPO algorithms.

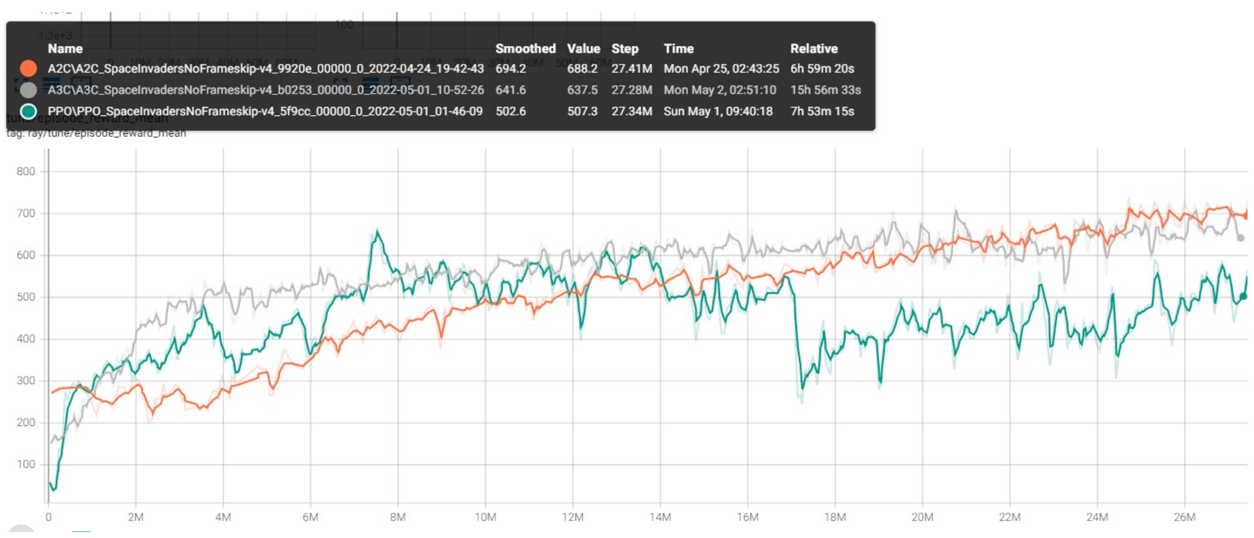
In Fig. 3, we illustrate the variation of running average of average reward for A2C, A3C and PPO agents with respect to the time steps in PongNoFrameskip-v4 environment. For pong environment, PPO was able to learn in lesser times steps when compared with A2C. However, the A3C agent was not able to learn how to win the game against the opponent even though it had learnt to move the paddle up and down.



	Name	Smoothed	Value	Step	Time	Relative
○	A2C\A2C_BreakoutNoFrameskip-v4_75d2f_00000_0_2022-04-26_16-26-05	387.4	393.2	27.67M	Tue Apr 26, 23:25:03	6h 57m 34s
●	A3C_without_tuning\A3C_BreakoutNoFrameskip-v4_48237_00000_0_2022-04-26_16-24-52	1.523	1.76	25.7M	Wed Apr 27, 10:42:29	18h 15m 34s
●	PPO\PPO_BreakoutNoFrameskip-v4_8773d_00000_0_2022-05-02_14-39-11	2.366	2.25	19.56M	Mon May 2, 20:17:49	5h 37m 51s
●	PPO_without_tuning\PPO_BreakoutNoFrameskip-v4_94be2_00000_0_2022-04-27_01-09-26	31.4	30.32	27.65M	Fri Apr 29, 01:55:32	2d 0h 43m 2s

Figure 4: Running average of reward obtained for BreakoutNoFrameskip-v4 environment using A2C, A3C, PPO algorithms.

In Fig. 4, we illustrate the variation of running average of average reward for A2C, A3C and PPO agents with respect to the time steps in BreakoutNoFrameskip$-v4$ atari environment. For breakout environment, only A2C agent performed well. The PPO agent was able to reach average reward of 31.4 and remained same even after 25 million $(25M)$ time-steps, this reward is very less for the breakout environment. Also, the A3C agent was not able to learn resulting in very low reward for the entire training period.



	Name	Smoothed	Value	Step	Time	Relative
🟠	A2C\A2C_SpaceInvadersNoFrameskip-v4_9920e_00000_0_2022-04-24_19-42-43	694.2	688.2	27.41M	Mon Apr 25, 02:43:25	6h 59m 20s
⚪	A3C\A3C_SpaceInvadersNoFrameskip-v4_b0253_00000_0_2022-05-01_10-52-26	641.6	637.5	27.28M	Mon May 2, 02:51:10	15h 56m 33s
🟢	PPO\PPO_SpaceInvadersNoFrameskip-v4_5f9cc_00000_0_2022-05-01_01-46-09	502.6	507.3	27.34M	Sun May 1, 09:40:18	7h 53m 15s

Figure 5: Running average of reward obtained for SpaceInvadersNoFrameskip-v4 environment using A2C, A3C, PPO algorithms.

In Fig. 5, we illustrate the variation of running average of average reward for A2C, A3C and PPO agents with respect to the time steps in SpaceInvadersNoFrameskip-v4 environment. For space-invaders environment, all the 3 agents were able to lean how to play and the running average rewards were more than 500 aver $25M$ time-steps for all the 3 agents. The A2C and A3C agents performed well when compared with the PPO agent for space-invaders environment.

Since, Ray rllib [1] does not have implementation of TRPO agent and moreover the stablebaselines3 TRPO agent did not perform as expected in any of the atari environment, we have not included the results of it. Also, we used the agent specific configurations as given in the Ray rllib documentation for each agent while training the agents for all 3 atari environments.

Table 1: Performance of A2C, A3C and PPO agents for pong, breakout, space-invaders environment

Agents	**Average Rewards**		
	Pong	Breakout	Space Invaders
A2C	$18.72@16M$	$387.4@27M$	$694.2@27M$
A3C	$-20.26@21M$	$1.5@25M$	$641@27M$
PPO	$18.52@10M$	$31@27M$	$507@27M$

7 Future Work

- Since A3C did not work well for Pong and Breakout environment we would like to fine tune some of the hyperparameters to make it work.
- Also we wish to implement the TRPO algorithm and make it work for the atari environment.

References

[1] Ray rllib available algorithms. URL: `https://docs.ray.io/en/latest/rllib/rllib-algorithms.html`.

[2] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[3] Jonathan Hui. Conjugate Gradient Explained. URL: `https://jonathan-hui.medium.com/rl-conjugate-gradient-5a644459137a`.

[4] Jonathan Hui. RL — Trust Region Policy Optimization (TRPO) Explained. URL: `https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9`.

[5] Sergey Levine. CS285- Deep Reinforcement Learning Course. URL: `https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-9.pdf`.

[6] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016. URL: `https://arxiv.org/abs/1602.01783`, `doi:10.48550/ARXIV.1602.01783`.

[7] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015. URL: https://arxiv.org/abs/1502.05477, doi:10.48550/ARXIV.1502.05477.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL: https://arxiv.org/abs/1707.06347, doi:10.48550/ARXIV.1707.06347.

[9] Lilian Weng. Policy Gradient Algorithms. URL: https://lilianweng.github.io/posts/2018-04-08-policy-gradient/.