# Learning-Based Continuous Control with Deep-RL

Haoyu Zhou, SYDE 655, University of Waterloo

*Abstract*— **This project report implements and compares D4PG and A3C, two popular deep reinforcement learning (RL) algorithms, for continuous control tasks using the Unity ML-Agents framework. D4PG utilizes distributed training with replay buffer and distributional value estimation, while A3C employs asynchronous updates with parallel agents. Experiments on the Reacher environment show that A3C converges faster with higher sample efficiency due to parallel training, while both algorithms achieved convergence with high scalability. Strengths, weaknesses, and practical considerations of both algorithms are discussed for real-world applications of deep RL.**

## I. Background

REINFORCEMENT Learning (RL) has gained significant attention in recent years for its ability to enable agents to learn optimal policies through trial-and-error interactions with environments. One of the challenging problems in RL is continuous control, where the agent needs to select actions from a continuous action space. Traditional RL methods that rely on discrete action spaces are not directly applicable to continuous control tasks, as they require selecting actions from a finite set of options. However, recent advancements in deep RL, which combines deep neural networks with RL algorithms, have shown promise in addressing this challenge.

The purpose of this project report is to compare different deep RL models against each other using the Unity ML-Agent environment "Reacher", which involves controlling robotic arms to reach a target location. In particular, we will focus on two popular deep RL algorithms for continuous control: Deterministic Policy Gradients (DPG) and Actor-Critic methods.

DPG is a policy optimization algorithm that learns a deterministic policy, mapping states to continuous actions, and uses gradient-based optimization to update the policy parameters. On the other hand, Actor-Critic methods combine the advantages of both policy-based and value-based methods by using an actor to select actions and a critic to estimate the state-action value function. These algorithms have been widely used in continuous control tasks and have shown promising performance in terms of sample efficiency and learning stability.

Hao Yu Zhou is with the Mechatronics Engineering Department, University of Waterloo, Ontario, CANADA

In this report, we will provide an overview of the theoretical foundations of DPG and Actor-Critic methods, including their key concepts, algorithms, and techniques. We will discuss the strengths and weaknesses of each algorithm and analyze their performance in the context of the "Reacher" environment. We will also review the current state-of-the-art in these methods, including recent advancements and cutting-edge techniques. Furthermore, we will conduct empirical experiments using the Unity ML-Agent environment "Reacher" to compare the performance of DPG and Actor-Critic methods against each other, analyzing their learning curves, sample efficiency, and generalization capabilities. Finally, we will discuss the findings and insights obtained from our training experiments and provide recommendations for practical considerations and best practices when applying DPG and Actor-Critic methods in continuous control tasks.

## II. Literature Survey

The Unity ML-Agents environment is very popular among developers to simulate the physical world in a much more convenient way. Thus, there are many implementations of the solution to this environment available online.

This post, “Solving Continuous Control using Deep Reinforcement Learning (Policy-Based Methods)” in the Medium forum provides a standard implementation of the DPG algorithm on the Reacher environment [2]. The author includes the hyperparameters as well as the adjustments in the reward function as codes.

For this project, I decided to propose using the more state-of-the-art algorithms (D4PG and A3C) to formulate and train the Reacher environment. After training the simulation, I will analyze the difference between the two algorithms to decide which is most suitable for solving continuous control problems.

## III. Kinematics and Algorithms

The Unity ML-Agents “Reacher” environment contains a 2-DOF robot arm and a goal “ball” position that is continuously moving in the state-space [1]. The ball travels with varying speeds and directions around the robot arm. The goal of the RL control is to maintain the end effector of the agent (robot arm) in the target “ball” location. For each episode, the longer the end effector stays inside the target location, the higher the score it achieves.

Formulating the kinematics of the environment, the robot observation state has a 33-D vector, each corresponding to the position, rotation, velocity, and angular velocity of the robot arm. The robot takes actions using a 4-D vector corresponding to the torque applied at the two joints which allows the robot arm to travel in 3D space. The diagram in Figure 1 shows the formulation of the kinematics in detail. Essentially, the torque corresponding to the $\theta_w$ and $\theta_e$ control the movement of the robot arm towards to target location. The optimal policy will be determined by randomly sampling in the action space with various trajectory steps.

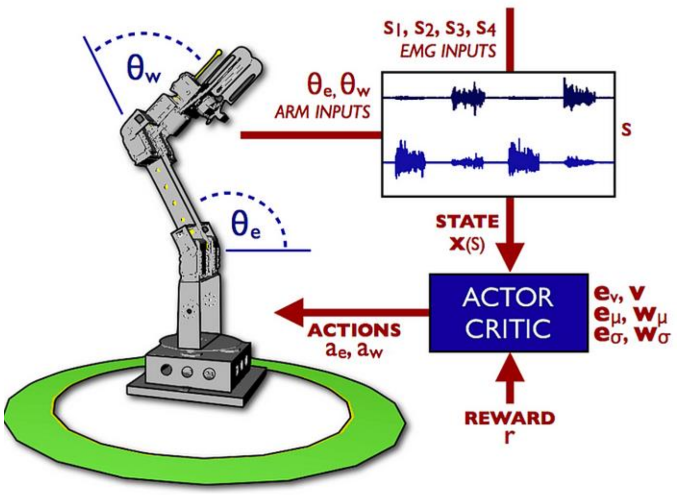


Figure 1. Kinetics formulation of the Robot Arm in Reacher Environment [2]

For this project, two algorithms will be used to analyze the RL approach in solve the environment.

## A. D4PG

D4PG is an extension of the original DDPG algorithm that incorporates a distributed architecture to improve learning efficiency in continuous control problems [3]. The agent interacts with an environment that has continuous state space denoted by $s_t$ and continuous action space denoted by $a_t$, and the objective is to learn an optimal policy that maximizes a task-specific cost function denoted by $J(s_t, a_t)$. The algorithm of DDPG from the original paper is shown in the appendices section.

D4PG uses an actor-critic architecture, where the actor selects actions based on the current state $s_t$, denoted by $\pi(a_t|s_t)$, and the critic estimates the value function or expected return from the current state-action pair $(s_t, a_t)$. The critic's loss function is typically the mean squared Bellman error, which is shown in equation (1) below.

$$L_{critic} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - Q\left(s_i, a_i \middle| \theta_Q\right)\right)^2 \tag{1}$$

From equation (1), $Q\left(s_i, a_i \middle| \theta_Q\right)$ is the estimated value function from the critic network, and the target value $y_i$ is computed as:

$$y_i = r_i + \gamma \int Q'\left(s_{i+1}, \pi'(a_{i+1}|s_{i+1})\middle|\theta_{Q'}\right) da_{i+1} \tag{2}$$

where $r_i$ is the reward obtained after taking action in the current state, $\gamma$ is the discount factor, and $\theta_{Q'}$ is learned from the Neural Network for the critic function.

The original DDPG algorithm often adds the Ornstein-Uhlenbeck noise to the selected actions to encourage exploration [3]. On the other hand, the D4PG algorithm uses a simple random noise from normal distribution. D4PG also incorporates distributional value estimation to estimate a distribution over the possible returns for each state-action pair, denoted by $Z(s_t, a_t)$. The distributional Bellman operator is used to update the distributional value estimates, and the loss function for the critic is computed using some distance metrics $d$. In this project, I used the simple normal distance as $d$ in the critic update function for simplicity.

For the actor, D4PG uses the deterministic policy gradient (DPG) algorithm, which aims to maximize the estimated value function while taking into account the exploration strategy. The actor's loss function is typically computed as the negative expected value of the critic's estimated value function at the current state-action pair, which is given by:

$$L_{actor} = -\frac{1}{N}\sum_{i=1}^{N} Q\left(s_i, \pi(a_i|s_i)\middle|\theta_Q\right) \tag{3}$$

To optimize the parameters of the critic and actor networks, I used the Adam optimizer for the actor networks to update the network weights based on the gradients of the loss functions. Additionally, target networks are used to stabilize the learning process, where the target critic network and target actor network are updated periodically with a soft update or a hard update.

## B. A3C

The A3C (Asynchronous Advantage Actor-Critic) algorithm is a popular reinforcement learning control theory used in continuous control problems [4]. It is an asynchronous variant of the actor-critic architecture that allows multiple agents to learn in parallel, making it well-suited for distributed and parallel computing setups. The algorithm from the original A3C paper is attached to the appendices section of this report.

The cost function for the critic in A3C is typically computed as the mean squared error between the estimated value function and the target value function. The target value function is often computed using the Bellman equation, which expresses the value of a state-action pair as the sum of the immediate reward and the discounted value of the next state. The mean squared error is used as the loss function for training the critic network, as seen in equation (4). $V(s_i|\theta_V)$ is the estimated value function with the learned NN parameter $\theta_V$. $R_i$ is the target value for the i-th state, and N is the batch number.

$$L_{critic} = \frac{1}{N}\sum_{i=1}^{N}\left(V(s_i|\theta_Q) - R_i\right)^2 \tag{4}$$

For the actor, A3C uses the advantage function, which measures the advantage of taking a certain action at a certain state compared to the estimated value function. The advantage function is given by equation (5). The advantage function is used to compute the policy gradient, which determines how the actor's policy should be updated.

$$A(s_i, a_i|\theta_A) = Q\left(s_i, a_i\middle|\theta_Q\right) - V(s_i|\theta_V) \tag{5}$$

The policy gradient is computed as the negative log probability of the selected action multiplied by the advantage function, and it is used as the loss function for training the actor network, given by:

$$L_{actor} = -\frac{1}{N}\sum_{i=1}^{N}\log\pi(a_i|s_i,\theta_\pi) * A(s_i, a_i|\theta_A) \tag{6}$$

In this project, the optimizer used by the NN critic-actor algorithm for A3C is Adam. Instead of using the single-agent environment, a multi-agent environment equivalent to the Reacher environment is used to experiment with parallel learning. It is expected to cost greater computational resources, but the training convergence should be much quicker.

## IV. Simulation training results

### A. *D4PG Training*

To start off, the D4PG algorithm was implemented with the state-action formulation for this project. Using the methodologies in the control theories section, the actor-critic agents are written with the help of instructional external resources [5]. The model is trained in the simulation environment Reacher with one robot agent with a maximum of 5000 episodes. The average training time per episode was around 8 seconds.

The first iteration of the training for D4PG did not converge in the end, with an ending score of 11.83 after 5000 episodes of training. I then tuned the hyperparameters and changed the number transitional trajectories from 1 to 10. Theoretically, this will grant the agent more possibilities of actions from the action space. After fine-tuning the hyperparameters, the training was restarted. After setting a desired score of 30, the training converged in 3868 episodes with a final score of 31.12. The training curve for the model after fine-tuning is shown in Figure 2 below.

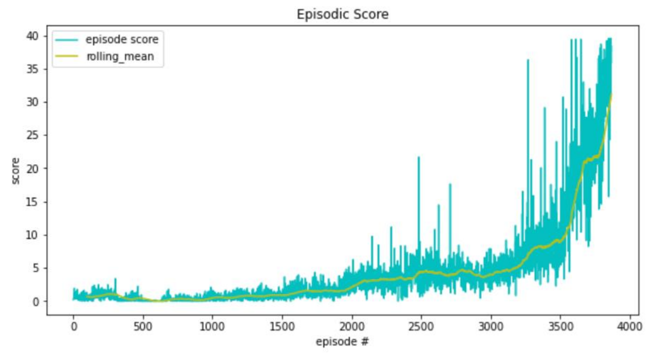


Figure 2. D4PG training curve over 3868 episodes.

After obtaining the desired training results, I tested the model over 10 more episodes. The model achieved an average score of 39.54, which validated the convergence. Overall, the whole training iteration was around 5 hours.

### B. *A3C Training*

After obtaining the results from the D4PG training, I reset the environment to start over the training with the A3C algorithm. One of the unique features of A3C is that it allows parallel training with multiple agents simultaneously. Based on the computational power of my computer, I decided to train 20 agents at the same time.

For the hyperparameters, I fine-tuned them with come validation episodes. Additionally, I used a 10-step trajectory as opposed to the 1-step used in D4PG training, with the hope of increasing prediction and sampling effectiveness. The critic and actor use the Adam optimizer within a 2-layer NN to learn the gradient parameters. The training converged to a score of 34.07 in 455 episodes. The average testing score of 10 episodes is 38.47. The training curve is shown in Figure 3 below.

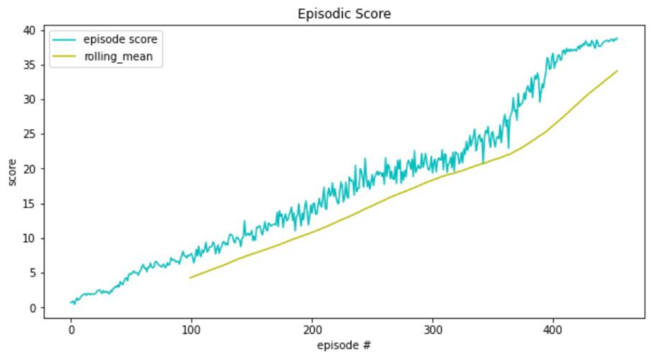


Figure 3. A3C training curve over 455 episodes.

## V. Analysis

Comparing the training results between the D4PG and the A3C algorithms, it can be seen that the training curve is much steeper for the A3C training. This was expected as A3C allows parallel training, which can significantly increase the training efficiency given enough computational resources. On the other hand, the D4PG algorithm was very effective in predicting the robot’s motion in 3D spaces. However, it required a much longer tuning process for its hyperparameters. This is also a result of the long training time it needs for the training curve to start an increasing trend.

Looking closely at the training curve of D4PG, it can be seen that the episode score first starts to increase after around 1500 episodes. After training for another 2000 episodes, the curve begins to increase sharply over the last 1000 episodes. This indicates that the hyperparameters are successfully tuned as the algorithm reached its peak performance in around 4000 episodes. For the A3C's training curve, the episode score shows a constant increment over the episodes. It is suspected that this phenomenon is caused by the concatenation of training parameters for all the 20 agents that are simultaneously trained. As the scores are concatenated together, the trained parameters become more robust, resulting in a smoother curve.

In terms of the hyperparameters, the main difference between the D4PG and the A3C training was that the D4PG used a 1-step trajectory action prediction, and the A3C used a 10-step trajectory. The higher number of step trajectory allows the model to predict more steps of the actions based on the observed states, which can increase the bias-variance tradeoff by decreasing the variance of the sampled distribution while increasing the bias. The 1-step trajectory is too small for this application which can in term result in suboptimal policies.

For the other hyperparameters, adjustment in a small range did not greatly impact the training results. The range was determined based on other implementations of the algorithms in other simulation environments, and these values are widely used for RL developers. The learning rate of the D4PG algorithm was slight adjusted after the first failed training session where it did not converge after 5000 episodes. I speculated that this was due to the learning rate being too large that the optimum of the loss function was skipped.

## VI. CONCLUSION AND FUTURE WORK

Based on the observations, it can be seen that both algorithms reached convergence, with the A3C algorithms having a much faster convergence due to parallel training. These two algorithms are both very effective in controlling the robot arm to follow the observed target location.

In addition, the tuning of the hyperparameters was essential for the success of the algorithms. For the two algorithms, the A3C was more efficient in tuning the parameters due to its faster training time. From this project, I learned two hyperparameters that had the most influence on the training results: the N-step trajectory and the learning rate. Without carefully tuning the parameters, the training could result in divergence.

For future work, more models and algorithms will be implemented and trained in the Reacher environment to analyze their corresponding performance. These algorithms could be the DDPG, PPO, etc. Another future work is to deploy this algorithm on my Capstone project, which is a 7-DOF robot arm targeting the continuous moving object of the user's mouth for the application of feeding food. In order to safely implement the trained policies, some constraints will be needed on the system.

## REFERENCES

[1] U. Technologies, "Machine learning agents," Unity. [Online]. Available: https://unity.com/products/machine-learning-agents. [Accessed: 17-Apr-2023]

[2] A. Chow, "Solving continuous control using deep reinforcement learning (policy-based methods)," Medium, 09-Jan-2021. [Online]. Available: https://ahtchow.medium.com/solving-continuous-control-using-deep-reinforcement-learning-policy-based-methods-64a871832496. [Accessed: 17-Apr-2023]

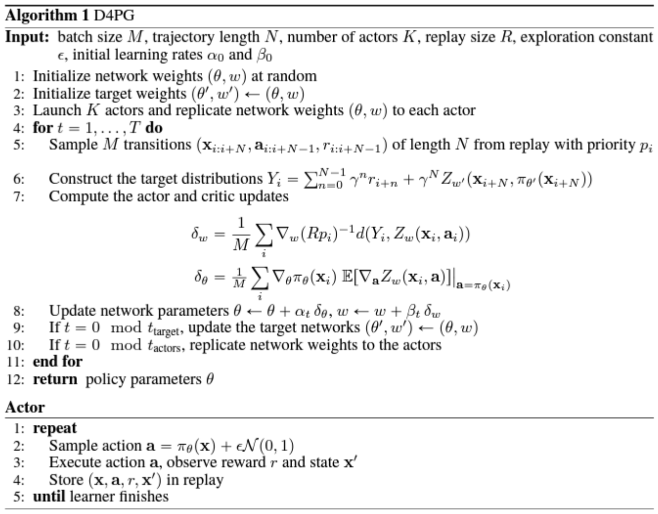
[3] Barth-Maron, Gabriel, et al. Distributed Distributional Deterministic Policy Gradients. arXiv, 23 Apr. 2018. arXiv.org, http://arxiv.org/abs/1804.08617. [Accessed: 17-Apr-2023]

[4] Mnih, Volodymyr, et al. Asynchronous Methods for Deep Reinforcement Learning. arXiv, 16 June 2016. arXiv.org, http://arxiv.org/abs/1602.01783. [Accessed: 17-Apr-2023]

[5] PacktPublishing, "Packtpublishing/deep-reinforcement-learning-hands-on: Hands-on Deep Reinforcement Learning, published by Packt," GitHub. [Online]. Available: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On. [Accessed: 17-Apr-2023]

## APPENDICES

### A. D4PG algorithm

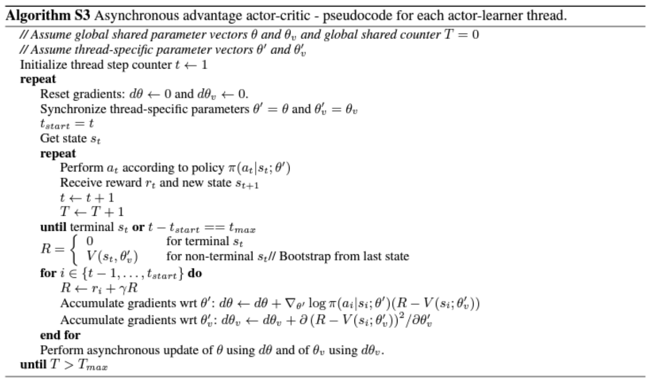


```
Algorithm 1 D4PG
Input: batch size $M$, trajectory length $N$, number of actors $K$, replay size $R$, exploration constant
       $\epsilon$, initial learning rates $\alpha_0$ and $\beta_0$
 1: Initialize network weights $(\theta, w)$ at random
 2: Initialize target weights $(\theta', w') \leftarrow (\theta, w)$
 3: Launch $K$ actors and replicate network weights $(\theta, w)$ to each actor
 4: for $t = 1, \ldots, T$ do
 5:   Sample $M$ transitions $(\mathbf{x}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ of length $N$ from replay with priority $p_i$
 6:   Construct the target distributions $Y_i = \sum_{n=0}^{N-1} \gamma^n r_{i+n} + \gamma^N Z_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$
 7:   Compute the actor and critic updates
          $\delta_w = \frac{1}{M} \sum_i \nabla_w (Rp_i)^{-1} d(Y_i, Z_w(\mathbf{x}_i, \mathbf{a}_i))$
          $\delta_\theta = \frac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i)\, \mathbb{E}[\nabla_{\mathbf{a}} Z_w(\mathbf{x}_i, \mathbf{a})]\big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i)}$
 8:   Update network parameters $\theta \leftarrow \theta + \alpha_t\, \delta_\theta$, $w \leftarrow w + \beta_t\, \delta_w$
 9:   If $t = 0 \mod t_{\text{target}}$, update the target networks $(\theta', w') \leftarrow (\theta, w)$
10:   If $t = 0 \mod t_{\text{actors}}$, replicate network weights to the actors
11: end for
12: return policy parameters $\theta$

Actor
1: repeat
2:   Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon\mathcal{N}(0, 1)$
3:   Execute action $\mathbf{a}$, observe reward $r$ and state $\mathbf{x}'$
4:   Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay
5: until learner finishes
```

Figure 4. D4PG original algorithm [3]

### B. A3C algorithm



```
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.
  // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
  // Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
  Initialize thread step counter $t \leftarrow 1$
  repeat
      Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
      Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
      $t_{start} = t$
      Get state $s_t$
      repeat
          Perform $a_t$ according to policy $\pi(a_t|s_t; \theta')$
          Receive reward $r_t$ and new state $s_{t+1}$
          $t \leftarrow t + 1$
          $T \leftarrow T + 1$
      until terminal $s_t$ or $t - t_{start} == t_{max}$
      $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
      for $i \in \{t-1, \ldots, t_{start}\}$ do
          $R \leftarrow r_i + \gamma R$
          Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
          Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$
      end for
      Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
  until $T > T_{max}$
```

Figure 5. A3C original algorithm [4]