

# Recursive Least Squares Advantage Actor-Critic Algorithms

Yuan Wang, Chunyuan Zhang, Tianzong Yu, Meng Ma

**Abstract**—As an important algorithm in deep reinforcement learning, advantage actor critic (A2C) has been widely succeeded in both discrete and continuous control tasks with raw pixel inputs, but its sample efficiency still needs to improve more. In traditional reinforcement learning, actor-critic algorithms generally use the recursive least squares (RLS) technology to update the parameter of linear function approximators for accelerating their convergence speed. However, A2C algorithms seldom use this technology to train deep neural networks (DNNs) for improving their sample efficiency. In this paper, we propose two novel RLS-based A2C algorithms and investigate their performance. Both proposed algorithms, called RLSSA2C and RLSNA2C, use the RLS method to train the critic network and the hidden layers of the actor network. The main difference between them is at the policy learning step. RLSSA2C uses an ordinary first-order gradient descent algorithm and the standard policy gradient to learn the policy parameter. RLSNA2C uses the Kronecker-factored approximation, the RLS method and the natural policy gradient to learn the compatible parameter and the policy parameter. In addition, we analyze the complexity and convergence of both algorithms, and present three tricks for further improving their convergence speed. Finally, we demonstrate the effectiveness of both algorithms on 40 games in the Atari 2600 environment and 11 tasks in the MuJoCo environment. From the experimental results, it is shown that our both algorithms have better sample efficiency than the vanilla A2C on most games or tasks, and have higher computational efficiency than other two state-of-the-art algorithms.

**Index Terms**—Deep reinforcement learning (DRL), advantage actor-critic (A2C), recursive least squares (RLS), standard policy gradient (SPG), natural policy gradient (NPG).

## I. INTRODUCTION

Reinforcement learning (RL) is an important machine learning methodology for solving sequential decision-making problems. In RL, the agent aims to learn an optimal policy for maximizing the cumulative return by interacting with the initially unknown environment [1]. For the past 40 years, RL has roughly experienced three historical periods, namely, tabular representation RL (TRRL), linear function approximation RL (LFARL) and deep RL (DRL). TRRL can only solve a few simple problems with small-scale discrete state and action spaces. LFARL can only solve some control tasks with low-dimensional continuous state and action spaces since its approximation capability is still limited [2]. In recent years, by combining with various deep neural networks (DNNs),

This work was supported by the National Natural Science Foundation of China under Grant 61762032 and Grant 11961018. (Corresponding author: Chunyuan Zhang).

Y. Wang, C. Zhang, T. Yu and M. Ma are with the School of Computer Science and Technology, Hainan University, Haikou 570228, China (email: 20085400210073@hainanu.edu.cn; 990721@hainanu.edu.cn; 20081200210007@hainanu.edu.cn; 20085400210053@hainanu.edu.cn).

DRL has shown a huge potential to solve real-world complex problems [3]-[6] and has received more and more research interest. Similar to LFARL classified in [7], DRL can also be classified into three categories: deep value function approximation (DVFA) [8], [9], deep policy search (DPS) [10] and deep actor-critic (DAC) methods [11]-[17]. DAC algorithms can be viewed as a hybrid of DVFA and DPS. They generally have a critic for policy evaluation and an actor for policy learning. Among three classes of DRL methods, DAC is more effective than DVFA and DPS for online learning real-world problems, and thus has received extensive attention.

In recent years, many novel DAC algorithms have been proposed. According to the policy type used in the actor, they can be roughly divided into two main subclasses. One subclass is the DAC algorithms with the deterministic policy. The actor-critic algorithms with the deterministic policy gradient (DPG) are first proposed in [11]. Based on this work, a few variants, such as deep DPG (DDPG) [12], recurrent DPG (RDPG) [13] and twin delayed DDPG (TD3) [14], have been suggested recently, but they can solve only continuous control tasks. The other subclass is the DAC algorithms with the stochastic policy gradient. In this subclass, the most famous algorithm is perhaps the asynchronous advantage actor-critic (A3C) algorithm [15]. A3C employs multiple workers to interacting with each own environment in parallel, uses the accumulated samples of each worker to update the shared model and uses the parameters of the shared model to update the model of each worker asynchronously. Compared with DPG-type DAC algorithms, A3C can solve both discrete and continuous control tasks without the experience replay. However, A3C doesn't work better than its synchronized version [18], namely the synchronous advantage actor-critic (A2C) algorithm [15], which uses all samples obtained by each worker to update the shared model and uses the shared model to select the action of each worker synchronously. A2C is easier to implement and has become a baseline algorithm in OpenAI. Therefore, we will focus on A2C in this paper.

How to improve the sample efficiency of the agent is an open issue in RL. Although A2C and A3C are very excellent, their sample efficiency still needs to improve more. In recent years, there have been some studies such as combining with the experience replay [19], [20] and the entropy maximization [21] to tackle this problem. Here we mainly focus on another way in which researchers try to use more advanced optimization methods for gradient updates. Schulman et al. propose the proximal policy optimization (PPO) [16], which uses a novel objective with clipped probability ratios and forms a pessimistic estimate of the performance of the policy.

Compared with the trust region policy optimization (TRPO) [17], PPO is much simpler to implement. Byun et al. propose the proximal policy gradient (PPG) [22], which shows that the performance similar to PPO can be obtained by using the gradient formula from the original policy gradient theorem. Wu et al. propose the actor critic using Kronecker-factored trust region (ACKTR), which uses a Kronecker-factored approximation [23] to the natural policy gradient (NPG) that allows the covariance matrix of the gradient to be inverted efficiently [24]. These algorithms are more effective than the vanilla A2C or A3C with ordinary first-order optimization algorithms, but have higher computational complexities and run slowly. By using Kostrikov's source code [25] to test on Atari games, we find that PPO and ACKTR are only 1/10 and 7/12 as fast as A2C with RMSProp [26].

In LFARL, traditional actor-critic algorithms usually use the recursive least squares (RLS) method to improve their convergence performance. In 1996, Bradtke and Barto first propose the least squares temporal difference (LSTD) algorithm and define a recursive version (namely RLSTD) [27]. In 1999, Konda and Tsitsiklis first propose a class of two-time-scale actor-critic algorithms with linear function approximations, and point out that it is possible to use LSTD for policy evaluation [28]. After that, Xu et al. propose an actor-critic algorithm by using the RLSTD( $\lambda$ ) algorithm as the critic [29]. In 2003, Peter et al. first propose the natural actor-critic (NAC) algorithm, which uses the LSTD-Q( $\lambda$ ) algorithm for policy evaluation and uses NPG for policy learning [30]. On this basis, Park et al. propose the RLS-based NAC algorithm [31], and Bhatnagar et al. provide four actor-critic algorithms as well as their convergence proofs [32]. There are some other RLS-based actor-critic algorithms such as KDHP [7] and CIPG [33]. RLS has been the baseline optimization method for traditional actor-critic algorithms. However, to the best of our knowledge, there aren't any RLS-based DAC algorithms to be proposed, since the DNN approximation is much more complicated than the linear function approximation. In our previous work [34], we propose a class of RLS optimization algorithms for DNNs and validate their effectiveness on some classification benchmark datasets. Whereas, there is a big difference between RL and supervised learning, and there are still some obstacles we need to overcome.

In this paper, we try to introduce the RLS optimization into the A2C algorithm. We propose two RLS-based A2C algorithms, called RLSSA2C and RLSNA2C, respectively. Both of them use the same loss function as the vanilla A2C, and employ the RLS method to optimize their critic networks and the hidden layers of their actor networks. The main difference between them is the policy learning. RLSSA2C uses the standard policy gradient (SPG) and an ordinary first-order gradient descent algorithm to update the policy parameters, and RLSNA2C uses the NPG, the Kronecker-factored approximation and the RLS method to learn the compatible parameter and the policy parameters. We show that their computational complexities have the same order as the vanilla A2C with the first-order optimization algorithm. In addition, we also provide some tricks for accelerating their convergence speed. Finally, we demonstrate their effectiveness

on 40 games in the Atari 2600 environment and 11 tasks in the MuJoCo environment. Experimental results show that both RLSSA2C and RLSNA2C have better sample efficiency than the vanilla A2C on most games. Furthermore, they can achieve a faster running speed than PPO and ACKTR.

The rest of this paper is organized as follows. In Section II, we introduce some background knowledge. In Section III, we present the detail derivation of our proposed algorithms. In Section IV, we analyze the computational complexity and convergence of our proposed algorithms, and provide three tricks for further improving their convergence speed and solution. In Section V, we demonstrate the effectiveness of our proposed algorithms on Atari games and MuJoCo tasks. Finally, we conclude our work in Section VI.

## II. BACKGROUND

In this section, we briefly review Markov decision processes (MDPs), stochastic policy gradient, convolutional neural networks (CNNs) and the vanilla A2C algorithm. In addition, we also introduce some notations used in this paper.

### A. Markov Decision Process

In RL, a sequential decision-making problem is generally formulated into a Markov decision process (MDP) defined as  $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , where  $\mathcal{S}$  is the state space,  $\mathcal{A}$  is the action space,  $p(s'_t|s_t, a_t) \in [0, 1]$  and  $r_t \in \mathcal{R}$  are the state-transition probability distribution and the immediate reward from the state  $s_t$  to the next state  $s'_t$  by taking the action  $a_t$  at time step  $t$ , and  $\gamma \in (0, 1]$  is the discount factor. The action  $a_t$  is selected by a policy, which can be either stochastic or deterministic. In this paper, we focus on the former and use a tensor or matrix  $\Theta$  to parameterize it. A stochastic policy  $\pi(a_t|s_t; \Theta)$  describes the probability distribution of taking  $a_t$  in  $s_t$ .

For an MDP, the goal of RL is to find an optimal policy  $\pi^*$  (also an optimal policy parameter  $\Theta^*$ ) to maximize the cumulative expected return  $J(\Theta)$  from an initial state  $s_0$ , namely

$$\Theta^* = \underset{\Theta}{\operatorname{argmax}} J(\Theta) = \underset{\Theta}{\operatorname{argmax}} \mathbb{E}_{\pi} \left[ \sum_{t=0}^{\infty} \gamma^t r_t | s_0 \right] \quad (1)$$

Unfortunately,  $J(\Theta) = \mathbb{E}_{\pi} [\sum_{t=0}^{\infty} \gamma^t r_t | s_0]$  is difficult to be calculated directly, since  $p(s'|s_t, a_t)$  is unknown in RL, and  $s'_t$  and  $r_t$  can be obtained only by the agent's interaction with the environment. DRL generally uses the DAC method to solve  $\Theta^*$ . At time step  $t$ , the critic uses the DVFA method to approximate the state-value function  $V^{\pi}(s_t) = \mathbb{E}_{\pi} [\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s_t]$  and the action-value function  $Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi} [\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s_t, a_0 = a_t]$  for evaluating the performance of the current policy  $\pi$ , and then the actor uses the policy gradient to update  $\Theta_t$ .

### B. Stochastic Policy Gradient

Currently, there are two main types of policy gradients: SPG and NPG. From the policy gradient theorem [35], SPG can be calculated as

$$\nabla_{\Theta_t} J(\Theta_t) = \mathbb{E}_{\pi} [A^{\pi}(s_t, a_t) \nabla_{\Theta_t} \log \pi(a_t|s_t; \Theta_t)] \quad (2)$$

where  $\nabla_{\Theta_t} J(\Theta_t)$  denotes  $\partial J(\Theta_t)/\partial \Theta_t$ , and  $A^\pi(s_t, a_t)$  is the advantage function.  $A^\pi(s_t, a_t)$  measures how much better than the average it is to take an action  $a_t$  [36]. It is defined as

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \quad (3)$$

Unlike SPG, NPG does not follow the steepest direction in the policy parameter space but the steepest direction with respect to the Fisher information metric [37]. It is defined as

$$\tilde{\nabla}_{\Theta_t} J(\Theta_t) = (\mathbf{F}(\Theta_t))^{-1} \nabla_{\Theta_t} J(\Theta_t) \quad (4)$$

where  $\mathbf{F}(\Theta_t)$  is the Fisher information matrix defined by

$$\mathbf{F}(\Theta_t) = \mathbb{E}_\pi [v(\nabla_{\Theta_t} \log \pi(a_t | s_t; \Theta_t)) v(\nabla_{\Theta_t} \log \pi(a_t | s_t; \Theta_t))^T] \quad (5)$$

where  $v(\cdot)$  denotes reshaping the given tensor or matrix into a column vector. To avoid computing the inverse of  $\mathbf{F}(\Theta_t)$ , (4) can also be redefined as

$$\tilde{\nabla}_{\Theta_t} J(\Theta_t) = m(w_t) \quad (6)$$

where  $m(\cdot)$  denotes reshaping the given vector into a matrix or tensor, and  $w_t$  is the parameter of the linear compatible function approximator [37] defined by

$$\tilde{A}(s_t, a_t; w_t) = w_t^T v(\nabla_{\Theta_t} \log \pi(a_t | s_t; \Theta_t)) \quad (7)$$

$\tilde{A}(s_t, a_t; w_t)$  is the compatible approximation of  $A^\pi(s_t, a_t)$ .

### C. Convolutional Neural Network

In DAC, CNNs are widely used to approximate  $V^\pi(s)$  and  $\pi$  for solving control tasks with raw pixel inputs. A CNN generally consists of some convolutional (conv) layers, pooling layers and fully-connected (fc) layers. Since there are no learnable parameters in pooling layers, we only review the forward learning of conv layers and fc layers. Let  $l, M, X_t^l, Z_t^l, Y_t^l, \Theta_t^l$  and  $f_l(\cdot)$  denote the current layer, the mini-batch size, the mini-batch input, the pre-activation input, the activation output, the parameter, the activation function in this layer at current time  $t$ , respectively. For brevity, we omit the bias term of each layer in this paper.

In a conv layer,  $X_t^l \in \mathcal{R}^{M \times C_{l-1} \times H_{l-1} \times W_{l-1}}$  is convolved with the kernel  $\Theta_t^l \in \mathcal{R}^{C_{l-1} \times C_l \times H_l^k \times W_l^k}$  and puts through  $f_l(\cdot)$  to form  $Y_t^l \in \mathcal{R}^{M \times C_l \times H_l \times W_l}$ , where  $C_l, H_l, W_l, H_l^k$  and  $W_l^k$  denote the number of output channels, the output image height, the output image width, the kernel height and the kernel width, respectively. Let  $\check{X}_{t(\cdot,\cdot,\cdot,h,w)}^l \in \mathcal{R}^{M \times C_{l-1} \times H_l^k \times W_l^k}$  denote the input selection of the output pixel  $Y_{t(\cdot,\cdot,h,w)}^l$ . Reshape  $\check{X}_t^l$  and  $\Theta_t^l$  as  $\hat{X}_t^l \in \mathcal{R}^{M \times C_{l-1} H_l^k W_l^k \times H_l W_l}$  and  $\hat{\Theta}_t^l \in \mathcal{R}^{C_{l-1} H_l^k W_l^k \times C_l}$ , respectively. Then,  $Y_{t(\cdot,\cdot,j)}^l$  is defined as

$$Y_{t(\cdot,\cdot,j)}^l = f_l(Z_{t(\cdot,\cdot,j)}^l) = f_l(\hat{X}_{t(\cdot,\cdot,j)}^l \hat{\Theta}_t^l) \quad (8)$$

In an fc layer,  $X_t^l \in \mathcal{R}^{M \times N_{l-1}}$  is weighed to connect all output neurons by  $\Theta_t^l \in \mathcal{R}^{N_{l-1} \times N_l}$  and puts through  $f_l(\cdot)$  to form  $Y_t^l \in \mathcal{R}^{M \times N_l}$ , where  $N_l$  denotes the number of output neurons. Namely,  $Y_t^l$  is defined as

$$Y_t^l = f_l(Z_t^l) = f_l(X_t^l \Theta_t^l) \quad (9)$$

### D. Advantage Actor Critic

A2C is an important baseline algorithm in OpenAI. In A2C, there are  $N$  parallel workers, a shared critic network and a shared actor network. The critic network and the actor network can be joint or disjoint. If both networks are joint, they will share lower layers but have distinct output layers.

The algorithm flow of A2C can be summarized as follows. At current iteration step  $t$ , it lets each worker interact with each own environment for  $T$  timesteps, and uses all state-transition pairs to form the mini batch  $\mathcal{M}_t = \{(s_{t,i}^{(k)}, a_{t,i}^{(k)}, s'_{t,i}^{(k)}, r_{t,i}^{(k)}, d_{t,i}^{(k)})\}_{i=1, \dots, N}^{k=1, \dots, T}$ , where  $d_{t,i}^{(k)} \in \{0, 1\}$  denotes that the next state  $s'_{t,i}^{(k)}$  of the  $i^{th}$  worker is the terminal state or not at the  $k^{th}$  timestep, and the mini-batch size  $M$  is equal to  $NT$ . Then, it calculates the loss function defined by

$$L(\Psi_t, \Theta_t) = L(\Psi_t) + L(\Theta_t) + \eta E(\Theta_t) \quad (10)$$

where  $\Psi_t$  and  $L(\Psi_t)$  are the parameter and the loss function of the critic network,  $\Theta_t$  and  $L(\Theta_t)$  are the parameter and the loss function of the actor network, and  $\eta E(\Theta_t)$  is the entropy regularization term with a small  $\eta > 0$ .  $L(\Psi_t)$  is defined as

$$L(\Psi_t) = \frac{1}{2NT} \|A(S_t, A_t)\|_F^2 \quad (11)$$

where  $S_t = [s_{t,1}^{(1)}, \dots, s_{t,N}^{(T)}]^T$ ,  $A_t = [a_{t,1}^{(1)}, \dots, a_{t,N}^{(T)}]^T$ , and  $A(S_t, A_t) \in \mathcal{R}^{NT}$  denotes the estimate value of  $A^\pi(S_t, A_t)$ , which is calculated as

$$A(S_t, A_t) = Q(S_t, A_t) - V(S_t; \Psi_t) \quad (12)$$

where  $V(S_t; \Psi_t)$  and  $Q(S_t, A_t)$  denote the estimate values of  $V^\pi(S_t)$  and  $Q^\pi(S_t, A_t)$ , respectively. The former is the actual output of the critic network, and the latter is the desired output of the critic network. Each element in the latter is calculated as

$$Q(s_{t,i}^{(k)}, a_{t,i}^{(k)}) = \begin{cases} r_{t,i}^{(T)} + \gamma(1 - d_{t,i}^{(T)})V(s'_{t,i}^{(T)}; \Psi_t), & k = T \\ r_{t,i}^{(k)} + \gamma(1 - d_{t,i}^{(k)})Q(s'_{t,i}^{k+1}, a_{t,i}^{k+1}), & k < T \end{cases} \quad (13)$$

where  $V(s'_{t,i}^{(T)}; \Psi_t)$  is also approximated by the critic network.  $L(\Theta_t)$  and  $E(\Theta_t)$  are defined as follows

$$L(\Theta_t) = -\frac{1}{NT} \sum_{i=1}^N \sum_{k=1}^T A(s_{t,i}^{(k)}, a_{t,i}^{(k)}) \log \pi(a_{t,i}^{(k)} | s_{t,i}^{(k)}; \Theta_t) \quad (14)$$

$$E(\Theta_t) = -\frac{1}{NT} \sum_{i=1}^N \sum_{k=1}^T \pi(a_{t,i}^{(k)} | s_{t,i}^{(k)}; \Theta_t) \log \pi(a_{t,i}^{(k)} | s_{t,i}^{(k)}; \Theta_t) \quad (15)$$

where  $\pi(a_{t,i}^{(k)} | s_{t,i}^{(k)}; \Theta_t)$  is the actual output of the actor network. Finally, A2C uses (10) to update  $\Psi_t$  and  $\Theta_t$  with some first-order optimization algorithms such as RMSProp.

### III. RLS-BASED ADVANTAGE ACTOR CRITIC

In this section, we try to integrate the RLS method into the vanilla A2C with CNNs. Under the loss function defined by (10), the RLS update rules for four different types of layers used in critic and actor networks are derived respectively. On the basis, we propose RLSSA2C and RLSNA2C algorithms.

### A. Optimizing Critic Output Layer

As introduced in Section I, the critic of traditional actor-critic algorithms widely uses LSTD for policy evaluation. From (11), (12) and (13),  $A(\mathbf{S}_t, \mathbf{A}_t) \in \mathcal{R}^{NT}$  is a temporal difference error vector. It means we can also use LSTD for updating the critic parameter. Let  $\mathcal{D}_t = \{\mathcal{M}_1, \dots, \mathcal{M}_t\}$  denote the state-transition mini-batch dataset from the start to the current iteration step  $t$ . On the basis, we define an auxiliary least squares loss function as

$$\tilde{L}(\Psi) = \frac{1}{2NT} \sum_{n=1}^t \lambda^{t-n} \|A(\mathbf{S}_n, \mathbf{A}_n)\|_F^2 \quad (16)$$

where  $\lambda \in (0, 1]$  is the forgetting factor. Then, the learning problem of the current critic parameter can be described as

$$\Psi_{t+1} = \operatorname{argmin}_{\Psi} \tilde{L}(\Psi) \quad (17)$$

In the critic network of A2C, the output layer is generally a linear fc layer with one output neuron. In other words, the activation function of this layer is the identity function. Thus, from (9),  $V(\mathbf{S}_n; \Psi)$  is calculated as

$$V(\mathbf{S}_n; \Psi) = \mathbf{X}_n^l \Psi^l \quad (18)$$

where  $\Psi^l$  is the parameter of this layer. Then, from (12), (16) can be rewritten as

$$\tilde{L}(\Psi) = \frac{1}{2NT} \sum_{n=1}^t \lambda^{t-n} \|Q(\mathbf{S}_n, \mathbf{A}_n) - \mathbf{X}_n^l \Psi^l\|_F^2 \quad (19)$$

By the chain rule for  $\Psi^l$ ,  $\nabla_{\Psi^l} \tilde{L}(\Psi)$  can be derived as

$$\nabla_{\Psi^l} \tilde{L}(\Psi) = -\frac{1}{NT} \sum_{n=1}^t \lambda^{t-n} (\mathbf{X}_n^l)^T (Q(\mathbf{S}_n, \mathbf{A}_n) - \mathbf{X}_n^l \Psi^l) \quad (20)$$

Let  $\nabla_{\Psi^l} \tilde{L}(\Psi) = 0$ . We can easily obtain the least squares solution of  $\Psi_{t+1}^l$ , namely

$$\Psi_{t+1}^l = (\mathbf{H}_{t+1}^l)^{-1} b_{t+1}^l \quad (21)$$

where  $\mathbf{H}_{t+1}^l \in \mathcal{R}^{N_{l-1} \times N_{l-1}}$  and  $b_{t+1}^l \in \mathcal{R}^{N_{l-1}}$  are defined as

$$\mathbf{H}_{t+1}^l = \frac{1}{NT} \sum_{n=1}^t \lambda^{t-n} (\mathbf{X}_n^l)^T \mathbf{X}_n^l \quad (22)$$

$$b_{t+1}^l = \frac{1}{NT} \sum_{n=1}^t \lambda^{t-n} (\mathbf{X}_n^l)^T Q(\mathbf{S}_n, \mathbf{A}_n) \quad (23)$$

Next, to avoid computing the inverse of  $\mathbf{H}_{t+1}^l$  and realize online learning, we try to derive the RLS solution of  $\Psi_{t+1}^l$ . Rewrite (22) and (23) as the following incremental update equations

$$\mathbf{H}_{t+1}^l = \lambda \mathbf{H}_t^l + \frac{1}{NT} (\mathbf{X}_t^l)^T \mathbf{X}_t^l \quad (24)$$

$$b_{t+1}^l = \lambda b_t^l + \frac{1}{NT} (\mathbf{X}_t^l)^T Q(\mathbf{S}_t, \mathbf{A}_t) \quad (25)$$

However, by using the Sherman-Morrison matrix inversion lemma [38] for (24), the recursive update of  $(\mathbf{H}_{t+1}^l)^{-1}$  still includes a new inverse matrix, since the rightmost term in (24) is a matrix product rather than a vector product. In our previous

work [34], we propose an average-approximation method to tackle this problem and reduce the computation burden. Using this method, we define

$$\bar{x}_t^l = \frac{1}{NT} \sum_{i=1}^{NT} \mathbf{X}_{t(i,:)}^l \quad (26)$$

$$\bar{q}_t = \frac{1}{NT} \sum_{i=1}^{NT} Q(\mathbf{S}_{t(i,:)}, \mathbf{A}_{t(i,:)}) \quad (27)$$

where  $\mathbf{X}_{t(i,:)}^l \in \mathcal{R}^{N_{l-1}}$  and  $Q(\mathbf{S}_{t(i,:}), \mathbf{A}_{t(i,:)}) \in \mathcal{R}$  are the column vectors sliced from  $\mathbf{X}_t^l$  and  $Q(\mathbf{S}_t, \mathbf{A}_t)$ , respectively. On the basis, we can rewrite (24) and (25) as follows

$$\mathbf{H}_{t+1}^l = \lambda \mathbf{H}_t^l + k \bar{x}_t^l (\bar{x}_t^l)^T \quad (28)$$

$$b_{t+1}^l = \lambda b_t^l + k \bar{x}_t^l \bar{q}_t \quad (29)$$

where  $k > 0$  is the average scaling factor. Let  $\mathbf{P}_t = (\mathbf{H}_t)^{-1}$ . Then, using the Sherman-Morrison matrix inversion lemma and (17), we finally obtain the following RLS update rules

$$\mathbf{P}_{t+1}^l \approx \frac{1}{\lambda} \left( \mathbf{P}_t^l - \frac{k \mathbf{P}_t^l \bar{x}_t^l (\bar{x}_t^l)^T \mathbf{P}_t^l}{\lambda + k (\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l} \right) \quad (30)$$

$$\Psi_{t+1}^l \approx \Psi_t^l + \frac{k \mathbf{P}_t^l \bar{x}_t^l (\bar{q}_t - \bar{v}_t)}{\lambda + k (\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l} \quad (31)$$

where  $\bar{v}_t$  is defined as

$$\bar{v}_t = \frac{1}{NT} \sum_{i=1}^{NT} V(\mathbf{S}_{t(i,:)}; \Psi_t) \quad (32)$$

In our previous work [34], we find that the RLS optimization can be converted into a gradient descent algorithm, which is easier to implement by using PyTorch or TensorFlow. Based on (10)-(12),  $\nabla_{\Psi_t^l} L(\Psi_t, \Theta_t)$  can be derived as

$$\nabla_{\Psi_t^l} L(\Psi_t, \Theta_t) = -\frac{1}{NT} (\mathbf{X}_t^l)^T (Q(\mathbf{S}_t, \mathbf{A}_t) - \mathbf{X}_t^l \Psi_t^l) \quad (33)$$

In (28) and (29), we ever use  $k \bar{x}_t^l (\bar{x}_t^l)^T$  and  $k \bar{x}_t^l \bar{q}_t$  to replace  $\frac{1}{NT} (\mathbf{X}_t^l)^T \mathbf{X}_t^l$  and  $\frac{1}{NT} (\mathbf{X}_t^l)^T Q(\mathbf{S}_t, \mathbf{A}_t)$ , respectively. On the basis, we can easily get

$$k \bar{x}_t^l (\bar{q}_t - \bar{v}_t) \approx \frac{1}{NT} (\mathbf{X}_t^l)^T (Q(\mathbf{S}_t, \mathbf{A}_t) - \mathbf{X}_t^l \Psi_t^l) \quad (34)$$

Thus, we can rewrite (31) as the following gradient update form

$$\Psi_{t+1}^l \approx \Psi_t^l - \frac{\mathbf{P}_t^l \nabla_{\Psi_t^l} L(\Psi_t, \Theta_t)}{\lambda + k (\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l} \quad (35)$$

where  $\frac{\mathbf{P}_t^l}{\lambda + k (\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l}$  is the learning rate. That means we needn't change the loss function defined by (10).

### B. Optimizing Actor Output Layer

In the actor network of A2C, the output layer is also an fc layer, which is to output  $\pi(\mathbf{A}_t | \mathbf{S}_t; \Theta_t)$ . For discrete control problems, A2C often uses the Softmax function to define  $\pi$ , called the Softmax policy. For continuous control problems, A2C usually uses the Gaussian function to define  $\pi$ , called the Gaussian policy. Unlike (11), (14) and (15) are difficult to be converted into least squares loss functions. In fact, for the

same reason, many traditional actor-critic algorithms only use the RLS method to update the critic parameter, but use the SPG descent method to update the actor parameter. Our first algorithm RLSSA2C will follow this method. Its actor output layer is optimized by an ordinary first-order optimization algorithm. For example, its update rules based on RMSProp are defined as follows

$$C_{t+1}^l = \rho C_t^l + (1 - \rho) \nabla_{\Theta_t^l} L(\Psi_t, \Theta_t) \odot \nabla_{\Theta_t^l} L(\Psi_t, \Theta_t) \quad (36)$$

$$\Theta_{t+1}^l = \Theta_t^l - \frac{\epsilon}{\sqrt{\delta + C_{t+1}^l}} \odot \nabla_{\Theta_t^l} L(\Psi_t, \Theta_t) \quad (37)$$

where  $C_t^l$  is the accumulative squared gradient,  $\rho$  is the decay rate,  $\epsilon$  is the learning rate,  $\delta > 0$  is a small constant,  $\nabla_{\Theta_t^l} L(\Psi_t, \Theta_t)$  is SPG, and  $\odot$  denotes the Hadamard product.

As introduced in Section II. B, there is another type of policy gradients, namely NPG [30], [37], which has yielded a few novel NAC algorithms. To avoid computing the inverse of the Fisher information matrix, some traditional NAC algorithms, often use the RLS method to approximate the compatible parameter  $W_t$ , and use  $W_t$  as NPG for updating the actor parameter [31], [32]. Following this way, we define an auxiliary least squares loss function based on  $\mathcal{D}_t$  as

$$\tilde{L}(w) = \frac{1}{2NT} \sum_{n=1}^t \lambda^{t-n} \|A(S_n, A_n) - \tilde{A}(S_n, A_n; w)\|_F^2 \quad (38)$$

where  $A(S_n, A_n)$  is calculated by (12). From (7), each element in  $\tilde{A}(S_n, A_n; w)$  is defined as

$$\tilde{A}(S_{n(i,:)}, A_{n(i,:)}; w) = w^T G_{n(i,:)}^{\Theta^l} \quad (39)$$

where  $G_{n(i,:)}^{\Theta^l}$  denotes  $v(\nabla_{\Theta_t^l} \log \pi(A_{n(i,:)}) | X_{n(i,:)}^l; \Theta_t^l) \in \mathcal{R}^{N_{l-1} N_l}$  for simplifying notations. Then, the learning problem of the current compatible parameter can be described as

$$w_{t+1} = \underset{w}{\operatorname{argmin}} \tilde{L}(w) \quad (40)$$

Let  $\nabla_w \tilde{L}(w) = 0$ . We can easily obtain the least squares solution of  $w_{t+1}$ , namely

$$w_{t+1} = (H_{t+1}^l)^{-1} b_{t+1}^l \quad (41)$$

where  $H_{t+1}^l \in \mathcal{R}^{N_{l-1} N_l \times N_{l-1} N_l}$  and  $b_{t+1}^l \in \mathcal{R}^{N_{l-1} N_l}$  are defined as follows

$$H_{t+1}^l = \frac{1}{NT} \sum_{n=1}^t \lambda^{t-n} (G_n^{\Theta^l})^T G_n^{\Theta^l} \quad (42)$$

$$b_{t+1}^l = \frac{1}{NT} \sum_{n=1}^t \lambda^{t-n} (G_n^{\Theta^l})^T A(S_n, A_n) \quad (43)$$

where  $G_n^{\Theta^l} = [G_{n(1,:)}^{\Theta^l}, \dots, G_{n(NT,:)}^{\Theta^l}]^T \in \mathcal{R}^{NT \times N_{l-1} N_l}$ . By using the same derivation method in Section III. A, we can easily obtain the RLS update rules for  $P_{t+1}^l$  and  $w_{t+1}$ . However, here  $P_{t+1}^l = (H_{t+1}^l)^{-1}$  will be  $N_l^2$  times as complex as in the critic output layer.

Since the forgetting factor  $\lambda$  is 1 or close to 1 in general,  $H_{t+1}^l$  can be approximately viewed as a Fisher information matrix. In addition, by the chain rule, we easily get

$$G_{n(i,:)}^{\Theta^l} = X_{n(i,:)}^l (G_{n(i,:)}^{Z_{n(i,:)}^l})^T \quad (44)$$

where  $G_{n(i,:)}^{Z_{n(i,:)}^l} = \nabla_{Z_{n(i,:)}^l} \log \pi(A_{n(i,:)}) | X_{n(i,:)}^l; \Theta^l$  and  $Z_{n(i,:)}^l = (\Theta^l)^T X_{n(i,:)}^l$ . Then, we have

$$(G_{n(i,:)}^{\Theta^l})^T G_{n(i,:)}^{\Theta^l} = X_{n(i,:)}^l (X_{n(i,:)}^l)^T \otimes G_{n(i,:)}^{Z_{n(i,:)}^l} (G_{n(i,:)}^{Z_{n(i,:)}^l})^T \quad (45)$$

where  $\otimes$  is the Kronecker product. To reduce the memory and computation burden, we use the Kronecker-factored approximation [23], [24] to rewrite (42) as

$$H_{t+1}^l \approx H_{t+1}^{(1)} \otimes H_{t+1}^{(2)} \quad (46)$$

where  $H_{t+1}^{(1)}$  and  $H_{t+1}^{(2)}$  are defined as follows

$$H_{t+1}^{(1)} = \frac{k}{NT} \sum_{n=1}^t \lambda^{t-n} (X_n^l)^T X_n^l \quad (47)$$

$$H_{t+1}^{(2)} = \frac{k}{NT} \sum_{n=1}^t \lambda^{t-n} (G_n^{Z_n^l})^T G_n^{Z_n^l} \quad (48)$$

where  $G_n^{Z_n^l} = [G_{n(1,:)}^{Z_{n(1,:)}^l}, \dots, G_{n(NT,:)}^{Z_{n(NT,:)}^l}]^T \in \mathcal{R}^{NT \times N_l}$ , and  $k > 0$  is another average scaling factor. Let  $P_t^{(1)} = (H_t^{(1)})^{-1}$  and  $P_t^{(2)} = (H_t^{(2)})^{-1}$ . Now, we rewrite (47) and (48) as the incremental forms, use the Sherman-Morrison matrix inversion lemma for each sample in the current mini batch, and average the recursive results. We can easily get

$$P_{t+1}^{(1)} \approx \frac{1}{\lambda} \left( P_t^{(1)} - \frac{k}{NT} \sum_{i=1}^{NT} \frac{P_t^{(1)} X_{t(i,:)}^l (X_{t(i,:)}^l)^T P_t^{(1)}}{\lambda + k (X_{t(i,:)}^l)^T P_t^{(1)} X_{t(i,:)}^l} \right) \quad (49)$$

$$P_{t+1}^{(2)} \approx \frac{1}{\lambda} \left( P_t^{(2)} - \frac{k}{NT} \sum_{i=1}^{NT} \frac{P_t^{(2)} G_{t(i,:)}^{Z_{t(i,:)}^l} (G_{t(i,:)}^{Z_{t(i,:)}^l})^T P_t^{(2)}}{\lambda + k (G_{t(i,:)}^{Z_{t(i,:)}^l})^T P_t^{(2)} G_{t(i,:)}^{Z_{t(i,:)}^l}} \right) \quad (50)$$

Plugging (46) into (41) yields

$$w_{t+1} \approx (P_{t+1}^{(1)} \otimes P_{t+1}^{(2)}) b_{t+1}^l = P_{t+1}^{(1)} m(b_{t+1}^l) P_{t+1}^{(2)} \quad (51)$$

where  $m(b_{t+1}^l)$  denotes reshaping the vector  $b_{t+1}^l$  into an  $N_{l-1} \times N_l$  matrix. Then, the rest deviation is similar to what we do in Section III. A. Using (43), (49) and (50), we can obtain

$$w_{t+1} \approx w_t - v(P_t^{(1)} m(\frac{1}{NT} \sum_{i=1}^{NT} G_{t(i,:)}^w) P_t^{(2)}) \quad (52)$$

where  $G_{t(i,:)}^w$  is defined as

$$G_{t(i,:)}^w = \frac{(A(S_{t(i,:}), A_{t(i,:)}) - w_t^T G_{t(i,:)}^{\Theta_t^l}) G_{t(i,:)}^{\Theta_t^l}}{(\lambda + k (X_{t(i,:)}^l)^T P_t^{(1)} X_{t(i,:)}^l)(\lambda + k (G_{t(i,:)}^{Z_{t(i,:)}^l})^T P_t^{(2)} G_{t(i,:)}^{Z_{t(i,:)}^l})}$$

Note that we don't use the average-approximation method used in Section III. A to update  $P_t^{(1)}$ ,  $P_t^{(2)}$  and  $w_t$ , since  $G_{t(i,:)}^{Z_{t(i,:)}^l}$  and  $G_{t(i,:)}^{\Theta_t^l}$  of different samples sometimes have large difference and this method will blur their difference.

Finally, the parameter of the actor output layer updated by using NPG can be defined as

$$\Theta_{t+1}^l = \Theta_t^l + \alpha m(w_{t+1}) \quad (53)$$

where  $\alpha$  is the learning rate.

### C. Optimizing Fully-connected Hidden Layer

In this subsection, we discuss the RLS optimization for fc hidden layers in the critic and actor networks. Generally, there is a nonlinear activation function in each hidden layer, which makes us difficult to derive the least squares solutions of  $\Psi_{t+1}^l$  and  $\Theta_{t+1}^l$  by using the same method introduced in Section III. A. In fact, this is the main reason why it is difficult to combine DAC and RLS.

Here we use the equivalent-gradient method, proposed in our previous work [34], to tackle this issue. For the current layer  $l$  of the critic network, we define an auxiliary least squares loss function as

$$\tilde{L}(\Psi) = \frac{1}{2NT} \sum_{n=1}^t \lambda^{t-n} \|Z_n^l - Z_n^l\|_F^2 \quad (54)$$

where  $Z_n^l$  is the corresponding desired value of  $Z_n^l = X_n^l \Psi^l$ . Then, by using the same derivation method in Section III. A,  $\Psi_{t+1}^l$  is defined as

$$\Psi_{t+1}^l \approx \Psi_t^l + \frac{k \mathbf{P}_t^l \bar{x}_t^l (\bar{z}_t^{l*} - \bar{z}_t^l)^T}{\lambda + k(\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l} \quad (55)$$

where  $\bar{z}_t^{l*}$  and  $\bar{z}_t^l$  are defined as follows

$$\bar{z}_t^{l*} = \frac{1}{NT} \sum_{i=1}^{NT} Z_{t(i,:)}^{l*} \quad (56)$$

$$\bar{z}_t^l = \frac{1}{NT} \sum_{i=1}^{NT} Z_{t(i,:)}^l \quad (57)$$

and the update rule of  $\mathbf{P}_t$  is the same as (30). Furtherly, from our previous work [34],  $\nabla_{Z_t^l} L(\Psi_t, \Theta_t)$  can be equivalently defined as

$$\nabla_{Z_t^l} L(\Psi_t, \Theta_t) = -\frac{1}{\mu NT} (Z_t^{l*} - Z_t^l) \quad (58)$$

where  $\mu > 0$  is the gradient scaling factor. Plugging (58) into (55), we finally get

$$\Psi_{t+1}^l \approx \Psi_t^l - \frac{\mu \mathbf{P}_t^l}{\lambda + k(\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l} \nabla_{\Psi_t^l} L(\Psi_t, \Theta_t) \quad (59)$$

Note that the derivation of  $\Theta_{t+1}^l$ , which is the parameter of the current fc hidden layer in the actor network, is the same as the above. For brevity, we directly give the result, namely

$$\Theta_{t+1}^l \approx \Theta_t^l - \frac{\mu \mathbf{P}_t^l}{\lambda + k(\bar{x}_t^l)^T \mathbf{P}_t^l \bar{x}_t^l} \nabla_{\Theta_t^l} L(\Psi_t, \Theta_t) \quad (60)$$

where the update rule of  $\mathbf{P}_t^l$  is also the same as (30).

### D. Optimizing Convolutional Hidden Layer

Conv layers are at the front of a CNN. They are usually used to learn spatial features of original state inputs. As defined by (8), a conv layer can be viewed as a special fc layer. That means we can also use the same RLS update rules as (59) and (60) to optimize the conv layers of critic and actor networks. However, there is a little different, since the current input  $\hat{X}_t^l$  of the conv layer  $l$  has three dimensions rather than two dimensions. In our previous work [34], we

define  $\bar{x}_t^l = \frac{1}{NTH_l W_l} \sum_{i=1}^{NT} \sum_{j=1}^{H_l W_l} \hat{X}_{t(i,:,j)}^l$  to tackle this problem. But in practice, we find that this definition will blur the difference among the input selections of different output pixels, which will worsen the performance of our algorithms.

To avoid this situation, we define

$$\bar{X}_t^l = \frac{1}{NT} \sum_{i=1}^{NT} \hat{X}_{t(i,:,:)}^l \quad (61)$$

where  $\bar{X}_t^l \in \mathcal{R}^{C_{l-1} H_l^k W_l^k \times H_l W_l}$ . On the basis, similar to that of  $\mathbf{P}_t^{l(1)} = (\mathbf{H}_t^{l(1)})^{-1}$  and  $\mathbf{P}_t^{l(2)} = (\mathbf{H}_t^{l(2)})^{-1}$  introduced in Section III. B, the recursive derivation of  $\mathbf{P}_{t+1}^l = (\frac{1}{NT} \sum_{n=1}^t \lambda^{t-n} (\hat{X}_n^l)^T \hat{X}_n^l)^{-1}$  will yield

$$\mathbf{P}_{t+1}^l \approx \frac{1}{\lambda} \left( \mathbf{P}_t^l - \frac{k}{H_l W_l} \sum_{j=1}^{H_l W_l} \frac{\mathbf{P}_t^l \bar{X}_{t(:,j)}^l (\bar{X}_{t(:,j)}^l)^T \mathbf{P}_t^l}{\lambda + k(\bar{X}_{t(:,j)}^l)^T \mathbf{P}_t^l \bar{X}_{t(:,j)}^l} \right) \quad (62)$$

Finally, we can obtain the update rules for  $\Psi_t^l$  and  $\Theta_t^l$  defined as follows

$$\Psi_{t+1}^l \approx \Psi_t^l - \tau \left( \frac{\mu H_l W_l \mathbf{P}_t^l o(\nabla_{\Psi_t^l} L(\Psi_t, \Theta_t))}{\sum_{j=1}^{H_l W_l} (\lambda + k(\bar{X}_{t(:,j)}^l)^T \mathbf{P}_t^l \bar{X}_{t(:,j)}^l)} \right) \quad (63)$$

$$\Theta_{t+1}^l \approx \Theta_t^l - \tau \left( \frac{\mu H_l W_l \mathbf{P}_t^l o(\nabla_{\Theta_t^l} L(\Psi_t, \Theta_t))}{\sum_{j=1}^{H_l W_l} (\lambda + k(\bar{X}_{t(:,j)}^l)^T \mathbf{P}_t^l \bar{X}_{t(:,j)}^l)} \right) \quad (64)$$

where  $o(\cdot)$  denotes reshaping a  $C_{l-1} \times C_l \times H_l^k \times W_l^k$  tensor into a  $C_{l-1} H_l^k W_l^k \times C_l$  matrix, and  $\tau(\cdot)$  does the reverse.

### E. RLSSA2C and RLSNA2C Algorithms

Based on the above derivation and the vanilla A2C, both RLSSA2C and RLSNA2C can be summarized in Algorithm 1, where  $L_C$  and  $L_A$  denote the total layer numbers in critic and actor networks. Note that the autocorrelation matrix  $\mathbf{P}_t^l$  in the critic network is different from that in the actor network and we use the same notation only for brevity. For RLSSA2C,  $\Theta_t^{L_A}$  is updated by using SPG and an ordinary first-order optimization algorithm. For RLSNA2C,  $\Theta_t^{L_A}$  is updated by using the compatible parameter  $w_{t+1}$ , which is updated by  $\mathbf{P}_t^{(1)}$  and  $\mathbf{P}_t^{(2)}$ . Except for those, the rest of RLSSA2C and RLSNA2C is the same. In practice, to avoid instability in training, the critic network and the actor network sometimes are joint [15], [19]. In this case, the shared layers are updated by the RLS optimization only once at each iteration.

## IV. ANALYSIS AND IMPROVEMENT

In this section, we analyze the computational complexity and the convergence of RLSSA2C and RLSNA2C in brief. In addition, we also present three practical tricks for further improving their convergence speed.

### A. Theoretical Analysis

First, we analyze the computational complexity of our proposed algorithms. Although their derivation seems complex and tedious, but in fact they are very simple and easy to implement. From (35), (52), (59), (60), (63) and (64), the RLS optimization for actor and critic networks can be viewed as a

**Algorithm 1:** RLS-Based Advantage Actor Critic

---

**1 Input:** critic parameters  $\{\Psi_0^l, P_0^l\}_{l=1}^{L_c}$ ; actor parameters  $\{\Theta_0^l\}_{l=1}^{L_A}, \{P_0^l\}_{l=1}^{L_A-1}$ , hyperparameters of the ordinary first-order algorithm or  $\{w_0, P_0^{(1)}, P_0^{(2)}, \alpha\}$ ; initial states  $\{s_{0,i}^{(1)}\}_{i=1}^N$  of  $N$  workers, discount factor  $\gamma$ , scaling factors  $k$  and  $\mu$ , forgetting factor  $\lambda$ , regularization factor  $\eta$ .

**2 for**  $t = 0, 1, 2, \dots$  **do**

**3   Excute:** let each worker run  $T$  timesteps and generate  
 $\mathcal{M}_t = \{(s_{t,i}^{(k)}, a_{t,i}^{(k)}, s'_{t,i}^{(k)}, r_{t,i}^{(k)}, d_{t,i}^{(k)})\}_{i=1,\dots,T}^{k=1,\dots,N}$ ,  
where  $a_{t,i}^{(k)} \sim \pi(a_{t,i}^{(k)} | s_{t,i}^{(k)}, \Theta_t)$  decided by the actor network

**4   Measure:** calculate the loss function by (10)

**5   Update critic network:**

- 6     update**  $\Psi_t^{l_C}, P_t^{l_C}$  in fc output layer by (35), (30)
- 7     update**  $\Psi_t^l, P_t^l$  in each fc hidden layer by (59), (30)
- 8     update**  $\Psi_t^l, P_t^l$  in each conv layer by (63), (62)

**9   Update actor network:**

- 10     update**  $\Theta_t^{L_A}$  by an ordinary first-order algorithm or  $w_t, P_t^{l(1)}, P_t^{l(2)}, \Theta_t^{L_A}$  by (52), (49), (50), (53)
- 11     update**  $\Theta_t^l, P_t^l$  in each fc hidden layer by (60), (30)
- 12     update**  $\Theta_t^l, P_t^l$  in each conv layer by (64), (62)

**13 Set**  $\{s_{t+1,i}^{(1)}\}_{i=1}^N = \{s'_{t,i}^{(T)}\}_{i=1}^N$  and **discard**  $\mathcal{M}_t$

---

special SGD optimization. For an fc hidden layer, a critic fc output layer, an actor fc output layer in RLSNA2C and a conv layer, the computational complexities of the RLS optimization are  $1 + \frac{N_{l-1}}{NT}$ ,  $1 + \frac{N_{L-1}}{NT}$ ,  $2 + \frac{N_{L-1}N_L}{NT}$  and  $1 + \frac{C_{l-1}H_l^kW_l^k}{NT H_l W_l}$  times as those of SGD, respectively. In practice, A2C generally uses 16 or 32 workers to interact with their environments for 5~20 timesteps at each iteration, that is,  $NT$  is 80~640. Thus, our RLS optimization is only several times as complex as SGD. In Section V, experimental results show that the running speeds of RLSSA2C and RLSNA2C are only 10%~30% slower than that of the vanilla A2C with RMSProp, but they are significantly faster than those of PPO and ACKTR.

Next, we analyze the convergence of our proposed algorithms. As shown in Algorithm 1, RLSSA2C and RLSNA2C are two-time-scale actor-critic algorithms. In LFARL, the convergence of this type of algorithm has been established [7], [32]. Namely, if the actor learning rate  $\alpha_t^A$  and the critic learning rate  $\alpha_t^C$  satisfy the standard Robbins-Monro condition [39] and a time-scale separation condition  $\lim_{t \rightarrow \infty} \alpha_t^A / \alpha_t^C = 0$ , actor and critic parameters will converge to asymptotic stable equilibria. However, unlike traditional actor-critic algorithms, DAC algorithms use non-linear function approximations. Thus, their convergence proofs are difficult. In recent years, there have been a few studies on this issue. Yang et al. establish the convergence of batch actor-critic with nonlinear function approximations and finite samples [40]. Liu et al.

establish nonasymptotic upper bounds of the numbers of TD and SGD iterations, and prove that a variant of PPO and TRPO with overparametrized neural networks converges to the globally optimal policy at a sublinear rate [41]. Assuming independent sampling, Wang et al. prove that neural NPG converges to a globally optimal policy at a sublinear rate [42]. Compared with the proof methods for traditional actor-critic algorithms, these methods have more assumptions and are more complex. Our algorithms are similar to the batch actor-critic algorithm studied in [40], and the RLS optimization can be viewed as a special SGD optimization. Therefore, we should establish the convergence of RLSSA2C and RLSNA2C by using the methods in [40], [41] and [40], [42], respectively. Intuitively, if actor and critic networks can be mapped into two linear function approximator with some error, the convergence proofs of RLSSA2C and RLSNA2C will be converted into the proofs of traditional actor-critic algorithms.

**B. Performance Improvement**

In Section III, to simplify the calculation, we import two scaling factors  $k$  and  $\mu$ . From (28), (29), (47), (48) and (58), they should be time-variant, so we present two new definitions for them. From (30), we can get

$$P_{t+1}^l \approx \frac{1}{\lambda} \left( P_t^l - \frac{P_t^l \bar{x}_t^l (\bar{x}_t^l)^T P_t^l}{\frac{\lambda}{k} + (\bar{x}_t^l)^T P_t^l \bar{x}_t^l} \right) \quad (65)$$

It is clear that the average scaling factor  $k$  also plays a role similar to the forgetting factor  $\lambda$ . That is, a big  $k$  will increase the forgetting rate, and vice versa. At the beginning of learning, the policy  $\pi$  is infantile and unstable, so we should select a big  $k$  to forget the historical information and emphasize the new samples for accelerating convergence. At the end of learning, the policy  $\pi$  is close to the optimal, so we should select a small  $k$  to stabilize it. In (49), (50) and (62),  $k$  also play the same role. Thus, we redefine  $k$  as

$$k_t = \max \left[ k_0 - \lfloor \frac{t}{t_\Delta} \rfloor \delta_k, k_{min} \right] \quad (66)$$

where  $k_0$ ,  $t_\Delta$ ,  $\delta_k$  and  $k_{min}$  denote the initial value, the update interval, the decay size and the lower bound of  $k$ , respectively. From (59), (60), (63) and (64), the gradient scaling factor  $\mu$  is a part of the learning rate. A big  $\mu$  can accelerate convergence, but it will also cause  $\pi$  to fall into the local optimum. Thus,  $\mu$  should gradually decay to a steady value. Here, we redefine it as

$$\mu_t = \max \left[ \mu_0 - \lfloor \frac{t}{t_\Delta} \rfloor \delta_\mu, \mu_{min} \right] \quad (67)$$

where  $\mu_0$ ,  $\delta_\mu$  and  $\mu_{min}$  denote the initial value, the decay size and the lower bound of  $\mu$ , respectively. In fact,  $\mu_t$  is different for each layer. In a DNN, deeper layers are more likely to suffer from the gradient vanishing, so we suggest that users choose a big  $\mu_0$  for these layers.

In addition, it is well known that the vanilla SGD can be accelerated by the momentum method [43]. Our RLS optimization is a special SGD, so we can also use this method

to accelerate it. Similar to in our previous work [34], (35) can be redefined as

$$\Phi_{t+1}^l \approx \beta \Phi_t^l - \frac{\mathbf{P}_t^l \nabla_{\Psi_t^l} L(\Psi_t, \Theta_t)}{\lambda + k(\bar{\mathbf{x}}_t^l)^T \mathbf{P}_t^l \bar{\mathbf{x}}_t^l} \quad (68)$$

$$\Psi_{t+1}^l \approx \Psi_t^l + \Phi_{t+1}^l \quad (69)$$

where  $\Phi_t^l$  denote the velocity matrix in the  $l^{th}$  layer, and  $\beta$  is the momentum factor. (59), (60), (63) and (64) can also be redefined as the similar forms. Note that we only suggest RLSSA2C to use this method, since we find that it will worsen the RLSNA2C's stabilities empirically.

## V. EXPERIMENTAL RESULTS

In this section, we will compare our algorithms against the vanilla A2C with RMSProp (called RMSA2C) for evaluating the sample efficiency, and compare them against RMSA2C, PPO and ACKTR for evaluating the computational efficiency. We first test these algorithms on 40 discrete control games in the Atari 2600 environment, and then test them on 11 continuous control tasks in the MuJoCo environment.

### A. Discrete Control Evaluation

Atari 2600 is the most famous discrete control benchmark platform for evaluating DRL. It is comprised of a lots of highly diverse games, which have high-dimensional observations with raw pixels. In this set of experiments, we select 40 games from the Atari environment for performance evaluation. For each game, the state is a  $4 \times 84 \times 84$  normalized image.

All tested algorithms have the same model architecture, which is defined in [15]. It has 3 shared conv layers, 1 shared fc hidden layer, 1 separate fc critic output layer and 1 separate fc actor output layer. The first conv layer has 32  $8 \times 8$  kernels with stride 4, the second conv layer has 64  $4 \times 4$  kernels with stride 2, the third conv layer has 32  $3 \times 3$  kernels with stride 1, and the fc hidden layer has 512 neurons. All hidden layers use ReLU activation functions, the fc critic output layer uses the Identity activation function to predict the state-value function, and the fc actor output layer uses the Softmax activation function to represent the policy. The parameter of each layer is initialized with the default settings of PyTorch. All algorithms use the same loss function defined by (10), where the discount factor  $\gamma = 0.99$  and the entropy regularization factor  $\eta = 0.01$ . At each iteration step, each algorithm lets 32 parallel workers run 5 timesteps, and uses the generated minibatch including 160 samples for training. All workers use an Intel Core i7-9700K CPU for trajectory sampling, and use a Nvidia RTX 2060 GPU for accelerating the optimization computation. To avoid gradient exploding, all parameter gradients are clipped by the  $L_2$  norm with 0.5.

Besides the above common settings, individual settings of each algorithm are summarized as follows: 1) For RMSA2C, the learning rate  $\epsilon$ , decay factor  $\rho$  and small constant  $\delta$  in RMSProp are set to 0.00025, 0.99 and 0.00005, respectively. 2) For RLSSA2C, all initial autocorrelation matrices are set to Identity matrices, the forgetting factor  $\lambda$  and momentum factor  $\beta$  are set to 1 and 0.5, the hyperparameters  $k_0$ ,  $\delta_k$ ,

$k_{min}$ ,  $\mu_0$ ,  $\delta_\mu$ ,  $\mu_{min}$ ,  $t_\Delta$  of scaling factors  $k_t$  and  $\mu_t$  are set to 0.1, 0.02, 0.01, 5, 0.1, 1 and 5000, and the actor output layer uses the same RMSProp as that in RLSNA2C. Note that  $\mu_t$  is only used for conv layers but is fixed to 1 for all fc layers.

3) For RLSNA2C, the momentum factor  $\beta$  is set to 0.0,  $\mathbf{P}_0^{(1)}$  and  $\mathbf{P}_0^{(2)}$  are also set to Identity matrices, and the learning rate  $\alpha$  of the actor output layer is initialized to 0.01 and decays by 0.002 per 5000 timesteps to 0.001. The other settings are the same as those in RLSSA2C. 4) For PPO and ACKTR, we directly use Kostrikov's source code and settings [25]. Note that the settings of RMSA2C are selected from [15], and the settings of RLSSA2C and RLSNA2C are obtained by tuning Pong, Breakout and StarGunner games.

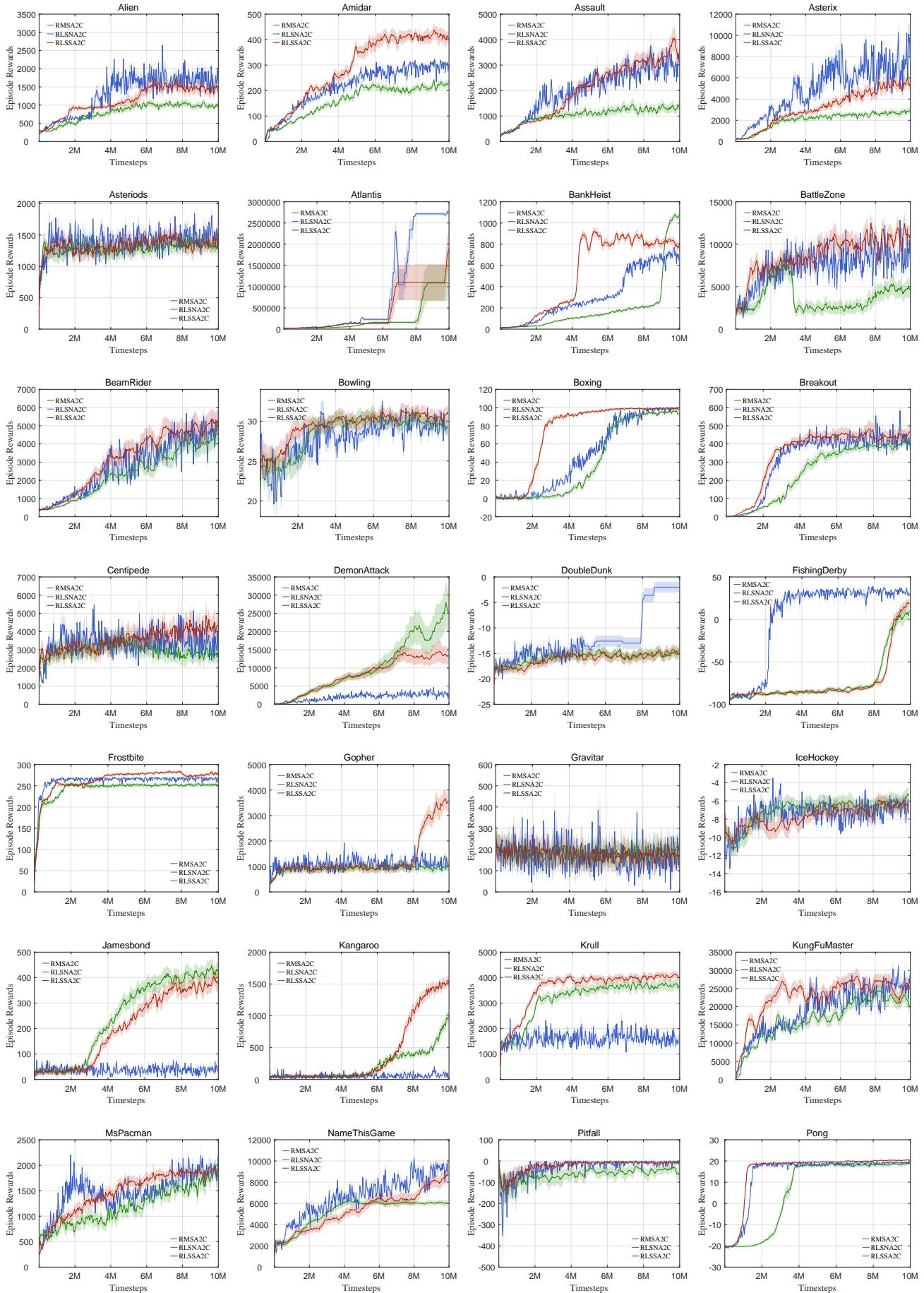
The convergence comparison of our algorithms against RMSA2C on 40 Atari games trained for 10 million timesteps is shown in Fig. 1. It is clear that RLSSA2C and RLSNA2C outperform RMSA2C on most games. Among these three algorithms, RLSSA2C has the best convergence performance (i.e., sample efficiency) and stability. Compared with RMSA2C, RLSSA2C wins on 30 games. On Alien, Amidar, Assault, Asterix, BattleZone, Boxing, Breakout, Kangaroo, Krull, Kung-FuMaster, MsPacman, Pitfall, Pong, Qbert, Seaquest and Tennis games, RLSSA2C is significantly superior to RMSA2C in terms of convergence speed and convergence quality. Compared with RMSA2C, RLSNA2C also wins on 30 games. On Asterix, Atlantis, DoubleDunk, FishingDerby, NameThisGame, Pong, Riverraid, Seaquest, UpNDown and Zaxxon games, RLSNA2C performs very well. But compared with RLSSA2C, it is not very stable on some games.

In Table I, we present the last 100 average episode rewards of our algorithms and RMSA2C on 40 Atari games trained for 10 million timesteps. From Table I, RLSSA2C and RLSNA2C obtain the highest rewards on 19 and 15 games respectively, but RMSA2C only wins on 6 games. Notably, on Assault, BattleZone, Gopher and Q-bert games, RLSSA2C respectively acquires 2.6, 2.4, 3.6, and 3.1 times scores as much as RMSA2C. On Asterix, Atlantis, FishingDerby, UpNDown and Zaxxon games, RLSNA2C respectively acquired 2.8, 2.5, 2.7, 4.8, and 45 times scores as much as RMSA2C.

The running speed comparison of our algorithms against RMSA2C, PPO and ACKTR on six Atari games is shown in Table II. Among these five algorithms, RMSA2C has the highest computational efficiency, RLSNA2C and RLSSA2C are listed 2nd and 3rd respectively, and PPO is listed last. In detail, RLSSA2C is only 28.1% slower than RMSA2C, but is 592.7% and 31.3% faster than PPO and ACKTR. RLSNA2C is only 27.7% slower than RMSA2C, but is 596.3% and 31.9% faster than PPO and ACKTR. By using Kostrikov's source code to test PPO and ACKTR, we find that our algorithms have the similar performance. Therefore, our algorithms achieve a better trade-off between high convergence performance and low computational cost.

### B. Continuous Control Evaluation

MuJoCo is a high-dimensional continuous control benchmark platform. In this set of experiments, we select 11 tasks from the MuJoCo environment for performance evaluation.



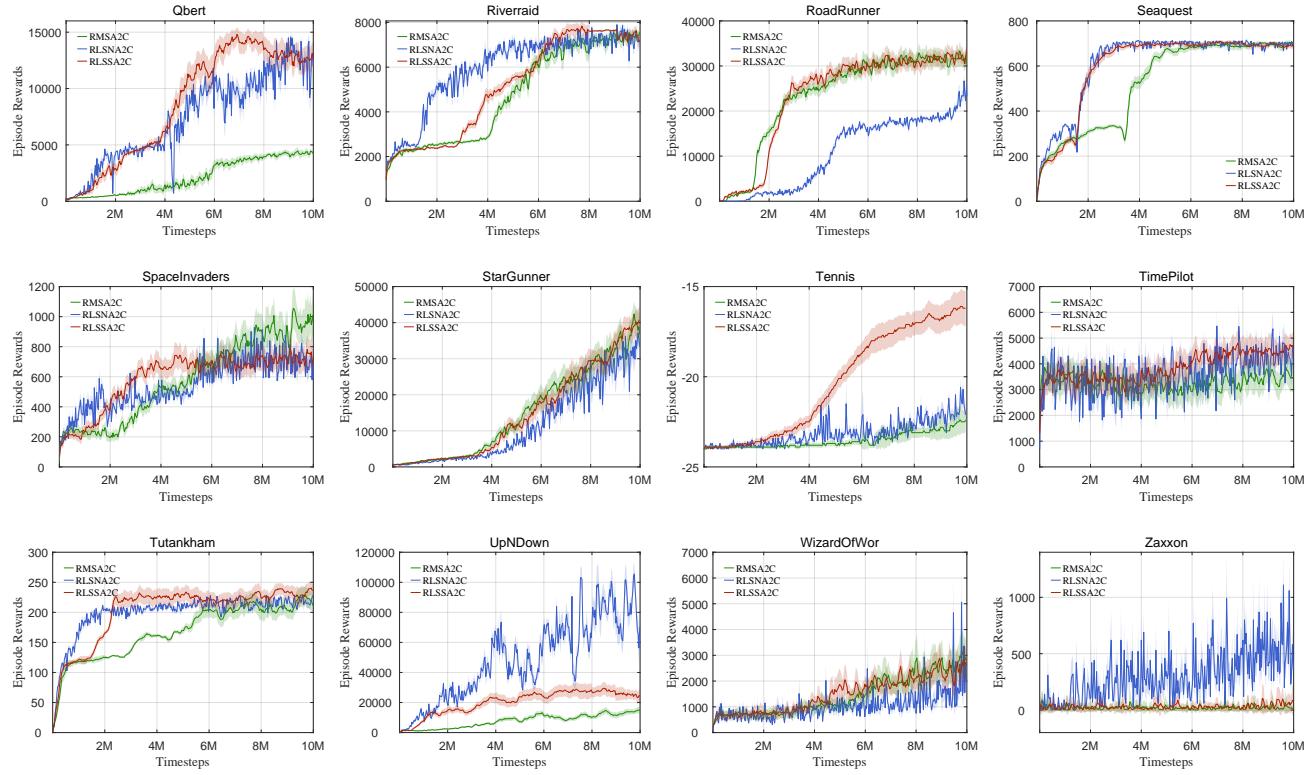


Fig. 1. Convergence comparison of our algorithms against RMSA2C on 40 Atari games trained for 10M timesteps.

TABLE I  
LAST 100 AVERAGE EPISODE REWARDS OF OUR ALGORITHMS AND RMSA2C ON 40 ATARI GAMES TRAINED FOR 10M TIMESTEPS

Game	RMSA2C	RLSSA2C	RLSNA2C	Game	RMSA2C	RLSSA2C	RLSNA2C
Alien	975.8	1375.7	<b>1675.0</b>	Jamesbond	<b>423.5</b>	379.0	30.0
Amidar	234.9	<b>405.6</b>	301.1	Kangaroo	992.0	<b>1512.0</b>	60.0
Assault	1281.7	<b>3346.9</b>	3136.4	Krull	7327.0	<b>7996.3</b>	3715.0
Asterix	2937.0	5138.5	<b>8140.0</b>	KungFuMaster	20427.0	25224.0	<b>29000.0</b>
Asteroids	1427.4	1344.8	<b>1626.0</b>	MsPacman	1846.9	1916.4	<b>2195.0</b>
Atlantis	1085676.0	1885303.0	<b>2747970.0</b>	NameThisGame	6054.1	<b>8592.8</b>	8555.0
BankHeist	<b>1077.1</b>	789.5	662.0	Pitfall	-63.4	-6.7	<b>-2.8</b>
BattleZone	4620.0	<b>11200.0</b>	8100.0	Pong	19.2	<b>20.5</b>	19.6
BeamRider	4602.9	5225.5	<b>5313.0</b>	Qbert	4267.5	<b>13064.5</b>	12020.0
Bowling	29.1	<b>30.9</b>	29.3	Riverraid	<b>7572.3</b>	7193.4	7125.0
Boxing	93.3	97.2	<b>100.0</b>	RoadRunner	30705.0	<b>33160.0</b>	23300.0
Breakout	389.9	<b>460.5</b>	445.9	Seaquest	1754.8	1728.8	<b>1756.0</b>
Centipede	2875.6	<b>4300.1</b>	2950.9	SpaceInvaders	<b>1001.2</b>	677.4	591.0
DemonAttack	<b>25309.3</b>	13252.4	2619.5	StarGunner	36820.0	<b>40808.0</b>	33280.0
DoubleDunk	-14.4	-15.1	<b>-2.0</b>	Tennis	-22.4	<b>-16.2</b>	-22.1
FishingDerby	10.2	18.3	<b>27.8</b>	TimePilot	3471.0	<b>4648.0</b>	4480.0
Frostbite	250.7	<b>277.2</b>	267.0	Tutankham	211.9	<b>236.5</b>	227.7
Gopher	986.0	<b>3585.0</b>	1088.0	UpNDown	14666.5	24466.6	<b>69848.0</b>
Gravitar	204.5	176.5	<b>240.0</b>	WizardOfWor	<b>2770.0</b>	2626.0	1920.0
IceHockey	-11.2	<b>-6.5</b>	-7.4	Zaxxon	8.0	87.0	<b>360.0</b>

TABLE II  
RUNNING SPEED COMPARISON ON SIX ATARI GAMES  
( Timesteps / Second )

Game	RMSA2C	PPO	ACKTR	RLSSA2C	RLSNA2C
Alien	2897	320	1690	2056	2021
Breakout	3033	321	1668	2057	2150
Pong	3265	324	1754	2411	2402
SpaceInvaders	3124	328	1731	2293	2306
StarGunner	3436	341	1794	2515	2512
Zaxxon	3201	336	1750	2301	2311
Mean	3159	328	1731	2272	2284

In contrast to in Atari, the state in MuJoCo is a multiple dimensional vector, and the action space is continuous.

All tested algorithms also use two disjoint FNNs defined in [15]. One is the critic network, the other is the actor network. Both networks have two same fc hidden layers with 64 Tanh neurons. The critic output layer is a linear fc layer for predicting the value function. The actor network uses a linear fc layer with bias to represent the mean and the standard deviation of the Gaussian policy [24]. All settings for five tested algorithms are the same as those in Section V. A, except for  $\alpha = 0.001$  in RLSNA2C.

The convergence comparison of our algorithms against RMSA2C on 11 MuJoCo tasks trained for 10 million timesteps is shown in Fig. 2. It is clear that RLSSA2C and RLSNA2C outperform RMSA2C on almost all tasks. Among these three algorithms, RLSSA2C has the best convergence performance and stability. Compared with RMSA2C, RLSSA2C wins on 10 tasks. On Ant, HalfCheetah, Hopper, Pusher, Reacher, Striker, Swimmer, Thrower and Walker2d tasks, RLSSA2C is significantly superior to RMSA2C in terms of convergence speed and convergence quality. Compared with RMSA2C, RLSNA2C also wins on 9 tasks. On Pusher, Reacher, and Swimmer tasks, RLSNA2C performs very well.

In Table III, we present the top 10 average episode rewards of our algorithms and RMSA2C on 11 MuJoCo tasks trained for 10 million timesteps. From Table III, RLSSA2C and RLSNA2C obtain the highest rewards on 7 and 2 tasks respectively, but RMSA2C only wins on 1 task.

The running speed comparison of our algorithms against RMSA2C, PPO and ACKTR on six MuJoCo tasks is shown in Table IV. Among these five algorithms, RMSA2C has the highest computational efficiency, RLSSA2C and RLSNA2C are listed 2nd and 3rd respectively, and PPO is listed last. In detail, RLSSA2C is only 8.9% slower than RMSA2C, but is 1851.9% and 30.1% faster than PPO and ACKTR. RLSNA2C is only 12.0% slower than RMSA2C, but is 1785.9% and 25.7% faster than PPO and ACKTR.

Obviously, RLSSA2C and RLSNA2C also perform very well on MuJoCo tasks. In summary, our both algorithms have better sample efficiency than RMSA2C, and have higher computational efficiency than PPO and ACKTR.

## VI. CONCLUSION

In this paper, we proposed two RLS-based A2C algorithms, called RLSSA2C and RLSNA2C. To the best of our knowledge, they are the first RLS-based DAC algorithms. Our both algorithms use the RLS method to train the critic network and hidden layers of the actor network. Their main difference is their policy learning. RLSSA2C uses SPG and an ordinary first-order gradient descent algorithm to learn the policy parameter, but RLSNA2C uses NPG, the Kronecker-factored approximation and the RLS method to learn the compatible parameter and the policy parameter. We also analyzed the complexity and convergence of our both algorithms, and presented three tricks for further accelerating their convergence. We tested our both algorithms on 40 Atari discrete control games and 11 MuJoCo continuous control tasks. Experimental results show that our both algorithms have better sample efficiency than RMSA2C on most games or tasks, and have higher computational efficiency than PPO and ACKTR. In future work, we will try to establish the convergence of our both algorithms and improve the stability of RLSNA2C.

## REFERENCES

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT press, 2018.
- [2] W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley, 2007.
- [3] D. Silver *et al.*, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, pp. 354-359, 2017.
- [4] K. Zhu and T. Zhang, “Deep reinforcement learning based mobile robot navigation: A review,” *Tsinghua Sci. Tech.*, vol. 26, no. 5, pp. 674-691, Oct. 2021.
- [5] B. R. Kiran *et al.*, “Deep reinforcement learning for autonomous driving: A survey,” *IEEE Trans. Intell. Trans. Syst.*, pp. 1-18, 2021.
- [6] A. Tsantekidis, N. Passalis, A. S. Toufa, K. Saitas-Zarkias, S. Chairitsanidis and A. Tefas, “Price trailing for financial trading using deep reinforcement learning,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 7, pp. 2837-2846, 2021.
- [7] X. Xu, C. Lian, L. Zuo and H. He, “Kernel-based approximate dynamic programming for real-time online learning control: An experimental study,” *IEEE Trans. Control. Syst. Technol.*, vol. 22, no. 1, pp. 146-156, Jan. 2014.
- [8] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529-533, 2015.
- [9] V. Hasselt, A. Guez and D. Silver, “Deep reinforcement learning with double Q-learning,” in *Proc. 30th AAAI Conf. Artif. Intell.*, Feb. 2016, pp. 2049-2100.
- [10] M. Williams and L. Sergey, “Guided policy search as approximate mirror descent,” in *Proc. 30th Adv. neural inf. proces. syst.*, 2016, pp. 4015-4023.
- [11] N. Heess, T. Degris, D. Wierstra and M. Riedmiller, “Deterministic policy gradient algorithms,” in *Proc. 31th Int. Conf. Mach. Learn. (ICML)*, 2014, pp. 387-395.
- [12] T. P. Lillicrap *et al.*, “Continuous control with deep reinforcement learning,” in *Proc. 4th Int. Conf. Learn. Represent. (ICLR)*, 2019.
- [13] N. Heess, J. J. Hunt, T. P. Lillicrap and D. Silver, “Memory-based control with recurrent neural networks,” 2015. [Online]. Available: <https://arxiv.org/abs/1512.04455>
- [14] S. Fujimoto, H. V. Hoof and D. Meger, “Addressing function approximation error in actor-critic methods,” in *Proc. 35th Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 1587-1596.
- [15] V. Mnih *et al.*, “Asynchronous methods for deep reinforcement learning,” in *Proc. 33th Int. Conf. Mach. Learn. (ICML)*, 2016, pp. 2850-2869.
- [16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, “Proximal policy optimization algorithms,” 2017. [Online]. Available: <https://arxiv.org/abs/1707.06347>
- [17] J. Schulman, S. Levine, P. Moritz, M. I. Jordan and P. Abbeel, “Trust region policy optimization,” in *Proc. 32th Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 1889-1897.

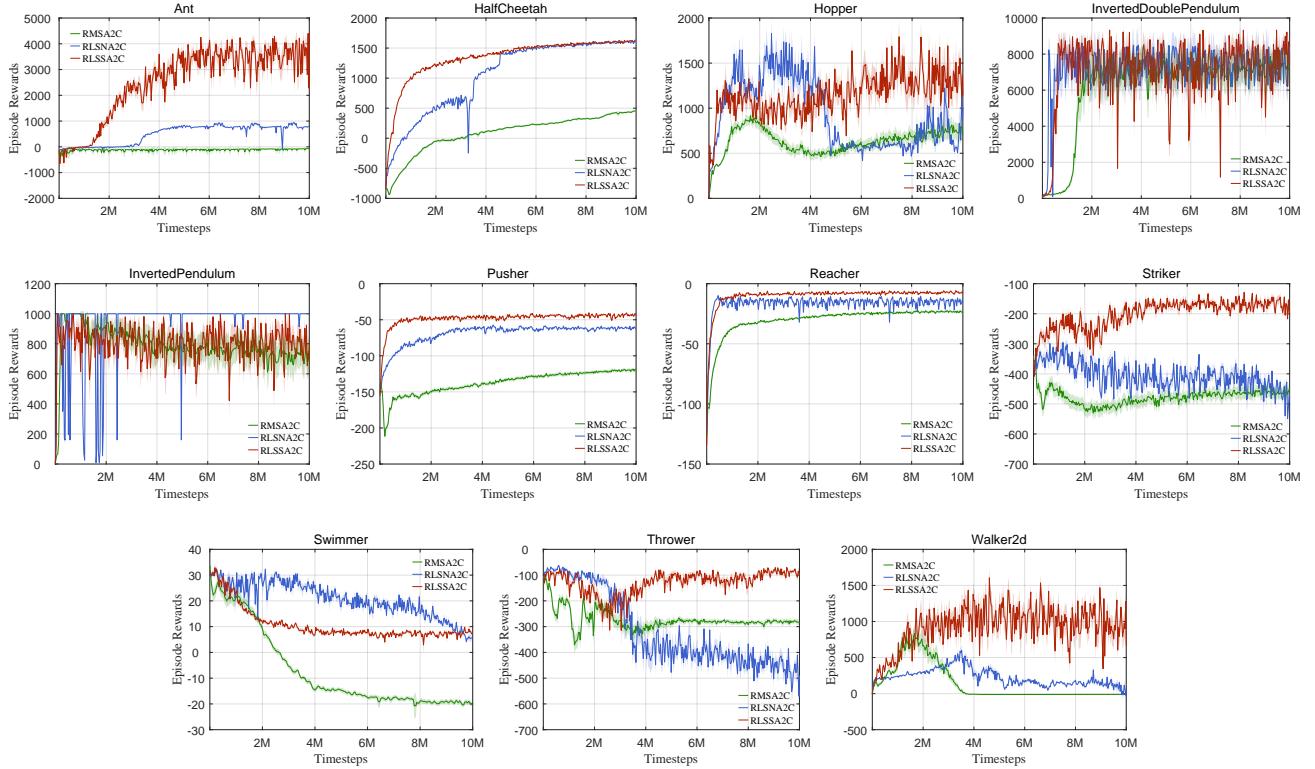


Fig. 2. Convergence comparison of our algorithms against RMSA2C on 11 MuJoCo tasks trained for 10M timesteps.

TABLE III  
TOP 10 AVERAGE EPISODE REWARDS OF OUR ALGORITHMS AND RMSA2C ON 11 MUJOCO TASKS TRAINED FOR 10M TIMESTEPS

Game	RMSA2C	RLSSA2C	RLSNA2C
Ant	-62.0	<b>4607.0</b>	941.5
HalfCheetah	454.6	<b>1630.3</b>	1625.4
Hopper	921.9	2068.9	<b>2133.6</b>
InvertedDoublePendulum	7957.0	<b>9333.4</b>	8500.0
InvertedPendulum	1000.0	1000.0	1000.0
Pusher	-117.7	<b>-39.0</b>	-56.6

Game	RMSA2C	RLSSA2C	RLSNA2C
Reacher	-21.9	<b>-4.9</b>	-9.2
Striker	-336.1	<b>-124.7</b>	-264.3
Swimmer	<b>33.6</b>	32.9	32.3
Thrower	-106.2	-68.5	<b>-59.3</b>
Walker2d	1682.0	<b>3306.9</b>	1363.6

TABLE IV  
RUNNING SPEED COMPARISON ON SIX MUJOCO TASKS  
(Timesteps/Second)

Game	RMSA2C	PPO	ACKTR	RLSSA2C	RLSNA2C
Ant	5911	372	4584	5646	5380
Hopper	7735	375	5574	7160	6909
InvertedPendulum	10076	384	6633	8997	8399
Pusher	7873	368	5558	7185	7073
Swimmer	9033	384	6068	7996	7927
Walker2d	7716	373	5438	7052	6855
Mean	8057	376	5643	7339	7091

[18] Y. Wu, E. Mansimov, S. Liao, A. Radford and J. Schulman, “Openai baselines : acktr and a2c.” [Online]. Available: <https://openai.com/blog/baselines-acktr-a2c/>

- [19] Z. Wang *et al.*, “Sample efficient actor-critic with experience replay,” in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, 2017.
- [20] X. Gong, J. Yu, S. Lä and H. Lu, “Actor-critic with familiarity-based trajectory experience replay,” *Inf. Sci.*, vol. 582, pp. 633-647, 2022.
- [21] T. Haarnoja, A. Zhou, P. Abbeel and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *Proc. 35th Int. Conf. Mach. Learn. (ICML)*, Jul. 2018, pp. 2976-2989.
- [22] J. Byun, B. Kim and H. Wang, “Proximal policy gradient: PPO with policy gradient,” Oct. 2020. [Online]. Available: <https://arxiv.org/abs/2010.09933>
- [23] J. Martens and R. Grosse, “Optimizing neural networks with kronecker-factored approximate curvature,” in *Proc. 32th Int. Conf. Mach. Learn. (ICML)*, Jul. 2015, pp. 2398-2407.
- [24] Y. Wu, E. Mansimov, S. Liao, R. Grosse and J. Ba, “Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation,” in *Proc. 31th Adv. neural inf. proces. syst.*, 2017, pp. 5280-5289.
- [25] I. Kostrikov, “PyTorch implementations of reinforcement learning algorithms,” *GitHub repository*, 2018. [Online]. Available: <https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail>
- [26] T. Tielemans and G. Hinton, “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Netw.*

- Mach. Learn.*, vol. 4, pp. 26-30, 2012.
- [27] S. J. Bradtko and A. G. Barto, "Linear least-squares algorithms for temporal difference learning," *Mach. Learn.*, vol. 22, pp. 33-57, Mar. 1996.
- [28] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Proc. 13th Adv. neural inf. proces. syst.*, 2000, pp. 1008-1014.
- [29] X. Xu, H. He and D. Hu, "Efficient reinforcement learning using recursive least-squares methods," *J. Artif. Intell. Res.*, vol. 16, no. 1, pp. 259-292, Jan. 2002.
- [30] J. Peters, S. Vijayakumar and S. Schaal, "Reinforcement learning for humanoid robotics," in *Proc. 3th IEEE-RAS Int. Conf. Hum.-Rob.*, 2003, pp. 1-20.
- [31] J. Park, J. Kim and D. Kang, "An RLS-based natural actor-critic algorithm for locomotion of a two-linked robot arm," in *Proc. Lect. Notes Comput. Sci.*, 2005, pp. 65-72.
- [32] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh and M. Lee, "Natural actor-critic algorithms," *Automatica*, vol. 45, no. 11, pp. 2471-2482, Nov. 2009.
- [33] L. Li, D. Li, T. Song and X. Xu, "Actor-critic learning control based on  $\ell_2$ -regularized temporal-difference prediction with gradient correction," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 12, pp. 5899-5909, Dec. 2018.
- [34] C. Zhang, Q. Song, H. Zhou, Y. Ou, H. Deng and T. Yang, "Revisiting recursive least squares for training deep neural networks," 2021. [Online]. Available: <https://arxiv.org/abs/2109.03220v1>
- [35] R. S. Sutton, D. McAllester, S. Singh and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. 12th Adv. neural inf. proces. syst.*, Nov. 1999, pp. 1057-1063.
- [36] P. Chou, D. Maturana and S. Scherer, "Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, 2017, pp. 1386-1396.
- [37] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, no. 7-9, pp. 1180-1190, Mar. 2008.
- [38] K. B. Petersen and M. S. Pedersen, *The Matrix Cookbook*. Technical University of Denmark, 2012.
- [39] H. Robbins and S. Monro, "A stochastic approximation method," *Annal. Math. Stat.*, vol. 22, no. 3, pp. 400-407, 1951.
- [40] Z. Yang, K. Zhang, M. Hong and T. Basar, "A finite sample analysis of the actor-critic algorithm," in *Proc. 57th IEEE Conf. Decis. Control*, Dec. 2018, pp. 2759-2764.
- [41] B. Liu, Q. Cai, Z. Yang and Z. Wang, "Neural proximal/trust region policy optimization attains globally optimal policy," in *Proc. 33th Adv. neural inf. proces. syst.*, 2019.
- [42] L. Wang, Q. Cai, Z. Yang and Z. Wang, "Neural policy gradient methods: Global optimality and rates of convergence," 2019. [Online]. Available: <https://arxiv.org/abs/1909.01150>
- [43] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.