Deep Reinforcement Learning: All Actions Optimization (AAO)

Jason Whitmore

1 Introduction

The field of reinforcement learning (RL) involves solving Markov Decision Processes (MDPs/ environments) which consists of a finite set of states ($s \in S$), actions ($a \in A$), a reward function ($R : S \times A \times S \rightarrow \mathbb{R}$), a state probability transition function ($p : S \times A \times S \rightarrow [0, 1]$), and a discount factor $\gamma \in [0, 1)$. The goal of reinforcement learning is to find an optimal policy $\pi^*(s)$ which maps states to actions such that the associated performance function, the value function $V_{\pi*}(s)$, is maximized.

More recently, the attention of reinforcement learning has shifted into the subfield of deep reinforcement learning (DRL) which utilize deep neural networks as function approximators for existing RL methods. Notable successes with DRL include AlphaZero, AlphaStar, and OpenAI Five, which were able to successfully play games of various complexity at skill levels equal or superior to human ability.

As impressive as these achievements are, there remain several issues with DRL. Most notably is that the cost of training DRL agents can be extremely high, both in money and time. Expending large amounts of resources to train a DRL agent can be described as poor learning efficiency, which remains an open research topic.

2 Related works

2.1 Monte carlo and TD learning

Reinforcement learning techniques are often classified into different groups based on the strategies and information they use to solve MDPs (or environments). On policy methods which estimate the value function, $V_\pi(s)$, for a particular policy, π, can achieve this primarily in two ways, using Monte Carlo or temporal difference (TD) learning. This process of value function estimation is called policy evaluation.

Monte Carlo methods use reward data directly collected from interacting with the environment to form estimates of the value function. In other words, the agent runs the policy in the environment for an entire episode, collecting rewards directly and then calculating the discounted reward sum as $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = G_t$ for every non terminal timestep t. G_t is then used as a target to adjust the value function estimation $v_\pi(s_t)$ for each state s_t encountered during the episode. This process often involves averaging together multiple G_t samples for a specific state s_t to improve value function estimation accuracy [1, p. 91].

The TD learning method of value function estimation relies on sampled information from the environment in addition to the dynamic programming definition of the value function to form targets for the value function estimation. Where Monte Carlo uses $G_t = r_t + \gamma G_{t+1}$ as the target for value estimation at some state s_t, TD learning uses $r_t + \gamma V_\pi(s_{t+1})$ as a target. This technique uses "bootstrapping" since the target of the value function is created by using the current estimate of the value function. Since this method only requires the immediate reward, r, and the next state, s_{t+1}, from the environment, the estimate of the value function can be adjusted at every timestep rather than waiting for the end of the episode [1, p. 120].

The primary advantage that TD learning has over Monte Carlo methods for value function estimation is that Monte Carlo methods can only feasibly work on environments with shorter episode lengths, while bootstrapping methods can work on very long episode lengths, or even infinite length episodes [1, p. 124]. For this reason, TD learning methods are of particular interest.

2.2 One step actor critic

The one step actor critic algorithm provides the basic template for TD learning using parameterized functions. The basic idea of the algorithm revolves around three steps: calculating the TD error, adjusting the value function (policy evaluation), and adjusting the policy function (policy improvement):

$$\delta = (r + \gamma v(s_{t+1}; \theta_v)) - v(s_t; \theta_v)$$

$$\theta_v \leftarrow \theta_v + \alpha_v \delta \nabla_{\theta_v} v(s_t; \theta)$$

$$\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \delta I \nabla_{\theta_\pi} ln(\pi(a_t|s_t; \theta))$$

[1, p. 332]. The TD error serves two purposes in this algorithm. First, it provides a learning signal for the value function output $v(s_t; \theta_v)$ in order to align more closely with the bootstrapped return, $r + \gamma v(s_{t+1}; \theta_v)$. Note that the lowercase function names, like "v", are used to denote a function approximation implementation of the uppercase counterpart.

Second, it provides a learning signal which determines whether or not the taken action, a_t, was better or worse for the agent compared to the policy's baseline performance at state s_t, $v(s_t; \theta_v)$. When $\delta > 0$, the action taken, a_t, produced higher returns than expected, and the probability of action a_t being selected at state s_t should increase. Likewise, when $\delta < 0$, the action taken, a_t, produced lower returns than expected, and the probability of action a_t being selected at state s_t should decrease.

These two steps represent the policy evaluation and policy improvement steps that are the core of on-policy learning.

Since this algorithm adjusts the policy based on the TD error of a single action, it belongs to a class of algorithms which will be called single action optimization (SAO) algorithms in this paper.

2.3 Cost of training deep reinforcement learning agents

Although the theory behind reinforcement learning is sound, in practice, the techniques tend to be very expensive to implement effectively in novel environments. For example, OpenAI's *DOTA2* deep reinforcement learning agent, OpenAI Five, used 256 GPUs, 128,000 CPU cores, and trained for 180 game hours every day [2]. Additionally, OpenAI Five's neural network models contained 158,502,815 parameters [3] that are used in expensive matrix and vector operations that neural networks perform. The *StarCraft II* AlphaStar bot uses 55 million parameters for inference [4]. Although exact figures are not available, these large amounts of computing resources are likely to be quite expensive to use.

Even among simpler environments, deep reinforcement learning agents take a long period of time to train. For example, Proximal Policy Optimization (PPO), a state of the art algorithm, takes 10 million timesteps to learn a good - but not optimal - policy, for playing the Atari 2600 game of *Breakout* [5].

These drawbacks to current state of the art techniques present a desire for more efficient DRL algorithms. It is plausible that a more efficient DRL algorithm combined with the current state of computing resources can provide better performance compared to current state of the art results.

3 Proposed method

3.1 Advantage learning

As discussed in sections 2.1 and 2.2, the TD learning model is often used in reinforcement learning due to its ability to be implemented in an online fashion, where adjustments to both the policy and value functions can be done at each timestep. For the policy improvement step, the TD error, δ_t, is used as a learning signal to either increase or decrease the probability of the action a_t taken at the state s_t. δ_t is calculated as:

$$\delta_t = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$$

The observation can be made that the term $r_t + \gamma V_\pi(s_{t+1})$ approximates $Q_\pi(s_t, a_t)$ since it is the estimated value under the current policy after taking action a_t, which happens to also be chosen using the current policy. This TD error approximation can now be written as:

$$\delta_t \approx Q_\pi(s_t, a_t) - V_\pi(s_t)$$

In order to make broader adjustments to the policy during training, adjustments considering all actions can be made, rather than only considering the specific action performed at timestep t, a_t. Although the action set A is part of the MDP framework, it can still be used in model free implementations since the agent is aware of what actions can be performed. This can be accomplished via the advantage function, $A_\pi(s, a)$:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$$

This advantage function serves as a learning signal similar to the TD error described in section 2.2. When $A_\pi(s,a) > 0$, then taking action a at state s will lead to greater expected returns compared to the current policy, and the probability of taking action a at state s, $\pi(a|s)$ should be increased. Likewise, when $A_\pi(s,a) < 0$, then taking action a at state s will lead to lower expected returns compared to the current policy, and $\pi(a|s)$ should be decreased.

One difficulty with using the definition for $A_\pi(s,a)$ is that a deep learning implementation would require 3 parameterized neural networks: $\pi(a|s;\theta_\pi)$, $q(s,a;\theta_q)$, and $v(s;\theta_v)$. Instead, since the value function $V_\pi(s)$ is the expected return, the function can be rewritten as a sum of probabilities of actions multiplied by the expected return of taking action a at state s under policy π:

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a)$$

Substituting this equation into the advantage function definition, a new definition is obtained which only uses a policy function $\pi(a|s)$ and a Q function, $Q_\pi(s,a)$:

$$A_\pi(s,a) = Q_\pi(s,a) - \sum_{x \in A} \pi(x|s) Q_\pi(s,x)$$

3.2 TD All Action Optimization (TD AAO)

As described in section 2, on policy reinforcement learning algorithms involve the policy evaluation and policy improvement steps. Using the TD error with traditional actor-critic algorithms is fairly straightforward as only $V_\pi(s_t)$ and $\pi(a_t|s_t)$ need to be adjusted.

However, a TD error algorithm that adjusts the policy considering all actions at a timestep requires more computations per step, but may be more beneficial in decreasing the number of environment interactions needed to achieve some high performance (resulting in a higher learning efficiency). This can be accomplished by training a Q function, then using it as part of an advantage function approximator to adjust the policy.

3.2.1 Policy evaluation

For policy evaluation, $Q_\pi(s_t,a_t)$ will need to be adjusted before the improvement step. As mentioned in section 3.1, the term $r_t + \gamma V_\pi(s_{t+1})$ approximates $Q_\pi(s_t,a_t)$ and $V_\pi(s_t) = \sum_a \pi(a|s_t) Q_\pi(s_t,a)$. Combining these terms together, the Q function error, δ_Q, can be determined using collected information, the Q function, and the policy function:

$$\delta_Q = (r_t + \gamma(\sum_a \pi(a|s_{t+1}) Q_\pi(s_{t+1},a))) - Q_\pi(s_t,a_t)$$

δ_Q provides the learning signal for the Q function. A $\delta_Q > 0$ means that $Q_\pi(s_t,a_t)$ is too low and should be increased and a $\delta_Q < 0$ means that $Q_\pi(s_t,a_t)$ is too high and should be decreased. This update rule can be expressed using a parameterized Q function:

$$\theta_q \leftarrow \theta_q + \alpha_q \delta_q \nabla_{\theta_q} q(s_t,a_t;\theta_q)$$

Where $\alpha_q > 0$ is a user defined learning rate. This learning rate follows the same intuition as learning rates in machine learning: too small of a value and learning becomes too slow, too high of a value and learning becomes unstable.

3.2.2 Policy improvement

The policy improvement step of this algorithm simply uses the advantage function $A_\pi(s_t, a)$ to determine whether or not an action's probability should be increased at state s_t. Unlike SAO techniques which simply use the single TD error, δ, this AAO method will use several error terms denoted δ_a, which represents $A_\pi(s_t, a)$ for every action. This can be expressed using parameterized functions:

$$\delta_a = q(s_t, a; \theta_q) - \sum_{x \in A} \pi(x|s_{t+1}; \theta_\pi) q(s_{t+1}, x; \theta_q)$$

δ_a provides the learning signal for the policy function. A $\delta_a > 0$ means that the probability of action a being taken at state s_t should be increased and a $\delta_a < 0$ means that the probability of action a being taken at state s_t should be decreased. These learning signals can be used for every action. This update rule can be expressed using as:

$$\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \sum_a \delta_a \nabla_{\theta_\pi} \pi(a|s_t; \theta_\pi)$$

Where $\alpha_\pi > 0$ is a user defined learning rate which follows the same constraints as α_q.

3.2.3 Algorithm pseudocode

The algorithm pseudocode is slightly modified from the learning rules expressed in the previous sections. These modifications make direct implementation easier, such as using the "prime" notation on states and rewards rather than timestep subscripts (s and s' versus s_t and s_{t+1}, etc).

```
for  episode = 1,2,3,4,... do
    Initialize $s$ with a starting state
    while $s$ is not a terminal state do
        Select action $a$ from the current policy
        Take action $a$, observe $s'$, $r$

        //Policy evaluation step
        $V' \leftarrow 0$
        if $s'$ is not a terminal state then
            for action = 1, 2, ..., $|A|$ do
                $V' \leftarrow V' + \pi(action|s';\theta_\pi) * q(s', action; \theta_q)$
            end
        end

        $\delta_q \leftarrow (r + V') - q(s, a; \theta_q)$
        $\theta_q \leftarrow \theta_q + \alpha_q \delta_q \nabla_{\theta_q} q(s, a; \theta_q)$

        //Policy improvement step
        $V \leftarrow 0$
        for action = 1, 2, ..., $|A|$ do
            $V \leftarrow V + \pi(action|s;\theta_\pi) * q(s, action; \theta_q)$
        end

        Let $\vec{g}$ be a zero vector with the same length as $\vec{\theta_\pi}$
        for action $= 1, 2, ..., |A|$ do
            $\delta_a \leftarrow q(s, action; \theta_q) - V$
            $\vec{g} \leftarrow \vec{g} + \delta_a \nabla_{\theta_\pi} \pi(action|s;\theta_\pi)$
        end
        $\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \vec{g}$

        s $\leftarrow s'$
    end
end
```

Algorithm 1: TD AAO algorithm

4 Experimental setup

In order to evaluate the performance of the TD AAO algorithm, a learning efficiency experiment will be conducted. For a baseline, a TD SAO algorithm will be used that closely resembles the form of the TD AAO algorithm, with the difference being it only optimizes for the sampled action a rather than all actions. The similarities between these two algorithms helps keep the experiment fair. This algorithm will be called TD SAO.

```
for  episode = 1,2,3,4,... do
    Initialize $s$ with a starting state
    while $s$ is not a terminal state do
        Select action $a$ from the current policy
        Take action $a$, observe $s'$, $r$

        $\delta \leftarrow r + \gamma V(s';\theta_v) - V(s;\theta_v)$

        //Policy evaluation step
        $\theta_v \leftarrow \theta_v + \alpha_v \delta \nabla_{\theta_v} V(s;\theta_v)$

        //Policy improvement step
        $\theta_\pi \leftarrow \theta_\pi + \alpha_\pi \delta \nabla_{\theta_\pi} \pi(a|s;\theta_\pi)$

        s $\leftarrow s'$
    end
end
```

Algorithm 2: TD SAO algorithm

It should be noted that this algorithm does not exactly contain the equations from the One Step Actor-Critic algorithm seen in section 2.2. Notably, this algorithm does not use the scalar $I = \gamma^t$ when adjusting the policy function. In the One Step Actor-Critic algorithm, the goal was to optimize the value of some starting state, $V_\pi(s_0)$, which means that the adjustments to the policy had to exponentially decrease with γ since future rewards mattered less than more immediate rewards. The I scalar was used to implement this effect.

In this paper, the goal is to optimize the value function at every state. Although this technically is not a policy gradient, since it involves optimizing for multiple scalars ($V_\pi(s)$ for all $s \in S$) rather than just one ($V_\pi(s_0)$), the two goals are fairly similar.

To simplify, the TD AAO algorithm will be referred to as "AAO" and the TD SAO algorithm will be referred to as "SAO" for the rest of this paper.

4.1 Environments

The environments to be used in the experiments are `CartPole-v2`, `LunarLander-v2` from the OpenAI Gym package, along with a custom environment called `Gridworld-10x10-1r-v0`. These environments were chosen for various reasons, such as difficulty level, observation space size, time horizon, and action set size.

The `CartPole-v2` environment is a simple environment commonly used for educational purposes. The goal of the agent is to balance a pole which is sitting upright on a cart. The agent has control over the cart's motion and receives rewards for each timestep that the pole remains upright. For most algorithms, this task is easy to solve due to the small action set (2 actions total for moving the cart left and right), simple observation space, and short episodes (200 timesteps maximum).

The `LunarLander-v2` environment is a slightly more complicated environment yet still simple in its design. The goal of the agent is to successfully land a spacecraft on a lunar surface while using as little fuel as possible. This environment is more complicated than

CartPole-v2 since it has more actions (4 actions total, for controlling thrusters and a NoOp action) and possibly a much longer time horizon. Some aspects of this environment are similar to CartPole-v2, such as the small observation space and rewards that are given often.

The Gridworld-10x10-1r-v0 environment is a simple custom environment created for this experiment. As the name suggests, the environment is a 10 by 10 gridworld, where a randomly placed agent must collect one reward that is also randomly placed. The action set contains 5 actions, moving the agent up, left, down, right, as well as a "pick up" action. In order to collect the reward, the agent must move to the same location as the reward, then select the "pick up" action. The reward function gives the agent a reward of 1 upon collecting the reward, but also gives a very small penalty proportional to the Manhattan distance between the agent and the reward, which helps speed up learning. The state space is represented by a vector of length 6, which contains the following information: agent x location, agent y location, episode progress (current timestep / 10,000 (maximum timesteps in an episode)), reward x location, reward y location, and a binary value indicating if the reward has been picked up or not.

These 3 environments will be used in the 2 experiments to evaluate the AAO algorithm.

4.2 Learning efficiency experiment

As mentioned in the introduction, it is desirable in DRL to find new techniques which are more learning efficient than previous techniques. Specifically, learning efficiency could be interpreted as achieving the highest change in overall reward while using the fewest resources, which could be time and/or compute costs. In this experiment, we will measure the learning efficiency as change in score over change in timesteps:

$$\text{Learning efficiency} = \frac{\Delta\text{score}}{\Delta\text{timesteps}}$$

Where score is defined as the summed rewards from a start state to the end of an episode. For the purposes of this experiment, timesteps can be used instead of Δtimesteps since the experiments will be done from timestep 0. Δscore will still need to be used, since even a random policy may produce scores that are not equal to zero. With these two considerations, the learning efficiency definition can be further defined:

$$\text{Learning efficiency} = \frac{\text{trained policy score} - \text{random policy score}}{\text{timesteps}}$$

Where a random policy outputs a random uniform distribution of action probabilities at every state. Since a random policy exists independently from a training algorithm, finding the random policy score in an environment is simple. Running a random policy in the 3 environments yields these results:

Environment name	Mean score from random policy ($n = 30$ episodes)
`Cartpole-v1`	23
`LunarLander-v2`	-166
`Gridworld-10x10-1r-v0`	-4.107

There are primarily two methods in which this experiment can be conducted. First, the agents could be trained on both algorithms for a fixed timestep limit. This would cause the demoninator of the learning efficiency ratio to become fixed and the numerator would be of interest.

Second, the agents could be trained on both algorithms until a certain score target is reached. This would cause the numerator of the learning efficiency ratio to become fixed, and the denominator (timestep) would be of interest.

For this experiment, the second method will be conducted. This is because the second method removes some of the guesswork in designing the experiment, since it is difficult to know ahead of time how many timesteps an algorithm needs to run but the "high score" to achieve is typically known as a property of the environment. The score targets for each environment are shown below:

Environment name	Score target
`Cartpole-v1`	495
`LunarLander-v2`	195
`Gridworld-10x10-1r-v0`	0

This experiment aims to determine the learning efficiency for both the AAO and SAO algorithms across all 3 environments and under similar conditions (see appendices A and B for specific hyperparameter values).

A single training run will start with a random policy and a randomly initialized q or v function. The policy's mean score will be calculated every 10,000 timesteps with a sample size of 30 episodes. If the evaluated mean score exceeds the target score or a timestep limit of 2,000,000 is reached, the training run ends. The final evaluated mean score and timestep number are then used with the random policy score to determine the learning efficiency of the training run.

Three of these training runs will be completed for each algorithm and environment pair so a mean learning efficiency can be calculated.

With a mean learning efficiency found for both algorithms in each environment, a learning efficiency ratio can be calculated to directly compare learning efficiency between algorithms:

$$\text{Mean learning efficiency ratio} = \frac{\text{AAO mean learning efficiency}}{\text{SAO mean learning efficiency}}$$

.

For example: a mean learning efficiency ratio of 1.5 means that AAO is approximately 50% more efficient than SAO under similar conditions.

In addition to learning efficiency, the evaluated mean score and timestep data will also be recorded so that a plot can be made, visually demonstrating training progress on all runs.

There are two research questions that can hopefully be answered by this experiment:

1. To what degree is the AAO algorithm more or less learning efficient compared to the SAO algorithm?

2. Does the AAO algorithm benefit more from environments with more actions to take compared to the SAO algorithm? The basic intuition here is that with an action set size of 2, the AAO algorithm will be able to consider 2 actions when adjusting the policy, rather than just 1 action with SAO. Can this relationship be modeled as a linear relationship between the learning efficiency ratio and the action set size?

5 Results and discussion

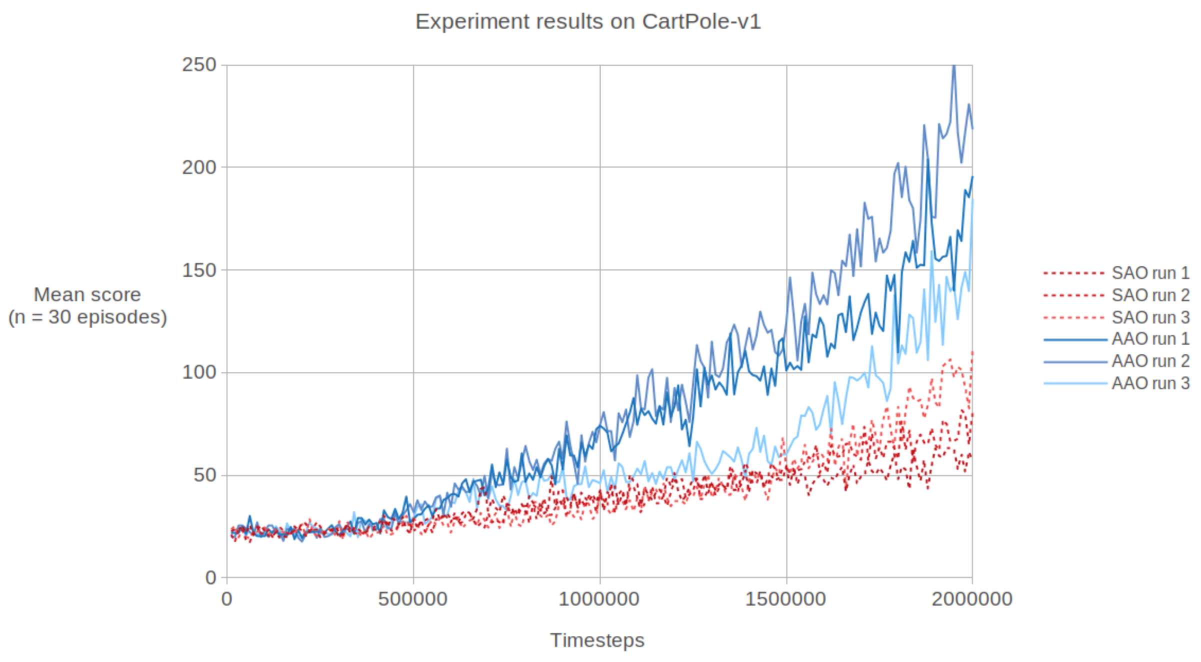
5.1 `CartPole-v1`

Experiment results table 1: AAO on `CartPole-v1`

Run Number	Mean score achieved	Number of timesteps	Learning efficiency
1	196	2,000,000	$\approx 8.65 * 10^{-5}$
2	219	2,000,000	$\approx 9.8 * 10^{-5}$
3	185	2,000,000	$\approx 8.1 * 10^{-5}$

Experiment results table 2: SAO on `CartPole-v1`

Run Number	Mean score achieved	Number of timesteps	Learning efficiency
1	57	2,000,000	$\approx 1.7 * 10^{-5}$
2	80	2,000,000	$\approx 2.85 * 10^{-5}$
3	111	2,000,000	$\approx 4.4 * 10^{-5}$



The AAO algorithm on `CartPole-v1` achieved a mean learning efficiency of approximately $8.85 * 10^{-5}$ over 3 runs. Under similar conditions, the baseline SAO algorithm on `CartPole-v1` achieved a mean learning efficiency of approximately $2.98 * 10^{-5}$ over 3 runs. The mean learning efficiency ratio is then:

$$\text{Mean learning efficiency ratio on } \texttt{CartPole-v1} \approx \frac{8.85 * 10^{-5}}{2.98 * 10^{-5}} \approx 2.97$$

These results suggest that on the `CartPole-v1` environment, the AAO algorithm is approximately 197% more learning efficient than the baseline SAO algorithm.

From the results in the `CartPole-v1` environment, it appears that the AAO algorithm is superior to the SAO algorithm in terms of learning efficiency. This superiority is visible on the plot as the gap between the AAO and SAO runs widen as training progresses. The most notable exception to this observation is AAO run 3, which lagged behind the other AAO runs and only made significant improvement compared to the SAO runs near the end of the training run. Even with this poor run, the AAO runs achieved better results compared to the SAO runs.

The mean learning efficiency ratio of 2.97 is higher than what was expected given the second research question. Since the `CartPole-v1` environment has an action set size of 2, it was expected that the mean learning efficiency ratio would also be equal to 2, but the results suggest that it is equal to approximately 2.97.

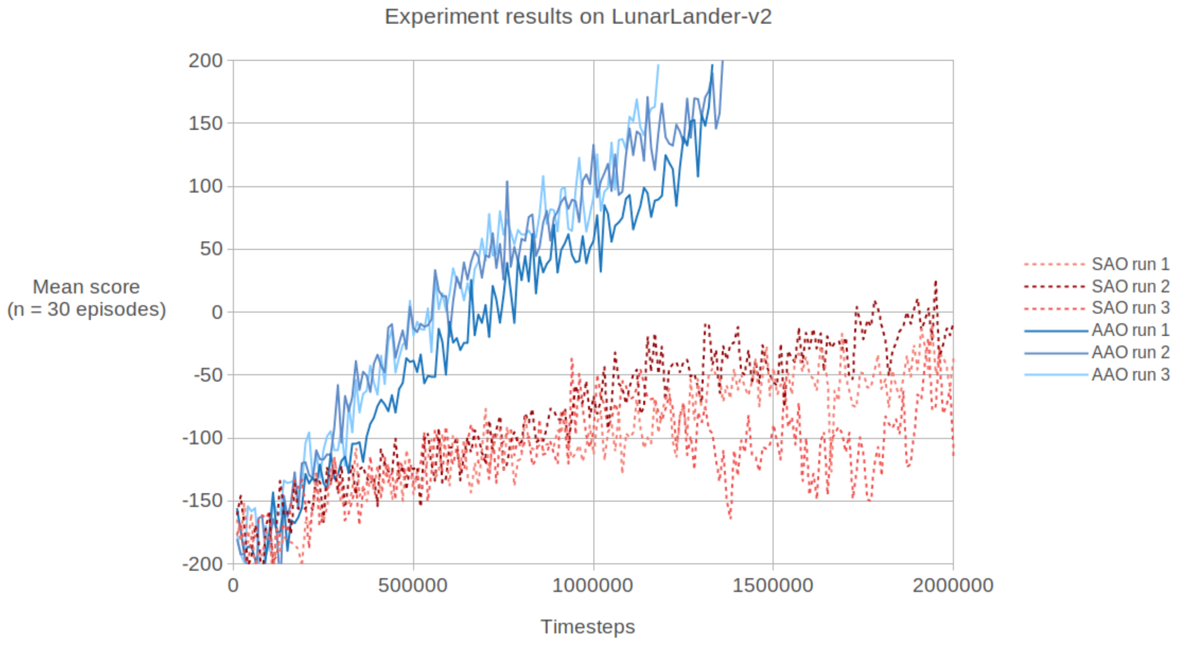
5.2 `LunarLander-v2`

Experiment results table 3: AAO on `LunarLander-v2`

Run Number	Mean score achieved	Number of timesteps	Learning efficiency
1	197	1,330,000	$\approx 2.73 * 10^{-4}$
2	207	1,360,000	$\approx 2.74 * 10^{-4}$
3	197	1,180,000	$\approx 3.07 * 10^{-4}$

Experiment results table 4: SAO on `LunarLander-v2`

Run Number	Mean score achieved	Number of timesteps	Learning efficiency
1	-34	2,000,000	$\approx 6.6 * 10^{-5}$
2	-7	2,000,000	$\approx 7.95 * 10^{-5}$
3	-155	2,000,000	$\approx 2.55 * 10^{-5}$



The AAO algorithm on `LunarLander-v2` achieved a mean learning efficiency of approximately $2.85 * 10^{-4}$ over 3 runs. Under similar conditions, the baseline SAO algorithm on `LunarLander-v2` achieved a mean learning efficiency of approximately $5.7 * 10^{-5}$ over 3 runs. The mean learning efficiency ratio is then:

$$\text{Mean learning efficiency ratio on } \texttt{LunarLander-v2} \approx \frac{2.85 * 10^{-4}}{5.7 * 10^{-5}} = 5$$

These results suggest that on the `LunarLander-v2` environment, the AAO algorithm is approximately 400% more learning efficient than the baseline SAO algorithm.

It appears that the AAO algorithm is superior to the baseline SAO algorithm when comparing learning efficiency in the `LunarLander-v2` environment. The plot of results shows a large gap in the mean score achieved between the two algorithms. In fact, the training runs ended early for the AAO algorithm since it was able to reach the score target before the timestep limit was reached. It also appears that the AAO runs were more stable, since one of the SAO runs had a significant drop in performance that diverged from the performance of the other SAO runs.

The mean learning efficiency ratio of 5 is higher than what was expected given the second research question, since the `LunarLander-v2` environment has an action set size of 4, it was expected that the mean learning efficiency ratio would also be equal to 4.

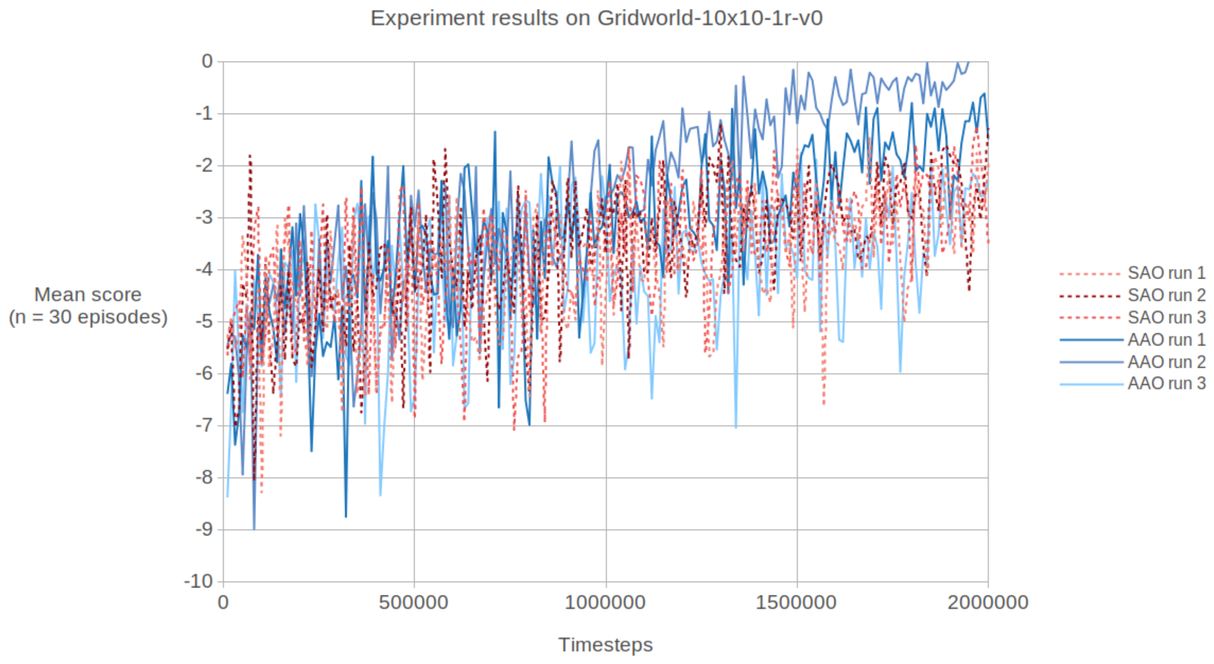
5.3 `Gridworld-10x10-1r-v0`

Experiment results table 5: AAO on `Gridworld-10x10-1r-v0`

Run Number	Mean score achieved	Number of timesteps	Learning efficiency
1	-1.491	2,000,000	$\approx 1.308 * 10^{-6}$
2	0.035	1,950,000	$\approx 2.124 * 10^{-6}$
3	-2.28	2,000,000	$\approx 9.135 * 10^{-7}$

Experiment results table 6: SAO on `Gridworld-10x10-1r-v0`

Run Number	Mean score achieved	Number of timesteps	Learning efficiency
1	-1.942	2,000,000	$\approx 1.08 * 10^{-6}$
2	-1.277	2,000,000	$\approx 1.415 * 10^{-6}$
3	-3.573	2,000,000	$\approx 2.67 * 10^{-7}$



The AAO algorithm on `Gridworld-10x10-1r-v0` achieved a mean learning efficiency of approximately $1.449*10^{-6}$ over 3 runs. Under similar conditions, the baseline SAO algorithm on `Gridworld-10x10-1r-v0` achieved a mean learning efficiency of approximately $9.207*10^{-7}$ over 3 runs. The mean learning efficiency ratio is then:

$$\text{Mean learning efficiency ratio on } \texttt{Gridworld-10x10-1r-v0} \approx \frac{1.449 * 10^{-6}}{9.207 * 10^{-7}} \approx 1.574$$

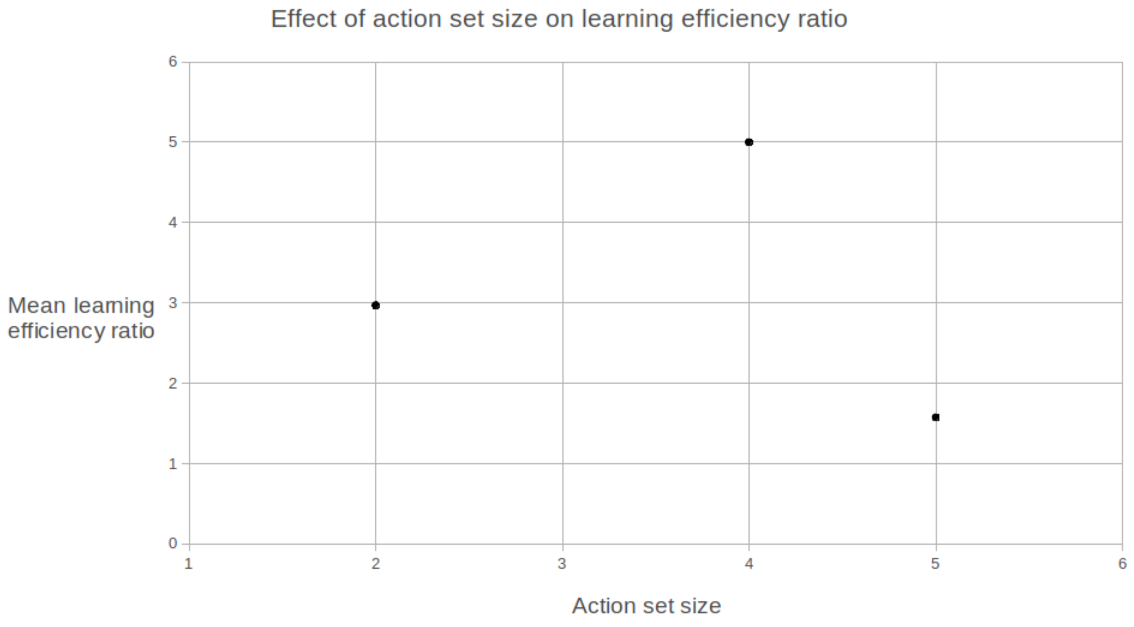
These results suggest that on the `Gridworld-10x10-1r-v0` environment, the AAO algorithm is approximately 57.4% more learning efficient than the baseline SAO algorithm.

Unfortunately, it is difficult to draw conclusions from the experiment results plot due to the noise of data samples. AAO runs 1 and 2 do appear to produce more stable score samples. These runs also appear to train policies that only produce slightly better scores compared to the SAO runs.

The mean learning efficiency ratio of 1.574 is lower than what was expected given the second research question. Since the `Gridworld-10x10-1r-v0` environment has an action set size of 5, it was expected that the mean learning efficiency ratio would also be equal to 5, but the results suggest that it is equal to approximately 1.574.

5.4 Action set size effect on mean learning efficiency ratio

With the results obtained from the experiments in sections 5.1, 5.2, and 5.3, the effect of action set size on the mean learning efficiency ratio can be analyzed. In order to do this, data points in the form (`action set size`, `mean learning efficiency ratio`) can be plot to show any potential relationship. The second research question expects a linear relationship between the data points, with a slope of 1.



As the plot shows, there does not appear to be any obvious relationship (linear or otherwise) that can be drawn from the 3 data points. Most notable is the mean learning efficiency ratio of 1.574 from the `Gridworld-10x10-1r-v0` experiment. This data point could either be an outlier to a linear relationship, or could be evidence that the AAO algorithm demonstrates worse learning efficiency compared to the SAO algorithm as the action set size increases.

Regardless of any possible relationship between the data points, it is noteworthy that all 3 experiments produced mean learning efficiency ratios greater than 1. This suggests that the AAO algorithm has a higher learning efficiency compared to the SAO algorithm.

6 Limitations

The primary limitation of the AAO algorithm appears to be related to time complexity. Specifically, the AAO algorithm takes up a significant amount of "real" time to complete a single timestep.

Using the RAM model to analyze time complexity, both the AAO and SAO algorithms have the same time complexity of $O(|A|)$ to complete a timestep. This is because both algorithms must select an action from the action probability distribution produced by the policy which involves looping over a maximum of $|A|$ elements.

Other than the action selection process (excluding obtaining the action probability distribution), the rest of the SAO algorithm only involves 1 prediction from the policy function, 2 predictions from the value function, 2 gradient calculations, and other minor operations during a timestep. The rest of the AAO algorithm involves $2|A|+1$ predictions of the policy function, $3|A|+1$ predictions of the q function, $|A|$ policy function gradient calculations, and 1 q function gradient calculation. Although these operations result in $O(|A|)$ time complexity, making predictions and calculating gradients of the functions is still a computationally expensive process.

This limitation may be alleviated via parallel computation. Specifically, there are 3 loops where the work completed can be done in any order: Calculating $V_\pi(s')$, $V_\pi(s)$, and the policy gradient $\vec{g}$. Since calculating $V_\pi(s')$ and $V_\pi(s)$ involves a sum of products (for example: $\pi(action|s';\theta_\pi) * q(s', action; \theta_q)$ for all actions), independent threads can calculate that product separately, before summing the scalar result to a shared memory location. Similarly, when calculating the policy gradient $\vec{g}$, independent threads can calculate $\nabla_{\theta_\pi}\pi(action|s;\theta_\pi)$ for each action, multiply the gradient with δ_a, then add the result to a shared memory location containing $\vec{g}$.

After these parameter changes are made to the q and policy functions, the new function parameters should be redistributed to each thread so that each thread has an accurate copy of the updated policy and q functions.

7 Future work

Although the experiment conducted in this paper provides some indication of the AAO algorithm's learning efficiency, there are a few ways the experiment can be altered to further improve confidence in the algorithm.

Like with most machine learning techniques, there is a significant cost to performing hyperparameter tuning. This is sometimes done via a grid search, where multiple hyperparameter values are evaluated for performance to determine which set of hyperparameters are best. These grid searches tend to be quite expensive to conduct, and was not done for this paper. In a future paper, more effort put into finding the best hyperparameters could improve the findings. Additionally, the learning efficiency experiment was conducted under similar conditions, which included the same learning rates for both the AAO and SAO algorithms. This was done to keep the experiment fair, but it is entirely possible that different learning rates for each algorithm could produce more learning efficient results.

More environments for the learning efficiency experiment would be beneficial for improving confidence in the results. Ideally, robust DRL techniques should aim to be "one size fits all" solutions for solving environments. Although the results from the 3 environments used in this paper provide some indication that the AAO algorithm is more learning efficient compared to SAO, utilizing more environments would help solidify these initial results. Specifically, environments with different levels of difficulty, reward sparsity, randomness, action set size, and state set size would be very useful to conduct a more extensive experiment. More environments with different action set sizes would help produce a more meaningful result when trying to determine if action set size has a linear relationship to the mean learning efficiency ratio. The 3 data points collected in this paper are inadequate to draw any sort of general conclusion.

8 Conclusion

By comparing the learning efficiency of the proposed All Actions Optimization algorithm to a traditional Single Action Optimization algorithm across 3 environments with different action set sizes, the following can be concluded to answer the research questions:

First, the proposed AAO algorithm is demonstrated to be more learning efficient compared to a traditional SAO algorithm under similar conditions and across multiple environments. In the worst case, the AAO algorithm was approximately 57.4% more learning efficient, and in the best case, was approximately 400% more efficient than SAO.

Second, there does not appear to be a relationship (linear or otherwise) between the action set size of an environment and the mean learning efficiency ratio between the AAO and SAO algorithms.

Based on the results of the experiments conducted in this paper, the proposed AAO algorithm is worthy of both further study and consideration in applied reinforcement learning settings.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, second edition. Cambridge, MA: The MIT Press, 2018.

[2] OpenAI, "OpenAI Five", *OpenAI*, June 25, 2018. [Online]. Available: https://openai.com/blog/openai-five/. [Accessed September 29, 2021]

[3] C. Berner, et. al, "Dota 2 with Large Scale Deep Reinforcement Learning", *OpenAI*, March 10, 2021. [Online]. Available: https://arxiv.org/pdf/1912.06680.pdf. [Accessed September 30, 2021]

[4] O. Vinyals, et. al, "Grandmaster level in StarCraft II using multi-agent reinforcement learning", *Nature*, vol. 575, October 30, 2019

[5] J. Schulman, et. al, "Proximal Policy Optimization Algorithms", *OpenAI*, August 28, 2017. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf

Appendix

A. Overall hyperparameters (all environments)

Hyperparameter	Value
Discount factor, γ	0.999
Number of hidden layers (all functions)	2
Number training timesteps	2,000,000
Number of timesteps between policy evaluations (evaluation interval)	10,000
Number of samples (episodes) used during policy evaluation	30

B. Environment specific hyperparameters

Hyperparameter	`CartPole-v1`	`LunarLander-v2`	`Gridworld-10x10-1r-v0`
Hidden layer sizes (all functions)	24	32	32
Policy learning rate	10^{-6}	10^{-5}	10^{-4}
Q, Value learning rate	10^{-5}	10^{-4}	10^{-2}
Frame stack	2	2	1

C. Implementation details

1. The frame (or observation) stacking implementation used to formulate the state vector has a special case when the number of timesteps encountered in an episode is less than the frame stacking amount. In this case, there are not enough observations to populate the state vector, and empty components in the state vector are simply filled with a value of -1 to represent no observation.

D. Reproducing the experiment

The source code used to implement the algorithms and perform the experiment is available at `github.com/Jason-Whitmore/rl_aao`. The `rl_aao.py` file contains all of the code needed to run the experiments. Specifically, at the bottom of the file contains the primary function calls. To switch between environments, simply change the `index` variable to either 0 (`CartPole-v1`), 1 (`LunarLander-v2`), or 2 (`Gridworld-10x10-1r-v0`). Environment specific hyperparameters are implemented as the lists `hidden_layer_sizes`, `score_targets`, `policy_lrs`, and `value_lrs` (which is also the q function learning rates). Fixed hyperparameters that do not depend on the environment are in the constructor or as a local variable at the bottom of the file.

When running the experiments, text will be displayed showing which version of `Tensorflow`, `Keras`, and `NumPy` are being used, and which version of each was used for this paper. It is highly recommended that the version numbers match for the code to run and behave in a similar manner.

As the experiment runs, a file will be written to disk in the same location as the python file with the name syntax of `exp_results_ENVNAME.csv`, where `ENVNAME` is the environment name. This output file contains the mean score that was collected over the course of training the policy. Each of these output files contains the data for one run.

E. Paper version history

Version number	Date	Changes made
1	June 9, 2022	N/A
2	June 15, 2022	Spelling and capitalization fixes.
3	July 16, 2025	Spelling fixes.

F. Acknowledgments

This paper used the Python programming language along with the TensorFlow, Keras, and NumPy packages. Sarah McDougald also deserves a special acknowledgment for helping support this paper.