Multi-Agent Path Finding by Reinforcement Learning Algorithm

Yifan ZUO
yzuoah@connect.ust.hk

Abstract

Multi-Agent Path Finding is a challenging problem in artificial intelligence, where a group of agents with different goals must navigate a shared environment without colliding with each other. In the past, the search algorithm constructs a tree of possible future states and actions that the agents can take. The algorithm then evaluates the resulting paths to find the optimal solution. This approach has been applied to MAPF, with popular algorithms such as A and its variants, including Conflict-Based Search (CBS) and its extensions. Compared to tree search, the DQN approach can handle more complex and dynamic environments, as it learns from experience rather. Also, DQN can handle continuous state and action spaces, whereas tree search is often limited to discrete spaces. However, DQN requires a large amount of training data and can suffer from the problem of credit assignment, where it is difficult to attribute rewards to specific actions in a multi-agent setting In recent years, deep reinforcement learning has shown promising results in solving complex decision-making problems, including MAPF. However, most existing RL-based methods for MAPF are decentralized, where each agent learns its policy independently without considering the actions of other agents. This can lead to suboptimal solutions and longer training times. In this project, we propose a centralized deep Q-network approach for MAPF, where all agents share a single neural network and learn a joint policy. Our method uses a centralized critic to estimate the value function of the joint state-action space and updates the network parameters using a variant of the DQN algorithm.*

1. Introduction

Background. Multi-Agent Path Finding (MAPF) is an important area of research in Artificial Intelligence and Robotics, which focuses on finding optimal or near-optimal collision-free paths for multiple agents navigating in a shared environment. In recent years, deep learning techniques such as Deep Q-Networks (DQN) have shown remarkable success in solving complex single-agent reinforcement learning problems. Centralized DQN approaches have been proposed for solving MAPF problems, where the joint action-value function is learned over the entire state space of all agents. This enables the agents to cooperatively learn efficient strategies for navigating in a shared environment.

Motivation. In this work, we present a novel Centralized DQN-based approach for solving MAPF problems, incorporating the variant Manhattan distance heuristic to guide the exploration process and improving the efficiency of the learning algorithm. The Manhattan distance heuristic is a widely used admissible heuristic in path-finding algorithms, which estimates the total cost of reaching the goal from a given state by summing the absolute differences in the x and y coordinates of the current position and the target position. Our proposed approach combines the power of deep reinforcement learning with the efficiency of heuristic search, leading to a robust and scalable solution for MAPF. The heuristic will be speeding the training process.

Approach Description. We used the Playing Atari with Deep Reinforcement Learning algorithm [1] as a base structure, it has already been shown to be effective in learning complex decision-making tasks. By fine-tuning the algorithm with a heuristic specific to MAPF, we can leverage the strengths of both the DQN algorithm and the heuristic to achieve even better performance.

2. Problem Definition

2.1. Define the State Space

We define the state space consists of a 10 by 10 grid map which can be viewed in Fig 1, where each cell in the grid is assigned a value to encode its status. The encoding scheme you proposed is as follows:

Item	Encoding
Free grid	1
Block	2
Agent	100
Goal	5

Table 1. State Space Encoding

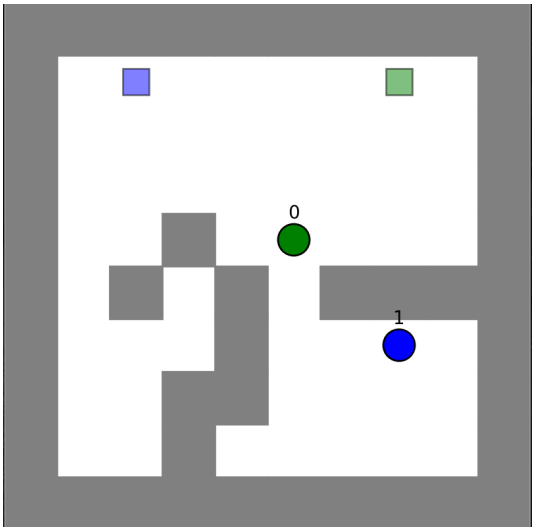

Figure 1. Grid map

The use of non-zero values to encode the map is a good idea to avoid zero gradient issues in the training process. Additionally, encoding the agent with a large number compared to other encodings is also a good strategy to emphasize the importance of agent position information in the training process. Note that for multi-agent, we use one hundred adding its index to indicate its position in the state map in order to differentiate agents. It's also worth noting that the map boundary is all block, which means that agents cannot move outside the boundaries of the map. This is a common assumption in grid-based MAPF problems and simplifies the problem by avoiding the need to handle edge cases at the boundaries of the map. The state space definition provides a comprehensive representation of the environment and allows the DQN algorithm to learn an optimal policy for the agents to navigate the map and reach their goals.

2.2. Action Space Definition

Based on my definition of the action space, it consists of five possible actions that each agent can take to navigate the environment. These actions are:

Action	Action encoding tuple
Go Up	(-1, 0)
Go Down	(1, 0)
Go Right	(0, 1)
Go Left	(0, -1)
Stay	(0, 0)

Table 2. Action Space Encoding

Each action is represented as a tuple of integers that indicate the direction that the agent should move. The "Stay" action corresponds to remaining in the current position. To determine the agent's position after taking an action, you can add the current position tuple to the action tuple. For example, if the agent is currently at position (3, 4) and takes the "Go Up" action, the resulting position would be (2, 4). However, the joint action space for multiple agents in a MAPF problem can grow exponentially with the number of agents. Specifically, if there are n agents in the environment and each agent has m possible actions, then the joint action space can be represented as m^n. For example, if there are four agents in the environment and each agent has five possible actions, then the joint action space would be $5^4 = 625$ possible joint actions. This exponential growth in the joint action space can make it difficult to find an optimal solution using traditional search-based methods. However, the use of reinforcement learning, such as the DQN algorithm, can help to address this challenge by allowing agents to learn a joint policy that maximizes the total reward over time.

3. DQN and Method Design

3.1. Neural Network Architecture

The overall algorithm can be viewed in Fig 2. The neural network in the DQN has three fully connected layers: an input layer, a hidden layer, and an output layer. The input to the network is a flattened representation of the current state of the environment, which includes the positions of all agents. The input layer has 100 nodes, which corresponds to the size of the flattened state representation. The first hidden layer has 20 nodes, and the second hidden layer has 40 nodes. Both hidden layers use the ReLU activation function, which is a non-linear activation function that introduces non-linearity into the network and makes it capable of learning complex functions. The output layer has size of joint action space number of nodes, which corresponds to the number of possible joint actions that can be taken by all agents. The output layer is not activated, which means that the output values are not constrained to a particular range. The network is initialized with random weights using a normal distribution with a mean of 0 and a standard deviation of 0.1. This helps to break the symmetry between the nodes and prevents the network from getting stuck in a local minimum.

3.2. Experience Replay

Experience Replay is a technique used in reinforcement learning to improve the efficiency and stability of learning. It works by storing past experiences (i.e., state-action-reward-next state tuples) in a replay buffer and randomly sampling a batch of experiences from the buffer to train the neural network. In my code, the replay buffer is implemented using a deque (double-ended queue) data structure [2], which allows efficient appending and popping of

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2. DQN Algorithm

elements from both ends of the queue. The buffer has a maximum size, which is specified by the parameter MEMORYCAPACITY. At each time step, the agent stores the current experience (i.e., the current state, action, reward, and next state) in the replay buffer using the append method.

here are more details about the compound data type we defined for the transition and the memory data structure, we defined a compound data type transitiondtype using the numpy dtype function. This data type specifies the structure of a single transition, which consists of five fields:

data	type
state	np.float64, (1, NSTATES)
actions	np.int64
reward	np.float64
nextstate	np.float64, (1, NSTATES)
done	bool

Table 3. Transition Data Definition

3.3. Epsilon-greedy Exploration

The epsilon-greedy exploration strategy is a simple and effective way to balance between exploration and exploitation in reinforcement learning. In my code, the epsilon-greedy strategy is implemented in the choose action method of the DDQNAgent class, which selects an action based on the current policy with probability epsilon and selects a random action with probability 1-epsilon. After a certain number of episodes, which is specified by the parameter, I increase the value of epsilon by 0.1 to make the agent explore less. This means that the probability of selecting a random action decrease, and the probability of selecting the action with the highest Q-value decreases. Increasing epsilon from low to high can be a useful strategy to encourage exploration and prevent the agent from getting stuck in a suboptimal policy. However, it should be done carefully, as increasing epsilon too quickly or too much can lead to poor performance and slow learning.

3.4. Target Network and Evaluation Network

The purpose of having both a target network and an evaluation network in Deep Q-Networks (DQN) is to improve stability and convergence during training. In DQN, the evaluation network is used to select actions during training, and its parameters are updated at each iteration using backpropagation. The problem with using a single network is that the Q-value estimates for the next state are also updated at each iteration, and this can lead to unstable training and oscillations in the learned Q-values. To address this issue, DQN introduces a second network, called the target network, which is a copy of the evaluation network with fixed parameters. During training, the target network is used to estimate the Q-value of the next state, and its parameters are not updated. The target network is updated periodically by copying the parameters of the evaluation network. This decouples the Q-value estimates used to select actions from the Q-value estimates used to compute the target values during training and helps to stabilize the training process. In summary, the evaluation network is used to select actions during training, and its parameters are updated at each iteration using backpropagation. The target network is used to estimate the Q-value of the next state and its parameters are fixed and updated periodically by copying the parameters of the evaluation network. The use of a target network helps to stabilize the DQN training process and improve convergence. In our design, after 100 times learning, assign the evaluation network to target network

4. Reward Function and Heuristic

4.1. Reward Function Design

The reward function is a critical component of any reinforcement learning algorithm, as it helps the agent to learn the optimal behavior by assigning values to the different actions taken in the environment. The goal of the reward function is to provide feedback to the agent about the desirability of its actions and guide it towards the optimal policy. In the context of the Agent class, the reward function assigns a value to each action taken by the agent based on the current state of the environment. The reward values are used to update the agent's Q-value estimates during training and to select the best action to take in each state.

The reward function in the Agent class assigns different values based on the following conditions: If the agent collides with an obstacle or another agent, a -5 reward is assigned to discourage such behavior. This is because collisions can cause delays and lead to suboptimal paths. If the agent reaches the goal, a positive 100 reward is assigned to encourage this behavior. This is because reaching the goal is the desired outcome of the task. If the agent is still moving towards the goal and has not collided with any obstacles or other agents, a small -0.5 reward is assigned to encourage

the agent to reach the goal as quickly as possible. This is to avoid the agent taking an unnecessarily long time to reach the goal. In summary, the reward function in the Agent class assigns values to different actions based on the current state of the environment and uses a heuristic distance value to guide the agent toward the goal efficiently. The reward function is a critical component of the reinforcement learning algorithm, and it plays a crucial role in guiding the agent toward the optimal policy.

4.2. Heuristic

The heuristic distance value is a measure of the estimated distance between the agent's current position and the goal position. In the Agent class, the heuristic distance is calculated using the Manhattan distance between the agent's current position and the goal position. The Manhattan distance is the sum of the absolute differences between the x and y coordinates of the agent's current position and the goal position. The heuristic is defined to be the subtraction value of the distance of the current position to the goal and the next position to the goal. The heuristic distance value helps the agent reach the goal by providing a measure of how close or far the agent is from the goal. During training, the agent's Q-values are updated based on the reward values assigned by the reward function. By incorporating the heuristic distance value into the reward function, the agent is encouraged to take actions that bring it closer to the goal. For example, consider an agent that is moving towards the goal but encounters an obstacle. The collision with the obstacle results in a negative reward value, which encourages the agent to avoid such collisions in the future. Additionally, the heuristic distance value is also updated based on the agent's new position, and this encourages the agent to take actions that move it closer to the goal while avoiding the obstacle.

4.3. Fine-Tune Heuristic

For the original heuristic, there is an issue of it. The heuristic value becomes smaller as the agent approaches the goal, This is because the heuristic distance measure used in the Agent class is based on the Manhattan distance, which is the sum of the absolute differences between the x and y coordinates of the agent's current position and the goal position. As the agent gets closer to the goal, the Manhattan distance becomes smaller, and hence the heuristic value becomes smaller.

Regarding the issue I mentioned about agents not knowing how to move when approaching the goal, this is a common problem in multi-agent systems where the agents' actions influence each other. In such systems, the agents may get stuck in local minima and fail to reach the optimal solution. we fine-tuning the heuristic distance value, using a function like $1/x$ can indeed help ensure that the heuristic value is small initially and becomes larger when approaching the goal. This can be useful in situations where the initial heuristic value is too large and causes the agent to take suboptimal actions. By fine-tuning the heuristic value, the agent can be encouraged to take more efficient paths toward the goal and avoid getting stuck in local minima.

5. Performance

Our project using centralized DQN with the heuristic method was able to achieve double agent path-finding! But agents are trying to achieve goals together since it gets the maximal reward, which can solve the general case of the pathfinding problem. However, it's true that in some cases, such as when agents need to go through the same point at the same time, the approach may not work as well. This is because the agents' actions are highly interdependent, and it may be difficult to assign credit for a particular outcome.

Regarding the three-agent path-finding problem, the credit assignment problem may indeed be more challenging in this case, especially if the agents' actions are highly interdependent. One way to address this issue is to use more sophisticated learning techniques, such as multi-agent reinforcement learning (MARL).

6. Limitation and Future Improvement

In a multi-agent system where all agents share a single network and receive a common reward, the actions of one agent may influence the actions of other agents. This is known as the credit assignment problem, where it may be difficult to determine which agent(s) should receive credit for a particular outcome. As the number of agents in a system increases, the credit assignment problem becomes more challenging, and it may be difficult for the agents to learn an optimal policy. In such cases, it may be necessary to use more sophisticated learning techniques, such as decentralized learning or multi-agent reinforcement learning. Decentralized learning involves training each agent independently using local information, without any direct communication or coordination with other agents. This approach can be effective in some cases, but it may lead to suboptimal results if the agents' actions are highly interdependent. Multi-agent reinforcement learning is a more sophisticated approach that allows agents to learn a joint policy that takes into account the actions of other agents. In MARL, each agent has its own local policy, which is trained using a decentralized approach. However, the agents also communicate with each other and share information about their observations and actions, which allows them to coordinate their actions and learn a joint policy. There are several approaches to MARL, such as centralized training with decentralized execution, where a centralized critic is used to train the agents' policies, but each agent executes its pol-

icy independently. Another approach is decentralized training with decentralized execution, where each agent trains its own policy using local information, and the agents communicate with each other to coordinate their actions.

7. Meeting Schedule

date	duration	meeting mode
2023 February 9^{th}	30mins	in person
2023 March 8^{th}	35mins	in person
2023 April 7^{th}	30mins	in person
2023 May 11^{th}	30mins	in person
2023 May 14^{th}	35mins	remote

Table 4. Meeting Time Schedule

References

[1] 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. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 1

[2] Zhihu. Deep reinforcement learning: from entry to depth. https://zhuanlan.zhihu.com/p/260703124, 2021. 2