ACERAC: Efficient Reinforcement Learning in Fine Time Discretization

Jakub Łyskawa and Paweł Wawrzyński, Member, IEEE

Abstract—One of the main goals of reinforcement learning (RL) is to provide a way for physical machines to learn optimal behavior instead of being programmed. However, effective control of the machines usually requires fine time discretization. The most common RL methods apply independent random elements to each action, which is not suitable in that setting. It is not feasible because it causes the controlled system to jerk and does not ensure sufficient exploration since a single action is not long enough to create a significant experience that could be translated into policy improvement. In our view, these are the main obstacles that prevent the application of RL in contemporary control systems. To address these pitfalls, in this article, we introduce an RL framework and adequate analytical tools for actions that may be stochastically dependent in subsequent time instances. We also introduce an RL algorithm that approximately optimizes a policy that produces such actions. It applies experience replay (ER) to adjust the likelihood of sequences of previous actions to optimize expected $n$-step returns that the policy yields. The efficiency of this algorithm is verified against four other RL methods [continuous deep advantage updating (CDAU), proximal policy optimization (PPO), soft actor–critic (SAC), and actor–critic with ER (ACER)] in four simulated learning control problems (Ant, HalfCheetah, Hopper, and Walker2D) in diverse time discretization. The algorithm introduced here outperforms the competitors in most cases considered.

Index Terms—Actor–critic, experience replay (ER), fine time discretization, reinforcement learning (RL).

I. INTRODUCTION

The subject of this article is reinforcement learning (RL) [1]. This field offers methods of learning to make sequential decisions in dynamic environments. One application of such methods is the literal implementation of “machine learning,” i.e., enabling machines and software to learn optimal behavior instead of being programmed.

The usual goal of RL methods is to optimize a policy that samples an action based on the current state of a learning agent. The only stochastic dependence between subsequent actions is through state transition: the action moves the agent to another state, which determines the distribution of another action. The main analytical tools in RL are based on this lack of other dependence between actions. For example, for a given policy, its value function expresses the expected sum of discounted rewards that the agent may expect starting from a given state. The sum of rewards does not depend on actions taken before the given state has been reached. Hence, only the given state and the policy matter.

Lack of dependence between actions beyond state transition leads to the following difficulties. In the physical implementation of RL, e.g., in robotics, the lack of dependence usually means that white noise is added to control actions. However, this makes control discontinuous and subject to constant rapid changes. In addition, this is often impossible to implement since electric motors to execute these actions cannot change their output too quickly. Even if such control is possible, it requires large amounts of energy, makes the controlled system shake, and exposes it to damage.

Control frequency for real-life control systems can be much higher than that of simulated environments for which RL methods are designed. The typical frequency of the control signal for environments commonly used as benchmarks for RL algorithms ranges from 20 to 60 Hz [2], while the control frequency considered for real-life robots is ten times higher, from 200 to 500 Hz [3], and can be even higher, up to 1000 Hz [4]. Therefore, finer time discretization should be considered to make RL more suitable for automation and robotics.

The lack of dependence between actions beyond state transition may also reduce the efficiency of learning as follows. Each action is then an independent random experiment that leads to policy improvement. However, due to the limited accuracy of (action)value function approximation, the consequences of a single action may be hard to recognize. The finer the time discretization, the more serious this problem becomes. The consequences of a random experiment distributed over several time instants could be more tangible and, thus, easier to recognize.

In addition, fine time discretization makes policy evaluation more difficult, as it requires accounting for more distant rewards. Technically, the discount factor needs to be larger, which makes learning more difficult for most RL algorithms [5]. The above problems are serious enough to prevent RL from wide applicability of RL in real-life control systems.

To avoid the above pitfalls, we introduce in this article a framework in which an action is both a function of state and a stochastic process whose subsequent values are dependent. In particular, these subsequent values can be autocorrelated, which makes the resulting actions close to one another. A part...
of action trajectory can create a distributed-in-time random experiment that leads to policy improvement. An RL algorithm is also introduced that optimizes a policy based on the above principles.

The contribution of this article may be summarized by the following points.

1) A framework is introduced here in which a policy produces actions based on the states and values of a stochastic process. This framework is suited for the application of RL to optimization of control in physical systems, e.g., in robots.

2) An actor–critic with experience replay (ACER) and Autocorrelated aCtions (ACERAC) algorithm, based on actor–critic structure and experience replay (ER), is introduced, which approximately optimizes a policy in the aforementioned framework.

3) An extensive study is described here with four benchmark learning control problems (Ant, Half-Cheetah, Hopper, and Walker2D) at diverse time discretization. The performance of the ACERAC algorithm is compared using these problems with state-of-the-art RL methods.

This article extends [6] in several directions. We introduce here the notion of adjusted noise, which is the input to the noise-value function. Also, when manipulating the policy parameter, the value of the noise-value function at the end of the action sequence is taken into account. The experimental study of the resulting algorithm is almost entirely new.

The rest of this article is organized as follows. The problem considered here is formulated in Section II. Section III overviews related literature. Section IV introduces a policy that produces autocorrelated actions along with tools for its analysis. Section V introduces the ACERAC algorithm that approximately optimizes this policy. Section VI presents simulations that compare the algorithm presented with state-of-the-art RL methods. Section VII concludes this article.

II. PROBLEM FORMULATION

We consider here the standard Markov decision process (MDP) model [1] in which an agent operates in discrete time $t = 1, 2, \ldots$. At time $t$, the agent finds itself in a state, $s_t \in \mathcal{S}$, takes an action, $a_t \in \mathcal{A}$, receives a reward, $r_t \in \mathbb{R}$, and is transited to another state, $s_{t+1} \sim P_s(\cdot|s_t, a_t)$, where $P_s$ is a fixed but unknown conditional probability.

The goal of the agent is to learn to designate actions to be able to expect at each $t$ the highest discounted rewards in the future. To ensure exploration, there is usually a random component introduced into the action selection.

We mainly consider the application of the MDP model to control physical devices. Therefore, we assume that both $\mathcal{S}$ and $\mathcal{A}$ are spaces of vectors of real numbers [7]. We also assume fine time discretization typical for such applications, which means that, designating actions, the agent should account for rewards that are quite distant in terms of discrete-time steps in the future. This translates into a discount parameter close to 1, e.g., $\gamma \in (0.995, 1)$. We require the reasons for the instability of learning with such a large $\gamma$ [5] to be overcome.

To ensure applicability to the control of physical machines, we require that the actions should generally be close for subsequent $t$, even if they are random. Also, the learning should be efficient in terms of the amount of experience needed to optimize the agent's behavior.

III. RELATED WORK

A general way to make subsequent actions close is the autocorrelation of the randomness on which these actions are based. Efficiency in terms of experience needed can be provided by ER. We focus on these concepts in the literature review in the following.

A. Stochastic Dependence Between Actions

An autocorrelated stationary stochastic process, now referred to as Ornstein–Uhlenbeck (OU) process, was analyzed in [8]. This process is the only autocorrelated Gaussian stochastic process that has the Markov property [9].

A policy with autocorrelated actions was analyzed in [10]. This policy was optimized by a standard RL algorithm that did not account for the dependence of actions. In [11], a policy was analyzed whose parameters were incremented by the OU stochastic process. Essentially, this resulted in autocorrelated random components of actions. In [12], a policy is analyzed, which produced an action that was the sum of the OU noise and a deterministic function of the state. However, no learning algorithm was presented in this article, which accounted for the specific properties of this policy.

B. Reinforcement Learning for Fine Time Discretization

In [13], RL in arbitrarily fine time discretization is analyzed. It is proven that RL based on the action-value function cannot be effective when time discretization becomes sufficiently fine, and note the importance of the dependence of the action noise in the next time steps. In the aforementioned work, the RL algorithm called deep advantage updating (DAU) for discrete actions and its variant for continuous actions [continuous DAU (CDAU)] are introduced. These methods are based on estimating the advantage function and are presented as immune to time discretization. They are based on deep Q-network (DQN) [14] and deep deterministic policy gradient (DDPG) [15] algorithms, respectively, and use the OU process as autocorrelated noise.

Integral RL (IRL) is an approach to learning control policies for continuous-time environments. IRL is based on the assumption that the control problem can be divided into a hierarchy of control loops [16]. This assumption is usually not satisfied in challenging tasks, and thus, IRL is not applicable to tasks with any state transition dynamics, only those belonging to a certain relatively narrow class [17].

C. Reinforcement Learning With Experience Replay

The actor–critic architecture for RL was first introduced in [18]. Approximators were applied to this structure for the first time in [19]. Basic online RL algorithms use consecutive events of the agent-environment interaction to update the policy. To boost the efficiency of these algorithms, ER can
be applied, i.e., storing the events in a database, sampling, and using them for policy updates several times per each actual event [20]. ER was combined with the actor–critic architecture for the first time in [21].

However, the application of ER to actor–critic encounters the following problem. The learning algorithm needs to estimate the quality of a given policy based on the consequences of actions that were registered when a different policy was in use. Importance sampling estimators are designed to do that, but they can have arbitrarily large variances. In [21], the problem was addressed with truncating density ratios present in those estimators. In [22], specific correction terms were introduced for that purpose.

Another approach to the aforementioned problem is to prevent the algorithm from inducing a policy that differs too much from the one tried. This idea was first applied in conservative policy iteration [23]. It was further extended in trust region policy optimization [24]. This algorithm optimizes a policy with the constraint that the Kullback–Leibler (K-L) divergence between this policy, and the one being tried should not exceed a given threshold. The K-L divergence becomes an additive penalty in proximal policy optimization algorithms, namely, proximal policy optimization (PPO)-penalty and PPO-Clip [25].

A way to avoid the problem of estimating the quality of a given policy based on the one tried is to approximate the action in the original version of DDPG, this was not adapted to this fact in any specific way. Added to the action in the original version of DDPG, this idea was first applied in [20]. ER was combined with the actor–critic architecture for that purpose.

In [22], specific correction terms were introduced for that purpose.

In the following, we analyze an example of \( \left( \xi_t \right) \) that meets the above requirements.

### A. Ornstein–Uhlenbeck process \( \left( \xi_t \right) \)

Let \( a \in [0, 1) \), \( C \) be a positively definite matrix, and

\[
\epsilon_t \sim N(0, C), \quad t = 1, 2, \ldots
\]

\[
\xi_t = \epsilon_t + \alpha \xi_{t-1} + \sqrt{1 - \alpha^2} \epsilon_t, \quad t = 2, 3, \ldots
\]

Fig. 1 demonstrates a realization of both the white noise \( \left( \epsilon_t \right) \) and \( \left( \xi_t \right) \). Let us analyze if \( \left( \xi_t \right) \) has the required properties. Their derivations can be found in Appendix A.

Both \( \epsilon_t \) and \( \xi_t \) have the same marginal distribution \( N(0, C) \). Therefore, \( \left( \xi_t \right) \) is stationary and zero-mean. Applying induction to (4), one obtains

\[
E_{\xi_{t+k}} \xi_T = a^{|k|} C \quad \text{and} \quad E_{\xi_{t+k}} \xi_{t+k} = a^{|k|} \text{tr}(C)
\]

for any \( t, k \). Therefore, \( \left( \xi_t \right) \) is autocorrelated, and this autocorrelation decreases with growing lag. Consequently, the values of \( \xi_t \) are closer to one another for subsequent \( t \) than the values of \( \epsilon_t \), namely,

\[
E\|\epsilon_t - \epsilon_{t-1}\|^2 = E(\epsilon_t - \epsilon_{t-1})^T(\epsilon_t - \epsilon_{t-1}) = 2\text{tr}(C)
\]

\[
E\|\xi_t - \xi_{t-1}\|^2 = E(\alpha - 1)\xi_{t-1} + \sqrt{1 - \alpha^2} \epsilon_t)^T(\alpha - 1)\xi_{t-1} + \sqrt{1 - \alpha^2} \epsilon_t
\]

\[
\times (\alpha - 1)\xi_{t-1} + \sqrt{1 - \alpha^2} \epsilon_t
\]

\[
= (\alpha - 1)^2 \text{tr}(C) + (1 - \alpha^2) \text{tr}(C)
\]

\[
= (1 - a)2\text{tr}(C).
\]
The Markov property of $(\xi_t)$ directly results from how $\xi_t$ (4) is computed. In fact, marginal distributions of the process $(\xi_t)$, as well as its conditional distributions, are normal, and their parameters have compact forms. Let us denote
\[ \pi_n = [\xi_t^T, \ldots, \xi_{t+n-1}^T]^T. \] (5)
The distribution of $\tilde{\xi}_n^\pi$ is normal
\[ N(0, \Omega_n^\pi) \] (6)
where $\Omega_n^\pi$ (21) is a matrix dependent on $n$, $\alpha$, and $C$. The conditional distribution $(\tilde{\xi}_n^\pi | \xi_{t-1})$ is also normal
\[ N(B_n^\pi \tilde{\xi}_{t-1}, \Omega_n^\pi) \] (7)
where both $B_n^\pi$ (24) and $\Omega_n^\pi$ (25) are matrices dependent on $n$, $\alpha$, and $C$.

B. Noise-Value Function

In policy (1), there is a stochastic dependence between actions beyond the dependence resulting from the state transition. Therefore, the traditional understanding of policy as the distribution of actions conditioned on the state does not hold here. Each action depends on the current state but also on previous states and actions. The analytical usefulness of the traditional value function and action-value function is, thus, limited.

Our objective now is to define an analytical tool in the form of a function that satisfies the following.

1) R1 (Hard Requirement): The function designates an expected value of future discounted rewards based on entities that this expected value is conditioned on.

2) R2 (Efficiency Requirement): A small change of policy corresponds to a small change of this function. While this is not necessary, it facilitates concurrent learning of the policy and this function approximation.

In order to meet the above requirements, we introduce an adjusted noise, $(u_t, \xi_t)_{t=1}^\infty$, as follows. $u_t$ and $\xi_t$ belong to the same space $\mathbb{R}^d$. Let
\[ f(\cdot; \theta, s) \] (8)
be a bijective function in $\mathbb{R}^d$ parameterized by $\theta$ and state. We have
\[ \tilde{\xi}_{t-1} = f(u_t-1; \theta, s_t), \]
\[ u_{t-1} = f^{-1}(\tilde{\xi}_{t-1}; \theta, s_t). \] (9)

Formally, we can apply $f$ to convert $\tilde{\xi}_{t-1}$ to $u_{t-1}$ and back whenever necessary.

As an analytical tool satisfying the aforementioned hard requirement R1, we propose the noise-value function defined as
\[ W^\pi(u, s) = E_x \left( \sum_{i \geq 0} \gamma^i r_{t+i} | \tilde{\xi}_{t-1} = f(u; \theta, s), s_t = s \right). \] (10)
The course of events starting in time $t$ depends on the current state $s_t$ and the value $u_{t-1}$. Because of the Markov property of $(\tilde{\xi}_t)$ (3) and the direct equivalence between $\tilde{\xi}_{t-1}$ and $u_{t-1}$, the pair $(u_{t-1}, s_t)$ is a proper condition for the expected value of future rewards.

To satisfy the aforementioned efficiency requirement R2, we design the $f$ function (8) based on $\pi$. It should make the distribution of an initial part of the action trajectory $(a_t, \ldots)$ similar for given $(u_{t-1}, s_t)$, regardless of the policy parameter $\theta$. Therefore, when $\theta$ changes due to learning, the arguments of the $W^\pi$ function (10) still define similar circumstances in which the rewards start being collected. This prevents large changes in the shape of $W^\pi$. An example of an appropriate $f$ function is provided in (18).

We can consider the pair $(u_{t-1}, s_t)$ a state of an extended MDP. Therefore, the noise-value function has all the properties of the ordinary value function. In particular, we consider $n$-step look-ahead equation in the form
\[ W^\pi(u_{t-1}, s_t) = E_x \left( \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n W^\pi(f(\tilde{\xi}_{t+n-1}; \theta, s_{t+n}), s_{t+n}) \right) | u_{t-1}, s_t. \] (11)
It says that the noise-value function is the expected sum of several first rewards, and the rest of them are also designated by the noise-value function itself.

The algorithm introduced below manipulates the policy $\pi$ (1) to make $n$-step sequences of registered actions more or less likely in the future. Let us consider
\[ \bar{s}^n = [s_t^T, \ldots, s_{t+n-1}^T]^T \]
\[ \bar{a}^n = [a_t^T, \ldots, a_{t+n-1}^T]^T \]
and
\[ \bar{\pi}(\bar{a}^n | \bar{s}^n, \tilde{\xi}_{t-1}; \theta) \] (12)
being a probability density of the action sequence $\bar{a}^n$ conditioned on the sequence of visited states $\bar{s}^n$, the preceding noise value $\tilde{\xi}_{t-1}$, and the policy parameter $\theta$. This density is defined by $\pi$ and the conditional probability distribution $\tilde{\xi}_n^\pi | \tilde{\xi}_{t-1}$. The algorithm defined in Section V updates $\theta$ to manipulate the above distribution.

C. Neural-AR Policy

A simple and practical way to implement $\pi$ (1) is given as follows. A feedforward neural network
\[ A(s; \theta) \] (13)
has input $s$ and weights $\theta$. An action is designated as
\[ a_t = \pi(s_t, \tilde{\xi}_t; \theta) = A(s_t; \theta) + \tilde{\xi}_t \] (14)
for $\tilde{\xi}_t$ in the form (1). Let us analyze the distribution $\bar{\pi}$ (12). In this order, the density of the normal distribution with mean $\mu$ and covariance matrix $\Omega$ will be denoted by
\[ \varphi(\cdot; \mu, \Omega). \] (15)
Let us also denote
\[ \bar{A}(\bar{s}^n; \theta) = [A(s_t; \theta)^T, \ldots, A(s_{t+n-1}; \theta)^T]^T. \] (16)
It can be seen that the distribution \(\tilde{\alpha}_n^s | \tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s\) is normal, namely, \(N(\tilde{\alpha}_n^s | \tilde{\xi}_n^s; \hat{\theta}) + B^n_{\tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s, \Omega_n^s}\) [see (6) and (7)]. Therefore, 
\[
\tilde{\pi}(\tilde{a}_n^s | \tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s; \theta) = \phi(\tilde{\alpha}_n^s; \hat{\alpha}_n^s, \hat{\theta}) + B^n_{\tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s, \Omega_n^s}.
\]

What is of paramount importance is the log-density gradient \(\ln \tilde{V}_0\) in \(\tilde{\pi}\). For \(\tilde{\pi}\) defined as (14), it may be expressed as
\[
\nabla \ln \tilde{\pi}(\tilde{a}_n^s | \tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s; \theta) = \nabla \phi(\tilde{\alpha}_n^s; \hat{\alpha}_n^s, \hat{\theta}) - \nabla \tilde{\pi}(\tilde{a}_n^s; \hat{\theta}).
\]

The f function (8) may have the form
\[
u_{n-1} = f^{-1}(\tilde{\xi}_{n-1}; \theta, s_n) = A(s_n; \theta) + \alpha \tilde{\xi}_{n-1}
\]
\[
\tilde{\xi}_{n-1} = f(u_{n-1}; \theta, s_n) = \alpha^{-1}(u_{n-1} - A(s_n; \theta)) \quad (18).
\]

\[\left. \begin{array}{c}
\text{Algorithm 1 Calculating Weights Update From a Single} \\
\text{Trajectory in ACERAC}
\end{array} \right. \]

A constant parameter of the algorithm is natural \(n\). It describes the length of action sequences whose probabilities the algorithm adjusts. For each time instant of the agent-environment interaction, the policy (1) is applied. Also, data are registered, which enable recall of the tuple \((\tilde{\xi}_n^s, \tilde{\xi}_n^s, \tilde{\xi}_n^s, s_{n+1})\), where \(\tilde{\pi}_n^s = \tilde{\pi}(\tilde{\alpha}_n^s | \tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s; \theta)\).

The general goal of policy training in ACERAC is to maximize \(W^\pi(u_{j-1}, s_j)\) for each state \(s_j\) registered during the agent-environment interaction. In this order, previous time instants are sampled, and sequences of actions that followed these instances are made more or less probable depending on their return. More specifically, \(j\) is sampled from \(t - M, \ldots, t - n\), where \(M\) is a memory buffer length, and \(\theta\) is adjusted along with a policy gradient estimate, which is derived in Appendix B. In other words, the conditional density of the sequence of actions \(\tilde{a}_j^n\) is being increased/decreased depending on the return
\[
u_{j+1} + \cdots + \gamma^{n-1} r_{j+n-1} + \gamma^n W(u_{j+n-1}, s_{j+n}; v)
\]
this sequence of actions yields.

A. Actor and Critic Training

At each \(t\)th instant of agent–environment interaction, ER is repeated several times in the form presented in Algorithm 1 to calculate actor and critic weight updates.

The RL algorithm presented in this section has an actor–critic structure. It optimizes a policy of the form (1) and uses the critic
\[
W(u, s; v)
\]
which is an approximator of the noise-value function (10) parameterized by the vector \(v\). The critic is trained to approximately satisfy (11).

A constant parameter of the algorithm is natural \(n\). It describes the length of action sequences whose probabilities the algorithm adjusts. For each time instant of the agent-environment interaction, the policy (1) is applied. Also, data are registered, which enable recall of the tuple \((\tilde{\xi}_n^s, \tilde{\xi}_n^s, \tilde{\xi}_n^s, s_{n+1})\), where \(\tilde{\pi}_n^s = \tilde{\pi}(\tilde{\alpha}_n^s | \tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s; \theta)\).

The general goal of policy training in ACERAC is to maximize \(W^\pi(u_{j-1}, s_j)\) for each state \(s_j\) registered during the agent-environment interaction. In this order, previous time instants are sampled, and sequences of actions that followed these instances are made more or less probable depending on their return. More specifically, \(j\) is sampled from \(t - M, \ldots, t - n\), where \(M\) is a memory buffer length, and \(\theta\) is adjusted along with a policy gradient estimate, which is derived in Appendix B. In other words, the conditional density of the sequence of actions \(\tilde{a}_j^n\) is being increased/decreased depending on the return
\[
u_{j+1} + \cdots + \gamma^{n-1} r_{j+n-1} + \gamma^n W(u_{j+n-1}, s_{j+n}; v)
\]
this sequence of actions yields.

In Line 2, the algorithm selects an experienced event to replay with the starting time index \(j\). In the following lines, the vectors of states \(\tilde{\xi}_j^n = [s_j^T, \ldots, s_{j+n-1}^T]\) and actions \(\tilde{a}_j^n = [a_j^T, \ldots, a_{j+n-1}^T]\) are considered.

In Lines 3 and 4, \(X_{j-1}\) and \(X_{j+n-1}\) are appointed to be values of the noise with which the current policy would designate the past actions. Then, in Lines 5 and 6, the corresponding adjusted noise values \(u_{j-1}\) and \(u_{j+n-1}\) values are calculated.

In Line 7, a temporal difference is computed. It determines the relative quality of \(\tilde{a}_n^s\).

In Line 8, a softly truncated density ratio is computed. The density ratio implements two ideas. First, \(\theta\) is changing due to being optimized; thus, the conditional distribution \(\tilde{\pi}_n^s | \tilde{\xi}_n^s, \tilde{\xi}_{n-1}^s\) is now different than it was at the time when the actions \(\tilde{a}_n^s\) were executed. The density ratio \(\tilde{\pi}(\tilde{a}_n^s | \tilde{\xi}_n^s, X_{j+n-1}; \theta)/\tilde{\pi}^n(\theta)\) accounts for this discrepancy of distributions. Second, to limit the variance of the density ratio, the soft-truncating function \(\psi_b\) is applied. For example,
\[
\psi_b(x) = b \tanh(x/b) \quad (19)
\]
for a certain \(b > 1\). In the ACER algorithm [21], the hard truncation function, \(\min[1, b]\), is used for the same purpose, which is limiting density ratios necessary in designating updates due to action distribution discrepancies. However, soft-truncating distinguishes the magnitude of density ratio and works slightly better than hard truncation.

In Line 9, an improvement direction for actor is computed. The sum of \(\nabla \ln \tilde{\pi}(\tilde{a}_j^n | \tilde{\xi}_j^n, X_{j+n-1}; \theta) d\beta^n_j(\theta, v, \nu, \rho)\) and \(\gamma^n \nabla W(u_{j+n-1}, s_{j+n}; v)\nu_j\rho_j\) is an improvement direction estimate of \(W^\pi(u_{j-1}, s_j)\) derived in Appendix B. It is designed to increase/decrease the likelihood of occurrence of the sequence of actions \(\tilde{a}_n^s\) proportionally to \(d\beta^n_j(\theta, v)\). \(L(s, \theta)\) is a loss function that penalizes the actor for producing actions that do not satisfy constraints, e.g., they exceed their boundaries.
In Line 10, an improvement direction for critic, \( \Delta v \), is computed. It is designed to make \( W(\cdot, \cdot; v) \) approximate the noise-value function (10) better.

In Line 7, the improvement directions \( \Delta \theta \) and \( \Delta v \) are applied to update \( \theta \) and \( v \), respectively, with the use of either ADAM, SGD, or another method of stochastic optimization.

Implementation details of the algorithm using the neural-AR policy (14) are presented in (17) and Appendix A.

VI. EMPIRICAL STUDY

This section presents simulations whose purpose was to compare the algorithm introduced in Section V to state-of-the-art RL methods. We compared the new algorithm to ACER [21], PPO [25], SAC [26], and CDAU [13]. We selected these algorithms as different state-of-the-art approaches that also apply trajectory updates (PPO and CDAU) or control exploration (SAC and CDAU). We used the RLlib implementation [27] of SAC and PPO, and the implementation of CDAU published by its authors.\(^1\) Our experimental software is available online.\(^2\)

For the comparison of the RL algorithms to be the most informative, we chose four challenging tasks inspired by robotics. They were Ant, Hopper, HalfCheetah, and Walker2D (see Fig. 2) from the PyBullet physics simulator [28]. A simulator that is more popular in the RL community is MuJoCo [29].\(^3\) Hyperparameters that assure optimal performance of ACER, SAC, and PPO applied to the environments considered in MuJoCo are well known. However, PyBullet environments introduce several changes to MuJoCo tasks, which make them more realistic and, thus, more difficult. In addition, physics in MuJoCo and PyBullets differ slightly [30]; hence, we needed to tune the hyperparameters. We based the hyperparameters in our experiments on their values used for MuJoCo environments reported in the original papers. We also followed their authors’ guidelines when selecting hyperparameters for tuning.

We do not limit the experiments only to the original environments. We also use modified ones with three and ten times finer time discretization. This is to verify how the algorithms work in these circumstances.

We used actor and critic structures as described in [26] for each learning algorithm. That is, both structures had the form of neural networks with two hidden layers of 256 units each.

A. Experimental Setting

Each learning run with basic time discretization lasted for three million time steps. Every 30 000 time steps of training a simulation were made with frozen weights and without exploration for five test episodes. An average sum of rewards within a test episode was registered. Each run was repeated five times.

In experiments with, respectively, three and ten times finer time discretization, the number of time steps for a run and between tests was increased, respectively, three and ten times.

\(^1\)https://github.com/ctallec/continuous-rl
\(^2\)https://github.com/lychanl/acerac
\(^3\)We chose PyBullet because it is freeware, while MuJoCo is commercial software.

Also, to keep the scale of the sum of discounted rewards, the discount parameter was increased from 0.99 to, respectively, 0.99\(^{1/3}\) and 0.99\(^{1/10}\), and the rewards were decreased, respectively, three and ten times. The number of model updates was kept constant for a different discretization. The data buffer was increased three and ten times, respectively. In ACER, the \( \lambda \) parameter was increased to \( \lambda^{1/3} \) and \( \lambda^{1/10} \), respectively. Also, in ACERAC, the \( n \) coefficient was increased three and ten times, respectively, and \( a \) was increased to \( a^{1/3} \) and \( a^{1/10} \), respectively.

For each environment-algorithm discretization, triple hyperparameters, such as step sizes, were optimized to yield the highest ultimate average rewards. The values of these hyperparameters are reported in Appendix C.

B. Ablation Results

Figs. 3–5, respectively, present results for ACERAC, for the original, three times, and ten times finer time discretization, with different \( a \)'s and \( n \)'s. The primary goal of these experiments was to verify whether the concepts introduced with ACERAC really contribute to performance. For \( a = 0 \), autocorrelation of actions is switched OFF. It is seen in the graphs that \( a = 0 \) yields inferior performance. \( n \) defines the length of sequences of actions whose density is manipulated in the course of learning. It is seen that our proposed default values usually yield optimal or close to optimal performance, but, in some cases, smaller \( n \) proves better.

C. Results

Figs. 6–8, respectively, present learning curves for all four environments and all four compared algorithms. The figures are for, respectively, the original, three times, and ten times finer time discretization. Each graph shows how a sum of rewards in test episodes evolves in the course of learning. Solid lines represent the average sums of rewards, and shaded areas represent their standard deviations.
Fig. 3. ACERAC with different $\alpha$’s and $n$’s for the original time discretization: average sums of rewards in test trials. Base: $\alpha = 0.5$ and $n = 2$. Environments: Ant, HalfCheetah, Hopper, and Walker2D.

Fig. 4. ACERAC with different $\alpha$’s and $n$’s for time discretization three times finer than the original: average sums of rewards in test trials. Base: $\alpha = 0.5^{1/3}$ and $n = 2 \cdot 3$. Environments: Ant, HalfCheetah, Hopper, and Walker2D.

It is seen in Figs. 6–8 that, in 12 combinations of tasks and time discretizations, ACERAC was the algorithm to yield the best performance in six cases; in two cases, it yielded the best performance ex equo with ACER, and in the rest of the cases, it still yielded reasonable performance.

A curious result of our experiments was the extraordinarily high rewards obtained in some experiments with time...
Fig. 5. ACERAC with different $\alpha$’s and $n$’s for time discretization ten times finer than the original: average sums of rewards in test trials. Base: $\alpha = 0.5^{1/10}$ and $n = 2 \cdot 10$. Environments: Ant, HalfCheetah, Hopper, and Walker2D.

It can also be seen that, for most discretization and problems, ACERAC obtained relatively good results in the

discretization ten times finer than the original. Namely, ACER and ACERAC obtained such results for Ant, and ACERAC for Hopper and Walker2D. Apparently, these environments require

the fast intervention of control, and no algorithm is able to learn it at coarser time discretization.

It can also be seen that, for most discretization and problems, ACERAC obtained relatively good results in the
initial training steps, which is a desirable feature in robotic control [31].

**D. Discussion**

The performance of the algorithms in our experiments with fine time discretization can be attributed to two features.

The first one is autocorrelated actions. ACERAC and CDAU use them, but only ACERAC utilizes their properties. Other considered algorithms do not use them. It can be seen in...
Figs. 6–8 that ACERAC achieved the best performance for eight discretization-environment pairs out of 12. Switching off autocorrelation from actions worsens the efficiency. The autocorrelated actions seem to be an efficient way to organize exploration, better than actions without any stochastic dependence beyond state transition. However, a policy with autocorrelated actions requires specialized training, which is provided in ACERAC.

The second factor is whether the algorithms use one-step returns (SAC and CDAU) or n-step returns (PPO, ACER, and ACERAC). The impact of this parameter on performance is quite well. This is visible in the upper part of Fig. 5, where t: paper.

Even though CDAU was designed in [13] to assure efficient RL in fine time discretization, that algorithm yielded poor returns (SAC and CDAU) or provided in ACERAC.

The dependence beyond state transition. However, a policy with dependence is a tool that enables RL in physical systems and introduced. Its efficiency was verified by simulations of Hopper, and Walker2D, at diverse time discretization. The dependence between subsequent actions beyond state transition. This is more demanding than simulations.

In this section, the key properties of the process (ξt) (4) are derived.

1) Stationary Distribution of ξt: From (4), one can see that if, for a certain t, it is true that ξt−1 ∼ N(0, C), then also ξt ∼ N(0, C). By induction, this leads us to the conclusion that ξt ∼ N(0, C) for all t.

2) Stationary Distribution of ξt+k: Applying induction to (4) for k ≥ 0, one obtains that

\[ \xi_{t+k}^* = \alpha^k \xi_t^* + \sqrt{1 - \alpha^2} \sum_{i=0}^{k-1} \alpha^{i} \xi_{t+i-1}. \]  

Consequently,

\[ E_{\xi_t^* \xi_{t+k}^*} = \alpha^{|k|} C \quad \text{and} \quad E_{\xi_t^* \xi_{t+k}^*} = \alpha^{|k|} \text{tr}(C). \]

Therefore,

\[ \xi_{t+k}^* = [\xi_t^T, \ldots, \xi_{t+k-1}^T]^T \sim N(0, \Omega_0^n) \]

\[ \Omega_0^n = \Lambda_0^n \otimes C, \quad \Lambda_0^n = [\alpha^{l-k}]_{l,k}, \quad 0 \leq l, k < n. \]  

The symbol “⊗” denotes the Kronecker product of two matrices. We have

\[ (\Lambda_0^n \otimes C)^{-1} = (\Lambda_0^n)^{-1} \otimes C^{-1}. \]

3) Conditional Distribution \(\xi_{t+k}^* | \xi_{t-1}^*\): From (20), we have that

\[ E(\xi_{t+k}^* | \xi_{t-1}^*) = \alpha^{k+1} \xi_{t-1}^* \]

and for 0 ≤ k ≤ l, we have

\[ \text{cov}(\xi_{t+k}^*, \xi_{t+l}^* | \xi_{t-1}^*) = E\left( \sqrt{1 - \alpha^2} \sum_{i=0}^{k} \alpha^{k-i} \xi_{t+i}^* \right) \left( \sqrt{1 - \alpha^2} \sum_{j=0}^{l} \alpha^{l-j} \xi_{t+j}^* \right)^T \]

\[ = (1 - \alpha^2)a^{l-k} (1 + a^2 + \cdots + a^{2k}) C \]

\[ = a^{l-k} (1 - a^{2k+2}) C. \]

Therefore, the conditional distribution \(\xi_{t+k}^* | \xi_{t-1}^*\) takes the form

\[ \xi_{t+k}^* | \xi_{t-1}^* \sim N\left(B^n \xi_{t-1}^*, \Omega_1^n \right) \]

\[ B^n = [\alpha I, \ldots, \alpha^n I]^T \]

\[ \Omega_1^n = \Lambda_1^n \otimes C, \quad \Lambda_1^n = [\alpha^{l-k} - \alpha^{l+k+2}]_{l,k}, \quad 0 \leq l, k < n. \]

4) Distribution of Actions’ Trajectory: For \(a_t = A(s_t; \theta) + \xi_t\) and \(n > 0\), we have \(\bar{a}_t^n = A(\bar{s}_t^n; \theta) + \xi_t^n\). The distribution of the actions that initiate a trial, \(\bar{a}(\bar{s}_t^n; \theta, \Omega_t^n)\), is, thus, normal \(N(\bar{A}(\bar{s}_t^n; \theta), \Omega_t^n)\). The distribution of further actions \(\bar{a}(\bar{s}_t^n| s_t^n, \xi_{t-1}^*; \theta)\) is also normal, namely, \(N(\bar{A}(\bar{s}_t^n; \theta) + B^n \xi_{t-1}^*, \Omega_t^n)\).

5) Retrieving \(\xi_{t-1}^*\) and \(u_{t-1}\) From Past Actions: For the OU processes, the values of \(\xi_{t-1}^*\) and \(u_{t-1}\) may be calculated from actions and actor’s outputs as

\[ \xi_{t-1}^* = a_{t-1} - A(s_{t-1}; \theta) \]

\[ u_{t-1} = A(s_t; \theta) + a_{t-1} \xi_{t-1}^* \]

\[ = A(s_t; \theta) + a_{t-1} (A(s_{t-1}; \theta) - a_{t-1}) \]

\[ = A(s_t; \theta) + a_{t-1} (A(s_{t-1}; \theta) - a_{t-1}). \]

If \(t\) is an initial instance of a trial, the conditional expected values of \(\xi_{t-1}^*\) and \(u_{t-1}\) are calculated from \(A(s_t; \theta)\) and \(a_t\), namely.

\[ \xi_{t-1}^* = a_{t-1} (a_t - A(s_t; \theta)) \]

\[ u_{t-1} = A(s_t; \theta) + a_{t-1} (a_t - A(s_t; \theta)). \]
B. Policy Gradient Estimator Derivation

In this section, we derive a policy gradient estimator, which is an estimator of a gradient of

\[ E_\pi \left( \sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \mid \tilde{s}_{j-1}, s_j \right) \]

with respect to the current policy parameter \( \theta \) for constant \( \tilde{s}_{j-1} = \tilde{s}_{j-1}(\theta) \).

Let us denote by \( \mathcal{A} \) the action space, by \( \theta \) the current policy parameter, by \( \pi \) the current policy, by \( \rho_j \) the policy parameter used when \( a_j \) was selected, by \( \pi(\theta_j) \) the policy used then, and the density ratio by

\[ \rho_j(\theta) = \frac{\pi(a_j|\tilde{s}^*_j, \tilde{s}^*_j; \theta)}{\pi(\tilde{a}_j|\tilde{s}^*_j, \tilde{s}^*_j; \theta)} \]

We have

\[
\frac{d}{d\theta} E_\pi(\theta) \left( \sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \mid \tilde{s}_{j-1}, s_j \right) = \frac{d}{d\theta} \int_{\mathcal{A}} \left( \sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \right) \times \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n
\]

\[
= \int_{\mathcal{A}} \left( \sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \right) \times \nabla_{\theta} \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n + \gamma^n \int_{\mathcal{A}} \nabla_{\theta} W^\pi (u_{j+n-1}(\theta), s_{j+n}) \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n
\]

\[
= \int_{\mathcal{A}} \left( \sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \right) \times \nabla_{\theta} \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n + \gamma^n \int_{\mathcal{A}} \nabla_{\theta} W^\pi (u_{j+n-1}(\theta), s_{j+n}) \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n
\]

\[
= \sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \times \nabla_{\theta} \ln \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n + \gamma^n \nabla_{\theta} W^\pi (u_{j+n-1}(\theta), s_{j+n}) \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n
\]

The analytical property

\[ E_{\pi(\theta)} \left\{ \nabla_{\theta} \ln \pi(\theta) \right\} = 0 \]

allows us to subtract any constant baseline from the sum of rewards above. Consequently, an unbiased estimator of the policy gradient may take the form

\[
\sum_{i=0}^{n-1} \gamma^i r_{j+i} + \gamma^n W^\pi (u_{j+n-1}(\theta), s_{j+n}) \times \nabla_{\theta} \ln \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n + \gamma^n \nabla_{\theta} W^\pi (u_{j+n-1}(\theta), s_{j+n}) \pi(\theta) \tilde{a}_j \: d\tilde{a}_j^n
\]

\[ + \gamma^n \nabla_{\theta} W^\pi (u_{j+n-1}(\theta), s_{j+n}) \rho_j(\theta). \quad (31) \]
The above estimator is not feasible. First, it is based on the noise-value function, which is unknown. Also, it uses the density ratio, which could make its variance excessive. In the noise-value function, which is unknown. Also, it uses the density ratio, which could make its variance excessive. In the noise-value function, which is unknown. Also, it uses the density ratio, which could make its variance excessive.

C. Algorithms’ Hyperparameters

This section presents hyperparameters used in the simulations described in Section VI. For the original time discretization, all algorithms used a discount factor equal to 0.99. Common parameters for the off-line algorithms (i.e., ACERAC, ACER, SAC, and CDAU) are presented in Table I. Hyperparameters specific for different algorithms are depicted in Tables II–IX. The hyperparameters were tuned using grid search over values spread by a factor of 3: \( \ldots, 10^{-6}, 3 \cdot 10^{-5}, 10^{-5}, \ldots \), with the exception of the clip parameter for PPO, whose only considered values were 0.1, 0.2, and 0.3, as suggested by the authors of this algorithm [25].

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.
[2] G. Brockman et al., “OpenAI gym,” 2016, arXiv:1606.01540.
[3] P. Khosla, “Choosing sampling rates for robot control,” in *Proc. IEEE Int. Conf. Robot. Autom.*, Mar. 1987, pp. 169–174.
[4] J. Schrimpf, “Sensor-based real-time control of industrial robots,” Ph.D. dissertation, Norwegian Univ. Sci. Technol., Trondheim, Norway, 2013.
[5] V. François-Lavet, R. Fonteneau, and D. Ernst, “How to discount deep reinforcement learning: Towards new dynamic strategies,” 2015, arXiv:1512.02011.
[6] M. Szulc, J. Lyskawa, and P. Wawrzyński, “A framework for reinforcement learning with autocorrelated actions,” in *Proc. Int. Conf. Neural Inf. Process.*, 2020, pp. 90–101.
[7] R. Liu, F. Nageotte, P. Zanne, M. de Mathelin, and B. Dresp-Langley, “Deep reinforcement learning for the control of robotic manipulation: A focussed mini-review,” *Robotics*, vol. 10, no. 1, p. 22, Jan. 2021. [Online]. Available: https://www.mdpi.com/2218-6581/10/1/22
[8] G. E. Uhlenbeck and L. S. Ornstein, “On the theory of the Brownian motion,” *Phys. Rev.*, vol. 36, pp. 823–841, Sep. 1930. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.36.823
[9] J. L. Doob, “The Brownian movement and stochastic equations,” *Ann. Math.*, vol. 43, no. 2, pp. 351–369, 1942. [Online]. Available: http://www.jstor.org/stable/1968873
[10] P. Wawrzyński, “Control policy with autocorrelated noise in reinforcement learning for robotics,” *Int. J. Mach. Learn. Comput.*, vol. 5, no. 2, pp. 91–95, 2015.
[11] H. van Hoof, D. Tanneberg, and J. Peters, “Generalized exploration in policy search,” *Mach. Learn.*, vol. 106, nos. 9–10, pp. 1705–1724, Oct. 2017.
[12] D. Korenkevych, A. R. Mahmood, G. Vasann, and J. Bergstra, “Autoregressive policies for continuous control deep reinforcement learning,” in Proc. 28th Int. Joint Conf. Artif. Intell., Aug. 2019, pp. 2754–2762.
[13] C. Tallec, L. Blier, and Y. Ollivier, “Making deep q-learning methods robust to time discretization,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2019, pp. 6096–6104.
[14] V. Mnih et al., “Playing atari with deep reinforcement learning,” 2013, arXiv:1312.5002.
[15] T. P. Lillicrap et al., “Continuous control with deep reinforcement learning,” 2015, arXiv:1509.02971.
[16] D. Vrabie and F. Lewis, “Neural network approach to continuous-time direct adaptive optimal control for partially unknown nonlinear systems,” *Neural Netw.*, vol. 22, no. 3, pp. 237–246, 2009.
[17] X. Guo, W. Yan, and R. Cui, “Integral reinforcement learning-based adaptive NN control for continuous-time nonlinear MIMO systems with unknown control directions,” *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 50, no. 11, pp. 4068–4077, Nov. 2020.
[18] A. G. Barto, R. S. Sutton, and C. W. Anderson, “Neuronlike adaptive elements that can solve difficult learning control problems,” *IEEE Trans. Syst., Man, Cybern.*, vol. 23, no. 1, pp. 109–119, Mar. 1987, pp. 169–174.
[27] E. Liang et al., “RLlib: Abstractions for distributed reinforcement learning,” in Proc. 35th Int. Conf. Mach. Learn., J. Dy and A. Krause, Eds., Stockholm Sweden, vol. 80, Jul. 2018, pp. 3053–3062.

[28] E. Coumans and Y. Bai. (2016–2019). Pybullet, A Python Module for Physics Simulation for Games, Robotics and Machine Learning. [Online]. Available: http://pybullet.org

[29] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: A physics engine for model-based control.” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst., Oct. 2012, pp. 5026–5033.

[30] T. Erez, Y. Tassa, and E. Todorov, “Simulation tools for model-based robotics: Comparison of bullet, havok, MuJoCo, ODE and PhysX,” in Proc. IEEE Int. Conf. Robot. Autom. (ICRA), May 2015, pp. 4397–4404.

[31] K.-T. Song and W.-Y. Sun, “Robot control optimization using reinforcement learning.” J. Intell. Robot. Syst., vol. 21, no. 3, pp. 221–238, 1998, doi: 10.1023/A:1007904418265.

Jakub Łyskawa received the M.E. degree in computer science from the Warsaw University of Technology, Warsaw, Poland, in 2020, where he is currently pursuing the Ph.D. degree.

His research interest is reinforcement learning.

Paweł Wawrzyński (Member, IEEE) was the Deputy Director for research of the Institute of Computer Science, Warsaw University of Technology (WUT), Warsaw, Poland, from 2018 to 2020. He is currently an Assistant Professor with WUT, where he leads the Machine Learning Team. He has authored five books and over 40 scientific papers. He holds four patents. His research interests cover reinforcement learning, artificial neural networks, and robotics.