Deep Reinforcement Learning Nanodegree Project 2 - Continuous Control

Elias Martins Guerra Prado

Abstract—This project uses deep reinforcement learning (DRL) Actor-Critic methods to train an agent to move a double-jointed arm to a target location. Each action the agent can take is a vector with four numbers that must be between -1 and 1, corresponding to torque applicable to two joints. The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the agent goal is to maintain its position at the target location for as many time steps as possible. The objective of the project is to train 20 identical agents at the same time, each with its own copy of the environment, to get an average score of +30 over 100 consecutive episodes, and over all the agents. To this end, in this work we implement two different agents based on actor-critic DRL methods: Deep Deterministic Policy Gradients (DDPG) and Proximal Policy Optimisation (PPO). After training, the performance of the agents was compared in order to evaluate which agent was able to achieve the objective in the least number of episodes, and which was able to achieve the objective in less time.

Index Terms—Deep Reinforcement Learning, Actor-Critic Methods, Deep Deterministic Policy Gradients, Proximal Policy Optimisation

1 Introduction

Despite the recent substantial advances in reinforcement learning (RL) techniques promoted by the integration with deep learning and the development of the "Deep Q Network" (DQN) [1] algorithm, a major problem remained unsolved. The DQN algorithm can solve problems with high-dimensional observation spaces, but it can only handle discrete and low-dimensional action spaces. Therefore, this algorithm can not solve physical control tasks, that have continuous (real valued) and high dimensional action spaces. To solve this problem some authors proposed to combine the DQN algorithm with actor-critic methods, allowing it to deal with continuous action spaces. Lillicrap et. al (2016) [2] proposed Deep Deterministic Policy Gradients (DDPG) algorithm. The technique combines Deterministic Policy Gradients (DPG) [3] algorithm, with the DQN algorithm to create a model-free, off-policy algorithm. DDPG was able to solve more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. The performance of the policies learned by the DDPG algorithm are competitive with those found by a planning algorithm with full access to the dynamics of the domain and its derivatives. Another proposed approach to solve continuous control problems is use Proximal Policy Optimisation (PPO) algorithm [4]. As demonstrated by the authors, this algorithm have some of the benefits of trust region policy optimization (TRPO) [5], but they are much simpler to implement, more general, and have better sample complexity (empirically). The PPO algorithm outperforms other online policy gradient methods on benchmark tasks of robotic locomotion.

This work is part of Udacity's Deep Reinforcement Learning Nanodegree course, and presents the submission of the second project of the course. In this project 20 RL agents must be trained to move a double-jointed arm to a target location. Each action the agent can take is a vector with four numbers that must be between -1 and 1, corresponding to torque applicable to two joints. The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the agent goal is to maintain its position at the target location for as many time steps as possible. The objective of the project is to train 20 identical agents at the same time, each with its own copy of the environment, to get an average score of +30 over 100 consecutive episodes, and over all the agents. To this end, in this work we implement and compared the performance of DDPG and PPO algorithms.

2 Implementation

As mentioned, in this work two different agents were implemented in order to solve the project environment (get an average score of +30 in 100 consecutive episodes). The DDPG agent was implemented based on the original DDPG algorithm [2], with the same actor and critic network architectures (low-dimensional). The PPO agent was implemented based on the original PPO algorithm, and on the actor-critic PPO implementation of Zhang available at https://github.com/ShangtongZhang/DeepRL. The state of the environment provided in the project is represented by a one dimensional vector, therefore, the hidden layers of the actor and critic networks implemented in this work are fully-connected layers (Linear layers). The neural networks were implemented using the PyTorch Python library.

2.1 Actor Network

The actor network for the DDPG agent has two hidden layers with 400 and 300 units respectively. All the hidden layers has ReLU activation except the last one. The final output layer was a $tanh$ layer. The final layer weights and biases were initialized from a uniform distribution [3 × 103,3 × 103]. The other layers were initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer. The PPO agent uses the DDPG actor network architecture with the addition of one more fully connected layer with 64 units followed by a $tanh$ layer.

2.2 Critic Network

The critic network for the DDPG agent has the same architecture and number of hidden units as the actor network, the difference is that after the first fully connected layer the output is concatenated with the action vector for then proceed to the other layers. The final output layer was a fully connected layer with 1 unit, corresponding to the state value. The final layer weights and biases were initialized from a uniform distribution [3 × 103,3 × 103]. The other layers were initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer. The critic network used by the PPO agent is the DPPG actor network architecture with the addition of one more fully connected layer with 64 units followed by a fully connected layer with 1 unit.

2.3 DDPG Training

The algorithm for training the DDPG agent executes the following steps. The agent observes the environment state then executes actions according to the actor network predictions. As suggested in the original DDPG implementation, exploration noise was added to the predicted actions, using Ornstein-Uhlenbeck process with $\theta = 0.15$ and $\sigma = 0.2$. After executing the action, the agent experiences (current state, action, reward, next state) are stored in the replay memory at each time-step. When the number of experiences on the replay memory is equal or larger then the batch size, the agent starts updating the actor and critic network weights. For this, at every 10 time steps the agent uniformly sample from the replay memory N samples, where N is equal to the batch size. These samples are then used to compute the loss and train the actor and critic networks. To update the critic network, first the target actor is used to predict the action of the next state, then the target critic is used to predict the state value of the next state and the corresponding predicted actions. The discounted cumulative return is then computed using the predicted state value for the next state. The loss of the actor network is computed by the mean squared error (MSE) between the cumulative discounted reward (CDR) of the next sated and the state value predicted by the local critic for the current state and corresponding actions. The critic network loss is computed by using the local actor to predict the actions based on the current state, and then using the local critic to predicted state values for the current state and the predicted actions. The mean of the predicted state values is then multiplied by -1 and used as loss for training the critic network. The target actor and critic networks weights are updated using soft update at each time-step, as follows:

$$\theta_i^- = \tau * \theta_i + (1 - \tau) * \theta_i^-$$

where θ_i^- and θ_i are the target and local network weights at time step i, respectively. τ is a constant between $[0, 1]$ that determine the magnitude of the update, with $\tau = 1$ corresponding to the case where the weights of the local network are copied to the target network.

This procedure was performed for each one of the 20 agents at each time step, that is, at each time step the weights of the actor and critic networks are updated 20 times, in a synchronous way.

2.4 PPO Training

Different from the DDPG agent, the PPO agent do not has a replay memory. For training it collects several trajectories (ordered secession of states, actions and rewards) at each time step, and sample from these trajectories to compute the losses. The algorithm for training the PPO agent executes the following steps. At each time step, each agent collect a trajectory with a maximum of 1000 observations (trajectory steps). For each state observation of the trajectory the actor and critic networks are used to predict the actions and the state values. As in the DDPG implementation, exploration noise was added to the predicted actions, using Ornstein-Uhlenbeck process with $\theta = 0.15$ and $\sigma = 0.2$. After computing the 20 trajectories, the algorithm calculated the CDR and the generalizaed advantage function (GAE) for each trajectory. The GAE values are then normalized by subtracting the mean and dividing by the standard deviation. Then the algorithm uses the collected trajectories and their respective CDR and normalized GAE to train the actor and critic networks. For this the algorithm randomly sample from the trajectories N samples, where N is the batch size. This samples are then used to compute the log probability ratios and the losses for the actor and critic networks.

3 Results and Discussion

The hiperparameters used in all the experiments are showed in Table 1. The buffer size for the DDPG agent replay memory are set to 1,000,000. In the PER implementation, the hiperparameter α used to compute the prioritization probabilities was set to 0.6, and β, used to compute the loss correction factor, was set to 0.4 and increased until it reaches 1 at the end of learning. The batch size was set to 64. The reward discount rate γ, was set to 0.99. The learning rate for the Adam algorithm was set to 0.0005 for the DQN and DDQN agents, and 0.0001 for the PER DDQN and PER Dueling DDQN agents. The soft update constant τ, was set to 0.001. Finally, in order to increase the learning efficiency the agents were set to learn (update the Q-Network weights) every k time steps, where k was set to 4. In this way, the agent take less time to finish an episode, because running the algorithm without learning requires less computation, allowing the agent to finish more episodes without significantly increasing the run-time. During training the agents were limited to make a maximum of 1000 action at each episode, and the total number of episodes was set to 2000.

Figures 1 and 2 shows the cumulative average score for the implemented agents after each episode. The PPO agent (Figure 1) obtained the best performance, reaching the cumulative score of +30 after 342 episodes which take 2 hours and 4 minutes. The DDPG agent (Figure 2) obtained lower performance, reaching the environment goal after

TABLE 1: Hyperparameters used for training the implemented agents

Hyperparameter	DDPG	PPO
Batch Size	128	128
Replay Memory Buffer Size	100,000	-
ϵ (PPO log probability ratio clipping)	-	0.1
β (PPO policy loss entropy regularization coeficient)	-	0.01
ϵ decay (decay of PPO log probability ratio clipping)	-	0.999
β decay (decay of PPO policy loss entropy regularization coeficient)	-	0.995
γ (reward discount rate)	0.99	0.99
Gradient Clipping (Actor and Critic)	1	1
Optimizer	Adam	Adam
Optimizer: Learning Rate (Actor and Critic)	510^{-5}	510^{-5}
Optimizer: β_1	0.9	0.9
Optimizer: β_2	0.999	0.999
Optimizer: ϵ	10^{-8}	10^{-8}
τ (soft update)	0.001	-
GAE τ (PPO)	-	0.95
Update weights every	20	-
Update times	10	10
Max actions per episode/ Max trajectory steps	1000	1000

100 episodes which take approximately 5 hours. Therefore, despite the lower number of episodes needed to train the DDPG agent, it take 2.5x more time than the PPO agent to reach the project goal.

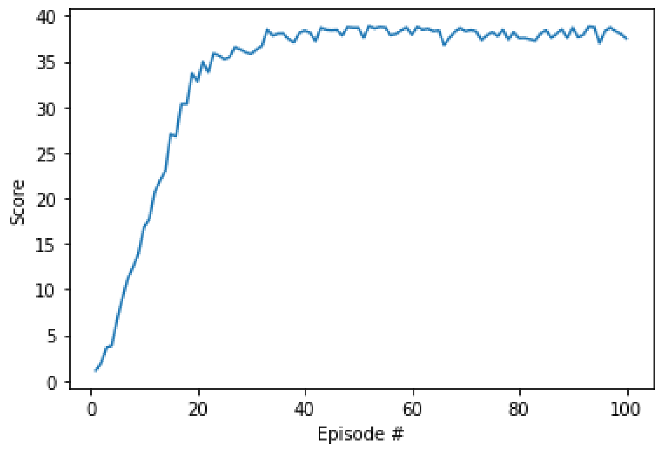

Fig. 1: DDPG learning performance on the Unity Continuous Control environment.

Analysing the learning curves (Figures 1 and 2), it can be observed that the agent trained with DDPG (Figure 1) maintains a steeper increase of reward over the first 20 episodes, archiving a maximum reward in less epísodes than the PPO agent. The curve for the PPO agent (Figure 2) is the least steep, but it continues to rise until the end of training. In addition, it can be seen that the DDQN agent obtained larger maximum scores during training.

4 Conclusion / Future work

The experiments developed in this works have shown that the implemented DDPG agent was able to successfully achieve the environment goal in less then 175 episodes (Udacity benchmark impementation). Beyond that, the PPO agent obtained the best performance, solving the environment 2.5x faster then the DDPG agent, and the DDPG

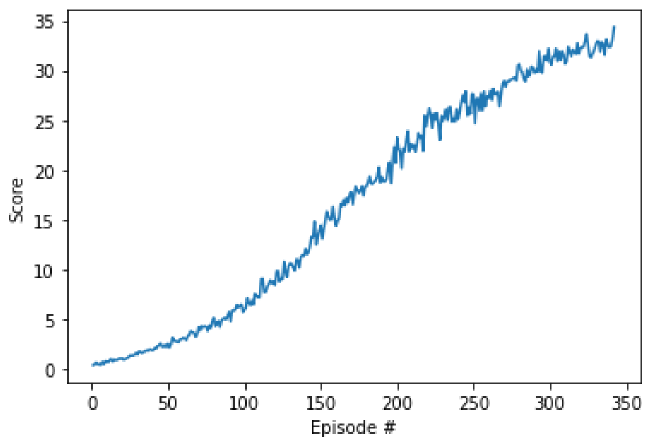

Fig. 2: PPO learning performance on the Unity Continuous Control environment.

agent obtained the maximum average reward of +33.67 over 100 consecutive episodes. Probably given the lack of time to tune the hiperparameters, the DDPG agent takes long time to solve the environment, however, when seen the overall performance, that can be measured by the maximum average scores, it can be seen that the DDPG agent implementation achieves a score higher then +30 at episode 20, after that the score reach a maximum of +38 at episode 40 and than stabilizes at this maximum. Therefore, the DDPG agent was able to solve the environment at episode 20, which takes 1/5 of the total training time, approximately 1h.

Despite the relatively good performance of the agents implemented in this work, there is still much room for improvement. As show in recent works [6], Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG) [7] achieve good performance on solving continuous action space environments. Also, recently another method for adapting DDPG for continuous control was proposed, called Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm [8]. In addition to these improvements many other papers have been published in the last 4 years showing new methods and techniques to increase the training performance of DRL agents in continuous control tasks. DRL is a newborn topic that is on the rise, and will probably remain hot for a long time due to its amazing ability and performance to solve RL problems.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] 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,” *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[3] 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 Machine Learning* (E. P. Xing and T. Jebara, eds.), vol. 32 of *Proceedings of Machine Learning Research*, (Bejing, China), pp. 387–395, PMLR, 2014.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.

[5] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization,” 2017.

[6] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, “Benchmarking deep reinforcement learning for continuous control,” *CoRR*, vol. abs/1604.06778, 2016.

[7] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization (trpo),” *CoRR abs/1502.05477*, 2015.

[8] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, “Distributional policy gradients,” in *International Conference on Learning Representations*, 2018.