Dynamic Multi-Agent Reinforcement Learning for Control Optimization

Derek Fagan  
e-mail: fagandr@cs.tcd.ie

René Meier  
e-mail: rene.meier@hslu.ch

Distributed Systems Group,  
School of Computer Science and Statistics,  
Trinity College Dublin,  
Ireland

Mobile Systems Group,  
Department of Computer Science,  
Lucerne University of Applied Sciences,  
Horw, Switzerland

Abstract— In this paper we analyze the use of Reinforcement Learning (RL) in control optimization within dynamic multi-agent systems. RL is an effective algorithm for single agent optimization but performs less well in dynamic multi-agent environments. We investigate this principle based upon three of the most common RL algorithms. We also introduce a novel RL algorithm that excels in both single agent optimization and adaptation within multi-agent environments. This algorithm takes into account not only its own current state but also the current states of each of its significant neighbor agents so as to significantly increase performance within multi-agent systems. It employs a model driven approach to facilitate effective adaptation as well as policy-based methods to enable efficient action selection.

Keywords-Multi-Agent Reinforcement Learning; Control Optimization, Multi-Agent Systems

I. INTRODUCTION

Multi-Agent Systems (MAS) are dynamic by their very nature. This is not only due to operating within a potentially changing environment, but also to the fact that agents within the system adapt their behavior based both on their own experiences within the system and on the already adapting behaviors of other agents within the system.

A common approach to agent-based systems is to define them as Markov Decision Processes (MDP). MDPs however typically fail to reflect the fact that an agent is not only affected by its own actions but also by the actions of other agents within the system. Thus despite their suitability for single agent learning MDP algorithms are typically ill-equipped for deployment within MAS. In order to choose the best action to perform an agent must therefore not only be aware of its own current state but also the current states of other relevant agents within the system. The agent must also apprehend the effects of its own actions as well as the actions of relevant neighbor agents upon itself and possibly even upon each of its neighbors. A significant challenge with this is that agents all choose their actions simultaneously and usually do not know beforehand which actions the other agents are going to perform. It is thus very difficult for an autonomous agent within the system to foresee the effects of the combination of performed actions upon itself or upon others.

Adaptation within MAS enables agents to make adjustments to their representation of the environment or the affects of their actions therein whenever there is a change in the environment or in the behavior of other agents. Whereas model based methods such as Dynamic Programming (DP) adapt quickly and accurately they typically have excessively high calculation costs [1]. Model free approaches to learning on the other hand have much lower calculation costs but are slower to adapt to changes and are less accurate.

Traditional Reinforcement Learning (RL) approaches [2] such as Q-Learning typically assume finite action spaces in order to achieve lower calculation costs. These action spaces are often manually defined through a tedious process of trial and error. Real world problems on the other hand usually require continuous action spaces in order to select the true optimum action that enables accurate optimization. Explicitly representing the policy function using methods such as Actor-Critic Reinforcement Learning (ACRL) overcomes this challenge so as to enable low calculation costs for action selection within continuous action spaces. As with other model free approaches traditional ACRL still suffers from relatively slow adaptation to changes.

In this paper we present a novel RL algorithm that combines the rapid adaptation and accuracy of model based approaches with the low computation costs of policy based methods. This novel algorithm builds up a model of its environment through observation and interaction. This model is used to calculate the long-term benefit of taking an action from a given state. The explicit representation of the agent’s policy significantly decreases the processing requirements of these calculations within continuous action spaces. Interestingly, this algorithm’s value function, policy, and model are all efficiently maintained within the same lookup table, despite being separate conceptual entities.

In the following section we discuss the current state of the art in MAS. Within this context we then present our novel approach to MAS learning. We then describe our comparative evaluation experiments on three of the most
common RL approaches as well as our suggested approach. We discuss the results obtained and their implications.

II. BACKGROUND AND RELATED RESEARCH

In this section we discuss the current state of the art in Multi-Agent Systems (MAS). We begin by talking about Markov Decision Processes (MDPs). We then discuss Dynamic Programming (DP) and Reinforcement Learning (RL) algorithms.

A. Markov Decision Processes

In this section we outline the Markov Decision Process (MDP) framework and a representative selection of the single agent learning algorithms that are defined within it.

An MDP is defined as a tuple (S,A,T,R). S represents the total set of all states that the agent can be in within the system. This is referred to as the state space. A represents the total set of all actions that the agent may perform and is referred to as the action space. T represents the agent’s state transition function. T: s x a x s’ is the agents transition from state s to state s’ after having taken action a. R represents the agent’s reward function with R: s x a x r being the reward that the agent receives when it performs action a when in state s. The agent’s policy π is a mapping of the actions that an agent will take when in any given system state. To solve an MDP an agent must discover it’s optimal policy π* i.e. a mapping of the best possible action for the agent to take from any given system state. A key element for most algorithms in finding this optimal policy is the utility or value function. The state value function V represents the agent’s long-term utility from any given state. This utility does not simply refer to the immediate reward that can be expected after performing an action from the given state but to the sum of future rewards that can be expected either when following the agent’s policy (on-policy) or when choosing the action that leads to the highest immediate reward (off-policy). These future rewards are usually discounted exponentially over time to mathematically represent the agent’s increasing uncertainty of increasingly distant future actions. An action value function Q can also be used, which represents the agent’s utility in performing a given action from a system state and then following the agent’s policy thereafter.

There are two main categories of methods for solving an MDP. The first of which is Dynamic Programming (DP) and the second is Reinforcement Learning (RL). We will now discuss each of these in turn.

1) Dynamic Programming

DP algorithms use a predefined model of the environment as well as Bellman utility equations in order to obtain the optimal policy π*. The two major DP approaches are value iteration and policy iteration.

a) Value Iteration

Value iteration uses a Bellman equation to calculate the utility of each state and then selects the action with the highest utility. As the Bellman equations are non-linear they cannot be solved simultaneously but must be solved iteratively. Each state is initialized with an arbitrary utility value. These values are iteratively updated from the utility of their subsequent states as given by the model. This iterative process is repeated until equilibrium is reached. The Bellman update used is given below.

\[
V(s) \leftarrow R(s) + \gamma \sum_{a \in A(s)} \sum_{s' \in S} T(s'|s,a)V(s')
\]

Where:

- \(V\) is a discount rate.

\(A(s)\) is a discount rate.

b) Policy Iteration

Policy iteration begins with an arbitrary policy and then iterates over the following two steps: policy evaluation and policy improvement. The policy evaluation step calculates the utility for each state given the current policy π. This is given in the equation below.

\[
V(s) \leftarrow R(s) + \gamma \sum_{a \in A(s)} \sum_{s' \in S} T(s'|s,a)V(s')
\]

This calculation is more straightforward than the equation used in value iteration due to the lack of the max element. The policy improvement step uses one-step look-ahead to calculate a new and improved policy. The iteration between these two steps continues until no changes are made to the utilities during a policy improvement step.

There are issues with DP algorithms however that render them unsuitable for many real world applications. The first issue that is often pointed out is that it a predefined model of the environment is required. Obtaining an accurate model of the environment beforehand is not a trivial task and is often not even possible. The second issue with DP algorithms is their exponentially high computational requirements for systems with large state or action spaces. DP algorithms cannot be used for MDPs with continuous state or action spaces.

2) Reinforcement Learning

Unlike DP algorithms, RL algorithms do not require a predefined model of the environment in order to obtain the optimal policy π*.

a) Adaptive Dynamic Programming vs. Temporal Difference Learning

One approach to RL is to learn the model of the environment and then applying a DP algorithm to solve the MDP. This method is referred to as Adaptive Dynamic Programming (ADP). When ADP learns of a change in the model it uses the DP algorithm to iteratively modify the state utilities. Another approach to RL is that of Temporal Difference (TD) learning. TD algorithms adjust state utility values based on the difference between the expected utilities and those that are observed. Thus the utilities are adjusted towards equilibrium. Unlike ADP’s iterative update, TD makes only a single adjustment to its utility function per observation. TD algorithms have other advantages over ADP in that they are simpler, require less computation, and do not
require a model of the environment. TD methods do however learn slower than ADP methods.

b) Exploration

RL algorithms can be either passive or active. Passive RL methods are similar to TD policy evaluation in that the policy is fixed and thus the aim is to simply learn the utility function $V$. Active RL however allows the agent to modify the policy so that different actions can be tried out from different states. This process of trying out different actions from different states is called exploration. Agents that do not perform any exploration, but who always choose to perform the action with the highest utility value are called greedy agents. These agents often never find the optimal policy and even end up with quite poorly performing policies. Other approaches try to balance exploration and exploitation of an agent's knowledge. One simple solution is that of the e-Greedy algorithm. This algorithm performs a random action a percentage of the time in accordance with the variable $e$. $1 - e$ percent of the time the action that has the highest utility value is chosen. The value of $e$ can be reduced over time as the state and action spaces are more completely explored.

c) RL Algorithms

Three of the currently most popular RL methods are TD methods. These methods are Q-Learning [2], SARSA [3], and Actor Critic Reinforcement Learning (ACRL) [4]. We will now discuss each of them in turn.

Q-Learning

Q-Learning is a very popular active TD algorithm. Q-Learning learns an action value function $Q$ instead of a state value function and thus does not require a model of the environment. The equation that Q-Learning uses to update its state action utility values is given below.

$$Q(s,a) \leftarrow Q(s,a) + \alpha [R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

(3)

Where:

$\alpha$ is the learning rate.

As can be seen in this algorithm Q-Learning uses the action with the maximum possible state action utility value in calculating it’s TD, regardless of the agent’s current policy. This makes Q-Learning an off-policy algorithm.

SARSA

SARSA is almost identical to Q-Learning with the exception that it works as an on-policy algorithm i.e. it bases its TD calculation on the action that is selected by the current policy, as shown below.

$$Q(s,a) \leftarrow Q(s,a) + \alpha [R(s) + \gamma Q(s',a') - Q(s,a)]$$

(4)

This approach is technically more accurate as it takes into account the current policy but is less flexible as exploration may throw off the results.

Actor-Critic Reinforcement Learning

ACRL is an on-policy approach that explicitly represents the agent’s policy. This has the advantage of rapid action selection even with a continuous action space. The TD error calculated by the value function is used to update both the value function itself and the policy. The ACRL architecture is illustrated below.

![ACRL Architecture](image-url)
ADP ACRL’s model is updated using the following two algorithms:

\[ T_i(s, a) = T_i(s, a) + \alpha_{mi}(T_i(s, a) - T_i(s, a)) \]  
\[ R(s, a) = R(s, a) + \alpha_{mi}(R(s, a) - R(s, a)) \]  

Where:

- \( T_i(s, a) \) is the expected transfer in state variable \( i \) when action \( a \) is taken from state \( s \).
- \( t_i(s, a) \) is the observed transfer in state variable \( i \) when action \( a \) is taken from state \( s \).
- \( \alpha_{mi} \) is the model learning rate.
- \( R(s, a) \) is the expected reward when action \( a \) is taken from state \( s \).
- \( r(s, a) \) is the observed reward when action \( a \) is taken from state \( s \).

The state action utility value is dynamically calculated using this model and current policy \( \pi \) as given in the equation below.

\[ Q(s, \pi(s)) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) Q(s', \pi(s')) \]  

As with traditional ACRL, our approach is an on-policy algorithm that explicitly represents the agent’s policy. The policy is updated using the following gradient ascent equation.

\[ \pi_{t+1}(s) = \pi_{t}(s) + \lambda \nabla_{\pi} Q(s, \pi(s)) \]  

Where:

- \( \lambda \) is a weight variable that forces the algorithm to take small steps.
- \( f(\pi(s)) \) is the distance between \( \pi(s) \) and its nearest neighbor action in the positive direction of the gradient \( \nabla \).

As the slope may at times lead to a local maximum instead of a global maximum the agent actively explores unknown areas of the state and action space by generating random policies in accordance with e-Greedy action selection. This exploration is commenced once a local or global maximum has been reached. In static environments it is a good idea to start with a high e value and decrease it over time. In dynamic environments this approach is not suitable as exploration should be kept at a more constant level due to dynamic environment changes. A random policy can be looked on as a mutation of the current policy. If the mutated policy performs better than the current policy then it is used as a starting point for which the one-step look-ahead equation begins updating.

ADP ACRL agents not only observe their own current states but also the current states of relevant agents within the system. This set of relevant agents is only a subset of the total set of agents within the system. It is assumed that the agent itself can distinguish which agents around it are actually relevant, whether this information is manually specified or is detected in some other way is dependent on the actual agent implementation. This principal of a subset of relevant agents is similar to Guettin’s Stochastic Game (SG) coordination graph framework [5]. In a distributed control environment such as an Urban Traffic Control (UTC) system this may consist of adjacent traffic controller agents while in a robot soccer match this may consist of the closest teammate and opponent agents. As this possibly changing set of relevant agents is only a subset of the full set of agents within the MAS this approach is much more scalable than many previous approaches.

ADP ACRL agents’ action selection is done independently and is not directly influenced by cooperative communication with any other agent. Despite this, the actions performed can significantly influence other agents within the system. For this reason it is important that an agent has some understanding of the effects that actions that are likely to be selected by significant neighbor agents are to have on it. This can be achieved by modeling each significant neighbor agent. This approach can increase learning rates in MAS where each agent has many significant neighbor agents. In more sparsely populated MAS it is sufficient for an agent to integrate its significant neighbor agents local state variables into its own state space. Only the neighbor agents’ local state variables are integrated into the state space, as opposed to recursively integrating all local and remote state variables. This would not only lead to excessive learning times, complexity, processing costs, and communication costs, but would also not lead to significant accuracy increases [6]. Incorporation of these simple remote state variables as opposed to overly detailed data not only increases efficiency but also assumes a level of probability with regards to actions to be selected by significant neighbor agents.

IV. Evaluation

To evaluate our approach we firstly compare our novel Reinforcement Learning (RL) method i.e. Adaptive Dynamic Programming Actor Critic Reinforcement Learning (ADP...
ACRL), to other popular RL methods, namely Q-Learning, SARSA, and Temporal Difference (TD) ACRL. Each of these algorithms use lookup tables to represent their value functions. ADP ACRL uses one single lookup table to also represent its policy and model. ε-Greedy action selection is used as the exploration strategy for each of these learning algorithms with a non-reducing ε value. This lookup table is implemented in a manner that only states that have been visited are actually created in memory. Because relatively few states are ever actually visited within even large state spaces this dramatically reduces memory requirements. Each state is accessed in a recursive manner to dramatically reduce processing requirements in searching the state space.

Our chosen application area for purposes of this evaluation is that of control optimization, specifically Urban Traffic Control (UTC). A different autonomous UTC software agent governs each physical traffic controller within a transport network. These agents attempt to optimize traffic flow within the transport network by adjusting traffic load, balance, and synchronization between controllers. Our evaluation experiments have been conducted using a traffic simulator of our own device that imitates a traffic network. Traffic is reduced on each traffic controller when cycle lengths, split times, and offset times are appropriate for the current level of traffic. Otherwise traffic is increased. A more complete evaluation will be conducted using the VISSIM traffic micro-simulator, but for the time being this test application provides a suitable development and evaluation environment. The experiments were conducted on a set of four connected agents in a square topology. Traffic flows in one direction from South to North and from East to West. Each agent has two neighbors but only neighboring agents to the east or south of any one agent are significant upstream neighbors due to the direction of the flow of traffic. All algorithms run with the same settings, such as exploration rate, learning rate (0.2), and discount factor (0.8). The results of all experiments shown are averages of ten random simulation runs.

A. Single Agent Experiments

This first set of experiments was conducted in a single agent situation. Here the agent has neighboring agents but none of them are able to perform any actions. These are used as base line experiments to show how well each of the algorithms performs within a fairly static environment. Figure 3 shows how accurate each of the learning algorithms perform in a set of experiments with regards to a manually set optimal offset value. In this case the optimal offset value was 5 seconds between the agent and its upstream neighbor. We can see that although SARSA and Q-Learning adapt rapidly they can only adjust their accuracy to within 5 seconds of the offset. In order to increase their accuracy their action spaces would need to be manually redefined and tweaked until the correct action values for this particular situation were found. TD ACRL on the other hand did not require manual setting of its action space and due to its policy ascent algorithm it was able to find a more accurate solution. We can also see that TD ACRL has a less stable solution as it varies so much. ADP ACRL is the most accurate algorithm with an accuracy error of less than half a second. It also achieves a stable convergence.

These results can be confirmed by looking at the agent’s reward as shown in Figure 4. We can see that ADP ACRL achieves a high, stable result. TD ACRL’s result is quite high, but again is less stable.

B. Multi-Agent Experiments

The next set of experiments places the agent in a multi-agent situation. Here each agent is able to adjust its offset. All agents are thus attempting to learn at the same time and thus create a dynamic environment.

![Figure 3. Single agent distance to optimal value](image3.png)

![Figure 4. Single agent reward](image4.png)

![Figure 5. Multi agent distance to optimal value](image5.png)
Figure 5 shows that both SARSA and Q-Learning become less accurate in this scenario, achieving a fairly steady convergence of 10 seconds from the optimal value. TD ACRL’s performance decreases significantly, as does the stability of its solution. From our observations, this is caused by the speed at which it learns. By the time one agent adjusts its policy to the behavior of its neighbor, its neighbor has already changed its behavior. When ADP ACRL does not take other agent state variables into account its performance drops below that of SARSA and Q-Learning. When neighbor agents states are taken into account ADP ACRL’s performance steadily improve over time until it gets to within 2-3 seconds to the optimal value. In other experiments we have found that TD ACRL’s performance is actually made worse when it takes into account neighboring agent state variables. We theorize that this is due to the increased learning rate caused by the increase in state space size.

As can be seen from the results of the experiments discussed within this section ADP ACRL performs particularly well in multi-agent scenarios compared to other RL algorithms. This increased performance with minimum computational cost consequences is achieved through the combination of ADP ACRL’s model based technique and its policy based approach to learning.

V. CONCLUSION

In this paper we introduce an adaptive Multi-Agent Reinforcement Learning (MARL) algorithm. This method uses novel Adaptive Dynamic Programming Actor Critic Reinforcement Learning (ADP ACRL) to combine the benefits of model based and policy based learning algorithms. This adaptive MARL algorithm not only takes its own current state into account but also neighbor agents’ current states into account to overcome dynamics within Multi-Agent Systems (MAS). These three components combine to handle the dynamics associated with multi-agent learning. Our experimental results show the significant increase in adaptation when using these concepts in combination. These results are contrasted to adaptation of other popular reinforcement learning algorithms.

ACKNOWLEDGMENT

This work was supported in part by the Irish Research Council (www.research.ie).

REFERENCES