Combining Deep Q-Learning and Advantage Actor-Critic for Continuous Control

Rae C. Jeong
Department of Mechanical and Mechatronics Engineering – University of Waterloo
raechanjeong@gmail.com

Abstract

We propose a hybrid method of deep Q-Learning and advantage actor-critic (A2C) for continuous control reinforcement learning problems. Advantage actor-critic is an effective algorithm for solving continuous control problems. Deep Q-learning is often used in finite action space and it indirectly improves the policy by learning the Q-values for each state and action pairs. We combine these two ideas by sampling multiple 'action suggestions' from the policy distribution and using the Q function to evaluate and act with the best action. We refer to this new technique as the advantage actor-suggester (A2S), as Q function is the actor that selects the actions and the policy is the suggester that suggests actions. We investigate the challenges of using Q-learning in continuous action space and how the combination with a A2C algorithm can alleviate this challenge. Both on-policy and off-policy formulations are provided for the A2S algorithm. The A2S method is tested on a set of robotics continuous control problems from Roboschool and achieves performance significantly exceeding that of A2C. The supplementary video and the code can be accessed at the following links:https://youtu.be/qhYEET0f1SM
https://github.com/raejeong/RaeboSchool

1 Introduction

Reinforcement learning is an area of machine learning which an agent explores an environment and through the rewards that it receives, learns to improve its behaviour to maximizes the expected sum of rewards over long term. Recently, reinforcement learning has proven to be successful in solving continuous controls tasks, especially for robotics controls problems [1][2][3][4]. For introductory reinforcement learning material we refer to the classic text by Sutton and Barto [5]. In this paper, we consider two families of reinforcement learning algorithms, which are action-value fitting method, deep Q-learning [6], and policy gradient methods, advantage actor-critic [7].

Action-value methods involve learning the state-action values, also known as Q-values. This involves fitting a function that maps from a specific state and a specific action to the expected return of taking that action at that state and following a particular policy. Q-learning [8] and SARSA [9] are two of many algorithms which learns this mapping from state-action pairs to the Q-values. SARSA is an on-policy algorithm in that it learns the Q-values of a specific policy. On the other hand, Q-learning is an off-policy algorithm, in that it learns the Q-values of the optimal policy and is invariant to the policy that is used to collect the experience for learning. Off policy algorithms are useful for situations where the interactions with the environment are expensive, such as in a real robotics task. In this paper, we will focus heavily on sample efficiency of the algorithm, because many ofthe application of a reinforcement learning algorithm for continuous control is for robots which have to interact with the real world.

Policy gradient methods directly optimize the policy by taking a learning step in the direction of the gradient of the increasing performance [10][11]. Actor-critic [12] methods are a type of policy

gradient algorithms which also learns the estimates of the value function for reducing the variance of the vanilla policy gradient algorithm. Typically, the actor-critic methods are on-policy algorithms.

In this paper, we combine these two methods in a hierarchical manner. The stochastic policy optimized by the policy gradient provides a distribution over actions which we can sample from. A Q function can be learnt from interacting with the environment by using Q-learning or standard supervised learning. By sampling multiple action suggestions from the stochastic policy and using the Q function to select the best action from the suggestions allows the A2S algorithm to gain more information from the experience, leading to a more sample efficient algorithm. We investigate the challenges in using Q functions in continuous domain and the implementation details of A2S algorithm. A2S algorithm along with the A2C algorithm are tested on the Roboschool [13], a robotics control task simulator by OpenAI.

1.1 Prior Work

In this paper we combined deep Q-learning and A2C algorithm. Examples of successful combination of deep Q-learning and A2C algorithm are the deep deterministic policy gradient (DDPG) [14] and the PGQL [15]. In a deep Q-learning, one needs to find the *argmax* over actions, but this introduces an optimization problem for continuous action space, because there are infinite actions to evaluate. DDPG is a model-free, off-policy and continuous domain algorithm. DDPG adapts deep Q-learning to continuous domain by learning a deterministic policy that maximizes the Q value. A different perspective of DDPG can be seen when we realize that DDPG algorithm uses a deterministic policy to perform or learn the *argmax* of the Q function. PGQL showed that the Bellman residual is small at the fixed point of a regularized policy gradient algorithm when the regularization penalty is sufficiently small. Deep Q-learning and A2C are combined by adding an auxiliary update to reduce the Bellman residual. Unlike how DDPG and PGQL found an elegant solution by deriving a new policy gradient which embeds Q-learning, or vise versa, we take a more modular and hierarchical approach where the Q-function and the policy are both trained in a traditional manner but differs in selecting the optimal action. Efforts in stabilizing the deep Q-learning algorithm and improving the A2C algorithm, such as *experience replay* [6], *soft target update* [14], and *generalized advantage estimation* (GAE) [16] are also discussed in the context of improving the A2S algorithm.

2 Reinforcement Learning

We consider the episodic, discounted, continuous state and action space Markov decision process, with state space $\mathcal{S}$, action space $\mathcal{A}$ and reward at each time-step denoted by $r_t \in \mathbb{R}$. The policy is stochastic and denoted by $\pi_\theta : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, where $\mathcal{P}(\mathcal{A})$ is the set of probability measures on $\mathcal{A}$ and $\theta \in \mathbb{R}^n$ is a vector of n parameters, and $\pi_\theta(a_t|s_t)$ is the conditional probability density at a_t associated with the policy. The agent in reinforcement learning interacts with the environment over many episodes. Episodes are trajectory of interactions with the environment of each time-step. At each time-step t the agent receives the state s_t and a reward r_t and selects an action a_t with the policy π_t. With the selected action, the agent moves to the next state with the *transition dynamics distribution* $\mathcal{E}$ with conditional density $p(s_{t+1}|s_t, a_t)$ which satisfies the Markov property $p(s_{t+1}|s_1, a_1, ..., s_t, a_t) = p(s_{t+1}|s_t, a_t)$, for any trajectory $s_1, a_1, s_2, a_2, ..., s_T, a_T$ where T denotes the final time-step of the episode. The return r_t^γ is the total discounted reward from time-step t onwards, $r_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$ where the discount factor γ is $0 < \gamma < 1$. Value functions are defined as expected discounted return of being in a particular state s and following policy π, $V^\pi(s) = \mathbb{E}[r_0^\gamma | s_0 = s; \pi]$.

2.1 Q-Learning

Q-learning, learns the Q function which maps the state-action pair to a real scalar number called action-value or Q-value. When the Q function is learnt, the action with the highest Q-value is chosen at every state, which is known as the *greedy policy*. The action-value, or Q-value, of being in a particular state s and taking action a and following policy π thereafter is given as the expected discounted return, $Q^\pi(s, a) = \mathbb{E}[r_0^\gamma | s_0 = s, a_0 = a; \pi]$. It is important to notice that the value of the state is simply the weighted sum of the Q-values weighted by the stochastic policy π in the discrete action setting. This definition can extended to the continuous action space where, $V^\pi(s) =$

$\int_{\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) da$. The Q function with the highest Q values over all the possible policies is called the optimal Q function, denoted as Q^* where $Q^*(s,a) = max_\pi Q^\pi(s,a)$.

One of the fundamental equations in reinforcement learning is the *Bellman equation* [17]. Bellman equation is a recursive condition that is a necessary for optimality in *dynamic programming* [17]. The optimal Q function with the Bellman equation is given as,

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim \mathcal{E}^\pi, a \sim \pi} \Big[r + \gamma \max_{a'} Q^*(s',a') \Big| s, a \Big] \tag{1}$$

For Q-learning [6], the objective is to reduce the *Bellman error* which is the error on whether or not the Q function we have is obeying the Bellman equation. The loss on the Bellman error is given as,

$$L_i(\theta_i) = \mathop{\mathbb{E}}_{s,s' \sim \mathcal{E}^\pi, a \sim \pi} \Big[((r + \gamma max_{a'} Q(s',a',\theta_{i-1})) - Q(s,a,\theta_i))^2 \Big] \tag{2}$$

where *i* is the iteration step and θ_i is the parameters of the Q function at iteration i. Note that this formulation of Q-learning is model-free, and off-policy because it learns about the greedy policy $a = max_a Q(s,a;\theta)$, while following a behaviour policy that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy policy that follows the greedy policy with probability $1-\epsilon$ and selects a random action with probability ϵ.

2.2 Advantage Actor-Critic

Advantage actor-critic algorithms uses the *advantage function*, A, to reduce the variance of the vanilla policy gradient algorithm which is sometimes referred to as the REINFORCE algorithm [18]. Policy gradient directly optimize the parametric policy via gradient ascent on the performance J. The policy gradient theorem [11] provides the gradient of J with respect to the parameters of the policy π,

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{s \sim \mathcal{E}^\pi, a \sim \pi_\theta} \Big[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \Big] \tag{3}$$

Intuitively, the policy gradient will increase the likeliness of a good action being sampled from the stochastic policy. When all the rewards are are positive, the policy gradient algorithm will try to increase the probability of taking all the action proportionally, which increases variance of the distribution. It is shown that an unbiased estimate of the gradient with lower variance can be obtained when a baseline is used. Baseline is a function of states that is subtracted from the Q function, which is a function of both states and actions [18]. Intuitively, this gives us a new quantity that tell us how much better is a certain action at that state compared to the average. This quantity is the *advantage function* and is defined as $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. Again, note that the value of the state is simply the weighted sum of the Q-values weighted by the stochastic policy π in the discrete action setting. This leads us to the A2C algorithm with the gradient update,

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{s \sim \mathcal{E}^\pi, a \sim \pi_\theta} \Big[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s,a) \Big] \tag{4}$$

The algorithm is called actor-critic because the policy which chooses the actions resembles an *actor* and the value function who estimates the action-value function resembles the *critic*. In practice, the Q function is estimated as the return by performing Monet Carlo sampling, which leads to the on-policy A2C algorithm. It is worth mentioning that the A2C algorithm is a simplified version of the popular *asynchronous advantage actor-critic* (A3C) [14] but without the asynchronous part since the performance improvements of the asynchronous interaction with the environments have shown to be small in some cases [13].

3 Q-Learning in Continuous Domain

Adapting the Q-learning algorithm to the continuous domain presents some challenges. First lets consider the naive implementation of the algorithm. Following the deep Q-learning algorithm [6]

architecture, lets consider a Q function which takes in a particular state and outputs all the Q-values of all the discrete actions possible. This architecture for low-dimensional discrete action space works quite well. For example, learning to play Atari video games from pixel where the number of actions are around 18. A naive approach to adapting the deep Q-learning from Atari to continuous domain is to simply discretize the action space. When put to practice, its apparent that the number of actions increases exponentially with the number of degrees of freedom. As mentioned in [14], when we consider a 7 degrees of freedom system like a human arm with coarsest discretization $a_i \in -k, 0, k$ for each joints (without the hand), the action dimensionality leads to $3^7 = 2187$. This problem only gets worse when we need finer control of the system and require a high resolution discretization. Large action spaces are difficult to explore efficiently leading to an intractable algorithm, especially when we consider that a real robotics tasks has a high cost of interacting with the environment.

Policy gradients handle continuous domain by make the policy output the parameters for a certain type of distribution. Most common distribution used is the normal distribution where the policy, often a neural network, outputs mean and the standard deviation for each degrees of freedom. It is also common to have a separate variable for the standard deviation and decrease it over time. Algorithms like DDPG [14] finds a link between policy gradient and Q functions with the deterministic policy gradient theorem [19] to get the benefits of Q-learning in continuous domain problems.

3.1 Choosing the Greedy Action

As stated previously, it is intractable to have a Q function that outputs all the possible actions for a reasonable size continuous domain task. The deep Q-learning algorithm outputs all the Q-values for possible discrete actions. This is beneficial because then the Q function, a neural network, only needs to perform one forward propagation to obtain the Q-values for all the actions. Since the number of possible actions is small for Atari video games, one can simple find the action with the highest Q-value by iterating through the vector of action-values. This operation $max_a Q^*(s, a)$ is performed when following the greedy policy, but also is computed in Equation 1 for $max_{a'} Q^*(s', a')$ for computing the Q-value of the greedy action in the next state. It is important to note that it is possible to compute a Q-value for a specific state-action pair, if the Q-function is architected in a classical way which takes in a state-action pair and outputs a scalar number which represents the Q-value. The problem arises in following the greedy policy which requires us to find the action that has the highest Q-value. To compute the *greedy action* for a sufficiently large discrete action space or a continuous action space would require a optimization operation.

4 Advantage Actor-Suggester

In this paper, we alleviate the optimization problem of adapting deep Q-learning for continuous domain in a modular and hierarchical manner. The idea is to substantially decrease the number of actions to evaluate to compute the greedy action by sampling *action suggestions* from the stochastic policy. By sampling the action suggestions from the policy and evaluating them with a Q function which takes in a state-action pair, the greedy policy can be followed. Therefore, a continuous domain version of deep Q-learning can be derived by combining policy gradient such as, A2C with deep Q-learning. We call this new algorithm *advantage actor-suggester* (A2S) as Q-function which chooses the best action is the *actor* and the policy which suggests different actions becomes the *suggester*. The A2S architecture is illustrated in Figure 1.

In A2S we train 3 separate networks, which are the policy network π, Q network Q, and although not shown in the diagram, the value network V. The policy network is trained using A2C algorithm and the advantage function is estimated by $A^\pi(s, a) \approx r_t^\gamma - V^\pi(s)$, where r_t^γ is the discounted return. The difference lies in training the Q network in either on-policy or off-policy manner and acting in the environment using the Q network. We first present an on-policy variant of the A2S algorithm, which uses supervised learning for Q-learning along with methods for stabilizing this variant of the algorithm. Next, we present an off-policy variant which uses the Bellman error for Q-learning, along with methods for stabilizing the algorithm.

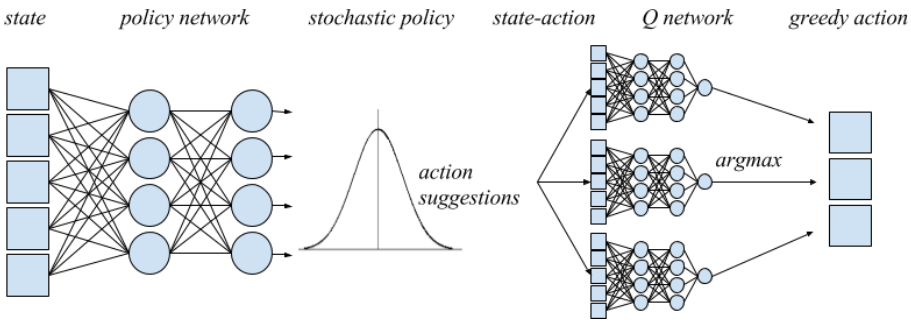

Figure 1: Advantage Actor-Suggester algorithm architecture

4.1 On-Policy Variant

The on-policy variant of the A2S algorithm is based on the idea of estimating the Q function with the episodic return by Monte Carlo sampling from interacting with the environment. Using this estimation in reverse, one can get the on-policy Q function Q^π labels by calculating the discounted return at end of each episode i, denoted as r_i^γ. This is exactly the same as training the on-policy value function using the discounted return in a typical actor-critic policy gradient methods.

After i^{th} episode, the discounted return can be calculated as, $r_t^\gamma = \sum_{k=t}^{T} \gamma^{k-t} r(s_k, a_k)$ and the actions taken can be stored. The discounted returns and the actions taken are used as the labels for supervised learning for the Q network and the value network uses only the discounted return as the label. The only difference between these two networks are the size of the network and the inputs. The Q network takes in the actions as part of the input, and therefore, can make the distinction between value of different actions in the same state while the value function is learning the value of being in a certain state when policy π is followed. With this method of supervised learning of the Q function, the on-policy A2S algorithm is presented.

Algorithm 1: On-Policy Advantage Actor-Suggester

Randomly initialize networks $\pi(a|s;\theta^\pi)$, $Q(s,a;\theta^Q)$, and $V(s;\theta^V)$
for *i = 1, M episodes* **do**
  Initialize the environment with s_0
  Initialize trajectory buffer $\mathcal{T}$
  for *t = 1, T episode time-step* **do**
    Compute the policy distribution from current state s_t, $\pi(a_t|s_t;\theta_t^\pi)$
    for *k = 1, K action suggestions* **do**
      Sample an action suggestion from the current policy distribution
      Evaluate the Q-value for the action suggestion with the current state, $Q(s_t, a_t^k;\theta^Q)$
      Store suggested action if it has the highest Q-value in the current state
    end
    Execute the action with the highest Q-value a_t^*
    Observe the reward r_t and observe the new state s_{t+1}
    Store the transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{T}$
  end
  Compute discounted return at each state-action pair from $\mathcal{T}$
  Update Q network by minimizing, $L = \frac{1}{T}\sum_i \left[(r_i^\gamma - Q^i(s_i,a_i;\theta^Q))^2\right]$
  Update value network by minimizing, $L = \frac{1}{T}\sum_i \left[(r_i^\gamma - V^i(s_i;\theta^V))^2\right]$
  Compute the advantage function, $A_i^\pi(s_i,a_i) = \left[Q_i^\pi(s_i,a_i) - V_i^\pi(s_i)\right]$
  Update the policy with gradient, $\nabla_{\theta^\pi} J(\pi_\theta) = \mathbb{E}_{s\sim\mathcal{E}^\pi, a\sim\pi_\theta}\left[\nabla_{\theta^\pi} \log \pi_\theta(a_i|s_i) A_i^{\pi_\theta}(s_i,a_i)\right]$
end

Since we are not changing the policy gradient algorithm but learning an additional Q network, we can expect the A2S algorithm to perform better than the A2C algorithm, as long as we are given the true Q function. When Q network is only an estimate of the true Q function, what can frequently happen is that the Q network becomes a local estimate of the true Q function which only acts in a locally optimal manner. The characteristic that the Q network can learn the local optimum quickly is actually the main benefit of the A2S algorithm which gives it its sample efficiency, but it must be balanced with exploration and stabilizing methods.

It is observed that reinforcement learning algorithms can make a bad gradient update and substantially drop its performance. A2S algorithm is quite vulnerable to this behaviour because when the policy suddenly changes and its return or performance changes drastically, the input distribution to the Q network changes for A2S. To mitigate this bad gradient updates, there are algorithms that constrains the KL-divergence of policy updates like, *trust region policy optimization* (TRPO) [20] and *proximal policy optimization* (PPO) [13]. There also have been simpler methods that adapt the learning rate to keep the KL-divergence around a desired value [13].

4.1.1 Recovering Performance

While there exists methods of stabilizing reinforcement learning algorithms like the ones mentioned above related to KL-divergence or *target networks* [6] and *soft target updates* [14], we present a different method of stabilizing the network. For sample efficiency, rather than being concerned with monotonically increasing performance, we put the focus in simply learning good behaviours quickly regardless of the sparsity in learning progress. We achieve this by simply recovering the best network so far when the learning progress is substantially worse than the best performance seen so far, we will refer to this technique as *recovery*. It is important to remind ourselves that often the agent needs to perform badly to find better solutions. For example, in walking task, one of the earlier sub-optimal solution is to balance in place, but to learn how to walk, the agent needs to try falling forward. For this reason, we perform the recovery in a stochastic manner where the probability of recovery is higher when the agent performs worse than the best performance it had. The condition for recovery is given as follows,

$$\alpha + 1 - \frac{|r - r^*|}{|r| + |r^*|} < U[0, 1) \tag{5}$$

Where α is a *recovery bias*, a hyper-parameter which controls how frequently we want recovery, r is the undiscounted return, r^* is the highest undiscounted return so far and U is the uniform distribution. Note that this only works for positive reward settings because when the r and r^* have different signs, the third term simply becomes 1. This is not an issue since a constant bias can be added to the rewards without changing the optimal solution and the variance in policy gradient is solved by actor-critic formulation. This condition is not just useful for recovery, but we can also reduce the learning rate, as well as the desired KL-divergence when using the adaptive learning rate to take a smaller gradient step after recovery.

There are few other tricks that we used to improve the A2S algorithm. First, we take multiple gradient steps for value network and Q network to improve sample efficiency. Second, we perform supervised learning on the policy network to output the mean μ for the policy distribution to be closer to what the Q network thinks is correct. Which has the loss function of the form,

$$L = \frac{1}{N} \sum_{n}^{N} \left[(\mu^Q - \mu^\pi)^2 \right] \tag{6}$$

Where N is the current batch size, μ^Q is what the current Q network would have chosen to be good actions with the current policy in the visited states and μ^π is the actions the current policy network outputs.

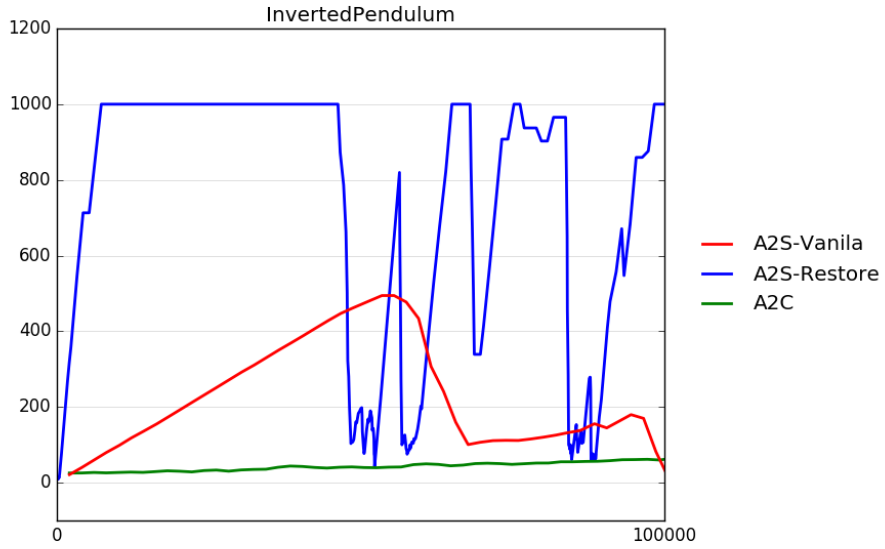

Figure 2: On-policy A2S performance comparison. The x-axis represent the time-steps

To demonstrate the A2S algorithm's behaviour of seeking local optimums and its instability, we show in Figure 2, the comparison of learning curves for vanilla version of A2S, A2S-Restore, which uses the stabilizing methods. The vanilla version of A2S is able to make progress quickly but after it reaches a sub-optimal performance, it quickly drops in performance. The stabilized A2S algorithm can solve the task in about 8K time-steps. However, the sparse nature of the A2S algorithm still remains, as shown in the Figure 2. Both version of A2S significantly outperforms the A2C in this simple classic controls task of inverted pendulum from Roboschool [13]. The performance difference is significant in this environment also because of the simplicity of the task.

4.2 Off-Policy Variant

While the on-policy algorithm is simple and straightforward, it does not have the benefits of the off-policy methods. Here we extend the A2S algorithm to the off-policy variant. First recall that the Equation 1, $Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}^\pi, a \sim \pi}[r + \gamma \, max_{a'} \, Q^*(s', a') | s, a]$, is the off-policy formulation of Q-learning. Since for the A2S algorithm the max operator is over the action suggestions a^s drawn from the stochastic policy, an expectation of the last term must be taken, leading to,

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim \mathcal{E}^\pi, a \sim \pi} \left[r + \gamma \mathop{\mathbb{E}}_{a^s \sim \pi} [\max_{a^s} Q^*(s', \pi(s'))] \middle| s, a \right] \tag{7}$$

$$L_i(\theta_i) = \mathop{\mathbb{E}}_{s, s' \sim \mathcal{E}^\pi, a \sim \pi} \left[((r + \gamma \mathop{\mathbb{E}}_{a^s \sim \pi} [\max_{a^s} Q^*(s', \pi(s'))]) - Q(s, a, \theta_i))^2 \right] \tag{8}$$

While this provides the off-policy formulation of the A2S algorithm, the calculation of the loss function is more complex than the on-policy A2S. The inner expectation is computed through Monte Carlo sampling of iteration N and the suggester, π, provides number of suggested actions K which must be evaluated through the Q function to perform the max operation. Although this grows in complexity quickly of $N \times K$, it is still feasible to compute when the number of Monte Carlo

sampling is low and the number of suggestions is low. With this formulation of the loss function for off-policy Q-learning, we present the off-policy A2S algorithm,

Algorithm 2: Off-Policy Advantage Actor-Suggester

Randomly initialize networks $\pi(a|s;\theta^{\pi})$, $Q(s,a;\theta^{Q})$, and $V(s;\theta^{V})$
for *i = 1, M episodes* **do**
  Initialize the environment with s_0
  Initialize trajectory buffer $\mathcal{T}$
  for *t = 1, T episode time-step* **do**
    Compute the policy distribution from current state s_t, $\pi(a_t|s_t;\theta^{\pi}_t)$
    for *k = 1, K action suggestions* **do**
      Sample an action suggestion from the current policy distribution
      Evaluate the Q-value for the action suggestion with the current state, $Q(s_t, a^k_t;\theta^{Q})$
      Store suggested action if it has the highest Q-value in the current state
    end
    Execute the action with the highest Q-value a^*_t
    Observe the reward r_t and observe the new state s_{t+1}
    Store the transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{T}$
  end
  Compute discounted return at each state-action pair from $\mathcal{T}$
  Set $y_i = r + \gamma \frac{1}{N} \sum_n \left[max_{a^s}\, Q^*(s', \pi(s')) \right]$
  Update Q network by minimizing, $L = \frac{1}{T} \sum_i \left[(y_i - Q^i(s_i, a_i;\theta^{Q}))^2 \right]$
  Update value network by minimizing, $L = \frac{1}{T} \sum_i \left[(r^{\gamma}_i - V^i(s_i;\theta^{V}))^2 \right]$
  Compute the advantage function, $A^{\pi}_i(s_i, a_i) = \left[Q^{\pi}_i(s_i, a_i) - V^{\pi}_i(s_i) \right]$
  Update the policy with gradient, $\nabla_{\theta^{\pi}} J(\pi_{\theta}) = \mathbb{E}_{s\sim\mathcal{E}^{\pi}, a\sim\pi_{\theta}} \left[\nabla_{\theta^{\pi}} \log \pi_{\theta}(a_i|s_i) A^{\pi_{\theta}}_i(s_i, a_i) \right]$
end

4.3 Implementation

With the off-policy variant of the A2S algorithm, there are several well known improvements that can be made. So algorithms like *prioritized experience replay* (PER) [21], *soft target updates* [14] and *double Q-learning* [22] can be used along with the previous mentioned stabilizing methods for on-policy A2S.

The main challenge for the implementation is architecture of the Q network. Since there are multiple action suggestions to evaluate when acting in the environment, it is beneficial to have the Q network accept multiple action suggestions with a single state. This becomes quite challenging in two aspects. First, the network needs to learn the spacial locations of the input actions and the outputs corresponding to the Q-values for each actions. Another problem is that this is inflexible of arbitrary number of actions. But most importantly, this architecture becomes quite cumbersome, when we need to evaluate a specific state-action pair for evaluating the max operation in the loss function or updating the priority of the PER. While this is learn-able by the network, because the sample complexity is the main concern, we have decided to use a naive architecture of the Q network where it only takes in one state and one action.

5 Results

When both versions of A2S were tested, it was evident that the off-policy variant is significantly slower than the on-policy variant while being outperform. It is also worth noting that the off-policy variant was difficult to stabilize. Therefore, we present results with the on-policy variant of the algorithm in 3 different continuous controls tasks along with the A2C algorithm. The 3 environments that the algorithms are tested on are the *Walker2d, HalfCheetah*, and *Hopper* from OpenAI Roboschool [13]. The results in Figure 4 are obtained by running each algorithm 3 times with different random seeds in each environment. All the runs are plotted in the background unfiltered in a

light blue and light green color. The blue and green color lines are the average of the 3 runs with additional moving average filter with window size of 3. A2S algorithm is the on-policy variant with the previously mentioned stabilizing methods.

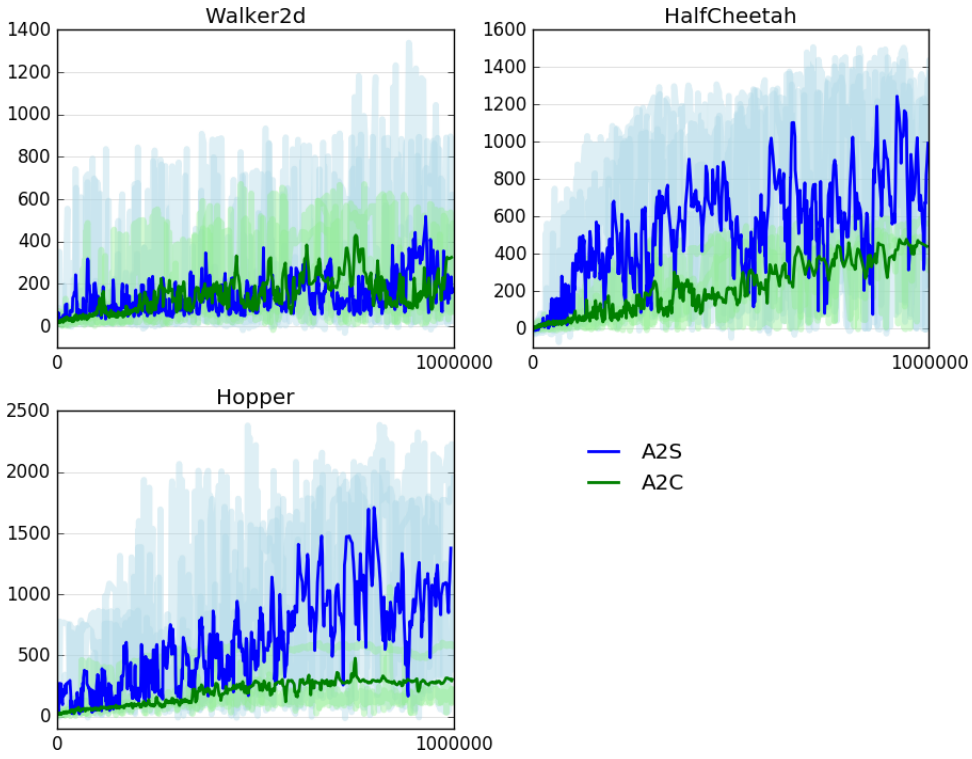

Figure 3: Learning curves for continuous control problems.

Table 1: Description of the environments [14]

Environment	**Brief description**
Walker2d	Agent should move forward as quickly as possible with a bipedal walker constrained to the plane without falling down or pitching the torso too far forward or backward.
HalfCheetah	The agent should move forward as quickly as possible with a cheetahlike body that is constrained to the plane.
Hopper	The agent should move forward as quickly as possible with a multiple degree of freedom monoped and keep it from falling.

Note that the A2S algorithm has a noisy learning curve when compared to the A2C algorithm. This characteristic can be seen clearly with the results from Walker2d environment. Because the learning curves are very noisy, when filtered, the performance seems mediocre but when we take a look at the raw performance curves in light colors or the their best scores in Table 2, it is clear that the A2S algorithm performs significantly better than the A2C algorithm. It is however worth mentioning that the A2S algorithm in its current state is significantly slower than A2C algorithm. This is mainly to do with the fact that the Q-learning architecture only can process state-action pair and must perform forward passes of the number of suggested actions. Also to provide some algorithm details, the A2S and A2C algorithms both did not use any target networks. Both the value network and policy network use a fully-connected MLP with two hidden layers of 64 units, and tanh nonlinearities with layer normalization for both A2S and A2C. The policy network outputs the mean of the action

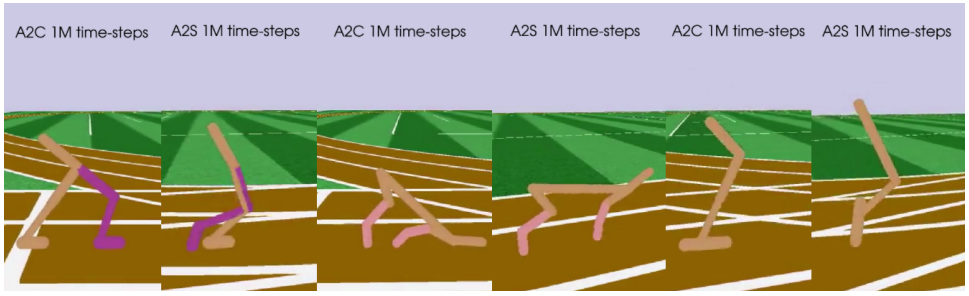

Figure 4: Screenshots of the environments.

Table 2: Best scores for A2S and A2C after 1M time-steps

Environment	**A2S**	**A2C**
Walker2d	1338	671
HalfCheetah	1478	695
Hopper	2381	1059

distribution and has a fixed standard deviation. The Q network is exactly the same architecture but has a hidden layer of 128 units followed by another hidden layer of 64 units. In Table 3, we list other important hyper-parameters that were used for both A2S and the A2C algorithms.

Table 3: Hyper-parameters for A2S and A2C

Hyper-parameter	**A2S**	**A2C**
Discount factor γ	0.97	0.97
Desired KL-divergence	1e-3	1e-3
Policy standard deviation	0.5	0.5
Recovery bias α	0.65	N/A
Number of Suggestions	10	N/A

6 Future Work

The work presented in this paper are quite preliminary and has lots to improve. First, most of the stabilizing methods used for the A2S algorithm are compatible with the A2C algorithm. When the A2C algorithm was briefly tested with some of these techniques, there were noticeable improvements. Although, this does requires significant resources in searching for the combination of techniques and its hyper-parameters that actually helps the A2C algorithm, since it is unclear whether or not all of the techniques would be beneficial for the A2C algorithm. The algorithm can be made more flexible by making the recovery condition invariant to the magnitude and signs of the reward by normalizing the return in a heuristic manner since the maximum return is not known. There exists many other reinforcement algorithms and techniques that are compatible to A2S. Since A2S is a modular and a hierarchical approach in combining deep Q-learning and A2C, most of the policy gradient algorithms should be compatible, such as A3C [14] and PPO [13]. Methods that improve policy gradients also can be adopted to A2S such as the GAE [16] and *soft target update* [14]. As it can be

seen the A2S algorithm is compatible with many other algorithms and the on-policy variant is very simple to implement.

The main obstacle in making the off-policy version of A2S feasible is the computational complexity of the Monte Carlo estimate of the maximum Q-value in the next state. This is mainly attributed to the fact that the Q network only takes in state-action pairs, rather than evaluating all the action suggestions in a single forward pass. Making the Q network to evaluate all the action suggestions with a different architecture of *Multi-Action Q Function*, the off-policy version of A2S can become computationally feasible. This will also significantly improving the run time for the on-policy variant of A2S. Further hyper-parameter search could improve the A2S algorithm since due to the lack of available computational resources, and the slow run-time of the A2S algorithm, an exhaustive hyper-parameters was not performed.

7 Conclusion

This paper introduced a new method called *advantage actor-suggester* for both on-policy and off-policy settings which successfully combines the *deep Q-learning* and *advantage actor-critic* algorithms. We have also demonstrated that this algorithm can learn to perform complex robotics tasks with superior sample efficiency when compared to the advantage actor-critic algorithm. Methods for stabilizing and improving the advantage actor-suggester algorithm which are compatible to many other algorithms have been presented. Advantage actor-suggester algorithm takes a modular and hierarchical approach to combine deep Q-learning and advantage actor-critic algorithms, making it compatible with many other policy gradient reinforcement learning algorithms.

References

[1] H. J. Kim, Michael I. Jordan, Shankar Sastry, and Andrew Y. Ng. Autonomous helicopter flight via reinforcement learning. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 799–806. MIT Press, 2004.

[2] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015.

[3] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *CoRR*, abs/1611.04201, 2016.

[4] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *CoRR*, abs/1603.02199, 2016.

[5] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[7] 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. *CoRR*, abs/1602.01783, 2016.

[8] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[9] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. CUED/F-INFENG/TR 166, Cambridge University Engineering Department, September 1994.

[10] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. In *ICML*, Beijing, China, June 2014.

[11] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[12] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4):1143–1166, April 2003.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[14] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[15] Brendan O'Donoghue, Rémi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. PGQ: combining policy gradient and q-learning. *CoRR*, abs/1611.01626, 2016.

[16] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

[17] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[18] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.

[19] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[20] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[21] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[22] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.