Solving the Harvest CPR Appropriation Problem with Policy Gradient Techniques

Alessio Falai
M.Sc. in Artificial Intelligence
University of Bologna
alessio.falai@studio.unibo.it

Abstract

This report explains methodology and results for the final project in the Autonomous and Adaptive Systems course, held by professor Mirco Musolesi for the M.Sc. in Artificial Intelligence at the University of Bologna (academic year 2020 - 2021). In this work, we tackle a Common Pool Resource appropriation problem often referred to as Harvest or Gathering, in which independently learning agents aim to collect as many apples as possible, while also trying to keep the CPR flow healthy. In the original work, agents are trained with value-based RL methods, while in this work we rely on policy gradient techniques such as VPG, TRPO and PPO. Moreover, we implement a form of cooperation through gifting, a social learning scheme in which agents can deliberately be generous and gift each other with collected resources.

1 Common Pool Resources

The term Common Pool Resource (CPR) usually refers to renewable natural resources that are freely accessible by agents in an environment. There are two main components in every CPR domain: the stock, which represents the amount of core resources; and the flow, which represents the amount of fringe units. Intuitively, the resource itself is defined by the CPR stock and whenever the stock gets depleted the CPR flow at the next timestep will be null. So, the flow is a consequence of the stock and represents how many resources can re-grow based on how many resources we have on the stock. After such resources have re-grown, they "flow" into the stock and the natural cycle goes on until agents are able to maintain the correct equilibrium between stock and flow.

CPR problems are mainly categorized in two buckets: provision, which deals with stock supply; and appropriation, which deals with allocation of the flow. The core issue in CPR appropriation is about over-appropriation, i.e. when an agent takes too many resources from the stock. Over-appropriation has negative consequences both for the agent itself, as collecting resources now may impact resource collection in the future, and for all the other agents, as whenever someone takes one resource from the pool all other agents have one resource less to collect.

Modeling CPR appropriation problems with artificial agents poses great challenges as such problems tend to have what's called a socially-deficient Nash equilibrium, meaning that the choice to appropriate is tipically dominant w.r.t. leaving the resource where it is to allow for re-growth. This is because the choice to restrain leads to no individual benefit, but to the cost of CPR exploitation by others. Thus, having a socially deficient Nash equilibrium means that if agents were to collaborate, each one of them would probably obtain far more resources than if they were to act in a completely independent way.

2 The commons game

2.1 Social outcome metrics

2.2 Tagging and gifting

3 Reinforcement learning setting

3.1 Value-based methods

A DQN (Deep Q-Network) [1] leverages neural networks from the point of view of universal function approximators, to estimate the optimal Q^* state-action function. The main idea behind DQN is to estimate Q^* with Q_θ and use the Bellman equation to update the weights θ of the network so that $Q_\theta \to Q^*$. In particular, the loss to be used is a regression one (e.g. MSE or Huber) like the following to minimize the so-called TD (Temporal Difference) error:

$$L(\theta) = \mathbb{E}_{(s_t, a_t, r_{t+1}, s_{t+1})}\left[\left(\underbrace{r_{t+1} + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)}_{TD}\right)^2\right]$$

About the network architecture, the literature does not follow a clear definition, so that the term DQN is mostly linked to the whole training pipeline required to reach good estimates of the optimal Q function. For example, in the original paper [1] DeepMind researchers use an initial CNN (Convolutional Neural Network), since their input is represented by videogame frames, followed by an MLP (Multi-Layer Perceptron). Instead, the original Harvest implementation only relies on the final fully-connected layers, where the number of input neurons is given by the linearization of the selected observation, the number of hidden cells and the number of hidden layers are hyperparameters, while the number of outputs is equal to the size of the action space.

Another thing to notice is that the weights used for the target versus the current (presumably non-optimal) Q values are different. In practical terms, a separate network is used to estimate the TD target. This target network has the same architecture as the function approximator but with frozen parameters. Every T steps (a hyperparameter) the parameters from the Q-network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while). Because of this, assuming θ to represent the current network weights and θ^- the target network weights, the loss function is updated as follows:

$$L(\theta) = \mathbb{E}_{(s_t, a_t, r_{t+1}, s_{t+1})}\left[\left(r_{t+1} + \gamma \max_{a_{t+1}} Q_{\theta^-}(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)\right)^2\right]$$

One important issue of standard DQNs is that they tend to overestimate Q values (which is mostly due to the $\max$ operator in the Bellman optimality equation), leading to unstable training and low quality policies. To mitigate the issue, in [2] and [3] authors propose two popular DQN variants, the double and dueling ones, respectively.

3.2 Policy gradient techniques

$$\begin{aligned} L_{VPG} = CE(&\hat{P}_i^{(e)} \cdot \hat{A}_i^{(e)}, A) \\ &\alpha \cdot MSE(\hat{V}_i^{(e)}, \hat{R}_i^{(e)}) \end{aligned} \tag{1}$$

$$\begin{aligned} L_{TRPO} = CE(&\frac{\hat{P}_i^{(e)}}{\hat{P}_i^{(e-1)}} \cdot \hat{A}_i^{(e)}, A) \\ &+ \alpha \cdot MSE(\hat{V}_i^{(e)}, \hat{R}_i^{(e)}) \\ &- \beta \cdot KL(\pi(\theta_p^{(e-1)}), \pi(\theta_p^{(e)})) \end{aligned} \tag{2}$$

Algorithm 1 Generic policy gradient pipeline

```
Require: policy $\theta_p^{(0)}$, baseline $\theta_b^{(0)}$, epochs $E$, batch size $B$, max steps $M$
1: $e \leftarrow 0$
2: while $e < E$ do
3:     Build a pool of trajectories $T_e = \{t_1, \ldots, t_B\}$ by running policy $\pi(\theta_p^{(e)})$
4:     Compute rewards-to-go $\hat{R}_i^{(e)} = [\hat{R}_i^{(0)}, \ldots, \hat{R}_i^{(M)}]$ for each trajectory $i$
5:     Compute values as $\hat{V}_i^{(e)} = V(s_i; \theta_b^{(e)}), s_i \in t_i$ for each trajectory $i$
6:     Compute advantage estimates as $\hat{A}_i^{(e)} = (\hat{R}_i - \hat{V}_i)$
7:     Compute log-probabilities of actions as $\hat{P}_i^{(e)} = \pi(s_i; \theta_p^{(e)}), s_i \in t_i$ for each trajectory $i$
8:     Compute model loss and back-propagate
9: end while
```

$$\begin{aligned} L_{PPO} = CE(&\min(r^{(e)} \cdot \hat{A}_i^{(e)}, clip(r^{(e)}, 1-\epsilon, 1+\epsilon) \cdot \hat{A}_i^{(e)}), A) \\ &- \alpha \cdot MSE(\hat{V}_i^{(e)}, \hat{R}_i^{(e)}) \\ &+ \gamma \cdot S(\pi(\theta_p^{(e)})), r^{(e)} = \frac{\hat{P}_i^{(e)}}{\hat{P}_i^{(e-1)}} \end{aligned} \tag{3}$$

4 Results

5 Conclusions

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

[2] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`.

[3] Z. Wang, N. de Freitas, and M. Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL `http://arxiv.org/abs/1511.06581`.