Experiments on learning to communicate

Tom Bosc adresse@umontreal.ca
Hugo berard

April 22, 2017

Abstract

1 Introduction

The goal of this project is to study how policy gradients methods and Deep Q-Learning (DQN) performs in a partially observable environement (POMDP) with several agents. In particular we wanted to study how agents can learn to exchange message and communicate in order to solve a task. Our approcah was insbyred by [1] and [2]. We tried to evaluate the different approaches on a grid world task where the goal was to find some landmarks indicated by some other agents.

2 Models

2.1 Recurrent Policy Gradients

Indirect RL approaches to control require learning a value-state function $Q(s, a)$. Policy are derived from Q by maximizing over actions (*greedifying*). On the contrary, with Policy Gradient (PG) methods, the policy is directly parametrized. There are several motivations for doing that[3]. Firstly, in control problems, the Q function is not the quantity of interest. Furthermore, computing this quantity can be a harder problem. Finally, by comparison with ϵ-greedy policy where the entropy or randomness of the policy is fixed using ϵ, here it is flexible and adjusted during the learning process. Here we consider learning stochastic policies. The policy is denoted by π_θ and the objective is then to find the parameters θ that maximize the value function of the policy evaluated in the start state:

$$argmax_\theta v_{\pi_\theta}(s_0)$$

PG methods require computing the gradient, or at least estimating it when the policy is not differentiable. The policy gradient theorem states that:

$$\nabla_\theta v_{\pi_\theta} = \sum_s d_{\pi_\theta} \sum_a \nabla \pi_\theta(a|s) q_{\pi_\theta}(s, a)$$

This sum over states and actions is actually an expectation where the policy π_θ is followed and the environment gives states:

$$\nabla_\theta v_{\pi_\theta} = E_{\pi_\theta}[\gamma^t R_t \nabla log \pi_\theta(A_t|S_t)]$$

We can estimate it with Monte-Carlo by using the empirical return R_t.

In POMDPs, the agent has to infer the state its in. Indeed, it only has access to an observation which is a function of the state. This partial observability is generally dealt with by adding memory to the agent: at each timestep, the agent updates its memory which contains probabilities to be in the true states. Based on this probability that we call belief, the agent can decide on actions.

Recurrent Policy Gradients[4] (RPG) is a policy gradient algorithm that can solve POMDP. It combines REINFORCE with Recurrent Neural Networks (RNN). REINFORCE can be used jointly with backpropagation as it is only a gradient estimator[5]. In general, if the policy π is a probability mass or density function parametrized by parameters $y(\theta)$ where y is a deterministic function, we can decompose the gradient using the chain rule:

$$E[\frac{\partial \pi}{\partial \theta_i}] = E[\frac{\partial \pi}{\partial y}]\frac{\partial y}{\partial \theta_i}$$

In the case of RPG, REINFORCE is computed on the stochastic outputs units of the RNN whereas regular backpropagation is used on the deterministic part of the RNN. Here, $y_t = f(h_t, \theta)$ where h_t is the hidden state of the LSTM at timestep t. The hidden state can be seen as the belief and the updates are learned and represented as the transition matrix. The inputs of the LSTM is the observation of the agent, the last action that was performed by the agent (it matters because the policy is stochastic) and the reward. The outputs of the LSTM are the parameters of the distribution from which the actions will be sampled.

A Long Short-Term Memory[6] (LSTM) RNN is chosen for convenience and performance.

To reduce variance, a baseline is used. Usually, baselines are taken to be v_π so that the log gradient term is weighted by the advantage function $q_\pi(s, a) - v_\pi(s) = A_\pi(s, a)$. Since we are in POMDP, we can't estimate the state value function but instead learn an approximation based on the belief state. It is recommended to use another RNN as a baseline which takes the same inputs (except for the reward) and produce an estimate of the empirical returns. We can train this RNN supervisedly as we have access to the empirical returns.

The learning process is done offline and with minibatches as the variance of the estimator used can be high.

2.2 Recurrent Deep-Q-Learning

When dealing with a large state space tabular representation becomes impractical to use. To remedy to this problem we can use function approximation and parametrize the Q-value function as a function of the state. Recently it has been shown [7] empirically that despite not having any convergence guarantee, neural networks performs well. The parameters are updated following the usual subgradient of the MSE between the predicted Q-value and the target:

$$\theta_{t+1} = \theta_t + \alpha(r + \max_a Q_{\hat{\theta}}(s_{t+1}, a) - Q_{\theta_t}(s_t, a_t))\nabla_{\theta_t} Q_{\theta_t}(s_t, a_t)$$

In order to work well, we need to introce several tricks:

Replay Memory: Instead of updating online, the transitions are stored in a replay memory, and at each time step a batch of transitions is sampled from the memory to train the model.

Target Network: When computing the target, the Q-value for the next-step, we use a copy of the network but with a set of parameters $\hat{\theta}$ only updated every n iterations.

Recurrent Network: To deal with the fact that the environement is partially observable, we add a recurrent layer so that the agent is able to remember the previous observation. To train the model, following [8] we sample random episodes from the replay memory and unroll the episode for n steps.

3 Experiments

3.1 Gridworld

This experiment is a simplified variant of [1]. The main difference is that our environment has discrete action space and state space. This allows for a fairer comparison between RDQN and RPG. Indeed, Q-learning updates are computed by maximizing the action-state value function $Q(s, a)$ over actions which is in general costly in continuous action space.

The environment is a POMDP. There are N agents and M landmarks which are defined by an integer i and their respective coordinates x_i on a squared grid. At each timestep, agents choose an action from $A = left, right, up, down$ which makes them move[1]. Agents know their own position as well as positions of the landmarks. In addition, each agent i observes its own goal g_i which is to direct another agent to one of the landmarks. All the agents are rewarded at once every first timestep one agent reach its assigned landmark. Because of partial observability of goals, agents need to communicate the location of the goal landmark to the goal agent. For that purpose, we allow the agent to emit one token $c \in C = \{1..V\}$ at each timestep which will be visible at the next timestep to everyone. Technically, the action space is now $C \times A$.

To sum up, at each timestep, the observation of agent i is a vector $o_i(t) = [i, x_i, c_{1..N}, g_i]$

The fact that the rewards are shared is essential. Without this, agents would have no incentive to make other agents fulfill their goals.

[1]For simplicity's sake, we allow several agents to be on the same landmark at the same time.

4 Results

For all experiments we use Adam [9] with $alpha = 1.10^{-3}, \beta_1 = 0.9, \beta_2 = 0.999$ and minibatches of size 32. During training we follow an ϵ-greedy policy starting with 1 and decaying lienarly to 0.1 over 5.10^4 steps. We used a replay memory of 25 000 transitions and the target network was updated every 10 000 updates.

4.1 2 Agents Fully Observable

To test the environement and the approach we first train 2 agents, on a fully observable task with 2 landmarks. We tried both using a recurrent layer and without. The architecture used consist of one hidden layer of 32 units, followed by a layer that output one single value per action corresponding to the Q-value of that function. For the recurrent architecture, the hiddent layer is replaced by a lstm layer with the same number of units, and the episode was unrolled for 4 time steps.

For both models the training is quite unstable, and starts to diverge as soon as the optimal policy is almost reached. We're not sure why this behavior occur, we tried to anneal the learning rate but it doesn't seem to work better.

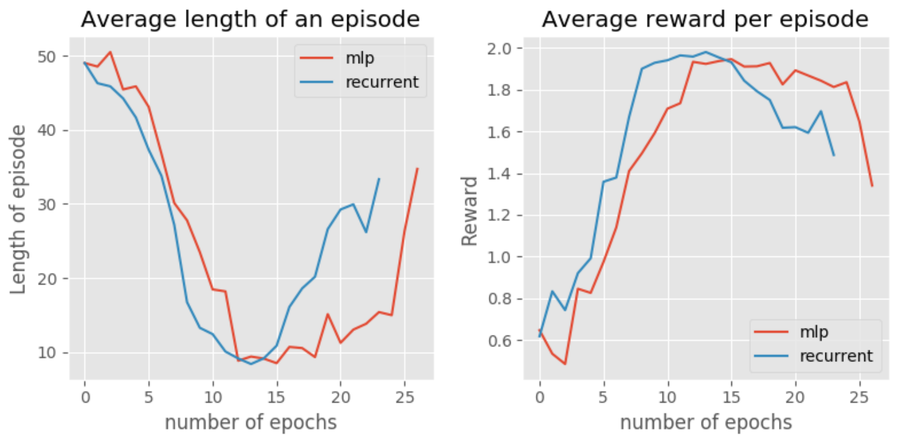

Figure 1: The average length and reward of the different models. The results are averaged over 100 episodes.

4.2 2 Agents Partially Observable

Unfortunately we were unable to make DQN works on that example.

References

[1] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908*, 2017.

[2] Jakob Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2137–2145, 2016.

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1.

[4] Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Recurrent policy gradients. *Logic Journal of IGPL*, 18(5):620–634, 2010.

[5] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[9] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.