Source Seeking using Deep Q Networks

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 covers the work done in training Deep Q Networks for the Source Seeking task.

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. The final outcome of this Reinforcement learning is to obtain a policy π^* that maps from a state to action. Section 2 gives a recap of Deep Q Networks. Section 3 discusses the training details for the Source Seeking task.

2 Deep Q Networks

To perform Reinforcement learning in a Deep learning setting, the first seminal paper developed the idea of Deep Q Networks (Mnih et al., 2013). Deep Q Networks estimate the action value function $q(s, a)$ given the state s as input. The state can be multiple images, direction vectors, etc. This is similar to Q learning, but here, the network replaces the Q-table to provide the Q-value for all actions. This network is applicable for a *finite set* of actions as the number of prediction heads are same as the number of actions (Figure 1).

Deep Q network minimizes over the same loss function as Q-learning, but here we use a network to do the same. The loss function is as given below:

$$\mathcal{L}_i(\theta_i) = \mathbb{E}[(R_{t+1} + \gamma \max_{a_{t+1}} q(s_{t+1}, a_{t+1}, \theta_{i-1}) - q(s_t, a_t, \theta_i))^2] \tag{1}$$

Here, θ_i represents the network weights at iteration i.

2.1 Fixed target Deep Q network

If we notice carefully, equation (1) uses the previous network weights θ_i for computing the next state's Q-values. This is the fixed target network whose weights are fixed, which is used as

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

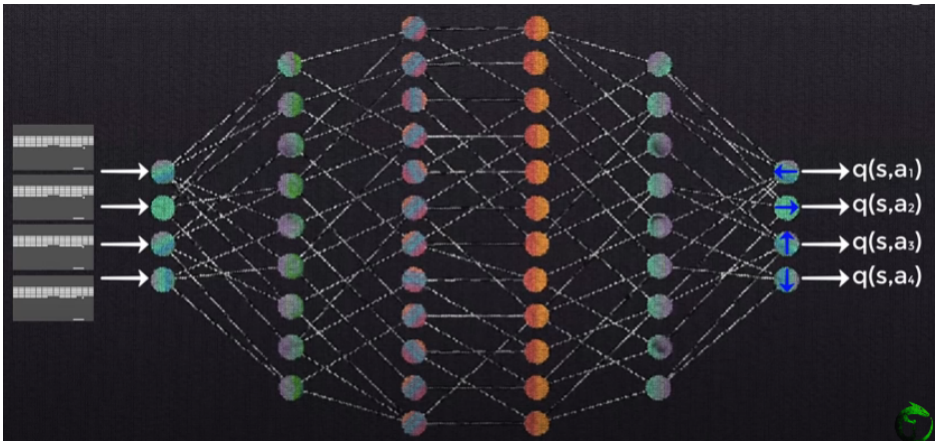

Figure 1: Deep Q Network: Given a state s as input, the network predicts the Q-values for all the actions taken at state s. Here, the Q-values are for the four actions up, down, left, and right. Image courtesy (Deeplizard, 2018)

a supervisory signal while performing the i^{th} weight update. This fixed network is regularly updated after finite set of 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.

2.2 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.

3 Design choices and code

This section dives into the design choices, code, and training details. The code can be found here: `https://github.com/RahulSajnani/Reinforcement-learning/tree/main/src/DeepQNetworks`

3.1 Rewards and heat sensor

I created a dummy heat signature sensor to capture the radiation of the human. The signal strength and the reward are directly proportional to the distance of the agent from the human. The rewards and signal strength code blocks are shown below:

Listing 1: Sensor signal strength code

```
def getSignalStrength(self, position):
    '''
    Function to obtain the signal strength of the sensor given the location of the RL agent

    Input:
        position - torch.Tensor 3 x 1 - x, y, z -
    '''

    distance = self.getDistanceFromDestination(position)
    signal = torch.tensor([[ 1 / (self.strength_factor * distance + 0.1)]])

    return signal
```

Listing 2: Reward function code

```
def getReward(self, position):
    '''
    Get the reward for reaching a position
    '''

    distance = self.getDistanceFromDestination(position)
    reward   = self.reward_factor * (- distance)
    reward   = torch.tensor([[reward]])

    return reward
```

Here, the *getReward* function uses a negative scale factor multiplied by distance as the reward so that the drone tries to minimize the distance. The reward scale factor and signal strength factor are set to 0.1 and 0.001 respectively. Collision with obstacles are given a reward of -100 and reaching the goal is rewarded with 1000.

$$d = distance(\mathbf{V}_{goal}, \mathbf{V}_{agent}) \tag{2}$$

$$R_t = -\alpha_1 * d \tag{3}$$

$$S_t = \frac{1}{\alpha_2 * d + 0.1} \tag{4}$$

$$\alpha_1 = 0.001; \alpha_2 = 0.1 \tag{5}$$

3.2 State variables and action space

In the case of source seeking, along with the camera images, I also consider the signal strength as an input. The intuition here is that if the source of the signal moves, the DQN in the inference stage should use the signal strength to reach the source. Following is the source code for the DQN architecture. **Note:** the input to the DQN model are images (variable x) and signal strength (variable x_sensor). The network takes the state as input and outputs a batch Q-values.

Listing 3: DQN architecture

```
class DQN(nn.Module):

    def __init__(self, in_channels=3, num_actions=6):
        super(DQN, 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.fc4 = nn.Linear(2881, 512)
```

```
        self.fc5 = nn.Linear(512, 64)
        self.fc6 = nn.Linear(64, num_actions)

    def forward(self, x, x_sensor):
        '''
        Inputs:
            x        - B, C, H, W   - Batch of images
            x_senor - B, 1          - Batch of signal strength
        '''

        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.fc4(x))
        x = F.relu(self.fc5(x))

        return self.fc6(x)
```

The action space contains 7 discrete actions. These are as shown in the code below:

Listing 4: Action space

```
def nextAction(self, action):
    '''
    Get change of position from action index
    '''

    scaling_factor = self.scaling_factor
    if action == 0:
        quad_offset = (scaling_factor, 0, 0)

    elif action == 1:
        quad_offset = (0, scaling_factor, 0)

    elif action == 2:
        quad_offset = (0, 0, scaling_factor)

    elif action == 3:
        quad_offset = (-scaling_factor, 0, 0)

    elif action == 4:
        quad_offset = (0, -scaling_factor, 0)

    elif action == 5:
        quad_offset = (0, 0, -scaling_factor)

    else:
        quad_offset = (0, 0, 0)

    return quad_offset
```

The scaling factor determines the offset (change of position) along each axis.

3.3 Environment

I 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 *Adam* optimizer having learning rate initialized to $1e-3$. I use learning rate decay factor of 0.1 every 70 epochs. The target model weights are

updated every 10 steps (soft update). 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 Constant epsilon factor vs. decaying epsilon factor

During training keeping a constant epsilon greedy factor does not work well and keeps oscillating in the same environment around the initial location. Hence, I decay the epsilon factor as the training iterations keep on increasing. The decay equation is as follows:

$$\epsilon = max(\epsilon_{min}, \epsilon_{max} - \frac{steps + 1}{stop\_decay}) \tag{6}$$

$$\epsilon_{max} = 0.9; \epsilon_{min} = 0.3; stop\_decay = 10000; \tag{7}$$

Here, $steps$ are the training steps. This decay allows the drone to venture more in the environment and learn.

3.6 Plots for loss and reward

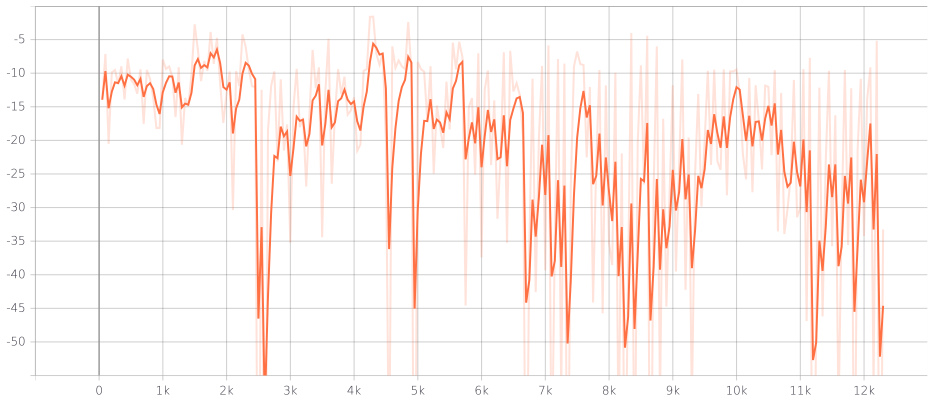

Figure 2: Reward vs. number of iterations

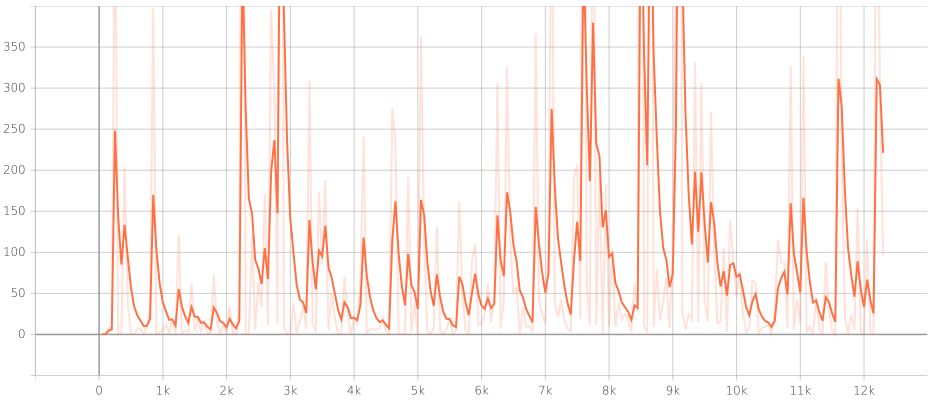

Figure 3: Training loss vs. number of iterations

4 Future work

The DQN does not perform well in this scenario. Even though the drone reaches near the goal (when reward is nearly 0 (Figure 2)) during training, it passes by it. To solve this problem, I plan to add an LSTM layer to the DQN network that keeps track of the previous states and the signal strength which can help determine differentiate if the next step is indeed a good one or not. In reinforcement learning tasks the state stores the rewards that we can obtain from from the current state onwards. By adding the LSTM block, we can have the features of the previous state which can give an idea about the change in the reward between my current and the previous state. This might help determine if the action taken to reach the current state is good or flawed.

References

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.

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

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.