Multiagent Systems

Final Report - 2019/2020

Giovanni Bindi

Università degli Studi di Firenze

Abstract

This project addressed the problem of efficiently exploring unseen environments. The aim was to build a framework where an agent can *learn to explore* an environment, visiting the maximum number of states in a limited number of steps. The environments considered are graph-structured state-spaces, since this formalization is able to generalize various applications (e.g. map building). This task was formulated as a Reinforcement Learning problem where the agent is positively rewarded for transitioning to previously unseen environment states.

1 Problem Formulation

Autonomously and efficiently exploring an environment is one of the fundamental researches in robotics, with many applications like SLAM, channel selection in TLC and software testing. This project analyzes a particular exploration setting, where the environment is a known undirected graph and the agent has to reason on it in order to maximize the node coverage in a fixed number of actions.
This exploration process can be seen as a finite horizon Markov Decision Process (MDP), with the number of actions (or *steps*) T being the total budget for exploration. At every step $t \in \{1, \dots, T\}$ the agent observes the graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and a coverage map $c_{t-1} : \mathcal{N} \to \{0, 1\}$ indicating which nodes have been visited so far. For the sake of simplicity only 5×5 grid graphs have been considered, so that the action space $\mathcal{A}$ has a maximum fixed cardinality of 4. These graphs, in addition, contain a variable number of obstacle nodes, i.e. nodes that cannot be visited, hence the agent has to learn how to avoid them during the exploration process.

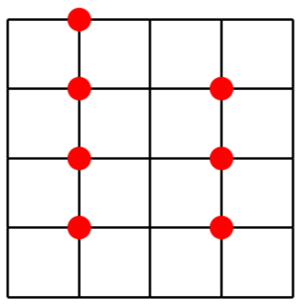


Figure 1: Example of a 5×5 grid graph with obstacles (in red)

Figure 1 shows one of the graphs artificially built for this project. This particular graph has 7 obstacle nodes, drawn in red, and 18 valid nodes (not drawn). Every graph considered in this project is unweighted, although this is not a mandatory assumption, since it only needs a re-definition on the exploration budget T to incorporate the weights of the edges.
The aim of the coverage process can be formalized as the following maximization problem:

$$\max_{\{a_1,\dots,a_T\}} \sum_{n \in \mathcal{N}_v} \frac{c_T(n)}{|\mathcal{N}_v|} \tag{1}$$

where $\{a_1, \dots, a_T\}$ is a sequence of T actions and $\mathcal{N}_v$ is the set of all valid nodes, i.e. the nodes that can be visited. In this project every environment considered had at least 1 obstacle node, hence $\mathcal{N}_v \subset \mathcal{N}$.

1.1 Formalism

A Reinforcement Learning (RL) agent interacts with an environment over time. At each time step t, the agent receives a state s_t in a state space $\mathcal{S}$ and selects an action a_t from an action space $\mathcal{A}$, following a policy $\gamma(s_t, a_t)$, which represents the agents behavior. The agent then receives from the environment a scalar reward r_t and transitions to the next state s_{t+1} according to the environment dynamics, through, respectively, a reward function $R(s_t, a_t)$ and a transition function $\varphi(s_{t+1}|s_t, a_t)$.
In this setting the state space coincides[1] with the set of valid nodes $\mathcal{N}_v$, hence the state s_t can be thought as the node n_t visited at time step t. The action space, in a generic node n, is composed of at most 4 actions: the agent can indeed move *left*, *up*, *right* or *down*.

1.2 Reward

The first (and most natural) reward function that has been considered was simply the normalized difference of visited nodes between the current time step and the previous one.

$$r_t = \frac{1}{|\mathcal{N}_v|}\Big[\sum_{n\in\mathcal{N}_v} c_t(n) - c_{t-1}(n)\Big] = \begin{cases} 1/|\mathcal{N}_v| & \text{if a new node is visited} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

This reward function (and a few variations of it considered) did not suit the training process, as it appeared unable to produce any improvement during the learning phase.
In the end the reward function that produced positive results, at least in the setting proposed, has this sparse definition:

$$r_t = \begin{cases} -0.1 \cdot vis(n) & \text{if } n \in \mathcal{N}_t^* \\ 0.1 & \text{if } n \notin \mathcal{N}_t^* \\ -0.5 & \text{if } n \notin \mathcal{N}_v \end{cases} \tag{3}$$

where $\mathcal{N}_t^* \subseteq N_v$ is the set of visited nodes up to time step t and $vis : \mathcal{N} \to \mathbb{N}$ is a function that counts the number of time that a node n has been visited. This reward function punishes the agent for visiting the same state again, with a penalty proportional to the number of visits, and gives a positive reward when the agent visits a new state. Finally, when the agent tries to reach an obstacle or tries to exit the map a negative feedback of -0.5 is given.

2 Implementation

Deep Reinforcement Learning (DRL) has been successfully applied to a number of problems [1], thanks to the recent advances in GPU computing. In this project DRL, and in particular the Deep Q-Network [2] algorithm (DQN), has been applied in order to experiment with a simple yet non-trivial exploration task. Since the environments considered in this application are graphs, the DRL agent has been equipped with two Graph Neural Networks [3] (GNN) in order to internally encode and represent the graph-structured data.
The Q-network[2] GNN here is used to approximate the optimal action-value function $Q^o(s, a)$, defined as

$$Q^o(s,a) = \max_{\gamma} \mathbb{E}\Big[r_t + \alpha r_{t+1} + \alpha^2 r_{t+2} + \ldots | s_t = s, a_t = a, \gamma\Big] \tag{4}$$

which is the maximum sum of rewards r_t at each time-step t, discounted by $\alpha \in [0, 1]$, achievable by a policy γ when the agent is in the state s and takes an action a. Another identically-structured GNN, namely the target network, is used to estimate the value of the action proposed by the online network. Naming $s = s_t$, $s' = s_{t+1}$, $a' = a_{t+1}$ and $r = r_t$ we can define the *experience* as $e = \{s, a, r, s'\}$, that, at each time-step t, is stored in a data-set $D_t : D_t = \{e_1, \ldots, e_t\}$. In order to reduce the correlation between the samples drawn from the data-set during the learning phase, a mini-batch B is uniformly sampled from D and Stochastic Gradient Descent (SGD) (in our case Adam) is applied on it. The target network then calculates the target value $r + \alpha \max_{a'} Q(s', a')$ and the Q-learning update is operated through the use of the Mean Squared Error loss function:

[1]In this case $\mathcal{S}$ is implemented as the graph configuration in addition to a minimum number of information, such as the number of visits for every valid nodes. This, however, do not drastically voids the simplification that $\mathcal{S} = \mathcal{N}_v$.

[2]Following the nomenclature in [2] the *Q-network* is the one responsible for approximating the action-value function, while the *target* network is the one used for estimating its value.

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\left[\left(r + \alpha \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{5}$$

where an explicit reference has been made on the different weights of the two networks: θ is used to parametrize the Q-network while θ' is used for the target network. Finally, at the end of every learning step, a soft weights update is operated using the following formula

$$\theta' \leftarrow \beta\theta + (1 - \beta)\theta' \tag{6}$$

2.1 Model

The GNN used in this project is a sequential model constructed over the Gated Graph Sequence Networks, a model proposed in [4] that augments the GNN structure with a Recurrent Neural Network. The GNN used in this project implements a mapping $G : (\mathcal{G}, \mathbb{R}^{\nu\times f}) \to [0, 1]^4$, taking as inputs both the graph $\mathcal{G}$ and a $\nu \times f$ feature matrix. This function then outputs a 4D real-valued vector, which contains the approximations for the optimal actions' value.
In our case $\nu = |\mathcal{N}| = 5^2 = 25$ and $f = 7$, indicating the fact that every node has 7 features. These 7 features for the i-th node at time t are indicated by $\mu_i^{(t)} = [x_i, y_i, o_i, vis_i^{(t)}, w, n_i^v, b^{(t)}]$ and are defined as

1. $x_i \in \mathbb{R}$: The x coordinate of the node.
2. $y_i \in \mathbb{R}$: The y coordinate of the node.
3. $o_i \in \{0, 1\}$: A boolean flags indicating if the node is an obstacle.
4. $vis_i^t \in \mathbb{N}$: The visit count for the node.
5. $w \in \{0, 1\}$: A boolean flag indicating if the node is part of the perimetral wall.
6. $n_i^v \in \mathbb{N}$: The number of node's neighbors that have not been visited yet.
7. $b^t \in \{0, 1\}$: A boolean flag indicating if the agent is currently visiting the node.

Features $4, 6$ and 7 are time-dependant, hence the feature matrix $\mathcal{F}^{(t)} = [\mu_1^{(t)}, \dots, \mu_\nu^{(t)}]^\intercal$ evolves through time, reflecting the actions taken by the agent.
The sequential model built for this application is schematized in Table 1, where the `GateGNN` layer refers to the operator defined in [4], `BatchNorm` to a batch normalization layer and `Linear` to a fully connected MLP. The dropout operation was carried with a drop probability of $p = 0.5$ while `GlobalPool` operator simply sums the rows of the input matrix on which it acts on. The ELU [5] function has been used as the activation function.

#	Layer	Input	Ouput	Params
1	`GatedGNN`(7, 4)	(25, 7)	(25, 7)	532
2	`BatchNorm`(7)	(25, 7)	(25, 7)	14
3	`GatedGNN`(64, 4)	(25, 7)	(25, 64)	26598
4	`BatchNorm`(64)	(25, 64)	(25, 64)	128
5	`GatedGNN`(128, 4)	(25, 64)	(25, 128)	164608
6	`GlobalPool`	(25, 128)	(1, 128)	–
7	`Linear`(128, 256)	(1, 128)	(1, 256)	33024
8	`Dropout`	(1, 256)	(1, 256)	–
9	`Linear`(256, 4)	(1, 256)	(1, 4)	1028

Table 1: Network structure

The `GatedGNN` implements a message passing operation that uses a RNN (a GRU [6]) that can be roughly schematized as this couple of equations

$$\begin{cases} m_i^{(l+1)} = \sum_{j\in\mathcal{N}_i} \Theta \cdot h_j^{(l)} \\ h_i^{(l+1)} = \mathrm{GRU}(m_i^{(l+1)}, h_i^{(l)}) \end{cases} \tag{7}$$

that are iterated from $l = 0$ up to $l = L - 1$, where L the number of the sequence length (in our case $L = 4$, being this the second argument of the `GatedGNN` layer). The matrix Θ is the weight matrix and $\mathcal{N}_i$ is the set of neighbors for the i-th node.
The whole implementation was made using `PyTorch` [7] and the Neural Network has been built with `PyTorch-Geometric` [8]

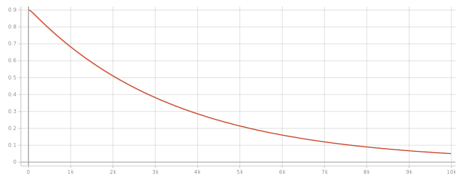
2.2 Experiments and Results

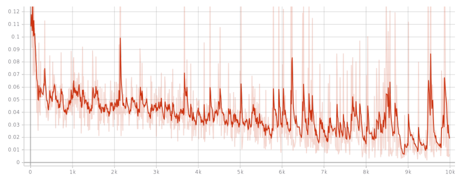
The general training scheme is shown in Algorithm 1, and it is substantially the same as the one proposed in [2] and briefly explained above, which makes use of the experience replay mechanism. Experiments have been made also using the Double Deep Q-Network (DDQN) algorithm [9], but the best results were reached with DQN.

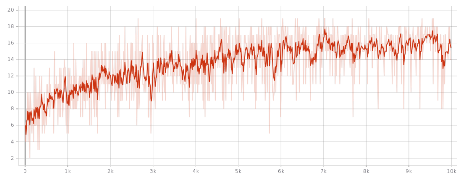
Algorithm 1 : GraphDRL

```
1:  Initialize experience replay memory $\mathbf{M}$
2:  Initialize environment $\mathcal{G}$
3:  Initialize networks weights $\theta, \theta'$
4:  **for** episode $e = 1 \dots E$ **do**
5:      Reset $\mathcal{G}$ and construct initial state $s_0$
6:      **for** step $t = 1 \dots T$ **do**
7:          Select action $a_t$ following an $\epsilon$-strategy
8:          Execute action $a_t$, receive reward $r_t$ and move to state $s_{t+1}$
9:          Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathbf{M}$
10:         Sample a random minibatch $B$ from $\mathbf{M}$
11:         Update $\theta$ by SGD for $B$
12:         Soft update $\theta'$
13: Return $\theta$
```

A total number of 10 environments has been used: 1 for training the agent and the other 9 for testing. These 10 simple graphs have been created artificially for this project. All the experiments have been carried out in the `Google Colab` environment, where a Tesla K80 GPU was available.
Several runs with different hyperparameters gave comparable results, but, due to the limited (yet fairly powerful) amount of resources available, an hyperparameters optimization procedure could not be accomplished. The results shown below may not be the best that can be reached by this model.
The hyperparameters used are $|B| = 32$, $\beta = 10^{-3}$, $\alpha = 0.99$ and a learning rate of 10^{-3}. The total number of episodes is $E = 10^4$ with $T = 25$ for each episode. In Figure 2 are shown a few metrics logged by `Tensorboard`. These metrics are, respectively, the ϵ-decaying policy (which is exponential) (2a), the loss function (2b), the number of nodes visited in an episode (2c) and the reward gained in an episode. These metrics were logged once in 10 episodes and the agent starting node has been randomized at the beginning of every step.


(a) ϵ-decay function


(b) Loss


(c) Nodes visited

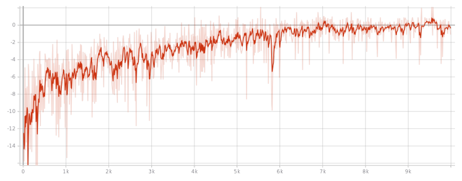

(d) Reward

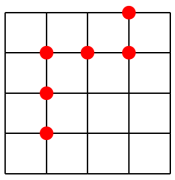
Figure 2: A few metrics from the training phase

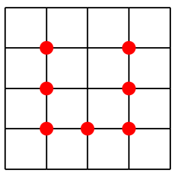
2.3 Testing

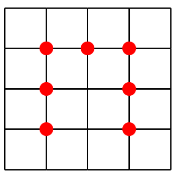
The testing phase has been executed in a similar manner. After the environment initialization the agent started exploring the graph, following exclusively the action proposed by the Q-network. This was done for 30 episodes of 25 steps each, for every environment in the test-set. At the end of every episode percentage of valid nodes visited by the agent has been registered. The results for these tests can be seen in Table 2, which are expressed in terms of mean (μ), standard deviation (σ) and *Best Run*. The latter is the best percentage (highest number of valid nodes visited) during an episode, by the agent.

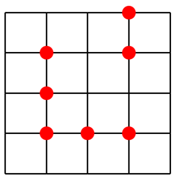
Env	μ	σ	*Best Run*
a*	89.8	8.3	100
b	61.9	**25.6**	88.9
c	78.5	10.7	94.4
d	*55.2*	20.7	88.9
e	74.0	9.5	89.5
f	62.6	18.0	83.3
g	71.4	19.1	88.2
h	63.5	11.1	*81.0*
i	56.0	23.9	85.0
j	**82.1**	*9.2*	**95.5**

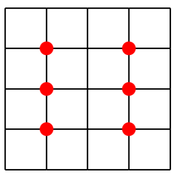
Table 2: Mean and standard deviation wrt the normalized node coverage, for every environment, during 30 episodes of 25 steps each. The graph marked with the asterisk* is the one on which the agent has been trained. The **maximum** and *minimum* for every column are marked in bold and italics, respectively.

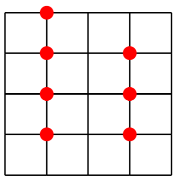

(a)

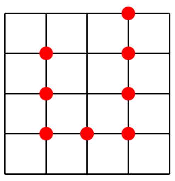

(b)

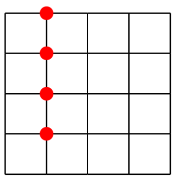

(c)

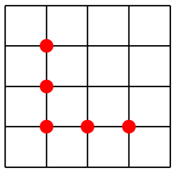

(d)

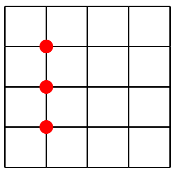

(e)


(f)


(g)


(h)


(i)


(j)

Apart from the training graph (a), we can see from Table 2 that the best results were achieved on graph (j). This is clearly the simplest graph in the set, hence the mean coverage is quite high (82.1%) and the standard deviation fairly low compared to the others. This graph is almost identical to (h), except for the node at the (0,1) position. The performances on this graph were significantly worse and this performance drop might be related to the node (or edge) connectivity.

Despite the simple structure of the environments, the policy obtained with this method seems adequate for being at least a simple baseline for comparisons with more complex algorithms. Deep Reinforcement Learning with Graph Neural Networks (and Geometric Deep Learning in general) is at the first stages of development and so this project can also be a starting point for future and more evolved (and theoretically grounded) experiments.

References

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[3] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[4] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015.

[5] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," 2015.

[6] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1724–1734, Association for Computational Linguistics, Oct. 2014.

[7] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[9] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI16, p. 20942100, AAAI Press, 2016.