Graph Convolutional Reinforcement Learning for Multi-Agent Pathfinding

CS249 Final Project Report - Group 7

Daniel Ahn
UCLA
dahn@g.ucla.edu

Tameez Latib
UCLA
tlatib@cs.ucla.edu

Mia Levy
UCLA
miamlevy@ucla.edu

Howard Xie
UCLA
howardx@cs.ucla.edu

ABSTRACT

Multi-agent path planning in dynamic environments is greatly needed in industrial applications such as deploying a fleet of robots in a warehouse. Our project proposes a multi-agent path planner using evolutionary reinforcement learning on an environment represented as a graph. The evolutionary reinforcement learning approach will help with training stability and performance. The environment represented as a graph is a decentralized approach that will allow agents to take into account other nearby agents when path planning.

KEYWORDS

neural networks, reinforcement learning, path finding, graphs

ACM Reference Format:
Daniel Ahn, Tameez Latib, Mia Levy, and Howard Xie. 2021. Graph Convolutional Reinforcement Learning for Multi-Agent Pathfinding: CS249 Final Project Report - Group 7. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As robots have become cheaper to deploy, they have become increasingly pervasive in industrial applications [2]. The problem with multi-robot deployments is avoiding collisions and making sure they reach their goal position. This problem can be categorized under multi-agent path planning. Multi-agent path planners can be classified as two types, *centralized* and *decentralized.* Robot deployments that use a centralized path planner rely on a central unit that gathers information and plans a path for each robot. This method of path planning is problematic when it comes to scaling up the number of robots in the fleet. Centralized path planning for multiple agents is problematic because it is rather inefficient, often taking seconds to minutes to recompute a path when agents encounter unexpected hurdles. Because of this, centralized multi-agent path planners cannot plan online in a real-world environment that contains noise and uncertainty. A decentralized multi-agent path planner allows agents to be able to perceive its surroundings and make decisions online. In our project, we propose to accelerate a high performing decentralized path planner.

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

2.1 Reinforcement Learning

Reinforcement learning (RL) is a subset of machine learning in which agents take actions in an environment to maximize reward, as defined in the environment. The goal in reinforcement learning is to learn the optimal mapping from states to actions that leads to the maximum cumulative reward. Reinforcement learning is based on Markov Decision Processes (MDPs). In an MDP, each action and state lead to a reward (or lack of reward), and a new state. Each state is assumed to be a Markov state, meaning that each state only depends on the state that came directly before it [5].

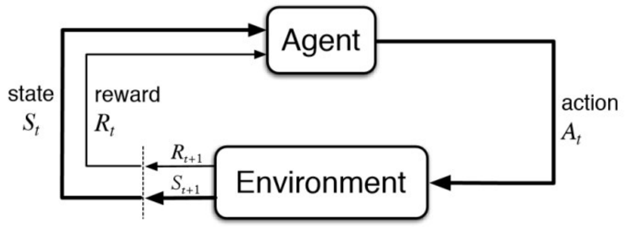


Figure 1: Markov Decision Process.

We would like for the agent to make the sequence of actions that results in the highest cumulative reward. This optimization problem can be formulated as the Bellman equation [5]:

$$Q(s,a) = r(s,a) + \gamma \underset{a}{max} Q(s',a)$$

The cumulative reward is $Q(s,a)$. The immediate reward given a state and action (s,a) is $r(s,a)$. We must choose an action a that maximizes the Q function at state s', which is where we ended up after taking action a at state s. In a deep Q network, a neural network is used to approximate Q values [5].

2.2 PRIMAL and MAPPER

Pathfinding via Reinforcement and Imitation Multi-Agent Learning (PRIMAL) is a decentralized multi-agent path finding (MAPF) algorithm that combines reinforcement learning (RL) and imitation learning (IL) [4]. In PRIMAL, agents learn single-agent paths via RL and learn to imitate a centralized expert using IL. In PRIMAL, each agent has access to a limited field of view around itself.

Much like PRIMAL, Multi-Agent Path Planning with Evolutionary Reinforcement Learning in Mixed Dynamic Environments (MAPPER) is a decentralized partially observable MAPF algorithm that also uses RL [3]. However, MAPPER proposes using evolutionary reinforcement learning to gradually eliminate low-performance policies during training to increase training efficiency and performance. In MAPPER, each agent independently makes its own decisions based on its own local environment and policies. After a certain number of training episodes, the accumulated reward of each agent is evaluated and the agent that performs best propagates its weights to the other agents with a certain probability. The way that we use MAPPER's algorithm in our training pipeline is described in greater detail in section 4.2.

2.3 DGN

Our training framework is primarily based on the Graph Convolutional Reinforcement Learning paper. We will refer to the algorithm described in this work as DGN [1]. DGN is a reinforcement learning algorithm based on deep Q networks, and is supplemented with graph convolution. In DGN, agents can only communicate with their neighbors, as defined by euclidean distance or some other metric depending on the experiment. However, by exploiting convolution with relation kernels, DGN can learn cooperative strategies among agents as the receptive field of each agent grows during convolution. By describing the environment as a graph and using graph convolution, DGN can enhance cooperation between agents compared to traditional reinforcement learning methods.

The DGN architecture can be divided into three primary sections: The observation encoder, the convolutional layer, and the Q-Network. In the observation encoder, local observation is encoded into a feature vector by MLP for low dimensional input, or CNN for visual input. In the convolutional layer, features in the local region (consisting of a local node and its neighbors) are fed as an input, and a latent feature is generated as an output. Convolutional layers are stacked to increase the receptive field of each agent, which increases the scope of cooperation among agents. The final layer is the Q-Network. Feature vectors from the convolutional layers are fed into the Q-Network, which models the policy based on what is input from the convolutional layers.

In DGN, there is a relation kernel that lives in the convolutional layer of its architecture. This relation kernel uses what is formally known as a multi-head dot-product attention as its convolutional kernel, which computes interactions between agents in the input graph. This is one of the keystone ideas that allows multiple agents to keep each other in consideration when taking actions.

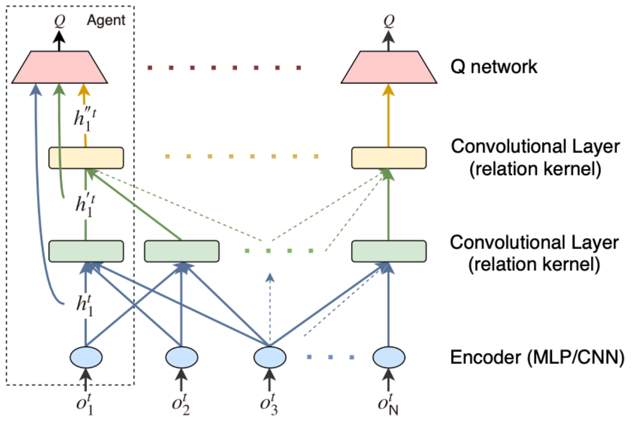


Figure 2: DGN Architecture [1].

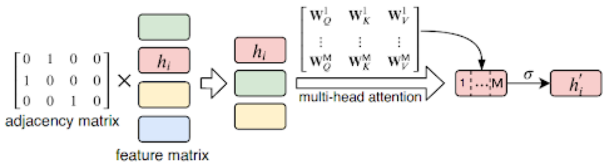


Figure 3: Computation of the convolutional layer with the relation kernel of multi-head attention [1].

3 METHOD

3.1 Environment as a Graph

Most multi-agent path planning problems treat the environment as if it were a 2-D grid world. However, the usage of a graph convolutional network allows for us to encode the environment as a graph, thereby encoding many of the useful qualities associated with this view. In the case of DGN, encoding the environment as a graph helps encode relationships between agents. The graphical view of the environment consists of the destinations of all packets which makes up the vertices. The edges are all connections between vertices that are a one hop distance away from each other.

3.2 Evolutionary Reinforcement Learning

In MAPPER, evolutionary reinforcement learning is used to improve stability and performance during training [3]. First, each of the N agents is initialized with random weights. Next, each agent's model is trained separately using the advantage actor-critic (A2C) method. If agent j has the largest normalized reward of all the N agents, then each agent will either keep its own weights or receive agent j's weights with a probability $1 - p_i$. The lower agent i's reward is, the higher the probability that agent i will receive agent j's weights.

Because we are using DGN's training framework, which trains all the agents jointly with the same set of weights, we cannot directly apply MAPPER's method of passing the best agents' weights to the rest of the agents. Instead, four separate global models are computed during training. Every k training episodes, the genetic algorithm selects the model with the best performance to pass its weights on to the other three models with some probability. This probability is computed in the same way as in MAPPER training.

After k training episodes, global model i will accumulate a reward denoted by $R_i^{(k)}$. We can compute model i's normalized reward as follows, where $R_{max}^{(k)}$ is the maximum reward among the models and $R_{min}^{(k)}$ is the minimum.

$$\overline{R}_i^{(k)} = \frac{R_i^{(k)}}{R_{max}^{(k)} - R_{min}^{(k)}}$$

If model j has the largest accumulated reward after k episodes, then model i will keep its current weights with probability p_i and replace its current weights with model j's weights with probability $1 - p_i$. p_i is computed as follows.

$$p_i = 1 - \frac{exp(\eta \overline{R}_i^{(k)})}{exp(\eta \overline{R}_j^{(k)})}$$

η is the evolution rate, where a higher η makes it more likely that a model's weights will be updated. For each model, a value m is sampled from a uniform distribution between 0 and 1. If $m < p_i$, then model i will get model j's weights.

4 PROBLEM DEFINITION AND FORMALIZATION

For our paper, we consider a routing problem in which agents are data packets that must traverse a graph from a source router to a destination router. The environment is given as a graph $G = (V, E)$ where V is the set of all routers and E is the connections between them. The state of the environment can be seen as the cartesian product of all agents that currently exist within the environment $\mathcal{S} = p_0 \times p_1, \times \cdots, p_n$ where each agent p_i is parameterized by its current node, destination node, and size, (V_{ci}, V_{di}, s_i). At every step, each agent can select an action a within the action space $\mathcal{A}$ which consist of selecting an edge e which is connected to V_c or remaining in the current router. The reward for this problem can be given as

$$r = \begin{cases} 10 & V_{ci} = V_{di} \\ -0.2 & \sum\limits_{j \in e_i} a_j > 1 \end{cases}$$

The agents obtain a reward of 10 if they are successfully able to enter the destination node and a reward of -0.2 if multiple agents attempt to enter the same connection.

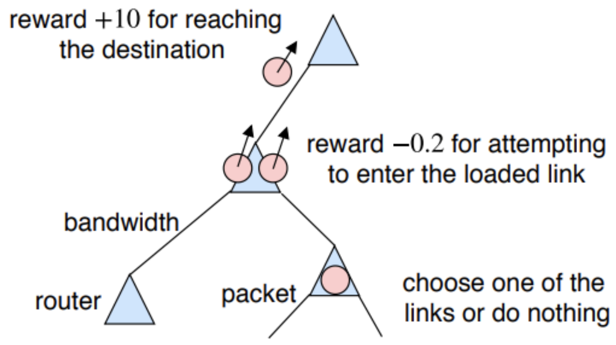
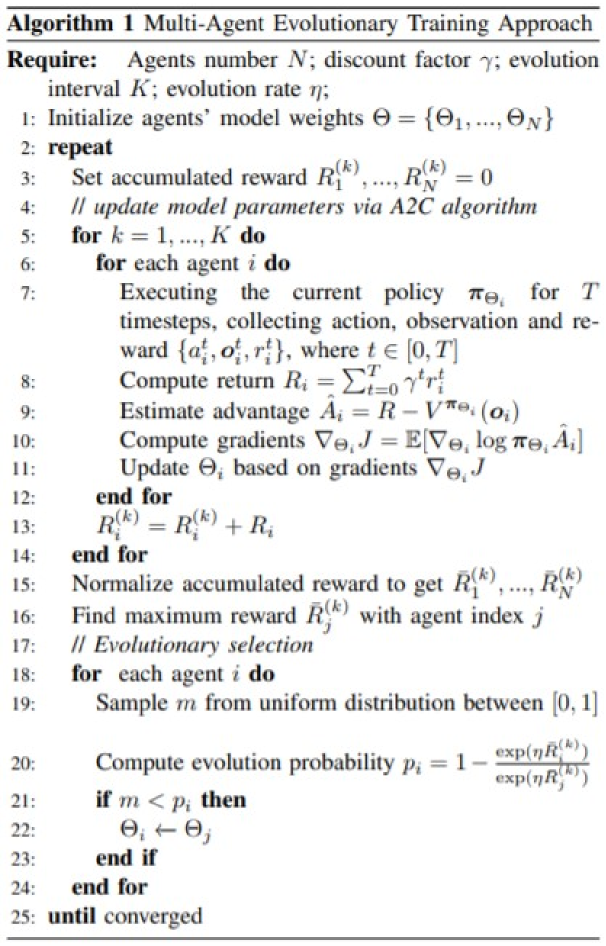

reward +10 for reaching the destination
reward −0.2 for attempting to enter the loaded link
bandwidth
router
packet
choose one of the links or do nothing

Figure 4: Routing experiment from DGN paper [1].

5 EVALUATION

5.1 Training

We model our training based on the idea from MAPPER, shown in the figure below.



```
Algorithm 1 Multi-Agent Evolutionary Training Approach
Require: Agents number $N$; discount factor $\gamma$; evolution
    interval $K$; evolution rate $\eta$;
 1: Initialize agents' model weights $\Theta = \{\Theta_1, ..., \Theta_N\}$
 2: repeat
 3:   Set accumulated reward $R_1^{(k)}, ..., R_N^{(k)} = 0$
 4:   // update model parameters via A2C algorithm
 5:   for $k = 1, ..., K$ do
 6:     for each agent $i$ do
 7:       Executing the current policy $\pi_{\Theta_i}$ for $T$
          timesteps, collecting action, observation and re-
          ward $\{a_i^t, o_i^t, r_i^t\}$, where $t \in [0, T]$
 8:       Compute return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$
 9:       Estimate advantage $\hat{A}_i = R - V^{\pi_{\Theta_i}}(o_i)$
10:       Compute gradients $\nabla_{\Theta_i} J = \mathbb{E}[\nabla_{\Theta_i} \log \pi_{\Theta_i} \hat{A}_i]$
11:       Update $\Theta_i$ based on gradients $\nabla_{\Theta_i} J$
12:     end for
13:     $R_i^{(k)} = R_i^{(k)} + R_i$
14:   end for
15:   Normalize accumulated reward to get $\bar{R}_1^{(k)}, ..., \bar{R}_N^{(k)}$
16:   Find maximum reward $\bar{R}_j^{(k)}$ with agent index $j$
17:   // Evolutionary selection
18:   for each agent $i$ do
19:     Sample $m$ from uniform distribution between $[0, 1]$
20:     Compute evolution probability $p_i = 1 - \frac{\exp(\eta \bar{R}_i^{(k)})}{\exp(\eta \bar{R}_j^{(k)})}$
21:     if $m < p_i$ then
22:       $\Theta_i \leftarrow \Theta_j$
23:     end if
24:   end for
25: until converged
```

Figure 5: Pseudo-code of MAPPER algorithm

Note that the MAPPER paper assumed multiple agents acting independently on the same problem or state. Each agent would then have its own model and its own rewards. So in a N-agent game, there would be N models, where at every timestep we update each agent according to their corresponding model. This is repeated for K steps, i.e. to play our K episodes. Then, the rewards can be normalised and an evolutionary algorithm may begin. In this case, the rewards may be used to find the best agent and scale the probabilities of changing agents. So that a weaker agent will have larger probability of being updated to stronger agent's models. Clearly, every agent will converge to the same model as time approaches infinity.

Naturally, it is reasonable to suggest that this approach is no different than training a single model for all agents, and updating that single model. However, there is a key difference. Because we have multiple agents in different states, these agents will view the world differently. Some agents may be closer to rewards or goals than others. This introduces inherent randomness, that is, different models will train differently based on their somewhat random state. Then, we can take the best agent/model and propagate their findings to their peers.

This idea behind MAPPER cannot be directly translated to the work on DGNs. The main difference is that the DGN approach has a singular model controlling its agents. Therefore our contribution and approach is to swap the loops in lines 5, 6 of the MAPPER algorithm. We have many of these DGN models that work independently. Then, each of these gets to play for K episodes. Once each agent has played K episodes, we then perform reward normalization and can do evolution. In this basic algorithm, we do not share any information between two different models.

The results of this basic algorithm can be seen in figure 10. The idea behind this is that the randomness of each model training with a different minibatch and on a somewhat different state can be beneficial. Since we always take the best model after K episodes of each model, we sort of guarantee that this algorithm will produce better results. The problem is the memory requirement, and the fact that now each 'episode' is actually N episodes.

We improve upon this basic algorithm by increasing the sharing between different models. When we share data such as a replay buffer, we actually see dramatic performance increase. The reason for this is quite simple- with a shared replay buffer, if one model does well its 'experiences' can of playing well can be shared to other models too. This means that not only are weights shared, but experiences are too. This is a similar idea to how GANs start off training very slowly, but as soon as they get somewhat good, they are able to train much faster. It's because the data that we are producing gets better and better. So in this shared algorithm, as seen in figure 11 we actually see a huge improvement because of these shared experiences in addition to taking the max model to propagate.

Lastly, we added an optional 'mutation' parameter, which treats a model as multiple different sub-models. For example in this mutation algorithm we may set the weights of the encoder or Q network separately. This increases the randomness and possibly works better when we have a large amount of agents. However, empirical analysis on this problem shows that the loss scales off around 0.15 for mutation with less than 5 agents. The theory is that with more agents, the randomness might actually be better for learning the most optimal solutions. But, with only a few agents, the probability that mutation is good is not high likely.

5.2 Results

The code outputs graphs that look like figure 6. Note that we have many agents on a graph type structure. Here we use only 20 routers with 3 edges each. This is a limitation of the original paper as well, as using tensorflow limits the graph such that all vertices have exactly the same number of edges. Nevertheless, we test on this same environment to show our improvement from baseline. Note that the code output from 6 moves to 7 after one episode. That is, all agents have moved once in the assigned direction based on the model.

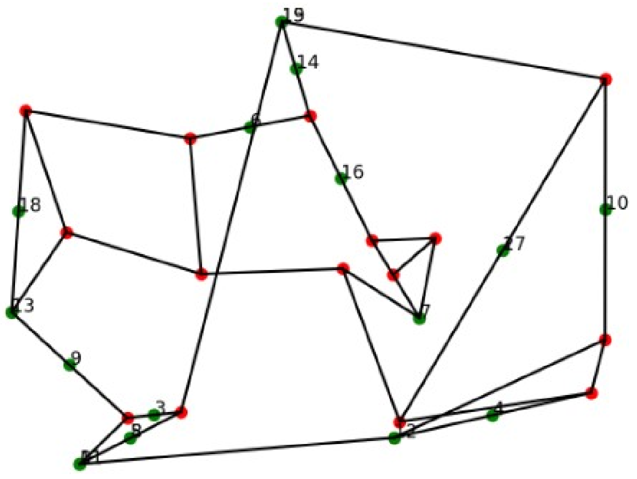


Figure 6: Code Output at t=0

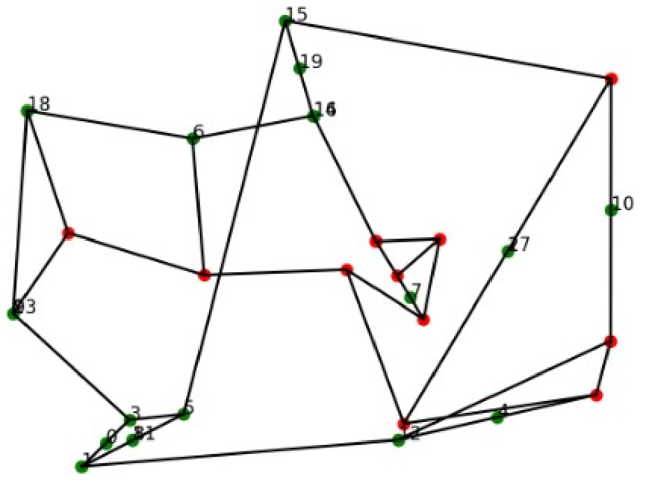


Figure 7: Code Output at t=1, one episode

In order to test our model, we first evaluate a baseline that is from the original DGN paper. Note that we start at loss = 0, this is because loss is actually averaged over previous 100 episodes.

Further, we only start training after 2000 items are in the replay buffer. Figure 2 shows a complete training of the baseline algorithm in the DGN paper, by running the code for >1 day. It also shows that we only need to consider about 20k-50k episodes for our results.

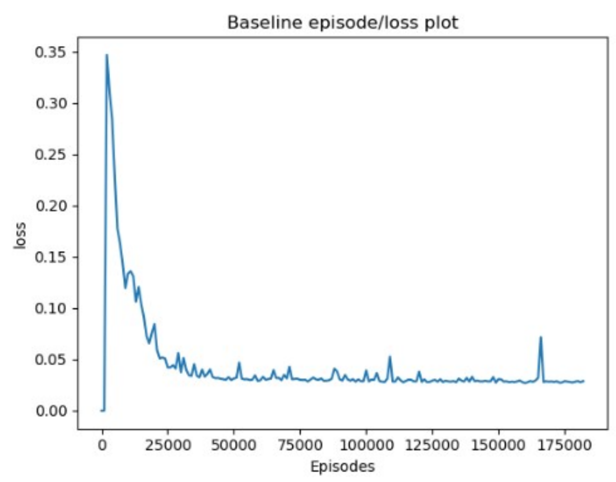


Figure 8: Baseline algorithm

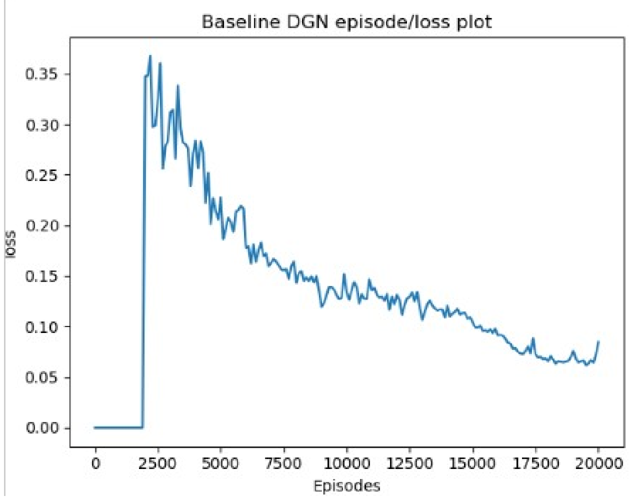


Figure 9: Baseline algorithm, up to 20,000 episodes

A magnified version of the baseline algorithm is shown in Figure 9- we only show up to 20k episodes. It is important to note that these trainings take a very long and we are unfortunately limited on time. Nevertheless, we may compare our first evolutionary algorithm, Figure 10, to Figure 9. Note that in Figure 10 we used 4 agents and set our evolution interval, or our K = 100. Since we modified the algorithm, we also changed some of the control for when the model trains. This means it started training earlier- in the baseline we start after 2000 episodes but here we start after 500 episodes (since 500 * 4 models = 2000 total episodes) That is, every model is run for 100 episodes and then, once all models have run 100 times we do evolution. And then repeat this process. The evolution without sharing is clearly worse off than baseline. While it looks like loss is better after the same number of episodes, we must consider that there are 4 models being trained, so the actual number of episodes is close to 80,000. The exact episode number where Figure 10 (non-sharing algorithm) passed the minima of Figure 9 (our baseline) is around 10,000, which means that we had 40,000 or about double the number of episodes to train to the baseline. This also does not take into account memory consideration. In Figure 11 we used 5 agents, and used the shared replay buffer with everything else kept other hyperparameters constant (from Figure 10). Ideally we should have used 4 agents here too to make a fair comparison, but due to time constraints we cannot re-run the experiment. We also cropped out the first loss = 0 part to make the loss plot look neater.

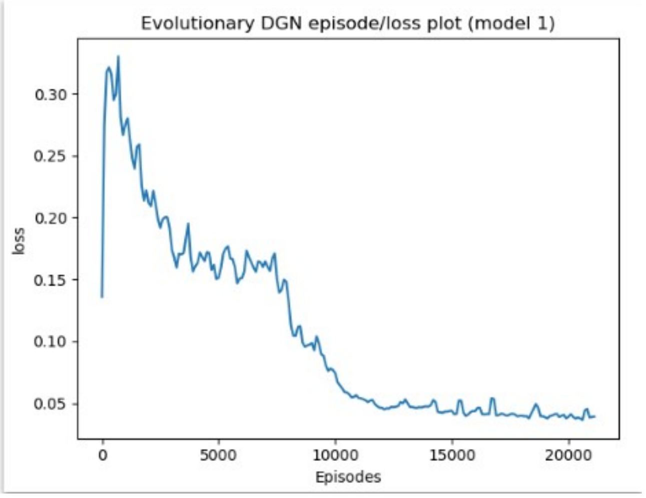


Figure 10: Evolutionary algorithm without information sharing

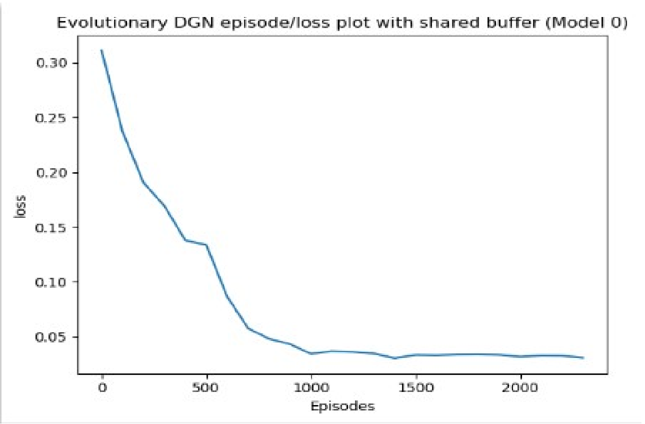


Figure 11: Evolutionary algorithm with shared replay buffer

However, our technique with shared buffer actually does show improvement. Note that the shared replay buffer conforms to the theory of training faster. We only have 10,000 (2000 episodes per agent, 5 agents total) episodes of experiences in the replay buffer, but because of the quality of these experiences, we train dramatically faster than the baseline algorithm. In addition to high-grade experiences, modifying our model to be best of 5 differently trained models further decreases the loss and improves training.

6 CONCLUSION

Our contribution is specifically adding an evolutionary algorithm inspired by MAPPER to a DGN approach. We see an improvement in our end result of multi-agent path planning at the cost of memory. With only 10,000 total episodes (spread among 5 agents), we achieve better loss than the baseline does at 20,000 episodes. While this is only a small-scale experiment, we hope future work can expand upon these findings by verifying similar results on different problems. For example, by choosing different states, rewards, and possibly a different environment. Further exploration can delve into different hyperparameters or genetic algorithm. Nevertheless, it is reasonable to suggest that our contribution can be applied to similar problems with the expectation of improved results.

REFERENCES

[1] Jiechuan Jiang, Chen Dun, Tiejun Huang, and Zongqing Lu. 2020. Graph Convolutional Reinforcement Learning. arXiv:1810.09202 [cs.LG]

[2] Conghui Liang, K.J. Chee, Y. Zou, Haifei Zhu, Albert Causo, Stephen Vidas, T. Teng, I-Ming Chen, K. H. Low, and C.C. Cheah. 2015. Automated Robot Picking System for E-Commerce Fulfillment Warehouse Application.

[3] Zuxin Liu, Baiming Chen, Hongyi Zhou, Guru Koushik, Martial Hebert, and Ding Zhao. 2020. MAPPER: Multi-Agent Path Planning with Evolutionary Reinforcement Learning in Mixed Dynamic Environments. arXiv:2007.15724 [cs.RO]

[4] Guillaume Sartoretti, Justin Kerr, Yunfei Shi, Glenn Wagner, T. K. Satish Kumar, Sven Koenig, and Howie Choset. 2019. PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning. *IEEE Robotics and Automation Letters* 4, 3 (Jul 2019), 2378–2385. https://doi.org/10.1109/lra.2019.2903261

[5] Shaked Zychlinski. 2020. Qrash Course: Reinforcement Learning 101 & Deep Q Networks in 10 Minutes. https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677