Q-Network Policy Selection For Multiagent Learning

By
Joshua Cook

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Mechanical Engineering
(Honors Associate)

Presented June 5, 2019
Commencement June 2019

AN ABSTRACT OF THE THESIS OF

Joshua Cook for the degree of Honors Baccalaureate of Science in Mechanical Engineering presented on June 5, 2019. Title: Q-Network Policy Selection For Multiagent Learning

Abstract approved:

Kagan Tumer

Controllers for robotic systems can be complex and difficult to write by hand. Learning offers an approach to improve a controller through direct feedback. Learning is not trivial, as the feedback does not tell the agent how to improve, only how well its current actions solve the given task. Learning in sparsely rewarded domains increases the difficulty of learning as the agent receives less feedback to learn from. This is compounded in multiagent domains which require complex coordination to complete the global objective. Despite the fact that dense rewards are typically easier to learn from, they are not always easy to define; many problems are inherently sparsely rewarded. This work presents an algorithm for learning complex coordination in sparsely rewarded multi-agent domains. The algorithm is split into two steps. The first step involves learning a set of skill, with the second step learn when to use each skill. Experimental evidence is presented, showing the effectiveness of the presented algorithm in a modified version of the rover domain. The algorithm also seeks to provide more explainable learned policies than traditional black-box learners.

Key Words: Reinforcement Learning, Multiagent, Multi-Reward, Sparse Reward

Corresponding e-mail address: cookjos@oregonstate.edu

Q-Network Policy Selection For Multiagent Learning

By
Joshua Cook

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Mechanical Engineering
(Honors Associate)

Presented June 5, 2019
Commencement June 2019

Honors Baccalaureate of Science in Mechanical Engineering project of Joshua Cook presented on June 5, 2019

APPROVED:

Kagan Tumer, Mentor, representing Robotics

Geoff Hollinger, Committee Member, representing Robotics

Connor Yates, Committee Member, representing Robotics

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

Joshua Cook, Author

Contents

1 Introduction

Writing controllers for robotic systems can be difficult. One reason for this is that the controller designer may not anticipate all of the problems the robot may encounter. A rover deployed on Mars for exploration is likely to experience difficulties due to the fact that the controller could not be effectively tested in the Martian environment. In this scenario, a learning-based controller may experience an unforeseen problem and then learn to better solve the problem during future encounters.

Reinforcement learning is a subfield of machine learning in which agents learn to maximize a reward received by taking some action in their environment. This type of learning is useful for solving complex control problems where a controller is difficult to write by hand. In these types of problems, it is generally easier to provide a metric which describes how well the agent or controller completes the desired task. This metric is the reward the agent receives from the environment. The agent seeks to improve the reward received through learning [1].

An example problem, which reinforcement learning could be applied to, would be an exploratory rover tasked with resource collection. Writing a controller for this type of agent would be difficult, as the agent would experience a variety of unknown scenarios which would be impossible to program for. It would be much easier to define a reward function which describes the performance of the rover, then have the agent learn to maximize the reward received. The agent would receive a large positive reward for collecting resources and a much lower reward when it is not. Over time, the agent would improve its performance through learning and gather resources more effectively in a variety of scenarios.

Many types of problems exist which are challenging for modern reinforcement learning algorithms. One of these problems is learning in multiagent systems with a shared goal. In these settings, each agent receives the same reward signal which represents the global reward or progress in completing the shared task. As a result, the problem of credit-assignment arises as each agent cannot differentiate which agent's actions contributed to the global reward received. Without the ability to directly correlate an action to a reward, the agent cannot learn [13].

An example of this would be an extension of the rover problem with multiple rovers. There may be a large rock which would require two rovers to move. If five rovers are present, and two rovers move the rock, all rovers would receive a positive reward for successfully moving the rock. From the rovers' perspective, it is unclear who contributed to the task due to the fact that every rover received a reward. As a result, the rovers who did not contribute may learn that they should not contribute the next time, due to the fact that in previous experience it yielded a positive reward. This would prove to be unhelpful if the rovers who did contribute were not present the next time the problem was encountered.

Another challenging type of problem is learning from sparse rewards. Using a dense reward, a policy can be evaluated and improved at every time step. In a sparsely

rewarded setting, the agent infrequently receives a reward. With these sparse rewards, learning becomes difficult as there is less information provided to the agent. Despite the fact that sparse rewards typically increase the difficulty of learning, they may provide more utility. In some cases, it may be impossible to define a dense reward, but trivial to define a sparse reward [2].

The rover problem is an example of a problem in sparse rewards are used. Resources could be separated by large distances, causing the reward received to be infrequent. A dense reward could not be used to evaluate the agent due to the inability to evaluate the agent's actions when the agent is not collecting resources. A sparse reward would be well suited to evaluate the agent as the agent discovers resources in a sparse manner.

Yet another difficulty is learning in continuous state-action spaces. Learning a mapping from a continuous state to a continuous action which maximizes the reward received, typically requires a fair amount of search due to the large search space. In robotics, these problems are important because often the state input is determined from a series of continuous sensors and the action output consists of a series of continuous values representing desired actuator speeds or positions. Determining a mapping of noisy sensors to a motor controller can be difficult as a model of the system dynamics is needed, valid assumptions need to be made, and the agent can be found in scenarios unexpected by the designer. These problems are solved by an agent which iteratively improves its control policy via learning [2].

Being able to understand the reasoning an agent uses to determine an action can be important in many domains. Many modern reinforcement learning algorithms involve training an end-to-end black-box agent, in which the agent learns to map a state to an action. These algorithms produce policies which provide no explanation of their method of determining an action. Due to this, is difficult for a human to trust the learned black-box policy [14]. An example would be an autonomous vehicle learning to drive. If the vehicle had a black box learned policy, it would be difficult for an engineer to trust that it would drive a person to a location safely. The engineer wouldn't know why it turned the wheel the to angle chosen, or why it would press the gas pedal a specific distance. In the event of a crash, an engineer would understand which actions were taken leading to the crash, but not why the actions were taken. Without understanding why the vehicle crashed, the engineer would have no intuition as to how to prevent further incidents other than to retrain the model. This is not a feasible method for solving every problem the learned policy may contain. If the model could be understood by a human, then the engineer may be able to determine the cause of the incident was that the vehicle decided to change lanes instead of turning.

In this work, we present an algorithm to learn from sparse reward signals in a multiagent domain with a continuous state-action space. The algorithm is divided into two parts. The first part of the algorithm learns a list of policies which represent the skills an agent needs to solve the problem. The second part of the algorithm

learns when each skill should be used to maximize the expected reward received. Additionally, the algorithm presented generates learned policies which are easier to explain than many black box learners.

2 Background

2.1 Reinforcement Learning

Reinforcement learning is centered around the concept of a learning agent. An agent contains sensors which with it can view the world via a state representation (S) of the world. The agent also contains actuators which allow it to interact with the environment by performing an action (A). The agent also receives a reward (R) which acts as a metric, defining how well the agent is performing in its environment. The overall goal of reinforcement learning is to learn a mapping of state space to the action space which maximizes the reward at each step in time, given by the reward function. Generally, the reward function provides a strong positive reward to incentivize good behavior, and a strong negative reward to penalize poor behavior [1].

This decision process is known as the Markov decision process (MDP). An important property to maintain in an MDP is the Markov property. This property states that the state representation provides all of the information needed to take an optimal action. With this property, the agent only needs the current view of the world to take an optimal action. As a result, there exists an optimal control policy which can be represented as a mapping of the state space to the action space [3].

One problem associated with the Markov property is that each reward is associated with an action-state pair at a given moment in time. One action previously taken could have a significant impact on the state the agent is currently in and the reward the agent currently receives. To account for this problem, the discounted reward function is used. Equation 1 represents the discounted reward function [3].

$$R_0 = \sum_{t=0}^{n} \gamma^t R(a_t, s_t) \tag{1}$$

Using this function, the rewards received by the agent are modified so that a portion of future reward is assigned to the action derived from the current state. Small values of γ cause the agent to become myopic, prioritizing short term rewards. With larger values, the agent prioritizes actions which maximize its future reward. Using this modification of the reward function, the agent still holds the Markov property, while taking actions which maximize its cumulative reward instead of each momentary reward.

One of the challenges in reinforcement learning is determining the amount of time spent exploring versus the amount of time spent refining current knowledge. Exploration allows for new areas of the state-space to be discovered, but it does not

allow for fine improvement. Exploitation, however, has the opposite effect, improving the existing policy performance. A mixture of exploitation and exploration allows the agent to discover new policies, while also improving upon existing knowledge.

2.1.1 Q-learning

Q-learning is a common form of reinforcement learning in discrete and finite state-action spaces. This is partially due to the guarantee that, given enough time, Q-learning will learn an optimal policy. In Q-learning the direct mapping of state and action to a reward, depicted in Equation 2, is learned.

$$Q : S \times A \rightarrow R \tag{2}$$

In discrete domains, the predicted rewards are stored in a Q-table with one axis representing a list of possible states and the other axis representing a list of possible actions. The learned policy consists of selecting the action with the highest predicted reward from the Q-table at state s_t. The entry in the table is then updated via the Equation 3. In this equation, α represents the learning rate [4].

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha R(s_t, a_t) \tag{3}$$

This implementation of Q-learning focuses on greedily selecting an action using current knowledge. This is problematic as it selects actions which provide decent rewards, while other actions may never be selected. As a result, there may be an action which provides the highest reward, but is never selected as the Q-learner always selects the action which provides a decent, but not optimal, reward. To improve this, there is a small probability in which the agent chooses an action at random instead of choosing from the Q-table. These random actions allow for exploration for a small portion of the time, while the Q-table greedily selects a majority of the actions [4].

2.2 Neural Networks

The artificial neural network was loosely inspired by the neural connections in the brain. Neural networks consist of a series of interconnected nodes or neurons. These neurons are organized into a series of layers, with each neuron in a layer connected to the neurons of the next layer. Figure 1 presents an example of a neural network structure [5].

In a typical feedforward neural network, information is input via the input layer. The information is multiplied by the weighted connections, summed at the next layer of nodes, and then transformed by an activation function as shown in Equation 4 [5].

$$a_j = g\left(\sum_{i=0}^{n} w_{ij} a_i\right) \tag{4}$$

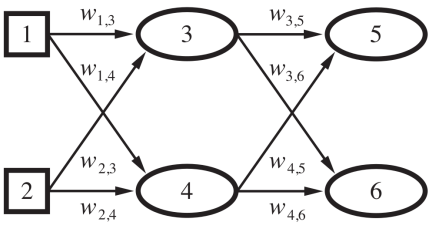


Figure 1: An example of a neural network consisting of input, output, and hidden layers of size 2. Nodes 1 and 2 represent the input layer, 3 and 4 the hidden layer, and 5 and 6 the output layer. Each layer is connected by weights w [5].

This process is repeated for each hidden layer until the information reaches the output layer. The activations of the output layer represent the output of the network. The weights of the connections of each layer can be stored as a series of weight matrices. These weight matrices are multiplied one after another to the input vector, along with activation functions applied in between, transforming the vector until it reaches the output layer. The purpose of the activation function in this process is to perform non-linear transformations on the activation vectors. Equation 5 represents the sigmoid function, an activation function which scales output to be between zero and one. This allows the neural network to learn non-linear functions. These non-linear transformations and a large enough hidden layer allow the neural network to approximate any continuous function [7].

$$g(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

2.2.1 Gradient Descent

An effective method of training a neural network in a supervised manner is through stochastic gradient descent. In this algorithm, a loss function is needed to evaluate the accuracy of the output of the neural network. Equation 6 shows the mean squared error loss function. Here, y_i represents the output of the neural network and $\hat{y}_i$ represents the correct value the network is attempting to predict.

$$L = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{6}$$

The gradient of the loss function is taken with respect to each weight vector. The weights are then updated with the update rule described by Equation 7. This process is repeated for each training sample until the neural network achieves a sufficiently small error [6].

$$w \leftarrow w + \eta \nabla_L \tag{7}$$

2.3 Genetic Algorithms

Genetic algorithms are part of a family of optimization algorithms which attempt to mimic the evolutionary optimization of genetic information in cells. This method is a gradient-free method, allowing for optimization in domains without a clear gradient to search. In order for a genetic algorithm to be successful, it needs a genetic representation of the solution, a population of solutions, a fitness function, and the application of genetic operators [8].

A fairly common genetic representation is a list of values which describe the solution to the problem. A population of these genes is stored and is represented as a bag of solutions. The fitness function is used to evaluate and assign scores to each gene based on the quality of the gene as a solution.

Given a population of scored genes, the genetic operators can be used to generate a new and potentially better population of genes. The first operator performed is selection. Using this operator, the best genes are kept and poor genes are removed. These best genes become the parents for the next generation of genes. In terms of search space, selection greedily saves the best outcomes and drives search in that direction.

Variation is added to the new genes via mutation. In the mutation operator, each segment of an offspring gene has a small chance of either being perturbed of randomly changed. Mutation is a form of exploration of the search space. This exploration allows new segments of genes to be created, which are not currently found in the populations. Given a valid genetic representation of the solution and a decent fitness function, the genetic operators typically can find high-quality solutions to many problems [8].

2.3.1 Neuroevolution

Instead of using gradient descent to train a neural network, neuroevolution seeks to find a set of optimal weights or topology using an evolutionary algorithm. In neuroevolution, the weights of the network represent the genetic information of the system. In supervised systems, the fitness function is typically the same as the negative loss function. In reinforcement learning, the fitness function is the same as the reward function. Mutation typically occurs as a perturbation of the weights of the network or modification of the topology. The exploration due to mutation accounts for the loss of exploration due to crossover. Lastly, a variety of types of selection can be used to select the best neural networks. Experimental evidence has found that neuroevolution can generate policies which have competitive performance with many modern reinforcement learning algorithms [9] .

2.4 Q-Network

A Q-table can provide a best discrete action, given a discrete state input. Due to this fact, a Q-table cannot effectively handle continuous state inputs. The state input could be discretized, but this could increase the size of the Q-table to the point that learning an effective Q-table becomes an intractable task. One method proposed to overcome this continuous state space obstacle is to have a neural network learn to model a Q-table, known as a Q-network. The Q-network returns the expected rewards for each action, given a state input. Effectively, the neural network returns the row of expected rewards given a state input [10, 4].

Instead of online learning from (S, A, R) tuples, as with traditional Q-learning, the tuples are stored in a replay buffer, which is randomly sampled from at training time. This has been empirically proven to increase the stability of the Q-network training. Conceptually, if the Q-network learned via pure online learning, recent experiences would be prioritized and older experienced rewards would be overwritten or "forgotten". The replay buffer prevents "forgetting" knowledge by forcing the neural network to model a larger set of experiences [10].

2.5 Multiagent Learning

Multiagent systems require learning a policy for each agent in a system. Agents in these systems are similar to agents in traditional reinforcement learning, with a modification to the reward function. Often, the agents much work together to achieve a common goal or complete a global objective. As a result, a single global reward function evaluates the population of agents. Each agent then receives the same reward signal. An example would be a series of robots organizing a warehouse. One of the objects in the warehouse may be too heavy for one robot to move, requiring the help of two others. The agents may need to learn to cooperate in moving the large object to its new location. The agents would then be rewarded based on the distance the large object is from its desired location [11].

Multiagent systems can be trained using homogeneous policies or heterogeneous policies. Heterogeneous policies are easier to learn because there is one policy learned and one reward function to optimize. As a result, there is a direct correlation to the actions the policy chooses and the reward received. Homogeneous policies are more difficult to learn, not only because the search space is larger, but because there is one global reward and multiple policies to learn. Due to the shared nature of the global reward, the problem of credit-assignment arises. With each agent receiving the same reward signal, it becomes difficult to determine which agent's contribution lead to the reward received. Learning is then challenging as it is unclear which action or actions taken amongst the agents lead to the reward received [16].

Despite the fact that homogeneous policies can be easier to learn, they may not be sufficient to solve the problem. With heterogeneous policies, agents can learn separate "roles" or more specific policies instead of one very general policy. In the

case of warehouse organization problem, some agents may learn to adopt a support role in which, the move around helping other agents move objects which are too heavy to move on their own. These support agents would then make the task of moving heavy objects easier for other agents [16].

2.6 Multi-Reward Learning

Multi-reward learning involves learning in a domain which offers multiple reward signals instead of a single reward signal. Multiple reward signals represent multiple objectives to learn to achieve. One proposed method of learning in this multi-objective setting is to learn one policy for each reward signal. Each policy then seeks to maximize the reward received from their respective reward signal. Once these policies are learned, the agent needs to learn which policy to use at a given time to optimize the overarching objective function. One recently proposed method for selecting policies in discrete state-spaces involved learning a Q-table. The Q-table was used to map the state representation to a list of expected rewards, one for each policy. The Q-table was also tested in a sparse reward setting. The overall learned policy converged to an optimal solution very quickly [12]. This method was not used in this work for comparison due to the need of a discrete state space.

3 Methodology

3.1 Learning Algorithms

3.1.1 Sub Policy Learning

The first piece of the learning algorithm involves learning a series of policies. These policies represent a series of "skills" which the agent would need to solve the overarching problem. These policies should be general enough that they perform the same expected skill in a variety of scenarios. Given these policies and the assumption the agents know when to select the policies, the agent should be able to solve the overall problem. If an agent needs an additional skill to solve the problem, another policy should be learned and added to the list. Because each policy represents a different learned skill, a separate hand-crafted reward function is needed to evaluate each policy. Algorithm 1 provides the outline for the policy training algorithm used in the experiments mentioned in this paper. In this work, neuroevolution is used to train the policies but other reinforcement learning algorithms could be used, assuming general policies are learned.

From an explainability standpoint, each policy performs a low-level skill. The policies act as the low-level controllers for the agents, generating continuous control given a continuous state.

Algorithm 1: Policy Trainer

foreach $r \in RewardFunctions$ **do**
    Initialize policy π_r
    Train π_r via neuroevolution to maximize reward received from r
    Store π_r
end

3.1.2 Policy Selector Learning

The next part of the learning algorithm involves learning which policy to use at each state. To accomplish this task, each agent contains a Q-network to model the expected reward received for each policy, given the agent's state view of the world. The agent selects the policy with the highest expected reward and uses it to take a series of actions given the same state representation. After a set number of time steps in which the policy is used, a new policy is selected using the same method. To encourage exploration, there is a small probability that a random policy will be selected instead of the policy selected by the Q-network. If the probability of selecting a random policy is sufficiently small, the policies chosen from one episode to the next will mostly be the same. Conceptually, these small changes allow the agent to better associate their change in policy selected with the potential change in global reward.

Each experienced state, policy chosen, and discounted global reward are stored in a replay buffer. The global reward is discounted with a γ of 1.0 in sparse reward settings, allowing the neural network to predict the expected global reward for the episode. This lets the Q-network learn effectively in sparse reward settings. At the end of each episode, the agent randomly samples from the replay buffer and learns from these samples using gradient descent. Algorithm 2, describes the training process for the Q-networks.

Algorithm 2: Policy Selector

for q ... N_{agents} **do**
  Initialize Q-Network with weights ϕ_q
  Initialize Replay Buffer D_q
end
Load learned policies π
foreach *Episode* **do**
  Reset Environment
  for $t = 1...T$ **do**
    for $q = 1...N_{agents}$ **do**
      Observe State Representation $S_{t,q}$
      if $t \bmod 10 \equiv 0$ **then**
        I=*Argmax*$(Q(\phi_q, S_{t,q}))$
        With small probability, randomly select valid I
      end
      $A_{t,q} = \pi_I(S_{t,q})$
      Take action $A_{t,q}$
      Receive Global Reward R_t
      Store $(S_{t,q}, I, R_t)$ *in* D_q
    end
  end
  for $q = 1...N_{agents}$ **do**
    foreach $(S, I, R) \in sample(D_q)$ **do**
      r=$Q(\phi_q$,$S)$
      $r_I = R$
    end
    Perform Gradient Decent w.r.t. the L2 Loss Function $(r - Q(\phi_q,S))^2$
  end
end

The output of a Q-network is fairly easy for a human to understand. It effectively predicts the reward that the agent would receive for each policy. The agent then selects the policy or "skill" with the highest expected reward. The Q-network acts as the high-level control policy, deciding when to use each policy from it's averaged previous experience.

3.2 Domain

The domain in which this learning algorithm is tested is a modified version of the rover domain as seen in Figure 2. In the rover domain, a team of agents is represented as rovers in a foreign location. These agents have homogeneous capabilities and functionalities. There are points of interest (POI) which the agents must observe.

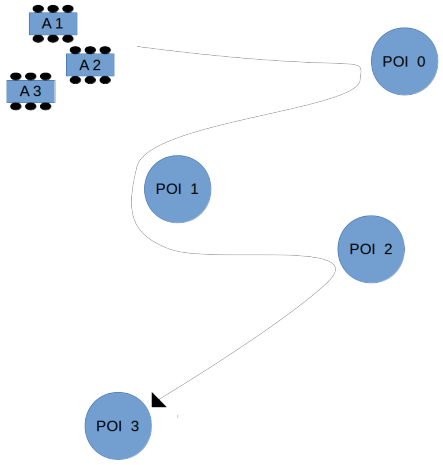


Figure 2: This is a generalized representation of the modified rover domain. A series of agents (A1-A3) must view POI 0 through POI 3 as a team. The curved arrow represents a plausible path which could be taken by the group to solve the task.

The POIs could represent mineral deposits, water sources, or other useful resource deposits for the rovers or some exploratory mission. In this modified version of the rover domain, the POIs are not homogeneous and there are multiple types. Each POI also cannot be observed by one lone agent; multiple agents need to cooperate and be within close proximity of the POI. The agents are given a global list of different POIs that must be viewed. The agents must then learn how to coordinate to observe the list of POIs in the given sequence.

The state input of the agent includes distance sensors for each POI and distance sensors for other agents. The distance sensors are divided so that there are 4 sensors for each POI type or agent. There are sensors on the front-right, front-left, rear-right, and rear-left of the agent. The state input also includes the progress of the agents in their viewing of the list of POIs. The action space of the agent consists of being able to move forward or backward and being able to turn left or right continuous amounts. The overall action space and state space is continuous in this domain. The global reward for the agents was given as the percentage of the list observed in order at the end of the episode, with the other non-terminal rewards being zero.

3.3 Experiment Setup

For the experiment setup, the modified rover domain includes eight agents, with four pairs of different POIs, labeled zero through three. The overall goal of the agents is to observe POI types 0, 1, 2, then 3. The list of policies learned included four

policies, one for each POI. Each policy represents a controller which will move the agent towards and stay near the policy's respective POI. These policies learned from their own separate hand-crafted reward function. To learn these control policies, each agent was given the same policy and the reward consisted of the number of agents within the observation radius of a POI type as shown in equation 8.

$$R_i = \sum_{agent=1}^{N_{agents}} \begin{cases} 0, & (x_{agent} - x_{poi,i})^2 + (x_{agent} - x_{poi,i})^2 > r_{observation} \\ 1, & otherwise \end{cases} \tag{8}$$

Each policy was represented as a neural network with a hidden layer of size 16 using the activation function tanh. Each policy was trained via neuroevolution. Mutation could randomly occur at each weight with a Gaussian distribution centered at the weight value with a standard deviation of 0.2. The probability of mutation occurring at any given weight was 0.1. Tournament selection is used to select the parents from randomly chosen pairs of policies. Lastly, the population size was set to 50 policies.

Once the list of policies is learned, a Q-network and replay buffer is initialized for each agent with a size of 100,000 samples. Each agent has one hidden layer of size 15 with the learning rate set to 0.001. The loss function used was the mean squared error. The ADAM optimizer was used for the gradient descent updates of the neural network. Rewards are discounted with a γ of 1.0, causing the Q-network to predict the expected global reward at the end of the episode. Each Q-network would select the policy to be used for the next 10 time steps. The probability of randomly selecting a policy was initialized at 0.2 and decayed to 0.05.

4 Results

4.1 Performance

To measure the performance of the overall learned policies, the final global reward at the end of each episode was recorded. The experiment was repeated multiple times for a total of 15 runs. The averaged reward curve is included in Figure 3. This reward curve is compared to a reward curve received from agents which learned via a cooperative co-evolutionary algorithm (CCEA). Also, due to the significant noise present in the curve, a moving average filter was applied to remove a portion of the noise.

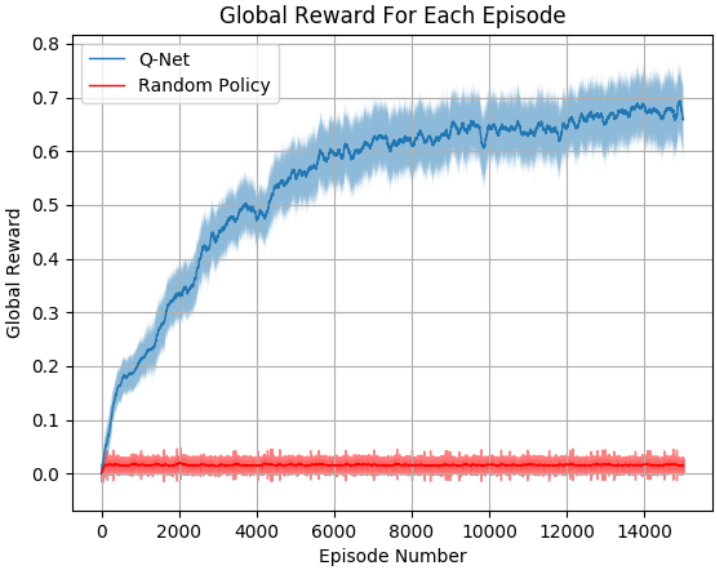


Figure 3: A plot of the global reward received over each episode, averaged across 15 runs. The average runs included exploration data point which lowered the average. The blue line represents Q-network policy selection, while the red line represents learning using CCEA.

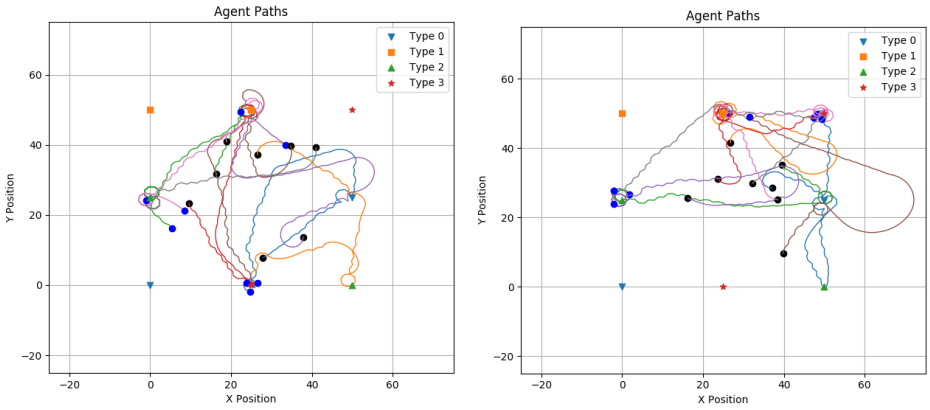
From Figure 3, it can be seen that the population of agents learned fairly steadily for the first 5,000 episodes with the minor improvements in learning over the next 10,000 episodes. The baseline which this algorithm was compared to was a population of agents trained via CCEA. CCEA performed very poorly as the reward was too sparse to learn from. It is clear from the figure that the Q-network performed much better than.

Generally, the proposed algorithm seems to learn well in the given domain. Despite the fact that on average, the overall learned policy received an average reward of 0.7 instead of the maximum of 1.0, optimal policies were often learned. A few of these policies are presented in the following sections.

4.2 Agent Behavior Analysis

The overall task is given to the group of eight agents is that they must view a series of four POI in order, with each POI needing to be observed by a group of three or more agents. One easy solution to learn would be to have multiple agents group up and then move from one POI to the next together. To increase the difficulty of the task, the amount of time given was limited so that each agent only had enough time to observe two or three different POIs. With this restriction, the agents would have to learn to divide up the task amongst themselves, with each agent learning to move to a different POI with a group.

Figure 4 presents the different paths each agent learned to take to achieve a global reward of 1.0. POIs are represented as a series of colored points which are placed in a rectangular pattern around the agents' starting points, represented by black points. The paths each agent took are represented by a different colored line for each agent, ending in a blue point representing their final positions at the end of the episode.



(a) Paths taken: Trial 14 (b) Paths taken: Trial 1

Figure 4: Two separate optimal learned paths are presented. Colored points represent the various POI. Black points represent starting locations of agents, colored lines represent paths, and blue points represent final positions. From the plots, it is clear that the agents seem to form two separate groups to solve the task. Note that different agent starting points were selected between experiments.

Figure 4 shows that the typical learned policy involved dividing the population into two different groups. One group would view POI type 0 while the other would view POI type 1. After these two types of POI were viewed, one group would view POI type 2 while the other viewed POI type 3. The agents effectively divided the problem into two smaller problems which each group then learned to complete. This fairly complex policy involved exploration at higher level, which was suitable for the Q-network.

4.3 Understanding Agent Decisions

In many real applications, understanding how an agent forms a decision is just as important as the decision that was made [14]. The overall learned policy generated by the Q-selection can be more easily explained than many black box learners. The agent chooses to use one of its policies, based on the reward it expected to receive from using the policy at a given moment in time. In this process, there is a clear division between the low-level policies which act as the direct controllers, and the high-level policy which decides which policy is best to use at a given moment in time. The explainability of this algorithm is explored in the following case study.

4.3.1 Case Study: Decision Boundaries

Figure 4 shows that an optimal solution consists of splitting into two groups and viewing POIs 0 and 1 simultaneously, then viewing 2 and 3 simultaneously. The learned policy seems simple but is more complex than expected. Because there are two different POIs of each type, if four agents decide to move to a POI of type 0, then two agents may observe one of the 0 type POIs while the other two agents observe the other POI of the same type. If this occurs, then POI of type 0 is not observed because there are not three agents at one of the two POIs of that type.

If an agent decides to view a POI, but there are no other agents to help, then staying near the POI is pointless, as the agent is unable to view it. As a result, the agent's decision to view a POI of a specific type is dependent upon, which POI is next in the list to be viewed, the distance to that POI, and the distance to the nearest agent. Figure 5 shows the decision boundary of the Q-network given that the next POI to be viewed is POI of type 0. This plot shows that the agent chooses to move to POI of type zero if there is an agent near it. If there is no agent near the POI, then the agent moves to another POI to potentially support another agent.

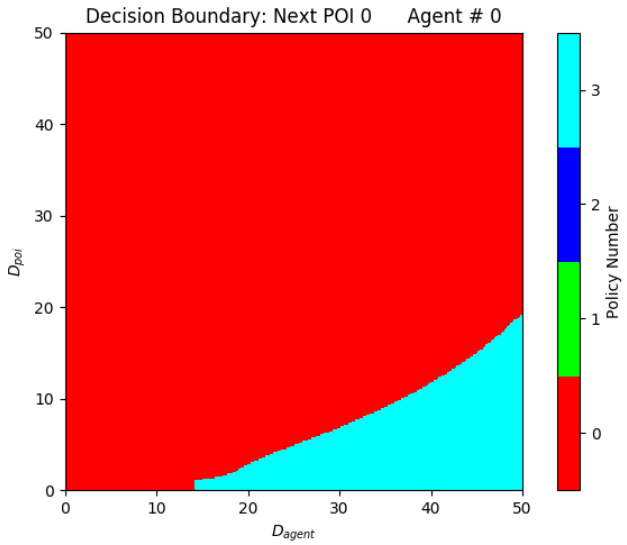


Figure 5: This plot shows the decision boundary of agent 0, needing to view POI of type 0. Each color represents a policy chosen: red-0, green-1, blue-2, cyan-3. Each policy number also corresponds to a POI type number. Ex. policy 0 allows an agent to move to POI type 0. The x-axis and y-axis represent the effect on the decision caused by, respectively, the distance sensed to the nearest agent and the distance sense to the nearest POI of the needed type

Figure 6 represents the scaling of these decision boundaries for each agent for each POI to be viewed. Each column represents the decision boundaries for each agent, while each row represents the POI to be observed from 0 to 3. In the first two rows,

it is clear that there are two groups of agents which seek to observe POI types 0 and 1. In the last two rows, the agents generally seek to view POI types 2 and 3.

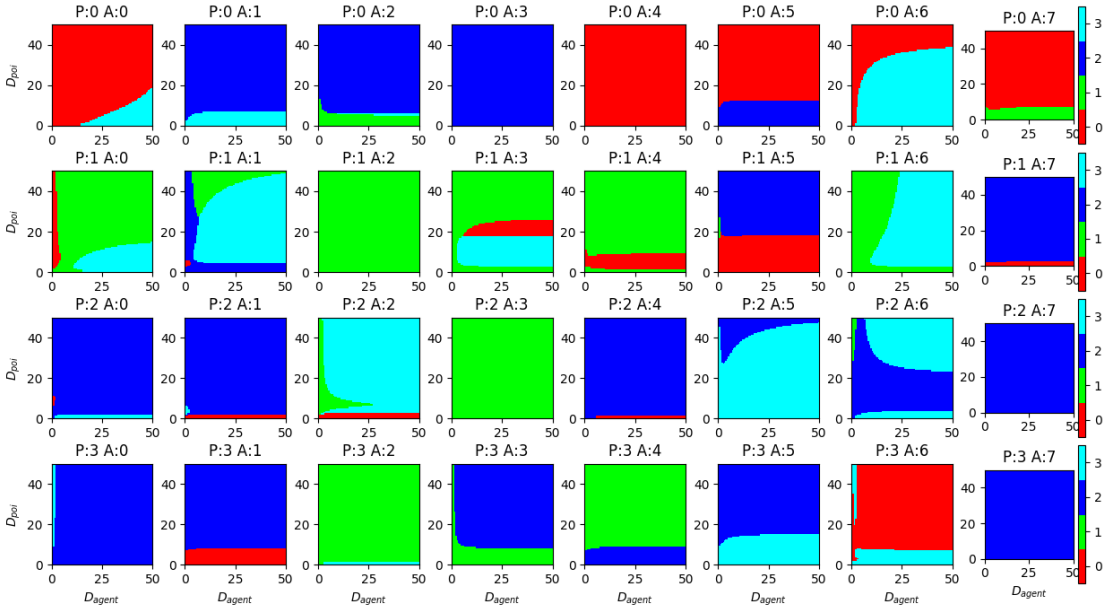


Figure 6: This plot represents the decision boundaries for a variety of scenarios. The columns of this plot represent the decision boundaries for each agent. The rows represent the next POI to be viewed by the group. The first row shows the decisions when POI type 0 needs to be viewed, the second row shows the decisions for POI type 1, etc.

4.4 Future Work and Improvement

There are two main difficulties associated with this learning algorithm. The first is that the list of sub-policies must consist of robust policies. The policies should perform their expected task in a variety of scenarios. If the policies are not robust enough, then a solution cannot be learned. An example of this was that originally policies were trained starting in the center of the map, and then having to move to a POI. The learned policies could only then move to a POI from the starting location. If the Q-network decided to move to POI of type 1 after POI of type 0 was reached, it couldn't because the agents were no longer in the center of the map. Without robust policies, the overall policy quickly reached a sub-optimal solution.

The second main issue with this algorithm is the difference in the time scale associated with the policies and the Q-network. For search to occur, the Q-network must randomly select a policy to determine its effect on the reward received at that moment in time. For the randomly selected policy to have a noticeable impact on the agent's performance, it must be used for multiple time steps. An example of this is an

agent randomly selecting a policy to view POI of type 3. For the agent to view POI of type 3, the corresponding policy must be used for a series of time steps before the agent arrives at POI of type 3. There was no clear method of determining how many time steps this would take, so a constant number of time steps were given to each policy. Future work could include learning the number of time steps it would take to complete the policy's goal, to improve the quality and efficiency of the Q-network policy selection.

One method of improving learning in this algorithm would be through the use of a prioritized experience replay buffer instead of the vanilla replay buffer. In almost all fifteen of the experimental trials, the agent discovered an optimal set of actions, yet it did not always learn an optimal policy from these experiences. This was most likely due to the infrequent occurrences of these optimal actions. Because these optimal actions were rare, they were rarely sampled from when learning from the replay buffer. A prioritized replay buffer should prioritize these rare optimal actions so that the Q-network is more likely to learn from them. Empirical evidence has found the prioritized replay buffer to outperform the vanilla replay buffer in many domains [15].

5 Conclusion

Sparsely rewarded multiagent domains which require complex coordination prove to be challenging but important domains for learning. Sparse rewards can be used to describe agent performance in more domains than dense rewards can. Sparse rewards are typically more difficult to learn from due to the reduction in direct feedback given to an agent [2].

This paper presents a method to learn in these sparsely rewarded environments. The algorithm accomplishes this by first learning a list of policies. These policies are trained via a series of hand-crafted reward functions, designed to divide the task into a series of skills. Once the policies are trained, each agent is given a Q-network to decide when to use each policy.

The modified rover domain was used to evaluate the effectiveness of the learning algorithm. Eight agents were given the task of viewing a series of POIs in a particular order, with each POI needing multiple agents to be successfully viewed. The algorithm presented learn well in this domain, with many learned policies which optimally completed the task.

Lastly, the learned policies were fairly easy for a human to understand in comparison to other black box methods. The policy list represents a list of low-level controllers which each perform some skill necessary to solve the problem. Then, at a higher level, a Q-network is used to select a skill with the highest expected reward. This allows for validation of the learned policies outside of the testing domain. As an example, in the rover domain, the decision boundary plots could be used to understand how the Q-network selected policies.

6 Bibliography

References

[1] R. Sutton and A. Bartow, *Reinforcement Learning: An Introduction*, 2016 p.1-25

[2] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothrl, T. Lampe and M. Riedmiller. Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards, 2017; arXiv:1707.08817.

[3] R. Sutton and A. Bartow, *Reinforcement Learning: An Introduction*, 2016 p.74-78

[4] R. Sutton and A. Bartow, *Reinforcement Learning: An Introduction*, 2016 p.158-160

[5] S. Russel and P. Norvig, *Artificial Intelligence A modern Approach*, 2010 p.727-737

[6] Y. LeCun, Y. Bengio, G. Hinton "Deep Learning," *Nature*, vol. 521, May, pp. 436-444, 2015

[7] G. Cybenko "Approximation by Superpositions of a Sigmoidal Function," *Math. Control Signals Systems*, vol. 2, May, pp. 303-314, 1989

[8] S. Russel and P. Norvig, *Artificial Intelligence A modern Approach*, 2010 p.125-130

[9] F. Such, V. Madhavan, E. Conti, J. Lehman, K. Stanley and J. Clune. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*, 2017; arXiv:1712.06567.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller. *Playing Atari with Deep Reinforcement Learning*, 2013; arXiv:1312.5602.

[11] M. Wooldridge, *An Introduction to Multi Agent Systems*, 2016 p.190-210

[12] C. Yates, E. Sachdeva, and K. Tumer *Multi-Reward Learning in Multiagent Systems*, 2019;

[13] A. Agogino and K. Tumer "Unifying temporal and structural credit assignment problems," *Proc. of the 3rd Intl. Jt. Conf. on Autonomous Agents and Multiagent Systems*, vol. 2, May, pp. 980-987, 2004

[14] A. Adada and M. Berrada "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)" *IEEE Access* 2018

[15] T. Schaul, J. Quan, I. Antonoglou and D. Silver. *Prioritized Experience Replay*, 2015; arXiv:1511.05952.

[16] L. Panait and S. Luke *Cooperative Multi-Agent Learning: The State of the Art*;

7 Appendix

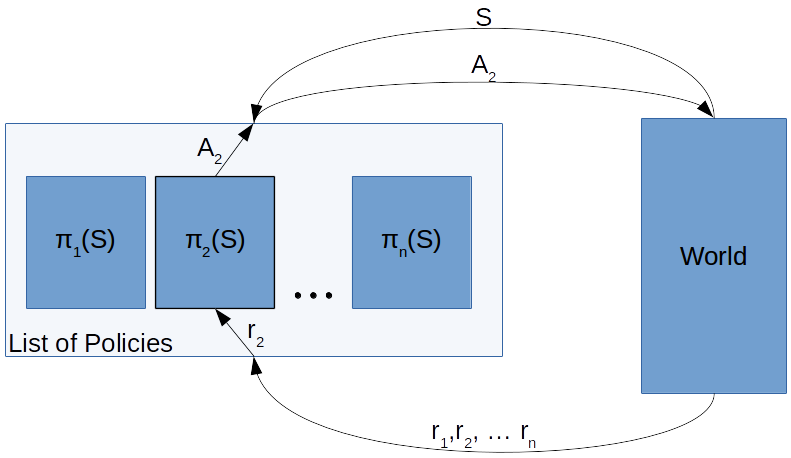


Figure 7: This figure represents the list of policies learned via neuroevolution and how they interact with the world.

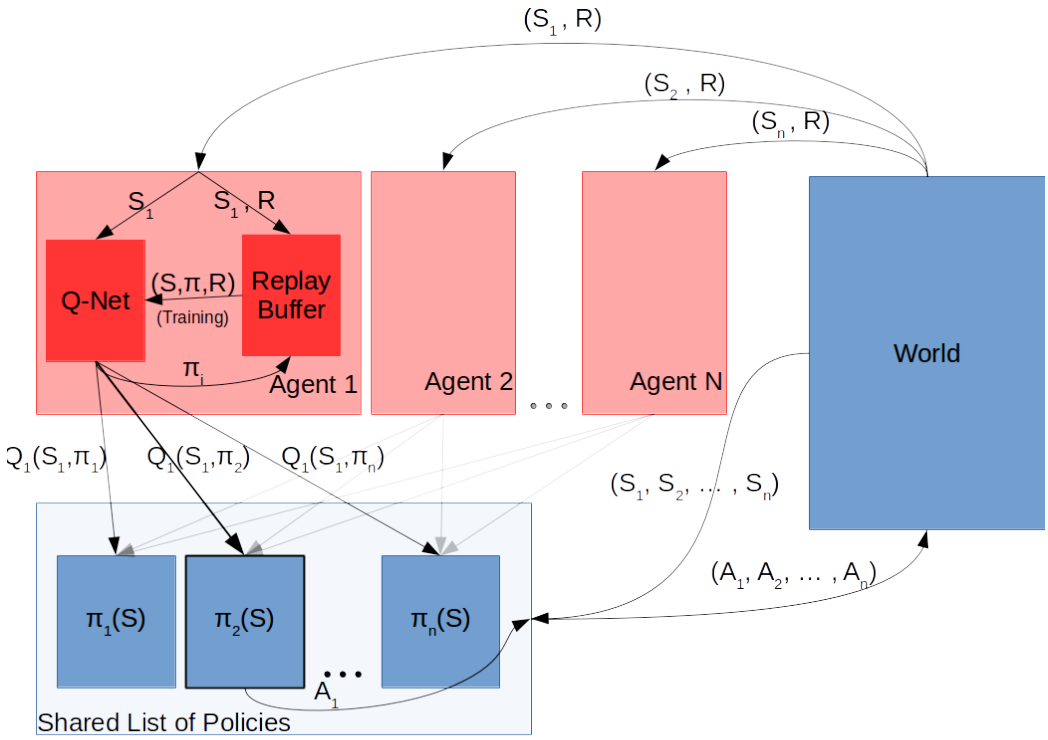


Figure 8: This figure shows the general learned policy architecture. Each light red rectangle represents an agent. Each agent contains a replay buffer and a Q-network. The Q-network is used to select the best policy, shown as a blue rectangle. The policy is used to generate an action for the agent to interact with the world. This process is repeated for each agent.