Playing Fishing Derby With Deep Reinforcement Learning

Liam McDevitt [1]

1. Introduction

In recent years deep reinforcement learning (RL) algorithms have been a hot topic in the world of Artificial Intelligence (AI) research (Silver et al., 2016). One of the best ways to test new deep RL algorithms is by running them on the Atari 2600 benchmark suite (Ye et al., 2021). This benchmark suite contains a vast array of different video games where an agent can be trained to play and often beat the preprogrammed computer character. This suite makes it relatively easy to visualize and understand how the algorithm is performing due to an initial understanding of the game being played. Benchmarks are essential in determining the relevance of new algorithms since there are already many results from state-of-the-art algorithms to compare against.

In this paper, five deep reinforcement learning algorithm variants are tested on the Atari 2600 game Fishing Derby: Advantage Actor-Critic (A2C) (n-step), Advantage Actor-Critic (A2C) (Generalized Advantage Estimate (GAE)), Proximal Policy Optimization (PPO) as an extension on A2C, and Asynchronous Advantage Actor-Critic (A3C) for both advantage estimate versions. The testing was made possible by SLM Lab's extensive deep reinforcement learning software framework and integration with the Atari 2600 benchmark suite (Keng & Graesser, 2017). The rest of this section will introduce the task, state-space, action space, and reward signal.

Fishing Derby is an Activision-published sports video game for the Atari 2600 console and was developed by David Crane in 1980. Two human players may play against each other, or you can play by yourself against the computer, as can be seen in Figure 1. The game involves two fishers sitting on opposite docks, each trying to catch 99 lbs of fish. Whoever catches 99 lbs of fish first wins the game. There are six rows of fish, with the fish near the top weighing less and the bottom weighing more. The top two rows have 2 lb fish, the middle two rows have 4 lb fish, and the bottom two rows have 6 lb fish. Only one player can real in a fish at a time. Once a fish is hooked, it will slowly move its way up to the surface. The reel speed can be increased by pressing the button on the Atari 2600 controller. A shark will roam near the top of the water at various random rates and will eat your fish if it gets near. Hence, the shark should be avoided to maximize the score. Fishing Derby is playable using deep RL algorithms, where the agent is the player versus the in-game computer.

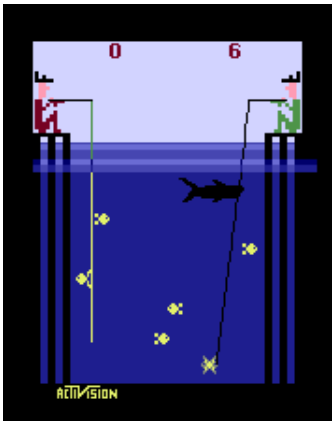

Figure 1. Fishing Derby: in-game snapshot

To play Fishing Derby with an RL agent, it must be represented in a way it can understand through proper environment characteristics. The state-space for the agent is an RGB image of the screen represented as an array of configurations (210, 160, 3). This is how the agent knows where it is and where it's possible to go. The action space comprises all actions available to the Atari 2600 controller, which involve a joystick and a button. The controller is how the finishing line will move throughout the space with the addition of the button for a faster reel. The reinforcing signal for the agent is how well it is doing compared to the other player. The score is the difference between the amount of fish caught in pounds, while the reward is the difference in the amount of fish it caught previously.

The rest of this paper is as follows: Section 2 provides the necessary background on all deep reinforcement learning algorithms used in this paper. The experiments performed are presented in Section 3. The results for those experiments are given in Section 4. Section 5 discusses all results obtained, and Section 6 concludes the paper.

[1]Department of Computer Science, Brock University, St. Catharines, Canada. Correspondence to: Liam McDevitt <lm15ue@brocku.ca>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

2. Methods

All necessary deep reinforcement learning methods used throughout this paper are presented in this section at a fundamental level. The techniques covered include Advantage Actor-Critic (A2C) with two different methods for estimating the advantage function, n-step returns and Generalized Advantage Estimate (GAE), Asynchronous Advantage Actor-Critic (A3C), and Proximal Policy Optimization (PPO).

2.1. Advantage Actor-Critic (A2C)

Today's Actor-Critic algorithms were initially presented in 1983 by Sutton et al. through the paper: *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems* (Barto et al., 1983). Advantage Actor-Critic (A2C) in an on-policy algorithm that works in both discrete and continuous environments for both continuing and episodic tasks by learning a state-value function $\mathbf{V}^\pi$ using Temporal Difference (TD) learning and a parameterized policy π_θ (Keng & Graesser, 2019).

Actor-Critic algorithms have two core components, an actor and a critic. They combine the ideas of a policy gradient and a learned value function. The actor is responsible for learning a parameterized policy π_θ while the critic's purpose relies on learning the value function $\mathbf{V}^\pi$. The critics learned value function $\mathbf{V}^\pi$ provides the reinforcing signal for the policy. In a sense, the critic tells the actor how well a particular policy performs. The reinforcing signal provided by the critic is most often the advantage function (Keng & Graesser, 2019).

The actor learns a parameterized policy π_θ from the following policy gradient equation.

Actor-Critic (Keng & Graesser, 2019):

$$\nabla_\theta \mathbf{J}(\pi_\theta) = E_t\left[\mathbf{A}_t^\pi(s_t, a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \quad (1)$$

The advantage function gives a numerical estimate of how good or bad an action is relative to the average available actions from the current state. Generating the advantage is left up to the critic.

Advantage function (Keng & Graesser, 2019):

$$\mathbf{A}^\pi(s, a) = \mathbf{Q}^\pi(s, a) - \mathbf{V}^\pi(s), \quad (2)$$

where $\mathbf{Q}^\pi$ is the action-value function, $\mathbf{V}^\pi$ is the state-value function, and s and a represent a state and action respectively. The main two methods for estimating the advantage function are n-step returns and Generalized Advantage Estimation (GAE) (Keng & Graesser, 2019).

By combining the actor and the critic, the A2C algorithm comes together, as seen in Algorithm 1 (Keng & Graesser, 2019). MSE stands for mean squared error in the algorithm, and SGD stands for stochastic gradient descent. In the algorithm, entropy regularization is used to improve policy gradients by encouraging various actions to be taken, enhancing its exploration abilities. The parameter β controls the balance between wanting more diversity and reducing the advantage term. Using entropy regularization is optional in A2C. If it is unwanted, it can be set to zero, and it won't be considered when calculating the policy loss (Keng & Graesser, 2019).

Algorithm 1 Advantage Actor-Critic (A2C)

Input:
$\alpha_A \geq 0$ # Actor learning rate
$\alpha_C \geq 0$ # Critic learning rate
$\beta \geq 0$ # Entropy regularization weight
Output: Optimized actor and critic with θ_A and θ_C
Randomly initialize actor and critic parameters θ_A, θ_C
for $episode = 0 \ldots MAX\_EPISODE$ **do**
  Act in environment using current policy
  $\hookrightarrow$ to gather & store data (s_t, a_t, r_t, s'_t)
  for $t = 0 \ldots T$ **do**
    Calculate predicted $\mathbf{V}$-value $\hat{\mathbf{V}}^\pi(s_t)$ using the critic network θ_C
    Calculate the advantage $\hat{\mathbf{A}}^\pi(s_t, a_t)$ using the critic network θ_C
    Calculate target $\mathbf{V}^\pi_{tar}(s_t)$ using the critic network θ_C & trajectory data
    Calculate entropy $\mathbf{H}_t$ of the policy distribution using the actor network θ_A
  end for
  Calculate value loss, for example using MSE:
  $\mathbf{L}_{val}(\theta_C) = \frac{1}{T}\sum_{t=0}^{T}(\hat{\mathbf{V}}^\pi(s_t) - \mathbf{V}^\pi_{tar}(s_t))^2$
  Calculate policy loss:
  $\mathbf{L}_{pol}(\theta_A) = \frac{1}{T}\sum_{t=0}^{T}(-\hat{\mathbf{A}}^\pi(s_t, a_t)\log \pi_{\theta_A}(a_t|s_t) - \beta\mathbf{H}_t)$
  Update critic parameters, for example using SGD:
  $\theta_C = \theta_C + \alpha_C \nabla_{\theta_C}\mathbf{L}_{val}(\theta_C)$
  Update actor parameters, for example using SGD:
  $\theta_A = \theta_A + \alpha_A \nabla_{\theta_A}\mathbf{L}_{pol}(\theta_A)$
end for

2.1.1. n-Step Returns

To calculate the advantage $\mathbf{A}^\pi$, we must acquire estimates for $\mathbf{Q}^\pi$ and $\mathbf{V}^\pi$. This is commonly done by first learning $\mathbf{V}^\pi$ and using the result along with a single trajectory reward to estimate $\mathbf{Q}^\pi$. Otherwise, we would either need two neural networks for both $\mathbf{Q}^\pi$ and $\mathbf{V}^\pi$, which would result in inefficient learning and additional work to ensure consistency between estimates. Or learn $\mathbf{Q}^\pi$ to estimate $\mathbf{V}^\pi$, which is a more complicated and computationally expensive alternative (Keng & Graesser, 2019).

To estimate $\mathbf{Q}^\pi$ from $\mathbf{V}^\pi$ we make use of Equation 3,

$$\begin{aligned}\mathbf{A}^\pi_{NSTEP}(\mathbf{s}_t, \mathbf{a}_t) &= \mathbf{Q}^\pi(\mathbf{s}_t, \mathbf{a}_t) - \mathbf{V}^\pi(\mathbf{s}_t) \\ &\approx \mathbf{r}_t + \gamma \mathbf{r}_{t+1} + \gamma^2 \mathbf{r}_{t+2} + \cdots + \gamma^n \mathbf{r}_{t+n} \\ &\quad + \gamma^{n+1} \hat{\mathbf{V}}^\pi(\mathbf{s}_{t+n+1}) - \hat{\mathbf{V}}^\pi(\mathbf{s}_t),\end{aligned} \tag{3}$$

where the Q-function is made up of expected rewards over n time steps plus the state-value function at the next nth step $\mathbf{V}^\pi(s_{n+1})$. The rewards from n-step come from only a single trajectory, making them unbiased but with a resulting high variance. The estimation $\hat{\mathbf{V}}^\pi(s)$ is calculated using a function approximator, making it biased; however, since it is the expectation over all trajectories, it has a lower variance. A bias-variance tradeoff when estimating the advantage is controlled by the hyperparameter n, which is the number of steps of reward. This tradeoff with n represents a hard choice, meaning it strictly determines the point at which the high-variance rewards are swapped for the $\mathbf{V}$-function estimate. Larger values of n result in the estimator having more bias and less variance, while smaller values result in the opposite, less bias and higher variance (Keng & Graesser, 2019).

2.1.2. Generalized Advantage Estimate (GAE)

In 2015, Schulman et al. proposed an enhancement on estimating the advantage function using n-step returns termed a Generalized Advantage Estimation (GAE) (Schulman et al., 2018). It does away with having to explicitly choose a value of n and instead mixes multiple different n-step returns in the form of a weighted average. The hope is to introduce as little bias as possible while simultaneously decreasing the variance of the estimator (Keng & Graesser, 2019).

Equation 4 presents the defined GAE as the exponentially weighted average overall n-step forward return advantages, each having a different variance and bias (Keng & Graesser, 2019).

$$\begin{aligned}\mathbf{A}^\pi_{GAE}(\mathbf{s}_t, \mathbf{a}_t) &= \sum_{\ell=0}^{\infty} (\gamma\lambda)^\ell \delta_{t+\ell}, \\ &\textbf{where } \delta_t = \mathbf{r}_t + \gamma \mathbf{V}^\pi(\mathbf{s}_{t+1}) - \mathbf{V}^\pi(\mathbf{s}_t)\end{aligned} \tag{4}$$

Hyperparameter λ is used to control the variance. The 1-step δ_t term is weighed the most by GAE, initially having low variance and high bias. Still, over numerous n steps, other estimators with high variance and low bias contribute to the calculation. As the number of steps increases, the term decays exponentially. Hence, λ is used to control the rate of exponential decay. Larger values of λ provide more weight to the actual rewards, while smaller values provide more weight to the $\mathbf{V}$-function estimate (Keng & Graesser, 2019). Since λ allows other contributing estimators outside of the strict range from weighing the $\mathbf{V}$-function estimate more to weighing the actual rewards more, its choice is considered soft.

Algorithm 2 Asynchronous Parallelization

Set learning rate α
Initialize global netowkr θ_G
Initialize N worker networks $\theta_{W,1}, \theta_{W,2}, \ldots, \theta_{W,N}$
for each worker **asynchronously do**
  Pull paramters from the global network and set $\theta_{W,i} \leftarrow \theta_G$
  Collect trajectories
  Compute gradient $\nabla_{\theta_{W,i}}$
  Push $\nabla_{\theta_{W,i}}$ to the global network
end for
On receiving a worker gradient, update the global network $\theta_G \leftarrow \theta_G + \alpha \nabla_{\theta_{W,i}}$

2.2. Asynchronous Advantage Actor-Critic (A3C)

Asynchronous Advantage Actor-Critic (A3C) is an asynchronously (nonblocking) parallelized Actor-Critic algorithm. The benefit of parallelized methods in reinforcement learning is to speed up the wall-clock training time and help on-policy algorithms generate more diverse training data. The idea is to create multiple agents parameterized by a network to continuously gather independent trajectory data and share it immediately (asynchronous) with a global network to update its parameters, where the workers are periodically updated by the global network making their parameters marginally different. The worker gradient calculation for asynchronous parallelization is given in Algorithm 2 (Keng & Graesser, 2019).

2.3. Proximal Policy Optimization (PPO)

Schulman et al. proposed a family of Proximal Policy Optimization (PPO) algorithms in 2017 as an efficient and straightforward extension on REINFORCE or Actor-Critic by replacing its objective with a modified surrogate objective (Schulman et al., 2017). A hurdle when training policy gradient algorithms is they are prone to struggle with performance collapse, where an agent's performance takes an abrupt drastic hit. This immediate drop in performance causes the agent to get stuck, rendering it unable to recover since it begins to generate poor trajectories which train the future policy. Also, on-policy algorithms cannot reuse data, making them sample-in-efficient. PPO's introduction of the surrogate objective tackles both of these issues, performance collapse and sample in-efficiency. The surrogate objective does away with performance collapse by guaran-

teeing the policy is continually improving, and it makes use of off-policy data during training (Keng & Graesser, 2019).

The original surrogate objective can be seen in Equation 5 (Keng & Graesser, 2019) as the conservative policy iteration (CPI),

$$\mathbf{J}^{CPI}(\theta) = \mathbf{E}_t[\mathbf{r}_t(\theta)\mathbf{A}_t^{\pi_{\theta_{old}}}], \tag{5}$$

where $\mathbf{r}_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ which is an approximation of the new policy using the old by importance sampling and $\mathbf{A}_t$ is the advantage.

PPO algorithms use straightforward, effective heuristics to solve the trust-region policy optimization problem, meaning it gives a reasonable approximation. There are two variants, one with an adaptive KL penalty and one which clips the objective. In the original PPO paper, the clipped surrogate objective outperformed the KL-penalized objective, was easier to implement, and was less computationally expensive, making it the preferred choice (Keng & Graesser, 2019).

2.4. PPO with Adaptive KL penalty

The PPO variant PPO with adaptive KL penalty turns the KL constraint, which controls how far the new policy can be from the last, into an adaptive KL penalty. The KL penalty is subtracted from the importance-weighted advantage $\mathbf{J}^{CPI}(\theta)$. The new objective to be maximized can be seen in Equation 6 (Keng & Graesser, 2019),

$$\mathbf{J}^{KLPEN}(\theta) = \max_\theta \mathbf{E}_t[\mathbf{r}_t(\theta)\mathbf{A}_t - \beta\mathbf{KL}(\pi_\theta(a_t|s_t)||\pi_{\theta_{old}}(a_t|s_t))], \tag{6}$$

where the size of the penalty is controlled by β, which contains the trust region, i.e., how big the neighbourhood is for considered policies in policy space. When β is large, the difference between the new policy and the old policy can be significant, and the opposite is true when β is small. β is adaptively updated after each policy update by a heuristic. Adaptively changing the KL penalty coefficient can be seen in Algorithm 3. Essentially, we predetermine a target δ, which limits how large the divergence can be between the new and old policy, to compare it with an estimated δ to adjust the parameter further β attempting to keep the computed δ as close to the target as possible. This approach is computationally expensive due to needing to calculate KL and still requires choosing a target δ. PPO with a clipped surrogate objective was introduced to take care of these issues (Keng & Graesser, 2019).

Algorithm 3 Adaptive KL Penalty Coefficient

Set a target value δ_{tar} for the expectation of KL
Initialize β randomly
Use multiple epochs of minibatch SDG to maximize the KL-penalized surrogate objective:$\mathbf{J}^{KLPEN}(\theta) = \mathbf{E}_t[\mathbf{r}_t(\theta)\mathbf{A}_t - \beta\mathbf{KL}(\pi_\theta(a_t|s_t)||\pi_{\theta_{old}}(a_t|s_t))]$
Compute $\delta = \mathbf{E}_t[\mathbf{KL}(\pi_\theta(a_t|s_t)||\pi_{\theta_{old}}(a_t|s_t))]$
if $\delta < \frac{\delta_{tar}}{1.5}$ **then**
  $\beta \leftarrow \frac{\beta}{2}$
else if $\delta > \delta_{tar} \times 1.5$ **then**
  $\beta \leftarrow \beta \times 2$
end if

2.5. PPO with clipping

PPO with clipped surrogate objective improves PPO with adaptive KL penalty since it does away with the KL constraint, is easier to conceptualize, and is more computationally efficient. The modified surrogate objective can be seen in Equation 7 (Keng & Graesser, 2019),

$$\mathbf{J}^{CLIP}(\theta) = \mathbf{E}_t\left[\min\left(\mathbf{r}_t(\theta)\mathbf{A}_t, clip(\mathbf{r}_t(\theta), 1-\epsilon, 1+\epsilon)\mathbf{A}_t\right)\right], \tag{7}$$

where ϵ is a tunable and decayable hyperparameter, which defines a clipping neighbourhood $|\mathbf{r}_t(\theta) - 1| \leq \epsilon$. $\mathbf{r}_t(\theta)$ is the original surrogate objective $\mathbf{J}^{CPI}$. The clip term $clip(\mathbf{r}_t(\theta), 1-\epsilon, 1+\epsilon)\mathbf{A}_t$ bounds $\mathbf{J}^{CPI}$ to the interval $[(1-\epsilon)\mathbf{A}_t, (1+\epsilon)\mathbf{A}_t]$.

Clipping prevents immense unsafe parameter updates to the policy π_θ. Limiting $\mathbf{r}_t(\theta)$ to the ϵ-neighbourhood removes encouragement of significant policy updates that could cause $\mathbf{r}_t(\theta)$ to escape the region, making policy updates safe. Sample efficiency for PPO comes from using $\pi_{\theta_{old}}$ to generate samples; hence the objective only depends on the current policy π_θ through the importance weight. A sample trajectory can be reused multiple times during a training step to update the parameters, further increasing the algorithm's efficiency. After each training step, the old trajectory is discarded, making PPO an on-policy algorithm (Keng & Graesser, 2019).

Algorithm 4 is a PPO extension on Actor-Critic with a clipping surrogate objective (Keng & Graesser, 2019).

Algorithm 4 PPO extension on Actor-Critic with clipping

Input:
$\alpha_A \geq 0$ # Actor learning rate
$\alpha_C \geq 0$ # Critic learning rate
$\beta \geq 0$ # Entropy regularization weight
$\epsilon \geq 0$ # Clipping variable
$\mathbf{K}$ # Number of epochs
$\mathbf{N}$ # Number of actors
$\mathbf{T}$ # Time horizon
$\mathbf{M} \leq \mathbf{NT}$ # Minibatch size
Output: Optimized actor and critic with θ_A and θ_C
Randomly initialize actor and critic parameters θ_A, θ_C
Initialize old actor network $\theta_{A_{old}}$
for $i = 1, 2, 3 \ldots$ **do**
  Set $\theta_{A_{old}} = \theta_A$
  for $actor = 1, 2, \ldots, N$ **do**
    Run policy $\theta_{A_{old}}$ in environment for T time steps & collect the trajectories
    Compute advantages $A_1, \ldots, A_T$ using $\theta_{A_{old}}$
    Calculate $\mathbf{V}^{\pi}_{tar,1}, \ldots, \mathbf{V}^{\pi}_{tar,T}$ using critic network θ_C and/or trajectory data
  end for
  Let *batch* with size NT consist of the collected trajectories, advantages, and target $\mathbf{V}$-values
  for $epoch = 1, 2, \ldots, K$ **do**
    for each minibatch m in *batch* **do**
      Each computed over whole minibatch m
      Calculate $\mathbf{r}_m(\theta_A)$
      Calculate the clipped surrogate objective $\mathbf{J}^{CLIP}_m(\theta_A)$ using advantages A_m and $\mathbf{r}_m(\theta_A)$
      Calculate entropy $\mathbf{H}_m$ using the actor network θ_A
      Calculate predicted $\mathbf{V}$-value $\hat{\mathbf{V}}^{\pi}(s_m)$ using the critic network θ_C
      Calculate policy loss:
      $\mathbf{L}_{pol}(\theta_A) = \mathbf{J}^{CLIP}_m(\theta_A) - \beta \mathbf{H}_m$
      Calculate value loss:
      $\mathbf{L}_{val}(\theta_C) = MSE(\hat{\mathbf{V}}^{\pi}(s_m), \mathbf{V}^{\pi}_{tar}(s_m))$
      Update critic parameters, for example using SGD:
      $\theta_C = \theta_C + \alpha_C \nabla_{\theta_C} \mathbf{L}_{val}(\theta_C)$
      Update actor parameters, for example using SGD:
      $\theta_A = \theta_A + \alpha_A \nabla_{\theta_A} \mathbf{L}_{pol}(\theta_A)$
    end for
  end for
end for

3. Experimental Setup

Experiments were done using SLM Lab (Keng & Graesser, 2017). SLM Lab is a software framework for reinforcement learning, and it works with Atari 2600 games via the Arcade Learning Environment (ALE) (Bellemare et al., 2013). A total of five deep reinforcement learning algorithm variants were tested on Fishing Derby: Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C), each using both advantage estimates, n-step and GAE, and Proximal Policy Optimization (PPO) as an extension of A2C using the clipped surrogate objective.

Algorithms were run through SLM Lab using a generated spec file, which includes all parameters for the run. Spec files were modified based on the Pong spec file for each associated algorithm, respectively. The only performed modification was the environment. All other hyperparameters remained the same. Here is a link to the A2C Pong spec file for an idea of the parameters used a2c_gae_pong.json. SLM Lab provides a plethora of experimental data once training is complete.

The main concerning metric for this paper is $mean\_returns\_ma$, which is a moving average of returns over the total number of sessions over the total number of trials. A moving average is preferred because it considers previous values to make the graph smoother. In addition to $mean\_returns\_ma$, SLM Lab records the strength (performance relative to random baseline), efficiency (speed), stability (strength loss at checkpoints), and consistency (average width of error relative to strength) to further capture algorithm performance (Keng & Graesser, 2019). However, most of these will be overlooked, focusing on final returns and strength.

4. Results

All results will be presented in this section. The order in which the algorithms are given is as follows: A2C (n-step), A2C (GAE), PPO, A3C (n-step), A3C (GAE). Mean return and strength plots are the moving average of a single trial over four sessions over 10^6 total frames. Note: A3C has 16 total sessions but still totals the same number of frames. Final $mean\_returns\_ma$ are given in Table 1. Insights and discussions regarding the results are presented in the following section.

Alg. / Env.	Fishing Derby
A2C (n-step)	-23
A2C (GAE)	-3.1
PPO	**31.6**
A3C (n-step)	-85.58
A3C (GAE)	-92.05

Table 1. Final $mean\_returns\_ma$ for all algorithms, where the bold entry signifies the best performing algorithm

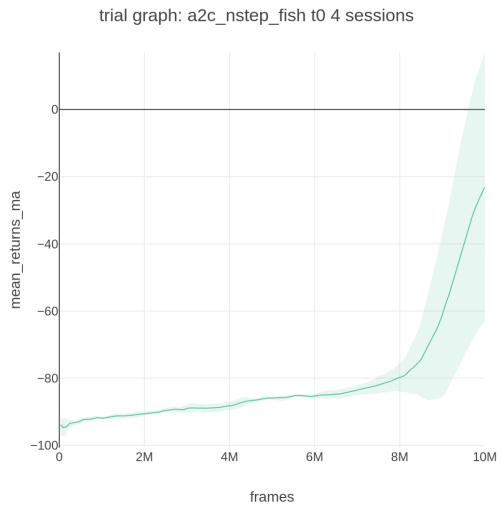

Figure 2. A2C (n-step) moving average of mean returns

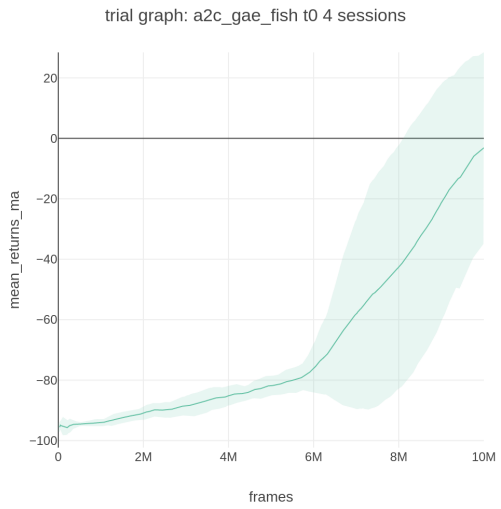

Figure 4. A2C (GAE) moving average of mean returns

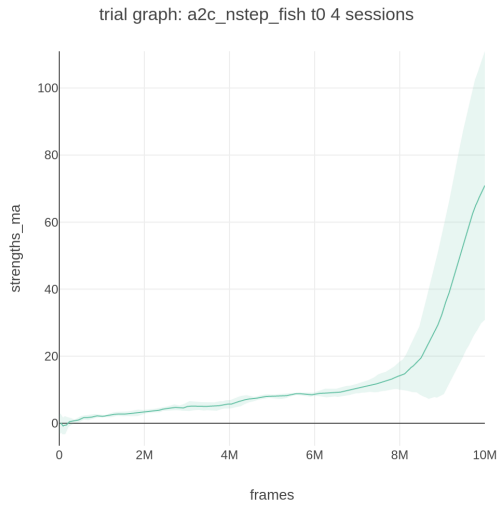

Figure 3. A2C (n-step) moving average of strength

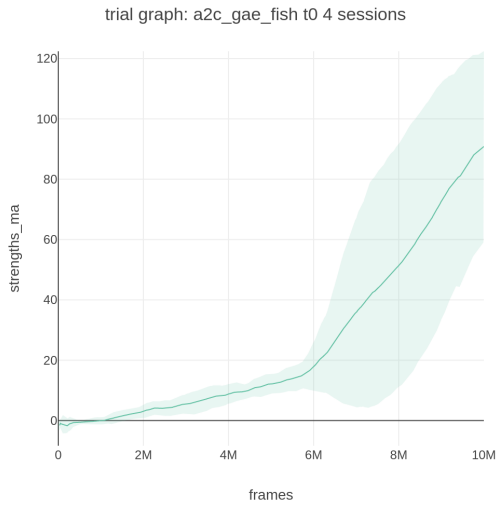

Figure 5. A2C (GAE) moving average of strength

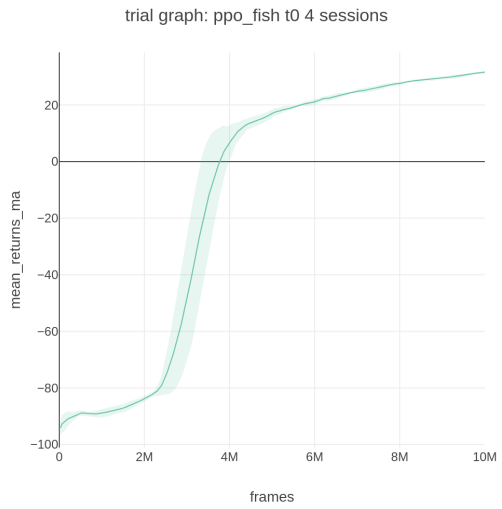

Figure 6. PPO moving average of mean returns

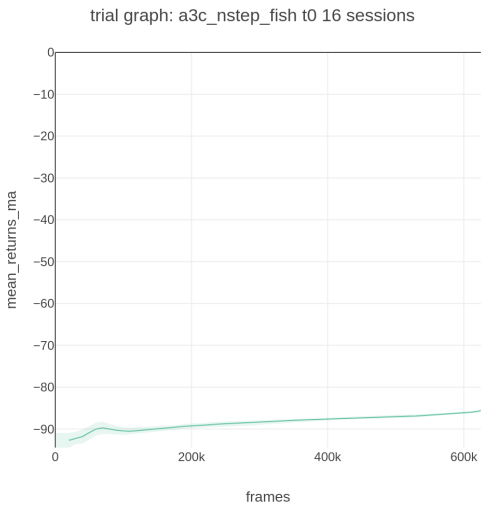

Figure 8. A3C (n-step) moving average of mean returns

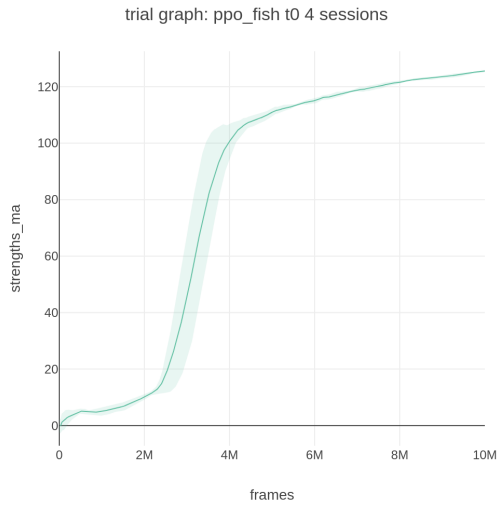

Figure 7. PPO moving average of strength

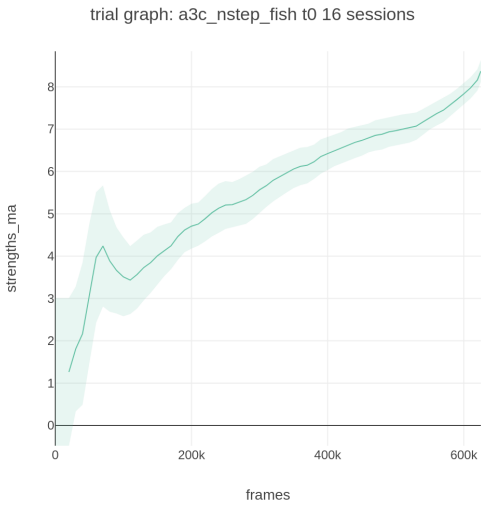

Figure 9. A3C (n-step) moving average of strength

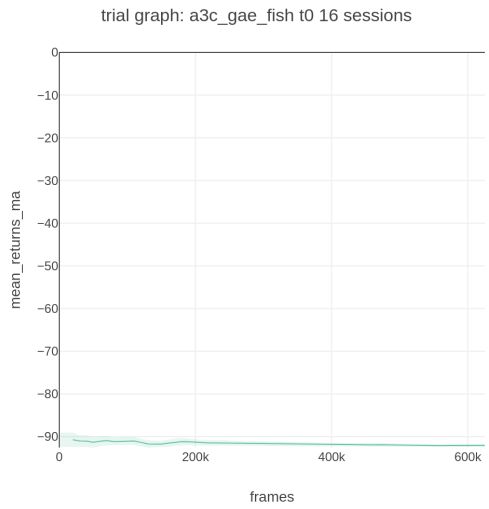

Figure 10. A3C (GAE) moving average of mean returns

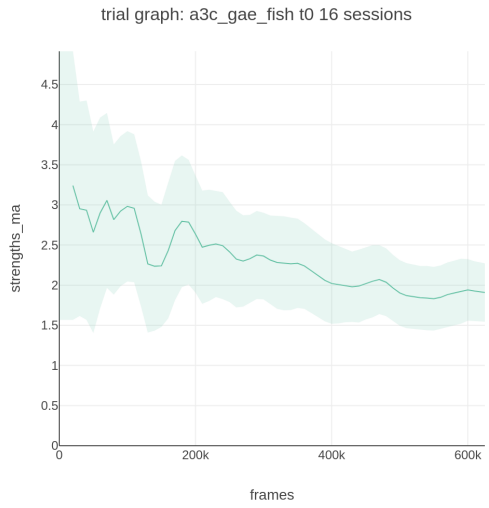

Figure 11. A2C (GAE) moving average of strength

Alg. / Env.	Fishing Derby
A2C (n-step)	1.65
A2C (GAE)	16.54
PPO	**36.03**
DQN	-88.14
DDQN+PER	-1.70

Table 2. Final $mean\_returns\_ma$ for algorithms used by the SLM Lab benchmark, where the bold entry signifies the best performing algorithm

Alg.	$strength$	$max\_strength$
A2C (n-step)	16.15	90.60
A2C (GAE)	23.85	106.19
PPO	87.84	130.30
A3C (n-step)	8.37	21.18
A3C (GAE)	1.91	8.02

Table 3. Final $strength$ and $max\_strength$ for all algorithms on the Fishing Derby environment

5. Discussion

Figure 2 presents the moving average of returns for A2C (n-step). The figure suggests that during the first eight million frames, the growth in returns is rather slow and then within the last two million frames, it grows rather quickly. It doesn't appear to have converged, so running the algorithm for more frames may result in greater returns.

A2C (GAE)'s moving average of returns is presented in Figure 4. There's slow linear growth for roughly the first six million frames and then begins to grow faster. The colour around the return line shows the standard deviation. For A2C, the standard deviation while performance is increasing is higher when using a general estimate of the advantage (GAE) in comparison to using the n-step. This is because GAE is an exponentially weighted average of multiple n-steps giving us more experiences.

The moving average of returns for PPO is given in Figure 6. PPO improves quickly between two million and four million frames and then seems to slow down quite drastically for the rest of the allotted frames. The slowdown suggests convergence, where the policy is unable to improve. Variance is consistent throughout the improvement of the algorithm's run due to the trust region created by the clipping method.

The performance of A3C for both n-step and GAE of Fishing Derby is underwhelming as seen in Figure 8 and Figure 10 respectively. Theoretically, A3C should perform on par

or better than A2C in a shorter amount of wall-clock time. It did perform significantly faster; however, it's not worth the time gained with the drop in performance. Speculation on the poor performance of A3C comes down to implementation. A3C was not tested along with the other algorithms on SLM Lab, so maybe it is not compatible with other Atari 2600 environments aside from Pong. Otherwise, it may be a machine-specific issue.

Table 1 gives the final $mean\_returns\_ma$ for all tested algorithms. The relative performance of A2C (n-step), A2C (GAE), and PPO are comparable to that achieved from SLM Lab. Final values achieved by these algorithms are not the same as those achieved by the SLM Lab benchmark, which suggests they're not entirely reproducible. The SLM Lab benchmarks for Fishing Derby are given in Table 2 (Keng & Graesser, 2017). Aside from A3C, the resulting performance of the algorithms is sensible. PPO is an improvement on A2C, and A2C (GAE) improves A2C (n-step), which has been further observed from the experiments. Therefore, the performance of the algorithms is in order theoretically and experimentally.

The strength of all algorithms in presented order can been seen in Figure 3, 5, 7, 9, and 11 respectively. Table 3 gives the $strength$ and $max\_strength$ of each of the algorithms. If $strength > 0$, then it is performing better than a random baseline. The $max\_strength$ signifies the peak performance even in the case of policy collapse (Keng & Graesser, 2017). As can be seen from Table 3 all algorithms out performed a random baseline. However, it is interesting to see that only the strength of A3C (GAE) was lower at the end of the run then at the beginning as can be seen in Figure 9.

6. Conclusion

In conclusion, five deep reinforcement learning algorithm variants were tested on the Atari 2600 game Fishing Derby: Advantage Actor-Critic (A2C) (n-step), Advantage Actor-Critic (A2C) (Generalized Advantage Estimate (GAE)), Proximal Policy Optimization (PPO) as an extension on A2C, and Asynchronous Advantage Actor-Critic (A3C) for both advantage estimate versions. It was found that PPO outperformed all other algorithms on the environment based on the final mean returns. All algorithms also exceeded a random baseline based on their resulting $strength$.

Future work involves testing different sets of hyperparameters for each algorithm independently to see how they influence the performance given the environment. Additional work should be done to reason why A3C performed so poorly compared to the other algorithms. This finding may come indirectly by testing various hyperparameters.

References

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

Keng, W. L. and Graesser, L. Slm lab. https://github.com/kengz/SLM-Lab, 2017.

Keng, W. L. and Graesser, L. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Professional, 2019.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation, 2018.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

Ye, W., Liu, S., Kurutach, T., Abbeel, P., and Gao, Y. Mastering atari games with limited data, 2021.