Research Article

Reducing Entropy Overestimation in Soft Actor Critic Using Dual Policy Network

Hamid Ali, Hammad Majeed, Imran Usman, and Khaled A. Almejalli

1National University of Computer and Emerging Sciences, Pakistan
2College of Computing and Informatics, Saudi Electronic University, Riyadh, Saudi Arabia

Correspondence should be addressed to Hammad Majeed; research.submission@gmail.com

Received 18 March 2021; Revised 11 May 2021; Accepted 29 May 2021; Published 10 June 2021

Abstract

In reinforcement learning (RL), an agent learns an environment through hit and trial. This behavior allows the agent to learn in complex and difficult environments. In RL, the agent normally learns the given environment by exploring or exploiting. Most of the algorithms suffer from under exploration in the latter stage of the episodes. Recently, an off-policy algorithm called soft actor critic (SAC) is proposed that overcomes this problem by maximizing entropy as it learns the environment. In SAC, the agent tries to be as random as possible while moving towards the maximum reward. This randomness allows the agent to explore the environment and stops it from getting stuck into local optima. We believe that maximizing the entropy causes the overestimation of entropy term which results in slow policy learning. This is because of the drastic change in action distribution whenever agent revisits the similar states. To overcome this problem, we propose a dual policy optimization framework, in which two independent policies are trained. Both the policies try to maximize entropy by choosing actions against the minimum entropy to reduce the overestimation. The use of two policies result in better and faster convergence. We demonstrate our approach on different well known continuous control simulated environments. Results show that our proposed technique achieves better results against state of the art SAC algorithm and learns better policies.

1. Introduction

The agent with the ability of decision-making learns by interacting with the environment. First, it interacts and then explores the environment to learn different behaviors. This process is slow and often takes a lot of time to fully understand the dynamics of the environment. Based on an agent’s ability, the learning time may increase with the increase in difficulty level of the environment.

Reinforcement learning (RL) is a branch of machine learning in which an agent’s goal is to learn a policy that enables it to take decisions while maximizing expected discounted sum of rewards. RL combined with deep learning enables RL agent to learn difficult task such as playing a game of Go [1], learning to play games from raw pixels [2] (playing Atari games), or performing continuous control tasks [3] (controlling humanoids in simulations). Use of neural networks as function approximates has removed the need of manual feature engineering, where policies are directly optimized from raw pixels or sensors output. However, use of neural networks poses many challenges. Two major challenges are that (1) they need high number of samples for learning because neural networks are slow learners. This makes NN infeasible to use in real world problems. (2) Their performance is very sensitive to hyperparameters and require significant amount of time to search for good hyperparameters. Bad choice of parameters can degrade the performance of algorithm and can cause unstable learning. Because of these problems, RL methods cannot be successfully applied to many real-life problems. Our focus in this paper is on continuous control problems and how we can guide our agent solving these problems to train as fast as possible.

One of the major reasons for high sampling complexity is the use of on-policy learning. In on-policy, the policy interacting with the environment is the same as that is being
trained. This means new samples are required for every training step. Recently proposed methods such as TRPO [4] and PPO [5] allow multiple gradient updates on same samples and require less samples as compared to other on-policy methods. But still, they are very data hungry. On the other hand, off-policy methods are less data hungry and can reuse past experiences. Sarsa max [6] and DQN [7] are good examples of off-policy learning. Off-policy learning combined with function approximations such as neural networks are unstable and often diverge. This creates further challenges when action space is continuous. A classical example of this architecture is Deep Deterministic Policy Gradient (DDPG) [8]. In case of high-dimensional states and actions, DDPG suffers and gets stuck into local optima. It creates high peak for some actions and then stops the exploration. To overcome this problem, soft actor critics (SAC) [9, 10] is introduced which adds an entropy term to the objective function. In SAC, the goal is to maximize expected rewards as well as the entropy term. This entropy causes uncertainty and stops agent from creating high peaks for some actions. This results in better exploration and fast learning. In this paper, we have studied the effects of maximizing and minimizing the entropy on the learning process. We have found that maximizing entropy causes an overestimation of the entropy term, which results in unstable learning and slow convergence. We propose that by using dual policy, the effects of overestimation can be reduced which can result in stable, better, and faster learning. Furthermore, it helps in achieving better sampling efficiency.

Exploration means looking for new knowledge, and exploitation means tweaking existing knowledge in search for optimal policy. For better learning, it is very important for an agent to perform both to understand how environment works. Most RL algorithms suffer because they are unable to explore the environment in a structured way. This problem further increases when we work with continuous action space [11], where there is a high chance of agent getting stuck in local optima. In value-based methods, exploration can be achieved through explicit interaction with the environment. Best way to explore environment in value-based methods is through GILE [7] algorithm. At the start, it completely explores the environment and then starts the exploitation.

In policy-based methods, exploration is achieved through adding entropy to the policy [2, 4]. This approach fails when we have a large action space [7]. SAC [9, 10] also works with the same idea but it adds entropy to the objective function and maximizes it during the learning process. This helps the agent to explore the environment especially in case to continuous and large action space. As the agent learns, the entropy term is optimized which causes better exploration. Our focus in this paper is on the effects of entropy on the learning process in continuous action space. Our aim is to improve entropy estimate so that policy can be guided towards the optimal path as fast as possible.

Most existing RL algorithms work with deterministic policies. This is because deterministic policies are easy to optimize and result in stable convergence [12–14]. However deterministic policies are not good for exploration [15]. For exploration, stochastic policies are used which return probability distribution of actions. Normally, exploration in stochastic policy is achieved through heuristics. By heuristics, we mean adding entropy to the stochastic policy or adding random noise. Stochastic policy helps the agent to explore environment in a structured way. Because of stochastic behavior, the agent can better generalize against the unseen states. In SAC, entropy is added to the objective function, where it is maximized along with the expected reward. We have seen that adding large entropy causes instability or slow convergence.

Actor critic methods are at the intersection of value-based methods such as DQN [7] and policy based such as reinforce [16]. Actor acts as a policy whose job is to output an optimal action. Critic’s job is to evaluate actor’s output using current state and action. Figure 1 shows a general actor critic framework. Actor is optimized using policy gradient method, and critic is optimized using value-based methods. Actor output is used to train critic, and then, critic is used to guide policy towards the optimal path.

Actor critic algorithms start learning from policy iteration. Policy iteration alternates between policy evaluation which means iteratively computing the value function for a policy and policy improvement which uses the value function output to iteratively obtain an optimal policy. For complex RL problems, it is not feasible to run both policy evaluation and policy improvement for infinite number of steps. Therefore, both are optimized concurrently. Basic actor critic framework uses entropy to encourage policy for exploration. Main difference between actor critic and SAC is that SAC adds entropy to the objective function (1).

\[
\pi^* = \arg\max_{\pi} \sum_r \mathbb{E}_{(s, a) \sim \pi}(r(s, a) + \alpha H(s))
\]

SAC maximizes entropy to encourage exploration. This maximization results in the overestimation of entropy. It is observed that the maximizing entropy can cause instability, especially in the early stages of learning when critic is not trained to differentiate between good and bad states. Normally, alpha temperature is used to reduce the effects of entropy term. Recently proposed SAC [10] uses entropy to optimally derive the value of alpha. That means alpha itself is dependent on entropy term. Maximizing entropy results in high alpha which means further adding uncertainty. To reduce this overestimation bias, we propose to train two different policies. During training iterations, we select the policy which gives us low entropy. Here, objective function is not changed, which is maximizing the entropy. To reduce the overestimation bias, we are choosing the policy which minimizes the entropy. This way, we are exploring the environment in structured way while making sure we are not overestimating the benefits of exploration against returns.

We have tested our proposed approach on different continuous control simulated environments created by MuJoCo [17]. Results show that our approach achieves good sampling efficiency as compared to state of the art RL algorithms. Our approach is also easy to implement and can result in better and faster learning.
2. Literature Review

In this section, we will first discuss some of the value-based methods which suffer from overestimation of action values and then different techniques to solve this problem. Then, we will discuss some policy gradient methods which work with continuous action spaces. Thirdly, we are going to discuss some state of the art methods which work with continuous actions. Lastly, we shall discuss the reasons of better performance of SAC and how SAC suffers from overestimation of entropy term.

2.1. Value-Based Methods

2.1.1. Deep Q Learning. Deep Q-Network (DQN) [4] combines neural networks with Q learning, where images (continuous state spaces) are passed as input to deep neural network. NN represents an action value function where last layer of NN corresponds to number of actions agent can take, and optimal policy is derived once optimal value function is found. Because DQN outputs the optimal action, so it cannot be used for continuous actions. DQN provides high sampling efficiency using experience replay. Experience replay is used to break the correlation between consecutive tuples that agent sees while encountering with environment. From deep learning, we have seen that NN overfits if same data is passed in same order again and again. Also, we do not want our agent to learn a sequence of state action pairs; rather, we want agent to learn a best policy. By randomly picking, data from experience replay helps us to break this correlation between sequence to state action pairs. SAC uses the same experience replay to achieve high sampling efficiency.

Q learning uses one step bootstrapping, and at every time step, it tries to minimize mean-squared error between the current state and next state.

\[ L_t(\theta) = E_{(s,a,r,s') \sim U(D)} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_t') - Q(s, a; \theta_t) \right]^2. \]  

(2)

Values of NN are evolved at each time step \( t \) by using (2). It creates a problem while converging because we are chasing a moving target. At every time step, when we move a head, our target moves too. This results in nonconvergence of NN. To overcome this problem, DQN uses fixed Q target or target network where a separate NN is used to find next state values. Values of policy are replaced with target network after 1000 number of interactions.

Q learning uses greedy policy to learn the environment which means to calculate the value of next state, it uses max value action in the next state using (2). This maximization results in overestimation of action values and creates high variance, especially when we are at the early stage of learning because Q values evolve very fast. Following approaches were proposed to reduce this overestimation.

2.1.2. Double Q Learning. First solution to address overestimation of action values was proposed in [18]. This paper proposed to use two different Q value estimators where action was chosen from both the estimators. Then, with equal probability, Q estimators are updated. At each time step, only one Q value estimator is updated. By using two different estimators, overestimation was reduced which resulted in reducing variance in Q values.

2.1.3. Double Deep Q Learning. Another solution to address the problem of overestimation of action values was proposed in double Q learning [19]. To overcome the problem of moving target values, a separate network was maintained which is known as target network in DQN [7]. In double DQN, the authors proposed the idea that instead of maintaining two separate estimators like in [18], they selected action using model network and got its Q value using target network. From (3), it is clear that model network was used to select action with maximum value but to obtain final Q value target network was used. In (3), \( \theta_t \) represents model weights, and \( \theta_t' \) represents target network weights.

\[ Y_t^{\text{Double}} = R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a; \theta_t'). \]  

(3)

2.2. Policy Gradient Methods

Now, we are going to discuss policy gradient methods and how they use Monte Carlo estimate to update the policy. Policy gradient methods are on-policy methods. Gradient ascent policy tries to maximize expected return while directly searching for optimal policy. Policy gradient methods work with trajectories where trajectory is just an episode with a fixed number of interaction with the environment. Policy gradient methods calculate Monte Carlo estimate over trajectories.

2.2.1. Reinforce. Reinforce [13] is a policy gradient algorithm where gradient of expected return is only calculated with respect to current policy parameters. Gradient of a trajectory is calculated with respect to policy parameters. For this, trajectory must be representative of the current policy. This makes reinforce very data hungry because it learns once from the collected data. Reinforce also suffers from credit assignment problem because Monte Carlo calculates the sum of
rewards at each time step. Here, Markov assumption is used which says to predict the future, you only need present.

2.2.2. Trust Region Policy Optimization. Trust Region Policy Optimization (TRPO) [4] uses the concept of Important Sampling (IS) for making the RL less data hungry. TRPO uses trajectories of the old policy to update current policy. This is achieved through a surrogate function (5). Surrogate function is used to compute the gradient of old policy. Furthermore, it enables us to use same batch of data multiple times to update policy. Here, \( \pi_\theta \), is the current policy which is updated through old policy \( \pi_{\theta_{\text{old}}} \). \( \hat{A}_t \) is an advantage function.

\[
\text{maximize}_{\theta} \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right], \quad (4)
\]

subject to \( \mathbb{E}_t \left[ \text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \leq \delta. \quad (5) \)

In theory, IS helps us to use old trajectories as many times as we want. But in practice, TRPO suffers from high variance and often diverge to unrecoverable path. Here, trust region methods are proposed to reduce variance. TRPO establishes a trust region using KL divergence (6) to prevent current policy from deviating too far from old policy. In (6), \( \beta \) is some coefficient which is used to enforce hard constraint.

\[
\text{maximize}_{\theta} \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]. \quad (6)
\]

TRPO can work with both discrete and continuous action spaces. TRPO adds entropy in policy for exploration which is hindered by the surrogate function. Recall that surrogate function stops the agent from going too far from the old policy. This means new policy cannot explore environment independently.

2.2.2.3. Proximal Policy Optimization. Proximal policy optimization (PPO) [5] is improved version of TRPO. In PPO, instead of establishing trust region, a policy clipping method (7) is used, clipping a policy between minimum of \( 1 - \epsilon, 1 + \epsilon \). This makes sure that new policy does not diverge more than \( \epsilon \) distance away from old policy. PPO clipping function is very important because it reduces variance and makes sure that old and new policies are not too far away from each other.

\[
L_{\text{CLIP}}(\theta) = \mathbb{E}_t \left[ \min \left( w_t(\theta)\hat{A}_t, \text{clip}(w_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t \right) \right]. \quad (7)
\]

\( w_t(\theta) \) will be greater than one if the particular action is more probable for the current policy than it is for the old policy. It will be between 0 and 1 when the action is less probable for our current policy. \( \hat{A}_t \) is an advantage function.

PPO initially allows policy to explore but with time, as the number of interactions increases, it gets stuck in local optima because policy is clipped if it moves too far away from old policy. Due to high sampling cost, it is not feasible to apply PPO on real world problems.

2.2.4. A3C and A2C. Both DQN and reinforce either suffer from variance or biasness. To create a balance between both estimates Asynchronous Advantage Actor Critic (A3C) [2] was proposed which uses \( N \) step bootstrapping. Actor learns advantage function and critic learns value function. Instead of using experience replay to break correlation between consecutive tuples, A3C creates multiple agents of environment at the same time. Every agent interacts with environment for \( n \) number of times to collect experiences. At any time step, the agent will receive multiple minibatches of correlated experiences. Experience will be decor-related because agent will see multiple states and actions at the same time. Asynchronous here means that every agent will have its own copy of environment, and it can be different from what other agents have. Different approach called synchronous actor critic (A2C) [3] uses same copy of environment for every agent. In A2C, all agents wait to complete segment of interaction, then network is updated once for all agents. A2C gives equal or better results than A3C.

A3C and A2C are faster as compared to TRPO and PPO because they create multiple copies of the same environment and can learn parallel. This is useful in case of simulated environments but cannot be applied to real world. Also, creating multiple copies and training them in parallel only helps with a faster learning. It does not help agent in terms of sampling efficiency.

2.3. Continuous Action Space Methods. Now, we are going to look some of the actor critic methods that can only work with continuous action spaces. These methods achieve good sampling efficiency as compared to on-policy methods because they uses off-policy updates. Also, these methods can work with large action spaces where on-policy methods suffer from curse of dimensionality of action space.

2.3.1. Deep Deterministic Policy Gradients (DDPG). DDPG was proposed in [8]. DDPG is a combination of both value- and policy-based methods which searches for deterministic policy and works with only continuous actions. DDPG has four neural networks, a deterministic policy network \( \theta^\mu \), Q network \( \theta^Q \), a target policy network \( \theta^{\mu'} \), and target Q network \( \theta^{Q'} \). DDPG directly maps states into action spaces. Target networks are the delayed copies of original networks to increase stability. Like DQN [7], DDPG also uses experience replay with off-policy updates to achieve high sample efficiency. In DDPG, the value network loss is calculated using Bellman equation (8). Then, value networks are updated using the mean-squared error between current and next Q values. SAC also use the same Bellman equation to update value networks. The only difference is that SAC adds entropy with next Q value to stop Q networks from creating high peaks for some actions.

\[
y_i = r_i + \gamma Q'(s_{i+1}, \mu' (s_{i+1} | \theta^{\mu'})) \left( \theta^{Q'} \right). \quad (8)
\]
To update policy, DDPG takes the derivative of expected return with respect to policy parameters \( \theta \). Then, it takes mean of gradients against minibatch that is sampled from experience replay to calculate policy loss.

\[
J(\theta) = \mathbb{E} \left[ Q(s, a) \mid s_{\pi}, a_{\pi}=\mu(s) \right]. \tag{9}
\]

To encourage policy for exploration, DDPG adds time-correlated noise [8]. But recent research says that uncorrelated, mean-zero Gaussian noise also works. In SAC, exploration is achieved by adding entropy with policy (13).

\[
\mu'(s_t) = \mu(s_t) + \mathcal{N}. \tag{10}
\]

Instead of replacing target networks after a fixed number of times, DDPG uses soft target updates in which target network weights are slowly updated by original networks weights. SAC also use soft target updates to update target networks.

2.3.2. Twin-Delayed DDPG (TD3). In [20], a new algorithm called Twin-Delayed Deep Deterministic policy gradients (TD3) was proposed. In TD3, old idea of double Q learning [18] is used. Twin in TD3 means that we are using two networks for critic. TD3 uses clipped Q learning where the minimum value of critic is used. This results in the underestimation of Q values thus better and stable learning curve. Our idea of reducing overestimation from entropy is inspired from TD3. SAC also uses the same approach to reduce overestimation in action values. Like TD3, we have decided to use two different policies to reduce overestimation in entropy term.

To encourage exploration, TD3 uses target policy smoothing. In DDPG during training, high peaks are created for some actions. These high peaks push policy towards picking action and stop the policy from exploration. This can be minimized by adding random noise (\( \varepsilon \)) 11 which act as a regularization. Noise is clipped to make sure it lies with in valid action range.

\[
a'(s') = \text{clip} \left( \mu_{\text{out}} \left( s' \right) \right. + \text{clip} (\varepsilon, -c, c), a_{\text{Low}}, a_{\text{High}} \right), \quad \varepsilon \sim \mathcal{N}(0, \sigma). \tag{11}
\]

3. Soft Actor Critic (SAC)

Generally, in RL, the goal is to learn a deterministic or stochastic policy to maximize the expected sum of rewards by using (12).

\[
\sum_t E_{(s, a) \sim \rho_n} [r(s, a)]. \tag{12}
\]

Standard RL algorithms trap in local optima due to reliance on only the reward (overestimation). Therefore, whenever a high peak in performance is achieved in some trajectory, it would be almost impossible to recover from there. Where as in SAC, the goal is to learn a stochastic policy that maximizes both the expected sum of rewards and entropy by using (13).

\[
J(\pi) = \sum_t E_{(s, a) \sim \rho_n} [r(s, a) + \alpha \mathcal{H}(\pi(\cdot|s_t))], \tag{13}
\]

where \( \rho_n \) is the state marginals of the trajectory induced by a policy \( \pi(\cdot|s_t) \), \( \alpha \) is the temperature, and the rest of the terms have conventional meanings. \( \alpha \) determines the relative importance of the reward over entropy. This effectively controls the stochasticity of the optimal policy. By setting \( \alpha = 0 \), SAC can be transformed into a standard RL that only maximizes the sum of rewards. Above-mentioned objective term (13) has many advantages. For example, it greatly rewards the policy with the ability of high exploration and rejecting the unpromising areas of the search space. Furthermore, it gives equal opportunity of selection to nearly equal good actions. This resolves the problem of overestimation in conventional RL to some extent. SAC can be extended to apply on infinite horizon problems by introducing a discount factor \( \gamma \). This ensures that the sum of the expected rewards and entropy is finite. Figure 1 shows the graphical representation of SAC framework. Two versions of SAC are proposed. They are discussed in detail in the following sections.
1: Set initial policy parameters \( \phi_1, \phi_2 \), Q-function parameters \( \theta_1, \theta_2 \), empty replay buffer \( \mathcal{D} \), \( \gamma \) discount reward
2: Set target parameters equal to main parameters \( \theta_{\text{target}} \leftarrow \theta_1, \theta_{\text{target}} \leftarrow \theta_2 
3: \textbf{repeat}
4: \text{[Observe state } s_t \text{, and select action against minimum entropy]}
5: \( a_k \sim \pi_{\phi_k} (\cdot | s_t) \) for \( k = 1, 2 \)
6: \( e_m, m \leftarrow \min_{k=1,2} \log \pi_{\phi_k} (a_k | s_t) \) \{m is the index of the policy with minimum entropy\}
7: Execute \( a_k \) in the environment
8: Observe next state \( s' \), reward \( r \), and done signal \( d \) to indicate whether \( s' \) is terminal
9: Store \((s, a_k, r, s', d)\) in replay buffer \( \mathcal{D} \)
10: \textbf{if} \( s' \) is terminal
11: \textbf{reset environment state}
12: \textbf{end if}
13: \textbf{if} it is time to update \textbf{then}
14: \textbf{for} \( \text{in range(no. of updates required)} \) \textbf{do}
15: Randomly sample a batch of transitions, \( B = (s, a, r, s', d) \) from \( \mathcal{D} \)
16: Compute targets for the Q functions:
17: \( \hat{a}^k, m \sim \pi_{\phi_k} (\cdot | s) \) for \( k = 1, 2 \)
18: \( e_m, m \leftarrow \min_{k=1,2} \log \pi_{\phi_k} (\hat{a}^k | s) \) \{m is the index of the policy with minimum entropy\}
19: \( y(r, s', d) = r + \gamma \min_{k=1,2} Q_{\theta_{\text{target}}} (s', \hat{a}^k_m) - ae_m \)
20: Update Q-functions by one step of gradient descent using
21: \( \mathbb{E}_B \left[ 1/|B| \sum_{(s, a, r, s', d) \in B} Q_{\theta_k} (s, a) - y(r, s', d) \right]^2 \) for \( i = 1, 2 \)
22: Update policy by one step of gradient ascent using
23: \( \hat{a}_k \sim \pi_{\phi_k} (\cdot | s) \) for \( k = 1, 2 \)
24: \( e_m, m \leftarrow \min_{k=1,2} \log \pi_{\phi_k} (\hat{a}_k | s) \) \{m is the index of the policy with minimum entropy\}
25: \( \mathbb{E}_B \left[ 1/|B| \sum_{(s, a) \in B} \min_{k=1,2} Q_{\theta_k} (s, \hat{a}_k_m) - ae_m \right] \)
26: \{where \( \hat{a}_m \) is a sample from \( \pi_{\phi_m} \) which is differentiable w.r.t. \( \phi_m \) via the reparametrization trick.\}
27: Update target networks with:
28: \( \theta_{\text{target}} \leftarrow \rho \theta_{\text{target}} + (1 - \rho) \theta_i \) for \( i = 1, 2 \)
29: \{where \( \rho \) is polyak. (Always between 0 and 1, usually close to 1.)\}
30: \textbf{end for}
31: \textbf{end if}
32: \textbf{until} convergence

\textbf{Algorithm 1: Proposed Soft Actor Critic Algorithm.}

3.1. \textit{Soft Actor Critic V1.} SAC was first proposed in [9]. In the first version, SAC uses fixed \( \alpha \). As discussed earlier, \( \alpha \) is used to vary the importance of reward over entropy and vice versa in the objective function. First version of SAC comprises of a parameterized state value function \( V_{\phi_\psi}(s_t) \), soft Q-function \( Q_{\theta}(s_t, a_t) \), and a tractable policy \( \pi_{\phi_\psi}(a_t | s_t) \). These functions are implemented as networks and use \( \psi, \theta, \phi \) as parameters. A policy function parameterized with \( \phi \), a value function parameterized with \( \psi \), and a Q function parameterized with \( \theta \). Q and value functions are related and help with stable convergence.

Value function is optimized using the objective function given in (14). It calculates the mean-squared error between the value network prediction and the prediction of the expected Q function plus the entropy term from the policy.

\[
J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\psi} \left[ \frac{1}{2} (V_\psi(s_t) - V_{a_t \sim \pi_\psi} [Q_{\theta}(s_t, a_t) - \log \pi_\psi(a_t | s_t)])^2 \right].
\]  

(14)

where \( \mathcal{D} \) is a replay buffer.

Q network is optimized by minimizing the error expression of (15).

\[
J_Q(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\psi} \left[ \frac{1}{2} (Q_\theta(s_t, a_t) - Q_\psi(s_t, a_t))^2 \right].
\]  

(15)

where

\[
\tilde{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim P} [V_\psi(s_{t+1})].
\]  

(16)

During this optimization, for all the (state, action) pairs in the replay buffer, we want to minimize the squared difference between the prediction of our Q function and the immediate (one time-step) reward plus the discounted expected value of the next state. \( V_\psi \) is the target value function which is updated after updating all the three networks.

Stochastic policy network outputs mean and standard deviation across all the actions. Actions are then sampled from the normal distribution which is created by using the mean and standard deviation generated by the policy. In
reparameterization trick is introduced to ensure that policy function remains differentiable while applying backpropagation. The new policy is defined as (17).

\[ a_t = f_\phi (\epsilon_t; s_t), \quad (17) \]

where \( \epsilon \) is a noise vector sampled from a Gaussian distribution \( \mathcal{N} \). By using this new objective function, policy network can be optimized by using (18).

\[ J_\pi \phi (\epsilon_t; s_t) = \mathbb{E}_{s_t \sim D, \epsilon_t \sim N} \log \pi_\phi \left( f_\phi (\epsilon_t; s_t) | s_t \right) - Q_\theta (s_t, f_\phi (\epsilon_t; s_t)). \quad (18) \]

3.2. Soft Actor Critic V2. The second version of SAC are proposed in [10]. Two things are changed in version 2. Value network is removed, and temperature value is automatically determined. The policy network is same as it was in version 1. According to [10],

"Simply forcing the entropy to a fixed value is a poor solution, since the policy should be free to explore more in regions where the optimal action is uncertain, but remain more deterministic in states with a clear distinction between good and bad actions."

In SAC V2, \( \alpha \) is optimized by using the objective function (19).

\[ J_\alpha = \mathbb{E} \left[ -\alpha \log \pi_\alpha (a_t | s_t) - \alpha \bar{H} \right], \quad (19) \]

where \( \bar{H} \) is the desired entropy. In our research, we have found out that scaling down entropy with alpha still results in the overestimation of action values. It is because alpha parameter is derived from entropy term.

4. Proposed Methodology

In this section, we will discuss our proposed approach in detail. As discussed earlier, RL algorithms suffer from overestimation and the best way to address it by using two different neural networks [20]. Our technique of reducing overestimation in entropy is based on SAC and inspired from this technique. We propose to use two different policies \( \pi_{\phi_1} \) and \( \pi_{\phi_2} \) parameterized by \( \phi_1 \) and \( \phi_2 \). Both the policies are initialized randomly. As entropy is determined from the policy itself therefore, the use of different policies enable us to estimate different entropy values. Figure 2 shows work flow of our proposed soft actor critic. For example, let us assume state
from the environment is fed to our agent. Both the policies \( \pi_{\phi_1} \) and \( \pi_{\phi_2} \) get the same state and output mean and standard deviation across all the actions used. Actions \( a_1 \) and \( a_2 \) are sampled from the normal distributions that are constructed from the means and standard deviations returned by the policies (20). Then, entropy is calculated against both the policies. The final action is selected against the minimum entropy using (21).

\[
\begin{align*}
    a_k &\sim \pi_{\phi_k}(\cdot|s_t) \text{ for } k = 1, 2, \quad (20) \\
    \min_{k=1,2} \log_{\pi_{\phi_k}}(a_k|s_t). &\quad (21)
\end{align*}
\]

Note that in our proposed approach, we are not changing the objective function which is to maximize entropy along with expected reward. Entropy is added to encourage exploration so that agent does not get stuck in local optima by creating high peaks for some actions. By using two policies, we are encouraging our agent to maximize entropy where the action distribution is relatively normal. Entropy makes sure that it does not create high peaks by repeatedly selecting the same action. We are optimizing the policy against low entropy which helps agent to learn the optimal action in case of unsure situations. Optimizing different policies at the same time enables exploration which was previously dependent on only the entropy. It also helps agent to capture different versions of optimal policy and stops it from premature convergence to local optima.

In [9], it is shown that alpha is a sensitive hyperparameter. Wrong alpha can drastically degrade the performance of our algorithm. Also, reward signal not only varies across different tasks but also changes within the same task as policy learns more about the environment. As the agent learns, entropy also changes with policy. Fixing policy to a static entropy value is a bad idea because it restricts the agent to explore the environment. Especially in regions where it is unclear about the optimal action. Complete algorithm of SAC with dual policy optimization is described in Algorithm 1.

In Algorithm 1, lines 1-9 are collecting and storing data in replay buffer for future learning. In line 6, \( e_m \) represents the minimum entropy produced by our dual policies. \( m \) contains the index of the policy generating \( e_m \). Lines 13-30 perform the update of \( Q \) and \( \pi \) networks. In lines 17-18 and 23-24, minimum entropy \( (e_m) \) along with the index of the policy generating the minimum entropy \( (m) \) are recorded. These are then used to update the \( Q \) and \( \pi \) networks.

In pure stochastic environment, both the policies \( \pi_{\phi_1} \) and \( \pi_{\phi_2} \) will be updated with equal probability. We believe that the delayed update of the policies has twofold benefits: (1) it stops one policy to take over by selecting some range of actions (overestimation avoidance); 2) results in quality update to the policy. Therefore, the policies will converge to optimal quickly.

5. Experimental Setup

We have tested our approach on different simulated environments created by MuJoCo [17]. MuJoCo is a library for modeling, simulation and visualization of multijoint

| Parameter                      | Value                        |
|-------------------------------|------------------------------|
| Optimizer                     | Adam ([21])                  |
| Max time steps                | 1 million                    |
| Learning rate                 | \( 3 \cdot 10^{-4} \)         |
| Discount \( (\gamma) \)       | 0.99                         |
| Replay buffer size            | \( 10^6 \)                   |
| # of hidden layers            | 2                            |
| # of hidden units per layer   | 256                          |
| # of samples per minibatch    | 256                          |
| Entropy target                | \( -\dim(\mathcal{A}) \) (e.g., \(-8 \) for Ant-v2) |
| Nonlinearity                  | ReLU                         |
| Target smoothing coefficient \( (\tau) \) | 0.005                      |
| Target update interval        | 1                            |
| Gradient steps critic         | 1                            |
| Reward scaling                | 1.0                          |
| Initial alpha                 | 0.2                          |
| Alpha learning rate           | \( 3 \cdot 10^{-4} \)        |
| Evaluation frequency          | 5e3                          |
| Evaluation episodes           | 10                           |
| Exploration time steps        | 25e3 for Ant-v2 and HalfCheetah |
|                               | 1e3 for Hopper-v2 and Walker2d |

Table 1: Hyperparameters.
Figure 4: Continued.
dynamics with contact. Both state and action spaces are continuous for all the environments. In our experiments, reward range is not bounded in specific range. Furthermore, we did not scale reward for any environment.

5.1. Environments Used. Figure 3 shows the environments that are used in our experiments.

First environment is Ant-v2 (look at Figure 3(a)). In Ant-v2, the goal is to make a four-legged ant walk as fast as
possible. It has 8 action dimensions and 111 state dimensions. All actions have range of -1 to +1. Environment reward range is infinite. Max time step to interact with the environment in one episode is 1000.

Second environment is HalfCheetah-v2 (look at Figure 3(b)). In HalfCheetah-v2, the goal is to make a two-legged cheetah walk as fast as possible. It has 6 action dimensions and 17 state dimensions. All actions have range of -1 to +1. Environment reward range is infinite. Max time step to interact with the environment in one episode is 1000.

Third environment is Hopper-v2 (look at Figure 3(c)). In Hopper-v2, the goal is to make a two-dimensional one-legged robot hop forward as fast as possible. It has 3 action dimensions and 11 state dimensions. All actions have range of -1 to +1. Environment reward range is infinite. Max time step to interact with the environment in one episode is 1000.

Fourth environment is Walker2d-v2 (look at Figure 3(d)). In Walker2d-v2, the goal is to make a two-legged bipedal robot walk as fast as possible. It has 6 action dimensions and 17 state dimensions. All actions have range of -1 to +1. Environment reward range is infinite. Max time step to interact with the environment in one episode is 1000.

We have defined the above-mentioned environments using Markov Decision Processes (MDP). An MDP consist of tuple $$(S, A, p, r)$$, where $$S$$ is a set of state space, $$A$$ is a set of action space, $$r$$ is a set of rewards, and $$p$$ is a state transition probability $$p(s', r | s, a)$$ also known as one step dynamics. State transition determines the probability of next state $$s'$$ given current state $$s_i \in S$$ and action $$a_i \in A$$. Reward range should be finite such that $$r : S \times A \rightarrow [r_{\text{min}}, r_{\text{max}}]$$. We used $${\rho}_f (t)$$ to represent trajectories $$t = (s_0, a_0, s_1, a_1, \cdots)$$ gathered by a policy $${\pi}(a_\theta | s_i)$. Our focus is on environments with continuous state and action spaces. The goal of experimentation is to determine how dual policy SAC help us to achieve better sample efficiency than previous SAC. We have evaluated our technique on continuous control tasks from the OpenAI gym environments (toolkit for developing and comparing reinforcement learning algorithms). For our implementation of dual policy SAC, we have used two feed forward neural network both with 256 neurons. On both actor and critic, we have used rectified linear units (ReLU) as an activation function. Critic receives both state and action as an input. Both networks are optimized using Adam optimizer [21] with a learning rate of 3e-4. At each time step, both networks are trained for a minibatch of 264 samples. Samples are obtained uniformly from a replay buffer. Replay buffer has a length of 1 million samples. Check Table 1 for hyperparameter details. Table 1 shows the list of hyperparameter that are used for training of our proposed dual policy soft actor critic. To stop policy from getting stuck into the local optima, we have used exploratory policy at the start of learning. After that, interaction with environment was dependent on policy.

6. Results and Discussion

Since RL algorithm shows a lot of variance during learning therefore, we have calculated the average of the evaluation roll outs calculated during training. For better comparison, we have trained five different agents of same algorithm with random seeds (0, 1, 2, 3, 4). Each agent is trained for one million time steps. Each agent is evaluated after 5000 time steps for 10 episodes, and average of 10 episodes is recorded.

Results of the experiments are shown in Figure 4. The solid curves show the average value across all the five agents. Shaded region shows the minimum and maximum values at the evaluation time step. The results show that dual policy SAC outperforms existing state of art and achieved high sampling efficiency. Dual policy SAC also outperforms existing algorithms in terms of learning speed.

As agent’s dynamics change with environment therefore, it is not always possible for the agent to learn on every environment [12, 22]. For this reason, TD3 is not able to train on HalfCheetah-V2 9. Our proposed dual-policy actor critic outperforms SAC while TD3 is not able to learn on this environment. It can be seen that our proposed dual-policy actor critic outperforms state or the art methods.

Table 2 shows the maximum reward that is achieved during the complete learning cycle and the average maximum reward that is achieved during evaluation roll-outs. It can be seen that other than Hopper-v2, our proposed dual-policy soft actor critic has achieved highest reward in all environments. Even in Hopper-v2, the highest reward of SAC is closed to what is achieved by our proposed algorithm. Last column shows the standard deviation.

7. Conclusion and Future Work

In this research, we have studied the difference between the objectives of SAC objective from the objective of conventional reinforcement learning algorithm. We have shown how maximum entropy framework works with RL objective and helps agent to explore environment in a structured way. We have discovered that maximizing the entropy results in the overestimation of entropy term. Recent proposed SAC automatically optimizes alpha using the entropy. This automation removes the need of manual alpha tuning but does not stop overestimation. It results in unstable and slow learning due to agent’s divergence from the optimal path.
In this research, we have proposed a new dual policy optimization technique to reduce the effects of entropy overestimation. Optimizing two policies gives us two additional benefits. First, it helps agent to further explore the state and action space. Second, it helps agent to capture different modes of optimal policy. We have tested our approach on different continuous control simulated environments created by MuJoCo [17]. Results show that our approach has achieved good sample efficiency as compared to state of the art RL algorithms. Our approach is also easy to implement and can result in better and faster learning.

In the future, we can train agent to learn from prioritized samples. Prioritizing sample from experience replay is proposed in [23]. Instead of prioritizing samples based on TD error, we can prioritize samples based on entropy. To encourage exploration and stable learning, we can prioritize samples based on low entropy which can further push agent towards exploration.

Data Availability

We have used MuJoCo (advanced physics simulation) for our experiments. For using it, access to the MuJoCo library is required which is available on request from https://www.roboti.us/license.html. The details of the setup and the parameters used by us are given in experimental setup section of the paper.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, “Dueling network architectures for deep reinforcement learning,” in 33rd International Conference on Machine Learning, ICML 2016, vol. 4, pp. 2939–2947, New York, USA, 2016.

[2] V. Mnih, A. P. Badia, M. Mirza et al., “Asynchronous methods for deep reinforcement learning,” in 33rd International Conference on Machine Learning, ICML 2016, vol. 4, pp. 2850–2869, New York, USA, 2016.

[3] M. Babaeizadeh, I. Froio, S. Tyree, J. Clemons, and J. Kautz, “Reinforcement learning through asynchronous advantage actor-critic on a GPU,” in 5th International Conference on Learning Representations, ICLR 2017- Conference Track Proceedings, Palais des Congrès Neptune, Toulon, France, 2017.

[4] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, “Trust region policy optimization,” in 32nd International Conference on Machine Learning, ICML 2015, vol. 3, pp. 1889–1897, Lille, France, 2015.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017, http://arxiv.org/abs/1707.06347.

[6] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, “Convergence results for single-step on-policy reinforcement-learning algorithms,” Machine Learning, vol. 38, no. 3, pp. 287–308, 2000.

[7] V. Mnih, K. Kavukcuoglu, D. Silver et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel et al., “Continuous control with deep reinforcement learning,” in 4th International Conference on Learning Representations, ICLR 2016- Conference Track Proceedings, Caribe Hilton, San Juan, Puerto Rico, 2016.

[9] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in 35th International Conference on Machine Learning, ICML 2018, vol. 5, pp. 2976–2989, January 2018, http://arxiv.org/abs/1801.01290.

[10] T. Haarnoja, A. Zhou, K. Hartikainen et al., “Soft actor-critic algorithms and applications,” 2018, http://arxiv.org/abs/1812.05905.

[11] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” in 4th International Conference on Learning Representations, ICLR 2016- Conference Track Proceedings, Caribe Hilton, San Juan, Puerto Rico, 2016.

[12] J. Duan, G. Yang, S. E. Li, Y. Ren, and B. Cheng, “Distributional soft actor-critic: off-policy reinforcement learning for addressing value estimation errors,” 2020, http://arxiv.org/abs/2001.02811.

[13] X. Ma, L. Xia, Z. Zhou, J. Yang, and Q. Zhao, “DSAC: distributional soft actor critic for risk-sensitive reinforcement learning,” 2020, http://arxiv.org/abs/2004.14547.

[14] W. Zhou, Y. Li, Y. Yang, H. Wang, and T. M. Hospedales, “Online meta-critic learning for off-policy actor-critic methods,” 2020, http://arxiv.org/abs/2003.05334.

[15] M. Hessel, J. Modayil, H. Van Hasselt et al., “Rainbow: combining improvements in deep reinforcement learning,” in 32nd AAAI Conference on Artificial Intelligence, AAAI 2018, pp. 3215–3222, Hilton New Orleans Riverside, New Orleans, Louisiana, USA, 2018.

[16] P. S. Thomas and E. Brunskill, “Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines,” pp. 1057–1063, 2017, http://arxiv.org/abs/1706.06643.

[17] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: a physics engine for model-based control,” in 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 5026–5033, Vila-moura-Algarve, Portugal, October 2012.

[18] H. Van Hasselt, “Double Q-learning,” in Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010, NIPS 2010, Whistler, Canada, 2010.

[19] B. Mavrin, H. Yao, and L. Kong, “Deep reinforcement learning with a stochastic actor,” in 2015, http://arxiv.org/abs/1503.05905.

[20] S. Fujimoto, H. Van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in 35th International Conference on Machine Learning, ICML 2018, vol. 4, pp. 2587–2601, Stockholmsmässan, Stockholm SWEDEN, 2018.

[21] D. P. Kingma and J. L. Ba, “Adam: a method for stochastic optimization,” in 3rd International Conference on Learning Representations, ICLR 2015- Conference Track Proceedings, The Hilton San Diego Resort & Spa, 2015.
[22] Y. Wang and T. Ni, “Meta-SAC: auto-tune the entropy temperature of soft actor-critic via metagradient,” 2020, http://arxiv.org/abs/2007.01932.

[23] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” in 4th International Conference on Learning Representations, ICLR 2016- Conference Track Proceedings, Caribe Hilton, San Juan, Puerto Rico, 2016.