Source Seeking using Reinforcement Learning

Rahul Sajnani
International Institute of Information Technology, Hyderabad, India
rahul.sajnani@research.iiit.ac.in

Abstract

We[1] tackle the problem of *Source Seeking* in a forest setting. This report details the implementation of solving the source seeking problem using Deep Deterministic Policy Gradients.

1 Introduction

Source seeking refers to the problem of searching and traversing to the source of a particular signal. The signal can be a heat signature of a person lost in a forest or someone requiring medical assistance. The goal of this project is to reach the source of the signal from a drone using Reinforcement Learning. This problem is difficult as the environment space and action space are continuous.

Reinforcement learning (RL) seeks to maximize the *cumulative reward* of a bot (drone in our case) by exploring the environment. This exploration enables the bot to obtain the most appropriate action to be performed at a given state. For this, we assume that the states follow a *Markov Process (MP)*. The final outcome of this Reinforcement learning is to obtain a policy π^* that maps from a state to action.

Section 2.3 delves into continuous space Reinforcement learning by providing a recap of Deep Deterministic Policy Gradients. Here, we also recap concepts, such as fixed weight target network and experience replay. Section 3 covers the training details and design choices for DDPG implementation. Section 4 details the problems we faced while training, some solutions to solving them, and some experiments by varying reward functions.

2 Preliminary

2.1 Experience Replay

During the journey of the agent in the environment, the observed states by the agent are highly correlated. This high correlation does not allow the deep learning model to update its weights appropriately, biasing it towards a particular action. To avoid this biased update, (Mnih et al., 2013) use an experience replay buffer that stores the observations and the outputs of the Deep Q Network over multiple episodes. The authors store a tuple $e_t = (s_t, a_t, R_{t+1}, s_{t+1})$ into a finite data set $\mathcal{D}$, from which a batch of tuples are sampled randomly during training.

[1] I have referred to myself as we throughout this report.

2.2 Fixed target Deep Q network

Fixed target networks are used to supervise the actor and critic networks whose weights are fixed. We perform a soft update regularly every 30 training iterations. Without the fixed target network, calculating the Q-values for state s_t and s_{t+1} might update the network in such a way, that as $q(s_t, a_t)$ moves closer to $q^*(s_t, a_t)$, the target $q^*(s_t, a_t)$ itself moves away from $q(s_t, a_t)$ (Deeplizard, 2018). This causes the outputs to chase its own tail and to avoid this a fixed target Q-network is used.

2.3 Deep Deterministic Policy Gradients

Deep Q Networks have a finite discrete action space that may not scale up to solve all the problems. Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2019) allows Deep networks to regress to the most optimum action in a continuous action space. It uses an actor and a critic network. The actor network μ determines the optimum policy. Given the state s as input, the actor network obtains the optimal action a to be performed. The critic network takes in the action a and the state s as input and obtains the Q-value for performing action a in state s.

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t \qquad \textbf{(Actor)} \tag{1}$$

$$y_t = q_(s_t, a_t|\theta^q) \qquad \textbf{(Critic)} \tag{2}$$

Here, θ^μ and θ^q are the network weights for the actor and critic networks respectively. The predicted action and Q-value are then supervised by cloned actor-critic fixed networks, as follows:

$$\hat{y}_t = R_t + \gamma q'(s_t, \mu(s_t|\theta^{\mu'})) \tag{3}$$

$$\mathcal{L}(\theta^q) = \mathbb{E}[(y_t(\theta^\mu, \theta^q) - \hat{y}_t)^2] \tag{4}$$

$$\mathcal{L}(\theta^\mu) = -\mathbb{E}[(y_t(\theta^\mu, \theta^q)] \tag{5}$$

Here, $\theta^{\mu'}$ and $\theta^{q'}$ are the network weights for the **fixed** actor and **fixed** critic networks respectively. $\hat{y}_t$ is the supervisory signal and the gradients of the Q-value also flow through the critic network to update its weights as shown in Figure 1. We need to maximize the Q-value for the predicted action, to enforce this into the loss, equation 5 takes the negative mean Q-value as the error signal to update the actor network.

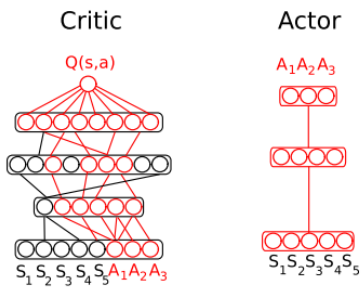

Figure 1: Deep Deterministic Policy Gradients: The actor network predicts the optimal action to take and the critic network determines the action value for taking action a from state s. Image courtesy (Olivier Sigaud, 2018)

3 Design choices and code

This section covers the implementation details and design choices for training Deep Deterministic Policy Gradients for *source seeking*. In this setting, the drone has a heat sensor that determines the signal strength of the source.

3.1 Reward functions

For the current setting, we use a continuous distance dependent reward function such that the agent is rewarded higher as the distance from the source keeps on reducing. This is different from binary rewards wherein a good step is assigned a fixed positive reward and a bad step is awarded a fixed negative reward.

$$R_t = -\left(\frac{d}{d_{max}}\right) \tag{6}$$

Note, the above reward R_t belongs to the range $[-1, 0]$. d and d_{max} represent the distance and max distance of the agent from the human source of the signal respectively. Here, we choose a negative reward so that the agent does not learn to take a detour and keep on accumulating positive reward to the human source. The signal strength of the sensor is inversely proportional to the distance from the source and is also given as an input to the network. Given is the equation for computing the signal strength s_t used while training.

$$s_t = \frac{1}{\alpha * d + 0.1} \tag{7}$$

$$\alpha = 0.01 \tag{8}$$

3.2 Model, State variables and action space

DDPG is majorly used to train RL agents over continuous action spaces. Unlike DQN, we do not select discrete actions, but instead regress to the optimum action. Since the action space is continuous, predicting a Q-value corresponding for every action is not possible. Therefore for training DDPG, we use actor and critic networks wherein the actor network determines the optimal action given a state and the critic network determines the Q value for that taking that action. The actor and critic models are defined below:

Listing 1: DDPG actor architecture

```
class Actor(nn.Module):
    def __init__(self, in_channels = 3, action_dim = 3, init_w=3e-3):
        super(Actor, self).__init__()

        self.conv1 = convrelu(in_channels, 64, 3, 2)
        self.conv2 = convrelu(64, 128, 3, 2)
        self.conv3 = convrelu(128, 64, 3, 1)

        self.pool = nn.MaxPool2d(3)

        self.fc1 = nn.Linear(2880 + in_channels // 3, 512)
        self.fc2 = nn.Linear(512, 64)
        self.fc3 = nn.Linear(64, action_dim)

        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
```

```
        self.init_weights(init_w)

    def init_weights(self, init_w):

        self.fc1.weight.data = fanin_init(self.fc1.weight.data.size())
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())
        self.fc3.weight.data.uniform_(-init_w, init_w)

    def forward(self, x, x_sensor):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(x.size(0), -1)
        x = torch.hstack((x, x_sensor))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        return self.tanh(self.fc3(x))
```

The actor network takes a batch of images x and sensor signal strength x_sensor as input and predicts an action. We use $Tanh$ activation to bound the output range between $[-1, 1]$.

Listing 2: DDPG critic architecture

```
class Critic(nn.Module):
    def __init__(self, in_channels = 3, action_dim = 3, init_w=3e-3):
        super(Critic, self).__init__()

        self.conv1 = convrelu(in_channels, 64, 3, 2)
        self.conv2 = convrelu(64, 128, 3, 2)
        self.conv3 = convrelu(128, 64, 3, 1)

        self.pool = nn.MaxPool2d(3)

        self.fc1 = nn.Linear(2880 + in_channels // 3 + action_dim, 512)
        self.fc2 = nn.Linear(512, 64)
        self.fc3 = nn.Linear(64, 1)

        self.relu = nn.LeakyReLU()
        self.tanh = nn.Tanh()
        self.init_weights(init_w)

    def init_weights(self, init_w):
        self.fc1.weight.data = fanin_init(self.fc1.weight.data.size())
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())
        self.fc3.weight.data.uniform_(-init_w, init_w)

    def forward(self, x, x_sensor, action):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(x.size(0), -1)
        out = torch.cat([x, x_sensor, action], 1)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out
```

Along with the batch of images x and sensor signal strength x_sensor, the critic network also takes the action as input and predicts the Q value. Note, the output for Q-value is not bounded and hence the range of the critic is not bounded.

For our experiments, each batch contains 4 images that provides motion information to the RL agent.

3.3 Environment

We use the $AirSim$ Neighborhood environment (Shah et al., 2017) for training the Multi-rotor drone.

3.4 Training details

The model is trained for 300 epochs with *SGD* optimizer having learning rate initialized to $2e-3$. We use learning rate decay factor of 0.1 every 100 epochs. The target model weights are updated in every step (soft update) with $\tau = 0.001$. The episode is reset every N steps. The N is determined by the distance between the goal and initial drone location.The training code is written using PyTorch (Paszke et al., 2019), Pytorch Lightning (Falcon and .al, 2019), and Hydra (Yadan, 2019).

3.5 Exploration

In the case of DDPG, we can not randomize between dicrete actions as the action space is continuous. In this scenario, we add noise generated from an $Ornstein-Uhlenbeck$ process. This process generates a mean-inverting process which essentially tries to drift towards the mean value at every time. We model the Ornstein Uhlenbeck noise as follows:

$$g \sim \mathcal{N}(0,1) \tag{9}$$
$$\Delta x = \theta(\mu_x - x)\Delta t + \sigma g \Delta t \tag{10}$$
$$x = x + \Delta x \tag{11}$$
$$\theta = 0.15, \Delta t = 0.01, \sigma = 0.2, \mu_x = 0 \tag{12}$$

As this process is mean-inverting, once the optimal policy is learnt, it drifts around the mean of the policy. But in the case when the agent has not learnt a policy, it provides a means of exploration. x is added to the predicted action of the actor network and clipped between $[-1, 1]$ to enable exploration.

3.6 Code

All the code for training the agents and reports are available here. $AirSim$ setup instructions for remote servers are here.

4 Difficulties of training Deep Deterministic Policy Gradients and plots for different reward functions

In the DDPG training procedure, we are trying to estimate both an optimal policy and an optimal critic. Achieving the optimal minima is difficult as both the critic and target networks are interdependent on one another. This optimization is highly non-linear and non-convex, making it highly dependent on the extent of exploration and initialization. The learning rates of actor and critic networks should complement each other. If this does not occur, the agent learns a local optima value near the start location. To solve this issue we add the exploration term as well as the vector directional term in the reward function. The exploration term rewards for moving

far away from the start location $R_t^{explore}$ and the vector directional term $R_t^{directional}$ rewards positively if the predicted action is in the direction of the goal. We also penalize the magnitude of the velocity vector if the agent is near the goal $R_t^{magnitude}$. This penalization ensures that if the agent is near the goal then its velocity must not be really high and must reach the goal safely (Bonsai, 2017)

$$R_t^{explore} = \left(\frac{d_{start}}{d_{max}}\right) \tag{13}$$

$$R_t^{directional} = CosineSimilarity(\mathbf{P}, \mathbf{v}) \tag{14}$$

$$R_t^{magnitude} = -L2norm(\mathbf{v}) * \left(\frac{d_{max}}{d + 100}\right) \tag{15}$$

$$R_t = -\left(\frac{d}{d_{max}}\right)^{0.4} + 0.07\left(\frac{d_{start}}{d_{max}}\right) + 0.1CosineSimilarity(\mathbf{P}, \mathbf{v})\left(\frac{d}{d_{max}}\right) \tag{16}$$

$$R_t = -\left(\frac{d}{d_{max}}\right) + 0.5CosineSimilarity(\mathbf{P}, \mathbf{v}) - 1e^{-4}L2norm(\mathbf{v})\left(\frac{d_{max}}{d + 100}\right) \tag{17}$$

Here, **P** and **v** is the directional vector towards the goal from the agent's current position and the predicted velocity vector. d_{start}, d, and d_{max} are the distance from the start location, distance from the goal and maximum distance respectively.

4.1 Plots for reward and losses

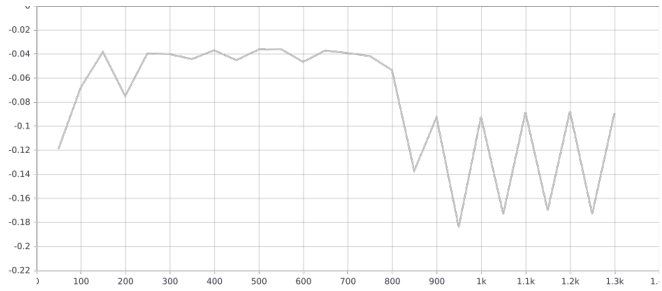

Figure 2: Oscillating reward vs. training iterations when using equation 6 as the only reward.

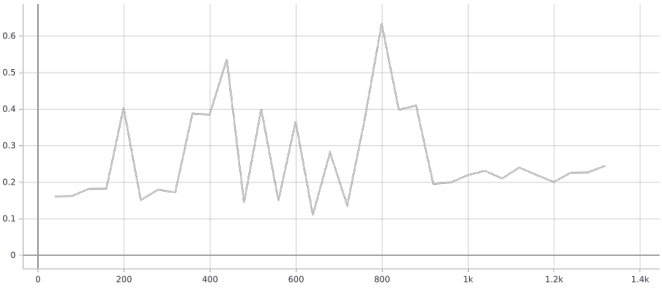

Figure 3: Policy loss vs. training iterations when using equation 6 as the only reward.

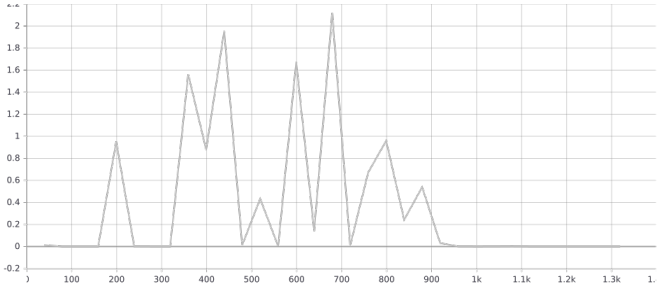

Figure 4: Critic loss vs. training iterations when using equation 6 as the only reward.

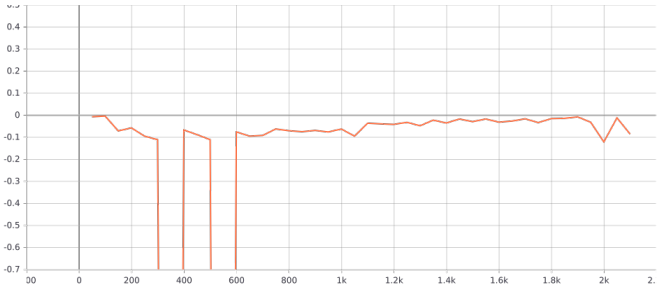

Figure 5: Reward vs. training iterations when using equation 16 as the reward.

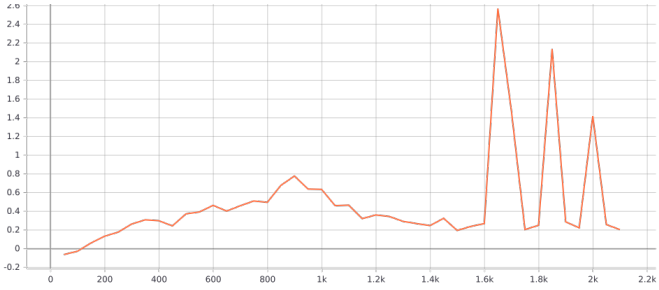

Figure 6: Policy loss vs. training iterations when using equation 16 as the reward.

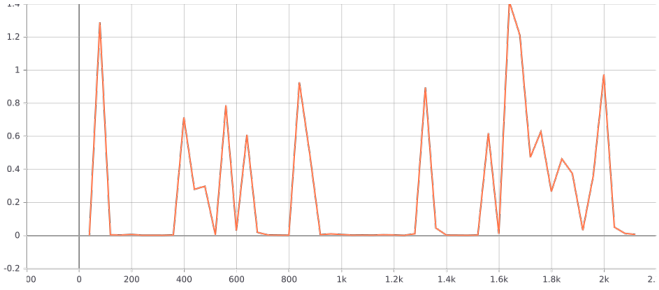

Figure 7: Critic loss vs. training iterations when using equation 16 as the reward.

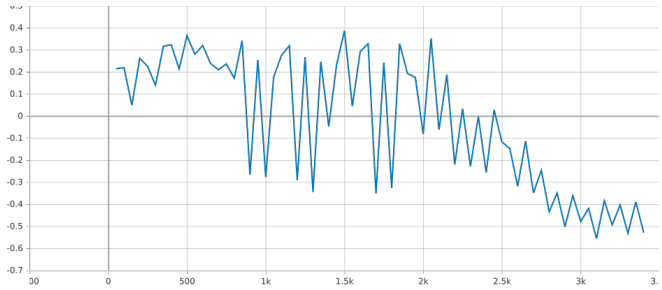

Figure 8: Reward vs. training iterations when using equation 17 as the reward.

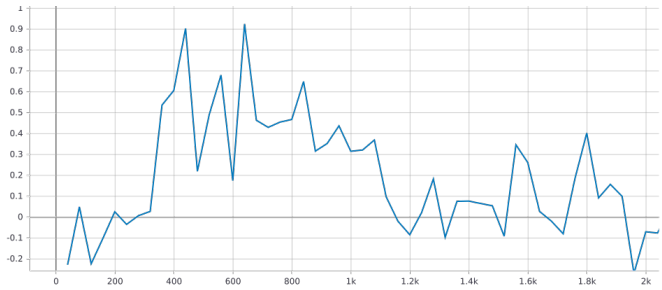

Figure 9: Policy loss vs. training iterations when using equation 17 as the reward.

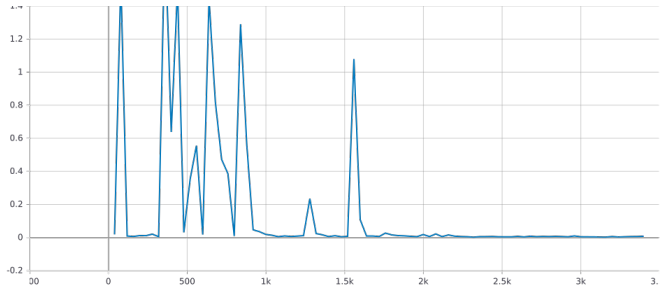

Figure 10: Critic loss vs. training iterations when using equation 17 as the reward.

4.2 Additional training difficulties

While training the DDPG algorithm we observed the following issues that can possibly reason out the reason for no convergence in our setting (Matheron et al., 2019):

1. **Oscillating around local minima**: Oscillating around local minima is a major issue as the agent keeps experiencing the same environment again and again. This fills up the replay buffer with the same states and trains the RL agent further to reach the local minima.

2. **Balancing learning between actor and critic**: in the actor-critic based off-policy learning, both the actor and the critic must learn at the same rate else we reach a local optima. This makes hyperparameter tuning very difficult.

3. **Momentum**: momentum based optimization do not do very well in the initial few stages of learning when the agent is exploring the environment. The high momentum during exploration affects training and increases the steps.

4. **Reward function**: choosing a good heuristic is one of the major deciding factors of how the agent will perform. In our case, reward equation 17 performs better than the remaining reward, but still does not converge. While designing the reward one must also take care of the range of the magnitude of the reward function as the critic network has to estimate the cumulative rewards for the same. Discrete rewards do not give a sense of appropriate direction for an agent and makes it harder for them to converge.

5 Acknowledgement

Thank you Dr. Harikumar Kandath sir for guiding me through the learning process and providing me an interesting Reinforcement Learning problem which is close to a real-world setting. Through this project I have learnt about the basics of reinforcement learning and how these can be extended to a deep learning setting in order to solve more complex problems. I also learnt how to use $AirSim$ and perform learning on a multirotor drone. I extended my understanding of Reinforcement Learning by trying to solve the source seeking task in a simulated environment. Lastly, thank you to all the online resources (blogs, lectures) for enabling better learning. In future I will focus on using other sophisticated methods such as TRPO or A2C to perform the source seeking task.

6 Some of my thoughts

Reinforcement learning depends on how an agent interacts with the environment to learn an appropriate policy/behavior. However, due to collision and other actions that can be disastrous in real-world scenarios, most of the reinforcement learning works focus on learning within a simulation. I (from the little that I know) see a need for safe reinforcement learning techniques that can train RL agents in real-world scenarios to solve more complex problems. I also see a need for better reward functions (maybe learnable reward functions).

References

https://www.youtube.com/watch?v=0R3PnJEisqk Bonsai. 2017. Writing great reward functions.

https://www.youtube.com/watch?v=xVkPh9E9GfE Deeplizard. 2018. Training a deep q-network with fixed q-targets - reinforcement learning.

WA Falcon and .al. 2019. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3.

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

Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud. 2019. The problem with ddpg: understanding failures in deterministic environments with sparse rewards.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning.

http://pages.isir.upmc.fr/~sigaud/teach/ddpg.pdf Olivier Sigaud. 2018. Deep deterministic policy gradient lecture resources.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703.

Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2017. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. *CoRR*, abs/1705.05065.

Omry Yadan. 2019. Hydra - a framework for elegantly configuring complex applications. Github.