CSE 410 Final Project: Multi-Agent Reinforcement Learning

Peter M. VanNostrand
Department of Computer Science
University at Buffalo
Buffalo, NY 14261
pmvannos@buffalo.edu

Abstract

In this project we implement a custom multi-agent reinforcement learning environment, then use this environment to explore the development of 2-4 cooperative reinforcement learning agents via tabular Q-Learning and Deep Q-Networks. We conclude with by presenting an evaluation of both agent types and discussing the benefits displayed by Deep Q-Network Agents.

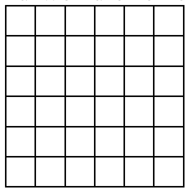
1 Problem Definition

In this project we explore the world of multi-agent reinforcement learning by creating a custom multi-agent learning environment then training a series of agents in this space.

1.1 Environment

The environment chosen for exploration of multi-agent reinforcement is a grid world. Starting with the grid world developed in assignment one, we extended this space to a square grid of size six as shown below.

Grid-world of size six



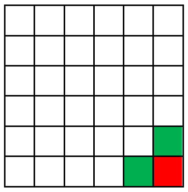
In this environment an agent can be in one of 36 possible states which are referred to by a row column pair. The rows are labeled as 0-5 from top to bottom and the columns are labeled as 0-5 from left to right. This makes the top left state $s = [0,0]$ and the bottom right state $s = [5,5]$

Within this environment an agent can choose from one of five actions, these are: move up, move right, move down, move left, and no move. Actions are indexed 0-4 in the order listed. The next state is selected deterministically based upon the current state and the provided actions. Actions which would result in an agent leaving the board, or colliding with another agent result in a transition back to the current state. The observation space of this environment is defined as the current row and column location of the agent.

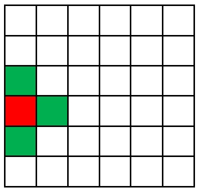
1.2 Task

Within this space we define the task of "enemy containment." This task is defined as follows: within grid-world one state will contain a static "enemy," the goal of the agents will be to cooperate in locating and surrounding the enemy such that it would be unable to move. For example, the following figures demonstrate possible containment schemes for n agent environments $n \in 2,3,4$ with the cooperative agents shown in green and the static agent shown in red

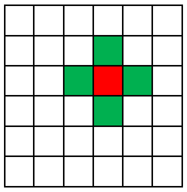
Enemy Containment for $n = 2$



Enemy Containment for $n = 3$



Enemy Containment for $n = 4$



This model could represent one of several possible real-world scenarios, such as placing roadblocks in a city grid to isolate the scene of a crime, but currently the most applicable scenario would be the containment of a viral outbreak. Here we could model each state as a city block or a hospital bed in an emergency ward containing infected individuals, the agents would then be tasked to determined in what locations testing stations could be placed to isolate the infected community or individuals.

1.3 Stochastic Games

Multi-agent reinforcement learning is broadly similar to single agent learning, however it introduces a number of complexities which make learning more difficult. In single agent reinforcement learning we formulate a task in terms of a Markov Decision Process (MDP). A MDP consists of a set of states S which define the environment, a set of possible actions A which the agent may take, some reward function $R(s \in S, a \in A)$ which maps the current state and action to a reward, a discount factor γ, and a transition probability distribution P which determines the next state s' based on the current state s and the action a.

In multi-agent reinforcement learning (MARL) we can extend this formulation to a stochastic game. A stochastic game is the combination of a MDP with a repeated game. A repeated game is a task in which multiple agents must consider the impact of their actions on the actions of other agents in the space, this can take the form or cooperative or competitive play. Mathematically we define a stochastic game similarly to a DMP: there are a set of N agents, each with a set of actions A_i to choose from, the total set of n independent actions form the joint action A, which a reward function $R(s, a)$ maps to a reward based upon the current state of the environment. In some stochastic games, agents can communicate with each other by passing information relating to their current state or intended actions. This allows for intelligent cooperation or communication between agents. In this environment agents will communicate with each other by providing other agents with their current location in the grid-world. This will enable agents to learn how to navigate the space in a way that avoids collisions. In addition to their current state s agents will also take in communication from other agents c. While s is a finite single pair of row and column coordinates, the communications c will grow with the number of agents as each communication will contain the row and column of each of the $n - 1$ other agents. This gives the communication a length of $c = 2(n - 1)$, which will add dimensionality to this task.

1.4 MARL Challenges

One challenge introduced by the formulation of a stochastic game is the moving target problem. In MARL the reward an agent receives can be dependent on the actions of other agents in the space, that means that for a single agent, taking action a from state s could result in wildly different rewards depending on the state and actions of other agents. As all the agents are learning in parallel the reward for this case is likely to change over time as other agents act unpredictably during training.

Another significant challenge arising from MARL is the curse of dimensionality. Essentially, as more agents are added to a space the possible number of states and actions increases linearly. This linear growth in state and action space can lead to an exponential growth in computational complexity during learning, this can be seen in the increased dimensionality of tables required during Q-Learning. With exponentially higher dimensionality there are exponentially mores state-action pairs to explore and a matching larger number of possibilities for discrete reward, these combined to force agents to learn significantly more information which can impact the speed of their learning. For large problems the extraordinarily high dimensionality of data can require alternative approaches to handle the increased memory demand on each agent.

2 Implementation

2.1 Environment

2.1.1 State and Action Modifications

To begin I started with the single agent deterministic grid-world and agents I defined as part of assignment one. I first expanded the world from size 4 to size 6 which was relatively trivial. Then I added an action option for not moving and remaining in the same place, this a no move is critical for this task as agents which reach a space adjacent to the enemy must remain there to contain the enemy. These modifications were made easier by the use of the OpenAI Gym environment, the state space was simply modified from a box of size 4x4 to a box of size 6x6, and the action space was expanded from 4 discrete actions to 5 discrete actions. To accommodate the fifth action an additional else-if statement was added to the movement function to allow for no move actions.

2.1.2 Vectorization

The next set of modifications was significantly more difficult. As there was now n agents with $n = 2,3,4$ I had to track down every instance in which the environment interreacted with the agent, and vectorize every operation to be performed. Rather than storing a reference to a single agent, the environment now stores a vector of n agent references. Operations on the agents were also modified to be performed iteratively, with each agent being moved and updated sequentially.

Handling the movement of multiple agents necessitated the introduction of "valid" and "invalid" movements. In the single agent environment, it was relatively simple to ensure that all actions the agent could take would move the agent without a problem as the bounds of the world were the only significant limitations on movement. However, in the multi-agent case this becomes more complex. For a realistic interpretation of the environment we make it impossible for an agent to occupy the same location as another agent or the enemy. Any movement which would result in such an overlap is deemed "invalid" and the agent is prevent from moving into that location. To enforce this I check the new location of each agent against the location of all other agents and the enemy. As agents are handled sequentially movement priority is given to earlier agents, that is to say that if two agents attempt to move into the same location the agent which appears earlier in the list of references will be moved and the later agent will remain in its current location.

2.1.3 Communication

In this environment each agent will receive the location of itself as the current state, as well as location of the other agents as a form of communication between the agents. This is done to allow the agents to make intelligent decisions about their actions, given the context of the other agents' locations. Therefore, the communicated location of the other agents is essentially additional observation information, so to implement inter-agent communication I simply expanded the observations returned by the environment to contain the location of ever agent rather than just a single agent. Observations are returned as a set of n pairs, with each pair representing the row and column location of the corresponding agent. This formulation made development of the agents significantly easier than implementing a more complex real time communication interface between the agents.

2.1.4 Termination Conditions

The environment is allowed to run until all n agents reach a distance of one from the enemy. At this point the enemy is considered fully contained, and the environment returns a done status. If this is not achieved within 18 timesteps, the environment terminates early and returns a done status to prevent agents from wandering indefinitely during training.

2.1.5 Reward Function

To reflect the new task of this environment the reward function was modified from that of assignment one. In this case each agent is punished based upon the difference in distance from the enemy as a result of its action. When an agent moves away from or maintains a constant distance to the enemy it received a reward of -2. When an agent moves closer to the agent it receives a reward of -1, and when an agent is directly adjacent to the agent it receives a reward of 0. This is summarized in the reward function below

$$r(d_t, d_{t+1}) = \begin{cases} -2 \; if \; d_{t+1} \geq d_t \\ -1 \; if \; d_{t+1} < d_t \\ \;\; 0 \; if \; d_{t+1} = 1 \end{cases}$$

This reward function incentivizes agents to approach the enemy, and then stay in a location immediately adjacent to it. The use of negative rewards helps to incentivize the agents to find the shortest possible path from their staring position to a position that contains the enemy. A reward of zero during the containment phase allows agents to stay at a position adjacent to the enemy and avoids accruing very large rewards as the agent remains in this position for an extended period of time while other agents move to complete the containment. This helps to keep the total reward between each agent relatively consistent and helps prevent unduly favoring agents which start closer to the enemy and therefore spend more time immediately adjacent to it.

2.2 Agents

To explore MARL two different types of agents were implemented, a tabular Q-Learning agent was developed with modification from the agents in assignment one, and a Deep Q-Network (DQN) agent was derived from the DQN agent implemented for assignment two.

2.2.1 Tabular Q-Learning Agent

Starting with the Q-Learning agent from assignment one I creating a Tabular Q-Learning agent capable of cooperating with n other agents to solve this task. While the actual bellman update equation used for training of the agent was kept the same, the formulation of the agents Q-Table was modified significantly to allow for multiple agents. In the single agent case the agents Q-table consisted of a three-dimensional array with the first dimension representing the agents row, the second the agents column, and the third the possible actions. However, this is insufficient for the multi-agent case. Here those same three dimensions are needed, as well as $2(n-1)$ additional dimensions to represent the location of the other agents. Using this structure of Q-Table allows the agent to learn different behaviors depending on the location of the other agents, at the cost of a higher dimensional space to explore and learn. Additionally, as the number of dimension of the Q-table is dependent on the number of other agents in the space some clever iterative dereferencing was needed to selected the correct set of actions for the give combination of agent positions.

2.2.2 DQN Agent

Staring from the DQN Agent of assignment two, much fewer modifications were needed to allow for multiple agent cooperation. As the agent stores its learned values in a neural network rather than a Q-Table I simply expanded the number of input nodes of the network and left the remainder of the agent largely unchanged.

2.3 Training

While the training loops from assignments one and two were largely applicable to this task, a few considerations were made for the presence of multiple agents. Firstly, the update, remember, and replay functions were mapped across the n agent at each time step. Secondly, the training hyperparameters were adjusted. To account for the more complex training environment and larger training duration was needed, with 1000-10000 iterations being performed. To allows for this longer training period the ϵ-decay process was modified to have a minimum value of $\epsilon_{min} = 0.03$. At the start of training ϵ changes as normal via exponential decay, then once the value of ϵ drops below 0.03 it is clamped to this value. This helps the agents to explore this larger and higher dimensional space more effectively. The discount factor γ was also raised to 0.95 to allow the agents to learn longer term action sequences.

3 Results

Using the described environment and agents, I trained both agents for $n = 2,3,4$ cooperative agents. During training I recorded the score, that is the total cumulative reward, of each agent for each epoch. Both agent types were trained for 1,000 iterations with $\alpha = 0.05$ and $\gamma = 0.95$. Exponential ϵ-decay was used with $\delta = 0.005$, $\epsilon_{floor} = 0.03$. This yields the following epsilon decay curve for all agents in all training cases.

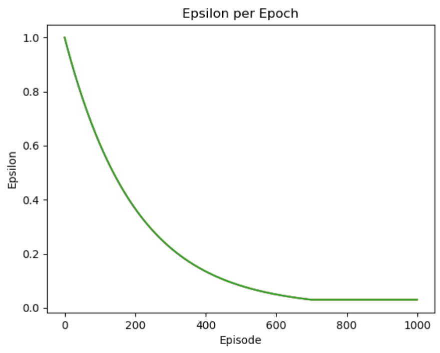


Figure 1: Epsilon Decay Curve for All Agents

The score metrics for each case are plotted below alongside the final learned movements. For clarity score plots also include an average line which represents the score averaged over a moving window of size 10.

3.1 Results $n = 2$

3.1.1 Tabular Q-Agents

I first trained two Tabular Q-Learning (TQ) agents in the environment described and determined the following score metrics and learned movements.

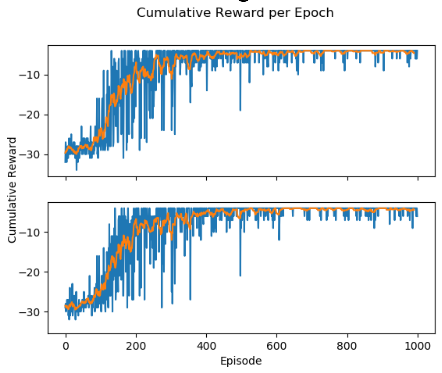


Figure 2: TQ Agents Scores, n=2

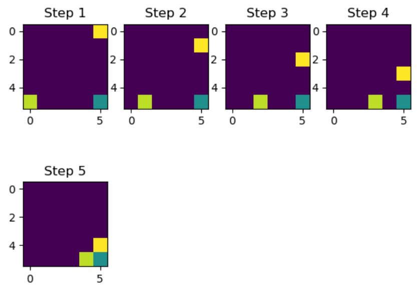


Figure 3: TQ Agent Movements, $n = 2$

Here we can see that the tabular Q-agents begin to learn an optimal path around epoch 100, and ultimately converge to an optimal solution. In the movements rendering the goal is shown in teal in the bottom right corner and the agents are shown starting from the top-right and bottom-left corners in yellow and light green respectively.

3.1.2 DQN Agents

I repeated the above training with two DQN agents and received the following results

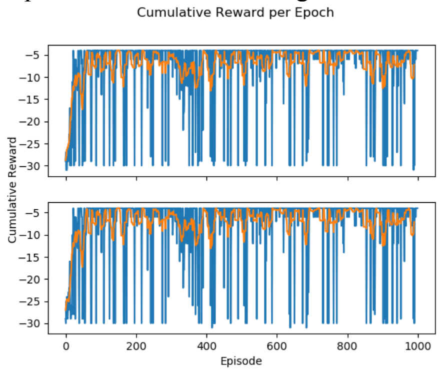


Figure 4: DQN Agent Scores, n=2

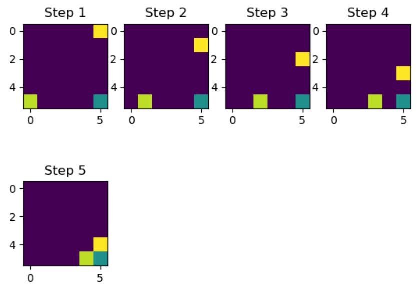


Figure 5: DQN Agent Movements, n=2

Comparing these results to the tabular agent results we can see that the DQN Agents begin learning an optimal path much earlier than the tabular agents, while these results are moderately noisier than the tabular agent scores we can see that the DQN agents have learned the same optimal path in a much shorter period of time.

3.2 Results $n = 3$

3.2.1 Tabular Q-Agents

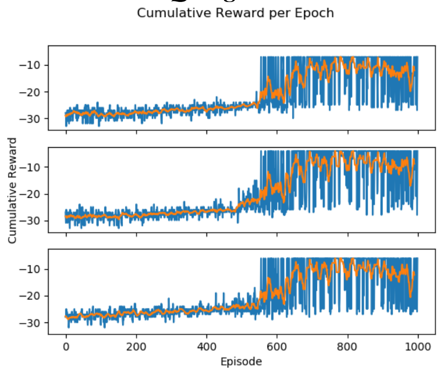


Figure 6: Score for TQ Agents, $n = 3$

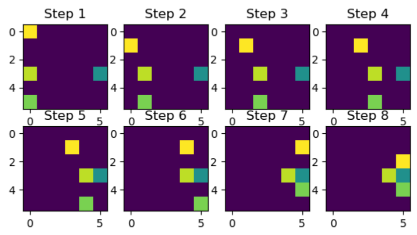


Figure 7: DQN Agents Movements, $n = 3$

3.2.2 DQN Agents

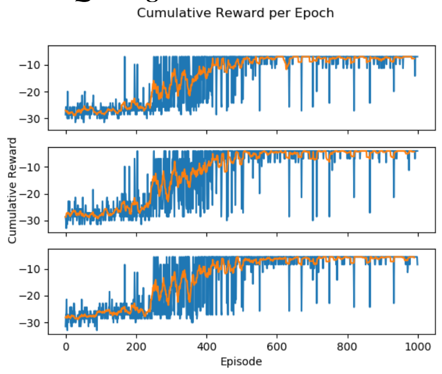


Figure 8: DQN Agent Scores, $n = 3$

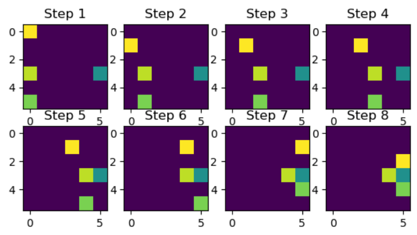


Figure 9: DQN Agent Movements, $n = 3$

For the $n = 3$ case we can see very similar results. Both the Tabular Q-Agents and the Deep Q-Network Agents learn optimal paths from their starting positions to the containment positions. However, the DQN agents do this much more quickly than the TQ agents. Here the agents are show in yellow, and two shades of green with the agent shown in teal.

3.3 Results $n = 4$

3.3.1 Tabular Q-Agents

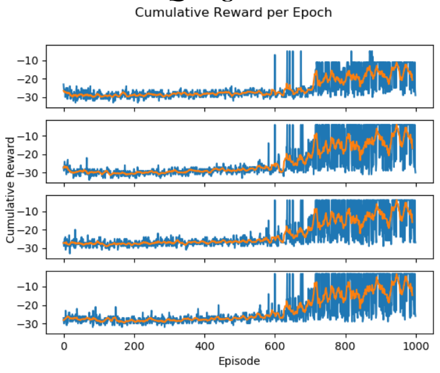


Figure 10: TQ Agent Scores, $n = 4$

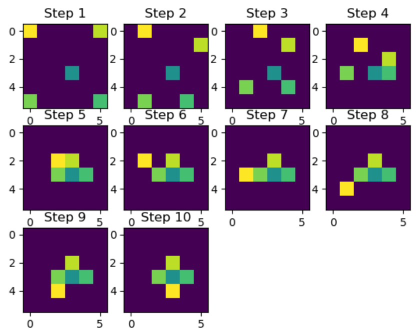


Figure 11: TQ Agent Movements, $n = 4$

Placeholder

3.3.2 DQN Agents

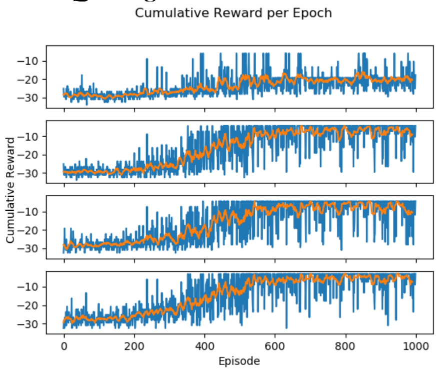


Figure 12: DQN Agent Scores, $n = 4$

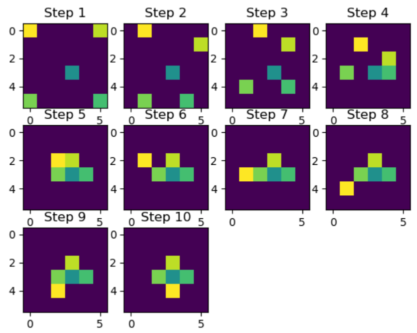


Figure 13: DQN Agent Movements, $n = 4$

These trends hold for the $n = 4$ case, again the DQN agents learn the same paths as the TQ agents, just more quickly. Here the disparity is quite large with the tabular agents only staring to improve their score around epoch 600, compared to the DQN agents increasing score around epoch 250.

3.4 Discussion

Across all numbers of agents $n = 2,3,4$ we found that the Deep Q-Network Agents and the Tabular Q-Learning Agents both learned successful paths from their staring positions to appropriate containment positions. However, in all cases the DQN agents did this more quickly, as evidenced by their comparatively larger scores at earlier epochs in the training process.

The decreased learning time of DQN agents is likely because these agents learn the environment space more efficiently. As the TQ agents store their Q-values in discrete tables, the learning of one state is largely independent of the learning of the other states, and with the high dimensionality of this environment it can take a long time to visit and learn every state-action combination. In contrast when the DQN agent updates its weights during training, that update effects the output Q-values for all state-action pairs. This allows the DQN agents to apply learning from one state to another by effectively interpolating what would be the best action using weights trained on other states. This increases the sample efficiency of the DQN agents and allows them to learn an equally good solution to this task in a shorter amount of time. It's likely that this advantage would become even more important as the size of the environment and the number of agents increases due to the curse of dimensionality.