2022

Deep Reinforcement Learning for Open Multiagent System

Tianxing Zhu
Oberlin College

Deep Reinforcement Learning for Open Multiagent System

TIANXING ZHU, Oberlin College, USA

In open multiagent systems, multiple agents work together or compete to reach the goal while members of the group change over time. For example, intelligent robots that are collaborating to put out wildfires may run out of suppressants and have to leave the place to recharge; the rest of the robots may need to change their behaviors accordingly to better control the fires. Thus, openness requires agents not only to predict the behaviors of others, but also the presence of other agents. We present a deep reinforcement learning method that adapts the proximal policy optimization algorithm to learn the optimal actions of an agent in open multiagent environments. We demonstrate how openness can be incorporated into state-of-the-art reinforcement learning algorithms. Simulations of wildfire suppression problems show that our approach enables the agents to learn the legal actions.

CCS Concepts: • **Computing methodologies** → **Multi-agent reinforcement learning**; *Planning and scheduling*; *Markov decision processes.*

Additional Key Words and Phrases: multiagent systems, open environment, deep reinforcement learning, neural networks

ACM Reference Format:

Tianxing Zhu. 2022. Deep Reinforcement Learning for Open Multiagent System. 1, 1 (April 2022), 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Multiagent systems consist of multiple intelligent agents (computational entities) that are interacting with the environment they are in [Shoham and Leyton-Brown, 2009]. In multiagent systems, reasoning about others' behaviors will make collaboration of artificial intelligence possible and reach the shared goal faster. However, many real-world environments involve openness where individual agents join or leave the environment over time, which makes decision-making especially challenging, because now agents not only need to understand other agents' actions to make good decisions, but also need to predict the presence of them before reasoning their behaviors. For example, in the case of self-driving cars, the total number of intelligent cars (agents) is fixed, but the number of them at a specific location changes over time. Even if individual cars can learn from others' behavior to help them analyze the traffic at the location, openness requires them to predict

Author's address: Tianxing Zhu, vzhu@oberlin.edu, Oberlin College, 135 W Lorain Street, Oberlin, Ohio, USA, 44074.

XXXX-XXXX/2022/4-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

whether others are present at the location in the first place. Trying to understand cars that are not present in the environment will result in inaccurate knowledge, which leads to non-optimal decisions.

In this paper, we focus on using reinforcement learning to learn how to make optimal decisions in open environments? The openness of the environment allows agents to be active or inactive in the environment over time; for real-world environments such as wildfire suppression, agents need to put out different scales of fires together while joining or leaving the area to refill the suppressants. We aim to make agents not only learn what actions to take to efficiently put out the fires, but also how to distribute their extinguishing power within a dynamic group. Previous work has focused on planning solution that uses a Monte Carlo tree search algorithm [Eck et al., 2019]. However, a planning solution requires the agent to have enough information of the system before they make the plan, and a lot of information might not be immediately available. A reinforcement learning solution uses the actual experience of the agent from its interaction with the environment to improve its understanding of the environment over time.

We propose a deep reinforcement learning approach to train the agents to reason in these situations. By adapting proximal policy optimization [Schulman et al., 2017], a state-of-the-art single-agent reinforcement learning algorithm, we bring its advantage in balancing exploration and exploitation, as well as the ease of hyperparameters tuning into multiagent systems. To promote learning of the dynamic caused by the openness of the environment, we incorporate the internal state of the agent, which gives a clue to the agent's presence in our state representation of the environment.

Our results show that in the simulation of the wildfire suppression problem [Chandrasekaran et al., 2016], our method enables agents to choose the legal actions that do not result in penalties on almost every time step. Further experiments demonstrate the advantages of our adaptations in both learning speed and performance compared to other regular deep reinforcement learning algorithms.

In the following sections, we will overview the background and related work, starting from the basics of reinforcement learning to the state-of-the-art proximal policy optimization algorithm for deep reinforcement learning. Then we will formulate our problem in the context of deep reinforcement learning and introduce our methods in detail. Finally, in the experiments section, we will talk about the strategies we used to tune the hyperparameters, followed our final results. We conclude by summarizing our work and propose promising directions for future work.

2 BACKGROUND

2.1 Reinforcement Learning and MDP

Reinforcement learning, as its name suggests, is learning by receiving reinforcement or *reward*. In the field of artificial intelligence, we define an *agent* as anything that perceives its *environment* through interaction with the environment. At each time step, the agent interacts with the environment by performing an *action* in the current *state* of the environment and receives feedback from the environment. The feedback received by the agent is the reward, which provides a direct evaluation on the quality of the agent's behavior. The state of the environment changes due to the action. At the next time step, the agent takes another action and receives another reward, and so forth. An

intelligent agent learns to maximize the reward by trial and error [Russell and Norvig, 2010]. For example, in the wildfire suppression environment, the firefighters are the agents. At every time step, the state of the environment could be represented by the set of fire intensities of each fire. Each agent chooses an action to perform, such as fighting one of the fire or refilling its suppressant. The environment then immediately gives a reward to the agent; a high reward might be given when a fire is put out, while a penalty (negative reward) might be given when a fire is burned out. The action affects the state of the environment, and thus the state changes.

We use a *Markov Decision Process (MDP)* to describe these sequential decision problems. Formally, an MDP consists of:

- S, the set of all possible states s of the environment.
- A, the set of all actions a.
- $T : S \times A \rightarrow P(S)$, the *stochastic* transition model, which gives the probability of arriving at a state s' after taking action a at a state s, or $p(s'|s, a)$.
- $R : S \times A \times S \rightarrow \mathbb{R}$, the reward function, which gives the reward obtained after taking action a in state s and arriving at state s', or $r(s, a, s')$.

A *policy* is the decision-making mechanism used by the agent to decide which action to take in a given state. A stochastic policy, formally, $\pi : S \rightarrow P(A)$ defines the probability of performing each action at a given state; while a deterministic policy, formally, $\mu : S \rightarrow A$ maps an action to a state, which means the agent will always take the same action at a given state. The policy acts as the brain of the agent. In reinforcement learning problems where agents have to learn how to act, It is crucial to have a stochastic policy so the agent can explore the environment by trying out different actions at a state rather than simply repeating the first action that seems promising.

An *optimal policy* is the policy that maximize the *utility* of the agent. Utility represents the combination of immediate reward (given by the reward function) and long-term reward. We use a utility function to balance them, formally,

$$U^{\pi}\left(s_0\right) = \sum_{t=0}^{H} \gamma^t R\left(s_t, \pi\left(s_t\right), s_{t+1}\right) \tag{1}$$

where H is the horizon representing the maximum time step, and $\gamma \in (0, 1]$ is the discount factor that controls how much we weight the future rewards [Russell and Norvig, 2010].

2.2 Bellman Equation and Q-Learning

To use our MDP model to improve the policy, or *planning* in the environment, we use the Bellman Equation:

$$Q\left(s_t, a\right) = \sum_{s_{t+1} \in S} T\left(s_t, a, s_{t+1}\right)\left[R\left(s_t, a, s_{t+1}\right) + \gamma * V\left(s_{t+1}\right)\right] \tag{2}$$

$$V(s) = \max_{a \in A} Q(s, a) \tag{3}$$

$$\pi(s) = \operatorname*{argmax}_{a \in A} Q(s, a) \tag{4}$$

The V-function $V(s)$, or the state-value function, measures the expected future utility (Eq. 1), which means it tells us how good or bad the current state s is following the current policy. The Q-function $Q(s, a)$, or the action-value function (Eq. 2), defines the value of taking action a in the state s under the current policy [Russell and Norvig, 2010]. The Bellman Equation shows that the value of taking action a at state s can be decomposed into two parts, the immediate reward plus the discounted future utility recursively.

When we have access to a complete MDP model (we know S, A, T, R), we can apply the Bellman Equation directly to exploit it and find a good policy. However, the transition function T and the reward function R is not always known in many problems, we need the agent to interact with the environment and gather information. Reinforcement learning methods use these information to infer the dynamics of the environment and estimate the model.

One of the popular reinforcement learning methods, *Q-learning algorithm*, approximates the Q-function which can be used by the agent to estimate the action-value at a state and construct the policy. Q-learning updates the approximation of the Q-function using the state, action, and reward experiences of an agent by interacting with the world continuously [Sutton and Barto, 2020].

2.3 Neural Network

At the moment, one of the best tools we have to handle the unconstructed reinforcement model and function approximation is the *neural networks*. We use neural networks to approximate state-value function or the action-value function [Sutton and Barto, 2020]. A neural network consists of one input layer, one or more hidden layers, and one output layer. Each hidden layer has *neurons* that act like functions taking inputs from the previous layer and calculating an output; they send outputs to neurons to the next layer. The output layer outputs a prediction y for a given input x. Neural networks are trained to minimize a *loss function* calculated with x and y by adjusting the parameters of the functions (or neurons).

For example, to approximate the state-value function, we can represent it as a parameterized functional from with some weight (or coefficient) w, then we can write $\hat{V}(s, w) \approx V_\pi(s)$ for the approximate value of state s given weight w. $\hat{V}$ can be a linear function with w being the vector of weights. More generally, $\hat{V}$ can be a function computed by a multi-layer neural network, or *deep neural network* since it has "deeper" layers, with w being the vector of connection weights in all layers [Sutton and Barto, 2020]. Neural networks try to construct the function that maps input state s to a output state-value by finding the right weights. By adjusting the weights, all kinds of functions can be implemented by neural networks.

2.4 Deep Reinforcement Learning and Deep Q Network (DQN)

Deep reinforcement learning combines deep learning with reinforcement learning by using deep neural networks used as function approximators. [Sutton and Barto, 2020] *Deep Q-Network (DQN)* trains a Q-function approximator by minimizing the loss function with the current state of the environment as input and the predicted Q-value of taking different actions at the state as output. Over the course of the training, the agent will use the network to choose action and update its policy by inputting feedback from the environment [Mnih et al., 2013].

DQN also solves the problem of the high correlation of data fed into the neural network: instead of using the collected transition experiences immediately to train the network, they are stored in a buffer called *experience replay*. New experiences replace the old ones once the buffer is full, and the network now samples random minibatches of data from the buffer to train the network. If the network is updated with only transitions that are localized in a short time window, Q-function approximated will overfit the recent situations. In experience replay, we sample transitions that occur anywhere across time to improve generalizability of the learned Q-values across all situations.

2.5 Advantage Actor-Critic

Another approach to solve the reinforcement learning problem is the *policy search*. This method uses a stochastic policy representations $\pi_\theta(s, a)$ which specifies the probability of selecting action a in state s. Planning with policy search is achieved by maximizing the objective function representing the sum of rewards generated by following the policy over some time steps, formally,

$$J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T} \gamma^t r\left(s_t, a_t, s_{t+1}\right)\right] \tag{5}$$

where τ is a sequence of state-action pairs generated by the policy [Russell and Norvig, 2010].

Using a neural network, we can approximate the objective function just like we approximate the Q-function in DQN. A commonly used objective estimator is:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_\theta\left(a_t \mid s_t\right) \hat{A}_t\right] \tag{6}$$

where $\hat{A}_t$ is an estimator of the advantage function at time t [Russell and Norvig, 2010].

The *Advantage* of a state is defined as:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{7}$$

where π is the current policy, $Q^\pi(s, a)$ is the approximated Q-value, and V^π is the approximated state-value, assuming policy π is being used forever; it is the difference between the estimated Q-value of the state by taking action a and the estimated state-value of that state. Intuitively, it indicates how many *extra* rewards than expected we are getting if we take action a based on the current policy. The higher the advantage is, the more likely the agent will take the action again when visiting the state.

The problem with approximating the objective function using a neural network is that now we need to approximate both Q and V functions, which are used to calculate the advantage for the objective estimator.

The solution is the *Advantage Actor-Critic* method, which uses an actor-critic architecture for the network that has two sets of layers: the actor outputs the policy π_θ (the probabilities for taking each action) of the input state s, while the critic outputs the predicted state-value $V(s)$ of the input state s. The state-value is then used to approximate the advantage $\hat{A}_t$ for the objective estimator using the *n-step advantage*:

$$A_t^n = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V^\pi\left(s_{t+n+1}\right) - V^\pi\left(s_t\right) \tag{8}$$

During the training, the actor is updated by maximizing the objective estimator, and the critic is updated by minimizing the error between the estimated state-value and its actual value. To solve the issue of highly correlated data, Advantage Actor-Critic method uses multiprocessing to simulate several environments in parallel, that is, to use multiple processes to experience their own individual environment and collect experiences for training the network. Each process explores the environment differently which diversify the data to improve generalizability [Mnih et al., 2016].

2.6 Proximal Policy Optimization (PPO)

Proximal Policy Optimization is one of the state-of-the-art policy search algorithms. It replaces the traditional objective estimator with the following *clipped surrogate objective*:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{9}$$

where $r_t(\theta)$ is the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, ϵ is the clipping parameter, and $\hat{A}_t$ is an estimator of the advantage function at time t [Schulman et al., 2017].

Compared to traditional objective estimators, PPO's clipped surrogate objective not only makes sure the policy update is not too dramatic by removing the incentives but also less sensitive to hyperparameter changes, which makes it flexible in dealing with a range of tasks and easy to tune.

2.7 Generalized Advantage Estimation (GAE)

Advantage estimator develops into different forms over the years. The one we are using is the *Generalized Advantage Estimation (GAE)*:

$$A_t^{\text{GAE}(\gamma,\lambda)} = (1-\lambda)\sum_{l=0}^{\infty} \lambda^l A_t^l = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{10}$$

It is based on the n-step advantage (Eq. 8) introduced in the A2C paper [Wu et al., 2017]. The n-step advantage attempted to mitigate the bias vs. variance trade-off when exploring the environment. It basically uses n next immediate rewards and approximates the rest with the state-value of the state visited n steps later. The GAE is simply the discounted sum of all n-step advantages [Schulman et al., 2015].

3 PROBLEM

We design our deep reinforcement learning method for an open multiagent system based on these previous works. We formulate our problem in the context of wildfire suppression [Chandrasekaran et al., 2016], where agents work together to put out fires of different sizes in different locations without any communication or prior coordination. This section will describe the environment we are dealing with in detail and why openness complicates the problem.

3.1 Open Multiagent Systems

As its name suggests, multiagent systems are the environments where multiple agents act in the environment and try to achieve their goals. Traditionally, multiagent system problems are solved with centralized or decentralized planning [Russell and Norvig, 2010]. Centralized planning occurs when a single planner chooses everyone's policies, which requires replacing actions with joint actions in the MDP model while having the cost of growing exponentially larger as more agents are introduced; decentralized planning, on the other hand, requires each agent to model other agents' reasoning, which needs more time and information that is not necessarily available to the planner.

In open systems, however, the problem is further complicated by the need to track other agents' presence and to understand only the present agents' actions. For example, in the wildfire suppression problem, each agent needs to know whether other agents are fighting fires or absent from the environment to recharge the suppressant, so the agent can make the right decision and not waste their suppressant on fires that cannot be put out by one agent.

3.2 Wildfire Domain

We use the wildfire domain to simulate the open multiagent environment, and we divide the agents into *frames*. For each frame of agents, we define the model $MDP_\theta = \langle S_\theta, A_\theta, T, R, \rangle$, where:

- θ is the frame that represents the agent's capabilities; in the wildfire domain, this is essentially (1) the locations of the agents that limit the fires they can fight and (2) the type of firefighter (e.g., ground firefighter vs. helicopter), which represents the ability of the agent to put out fires.
- S_θ is the set of possible states of an agent in frame θ. In our problem, this is defined as $S_\theta = S \times S_i$ where S is the set of states of the environment represented by the set of intensity levels of all fires in the environment; and S_i is the internal state of the agent. The internal state of an agent represents the level of its suppressant which can be consumed to fight fire.
- A_θ is the set of possible actions of an agent in frame θ. In the Wildfire domain, these are the individual fires that each agent can fight and the NOOP (no operation) action, which make them temporarily absent in the environment when they need to refill the suppressant.
- $T : S_\theta \times A \to P(S_\theta)$ is the stochastic state transitions function which gives the new state s of the environment and the new internal state s_i after taking action a.
- $R : S \times A \times S \to \mathbb{R}$ is the reward function that gives the reward of an agent after taking action a on state s.

In the wildfire environment, each fire has five levels of intensity from non-existent (0) to burned-out (4); fire intensity transitions stochastically based on the transition function. Generally, the intensity is likely to decrease when enough agents are fighting the fire, and increases or stays the same otherwise. Burned-out fires cannot be fought. Agents are equipped with a suppressant that takes on levels between empty (0) to full (2); each agent starts with a full suppressant that changes based on the transition function when the agent takes action. For each agent, the legal actions are: fighting an adjacent fire or taking a NOOP action; the NOOP action recharges the suppressant when it's empty. Agents receive a shared reward or penalty when a fire is put out or burned out. Taking illegal actions also result in a penaly. Detailed description of the wildfire domain can be found in

Individual Planning in Open and Typed Agent Systems [Chandrasekaran et al., 2016] and *Scalable Decision-Theoretic Planning in Open and Typed Multiagent Systems* [Eck et al., 2019].

4 SOLUTION

With all the related work discussed, as well as our problem being formally formulated, we can now describe our solution in detail. As mentioned in the previous section, we are assigning agents in the environment into frames. This is because each frame is essentially solving a different MDP, and thus has a different optimal policy, so our solution involves training one actor-critic neural network that provides a policy and an estimation of the state-value for each frame so that the agents of that frame can use the network to make their decision. Instead of using the traditional multiagent system learning model, we are simplifying it into individual single-agent MDP models in each frame. Each neural network for each frame is thus trained to learn the best action to take at some state when the agent is in that frame. For future work, we want to work on the more challenging problem that uses a multiagent reinforcement learning model.

4.1 Incorporating Openness

The openness of the problem is incorporated into our model in two ways. First, a NOOP action is available for agents in all frames; an agent with no suppressant taking NOOP means the agent leaves the environment temporarily to recharge the suppressant, while an agent with suppressant taking NOOP means it is doing nothing (not fighting fires or recharging).The NOOP brings openness to the learning process. Secondly, the internal state, which represents the suppressant level of the agent, is included in the state representation. Therefore, a non-zero internal state means the agent is in the environment, and a zero internal state means the agent is unlikely to be in the environment due to short of suppressants; this should allow the neural network to learn the effect of the absence of the agent.

4.2 Neural Network Architecture

A neural network is set up for each frame to learn the optimal policy for the agents in the frame. Each neural network model has an actor-critic architecture which is discussed in the early section. The input to the neural network is the current state of the environment as in that frame, an $(n_{fire} + 1) \times 1$ array where n_{fire} is a number of fires in the environment. The actor gives a distribution of possibilities over the possible actions, and the critic gives a state-value evaluating whether the input state is worth visiting. The two hidden linear layers for both the actor and the critic have a 64 output dimension, followed by a rectifier. The last linear layer for the actor has an $n_\theta + 1$ output dimension, where n_θ is the number of individual fires that an agent in frame θ can act on; and the output layer for the actor is a Softmax function that outputs a probability of distribution of the actions taken in the current state. This output is the stochastic policy for the agent. The last linear layer for the critic is fully-connected with a single output that predicts the state value $V(s)$ of the input state.

4.3 Adapted Algorithms and Procedure

We adapt proximal policy optimization in the actor-critic style. The advantage is calculated using the generalized advantage estimation. In reinforcement learning, an episode is defined as a sequence of interaction from the start time step to the final time step [Sutton and Barto, 2020]. The detailed procedure of our method is given below.

Algorithm 1 Adapted Algorithm

procedure Policy Optimization
- Initialize the actor-critic network for each frame θ
- for each process p in parallel:
  - Get a copy of the actor and critic for each frame θ
  - Reset the environment
  - *Sample*
- Wait for all processes to terminate
- for actor-critic network of each frame θ:
  - Get the batch of experience pairs from all processes
  - *Train*

procedure Sample
- for each frame θ:
  - Sample an episode of n steps by using the actor to choose actions for each agent
  - Collect the transition pairs $(state, action, reward, is\_done)$ on each time step
  - Calculate the generalized advantage estimation (Eq. 10) for each transition using the critic
  - Return the experience pairs $(state, action, reward, is\_done, advantage)$

procedure Train
- Randomize the experience pairs in the batch
- Make minibatches of experience pairs of size n
- for K epochs:
  - Get a minibatch of experience pairs
  - Reset gradient for the actor-critic network
  - Maximize the clipped surrogate objective (Eq. 9) for the actor
  - Minimize the clipped mean square error for the critic
  - Add entropy bonus
  - Update the actor and critic network

5 EXPERIMENTS

5.1 Setups

In the Wildfire domain, agents in different frames are asked to put out fires of different sizes together without any prior coordination. Putting out small, medium, large, and huge fires provides agents with shared rewards of 20, 50, 125, 300, respectively, while a fire burning out results in a shared penalty of 1, choosing illegal actions (such as fighting a fire that is already put out) gives individual agent penalty of 100.

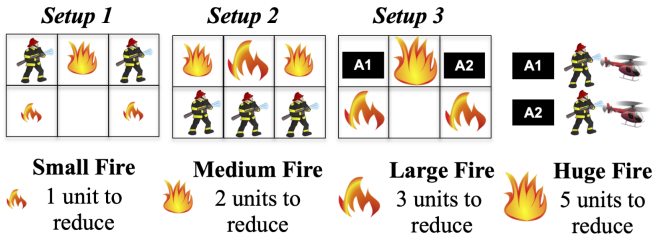


Fig. 1. Setups. Our setups involve a varying number, sizes, and positions of fires, as well as a varying number of agents and their types. Fires require different units of suppressant to reduce. Ground firefighters apply 1 unit, helicopters 2 units.

Three different setups are available: Setup 1, where two frames of agents each have a unique small fire that they can individually engage with, and a shared medium fire that requires the collaboration of the two frames of agents. Setup 2 is a more complicated situation where all fires require collaboration. Finally, Setup 3 has even more intensified fire while introducing different types of agents of different capabilities. Our experiments mostly focus on Setup 1 with one agent in each frame. We fix one agent's policy to choose the optimal action all the time, so it is useful to test the behavior of the other agent when tuning the hyperparameters.

5.2 Training Details and Hyperparameters

In these experiments, we set up one neural network for each frame of agents to learn the optimal policy of that frame. Each neural network is optimized using the Adam optimizer [Kingma and Ba, 2014] with a learning rate of $2.5e-4$. On each episode of the training, 8 processes simulate 15 steps for all gents in parallel in the Wildfire environment to collect transition experience. We choose 15 steps for an episode because if the number of steps is too small, agents might not be interacting with the environment enough to learn meaningful experience; on the other hand, the number of steps being too big will lead to biased learning, which favors situations where all fires are burned out, or all fires are put out.

Once all the processes finish collecting experience over 15 steps, we prepare the experience into $(state, action, reward, is\_done)$ pairs. For each episode, we will have $8 \times 15 = 120$ pairs; we then randomize the pairs and make minibatches of 15 pairs to avoid correlation between data that might bias the training. These minibatches are fed into the neural networks for training.

For calculating loss for proximal policy optimization, we choose the following hyperparameters within the range suggested by the original paper [Schulman et al., 2017]:

- $\gamma = 0.99$, which is the discount factor that controls the value of future rewards when calculating advantages.
- $\lambda = 0.95$, which is the smoothing parameter that reduces the variance in training which makes it more stable.
- $\epsilon = 0.2$, which is the clipping parameter that ensures the policy change does not exceed 0.2%. This is the recommended value in the paper.
- $c_1 = 0.5$, which is the value function coefficient. This is the recommended value in the paper.

- $c_2 = 0.01$, which is the entropy coefficient. The entropy coefficient is multiplied by the maximum possible entropy and added to the loss. This helps prevent early convergence of one action probability dominating the policy and preventing exploration. This is the recommended value in the paper.

Next, we discuss different strategies applied to improve the performance. We also compare different deep reinforcement learning methods, evaluate how state representations and reasoning models help analyze the training, and also how setting *is_done* flag (which forces the state-value to be 0) and randomizing experience before feeding into the neural network improve the performance.

We measure the performance by the sum of rewards received by all the agents in the environment over 15 steps on each training episode. To simplify the discussion, we will focus on Setup 1 in these experiments, with the settings discussed above. The plots below all shows the performance of the agent during the training.

5.2.1 Fixed Policy for One Frame. The first strategy we use to better understand the performance of our method is fixing the policy of one agent to always choose the optimal action. This is because all the agents in the wildfire domain share their reward: if one agent receives a reward on some time step by putting out a fire, the other agent will get the same reward on that time step as well. This not only hinders us from understanding the actual performance of an agent but also might hinder the agent from learning the best action because the reward could be deceiving. By fixing the policy of one agent to be optimal, we are minimizing the effect of potentially deceiving reward, essentially testing the performance of one neural network. *Thus, the figures below will only plot the performance of* $frame_0$ *agent in Setup 1.*

5.2.2 Categorical State Representations. Our state representations, as we discussed earlier, are the set of fire intensity levels concatenated with the suppressant level of the agent. For example, state (2, 2, 2, 1) means that all fires have level 2 intensity, and the suppressant level of the agent is 1. As we can see, these values are all numeric. We apply a technique called one-hot encoding by making these values *categorical* by making each value of the fire intensity or suppressant level unique. For example, the first 2 in the original state, which stands for the intensity of $fire_0 = 2$ becomes (0, 0, 1, 0, 0), where the binary number at each position is 1 if and only if the fire intensity equal to that positional number. Then the entire state above becomes (0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1). In this way, we aim to increase the accuracy of the network since it becomes sensitive to all values, rather than assuming higher numbers are more important. In the case of wildfire suppression, it should help the agent learn that NOOP is also useful when the fire intensity is 0 since fighting a zero-intensity fire will be illegal.

5.2.3 Training with Minibatches. After each process collects the transition pairs (or experiences) over 15 time steps in an episode, instead of directly feeding the batch into the neural network for training, we wait until all processes end and make new minibatches of randomized transition pairs for the neural network input. This is inspired by the DQN. The reason behind this is that experiences from the same batch are highly correlated, which might bias the training. For example, in an episode of our simulation, one process might be exploring how to put out the small fire, while another process might be exploring what to do after the small fire is put out. If we feed those experiences sequentially, the network will first update the policy trying to fight the fire in all states, then go the other way since a penalty will be given when trying to fight a non-existent fire. By

training with sampled experiences from processes altogether instead of in sequence, we try to reduce the instability of the network.

In Fig. 2, the x-axis represent training episode, the y-axis represent the sum of rewards on that episode (*same for all figures below*). Therefore, each line represents the performance of the agent over time, and an increase in the performance means the agent is learning to act better in the environment. We can see that training with minibatches (green) increases the sum of rewards over time and converges around -100, while training without minibatches (blue) converges at a much lower number, around -300.

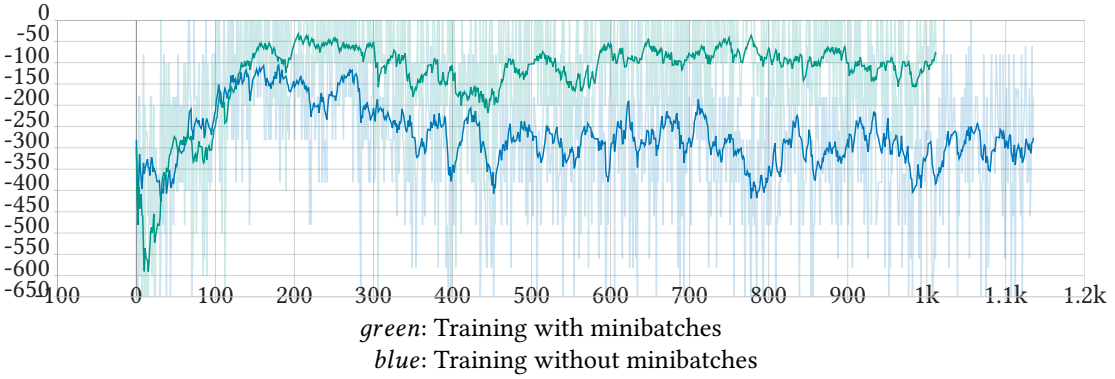
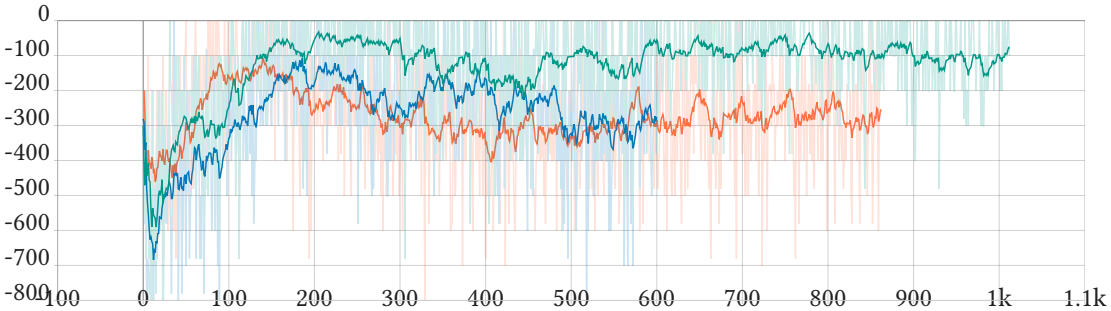


Fig. 2. Comparison of training with vs. without minibatches.

5.2.4 Setting the is_done Flag. The *is_done* flag functions as a mask for the state-value. If the flag is true ($is\_done = 1$), our method will force the state-value $V(s)$ to be 0; otherwise, the state-value $V(s)$ will be computed by the critics. A 0 state-value literally makes the state has zero value, which in turn tells the agent to visit the state as few times as possible during the training. Setting the flags for the right states will boost the learning speed since the flags will stop the agent from wasting time exploring the wrong states.

Below we tested the performance based on three settings: (1) no *is_done* flags, (2) $is\_done = 1$ if all the fires in the environment are burned out, and (3) $is\_done = 1$ whenever one of the fires is burned out. We can clearly see, with other hyperparameters at the same setting, setting (3) gives the best performance. It makes sense because in this setting, agents will try to avoid visiting states with any burned-out fire that will bring penalties. In Fig. 3, we can see that the third setting (green), which set $is\_done = 1$ whenever one of the fires is burned out, has a sum of rewards that converges to -100 over time, outperforming the other settings.

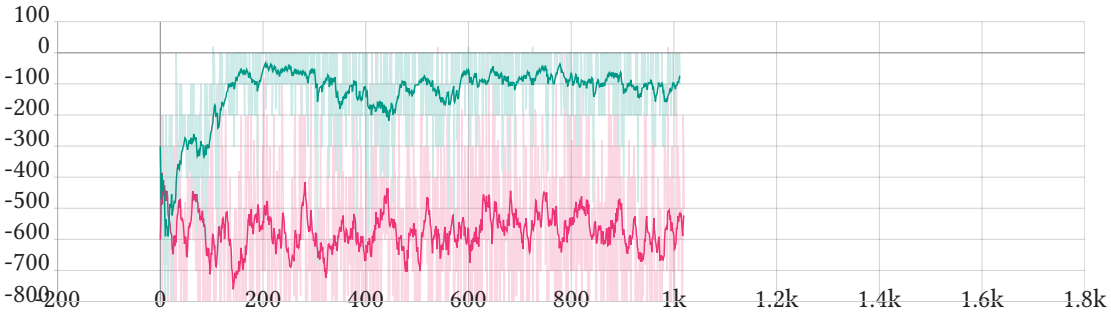
5.2.5 Comparison of Methods. Finally, we compare the performance of using the loss function derived from the regular policy optimization algorithm and the clipped surrogate objective introduced by the state-of-the-art PPO algorithm (Eq. 9). We can see in the figure below that the PPO has a clear advantage over the regular method (explain figure). When examining the policy during the training, we notice the policy, while using the regular method soon gets relatively deterministic (the most possible action has a probability over 99%) without sufficient exploration. PPO makes sure the deviation from the previous policy does not go too far when computing an update at each step that minimizes the cost function.



green: *is_done* = 1 whenever one of the fires is burned out
blue: *is_done* = 1 if all the fires in the environment are burned out
orange: no *is_done* flags

Fig. 3. Comparison of training with different *is_done* flag settings.

In Fig. 4, we can see a clear difference between the performance of training with the proximal policy optimization (green) and with the regular policy optimization (pink). With PPO, the performance improves until sum of rewards hit around -100, while with the regular method, the agent barely learns even after a thousand episode.



green: Training with proximal policy optimization
pink: Training with regular policy optimization

Fig. 4. Comparison of training with regular policy optimization vs. training with PPO.

5.3 Final Results

We plot the performance of agents of each frame for all setups, using our adapted policy optimization method. In these final experiments, we train individual neural networks for each frame at the same time instead of fixing the policy of some frames. Since different setups has different numbers of frames, the number of lines (which represent the performance of a frame) is also different.

From Fig. 5 to Fig. 8, We can observe that for all setups, the sum of rewards earned by agents in all frames converges to around −100 after over 500 training episodes. This means our neural networks for each frame are able to learn to take *legal* actions at almost every step after the training; because the penalty of taking one illegal action is −100, and the agents are able to put out zero or one fire most of the time, the sum of rewards of 15 steps being around −100 means most of the time they

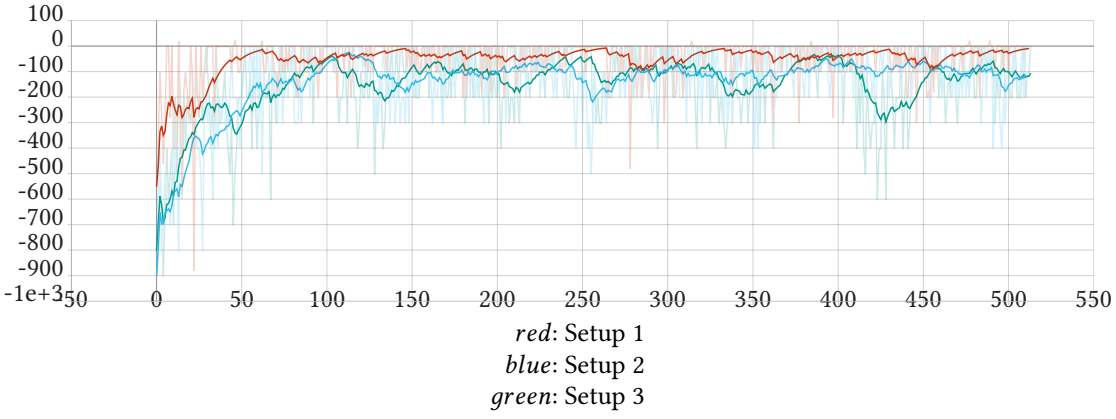


Fig. 5. Final results of $frame_0$ agent

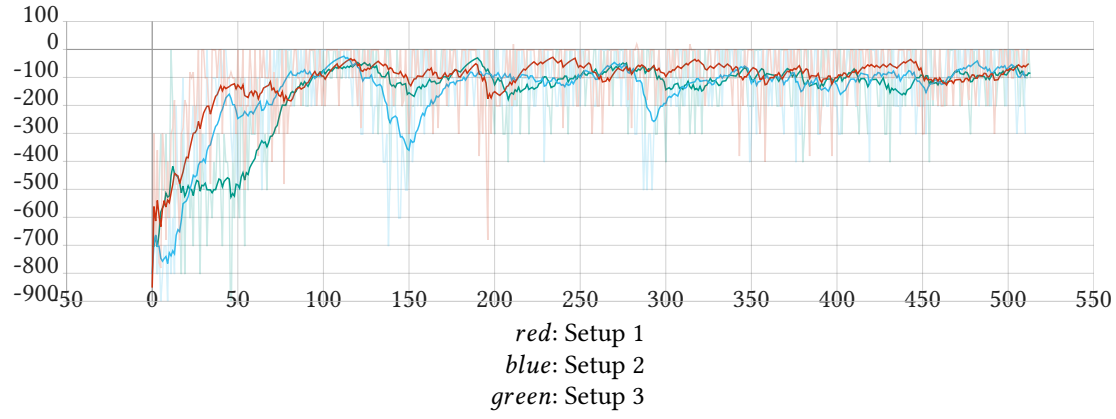


Fig. 6. Final results of $frame_1$ agent

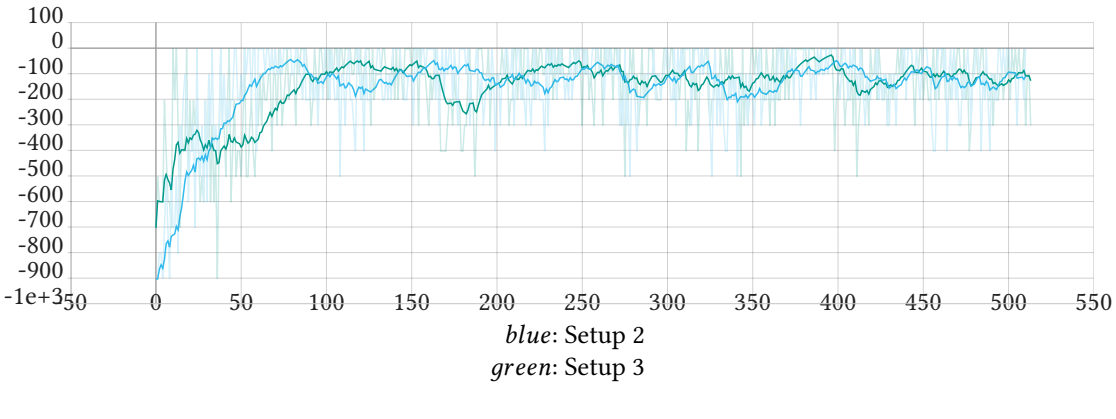


Fig. 7. Final results of $frame_2$ agent

are at least learn to take actions that do not punish them. When we take a look at each figure, we can see that setup 1 (red) has the best performance, especially for $frame_0$ agent, whose sum of

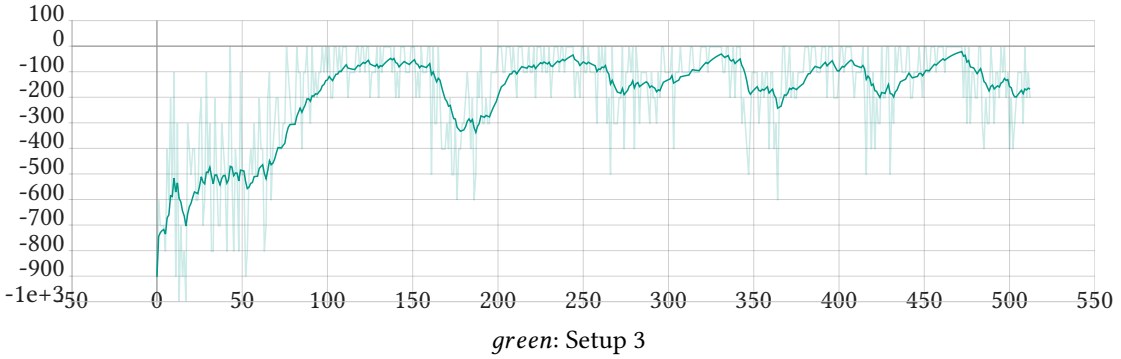


Fig. 8. Final results of $frame_3$ agent

rewards almost converges to 0, which means it rarely takes illegal actions. Setup 3 (green) shows the most unstable performance in all frames since it has the most complicated configuration of all setups.

However, once the policy is good enough to find the legal actions in a state, it ceases to improve. When we examine the policy for each frame during the training, we see that the policy became relatively deterministic by choosing the same action NOOP for each state, which means it tends to choose NOOP on every step. This is because taking other actions are capable of resulting in a penalty, for example, trying to fight the fire not adjacent to the agent gives a penalty, or even when the agent starts to learn that putting out fire gives them reward, continue fighting the now non-existent fire will result in a penalty as well. However, always NOOP results in 0 reward, which is not bad for the agent consider exploring other action are more likely to result in penalties.

This hinders the agent from learning actions that earn more reward in the long term than the "safety" actions. Since this is a *sparse reward* problem, where the big reward earned by putting out a fire requires the agent to fight the same fire several times to get the reward. The agent might quit fighting the fire before it is put out. Unfortunately, we are unable to improve our method further to overcome the problem and have the agents constantly earn positive rewards, but we have many insights into how to adapt the existing deep reinforcement learning method to open multiagent environments.

6 CONCLUSION AND FUTURE WORK

Real-world multiagent problems often involve openness, where agents may leave and join the environment over time. Simulation in the wildfire suppression problem shows that our adapted method based on the proximal policy optimization algorithm for open multiagent systems enables the agent to take legal actions in the environment almost without punishments. For future work, we plan to overcome the problem that agents stop learning once the safety actions are learned; we also plan to incorporate a multiagent model into the method instead of training individual models for each frame.

REFERENCES

M. Chandrasekaran, A. Eck, P. Doshi, and L. Soh. Individual planning in open and typed agent systems. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, UAI'16, page 82–91, Arlington, Virginia, USA, 2016. AUAI Press. ISBN 9780996643115.

A. Eck, M. Shah, P. Doshi, and L.-K. Soh. Scalable decision-theoretic planning in open and typed multiagent systems, 2019. URL https://arxiv.org/abs/1911.08642.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016. doi: 10.48550/ARXIV.1602.01783. URL https://arxiv.org/abs/1602.01783.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 07458, 3rd. edition, 2010.

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015. URL https://arxiv.org/abs/1506.02438.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 1st. edition, 2009.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, 2nd. edition, 2020.

Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation, 2017. URL https://arxiv.org/abs/1708.05144.