MASTER

Guiding Simultaneous Move Monte Carlo Tree Search via Opponent Models

Aljabasini, Obada

Award date:
2021

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Guiding Simultaneous Move Monte Carlo Tree Search via Opponent Models

Obada Aljabasini

Supervisors:
Dr. Hendrik Baier (CWI)
Dr. Meng Fang (TU/e)
Dr. Michael Kaisers (CWI)

Eindhoven, August 2021

Acknowledgement

I would like to express my sincere appreciation to my CWI supervisors, Dr. Michael Kaisers and Dr. Hendrik Baier, for giving me the chance to conduct this research at CWI. Furthermore, I am thankful to my university supervisor, Prof. Meng Fang, for accepting me for this master thesis internship at the TU/e.

Michael's insights were of invaluable importance in formulating the research question and guiding this research. He is a sparkling fountain of insights and ideas that never runs dry. His broad knowledge always surprises me; I encountered many difficulties during my work that were solved by methods he invented before. Michael never fails to make me more curious and enthusiastic about reinforcement learning.

Hendrik is a true scientist: he questioned and challenged my ideas throughout this work, pushing me to sharping my thinking and improving my work. In fact, I cannot even recall a planning paper I brought up during our meetings that he has not analyzed in-depth before. In short, Hendrik is a perspicuous scientist who is incurably curious. Above all, Hendrik and Michael are not only extremely knowledgeable and intelligent but also very humble and kind.

I would also like to thank my university supervisor, Prof. Meng, for his continuous support and commitment to ensuring that I graduate on time. Your insightful and detailed feedback on the experimental results brought my work to a higher level. Whenever I got stuck while writing this thesis, Meng was always there to help.

Last but not least, I would like to express my deepest gratitude to my parents, Nada and Khaldon, for their unceasing love, encouragement, and support. I would not be where I am today without them; the least I can do is dedicating this thesis to them.

Abstract

In n-player simultaneous-move games, the exponential growth of the search tree is amplified: the size of the joint action space is the product of the individual action spaces. Even for just a few players, the size of the action space might impede tree search algorithms from exploring deeper levels under realistic sample limits. This challenge necessitates striking a balance in-between exploring all the combinations of the actions and ignoring other players' moves. One way to mitigate this exponential growth, applied in Paranoid Search, is assuming specific types of opponent models that allow for highly effective search tree pruning, even if those models are often incorrect. Another way to alleviate this rapid growth, utilized in Progressive Widening, is modifying the selection policy to progressively increase the number of considered actions with the number of visits, enabling the search tree to explore levels at a greater depth. In this thesis, we combine both methodologies and present a novel planning algorithm called Attention-Guided Simultaneous-Move Monte Carlo Tree Search (AG-SM-MCTS). In particular, we build accurate opponent models that estimate opponents' influence and leverage them afterward to modify the sampling policy and prune the search tree. To the best of our knowledge, this is the first work that uses opponent models to improve both learning and planning in multi-player general-sum simultaneous-move games. We demonstrate that our method can yield a more skilled player, paving the way to incorporate our approach into generic meta-training frameworks, such as Expert Iteration.

Keywords: multi-agent reinforcement learning, deep learning, Simultaneous-Move Monte Carlo Tree Search, multi-player general-sum simultaneous-move games, attention mechanism.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Planning is a fundamental ability of the human mind, allowing human beings to operate in complex and dynamic environments. Indeed, many studies, such as [77, 46], suggest that such behaviors may emerge from building a predictive model that is continuously updated with new experiences to simulate the response of the environment. Motor planning and goal-oriented driving tasks, to name a few, are exemplars of such a learning process, where the predictive model is used to plan a collision-free movement trajectory [46], guided by a reward that assess how an action contributes to achieving the goal. Moreover, should the environment in which the human operates in contain other entities, such as other humans, the human brain has to model those different actors to behave flexibly. These entities are called opponents, where an opponent does not necessarily hold malicious intent; rather, it is merely another entity, running in the same environment [11].

1.1 Motivation

One approach to take opponents into account is to presume optimal behavior thereof. Afterward, one can find a safe strategy to minimize what is defined as a failure and act accordingly. However, even the most skilled and intelligent human beings, as well as other types of agents, make mistakes that could be exploited, indicating that modeling other actors in the environment may prove beneficial. In other words, instead of relying on unrealistic assumptions, such as full opponent rationality, we build statistical models of opponents based on their observed behavior. This notion of modeling other entities to exploit their strategies is known as Opponent Modeling (OM). In short, opponent modeling is an opportunity: the agent's decision-making process could be improved should more information about its opponents be provided [11].

Sampling-based planning, the use of statistics obtained via sampling to select the best action at each timestep, has been proven successful in many sequential-move complex games, such as Poker [27], Chess, and Go [72]. Opponent models could be incorporated into sampling-based planning algorithms and built by a wide variety of approaches, including Multi-agent Reinforcement Learning. Multi-Agent Reinforcement Learning (MARL) offers a way to learn behaviors through trial and error, guided by a reward based on the goal [31]. Furthermore, the rise of Neural Networks (NNs), biologically-inspired statistical models, as very powerful universal function approximators has opened new horizons for many complex MARL tasks. Combining sampling-based planning methods with Multi-Agent Deep Reinforcement Learning (MADRL) and opponent models holds considerable promise of solving complicated sequential decision-making problems, as shown in the first computer program that has beaten the world's best Go players [4]: AlphaZero [72].

However, little attention has been paid to multi-player general-sum simultaneous-move environments, leaving much room for improvement. In multi-player general-sum simultaneous-move environments, the action space grows exponentially with the number of agents, the sum of the rewards does not necessarily equal zero, and the players choose an action simultaneously at each timestep. This narrow focus could be potentially explained by the ubiquity of using board games, which are primarily sequential-move environments, as testbeds for evaluating planning and learning algorithms. Moreover, many theoretical guarantees for sampling-based planning methods, such as converging to a Nash equilibrium, do not hold in multi-player general-sum simultaneous-move games. In fact, even solving two-player zero-sum simultaneous-move games is quite challenging [17].

One way to utilize opponent models in multi-player general-sum simultaneous-move environments is to leverage the attention mechanism to reduce the branching factor. In particular, we prioritize opponents' actions based on opponents' estimated importance. For example, a football player can decide their next action based on their opponents' (i.e., their team and the competing team) possible moves. Concretely, they might advert more attention to closer players and plan over their actions extensively while paying less attention to farther opponents. The idea of estimating the importance of a component via the attention mechanism has been studied extensively in many Artificial Intelligence (AI) fields, such as Natural Language Processing (NLP) and Computer Vision (CV). However, it still has not been completely elucidated in RL.

1.2 Problem statement

Recent works, such as multi-player AlphaZero [61], have been applied to many multi-player sequential tasks, such as board games, but did not tackle the problem of having a large number of opponents whose moves should be explored. This challenge is further amplified when the actions are simultaneous, as even one lookahead step would require searching a vast action space. For instance, if the multi-agent system consists of ten agents and the cardinality of the action set for each agent is ten, then we might need to consider all the possible combinations of those actions, which amounts to 10^{10}. To put the complexity of such systems into perspective, the average number of legal moves (i.e., the branching factor) of Chess is approximately 31, which is much smaller than relatively simple simultaneous-move games such as Pommerman [63], whose branching factor is 1296. Therefore, our main research question is the following:

Could opponent models be exploited to improve learning and planning performance in multi-player general-sum simultaneous-move games?

Specifically, we aim to answer the following subquestions:

- Assuming the availability of opponents' moves during learning, how can we learn opponent models?
- Given opponent models, whether they were provided or learned, how can they be exploited to search at a greater depth?

1.3 Related work

The literature on sampling-based planning has been on the rise in recent years to the extent that even enumerating relevant surveys, such as [18, 93], becomes difficult. This rapid growth can be attributed to the rise of computing power and the ubiquity of multi-agent systems, as most real-world problems are multi-agent by definition. For our purposes, we focus on search tree pruning methods that either use opponent models or modify the sampling distribution, as we combine both in our work.

Numerous works build opponent models to (hard) prune the search tree by assuming typically inaccurate opponent models. For instance, Paranoid Search [75] assumes that opponents form a coalition, making the game two-player and zero-sum, such that classical pruning techniques, such as Alpha-Beta pruning [74], can be applied. This assumption does not usually hold in practice but still allows exploration of deeper levels.

Best Reply Search (BRS) [67] follows the same assumption but restricts the moves of the coalition to the opponent with the strongest move against the agent [14], alleviating the pessimistic supposition. However, such a restriction may result in illegal states. BRS+ [23] fixes this issue if a move ordering function is provided [14]. In particular, opponents whose moves were not the strongest against the agent opt for their highest order move. In contrast to these methods, we assume accurate opponent models that could be learned if not provided.

Other methods (soft) prune the search tree by altering the selection policy. First-Play Urgency (FPU) [84] assigns a constant c to the upper confidence bound of unexplored actions (instead of ∞), yielding a better performance should the constant be adequately selected. Progressive Widening (PW) [22] progressively increases the number of considered actions with the number of visits; however, it requires a heuristic function to sort the actions; otherwise, actions will be explored randomly.

Multi-player AlphaZero [61] leverages a neural network to estimate the action probabilities, which are used to (soft) prune actions in the short run. The use of the estimated action probabilities to modify the upper confidence bound is known as PUCT (Predictor + upper confidence bound applied to trees) [49]. However, opponents' actions are still equally explored. In contrast, our work modifies progressive widening to incorporate opponent models and explore opponents' actions according to their influence on the agent. Additionally, our model learns a move ordering function from scratch, whereas most previous works, such as [55], hand-craft it.

1.4 Outline

This thesis is arranged as follows. Chapter 2 provides the necessary background for the development of our method, discussing Monte Carlo Tree Search and reinforcement learning, which play a crucial role in our model. Chapter 3 is the core of this thesis, where we introduce the proposed model and describe its building blocks, starting from altering Simultaneous-Move Monte Carlo Tree Search to accommodate the estimated opponents' importance and ending with presenting our main contribution: Attention-Guided Simultaneous-Move Monte Carlo Tree Search (AG-SM-MCTS). Afterward, in Chapter 4, we benchmark our method against the baseline using the computer game Pommerman [63], showing a significant increase in the score over the baseline, even when the baseline is enhanced with several state-of-the-art algorithms. Finally, in the last chapter, Chapter 5, we summarize our findings and discuss potential research directions that could be pursued further.

Chapter 2

Background

In this section, we lay the theoretical ground of our work. First, we introduce Monte Carlo Tree Search (MCTS) as a prominent sampling-based planning technique that could be combined with numerous learning algorithms. Then, we present an extension of MCTS that accommodates simultaneous moves: Simultaneous-Move Monte Carlo Tree Search (SMMCTS). Next, we discuss Reinforcement Learning (RL) as another approach to solving sequential decision-making problems, where opponent models can be implicitly learned via self-supervised learning. Afterward, we introduce Neural Networks (NNs) as universal function approximators that could be leveraged in RL to estimate action probabilities and value functions. Furthermore, we elaborate on two of the most commonly used NN architectures in RL: Convolutional Neural Networks (CNNs) and Long Short Term Memory networks (LSTMs). Finally, we describe the attention mechanism, an emergent algorithm to selectively concentrate on the components of a deep learning model based on their importance.

The importance of Game Theory (GT), the mathematical study of interaction among decision-makers [53], has grown exponentially since the second world war [62]. Such growth could be explained by the applicability of GT in many scientific fields, such as robotics and economics, as they involve several agents (or players) that interact and affect each other. However, simulating these real-world scenarios and evaluating any proposed solution can be risky, as it may entail severe damage that may even lead to life-threatening situations. For example, two men were killed in Texas due to a malfunction of the Tesla's autopilot system [1]. Because computer games are easier to simulate, pose similar challenges to those of real-world tasks, and can be seen as abstracted versions of real-world scenarios, they are a useful tool to assess any game theory algorithm [62].

To this end, we formalize the mathematical framework we build on. We define a Markov game G [70] by a tuple (n, H, S, A, P, R, γ), where:

- n is the number of agents.
- H is the horizon, which defines the maximum length of an episode.
- S is the set of the joint states.
- A is the set of the joint actions.
- P: S $\times$ A $\times$ S $\to [0, 1]$ is the transition probability function, which represents the probability of transitioning from state s to state s' by taking the joint action a.
- R: S $\times$ A $\to \mathbb{R}$ is the reward function, which denotes the immediate rewards the agents receive by taking a joint action a.
- γ is the discount factor, which determines the relative importance of future rewards compared to immediate rewards.

The game should also satisfy the Markov property; concretely, the probability of transitioning to a S_t and obtaining a reward of R_t depends only on the immediately preceding state S_{t-1} and action A_{t-1} [76]. We define a policy (or a strategy) of player i, π_i, as a distribution over actions $\pi_i(S)$ for each state S. A strategy profile Π is a collection of the strategies of the players

$\Pi = \{\pi_1, \pi_2, .., \pi_n\}$. Finally, we define Π_{-i} as the strategies in Π excluding the i-th player's strategy, π_i [62].

Markov games are characterized by several factors, including the number of players [62]. For our purposes, we assume that we control one player (called the agent), and denote other players as opponents (i.e., we do not differentiate between teammates and enemies). Furthermore, we assume that opponents are independent: they do not form teams (also known as a coalition in GT), where the correlations between these opponent models may warrant modeling them explicitly. Moreover, we assume that the game is general-sum: the sum of the reward is not necessarily zero. Despite these restrictions, many computer games and real-world settings comply with this framework.

Now that we have our theoretical framework established, we can investigate the central question in GT and sequential decision-making in general:

How do we find a "good" policy?

The most common way, and one of the fundamental concepts in GT, to find a reasonable strategy is computing a best-response strategy. A best-response strategy for player i given strategy profile Π is a strategy that maximizes the i-th player's expected reward. However, the agent is oblivious of opponents' strategies, warranting finding best-response strategies for its own best-response strategy. In particular, we define an ϵ-Nash-Equilibrium (NE) as a strategy profile that satisfies [62]:

$$\forall i \in \{1, 2, .., n\} : R_i(\Pi) + \epsilon \leq \max_{\pi'_i} R_i(\pi'_i, \Pi_{-i})$$

where $R_i(\Pi)$ is the reward agent i receives according to the strategy profile Π and $R_i(\pi'_i, \Pi_{-i})$ is the reward agent i receives if it follows the strategy π'_i and the other agents follow the strategy profile Π_{-i}.

Note that if $\epsilon = 0$ (called an exact Nash equilibrium), no agent needs to change its strategy, as deviating from a Nash equilibrium decreases the expected reward. A sensible approach, then, is to compute the Nash equilibrium, as rational opponents will not intentionally decrease their reward. Yet, computing an exact Nash equilibrium is PPAD-complete [57]; thus, most works instead resort to approximating a Nash equilibrium (i.e., finding a ϵ-Nash-Equilibrium (NE)).

Generally, two types of approaches to finding an approximate Nash equilibrium exist in the literature: abstracting (i.e., simplifying) the game and utilizing sampling-based planning algorithms. The former methodology is game-specific and cannot be generalized, but has shown success in numerous games, such as Poker [16, 92], nevertheless. Sampling-based planning approaches, in contrast, are generic and could be applied to any game. However, most sampling-based planning methods are not guaranteed to converge to a NE when applied to multi-player general-sum simultaneous-move games [39], although they may still yield strong players [39]. Moreover, finding a NE might be sensible if no information about opponents is provided, which might not be the case in many settings.

Thus, finding a NE is cumbersome, overly pessimistic (i.e., assumes fully rational opponents), or both. This limitation motivates learning a best-response strategy to opponents' estimated strategies, which begs the following question: given opponent models, how can we find a corresponding best-response strategy? Furthermore, if the action space is combinatorial, how can we effectively search opponents' moves? To answer these questions, we first need to introduce two approaches that could be used to find a good strategy in general: planning and reinforcement learning.

2.1 Planning

Human beings do not consider only immediate rewards; rather, they imagine how opponents and the environment would react to their actions and modify their strategy accordingly. Tree-based search algorithms are built upon this idea: at every timestep, the root player imagines what the opponents could do and builds a tree progressively. However, many games have a large action space and do not finish in 5 or 6 moves. Thus, computers cannot consider all possible sequences of actions. One way to mitigate this challenge is to expand the tree to a certain level and then apply an evaluation function that assesses how good a state is. However, this kind of evaluation function is game-specific and relies on domain knowledge, impeding the usage of such tree-based methods for an arbitrary game. While many works, such as [14], utilize such domain-dependant evaluation functions, it is not trivial to invent such functions in many games. This limitation imposes a new problem: how can the evaluation function be devised? One way to elude this problem in light of tree search methods is to rely on randomness instead, which is the premise of Monte Carlo Tree Search (MCTS).

2.1.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [19] is a sampling-based planning approach that relies on randomness by performing Monte Carlo simulations to evaluate a state. Specifically, a (usually random) policy is used to sample trajectories until a terminal state is reached, for which the reward is computed. Thus, MCTS could be used out-of-the-box without having to hand-craft an evaluation function. For this purpose, a visit count n_s for the corresponding state, a visit count for each action $n_{s,a}$, and a value estimate for each action $\bar{X}_{s,a}$ are maintained in each node. A vanilla MCTS iteration (see Algorithm 1) consists of four steps [19], as follows:

- **Selection:** at each node, an action is selected to be explored. Numerous ways to select this action exist in the literature. A simple and effective way is using Upper Confidence Bound (UCB) [37] which strikes a balance between exploitation and exploration, controlled by an exploration parameter C. Concretely, the UCB score is defined as follows:

$$\text{UCB}(s,a) = \bar{X}_{s,a} + C\sqrt{\frac{\log n_s}{n_{s,a}}} \tag{2.1}$$

  where $\bar{X}_{s,a}$ is the online value estimate for an action a at state s, C is the tunable exploration parameter, n_s is the number of times state s has been visited, and $n_{s,a}$ is the number of times an action a has been selected from state s, so far. For instance, Figure 2.1 depicts a chosen path (marked in red).

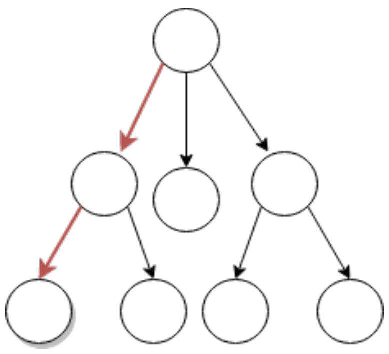


Figure 2.1: MCTS's selection step, where the red arrows indicate the chosen path. Image reproduced from [18].

- **Expansion:** if the reached state does not exist in the tree, a new node will be created and added as a new leaf. Figure 2.2 shows such a case, where the blue arrow points to the new node.

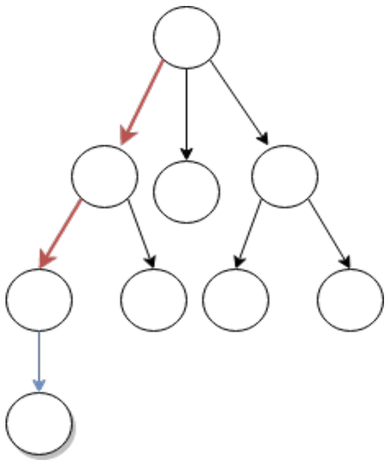


Figure 2.2: MCTS's expansion step, where a new child is added to the selected node. Image reproduced from [18].

- **Rollout:** to evaluate the new node, a policy is used to select actions to be performed until a terminal state is reached. Figure 2.3 illustrates how a Monte Carlo simulation is carried out starting from the new node.

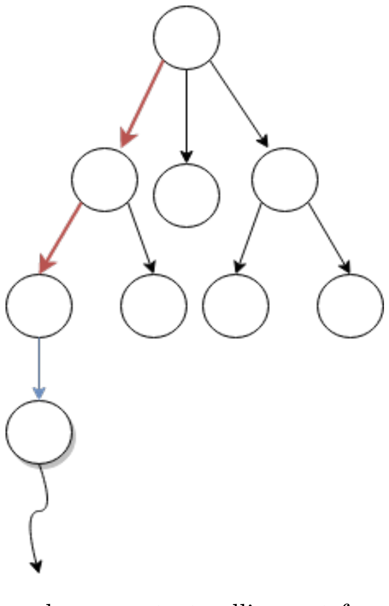


Figure 2.3: MCTS's rollout step, where we start rolling out from the new node until we reach a terminal state. Image reproduced from [18].

- **Backpropagation:** after the rollout, the estimated value is backpropagated through the path from which the node was reached. In particular, the reward is added to the online value estimate $\bar{X}_{s,a}$, and n_s and $n_{s,a}$ are increased by one. Figure 2.4 demonstrates which nodes are updated for the chosen path (marked in red).

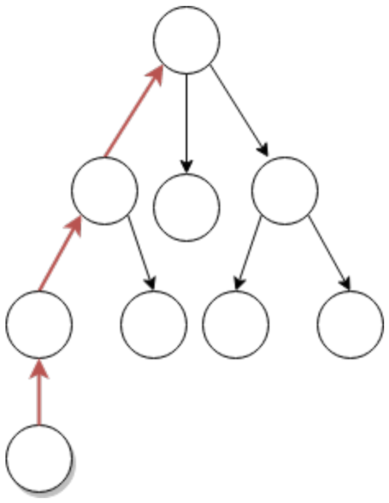


Figure 2.4: MCTS's backpropagation step, where the value estimate and visit counts of each node that lay in the chosen path are updated. Image reproduced from [18].

After a predefined number of iterations m, an action from the root node is chosen for the next timestep. Several mechanisms to choose that action are present in literature [20]. For instance, the action associated with the max child (the one whose online-estimated value is the highest) could be selected. Yet, most recent works opt for choosing the child who has been visited the most, as this kind of selection is more robust compared to the other approaches (which is the reason this method is sometimes called robust child) [20].

Algorithm 1 Monte Carlo Tree Search (MCTS)

```
1: procedure MCTS($S$)
2:     if is_terminal($S$) then
3:         return reward(S)
4:     valid_actions = valid_actions($S$)
5:     ucbs = UCB(S, valid_actions)
6:     $a$ = argmax(ucbs)                                  ▷ Selection step
7:     next_state = step($S, a$)                ▷ Compute the next state
8:     if $a \notin$ children($S$) then
9:         $v$ = rollout(next_state)                         ▷ Rollout step
10:        add_child($S, a, v$)                            ▷ Expansion step
11:    else
12:        $v$ = MCTS(next_state)
13:    backpropagate($S, v$)                      ▷ Backpropagation step
14:    return $v$
```

A fundamental problem of vanilla MCTS is the assumption of non-triviality [84]: the presumption that the number of iterations is reasonably larger than the branching factor. Should this assumption not hold, the tree will be very shallow and barely hold any meaningful estimations. One of the simplest methods to tackle this problem is First-Play Urgency (FPU) [84]. If the action a is not visited yet, the equation 2.1 will yield an infinite value, which forces MCTS to explore all the actions before exploiting any action, even if some actions were promising (i.e., yielded a win). A constant c could be used instead for unexplored actions [84]: $n_{s,a} = 0 \Rightarrow \text{UCB}(S, a) = c$. For instance, a reasonable value of c is $\frac{1}{n}$, where n is the number of the opponents, as it reflects the assumption that most game states are balanced in terms of winning probability.

While FPU is an intuitive and straightforward approach, it only slightly alleviates the curse of dimensionality by guiding MCTS to more promising actions, as it does not change over search time. In contrast, Progressive Widening (PW) [22] commences by ordering actions according to a move ordering function and then restricts the selection step to a small number of actions. Thereafter, it progressively (hence the name) considers more actions as the number of visits increases. Although it may be counter-intuitive, Wang et al. [83] showed that PW can still boost MCTS's performance, even if the actions were ordered randomly [22]. Algorithm 2 shows the PW algorithm, applied to a state S, where the widening factor C and the widening exponent α are hyperparameters that control the growth of the search tree.

Algorithm 2 Progressive Widening (PW)

```
1: procedure PW($S$)
2:     $n$ = number_of_visits($S$)
3:     $k = \lceil Cn^{\alpha} \rceil$
4:     actions = valid_actions($S$)
5:     sorted_actions = sort(actions, $f$)                ▷ f is a move ordering function
6:     selected_actions = select_first(sorted_actions, $k$)
7:     ucbs = UCB(S, selected_actions)
8:     $a$ = argmax(ucbs)
9:     return $a$
```

Nevertheless, we are back to square one: instead of finding a way to design a state evaluation function, which was solved using randomness, we need to devise a move ordering function. Learning this move ordering function using machine learning might be a reasonable approach, and we will later introduce opponent models for this purpose. Before introducing opponent modeling and Reinforcement Learning in general, we need to extend MCTS to accommodate simultaneous-move games, as sequential-move environments are only a subset of the former. In particular, sequential-move environments are a special case of simultaneous-move environments, where we assume that only one agent takes an action and the other agents choose the "no-operation" action (i.e., stand still).

2.1.2 Simultaneous-Move Monte Carlo Tree Search

MCTS was originally designed for sequential-move games. However, many real-world tasks are simultaneous, as agents do not wait for each other's move. MCTS could be modified to accommodate simultaneous-move games by considering joint actions instead of a single-player action at each node. Simultaneous-Move Monte Carlo Tree Search (SMMCTS) [39] was introduced for this purpose, where each node holds a payoff matrix of m dimension (where m is the number of agents) that represent the estimated reward when specific indices (actions) are chosen. While a variety of methods can be used to select actions in SM-MCTS, decoupled UCT is the most straightforward one [39], where each player selects the best action for itself without considering other players' moves. The UCB score is then computed for each player individually, as follows:

$$\mathrm{UCB}_i(S,a) = \bar{X}^i_{s,a} + C_i\sqrt{\frac{\log n_s}{n^i_{s,a}}}$$

A search iteration in SM-MCTS is very similar to that of MCTS, with only minor modifications to handle simultaneous moves [39], as shown in Algorithm 3. SM-MCTS, enhanced with PW for each player, can produce a strong player, but a heuristic function for each player to sort its actions is still needed. The following section presents the framework of reinforcement learning, in which opponent models will be learned.

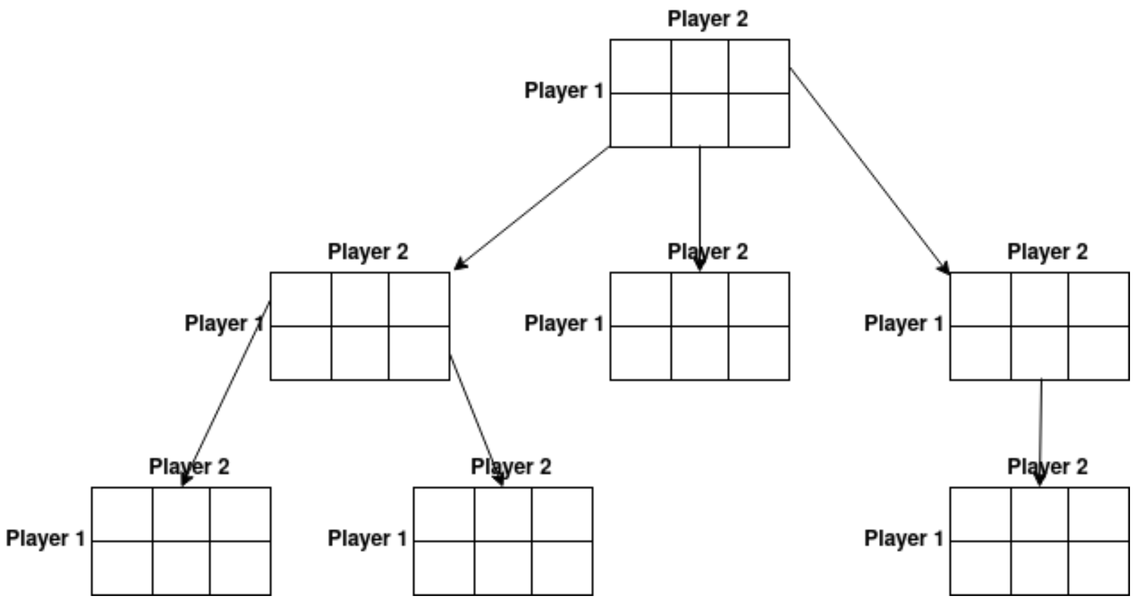


Figure 2.5: Simultaneous-Move Monte Carlo tree search (SM-MCTS).

Algorithm 3 Simultaneous-move Monte Carlo Tree Search (SM-MCTS)

```
1: procedure SMMCTS($S$)
2:     if is_terminal($S$) then
3:         return rewards(S)
4:     actions = []
5:     for $i \in$ [1..number_of_agents] do
6:         actions = valid_actions$_i$($S$)
7:         ucbs = UCB$_i$($S$, selected_actions)
8:         $a$ = argmax(ucbs)
9:         actions.append($a$)
10:    next_state = step($S$, actions)
11:    if actions $\notin$ children($S$) then
12:        $v$ = rollout(next_state)
13:        add_child($S$, actions, $v$)
14:    else
15:        $v$ = SMMCTS(next_state)
16:    backpropagate($S$, $v$)
17:    return $v$
```

2.2 Reinforcement Learning

Reinforcement Learning could be defined as learning what to do to maximize a numerical reward signal [76]. Specifically, the agent learns to choose actions that yield the largest cumulative reward by interacting with the environment [76]. Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a policy, a reward signal, a value function, and a model of the environment [76]. A policy π is a map from states to actions, and it may be deterministic or stochastic (i.e., a probability distribution over actions). A reward R is the immediate evaluation of an action for the goal the agent aims to achieve. Rewards indicate what is good in an immediate sense; however, we are usually interested in what is good in the long run [76]. A value function $V_\pi(S)$ measures exactly that, namely the expected accumulated reward starting from a specific state S and following the policy π. Figure 2.6 depicts the interaction between the agent and the environment, where the agent takes an action A_t at a timestep t, and observes a reward R_{t+1} and a new state S_{t+1}.

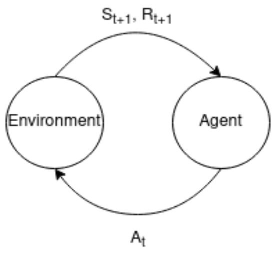


Figure 2.6: The interaction between the agent and the environment.

Formally, maximizing the cumulative reward could be framed as solving a Markov Decision Process (MDP). An MDP is a single-agent Markov game, which is defined by a quadruple (S, H, A, R, P, γ) [76], where S is the set of states, H is the maximum length of an episode (called the horizon), A is the set of actions, R is the immediate reward function, P is the transition probability function, and γ is the discount factor. The reward and transition probability condition on the current state, next state, and the chosen action [76]. In particular, $R(s, s', a)$ denotes the reward received by taking an action a and transitioning from state s to state s', and $P(s_{t+1} = s'|s_t = s, a_t = a)$ is the probability of moving from state s to state s' by taking the action a. Furthermore, all states should satisfy the Markov property: the probability of each possible value for S_t and R_t depends only on the immediately preceding state S_{t-1} and action A_{t-1}[76]. The discount factor γ determines the relative importance of future rewards for the agent.

Figure 2.7 shows a simple MDP, where S = {"S1", "S2", "S3", "S4"} and each arrow carries a tuple that indicates the transition probability and the immediate reward, respectively. Generally, three types of methods could be used to solve an RL problem: value-based approaches, policy-based approaches, and actor-critic approaches (the combination of value-based and policy-based approaches).

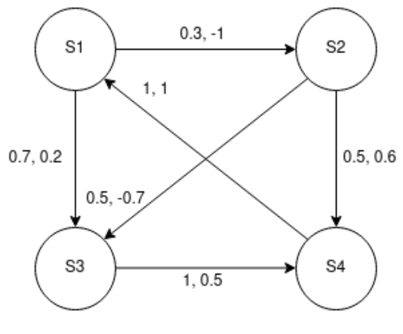


Figure 2.7: A simple MDP of four states.

2.2.1 Value-based methods

Value-based methods estimate the value function $V(S)$ or the action-value function $Q(S, a)$, and use these estimations to select the most promising action. Concretely, let $Q_\pi(S, a)$ denote the expected reward starting from the state s, taking the action a and following the policy π afterward, then the Q-function is [76]:

$$Q_\pi(S, a) = \sum_{s', r} p(s', r|s, a)(r + \gamma V_\pi(s'))$$

Let $Q^*(S, a)$ denote the expected cumulative reward obtained by choosing the action a from state s and, thereafter, following the optimal policy. In that case, the optimal policy can be computed by taking the action that yields the largest expected reward in all states (i.e., $\pi^*(S) = \text{argmax}_a Q^*(S, a)$). The Q-value iteration algorithm starts from initial Q-values and iteratively updates the Q-values using the Bellman equation such that the Q-value function converges to Q^* [76]:

$$Q_{k+1}(S, a) = \sum_{s'} p(s'|s, a)(R(s, s', a) + \gamma \text{max}_{a'} Q_k(S', a'))$$

This iterative algorithm is called Q-learning. Q-learning cannot scale to a large number of states and actions due to the curse of dimensionality, as the Q-value for each combination of states and actions needs to be maintained. Worse, if the action or state space is continuous, the Q-values cannot be tabulated or enumerated. Heeding this limitation, Deep Q-Learning [50] utilizes a neural network to approximate Q-values, which helps generalize across seen and unseen states.

2.2.2 Policy-based methods

Instead of computing the value or Q-value functions as a detour to finding a policy, we can try to directly compute the policy, which offers two advantages. First, computing $\text{argmax}_a Q(s, a)$ could be computationally expensive for high dimensional action spaces. Second, the policy could sometimes be simpler to compute than Q-values [9].

Let $U(\theta)$ be the expected sum of rewards using the policy π_θ (i.e., the policy is parameterized by θ). Then, we are interested in finding the optimal θ that maximizes the expected sum of rewards [9]:

$$\theta^* = \text{argmax}_\theta U(\theta) = \text{argmax}_\theta \mathbb{E}_T[R(T)] \approx \text{argmax}_\theta \sum_T P(T)R(T)$$

where T is a trajectory (i.e., a sequence of states and actions) and $R(T)$ is the sum of rewards for T.

To find the optimal θ, we compute the gradient of $U(\theta)$ with respect to θ and use gradient ascent [65] afterward. The gradient $\nabla_\theta U(\theta)$ can be estimated via sampling, as follows:

$$\nabla_\theta U(\theta) \approx \frac{\sum_{i=1}^{n} \nabla_\theta \text{log}(P(T_i))R(T_i)}{n}$$

where n is the number of sampled trajectories and T_i is the i-th trajectory. The gradient ascent algorithm will automatically change θ such that the probability of trajectories with larger rewards increases. However, this means that it will still raise the probabilities of trajectories with small rewards [9]. Instead, we want to decrease the probabilities of those trajectories, even if rewards are always positive. One way to achieve this is to subtract a baseline b from the reward, which makes the right-hand side negative if the trajectory's reward is small [86]. Moreover, subtracting a baseline may reduce the variance of the gradient estimator [9]. Algorithms that compute this baseline are called actor-critic methods.

2.2.3 Actor-critic methods

One very prominent approach that has been very successful in computing the baseline in recent years is estimating the value function. This wide usage of the value function as a baseline may be attributed to having fewer degrees of freedom than the Q-function. To reduce the variance even further, we drop the rewards of previous actions, as older actions should not be considered for updating the probability of newer actions [9]. Furthermore, the immediate reward for an action can be approximated by the Q-function evaluated at that action, which might also decrease the variance [9]. Finally, the same model can be used to approximate both the Q-value and value functions, as shown by the Bellman equation [76]:

$$Q(s,a) = \mathbb{E}[r_0 + \gamma V(s_1) | s_0 = s, a_0 = a]$$

where only one lookahead step is used, which can be also varied [9].

The function approximator of the value function is called the critic, whereas what updates the policy based on the critic is called the actor [51]. The difference between the Q-function and the value function is called the advantage function $A(s,a) = Q(s,a) - V(s)$, as it indicates the reward obtained by taking that particular action compared to the expectation (i.e., the value of the value function at state s). Actor-critic approaches could be perceived as a combination of value-based and policy-based approaches because learning involves estimating both the value and policy functions. In conclusion, the gradient of $U(\theta)$ is computed as follows:

$$\nabla_\theta U(\theta) \approx \frac{\sum_{i=1}^{n} \sum_{t=0}^{H} \nabla_\theta \log(P(T_i)) A(s_t^i, a_t^i)}{n}$$

While this approach could be used for solving many RL problems, it suffers from two drawbacks. First, it only exploits one part of the state space [51], as only one agent is used to interact with the environment. Second, the gradient ascent updates are correlated, as, once again, only one worker collects trajectories. To overcome these limitations, Asynchronous Advantage Actor-critic (A3C) [51] uses multiple workers for exploring the state space to reduce the correlation between the updates. Specifically, each agent runs on a separate process, and a shared model is updated asynchronously. Figure 2.8 illustrates the architecture of A3C, where multiple workers copy the shared model's parameters before collecting a trajectory and then interact with their own environment. Afterward, the gradients, computed by gradient ascent, are used to update the shared model's parameters.

A3C employs a neural network with two heads (outputs) to estimate the policy and value functions, which correspond to the probability of the actions and the expected value of the state, respectively. In particular, the policy head's output is an n-dimensional vector, where n is the number of actions, and the value head's output is a scalar that indicates the expected cumulative reward starting from the input. Intuitively, the loss function consists then of two terms: a policy loss function and a value loss function [51], as follows:

$$L = L_{\text{policy}} + L_{\text{value}}$$

However, the model might overfit its world, which might increase the peakedness of the policy function π [87]. Thus, an entropy term is usually added to encourage exploration and hinder the convergence to deterministic policies [87]. Most of the state-of-art RL algorithms, such as Proximal Policy Optimization (PPO) [68], incorporate a similar term. Hence, the total A3C loss is defined as follows [33]:

$$\begin{aligned} L &= L_{\text{policy}} + L_{\text{value}} + L_{\text{entropy}} \\ &= -\lambda_{\text{policy}} \mathbb{E}_{s\sim\pi}[R_{1:\infty}] + \lambda_{\text{value}} \mathbb{E}_{s\sim\pi}[(R_{t:t+n} + \gamma^n V^\pi(s_{t+n+1}) - V^\pi(s_t))^2] - \lambda_{\text{entropy}} \mathbb{E}_{s\sim\pi}[H(\pi(s)] \end{aligned}$$

where n is the number of lookahead steps, $H(\pi(s))$ is the entropy of the policy function, λ_{policy} is the weighting parameter for the policy loss, λ_{value} is the weighting parameter for the value loss, and $R_{a:b}$ is the discounted accumulated reward between the timesteps a and b.

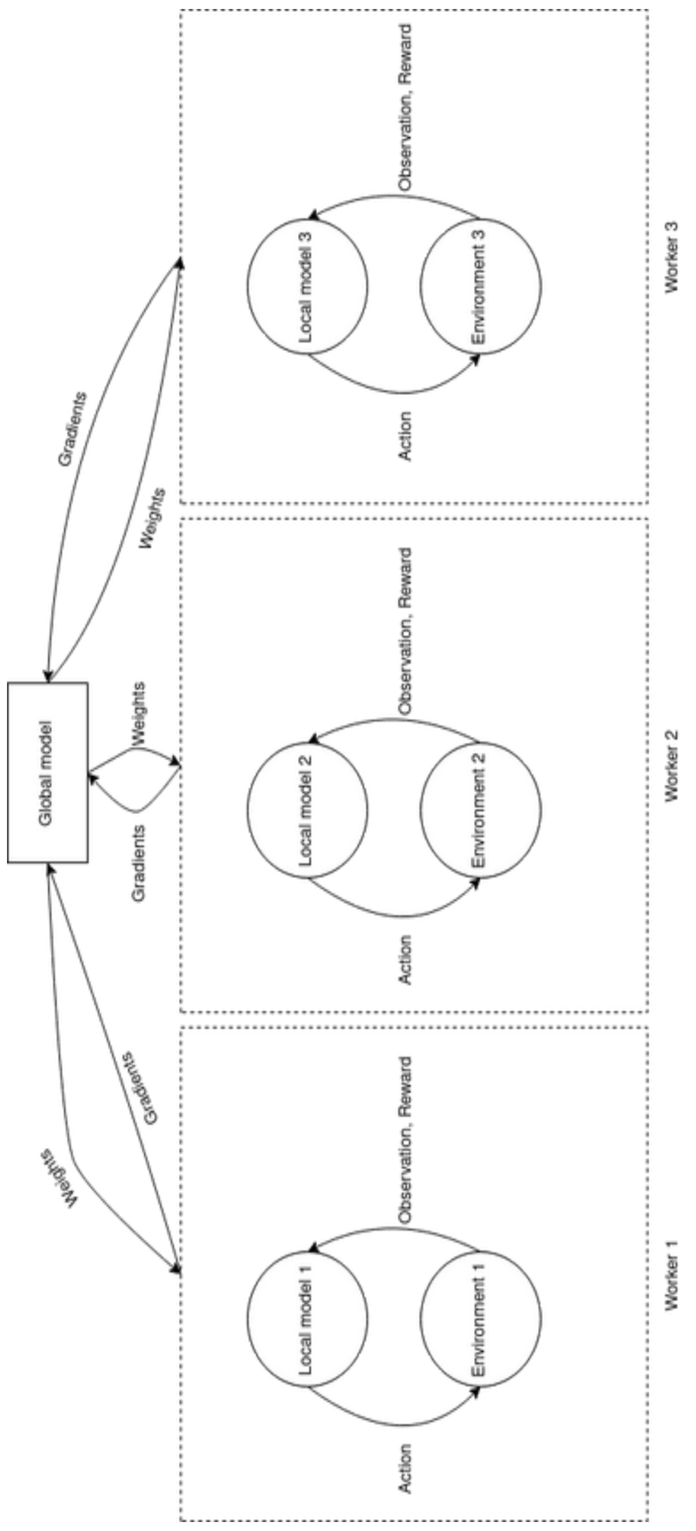


Figure 2.8: The A3C architecture, where only three workers are shown for simplicity.

The very successful single-agent RL algorithms do not perform well when applied to multi-agent environments, as treating other agents as part of the environment, even if their policies are stationary, renders the problem non-stationary [29]. Furthermore, the actions of each agent affect the behavior of the others, and the agent cannot determine whether its actions serve its goal. This problem is widely known as the credit assignment problem: the agent's performance is affected by the environment and behaviors of other agents. These two challenges, namely the non-stationarity of the environment and credit assignment, have been well-studied in the literature [91, 58] but still pose a difficulty for multi-agent RL algorithms.

One way to tackle these problems is opponent modeling. Opponent modeling could be used to ameliorate the agent's decision-making reasoning [11]. If we assume that opponents' strategies are stationary and use opponent models, then finding a (best-response) policy in a multi-agent environment could be reduced to solving a standard MDP, where single-agent RL algorithms can be used [30]. Thus, A3C, endowed with opponent models and when opponents' policies are stationary, can be used to solve multi-agent problems. Self-supervised learning is a straightforward method to learn opponent models, where we assume opponents' actions can be observed. In particular, we follow the framework of centralized-training-decentralized-execution (CTDE), where the agent can observe the actions of the other agents during training. In contrast, in the decentralized-training-decentralized-execution (DTDE) framework, the agent estimates other agents' actions using only its observations, as done in works such as [88].

2.2.4 Self-supervised learning

The term self-supervised learning (SSL) is quite overloaded and has been redefined in several fields. Nevertheless, the core methodology is the same: the automatic generation of supervisory signals to shape the features of the main model. Or in other words, SSL is defined as training a supervised model from signals that could be extracted from the task to facilitate representation learning. In RL, SSL typically takes the form of sharing a common representation between the main task, which is finding the optimal policy, and one or more auxiliary tasks.

For instance, in [38], uses a separate neural network that shares the same representation of the observation with the main model to predict game features, such as the existence of an opponent or a weapon in the current frame. Similarly, two auxiliary tasks were added in UNREAL [33] to improve representation learning: predicting the next reward giving the last three frames and learning a separate policy to change pixels maximally. AMFA3C [30], which serves as the backbone of our model, treats opponent modeling as an auxiliary classification task, where the ground truth is the action the opponent took. By minimizing the cross-entropy loss [56], the agent can approximate the policy of the opponent. These methods share the same idea, namely adding an auxiliary loss function to the original loss function, as follows:

$$L = L_{\text{RL}} + L_{\text{auxiliary}}$$

For instance, in AMFA3C, the loss function is:

$$L = L_{\text{A3C}} + \sum_{i=1}^{m} L_{CE}^{i} = L_{\text{A3C}} + \sum_{i=1}^{m} \lambda_i (\frac{1}{n} \sum_{j=1}^{n} a_i^j \log(\hat{a}_i^j)])$$

where m is the number of opponents, n is the number of actions, and λ_i is the weighting parameter for opponent i. The effect of incorporating auxiliary tasks in RL is not well understood and seems to be quite sensitive to the weighting parameters [71], although it could yield promising results even if those hyperparameters are not thoroughly tuned [71].

2.2.5 Reward shaping

The beforementioned RL formulation and methods presume that the agent receives a dense reward at each timestep [76]. The sparsity of rewards (i.e., only receiving a reward at the end of the episode) may hinder (and even prevent) learning [76], as winning an episode might require the agent to be very skilled, especially when faced with skilled opponents. Consequently, incorporating an intrinsic reward, also known as reward shaping [47], encourages exploration and hence expedites learning [34]. Reward shaping is a very active research area; however, we only focus on novelty-based intrinsic rewards for our purposes. Curiosity, a form of novelty-based rewards, is inspired by human infants' propensity to explore unencountered states, even if no extrinsic reward is obtained [34]. In discrete state space environments, a curiosity-based reward can be added to the extrinsic reward if a new state is encountered. In other words, the agent receives a higher reward when it encounters frequently-less-visited states [34]. In particular, the Bellman equation is modified to include a curiosity-based reward:

$$Q(S,a) = \sum_{s'} p(s'|s,a)(R(s,s',a) + \gamma \text{max}_{a'} Q(S',a') + N(s))$$

where $N(S)$ is the novelty-based reward. Before discussing our main method and contribution, a brief introduction to two commonly used neural network architectures in RL is vital to grasp our model.

2.3 Neural Networks

Representation learning is a crucial component in RL, as extracting features from high-dimensional spaces is complex and cumbersome. Neural Networks (NNs) overcome this limitation by automatically learning key features using backpropagation [66] in a fully differentiable framework. This section discusses two relevant NN architectures to our work, namely Convolutional Neural Networks (CNNs) and Long Short Term Memory Networks (LSTMs). Furthermore, we describe the attention mechanism, an effective technique to estimate components' importance.

2.3.1 Convolutional neural network

Convolutional Neural Networks (CNNs) could be the most widely used NN architecture. They continue to be an omnipresent component in deep learning, especially in computer vision and RL. This wide usage may be attributed to the ubiquity of data that exhibits spatial dependencies, which can be captured by CNNs. In RL, the usage of CNNs in the infamous DQN paper [50] revolutionized the field, as extracting features from raw pixels is not warranted anymore. Instead, CNNs can gradually capture these features, starting from basic shapes like triangles and squares to more domain-based features, as the depth of the network increases. Thus, CNNs are almost universally used in all deep RL applications and provide learning bias in the network architecture by imputing it with relevant invariances.

A CNN [40] typically consists of three types of layers: convolutional layers, pooling layers, and fully connected layers. A convolutional layer, the core component of a CNN, contains n filters (called kernels) that convolve over the input and produce n maps. The weights of those filters are learned using backpropagation [66]. Sharing the weights of those filters across all the input patches serves two purposes. First, the number of parameters is drastically reduced. Second, the local spatial features are effectively extracted because convolution is translation invariant (i.e., yields the same results regardless of the input's shift in space). The output of a convolutional layer is usually followed by a nonlinear activation function, which helps in approximating any (measurable) function.

A pooling layer downsamples the input and applies a nonlinear function to input patches, which makes CNNs more robust to small shifts in the input. Finally, fully connected layers are then used to estimate the target. For instance, if a CNN were used to classify images, the output of the fully connected layers would be the probability of each class. In RL, the output is usually an estimate of the value function V, the policy π, or both. Figure 2.9 shows a simple CNN, where only one convolutional layer and one pooling layer are used for simplicity.

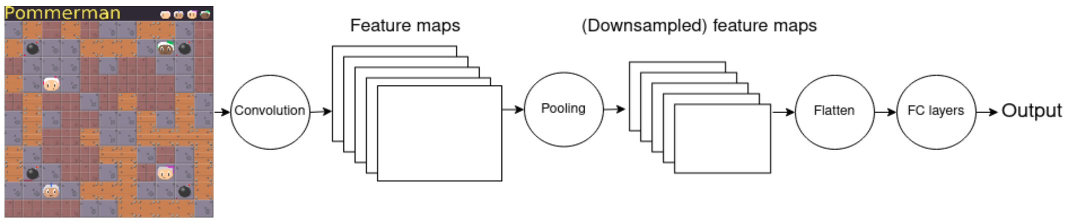


Figure 2.9: A simple convolutional neural network that consists of a convolutional layer, a pooling layer, and several fully connected (FC) layers.

2.3.2 Long short term memory network

Long Short Term Memory Networks (LSTMs) [32] are recurrent models capable of learning dependency in sequential data. Similar to CNNs, LSTMs are typically leveraged to learn the policy or the value functions, with the benefit of taking into account the dependency between the intermediate representation of the states. An LSTM (see Figure 2.10) encapsulates the history by two vectors: the cell vector and the hidden vector. These two vectors are continuously updated as new inputs are fed to condense the entire history. To do so, an LSTM employs three gates, where a gate controls the amount of passed information by applying the sigmoid function σ [56] on the input. The first gate is the forget gate, which selects the variables from the cell vector that should be updated based on the input. Next, an input gate is used to update the cell vector based on the input. Finally, an output gate is employed to compute the output, often followed by fully connected layers with an activation function that depends on the purpose of the LSTM (e.g., a softmax function if a probability distribution is being modeled).

Since the input is only fed unidirectionally, namely from the past to the present, future observations will not be used. However, humans do use the future to infer the past: we change our interpretation of speech after hearing new words [28]. A bidirectional LSTM (see Figure 2.11) offers this capability by feeding the input bidirectionally (i.e., from the past to the present and from the future to the present), which preserves the information from both the past and the future. The output vectors of the forward and backward passes are then concatenated.

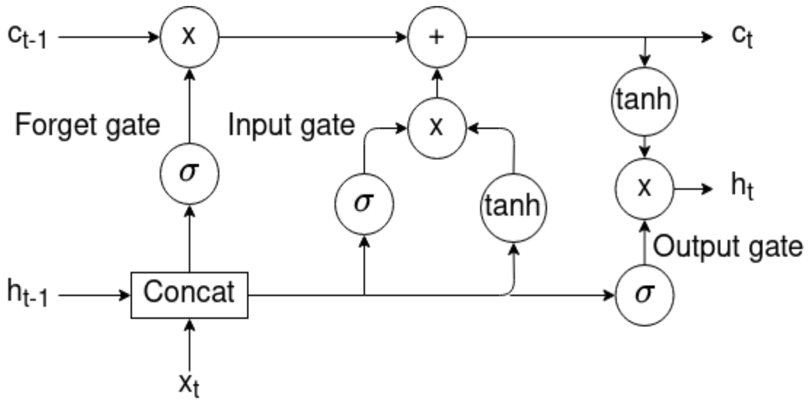


Figure 2.10: The architecture of a long short term memory network. Image reproduced from [90].

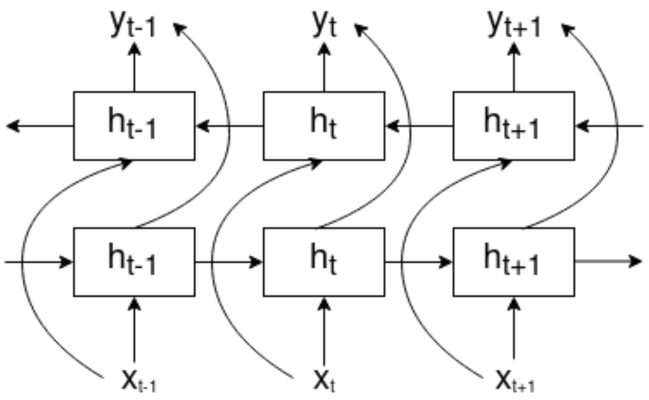


Figure 2.11: The architecture of a bidirectional LSTM. Image reproduced from [82].

2.3.3 Attention mechanism

Although LSTMs were built to handle long sequential data, the use of only two vectors to represent an entire sequence (that may be hundreds of items long) might deter LSTMs from accurately representing long sequences. Instead, all the intermediate hidden states could be used to predict the outputs. The attention mechanism [13] (see figure 2.12) was invented to address this issue by assigning different weights to the inputs to predict the outputs. As its name alludes to, the attention mechanism allows putting more attention on the more relevant inputs. The canonical example of applying the attention mechanism is machine translation. Specifically, we need to pay more attention to the words that affect the output the most. For instance, if the original sentence in Dutch was "hij is voetbal aan het spelen", which means he is playing football, then the model should focus mostly on the word "spelen" (which means playing) when it predicts the word "playing" in English. The attention mechanism can be generalized to any case where the output is affected by the main vector y and other vectors $x_1, x_2, .., x_m$.

In essence, the attention mechanism consists of four steps:

- Computing the alignment score: to quantify the attention value, an alignment score is computed between y and each x_i. Numerous methods were proposed in the literature for computing this score. For instance, the dot product is one of the most common ways to compute the attention value [85], as follows:

  $$\text{score}(y, x_i) = (W_s\mathbf{y})^\intercal W_t x_i$$

  where W_s and W_t are learnable weight matrices.

- Normalizing the alignment scores: the alignment scores are absolute, and should these scores be left unnormalized, the magnitude of the output will change based on the number of vectors. Thus, the softmax function [56] is used to normalize the scores and turn them into a categorical probability distribution:

  $$\alpha_i = \frac{e^{\text{score}(y,x_i)}}{\sum_{k=1}^{m} e^{\text{score}(y,x_k)}}$$

- Computing the output vector: after computing the attention (normalized alignments), the output vector is computed as a linear combination of the inputs weighted by their respective attention, as follows [85]:

  $$o = \sum_{k=1}^{m} \alpha_k W_c x_k$$

  where W_c is a learnable embedding matrix

Multi-head soft attention

To allow the model to attend the input from different representation subspaces [78] and stabilise the learning process [79], the same computation is repeated with different weight matrices W_s^h, W_t^h, W_c^h [85]. Afterward, the resulting vectors are either averaged or concatenated; we opt for the former for our purposes. The output vector is then computed as follows (where H is the number of attention heads):

$$o = \frac{1}{H} \sum_{i=1}^{H} \sum_{j=1}^{m} \alpha_{i,j} W_c^i x_j$$

The aforementioned (soft) attention mechanism assigns non-zero attention to all the inputs [44]. However, we might want to ignore some components, which could be achieved by setting a threshold upon which only the relevant components are selected. However, we instead learn to select only a subset of these components as opposed to soft attention, where we select all the components with different weights.

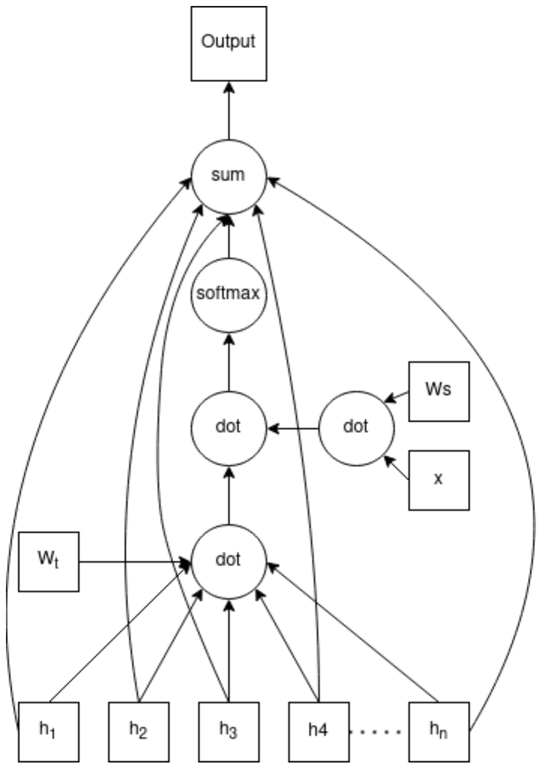


Figure 2.12: The attention mechanism [21].

Hard attention

One way to implement hard attention is to assign a Bernoulli random variable for each input and estimate the success probability p using a neural network. However, this implementation would result in a non-differentiable model, as backpropagation cannot be used to compute the derivative of a stochastic node. Still, we reduced our problem down to sampling from a Bernoulli distribution (i.e., a categorical distribution with two outcomes), which could be tackled by the Gumbel-softmax trick [35]. The Gumbel-softmax trick separates the stochastic part from the deterministic part by computing the class probabilities first and then adding noise drawn from the Gumbel distribution. This reparameterization trick allows the gradient to flow through the neural network, making the model fully differentiable and, thereby, trainable using backpropagation.

Figure 2.13 illustrates the difference between soft and hard attention, where soft attention selects all the pixels with a varying degree of importance, whereas hard attention drops most of the pixels and only picks the most relevant ones. The combination of soft and hard attention is quite powerful and of great importance for our work as it allows the models to choose only relevant components and then estimate their influence.

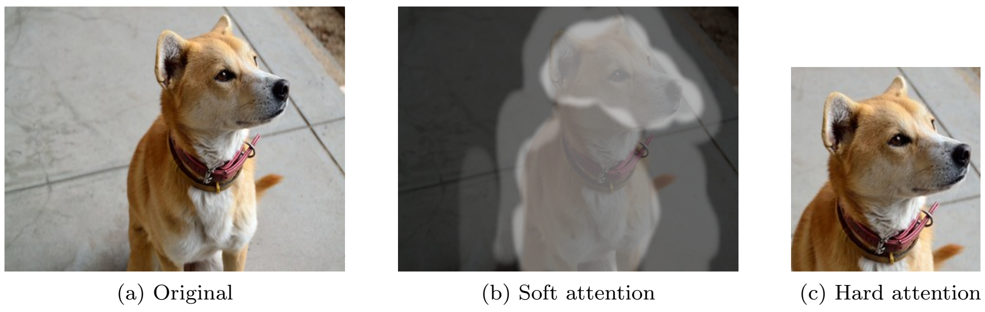

(a) Original (b) Soft attention (c) Hard attention

Figure 2.13: Soft versus hard attention, where hard attention picks a subset of pixels. Original image by Giovanna Durgoni [3].

Chapter 3

Method

In this chapter, we discuss our approach for integrating opponent models with SM-MCTS. We start by analyzing Pommerman, a computer game that gives rise to several challenges. Then, we motivate and demonstrate the benefits of our approach when applied to several Pommerman configurations. Next, we present our neural network model that can learn to estimate the importance of opponents and the action probabilities thereof to sort actions for PW. Finally, we introduce our main contribution that combines SM-MCTS with opponent models: Attention-Guided Simultaneous-Move Monte Carlo Tree Search (AG-SM-MCTS).

3.1 Motivation

The previously discussed planning techniques can be used as-is for many sequential tasks. However, the action space could be too large to explore if the actions are simultaneous. For instance, if the number of valid actions for an agent is ten on average in a game that contains three agents, then even exploring three levels would be very computationally expensive. In particular, if we set the number of levels to 3, we would need to explore $1000 + 1000^2 + 1000^3 = 1,001,001,000$. We hypothesize that in many cases, most of these actions are irrelevant for the agent, and, sometimes, some opponents could be completely omitted if they do not significantly affect the agent.

To illustrate this concept further, we use the game named Pommerman as a testbed. Pommerman [63] is a deterministic, multi-player, general-sum, and simultaneous-move game that is inspired by the Atari game Bomberman. The Pommerman environment is an 11×11 board that contains four agents that can choose between six actions: moving left, up, right, or down, staying put, and planting a bomb. A Pommerman game can last for up to 800 timesteps. Each board's cell is either a passage, a wooden wall, or a rigid wall.

A passage is an area the agents could move to, whereas a wooden wall can only be passed across after bombing it. In contrast, a rigid wall is impassable and cannot be demolished. Each agent can plant a bomb that explodes after ten timesteps, which kills nearby agents and destroys nearby wooden walls. The agents are also vulnerable to their own planted bombs and can be killed by them. When a wooden wall is destroyed, three types of power-ups can be revealed: increasing the number of the bombs the agent possesses, raising the blast range of bombs, and acquiring the ability to kick bombs.

When the game ends, the last-standing agent receives a reward of 1, whereas dead agents receive a reward of -1. If the game ends in a tie (i.e., all the agents are dead or more than one agent is still alive after 800 timesteps), all the agents receive a reward of -1. Without loss of generality, we assume that we always control the top-left agent because the board can be flipped otherwise. Figure 3.1 depicts a Pommerman board, where we, from now on, denote the bottom-left agent as the first opponent, the bottom-right agent as the second opponent, and the top-right agent as the third opponent.

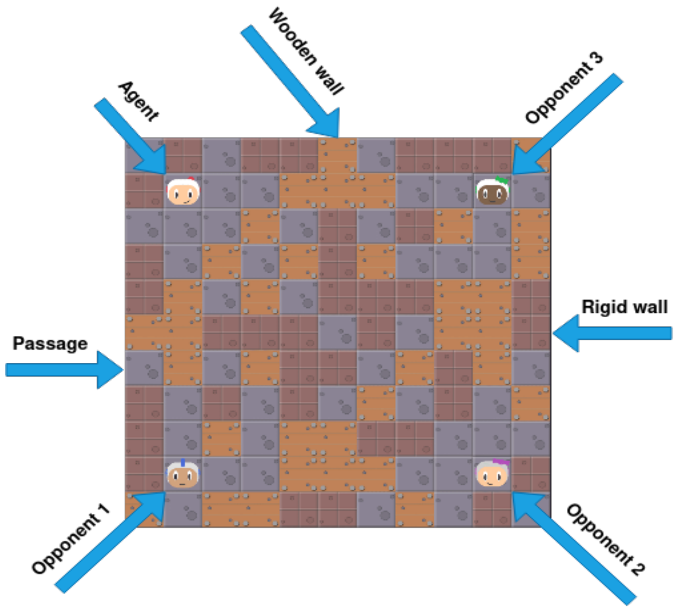


Figure 3.1: Pommerman board.

Although the Pommerman environment might seem simple, it poses a wide variety of challenges, including but not limited to [26]:

- Combinatorial action space: because the action space is large, even considering one lookahead step is computationally expensive.
- Sparse and deceptive rewards: the agent only receives a reward when the game ends, which may hamper learning, as it is more challenging for the agent to assess its actions based only on the final reward. Furthermore, the rewards are deceptive. In particular, the agent could win by just letting its opponents commit suicide instead of learning how to play.
- Hard exploration: the agent is encouraged not to explore the environment because successfully planting a bomb requires various skills, such as evading the flames and cornering opponents. Additionally, it is not unlikely for the agent to accidentally commit suicide after planting a bomb.

The second and third problems could be alleviated by shaping the reward, whereas the first problem is our main motivation for this research. To motivate our solution, we need to recap how SM-MCTS operates. Figure 3.2 demonstrates how the branching factor could be reduced in two ways: by reducing actions and by (soft) pruning players.

Reduction of actions could be achieved using PW. However, we still need a way to order opponents' actions. Estimating opponents' action probabilities and sorting them by their likelihood is one way to learn the move ordering functions from scratch. Thus, we use opponent models to shape the agent's features and order opponents' actions.

Pruning opponents is more complicated, as classical search tree reduction algorithms, such as FPU and PW, were not invented for entirely omitting an opponent. One method to perform this type of reduction is estimating opponents' influence on the agent and guide the tree search accordingly. In particular, we want to direct more attention to the opponents that matter the most and reduce the branching factor afterward. One approach to implement this idea is to make the widening exponent, α, of PW state-dependent based on the estimated influence. After that, opponents who affect the agent more will have a higher α than opponents who are less significant to the agent.

We investigate three cases to demonstrate how the second type of reduction could prove beneficial. Figure 3.3 depicts a Pommerman setting where the agent does not need to plan over opponents' actions, as they are too far away to consider. In contrast, Figure 3.4 shows a case where only one opponent affects the agent remarkably, necessitating planning over its actions. Finally, Figure 3.5 illustrates a situation where both opponents could kill the agent, meriting exploring their actions.

Algorithms 4 and 5 demonstrates how SM-MCTS could be modified to accommodate opponents' influence where α, which controls the growth of the tree, now depends on the state. These two algorithms work when a model that can estimate opponents' influence and action probabilities is provided. However, this is not the case for many games. For instance, the influence of the opponents on the agent in Pommerman is not trivial to estimate, as it could be a complex combination of many functions, such as the distance and the number of rigid walls between the agent and the opponent.

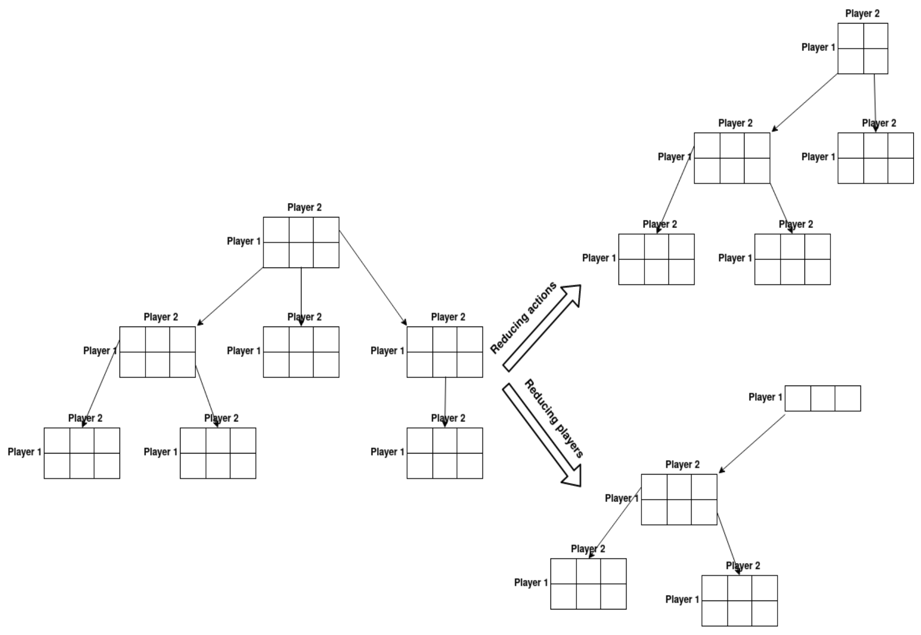


Figure 3.2: SM-MCTS types of reduction: opponents reduction and actions reduction.

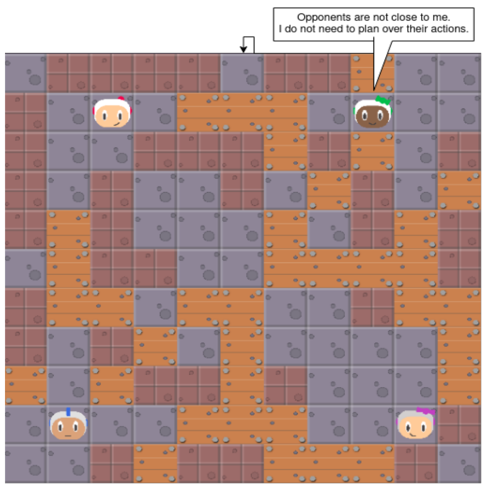

Opponents are not close to me.
I do not need to plan over their actions.

Figure 3.3: A Pommerman setting the agent does not need to explore opponents' actions.

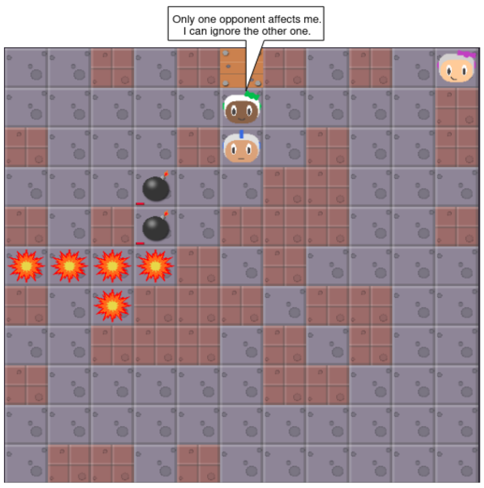


Figure 3.4: A Pommerman setting where the agent should consider the closer opponent.

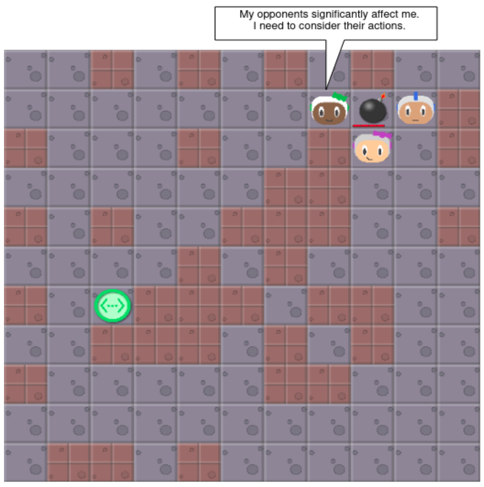

My opponents significantly affect me.
I need to consider their actions.

Figure 3.5: A Pommerman setting where the agent has to plan over all opponents' actions.

Algorithm 4 State-dependant Progressive Widening

```
1: procedure SDPW($S, \alpha$, action_probabilities)
2:     $n$ = number_of_visits(S)
3:     $k = \lceil Cn^{\alpha} \rceil$
4:     valid_actions = valid_actions(S)
5:     sorted_actions = sort(valid_actions, action_probabilities)
6:     selected_actions = select_first(sorted_actions, $k$)
7:     ucbs = UCB(S, selected_actions)
8:     $a$ = argmax(ucbs)
9:     return $a$
```

Algorithm 5 State-dependant Simultaneous-Move Monte Carlo Tree Search

```
1: procedure SDSMMCTS($S$)
2:     if is_terminal($S$) then
3:         return rewards(S)
4:     actions = []
5:     $\alpha$, action_probabilities = estimate_influence(S)
6:     for $i \in$ [1..number_of_agents] do
7:         $a$ = SDPW($S, \alpha_i$, (action_probabilities)$_i$)
8:         actions.append($a$)
9:     next_state = step($S$, actions)
10:    if actions $\notin$ children($S$) then
11:        $v$ = rollout(next_state)
12:        add_child($S$, actions, $v$)
13:    else
14:        $v$ = SDSMMCTS(next_state)
15:    backpropagate($S, v$)
16:    return $v$
```

3.2 Proposed model

To estimate opponents' action probabilities and influence on the agent, We propose the neural network model depicted in Figure 3.7, which consists of three modules. First, the features extractor (a CNN) that extracts key features from raw observations. Second, an agent module that models the action probabilities and the value function of the agent. Third, an opponent module that predicts the action probabilities of each opponent. Furthermore, the soft and hard attention block (see Figure 3.6), which is similar to [44], acts as a bridge between the agent module and the opponent module and serves two purposes. First, the hard attention filters a subset of opponents via a bidirectional LSTM followed by a two-neuron Gumbel-softmax layer that determines whether an opponent should be considered or not. Concretely, the latent variables of the agent and each opponent are concatenated and then fed into a bidirectional LSTM [44]. Second, the agent's latent variables and the selected opponents' latent variables are fed into a multi-head soft attention layer, as explained in Chapter 2, to estimate their influence on the agent.

Specifically, the main vector will be the agent's latent vector $y = h_{\text{agent}}$ and the input vectors will be the agent's latent vector and the opponents' latent vector $x_1 = h_{\text{agent}}, x_2 = h_{\text{opponent\_1}}, .., x_{m+1} = h_{\text{opponent\_m}}$. Note that the agent latent is also considered for computing the alignment score, as opponent actions' could be of more importance than the agent's actions at some states. Similar to [30], the loss function is the sum of the A3C loss and weighted opponent losses, as follows (where m is the number of opponents):

$$L = L_{\text{policy}} + L_{\text{value}} + L_{\text{entropy}} + \frac{1}{m}\sum_{i=1}^{m} \lambda_i L_i$$

The loss function for opponent i, L_i, is the cross-entropy loss [45] as predicting the action probabilities of an opponent is simply a classification problem. Note that assuming that the agent can observe opponents' actions during training, under the framework of CTDE, simplified the problem to a classification problem; otherwise, we would have to estimate those actions from local observations only.

To summarize, the total loss function is then:

$$L = L_{\text{policy}} + L_{\text{value}} + L_{\text{entropy}} + \sum_{i=1}^{m} \lambda_i (\frac{1}{n} \sum_{j=1}^{n} a_i^j \log(\hat{a}_i^j))$$

$$= -\lambda_{\text{policy}} \mathbb{E}_{s\sim\pi}[R_{1:\infty}] + \lambda_{\text{value}} \mathbb{E}_{s\sim\pi}[(R_{t:t+n} + \gamma^n V^\pi(s_{t+n+1}) - V^\pi(s_t))^2] - \lambda_{\text{entropy}} \mathbb{E}_{s\sim\pi}[H(\pi(s)]$$

$$+ \sum_{i=1}^{m} \lambda_i (\frac{1}{n} \sum_{j=1}^{n} a_i^j \log(\hat{a}_i^j)])$$

where n is the number of actions, a_i^j is a one-hot encoded vector that represents the action the i-th opponent took, $\hat{a}_i^j$ is the predicted action probabilities for the i-th opponent, and λ_i is the weight of the i-th opponent.

Note that if the opponent's attention is zero (i.e., the hard attention does not select that opponent), only the most likely action will be explored. To incorporate our proposed model into Algorithm 5, we only need to call our model in the function estimate_influence(S). We call our approach Attention-Guided Simultaneous-Move Monte Carlo Tree Search (AG-SM-MCTS), as the attention values guide the search tree to spend more search time on opponents that affect the agent more. To summarize, Figure 3.8 illustrates the full framework, where the learning part is trained offline to be deployed later by the planning part.

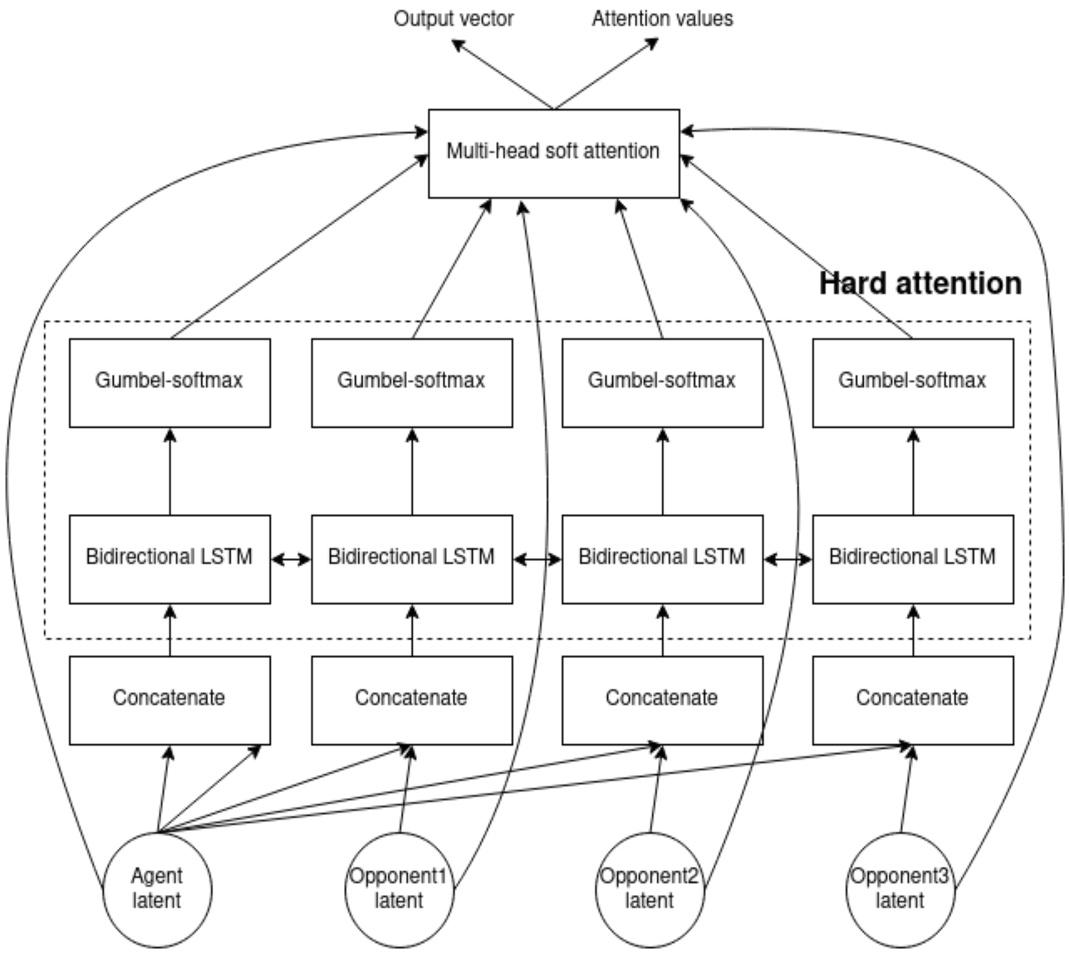


Figure 3.6: The soft hard attention mechanism [44].

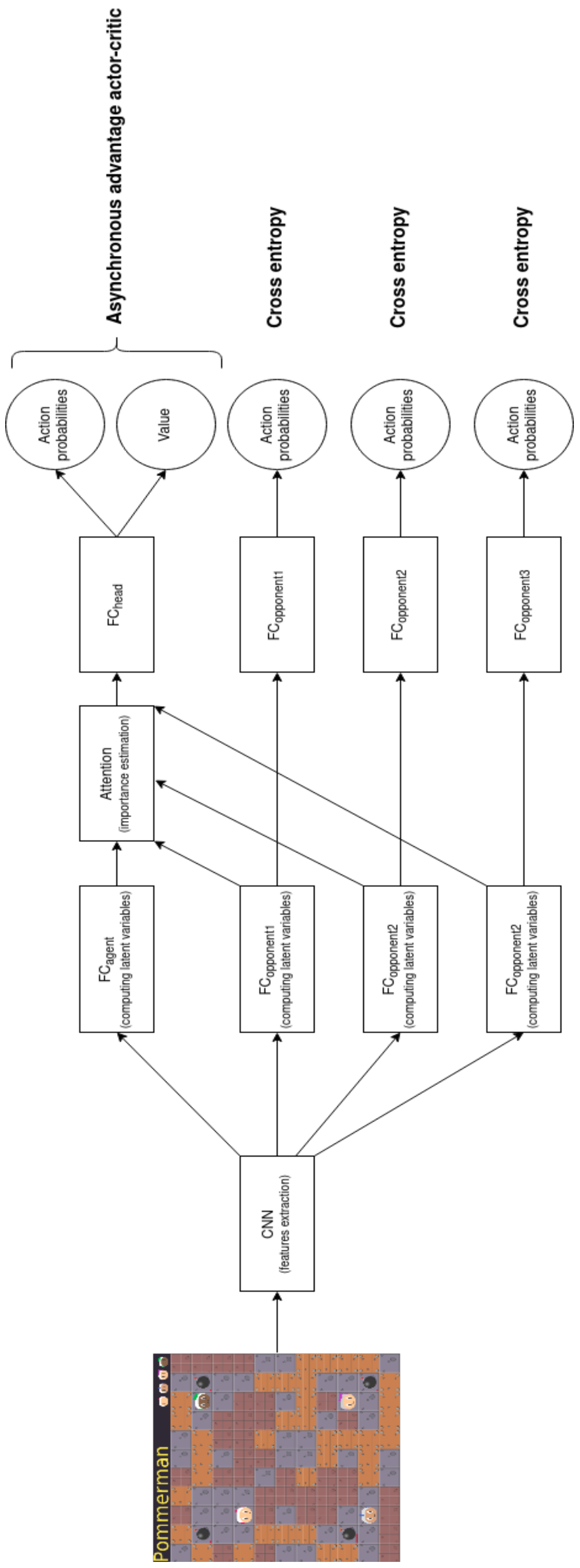

Asynchronous advantage actor-critic
Cross entropy
Cross entropy
Cross entropy
Action probabilities
Value
Action probabilities
Action probabilities
Action probabilities
FC_{head}
$FC_{opponent1}$
$FC_{opponent2}$
$FC_{opponent3}$
Attention (importance estimation)
FC_{agent} (computing latent variables)
$FC_{opponent1}$ (computing latent variables)
$FC_{opponent2}$ (computing latent variables)
$FC_{opponent2}$ (computing latent variables)
CNN (features extraction)
Pommerman

Figure 3.7: The proposed neural network model.

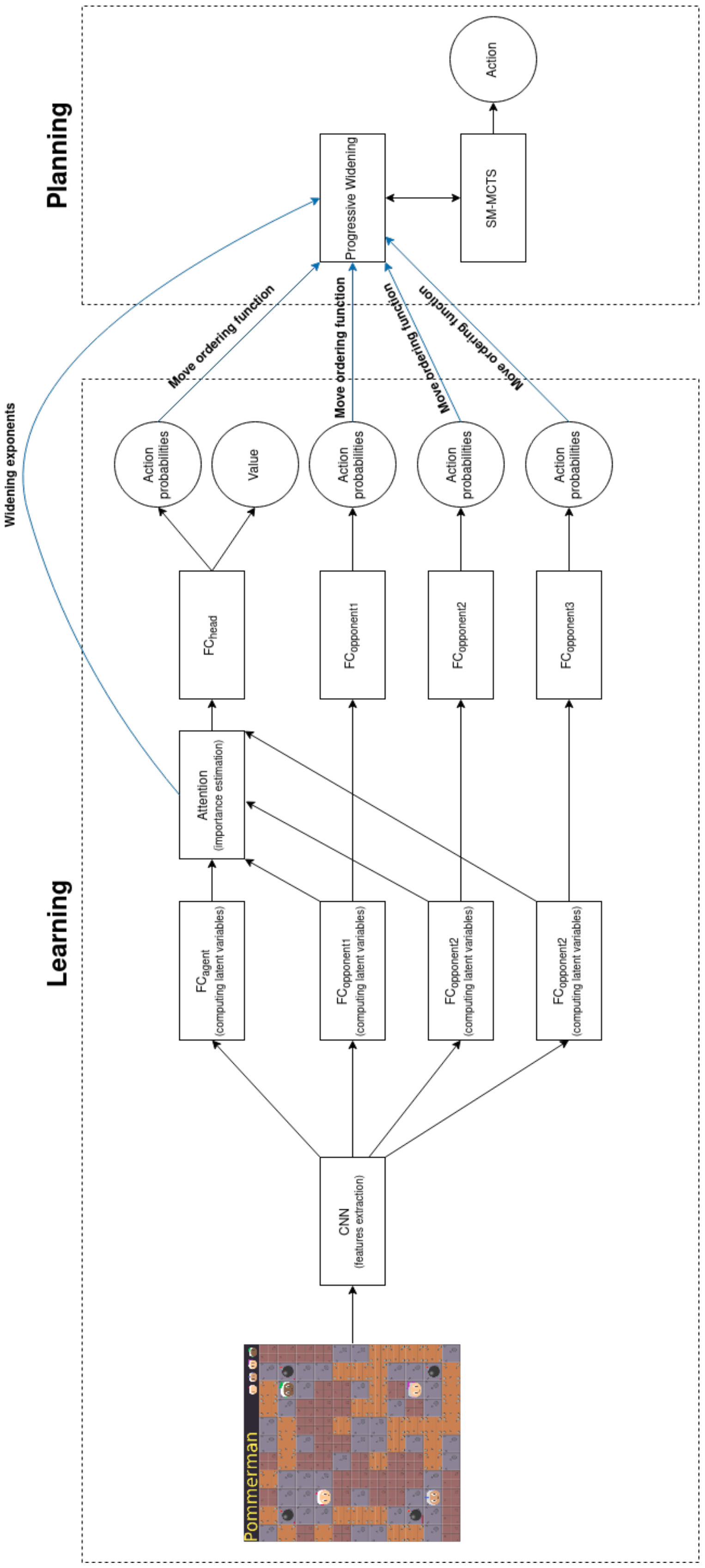


Figure 3.8: The proposed learning and planning framework.

Chapter 4

Experiments

In this chapter, we elaborate on the implementation details of our approach and experimentally compare it to SM-MCTS. We start by describing the setup of the experiments, including the used software and hardware. Next, we investigate the performance of the learning module, detailing how the reward was shaped and analyzing the trained agent's performance. Furthermore, we analyze the trained opponent models, showing how well they were learned. Finally, we compare the performance of our approach to SM-MCTS, demonstrating that our approach achieves a significant performance increase.

4.1 Setup

To simulate experiments, we used Cartesius [2], the Dutch national supercomputer, and utilized 48 CPUs of type "Intel Xeon Processor E5-2697A v4" [5]. Additionally, since the Pommerman Python implementation is quite slow, we modified an open-source Pommerman environment [48] where Cython [15] was used to accelerate simulations. The neural network model was implemented using PyTorch [60]. Training and testing were carried out against three opponents that use a stochastic rule-based policy provided by the Pommerman package. These opponents exhibit stochastic behavior and use Dijkstra's algorithm at each timestep to find the shortest path [30]. They are also skilled in evading bombs, making them difficult opponents to model and play against.

4.2 Analysis of the learning module

To show that our learning module provides the planning module with an accurate prior, we analyzed the following questions:

- How well does our proposed model learn to play against opponents?
- How well does our proposed model learn to model opponents?

Training the proposed model as-is is problematic for two reasons. First, rewards in Pommerman are sparse, which makes learning more challenging. Second, feeding the raw observation will force the agent to understand raw pixels, adding a computer vision challenge that is irrelevant for our work. In the following sections, we discuss how the reward was reshaped and how the input was encoded to alleviate these two hurdles before examining the previous questions. Additionally, we describe the architecture of the proposed models, providing the used hyperparameters of each component and how they were selected.

4.2.1 Reward shaping

Training an agent with the sparse and deceptive rewards of Pommerman could potentially lead to an agent that does nothing. To win a game, the agent needs to demolish wooden cells, plant a bomb near each opponent, and evade the flames produced therefrom to obtain a positive reward. Thus, the agent might not explore the environment and instead wait for its opponents' demise. Additionally, the agent might need to acquire an arsenal of complex and varied skills, including chasing opponents, planting a bomb, evading flames and bombs, and cornering opponents. Learning to handle all these situations from scratch merits shaping the reward to expedite learning. Table 4.1 shows the types and values of the reward shaping approaches that were used to encourage the agent to explore and overcome the sparse reward limitations. Most of these reward shaping methods are inspired by [25, 6]. The agent receives a mobility reward (i.e., a novelty-based reward) if it moves to a new cell that has not been visited in the last 90 timesteps. While some of these rewards could be dropped, such as the illegal moves negative reward, they still help accelerate learning.

Reward type	Reward value
Killing an opponent	0.5
Catching an opponent	0.001
Not avoiding flames	-0.01
Mobility (novelty-based reward)	0.005
Illegal moves (e.g., moving to a rigid wall)	-0.03
Picking a powerup	0.02
Planting a bomb near a wooden wall	0.001

Table 4.1: Types of rewards.

4.2.2 Input semantic encoding

Instead of forcing the agent to learn to see, we chose to represent the board semantically to help the neural network in extracting useful features. Specifically, we adopted how [64] represented the board by 17 feature maps. Table 4.2 explicates how each map is computed, where map cells' value are set to 0 by default.

Map	How it is computed
1	The blast strength of each bomb, placed at the corresponding bomb position
2	The remaining life of each bomb, placed at the corresponding bomb position
3	Setting the cell that correspond to the agent's position to 1
4	All cells are set to the number of the bombs the agent possess
5	All cells are set to the blast radius of agent's bombs
6	All cells are set to 1 if the agent can kick bombs
7	Setting the cell that correspond to the first opponent's position to 1
8	Setting the cell that correspond to the second opponent's position to 1
9	Setting the cell that correspond to the third opponent's position to 1
10	Setting the cells that correspond to passages to 1
11	Setting the cells that correspond to wooden walls to 1
12	Setting the cells that correspond to rigid walls to 1
13	Setting the cells that correspond to flames to 1
14	Setting the cells that correspond to extra-bomb power-ups to 1
15	Setting the cells that correspond to increase-blast-strength power-ups to 1
16	Setting the cells that correspond to kicking-ability power-ups to 1
17	All cells are set to $\frac{\text{current\_step}}{\text{total\_number\_of\_steps}}$

Table 4.2: Input feature maps.

4.2.3 Network architecture

We next describe the architecture of the proposed model (shown in Figure 3.7). Similar to [30], the CNN module consists of four convolutional layers, with 32 filters of size 3×3 and a stride of 1 for each, whereas fully connected layers comprise of 128 hidden units each. The embedding dimension of the attention mechanism was set to 128 (i.e., all the weight matrices W_s^i, W_t^i, W_c^i were of dimension 128×128). Similar to [30], we used the ELU (Exponential Linear Unit) activation function [56] for all the layers, except the output. Additionally, we set the entropy weight to 0.01, the value loss weight to 0.5, the policy loss weight to 1, and the discount factor to 0.95. Moreover, and similar to [30], the Adam optimizer [36] was used with learning_rate = 0.005, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and a weight decay of 10^{-5}. Furthermore, we used five soft attention heads, and set the dimension of the hidden state of the bi-LSTM to 128. The opponent loss weight parameters were all set to 0.05. Finally, gradients' norm were clipped to 0.8 [59].

4.2.4 Analyzing the agent module

To analyze the performance of the trained agent against opponents, we plotted the accumulated reward over episodes. If the curve exhibits an upward trend, then the agent is learning to increase its reward. Although the agent's policy loss and value loss could help debug the model, they provide no insights to understand model performance as they do not necessarily decrease during training. This possible rise can be attributed to the increase in the number of steps per episode, as the agent learns to stay alive longer as the training progresses. Figure 4.1 illustrates the moving average of the accumulated reward of our approach compared to A3C, averaged over five runs (shown by the shaded area). Our approach outperforms A3C by a significant margin, showing that incorporating opponent models and the attention mechanism expedites learning and yields a more skilled agent. This result matches what was found in [30], though our model uses the attention mechanism instead of multiplying the latent vectors.

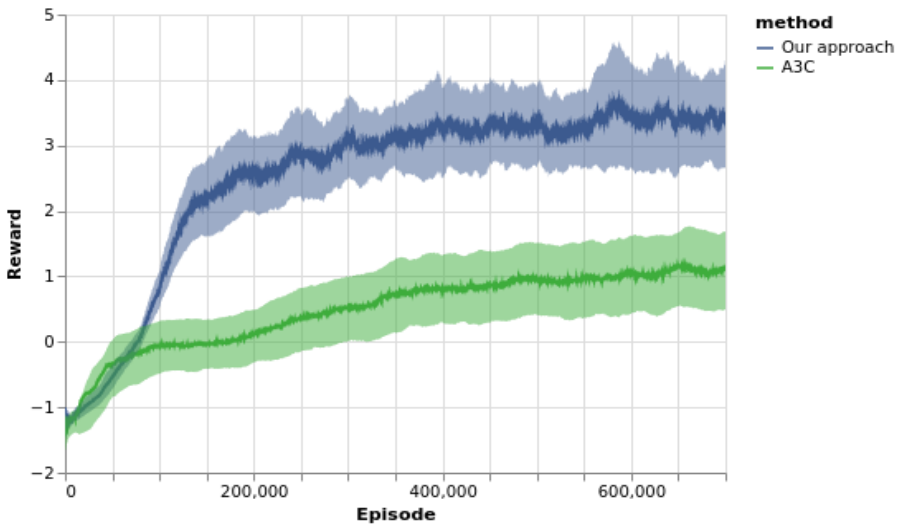


Figure 4.1: The moving average of the accumulated reward over five runs.

The attention values are not trivial to interpret, as they may involve a complex mix of different factors, including the distance to a given opponent. However, we drew two insights by investigating the attention values during testing. First, although the attention values should not exactly resemble a distance function, we still expect a relationship between the attention values and the distances between the agent and the opponents. In particular, we expect opponents that are further away to be less relevant for an agent's decision. To examine this possibility, we computed Spearman's correlation coefficient [52] between the attention values and opponents' distances over time. Spearman's correlation [52] measures the strength of the monotonic relationship between two variables [7].

Thus, we expect a negative Spearman's correlation coefficient for all opponents (i.e., when the distance increases, the attention value decreases). Indeed, Table 4.3 shows the computed Spearman's correlations over 10 episodes, where all the Spearman's correlations were significant (i.e., $p < 0.05$). Noticeably, the first opponent's Spearman's correlation is considerably larger than the other two opponents, which could be due to its quick interaction with the agent. In contrast, the second opponent rarely interacts with the agent on average, potentially explaining why its corresponding Spearman's correlation is the smallest.

Opponent	Spearman's correlation	p
Bottom-left opponent	-0.3646	2.319e-05
Bottom-right opponent	-0.1861	2.8839e-10
Top-right opponent	-0.2206	2.6770e-15

Table 4.3: Spearman's correlation coefficients between the Manhattan distances between the agent and the opponents, and the attention values.

To obtain more insights, we investigated how the attention value evolves over time, first looking at two specific episodes and then looking at the average of 30 episodes. Figure 4.2 shows the attention values changes of the agent, the first opponent, the second opponent, and the third opponent. On average, self-attention is the largest, followed by the first opponent and the third opponent. Indeed, we would expect the agent's actions to affect the agent the most, followed by the opponent with whom it interacts the most (i.e., the first opponent). In contrast, and as explained before, the agent rarely interacts with the second opponent, which may justify why its average attention value is the smallest. Additionally, one can notice that the attention values in the two shown episodes differ significantly, indicating that the attention values depend on the situation.

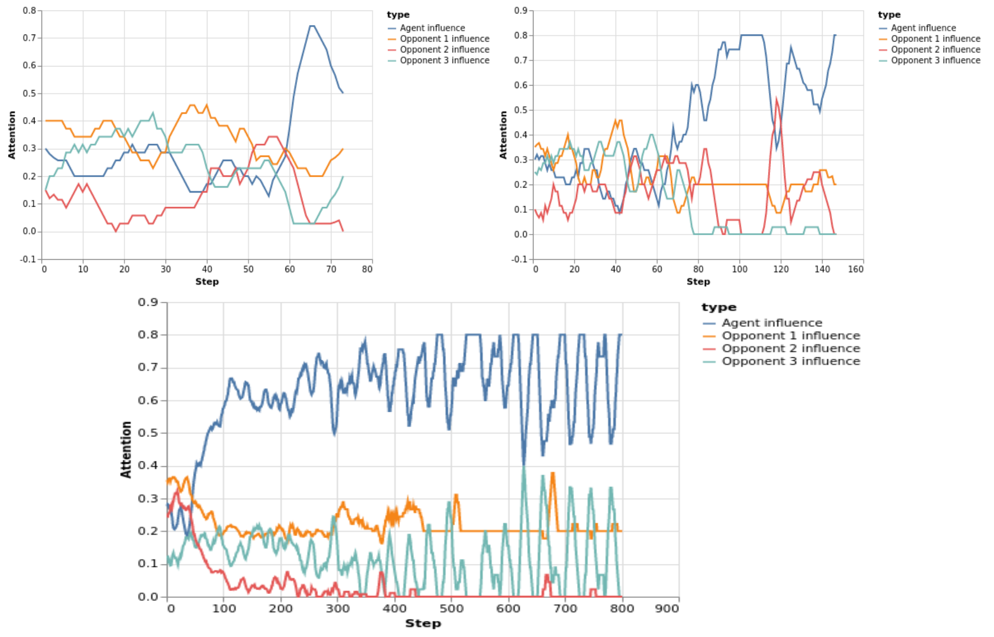


Figure 4.2: Top: two example episodes illustrating how the attention values evolve over time. Bottom: the evolution of attention values on average for 30 episodes.

4.2.5 Analyzing the opponent module

To study opponent models' performance, we plotted opponents' cross-entropy loss over time [56], as opponent modeling is a classification task in our proposed model. If the cross-entropy loss declines with the number of episodes and reaches a plateau, then the model learned to model the categorical distribution of each opponent. Figure 4.3 matches our expectations, where the opponent loss starts from nearly 0.2 and quickly drops to approximately 0.13. The fluctuations the curve is showing are not due to overfitting; rather, because the total loss could be decomposed to the individual opponent loss, adding those curves will show an oscillating behavior.

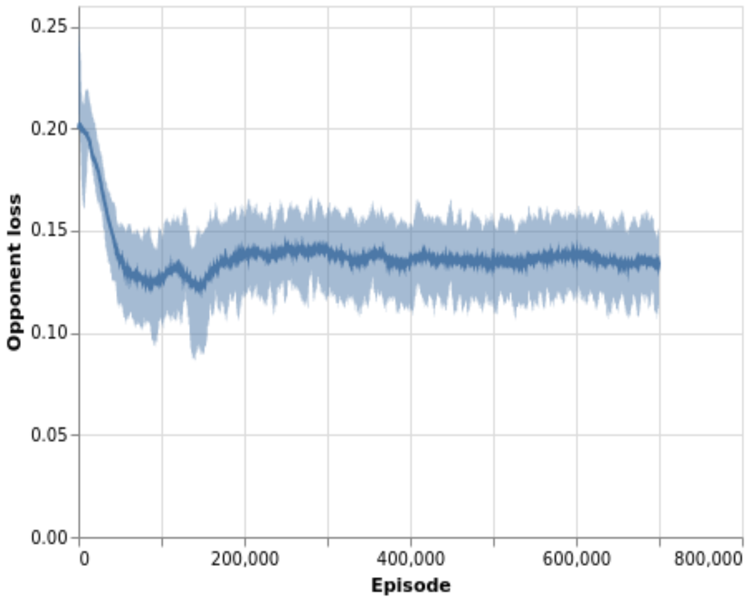
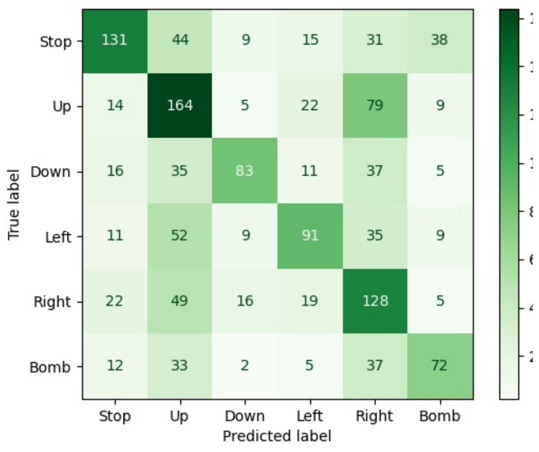


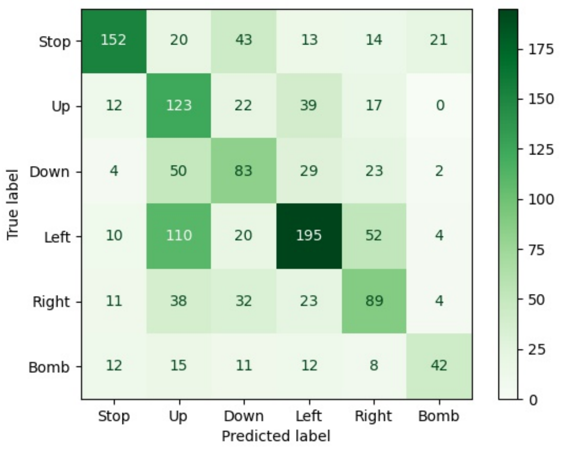
Figure 4.3: Opponent loss over episodes.

To understand how well the model learned to model opponents, we examined the confusion matrix of each opponent model. Figure 4.4 illustrates the confusion matrices of opponents. The model captured opponent behaviors reasonably well, although the stochasticity thereof thwarted the model from modeling them perfectly. To quantify these opponent models by a score, we computed the F1-score for each model. In particular, we used the weighted multi-class F1-score, as the data is heavily imbalanced. We found that $\mathrm{F}_1^{\text{opponent 1}} = 0.4974, \mathrm{F}_1^{\text{opponent 2}} = 0.5167, \mathrm{F}_1^{\text{opponent 3}} = 0.4937$.

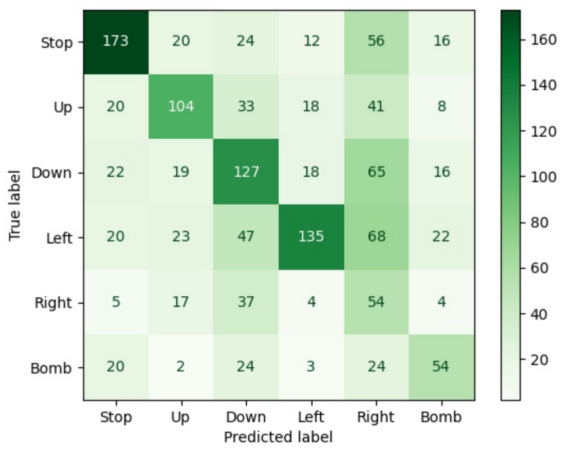
Furthermore, we visualized the importance of each cell in the input to explain why opponent models opt for particular actions. In particular, Gradient-weighted Class Activation Mapping (GradCAM) [69] was used to produces a heatmap that indicates the important regions that led to selecting a particular action. Figure 4.5, which was obtained in the first timestep, demonstrates that the cells that correspond to the agent's and opponents' position were the most important. These cells are essential because the CNN is shared between the agent module and the opponent module.



(a)

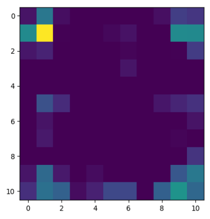


(b)

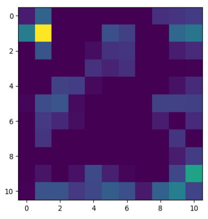


(c)

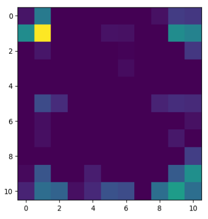
Figure 4.4: The confusion matrices of the opponents (left-right-bottom).



(a) First opponent's heatmap



(b) Second opponent's heatmap



(c) Third opponent's heatmap

Figure 4.5: Regions importance heatmaps, produced by applying GradCAM [69], of each opponent model (left to right).

4.3 Comparing AG-SM-MCTS to SM-MCTS

To compare SM-MCTS with AG-SM-MCTS, we need to select a metric (or a score) to base our comparison on. The win rate is one of the simplest and most straightforward score that could be used to evaluate and compare various approaches in our setting. The win rate is defined as the percentage of games the agent wins against its opponents out of all games. However, the tie rate should also be considered because even if two methods yield the same win rate, the one with the higher tie rate is better in general. We treat a tie as a quarter win (as only one player can win) and define score = $\frac{\text{wins}+0.25*\text{ties}}{n}$, where n is the number of games. We define a game as a fixed board, randomly initialized by a seed. First, we generated 200 different random seeds sampled uniformly from the range $[0, 1000000]$ to tune the hyperparameters of each method. Afterward, the best hyperparameters are used to play 500 games to approximate the win rate, the loss rate, and the tie rate for each approach. For simplicity, we denote SM-MCTS (ignore) as SM-MCTS when opponent actions are ignored, PW (random) as SM-MCTS endowed with PW with a random move ordering function, and PW (policy) as SM-MCTS enhanced with PW when actions are sorted according to the estimated opponent action probabilities.

Due to the computing power needed to run even one experiment, a budget of 750 SM-MCTS iterations per move was used. We used grid search [42] on 2 or 3 values for each hyperparameter to select the best hyperparameters. In particular, grid search was performed over the values $\{0.4, 0.8, \sqrt{2}\}$, $\{0.25, 0.5, \infty\}$, $\{0.5, 1, 2\}$, and $\{0.25, 0.65\}$ for the exploration parameter, the FPU value, the widening factor C, and the widening exponent α, respectively. We used the same hyperparameter value for each agent (e.g., $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4$). Table 4.4 shows the hyperparameters that yielded the largest score for each method.

Method	Exploration coefficient	FPU	C	α
SM-MCTS	0.8	0.5	Not used	Not used
SM-MCTS (ignore)	0.8	0.5	Not used	Not used
PW (random)	0.8	0.25	2	0.65
PW (policy)	0.4	0.25	1	0.25
AG-SM-MCTS	0.4	0.25	1	Dynamic

Table 4.4: Hyperparameters that yielded the largest score for each approach.

We hypothesize that being more cautious in Pommerman generally increases the score, as rule-based agents may commit suicide. Thus, evading their bombs diligently might be enough to lead to their death. A similar observation is also mentioned in [43]. Additionally, it is clear that larger C and α for PW (random) yielded better results, as sorting actions randomly may result in "bad" actions first, which could be mitigated by increasing the values of PW's hyperparameters. Furthermore, a progressive widening factor C of 1 resulted in a larger score for PW (policy) and AG-SM-MCTS. This raise could be attributed to having accurate opponent models such that the agent does not have to consider most of its opponents' actions. Should these opponent models be perfectly accurate, we would expect that a smaller widening factor, such as 0.5, would result in a higher score.

To conclude, Figure 4.6 depicts the performance of the discussed approaches where, similar to [14], whiskers indicate the 95% confidence intervals of the estimated score. We decomposed the score into its two components: the win rate (shown in blue) and the tie rate (shown in orange) for clarity. The confidence intervals were computed using the Wilson score interval method with continuity correction [54], as estimating the score could be perceived as estimating the probability of success p of a Bernoulli experiment. Furthermore, to test whether the difference in score between two specific approaches is statistically significant, we used the two-proportion z-test [8] for each pair of methods.

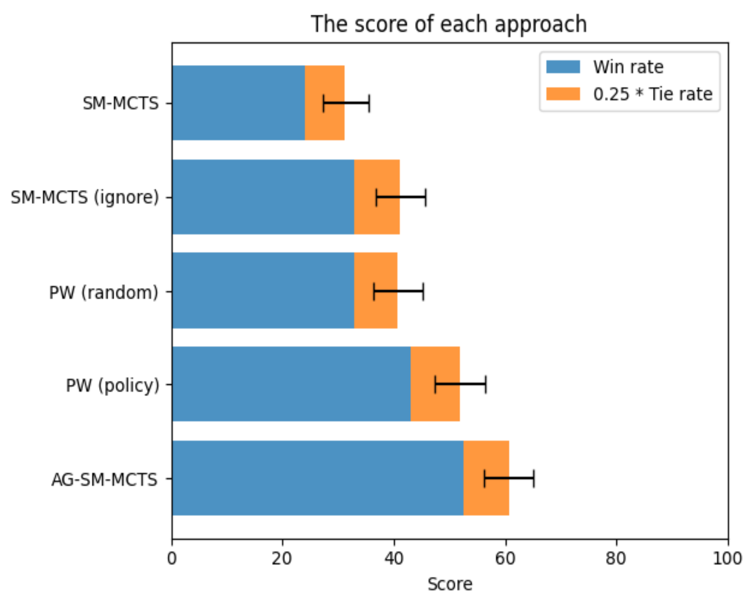


Figure 4.6: The score of each approach. All the pairwise differences in score were significant, except the difference between the scores of SM-MCTS (ignore) and PW (random).

The difference in score between SM-MCTS and SM-MCTS (ignore) is significant, potentially attributed to the small number of iterations compared to the large branching factor. Specifically, the agent may explore deeper levels if it completely ignores opponents' actions, which may be better than only exploring 2-3 levels while focusing equally on its opponents' moves. This reasoning is also supported by the attention changes during episodes shown earlier, where self-attention is larger than opponents' attention values on average. Although completely overlooking opponents' actions might be better than considering all of them, selecting some of these actions might improve performance.

Even with a random ordering function, SM-MCTS with PW was on par with SM-MCTS (ignore) in terms of performance. This proximity signifies that once SM-MCTS is provided with a better moves ranking order function, it might outperform SM-MCTS (ignore). The considerable surge in the score when PW is equipped with the policy function from the trained model indicates such possibility. Finally, modifying the PW's widening exponent to be state-dependent using our model (i.e., AG-SM-MCTS) boosted the score even further, outperforming SM-MCTS endowed with state-of-the-art enhancements, namely FPU and PW.

Chapter 5

Conclusion

In this thesis, we proposed a new planning algorithm, called Attention-Guided Simultaneous-Move Monte Carlo Tree Search (AG-SM-MCTS), which addresses the exponential growth of the search tree in multi-player general-sum simultaneous-move games. In particular, given opponent models, the agent can employ a strategy that plans over several future opponents' moves. If these opponent models are not provided, we proposed a neural network model that builds these models by playing against these opponents. These opponent models are then used to reduce the branching factor, as they provide a reasonable prior for the search tree.

To answer our main research question, we devised two subquestions:

- Assuming the availability of opponents' moves during learning, how can we learn opponent models?
- Given opponent models, whether they were provided or learned, how can they be exploited to search at a greater depth?

The first question was addressed using self-supervised learning by interacting with the environment and perceiving the actions of the opponents. Afterward, the innate models are updated to comply with the observed actions. In particular, our model leverages deep neural networks to represent opponents and condition the agent's decision on (a compact representation of) their estimated strategies. Because these models act as a prior to the planning algorithm, our approach could still work well even if these models do not completely match opponents' strategies.

The second question was tackled by altering the planning algorithm to incorporate the estimated opponents' importance. Specifically, the search tree is explored according to the relative importance of each opponent: the more an opponent affects the agent, the more its actions are explored. Those estimated influences were obtained using the attention mechanism, which serves as a general technique to prioritize different parts based on their influence.

In conclusion, this thesis investigated how opponent models could be utilized in multi-player general-sum simultaneous-move games to outperform classical sampling-based planning algorithms. Planning in such games is an underdeveloped research area, although they constitute many real-world scenarios, as most of them are simultaneous. We believe that this work serves as a starting step in this research direction, which, as hypothesized in [73], could pave the way to Artificial General Intelligence (AGI) in the future.

Future work

Nonetheless, numerous research directions could be pursued to expand on this work. In particular, we foresee five avenues to pursue: generalizing our method to non-stationary opponents' policies, tackling the policy-bias problem, performing more extensive hyperparameters tuning, investigating attention values further, and integrating our approach into generic meta-training frameworks.

First, our approach assumes that opponents' policies are stationary, as non-stationary policies (i.e., policies that change over time) cannot be estimated using simple classification. However, this limitation is difficult to tackle and might require major changes to our model, as the target (i.e., the policy of opponents) will continually change. Moreover, if the assumption of stationarity does not hold, a Markov game cannot be then reduced to an MDP, where single-agent RL can be used.

Second, SMMCTS's online value estimates could be refined if estimating opponents' value function is added to as an auxiliary task, as it might make these values more accurate. However, opponents are not aware of the other agents in our proposed model; thus, their estimated value function might not converge to their true value function, a problem known as policy-bias [10]. This issue could be addressed by making opponent models aware of the remaining agents, resulting in a new architecture akin to Graph Attention Networks [79].

Third, due to the computing power and time required for even conducting one experiment, which might take up to six days, no extensive hyperparameters tuning was performed. Furthermore, the weighting parameters of the opponent loss are empirically found to be quite sensitive [71]. Therefore, we expect a considerable improvement should these hyperparameters be tuned appropriately. Most importantly, we recognize that our experiments were limited and may not generalize as we used grid search on only 2 or 3 values for each hyperparameter. However, scaling more advanced hyperparameters optimization algorithms, such as Bayesian Optimization [24] and bandit-based approaches [41], to run on a cluster is a research question on its own that we cannot tackle in this work.

Fourth, the attention values, as described earlier, did not exactly match our intuition. Specifically, we expected those values to be more interpreted, as seen in many AI domains, such as natural language processing and computer vision, as shown in [80, 81, 89]. The difficulty of comprehending the meaning of these attention values could be ascribed to the complexity and changing dynamics of most multi-agent environments. Nevertheless, we leave investigating and interpreting these attention values further for future work.

Finally, our approach only uses SM-MCTS in test-time as opposed to Expert Iteration [72]. In the Expert Iteration framework, SM-MCTS is perceived as the expert, whereas the neural network is viewed as the apprentice, honing opponent models even further and eliminating the decoupling between opponent models (used during training) and SM-MCTS (used during testing). In essence, our model could be leveraged as the apprentice in meta-training frameworks such as Expert Iteration [12].

Bibliography

[1] 2 killed in driverless tesla car crash. https://www.nytimes.com/2021/04/18/business/tesla-fatal-crash-texas.html. Accessed: 2021-01-07. 4

[2] Cartesius: the dutch supercomputer. https://userinfo.surfsara.nl/systems/cartesius. Accessed: 2021-06-08. 29

[3] Dog image by giovanna durgoni. https://www.flickr.com/photos/giopuppy/8409370600. Accessed: 2021-06-30. v, 20

[4] Google's go-playing ai still undefeated with victory over world number one. https://www.theguardian.com/technology/2017/may/25/alphago-google-ai-victory-world-go-number-one-china-ke-jie. Accessed: 2021-07-23. 1

[5] Intel xeon processor e5-2697a v4. https://ark.intel.com/content/www/us/en/ark/products/91768/intel-xeon-processor-e5-2697a-v4-40m-cache-2-60-ghz.html. Accessed: 2021-06-08. 29

[6] Pommerman rewards. https://github.com/haidertom/Pommerman. Accessed: 2021-07-01. 30

[7] Spearman's rank-order correlation. https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php. Accessed: 2021-08-01. 32

[8] *Statistical Inference to Compare Parameters from Two Populations*, chapter 11, pages 349–378. John Wiley & Sons, Ltd, 2020. 35

[9] Pieter Abbeel and John Schulman. Deep reinforcement learning through policy optimization. https://people.eecs.berkeley.edu/~pabbeel/nips-tutorial-policy-optimization-Schulman-Abbeel.pdf. 12, 13

[10] Sherief Abdallah and Michael Kaisers. Addressing environment non-stationarity by repeating q-learning updates. *J. Mach. Learn. Res.*, 17(1):1582–1612, January 2016. 38

[11] Stefano V. Albrecht and Peter Stone. Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95, May 2018. 1, 15

[12] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search, 2017. 38

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. 18

[14] Hendrik Baier and Michael Kaisers. Guiding multiplayer mcts by focusing on yourself. *2020 IEEE Conference on Games (CoG)*, pages 550–557, 2020. 2, 3, 6, 35

[15] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, 2011. 29

[16] Darse Billings, Neil Burch, Aaron Davidson, Robert Holte, Jonathan Schaeffer, Terence Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. pages 661–668, 01 2003. 5

[17] Branislav Bošanský, Viliam Lisý, Marc Lanctot, Jiří Čermák, and Mark H.M. Winands. Algorithms for computing strategies in two-player simultaneous move games. *Artificial Intelligence*, 237:1–40, 2016. 2

[18] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. v, v, v, v, 2, 6, 7, 8

[19] Guillaume Chaslot, Sander Bakkes, Istvan Szitaand, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai, 2008. `https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf`. 6

[20] Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04:343–357, 11 2008. 8

[21] Andry Chowanda and Alan Chowanda. Generative indonesian conversation model using recurrent neural network with attention mechanism. *Procedia Computer Science*, 135:433–440, 01 2018. v, 19

[22] Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 433–445, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 3, 9

[23] Markus Esser, Michael Gras, Mark Winands, Maarten Schadd, and Marc Lanctot. Improving best-reply search. volume 8427, 08 2013. 3

[24] Peter I. Frazier. A tutorial on bayesian optimization, 2018. 38

[25] Chao Gao, Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. Skynet: A top deep rl agent in the inaugural pommerman team competition, 2019. 30

[26] Chao Gao, Bilal Kartal, Pablo Hernandez-Leal, and Matthew E. Taylor. On hard exploration for reinforcement learning: a case study in pommerman, 2019. 22

[27] G. Gerritsen. Combining monte-carlo tree search and opponent modelling in poker. 2010. 1

[28] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2047–2052 vol. 4, 2005. 17

[29] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz de Cote. A survey of learning in multiagent environments: Dealing with non-stationarity, 2019. 15

[30] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. Agent modeling as auxiliary task for deep reinforcement learning, 2019. 15, 25, 29, 31

[31] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, Oct 2019. 1

[32] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. 17

[33] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks, 2016. 13, 15

[34] Andrew Jaegle, Vahid Mehrpour, and Nicole Rust. Visual novelty, curiosity, and intrinsic reward in machine learning and the brain, 2019. 16

[35] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax, 2017. 19

[36] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. 31

[37] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. volume 2006, pages 282–293, 09 2006. 6

[38] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning, 2018. 15

[39] Marc Lanctot, Viliam Lisy, and Mark H.M. Winands. Monte carlo tree search in simultaneous movegames with applications to goofspiei, 2013. `https://dke.maastrichtuniversity.nl/m.winands/documents/wcg13-smmcts.pdf`. 5, 9

[40] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998. 16

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization, 2018. 38

[42] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for nas, 2019. 35

[43] Diego Perez Liebana, Raluca D. Gaina, Olve Drageset, Ercüment Ilhan, Martin Balla, and S. Lucas. Analysis of statistical forward planning methods in pommerman. In *AIIDE*, 2019. 35

[44] Yong Liu, Weixun Wang, Yujing Hu, Jianye Hao, Xingguo Chen, and Yang Gao. Multi-agent game abstraction via graph attention neural network, 2019. v, 19, 25, 26

[45] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press, 2003. 25

[46] Yeganeh M. Marghi, Farzad Towhidkhah, and Shahriar Gharibzadeh. Human brain function in path planning: a task study. *Cognitive Computation*, 9(1):136–149, 12 2017. 1

[47] Maja J Mataric. Reward functions for accelerated learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 181–189. Morgan Kaufmann, San Francisco (CA), 1994. 16

[48] Tambet Matiisen. Pommerman baselines. `https://github.com/tambetm/pommerman-baselines`, 2018. 29

[49] Kiminori Matsuzaki. Empirical analysis of puct algorithm with evaluation functions of different quality. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 142–147, 2018. 3

[50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602, December 2013. 12, 16

[51] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1602.01783, February 2016. 13

[52] Mavuto Mukaka. Statistics corner: A guide to appropriate use of correlation coefficient in medical research. *Malawi medical journal : the journal of Medical Association of Malawi*, 24:69–71, 09 2012. 31, 32

[53] R. Myerson. Game theory - analysis of conflict. 1991. 4

[54] R. Newcombe. Two-sided confidence intervals for the single proportion: comparison of seven methods. *Statistics in medicine*, 17 8:857–72, 1998. 35

[55] J. Nijssen. Monte-carlo tree search for multi-player games. 2013. 3

[56] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning, 2018. 15, 17, 18, 31, 33

[57] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994. 5

[58] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V. Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning, 2019. 15

[59] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013. 31

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. 29

[61] Nick Petosa and Tucker Balch. Multiplayer alphazero, 2019. 2, 3

[62] M. Ponsen, S. Jong, and Marc Lanctot. Computing approximate nash equilibria and robust best-responses using sampling. *ArXiv*, abs/1401.4591, 2011. 4, 5

[63] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A multi-agent playground, 2018. 2, 3, 21

[64] Cinjon Resnick, Roberta Raileanu, Sanyam Kapoor, Alexander Peysakhovich, Kyunghyun Cho, and Joan Bruna. Backplay: "man muss immer umkehren", 2018. 30

[65] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017. 12

[66] D. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. 16

[67] Maarten P. D. Schadd and M. Winands. Best reply search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:57–66, 2011. 2

[68] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. 13

[69] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, Oct 2019. v, 33, 34

[70] L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953. 4

[71] Craig Sherstan, Bilal Kartal, Pablo Hernandez-Leal, and Matthew E. Taylor. Work in progress: Temporally extended auxiliary tasks, 2020. 15, 38

[72] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. 1, 38

[73] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021. 37

[74] Shubhendra Pal Singhal and M. Sridevi. Comparative study of performance of parallel alpha beta pruning for different architectures. *2019 IEEE 9th International Conference on Advanced Computing (IACC)*, Dec 2019. 2

[75] Nathan Sturtevant and Richard Korf. On pruning techniques for multi-player games. pages 201–207, 01 2000. 2

[76] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA, 2018. 4, 11, 12, 13, 15, 16

[77] Donald Thistlethwaite. A critical review of latent learning and related experiments. *Psychological bulletin*, 48(2):97, 1951. 1

[78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. 19

[79] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018. 19, 38

[80] Jesse Vig. A multiscale visualization of attention in the transformer model, 2019. 38

[81] Jesse Vig. Visualizing attention in transformer-based language representation models, 2019. 38

[82] P. Wang, Yao Qian, F. Soong, L. He, and Zhao Hai. A unified tagging solution: Bidirectional lstm recurrent neural network with word embedding. *ArXiv*, abs/1511.00215, 2015. v, 18

[83] Yizao Wang, Jean-Yves Audibert, and Remi Munos. Algorithms for infinitely many-armed bandits. pages 1729–1736, 01 2008. 9

[84] Yizao Wang and Sylvain Gelly. Modifications of uct and sequence-like simulations for monte-carlo go. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 175–182, 2007. 3, 8

[85] Hua Wei, Nan Xu, Huichu Zhang, Guanjie Zheng, Xinshi Zang, Chacha Chen, Weinan Zhang, Yanmin Zhu, Kai Xu, and Zhenhui Li. Colight. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, Nov 2019. 18, 19

[86] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992. 12

[87] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *ICLR*, 2017. 13

[88] Annie Xie, Dylan P. Losey, Ryan Tolsma, Chelsea Finn, and Dorsa Sadigh. Learning latent representations to influence multi-agent interaction, 2020. 15

[89] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention, 2016. 38

[90] Xiaofeng Yuan, Lin Li, and Yalin Wang. Nonlinear dynamic soft sensor modeling with supervised long short-term memory network. *IEEE Transactions on Industrial Informatics*, PP:1–1, 02 2019. v, 17

[91] Meng Zhou, Ziyu Liu, Pengwei Sui, Yixuan Li, and Yuk Ying Chung. Learning implicit credit assignment for cooperative multi-agent reinforcement learning, 2020. 15

[92] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, page 1729–1736, Red Hook, NY, USA, 2007. Curran Associates Inc. 5

[93] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications, 2021. 2