Solving Unity Machine Learning Agents Reacher Environment Using Deep Deterministic Policy Gradient

Udacity - Project 2 Report: Continuous Control

Ryan Herchig

1 Introduction

The purpose of this project is to implement a reinforcement learning algorithm to solve the Unity Reacher environment [1]. The Reacher environment consists of a double-jointed arm can that can move to target different locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

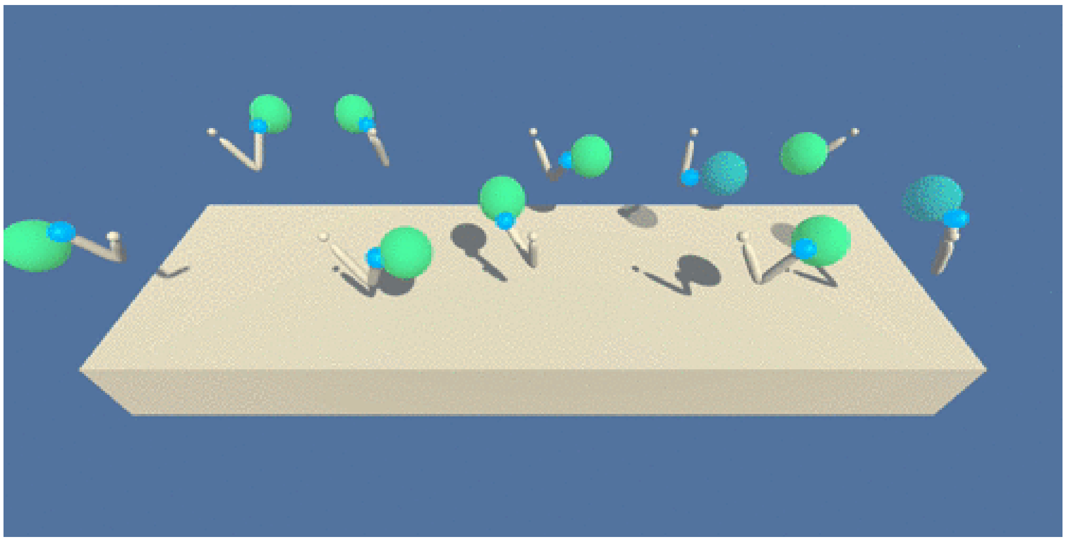

Figure 1: Image of the Unity Reacher environment for the multiple identical agents case [1].

For the Reacher environment, both the state (observation) and action space are continuous. The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1. There are two possible training configurations corresponding to single agent vs. distributed multi agent training. The multi agent version contains 20 identical agents, each with its own copy of the environment. These desperate agents are able to interact with their own individual environment, allowing the sharing of information between agents to accelerate the learning.

The barrier for solving the second version of the environment is slightly different, to take into account the presence of many agents. In particular, your agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically, After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of these 20 scores. This yields an average score for each episode (where the average is over all 20 agents). As an example, consider the plot below, where we have plotted the average score (over all 20 agents) obtained with each episode. The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30 [2].

2 Methodology

Simple approaches like deep Q-learning do not extend well to environments with continuous action spaces. This is because finding the greedy policy, finding the action with the maximum probability, requires a full optimization of a_t at each time step. Given the number of time steps required to train the typical agent in continuous state and action spaces, this is too computationally expensive to be feasible. The deep deterministic policy gradient (DDPG) algorithm was developed to address this issue [3]. It combines an actor-critic architecture, as is found in deterministic policy gradient [4], with some of the insights from the deep Q-learning model, such as a target network and an experience replay buffer. Having both a target network and a prediction network allows for stable target estimates to consistently train the critic network without divergence. The experience replay buffer solves the problem of decorrelating the samples, further stabilizing the training.

Given that both the state and action space are continuous in the Unity Reacher environment, a logical choice of reinforcement learning algorithm is deep deterministic policy gradient. This algorithm was created specifically for solving the problem of control in continuous environments, as is typically the case for robotics control problems. For continuous spaces, the number of possible values the variable can take is infinite (to floating point precision). This necessitates the use of a function approximator to allow the agent to generalize to cases that it hasn't seen exactly, but that its seen something similar to. The obvious choice for a function approximator, and one that allows the agent to learn non-linear policy functions and state-action value functions, is a neural network. With its actor-critic architecture, DDPG easily handles both continuous state and continuous action spaces.

The target network soft update equation is given by

$$\theta_{target} = \tau\theta_{local} + (1 - \tau)\,\theta_{target} \tag{1}$$

where τ was set to 0.001. Equation 1 shows that the parameters of the local network are slowly blended into the target network at each step. At each step,the new weights are composed of 99.9 % of the target network weights and 0.1 % of the local network weights. This helps ensure that at each learning step the TD target and the prediction are independent as they do not depend on the same neural network parameters.

The exploration in the model is accounted for by injecting Ornstein-Uhlenbeck (OU) noise into the action policy. Ornstein-Uhlenbeck process noise is used to generate temporally correlated exploration [3]. The exploration policy, which is the sum of the actor policy and the OU noise, is given by

$$\mu\,(s_t) = \mu'\left(s_t|\theta_t^{\mu}\right) + \theta_{OU}\,(\mu_{OU} - s_t) + \sigma_{OU}\mathcal{N}\,(0, 1) \tag{2}$$

where $\mathcal{N}\,(0, 1)$ is the standard normal distribution, θ_{OU} (0.15), μ_{OU} (0.0), and σ_{OU} (0.2) are the growth-rate or decay-rate, the diffusion coefficient, and the drift parameter respectively. Ornstein-Uhlenbeck is used to create

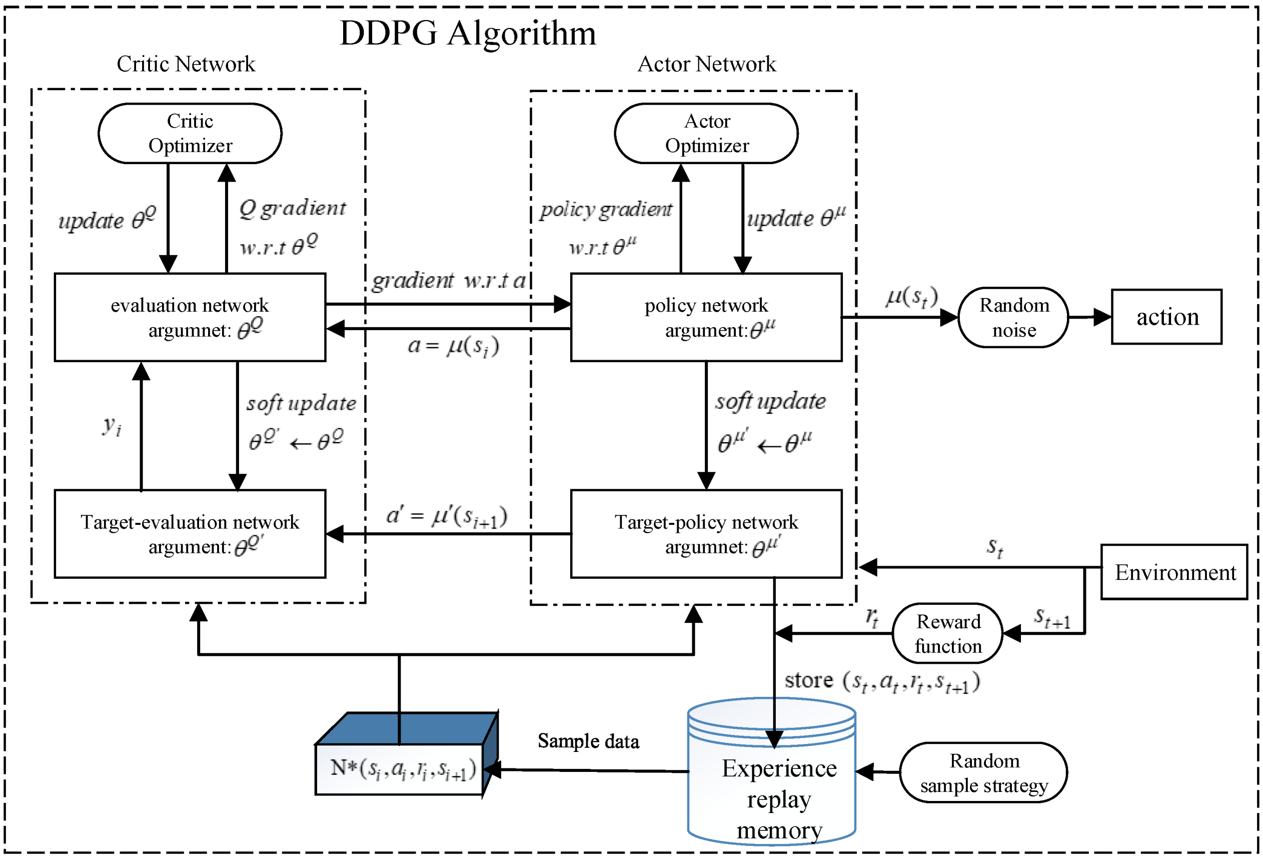
DDPG Algorithm
Critic Network
Actor Network
Critic Optimizer
Actor Optimizer
update θ^Q
Q gradient w.r.t θ^Q
policy gradient w.r.t θ^μ
update θ^μ
gradient w.r.t a
evaluation network argumnet: θ^Q
policy network argument: θ^μ
$a = \mu(s_i)$
$\mu(s_t)$
Random noise
action
y_i
soft update $\theta^{Q'} \leftarrow \theta^Q$
soft update $\theta^{\mu'} \leftarrow \theta^\mu$
Target-evaluation network argument: $\theta^{Q'}$
$a' = \mu'(s_{i+1})$
Target-policy network argumnet: $\theta^{\mu'}$
s_t
Environment
r_t
Reward function
s_{t+1}
store (s_t, a_t, r_t, s_{t+1})
$N*(s_i, a_i, r_i, s_{i+1})$
Sample data
Experience replay memory
Random sample strategy

Figure 2: Schematic diagram of the deep deterministic policy gradient architecture, adapted from [5].

temporally correlated noise that efficiently explores physical environments that have momentum. It is used to solve exploration-exploitation dilemma, given that the random noise distorts the action policy prediction produced by the actor network. Something similar to an ϵ-greedy policy was followed be defining a parameter ϵ as the probability of injecting noise into actor policy which decreased with episode count. An alternative method for reducing exploration could be to instead adjust the parameters of the OU noise, but add it to the actor policy with 100% probability. The parameters could be adjusted in such a way as to produce noise which is less random as the simulation progresses.

Figure 2 shows a schematic diagram of the DDPG algorithm. The experience replay buffer stores the memories (s_t, a_t, r_t, s_{t+1}) generated by the agent interacting with the environment. These are sampled randomly and used as input to the target actor, local actor (policy network), target critic, and local critic network (evaluation network), four distinct networks in total. Soft updates are used to slowly blend the parameters from the local networks to their respective target networks. The random noise which is added to $\mu(s_t)$ is Ornstein-Uhlenbeck noise; the equation for $\mu(s_t)$ given in 2. Gradients are taken with respect to θ^Q to train the local critic network (evaluation network), and with respect to θ^μ and a to train the local actor network (policy network).

The equation used to train the local critic network is

$$\mathcal{L}_{critic} = \frac{1}{N} \sum_i \left[y_i - Q\left(s_i, a_i | \theta^Q\right) \right]^2 \tag{3}$$

where θ^Q are the parameters of the evaluation network and N is the batch size. The value for y_i is given by

$$y_i = r_i + \gamma Q' \left[s_{i+1}, \mu' \left(s_{i+1} | \theta^{\mu'} \right) | \theta^{Q'} \right] \tag{4}$$

where γ is the discount factor, $\theta^{\mu'}$ and $\theta^{Q'}$ are the parameters of the target policy and target evaluation networks respectively, and r_i is the scalar reward. The policy update to the actor network is calculated using

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q\left(s, a|\theta^Q\right)\big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu\left(s|\theta^\mu\right)\big|_{s_i} \tag{5}$$

where θ^μ are the parameters of the local policy network, ∇_a and ∇_{θ^μ} are the gradients with respect to the action and the local policy network parameters.

The actor and critic networks were implemented using the PyTorch deep learning API [6]. For this project, the actor network consisted of three fully connected layers with rectified linear unit (ReLU) activations after the 1st and 2nd layers, and a hyperbolic tangent (tanh) activation in the final layer. The tanh activation layer was used to ensure that the actions mapped to values between -1 and 1 (corresponding to torque applicable to two joints). The input layer of the actor network had dimensions of the state size (33) by 512 neurons. The hidden layer had dimensions of 512x256, serving to reduce the dimensionality of the latent tensor during the forward propagation. The final layer had an input size of 256 and an output size equal to the number (4) of possible actions. The critic network had layers with the same input and output sizes, except that the final layer had a single output, corresponding to the state-action value estimate. The critic network also used ReLU activations in the hidden layers, but the output layer had no activation to produce a continuous value from $(\infty, -\infty)$. Additionally, while the states were fed into the critic network at the input (1st) layer, the actions were concatenated with the outputs of the 1st layer and the resulting tensor was subsequently fed into the 2nd layer. In this way, information from both the states and actions was used to estimate the state-action values.

The size of the replay buffer, the number of transition tuples it can store at a time, was 100000. The training was done using batch sizes of 128 randomly selected samples. The discount factor γ was set to 0.99 while the learning rates for both the actor and critic networks were set to 1x10$^-$4. The weights were initialized following the example from the original DDPG paper [3]. Specifically, the weights of the input and hidden layers were initialized as $\left[-\frac{1}{f}, \frac{1}{f}\right]$, where f is the fan-in, equal to the number of inputs to the layer. The weights of the output layers were initialized in the range $\left[-3\text{x}10^{-3}, 3\text{x}10^{-3}\right]$. The Adam optimizer [7] was used to train both the actor and critic networks. In each case, the default PyTorch Adam optimizer parameters were used aside from the learning rates.

3 Results

For the distributed multi agent training environment, agents must get an average score of +30 (over 100 consecutive episodes, and over all agents) for the environment to be considered solved. Specifically, after each episode, the rewards that each agent received (without discounting) are added up to get a score for each agent. This yields 20 (potentially different) scores. The mean average of these 20 scores yields an average score for each episode (where the average is over all 20 agents). The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

The results from training the DDPG algorithm for the 20 agent case are shown in fig. 3. At episode 24 the average score was 32.88, greater than the required +30. Over the subsequent 100 episodes, the average score remained above +30. The average score from the final 40 episodes was calculated to be 37.48 with a variance of 0.991. The small variance is reflected in fig. 3 as relatively small fluctuations in the scores from 80 episodes onwards, indicating that the algorithm has learned a stable policy.

From episode 1 to episode 17, the agent's scores increase modestly from 0.16 to 6.40. However at episode 18,

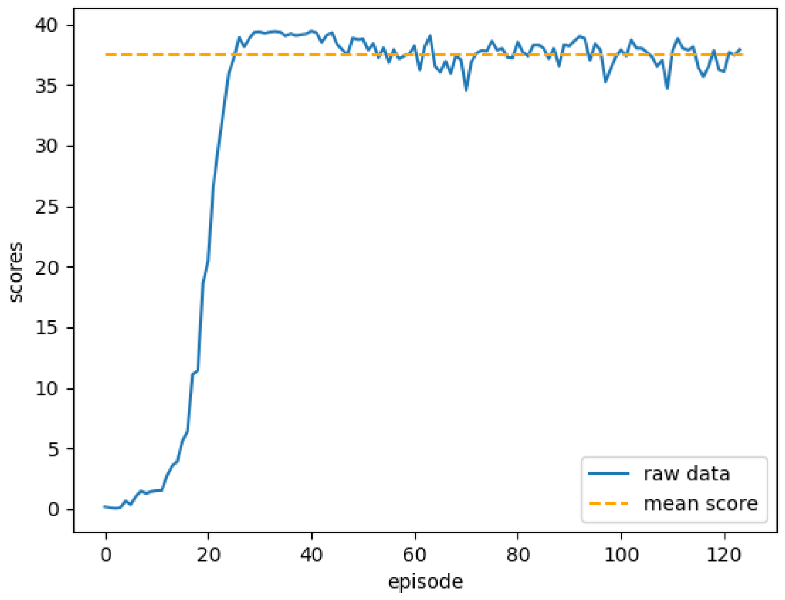

Figure 3: Scores for DDPG agent. The dashed orange line represents the mean score taken by averaging from 80 episodes until the end of the training.

the score increases to 11.10 and from episode 18 until episode 24, the agent's score increases much more rapidly. This suggests that during the initial stages of learning, the agent acts mostly using a random policy while the model keys in on the most fundamental characteristics of its task. After the model has learned the general goal of the environment, it quickly converges on a good behavioral policy. Additionally, since the actor network is being used to generate the samples used for learning, as the agent gets better at the task fewer of the samples in the experience replay buffer are from the mostly random policy and more are examples of when the agent is performing better at the task. This further serves to accelerate the learning.

4 Conclusion & Future Work

As was alluded to earlier in this report, an alternative to reducing the probability that OU noise with the same standard deviation will be injected into the action policy, decaying the standard deviation of the OU noise could provide an alternative way of decreasing the amount of exploration by the agent over the course of training. Essentially, the noise is always being injected into the action policy, except that the random fluctuations are decreased as the training progresses. In order to completely take advantage of the distributed multi agent training, implementing a different RL algorithm like asynchronous advantage actor-critic (A3C) [8] would be an interesting option. Asynchronous advantage actor-critic works for continuous action space and completely takes advantage of parallel agents training due to its asynchronous and distributed architecture. Each agent gets a copy of the network, explores the environment while learning better actor and critic policies, and periodically updates a central target network that represents the current best and most generalized overall policy.

Another possibility for future work would be to do a comprehensive hyperparameter tuning to optimize the performance of the algorithm with respect to maximum average score, stability, and training time. Several libraries exist which work seamlessly with the PyTorch library, namely the high-level PyTorch framework Catalyst [9] and the hyperparameter optimization framework Optuna [10]. Catalyst wraps standard PyTorch code, providing callbacks similar to those found in Tensorflow [11] and improving the reusability and generalizability of PyTorch code. The Optuna library is designed to automate hyperparameter searches by combining two main concepts, efficient search trees which utilize a history of trials and pruning algorithms which quickly abort runs using unpromising hyperparameter combinations. Using these tools would allow for optimal performance from the DDPG algorithm as currently implemented.

References

[1] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2020.

[2] alexisbcook, "Project 2: Continuous control." https://github.com/udacity/deep-reinforcement-learning/tree/master/p2_continuous-control, 2018.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.

[4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, p. I–387–I–395, JMLR.org, 2014.

[5] S. Guo, X. Zhang, Y. Zheng, and Y. Du, "An autonomous path planning model for unmanned ships based on deep reinforcement learning," *Sensors*, vol. 20, no. 2, 2020.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[7] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[9] S. Kolesnikov, "Catalyst - accelerated deep learning R&D." https://github.com/catalyst-team/catalyst, 2018.

[10] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," 2019.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.