Report on DDPG Algorithm

Zhang Long
Peking, CHN
zhanglong0922@gmail.com

Abstract

In reinforcement learning, DQN can efficiently predict discrete action spaces with continuous states as inputs. However, when dealing with continuous action spaces DQN is unable to converge to an ideal result. This article solves a continuous-action problem by employing a Deep Deterministic Policy Gradient (DDPG) method to train 20 agents (machine arms) with 4 continuous action spaces in 214 episodes within 40 minutes. The model can also scale to solve more complex problems as batch normalization is involved.

1 Introduction

It is known that DQN can solve problems with discrete action spaces (i.e., low dimensions), by combining a Q-Network and a Deep Neural Network, to approximate a continuous state observation and non-continuous actions; yet when faced with continuous action space, DQN fails to work well. If we come up with a solution to discretize continuous actions, there would be an exponential growth of dimensions. Take human arms (with 3 joints) as an example; we could use {-k, 0, k} as three individual sets to approximate human-arm actions with 7 degrees. That would lead to a distribution 3^7=2187 probabilities of action space, which not only slows the approximation of action by a great deal, but also takes a heavy toll on converging to the best training result, by yielding to unstable performance and divergence during training.

As a result, Deterministic Policy Gradient (DPG) algorithm (Silver et al., 2014) was introduced to solve the problem instead, with a simple actor-critic method to approximate parameters to find continuous optimized actions (see details in actor-critic algorithms). However, this method is proved to be unstable to scale to more complicated problems (Timothy P, 2016). Google DeepMind thus developed DDPG in 2016 by tapping the advantages of both DQN and DPG.

The DPG algorithm $\mu(s_t|\theta^\mu)$ specifies actions taken at the state t following a deterministic policy, given θ^μ, the weight, to be updated, thus by optimizing the gradient θ^μ the best policy is likely to be found as the agent collects more experiences.

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)}\nabla_{\theta_\mu}\mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned}$$

DQN collects uncorrelated transitions (states, actions, rewards, the next states) at each time-step as a replay buffer-zone for the agent to sample from, so that Q-values are able to be calculated.

Deep DPG (DDPG) is an off-policy algorithm that employs a large replay buffer (1e+5 experiences in our model) to update critic as well as the actor by sampling a minibatch uniformly from the replay buffer. However, if we use Q-Network $Q(s,a|\theta^Q)$ to update the target value straightforwardly the Q-update could diverge. An alternative method here is to copy the weights of $Q(s,a|\theta^Q)$ as well as the weights of $\mu(s|\theta^\mu)$ in critic and actor respectively. This method is called soft-updates ($Q(s,a|\theta^Q) \sim Q'(s,a|\theta^{Q'})$, $\mu(s|\theta^\mu) \sim \mu'(s|\theta^{\mu'})$). But that is not enough, the aim of soft-updates is to calculate the target Q-value. In other words, if the target stays unchanged it will be pointless to duplicate the weights as the Q-value is still prone to diverge. So an effective way is to **track the learned actor-critic networks** slowly by tuning the target slightly: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau<<1$. Note that **θ and θ' are not equivalent** in the implementation because both of the weights θ' (actor-critic) are initialized before the updates of θ, and τ here isn't set to 0 to keep track of the **updated** actor-critic network, which **greatly reduces instability during training because it will take a long time for the duplicates to get close to the learned weights**, let alone being affected a lot by them. It does cost a considerable time, but the training process can be much more stabilized without too much impact from the Q-Network. The learning process is solitude and undisturbed.

Note that the actions for now are selected purely based on the Network, optimizing policy gradient during entire training, but as we optimize our policies exploration is still needed otherwise the training network could easily get stuck in a local maximum. Thus, Ornstein-Uhlenbeck process (Uhlenbeck & Ornstein, 1930) was introduced to generate some random noise (N) to actions.

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

2 Model Briefing

When constructing the deep neural network, a batch normalization (BN) layer was added so that the model architecture could generalize well in other tasks:

```
def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)
```

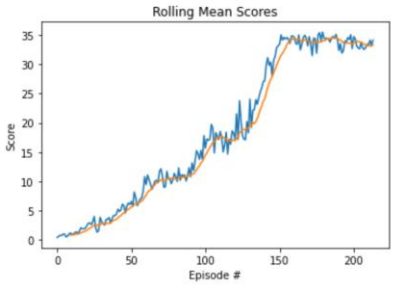
Besides, the model consists of an actor model (states → actions) and a critic model (state → action values) respectively and both are assigned with two hidden layers activated with Rectified Linear Units (ReLu) with 128 nodes for each. Note that at the top layer of the actor tanh is used as the activation to generate an action, while in the critic model in the middle the layer concatenates the previously trained state outputs and action inputs to generate an action value with a ReLu function instead.

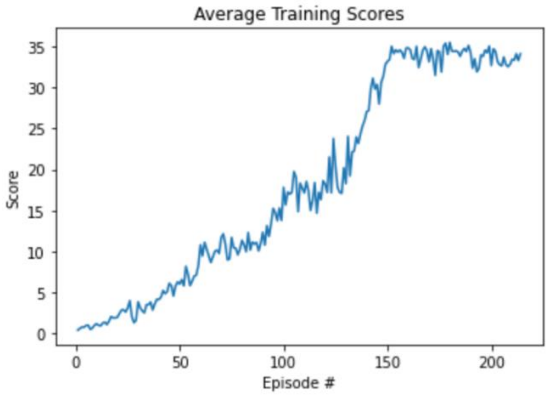
3 Training Plot

The training took merely 214 episodes to converge (+30.0) in 40 minutes locally (RTX 2060).

```
Episode 100     Average Score: 6.68
Episode 200     Average Score: 27.77
Episode 214     Average Score: 30.03
Envionment solved in 214 episodes!      Average Score: 30.03

Total training time = 40.8 min
```





4 Ideas for Future Work

1. We could try A3C methods to see if there are any better results.
2. Try n-step bootstrapping to see if the model could converge better at earlier steps.

Appendix

1 Algorithm:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

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

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

2 Further Thoughts

In DQN learning, the number of actions is discrete. The usual way to implement a DQN network is by mapping a state to multiple actions.

It is known that in reinforcement learning, we can either use on-policy (aka policy-based) learning, with actions taken on the policy selected either randomly or deterministically [1][i] when updating Q-values (aka action values or action-value estimates) at the same time or use off-policy (aka value-based) learning when actions are based on Q-values, which are updated based on rewards yielded by previously selected actions.

In reinforcement learning, both of on-policy learning and off-policy learning have shortcomings. In a policy-based method, the agent tends to end up with an unbiased optimized result but with high variance. In other words, the same architecture might fail to work at handling more difficult problems. For off-policy learnings, an agent cannot converge easily because action values are prone to deviate with more and more episodes completed, so that the model could become stabilized but more biased.

REFERENCES

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra *Continuous control with deep reinforcement learning*, 2019

---

[1] A deterministic policy is by mapping states to certain actions, while a stochastic policy maps states to uncertain actions, i.e., a distribution of action probabilities.