©*Journal of Aerospace Information Systems*, 2025

Deep Reinforcement Learning in Wargaming

Giacomo Del Rio * and Oleg Szehr† and Alessandro Antonucci‡ and Marco Pierbattista§
Dalle Molle Institute for Artificial Intelligence (IDSIA) – SUPSI/USI, Via la Santa 1, Lugano-Viganello, 6962, Ticino, Switzerland

Matthias Sommer¶ and Michael Rüegsegger‖
Armasuisse Science & Technology, Feuerwerkerstrasse 39, Thun, 3602, Bern, Switzerland

We investigate the potential of deep *reinforcement learning* (RL) for the development of autonomous wargaming agents. We discuss the relevant characteristics of wargaming environments for the design of learning systems, the choice of learning framework and algorithms. While deep RL has been demonstrated to achieve superhuman levels in various games, we argue that these findings can be only partially transferred to practical wargaming. This is due to real-world limitations such as the availability of financial and data resources, but also architectural system requirements that might rarely be satisfied in the wargaming area. The high degree of realism of modern warfare simulation environments is often accompanied with a system latency that entails impractical training times. For an empirical analysis, we adapt various deep RL techniques to the popular *Command: Modern Operations* simulation environment providing a proof-of-concept for deep RL training applications in this environment.

I. Introduction

Within the defence modelling and simulation domain, wargaming is a widely accepted technique for operations planning, personnel training, procurement, among others [1]. Wargames provide safe-to-fail environments to explore defence scenarios and to identify which *Course of Actions* (CoAs) lead to a win/success and which ones lead to a loss/fail, typically at a lower cost as compared to the real-world investigation [2]. However, traditional wargaming, where military planners compete with human adversaries (so-called *Red Team* operators), faces important limitations. Tracking of cause and effect is challenging and outcomes often depend on the subjective assessments of human experts. The typical reliance of the Red Team upon the established tactics and procedures quickly leads to over-constrained CoAs.

*Researcher, IDSIA, giacomo.delrio@idsia.ch (corresponding author)
†Researcher, IDSIA
‡Senior Researcher, IDSIA
§Associate Fellow, IDSIA
¶Scientific Project Manager, Armasuisse S+T
‖Head of Competence Center for Artificial Intelligence and Simulation, Armasuisse S+T

Exploring wider CoAs is limited both by the availability of financial and human resources. In real-world scenarios and complex simulations (characterised by continuous space and time coordinates, large branching factors, high system latency, geographically dispersed computation and other features) the relatively small number of explored CoAs leaves doors open for an overall biased and vulnerable decision-making.

In recent years, the integration of automation [3], specifically distributed interactive simulation [4], and *Artificial Intelligence* (AI) [5] has significantly enhanced the capabilities of wargaming, supporting real-time decision-making, the exploration of more diverse and adaptive scenarios, and the integration of computation among dispersed computational resources. Among these techniques, *deep Reinforcement Learning* (deep RL) showed the most promising results. These AI-enhanced approaches have broadened the strategic scope of wargaming, enabling more complex and adaptive scenario analysis in areas ranging from cyber defence [6] to international relations [7].

One of the tools increasingly used in defence simulations is *Command: Modern Operations*[*] (CMO), an off-the-shelf wargaming platform with professional-grade capabilities. CMO is a warfare video game where the player takes the role of a mission commander in various conflict scenarios. CMO offers global satellite mapping, a wide array of military assets, high-fidelity physical dynamics, and realistic sensor modelling. Its professional edition includes advanced capabilities specifically designed to meet the operational and analytical needs of defence organisations. This includes the possibility of configuring military assets with real-world parameters, (Monte Carlo) simulation engines for the statistical analysis of combat scenarios, and an *Application Programming Interface* (API) that provides direct access to the game's internal state. Overall CMO's high degree of realism makes it a natural choice for reliable scenario analysis and wargaming. An example of a wargaming scenario in CMO is in Fig. 1.

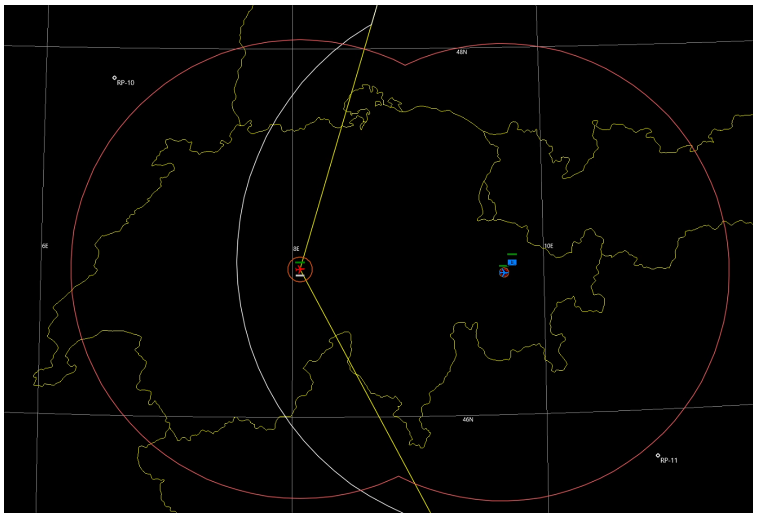

Fig. 1 A wargaming scenario simulated in CMO.

Our goals include assessing the feasibility of applying deep RL in the context of CMO, analysing and discovering

[*]`https://command.matrixgames.com`.

tactics to improve operational outcomes within the existing capabilities, and applying new capabilities to identify procurement priorities. This work originates from a partnership between the Dalle Molle Institute for Artificial Intelligence (IDSIA) and the Centre of Technology of the Swiss Department of Defence (Armasuisse S+T).

The paper is organised as follows. In Sec. II we review the current literature about the application of deep RL to wargaming. In Sec. III we describe the main hindering factors of wargames in the light of the applications of deep RL. In Sec. IV we discuss the design choices that allows to define an effective deep RL algorithm in case of wargames. In Sec. V we present the simulation software CMO together with `war-sim`, a lighter alternative we implemented to overcome CMO's high latency. In Sec. VI, we apply our findings to simple wargaming scenarios in CMO and `war-sim`. We report the training performance of different RL schemes both for CMO and `war-sim`. Conclusions and outlooks are in Sec. VII. In the appendixes, we gather background concepts about traditional search algorithms (App. A), RL (App. B), and deep RL (App. C), together with a discussion on to choice of the most suitable RL algorithm for a given wargame (App. D).

II. Related Work

We investigate the possibility of developing autonomous agents to analyse air defence scenarios, a field in which CMO possess unique simulation capabilities. Our work stands in a broader line of research, that targets the development of AI for wargaming (see, e.g., the surveys [3, 5, 8, 9]). Yet, only few research groups, such as DARPA and Northrop Grumman [10], and the Széchenyi István University [11] focused on the practical development of agents for CMO.

Deep RL, i.e. *reinforcement learning* (RL) in conjunction with deep learning techniques, particularly the training of *neural networks* (NNs), has recently become the state-of-the-art method for the creation of AI agents in games. Given sufficient data, deep learning systems can learn to replicate, and in some cases exceed, human-level performance in a variety of game domains [12, 13]. Notably, DeepMind's application of deep RL to Atari games [14] demonstrated that agents could learn complex control tasks entirely from scratch (*tabula rasa*), without prior knowledge or expert demonstrations. Modern deep RL systems typically combine data generation and deep learning techniques during the training process. Training typically begins with randomly generated CoAs, which are evaluated and refined by a deep learning system. As training progresses, the system increasingly biases future sample generation toward more effective CoAs, creating a feedback loop where both the model and the data improve iteratively. This approach removes the need for manually curated human gameplay data and eliminates constraints on both the volume and quality of training experiences. As a result, deep RL agents have surpassed human performance in a variety of control and coordination tasks - including Atari games based on raw pixel input [14, 15], drone racing [16], car racing simulations [17], and general-purpose benchmarks [18]. Deep RL agents also surpassed human tactical and strategic capabilities in combinatorial planning problems such as chess or Go [19]. In combined control and strategic tasks (typical of real-time games), deep RL systems (with supervised pre-training) have demonstrated considerable potential, matching/surpassing human level in the case of Dota 2 [20] and Starcraft 2 [21]. These environments bear considerable similarities with contemporary wargames, which motivates our investigation of deep RL in the context of CMO. In the defence area deep RL has been applied to numerous simulation problems, ranging from aerial combat [22, 23] over target tracking [24, 25], to applications in wargaming [26, 27].

To the best of our knowledge, aside from the present work, the only published work on an existing application of deep RL algorithms to CMO is presented in [11], where exclusively the PPO algorithm was employed. In contrast, our study explores a broader range of algorithms, thereby enabling a more sophisticated integration with the CMO engine.

III. Factors Hindering the Application of RL to Wargaming

Let us first discuss the features of wargames that should be taken into account when designing a scenario to be solved by RL. Modern wargames are often designed to reflect real-world warfare scenarios accurately, both on the tactical and operational level. Inevitable abstractions are positioned in a way that preserves the real-world scenario's main features in view of the objectives of the wargame. The application of deep RL to real-world scenarios comes with specific challenges, ranging from the ability to learn from a small number of available samples, over the ability to deal with high-dimensional state and action spaces with partial observability, to physical system limitations such as system delays and system breakdowns [28]. Besides that, most of the challenges responsible for the complexity of real-world deep RL should be also expected in a similar form in wargames. Typically, this makes wargames very complex environments, and the construction of strong artificial players a challenging endeavour. In what follows, we summarise the hindering factors of modern computer-assisted wargames in view of their application of deep RL techniques.

A. State Space

The state space is the set of possible states the game might reach. The size of the state space contributes to the size of the *game tree*[†] and its complexity. For chess and Go, the possible configurations are estimated to be 10^{47} and 10^{170}, respectively. In real-world scenarios with continuous variables such as unit coordinates, velocities, angles of orientation, among others, the state space is technically infinite. Thus, realistic wargames lie beyond the access of exhaustive computation (but they remain finite because a binary computer may only take a finite number of states). A common simplification is that the *value function* (i.e., the function that assigns the expected outcome of the game to the current state) is locally continuous: for combinatorial games like chess and Go the exact position of pieces is crucial, even a “small” change might have a drastic effect to the game's outcome. In contrast, the value function in wargames is often less sensitive along the continuous variables. Moving, for instance, an asset by 1 cm will usually have little impact on the outcome of multi-asset operations. There are limitations to this phenomenon, e.g., when leaving a shield or stronghold even by a small margin.

[†]The *game tree* [29], defined as the graph representation of all possible states that can be reached, plays a key role in determining the complexity of the game. The larger is the game tree, the harder the game will usually be for computational methods, from classical search to modern deep RL.

B. Action Space

The action space contains all possible actions that can be taken in the course of a game, which is another important factor in assessing a game's complexity. For Go the size of the action space is 361 (corresponding to a 19×19 board). For real-world scenarios with continuous choices, the action space is technically infinite. In wargames, the action space often consists of a combination of continuous and discrete choices. For instance, one might choose a (continuous) position on a map or decide on the (discrete) number of available missiles to launch.

C. Branching Factor

The branching factor is the number of consecutive states that can be reached from a given state. For deterministic games, the branching factor is bounded by the possible actions, i.e., the size of the action space. In the presence of stochasticity or partial (imperfect, see below) information, the branching factor can be much larger. In such cases a chosen action might result in multiple CoAs, which must be accounted for during decision making. The branching factor can be huge in wargames. E.g., for Starcraft, a total of around 10^{26} actions per round are estimated [21].

D. Planning Horizon

The number of actions taken during a game is the depth of the game tree. Typically, the dependency of the size of the game tree on the number of actions is exponential, making exhaustive computation prohibitive even for a moderate number of actions. It is, therefore, common to introduce a *planning horizon* for algorithms, which reflects the maximum depth to which the algorithm plans. The states at the end of the planning horizon are often valued by a heuristic method, such as a tailored position evaluation function in traditional α/β search for Chess, or a trained NN in the AlphaZero algorithm. In the case of discrete-time games, the number of actions can be counted once the game is completed. For continuous-time games it is common to introduce tick frequency and a fixed number of actions per tick to discretise the time coordinate. With realistic estimates for the frequency of human input, this can easily lead to a total number of actions in the order of magnitude of 10^4 for modern wargames [21].

E. Stochasticity

In RL, stochasticity refers to the degree of randomness associated with the outcome of a selected action given a known state [30]. This occurs in games whose state transitions depend on the player's choices and random processes, such as the turn of a roulette. In stochastic games, the player is faced with considerations about the outcomes of CoAs and identifying their probability distributions. The *observation space* refers to the part of the state space visible to the player and known with certainty. If observation is limited to part of the game's state, the agent is faced with *imperfect information*. Such an imperfection entails random outcomes (from the player's perspective) even when the overall game dynamics is deterministic. Moreover, it is common in information theory to interpret any form of randomness as the result of partially observed deterministic behaviour of a larger system. For example, similarly to stochastic games, the player will assign probabilities over unknown states in card games, where the own hand is observable but not the hands of other players. In wargames, “fog of war” is generally used to describe a common source of imperfect information, e.g., the player might only see their units and those of the enemy within line-of-sight. Randomness increases a game’s complexity because the possibility of multiple outcomes given state-action pairs corresponds to a larger branching factor. This leads to the “paradoxical” property that a smaller observation space makes the game more complex.

Different levels of information and decision-making authority are present in the hierarchy of defence organisations. The transition from lower to higher hierarchy goes along with an information abstraction process, from individual unit observations on the battlefield to strategic position, strength and condition of armed forces at the commander level. There is no uniform notion of imperfect information in real-world conflicts. Instead, information is filtered through many layers of communication and processing. The form, role, and impact of the “fog of war” thus differ within the hierarchy, which is instrumental for the design of AI systems. This significantly impacts the choice of RL algorithms. However, the number of wargames that support information filtration is currently limited.

Finally, the notion of *incomplete information* is worth mentioning. This refers to a lack of knowledge about the opponents’ objectives, as opposed to a lack of knowledge about the state of the game in case of imperfect information. In real-world defence scenarios, the enemy’s objectives constitute an important consideration. A hostile operation might serve the immediate objective, but it might also be a deflection to reach strategic goals elsewhere. Despite its importance, incomplete information is rarely present in wargames. As an example, Hasbro’s game *Risk*[‡] is a widely played wargame with stochasticity and perfect but incomplete information.

F. Motivations

The high complexity of wargames, the size of their game tree and the unavoidable limitations in terms of resources limitations makes unrealistic the development of a general game-playing agent (similar to AlphaStar). Pre-set scenarios should be considered instead. Deep RL agents should be trained within a specific scenario and adversary playing style (i.e., overfitting to scenario and strategy). Yet, the analysis of specified scenarios and strategies carries considerable value, as detailed by the following four points.

- Agents can be trained given moderate resources. This allows for the online assessment of time-critical scenarios, monitoring a superhuman number of potential CoAs.
- In scenarios without time-criticality, e.g., in Red-teaming, an artificial agent might cover greater width and depth of CoAs. Confronting Blue Team operators with unexpected situations enables them to improve their strategies.
- In procurement projects, the performance of varying assets in specific combat scenarios might be simulated to decide on their value for the armed forces. The increased variability of AI-based simulations enhances the

[‡]`https://www.hasbro.com/common/instruct/risk.pdf`.

robustness of the analysis.

- More broadly, deploying artificial agents constitutes a step towards automation and cost saving. Human resources might be freed up by AI agents that take their roles on wargaming platforms.

In the next section, we discuss how this can be practically achieved by RL.

IV. On the Application of RL to Wargaming Environments

In this section, we discuss how the challenges of applying RL to a wargame scenario, highlighted in the previous section, can be addressed through dedicated algorithmic and design choices.

Let us first introduce some basic concepts. RL is an iterative process in which an agent learns to accumulate rewards through interaction with an environment [31]. The learning process is commonly organised in several independent training *episodes*, each composed of individual agent-environment interactions. Each episode corresponds to a complete execution of the agent's task, such as playing a game from beginning to end. In each interaction, during every episode, the agent considers the current state of the environment and executes a corresponding action. The environment subsequently transitions to a consecutive state, granting a reward to the agent. The rewards are accumulated over the entire episode. Although the behaviour of the environment can be stochastic, the main underlying assumption is that there is one *fixed* probability that determines the transitions of the environment and the rewards over the different interactions. The main idea behind RL is that while running over many episodes, the agent adapts to the transition probabilities of the environment learning to maximise the accumulated rewards. In two-player games like chess, the active player takes the role of the agent, while the position on the board corresponds to the state of the environment. From the agent's perspective, the opponent's action corresponds to a transition of the environment. The main underlying assumption is defined as *Markovianity* and indicates that chosen actions depend on the current game state (position on the board) but not on how this position has emerged.

What follows is a discussion of the design choices one has to take when applying RL to a given scenario.

A. Rewards

The *reward signal* describes the sequence and quantity of rewards granted during an episode. The underlying RL task determines the structure of the reward signal. Some simpler games have *immediate* rewards, i.e., the consequences of an action become obvious directly after its execution. Complex games (like chess and Go) often possess *episodic* rewards, where only at termination the entire sequence of decisions is evaluated (granting rewards of, e.g., 1/0/-1 for win/draw/loss). Reward signals are composed of immediate and episodic components in the case of realistic wargames. In other words, taking a sub-optimal action regarding the immediate consequences (immediate reward) to achieve a better overall result (a higher accumulated reward) might be beneficial. In a wargame, this might occur when a unit is sacrificed to achieve the objectives of a mission. It is typically hard to learn from episodic reward signals when episodes come with large numbers of possible CoAs because the signal is sparse in the space of CoAs. From an engineering perspective, there is often flexibility to design the reward signal to facilitate the learning process. For example, one might provide a current material count to facilitate learning in chess. In wargames, rewards might be assigned for completing sub-goals, such as conquering an enemy stronghold. On the one hand, designing an informative reward signal might be a prerequisite for successful learning. Still, any manual intervention on the reward signal entails a bias that might shed important CoAs. A common and advisable approach is to begin training with an engineered immediate reward signal, moving successively to episodic rewards in the course of the process.

B. Exploration & Exploitation

To learn, the agent must acquire knowledge, which might be represented as statistics of rewards given states and actions. This leads to a natural question: should the agent choose a lucrative action or try sub-optimal actions to sharpen his statistical estimates? This dilemma is at the core of RL, with a significant body of literature dedicated to its variations [31]. The trade-off between exploration and exploitation is present at each game tree node. Due to its exponential size, even tiny losses from sub-optimal exploration will significantly impact the outcomes of training. For complex wargames, close-to-optimal exploration is therefore vital and should be achieved whenever possible.

C. Model-Based Versus Model-Free

Model-free RL agents have no knowledge of their environment, learning exclusively from reward signals [31]. They face a situation where actions are executed consecutively, and the only available information is the observed sequences of states, actions and rewards. Model-based RL agents, in contrast, possess a model of their environment (i.e., of the state-action-state-reward probabilities) that allows them to plan reward outcomes before taking an action. In practice, it is more common that a *simulator* rather than a complete environment model is available to take samples from the environment. Unsurprisingly, making use of the additional environment simulator planning-based approaches learn better actions faster, reaching higher performance scores than the best model-free methods. It should be mentioned that the headline agents for playing chess, Go, and StarCraft (AlphaZero, AlphaStar) are model-based. They are trained via *self-play*, where another version of the learning agent takes the role of the environment simulator. Roughly speaking, to plan its actions, AlphaZero executes tentative moves on a virtual board and considers the replies that it would have given in the emerging positions. It is a primary distinction in developing RL methods for wargames to identify whether the application of model-based RL is feasible and beneficial. For now, we mention that training model-based agents comes at the price of instantiating many copies of the game engine to analyse promising CoAs. This usually entails a more complicated software infrastructure than model-free methods.

D. Training

Many RL algorithms possess strong convergence guarantees. See, for instance, the results in [32] for model-free Q-learning and those in [33] for model-based Monte Carlo Tree Search (MCTS). Yet, training RL agents for complex games without prior information (the so-called *tabula rasa* learning) comes at horrendous computational costs. Deep RL combines RL with deep learning techniques for improved representation, generalisation and faster evaluation. In practice, this often leads to a significant improvement in performance and convergence speed. Still, the computational costs might remain impractical for complex games, and the convergence guarantees are lost (as training can get stuck in a poor local optimum). The cost for one training cycle of tabula rasa algorithm AlphaZero applied to chess and Go is estimated around USD 250'000 [9].

Starcraft can serve as a reference point for wargames with continuous state and action spaces. The AlphaStar agent performs only moderately when trained tabula rasa [21]. The main hurdle lies at the beginning of the training process. If an agent plays random actions in a complex game, the outcome will be a loss with overwhelming probability. In all such cases, the agent receives the same reward but no increase in knowledge. Supervised pre-training is a method to overcome this hurdle that is commonly employed in practice [34]. Purely supervised learning based on (human) expert guidance can achieve remarkable performance in many games, including chess, Go and Starcraft. However, supervised learning is limited by the amount and quality of available data [35]. Once the data is exhausted, RL can fine-tune and improve the pre-trained agents (e.g., reaching superhuman performance in Starcraft). If no data is available, it is sometimes possible to overcome the entrance barrier by adapting the reward signal as training progresses. For many wargames, obtaining sufficient data from expert play can be difficult. In such cases, one might begin training based on asset count and switch to episodic mission-level reward only after a specific strength has been reached.

As it is the case of supervised learning, deep RL systems carry the risk of overfitting. For wargames, this might arise when an agent "learns by heart" the sequence of actions that accomplishes a given mission but performs poorly otherwise. A more subtle form of overfitting occurs when an agent adapts to the tactics and strategy of a particular type of adversary or a specific scenario. Such agents are limited by the versatility of their CoAs, which can be exploited easily by more versatile agents [36].

E. Wargames are Slow Environments

Wargames originate from a time when the application of RL to investigate defence scenarios was elusive. Their evolution has focused on real-time gaming experience on the side of the gaming industry and on simulation and accuracy in the defence area. The latency of simulation and the access to the game's internal operations, which are key features from an RL perspective, have less been in the focus of the development. From an RL perspective, this typically makes wargames *slow* environments.

In modern wargames, the training process comprises the evaluation of huge (commonly of the order of magnitude of billions) numbers of game states and the execution of an analogous number of actions. This makes fast access to the game's internal state and a lean implementation of the game's dynamics vital. This leads to an *accuracy versus speed trade-off*, where higher simulation accuracy makes the environment more realistic but learning more difficult. The best performance of a wargaming AI (as measured by the objectives of wargaming) can thus be expected in intermediate environments that are relatively accurate but still sufficiently fast. The delicate trade-off between simulation accuracy and accessibility for advanced learning techniques remains a crucial open point in the design of commercial wargames.

Aside from wargaming, slow environments constitute a common challenge for real-world RL [28]. Common reasons for high environment latency include physical limitations, e.g., when training a robot [37], or intrinsic delays in the reward signal, e.g., in a recommender system, where the true reward is based on the user's acceptance [38]. Two important techniques to address slowness are data storage and the application of simplified dummy environments. Storing data means that the outcomes of all training episodes are saved and re-used for the initialisation of subsequent training cycles. A dummy environment simulator should capture the main aspects of the actual environment but allow for significantly faster execution. The idea is to train the agent in the dummy environment first and then fine-tune it in the real environment, see [39] for an application in wargaming. Both techniques are closely related to supervised pre-training because they aim at "pre-heating" the learning machine to overcome the RL starting hurdle.

F. State Access and Action Execution

The complexity of wargames also makes them prone to system crashes and other forms of instability. Even if errors occur at a frequency so low that it has no influence on human gaming experience, the impact on RL might be substantial. Wargames are usually commercial software tools, which implies limited access to the game's internal state, limited opportunities to develop faster access options and limited opportunities for debugging. Finally, game state access and manipulation require a dedicated API, which might not be present in off-the-shelf software.

Model-based RL systems (such as MCTS and its derivative AlphaZero) create a look-ahead planning tree to evaluate the consequences of actions before their execution. To generate the planning tree, the algorithm traverses sequences of states and actions on a multitude of environment copies. This requires the capability to maintain many copies of the wargame and the ability to navigate through the planning tree in a non-consecutive way, jumping between game nodes at varying instances of time. Moving to a non-consecutive state means that the game must be loaded at a specific state, which introduces an additional latency that might pose a bottleneck for applying such methods to wargames. Finally, it should be mentioned that planning-based methods are particularly vulnerable to systems instabilities. To avoid errors in parallel processes resulting in the loss of the entire planning tree, error handling and parallelisation mechanisms must be coordinated in a dedicated software architecture.

The ideas discussed in this section will guide the experiments with our simulator, which is introduced here below.

V. The CMO Simulation Environment

Our experiments are based on the CMO simulator, which raised considerable interest as a simulation environment for defence-related professionals and organisations [10]. Let us report here an overview of its capabilities and characteristics.

A. The CMO Environment

CMO's state and action spaces contain continuous and discrete components. Assets can have discrete (such as the amount of carried ammunition, the *engaged mode*, i.e., an asset has made enemy contact and enters combat) and continuous (geographic location and terrain properties as well as spatial orientation) features. CMO's game state can be written out in XML format with an *order of magnitude* (OOM) of 1'000 leaves for a simple scenario. Available actions include the navigation of assets to any coordinates of the world map (continuous) and the choice of ammunition (discrete). Other possible action types include assigning a new mission to a unit, turning detectable emissions (such as sonars or radars, emission control) on or off, returning to base, attacking, among others. An OOM of ten main action types can be issued to the game engine.

Overall CMO's environment characteristics share similarities with the complex Starcraft environment. Both environments comprise discrete and continuous components in state and action spaces, continuous time, the availability of a variety of complex units, the presence of stochasticity and imperfect information that depends on the line-of-sight of units and many other features. Like Starcraft, a rough discretisation of coordinates into a 360×360 latitude/longitude-grid leads to very high branching factors (10^{26} has been estimated for Starcraft).

CMO models asset motion with physically accurate dynamics. Rocket motion, for example, is based on a physical energy-based model that takes rocket weight and fuel into account. Various sources of randomness are present in CMO. The player is faced with imperfect information about the location and capabilities of enemy units at the beginning of the scenario. If enemy units are visible, their capabilities might only be partially known. The game environment is also stochastic: the damage done by one unit to another and the mechanism of detection of enemy units (e.g., the implementation of radars) carry randomness, and this influences the size of the observation space. There is also uncertainty about the mission targets of adversaries, i.e., CMO could carry incomplete information.

B. CMO Advanced Functionalities

CMO's professional edition offers advanced functionalities, such as database-editing access, statistical analysis of combat scenarios and an enhanced TCP/IP LUA scripting API (LUA-API) that provides access to the simulation engine. The LUA-API offers direct interaction with the CMO game state and core engines, allowing, e.g., starting/stopping of simulations and saving/restoring of a state of the simulation. Database editing enables the creation of new assets and the configuration of existing assets with realistic parameters. An internal AI engine controls asset behaviour and provides basic asset capabilities and interaction dynamics: according to our observations, this AI appears to execute a combination of stochastic and rule-based routines. The professional edition also supports multi-player modes for the analysis of multi-party conflicts.

C. CMO as a Slow Environment

From a deep RL perspective, CMO is a slow environment. This refers to the execution of consecutive actions, the loading of a given state and the instantiation of the entire game engine. We measured a latency of the OOM of 1 s for executing consecutive actions through the LUA-API. If latency were assumed for the model-free RL in benchmark environments [40, 41], this would entail about 10 min for training a Q-learning agent in a 6×9 grid world, and 1.5 h [40] for the mountain car benchmark [42]. This assumes that the best hyper-parameters (i.e., no hyper-parameter tuning). Model-based RL employs look-ahead planning. Frequent jumps to specified game states are inherent to this form of planning. CMO's API supports in-game state transitions by saving and restoring game states; we measured a latency of around 1 s for this process. This makes the construction of complex planning trees in model-based RL resource intensive. It is not obvious whether model-free or model-based methods would achieve higher performance in slow environments with limited hardware resources.

D. Training Process

CMO's similarity to StarCraft suggests that the two environments might share similar RL characteristics. One might, therefore, expect similar challenges during the training process and the utilisation of AlphaStar's training schemes also in CMO. However, we abandoned using an AlphaStar-type architecture for various reasons. First, the aforementioned issue related to the lack of stability of the CMO API. Second, supervised pre-training played an important role in AlphaStar's training process, and no CMO data has been available at our disposal. Without pre-training, AlphaStar reaches a modest performance of approximately 150 ELO compared to a purely supervised agent at approximately 900 ELO. We expect a similar role of pre-training also for the CMO. Third, training AlphaStar requires enormous resources [20]: "The AlphaStar league was run for 14 days, using 16 TPUs for each agent. During training, each agent experienced up to 200 years of real-time StarCraft play". We therefore decided to train two model-free and one model-based agent, building on established RL algorithms. We employed variations of DQN and PPO for the model-free agents. The model-based agent is a derivative of the AlphaZero algorithm, whose planning capability is based on MCTS. Although the mentioned algorithms are established, the latency and instability of the CMO environment require custom development of parallelisation and error-catching techniques.

E. A Simpler and Faster CMO Alternative

To conduct RL training experiments despite the lack of pre-training data and CMO stability and speed, we developed a lightweight Python-based wargaming simulator, called `war-sim`. The code of `war-sim` together with an implementation of the MCTS algorithm, the abstraction layer to connect the various RL algorithms with CMO, and the scripts to reproduce the experiments discussed in the final part of this paper are freely available in a public repository[§]. Similar to CMO, `war-sim` uses 3-D world maps, based on the *CartoPy* package for cartography [43] and takes advantage of geodesic curves in the computation of asset motion using the *geographiclib* package [44]. In `war-sim` only a limited number of assets is available with no autonomous operation capability, and the dynamics models are reduced to a minimum. Despite being written in Python, `war-sim` is one OOM faster than CMO. Tab. 1 compares some basic features and environment latency times of CMO with `war-sim`.

Feature	CMO	`war-sim`
Asset types	> 100	2 (Aircraft, SAM Battery)
Physical Dynamics	Accurate	Approximated
Integrated AI	Yes	No
Simulation step time	~ 1.5 s	~ 0.1 s
State restore time	~ 0.1 s	~ 0.001 s
Engine startup time	~ 15 s	~ 0.1 s

Table 1 A comparison of CMO and `war-sim` main features.

We employ the `war-sim` environment to test RL algorithm choices, measure performance, and transfer model setup and hyper-parameters to the CMO environment. A modified version of `war-sim` has been also used to investigate multi-agent air combat scenarios [22].

VI. Experiments

Modern wargames often combine control and planning exercises. In this section, we analyse the performance of deep RL on three wargaming scenarios. In spite of their simplicity, these scenarios are designed to cover typical wargaming situations. Following our discussion, PPO, DQN and AlphaZero appear to be natural candidates for this application.

A. Design

We consider simple scenarios designed to investigate the learning potential of the chosen RL algorithms in complementary setups. Ideally, the performance should be assessed on a range of scenarios: 1) *control* scenarios, where the focus lies on the accurate control of the motion of individual units; 2) *planning* scenarios, where the agent's capability of assessing CoAs is evaluated; 3) *combat* scenarios, which involve a combination of planning and control. These scenarios often come with characteristic reward signals. In control scenarios a reward can be granted after each simulation step (since actions often can be evaluated immediately after execution), whereas in planning scenarios the reward signal is often sparse and episodic (since the quality CoAs can be assessed only after a number of simulation

[§]`https://github.com/armasuissewt/drl-wargaming`.

steps). Accordingly, combat-type scenarios have combined reward signals that reflect the individual contribution of control and planning exercises.

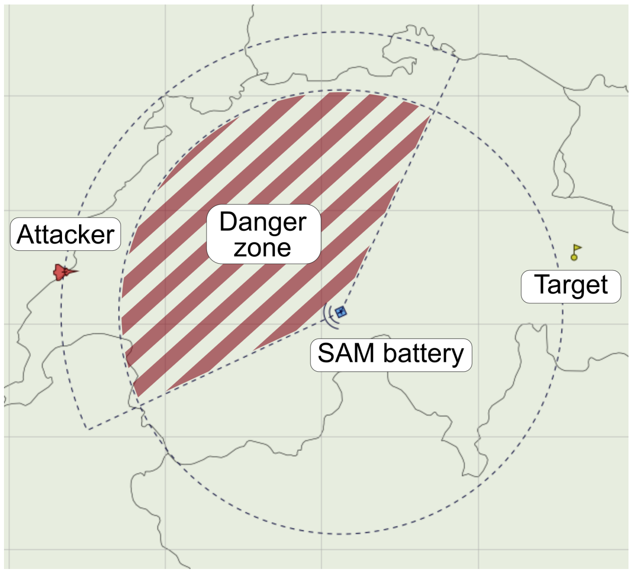

Fig. 2 ***Control*** **and** ***planning*** **scenarios: the attacker should reach mission target avoiding the danger zone.**

In the control scenario an attacking aircraft receives order to reach given target coordinates. Between the attacker's starting and target coordinates a *Surface-to-Air Missile* (SAM) battery is positioned that can destroy the attacker if it enters the cone of the SAM radar and fire, see Fig. 2. The attacker cannot fire at the SAM battery or use any countermeasure; the goal is to learn to avoid the dangerous radar cone of the SAM battery. The optimal strategy is straightforward: it consists of flying a path to the target that avoids the SAM's reach. The planning scenario is identical but only episodic rewards are granted to the agent. The combat scenario, depicted in Fig. 3, involves three units: an attacker aircraft, that can fire both *air-to-air* (A2A) and *air-to-land* (A2L) missiles, a passive SAM battery, and an active defender aircraft. The SAM battery only serves as a target and cannot fire. In this scenario the behaviour of the defender aircraft is hard-coded: it patrols the target, and as soon as an attacker appears on its radar, the defender attacks it.

	Control	Combat	Planning
PPO	Yes	?	?
DQN	Yes	?	?
AlphaZero	Yes	Yes	?

Table 2 **Expected convergence of learning algorithms for different scenarios according to literature.**

As the performance of PPO and DQN on sophisticated control scenarios is well documented in the literature, we forgo them in our experiments. Tab. 2 summarises the convergence result expected from published research for each scenario/algorithm combination within our `war-sim` simulator. A positive finding in this table corresponds to a successful learning of a near optimal strategy.

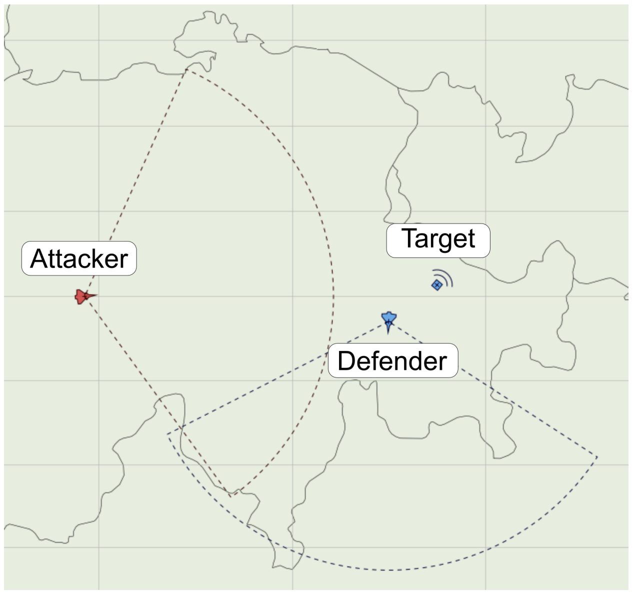

Fig. 3 The *combat* scenario: the attacker should destroy the target.

To assess how simulator properties affect the overall learning process, the planning and combat scenarios are implemented in CMO and `war-sim` as simulator back-ends. For the purpose of these experiments, the `war-sim` simulator offers environment dynamics that deviate from those of CMO only marginally, but with lower environment latency and higher environment stability. Although Tab. 3 shows positive results for `war-sim`, all algorithms ran into difficulties even on simplistic scenarios when executed on CMO directly. Before discussing the implications of our experiments, we begin by providing details on our algorithm design choices and configuration.

B. Environment Characteristics and Observation Encoding

Both CMO and our replica `war-sim`, advance the simulation in discrete steps of simulation time of approximately 0.1 s to 1 s. Such a fine-grained simulation would make the overall computational effort for the considered RL algorithms impractical. Therefore, we aggregate multiple steps of the simulator into one learning step of the RL process. More specifically, for the planning scenario, we set 20 s of simulation time for one step of the RL process, while for the combat scenario, we set instead 15 s. The finer granularity of the combat scenario allow the attacker to optimise the timing of missile launches.

Our subsequent design choices are driven mainly by the scenario geometry and the results observed during our preliminary tests. Observations are encoded in form of a matrix. For the control and planning scenarios, we discretise the map into a grid of 100×100 cells and encode the attacker by a square matrix of dimensionality 100 that contains a single one at the attacker's position. With this discretisation, each cell has side-length of approximately 5 km. This representation is augmented by two real numbers encoding the sine and cosine of the aircraft heading. The combat

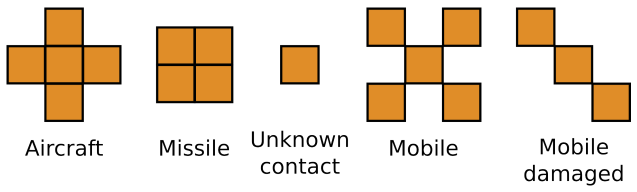

Fig. 4 Markers for different types of units and contacts.

scenario requires a more complex encoding. As before we start from a zero matrix that is filled out according to the following rules:

- For each asset, a marker with a specific shape (see Fig. 4) is placed at the unit's position. The altitude of the unit determines the intensity (filling value) of the marker: the higher the altitude, the higher the intensity.
- A trace connecting the last four positions of each moving unit is placed on the grid with the respective encoding of the altitude.
- Two 3×3 squares filled with 0.5 are placed at the top corners of the grid to indicate the availability of an A2L or A2A missile.
- A 3×3 square filled with 0.25 is drawn only when the attacker aircraft executes a bad firing action, i.e., a firing action when none of the available targets is in the scope of its radar.
- A 3×3 square is superimposed on top of a unit's position in case that the unit has been hit by a missile.

Trails and markers have positive values for the attacker side, while they are inverted (negative values) for the defender side. This encoding not only contains the current state of the simulation but, thanks to the trails, it also encodes a limited *history* of previous observations. An example is in Fig. 5.

C. Action Encoding

The number of the actions that the algorithm can select directly impacts the problem complexity: the more actions, the larger the branching factor. To reduce the overall computational effort, we choose the minimal number of actions needed to solve a scenario, but no more than that. We assume the aircraft's altitude and speed are constant and we discretise its turn angles. For the planning scenario, we assume that the aircraft has no missiles and cannot fire. In this case we consider only three actions corresponding to turning $10°$ left or right and moving forward. For the combat scenario, we consider two additional actions corresponding to firing an A2A or an A2L missile. In this scenario, the attacker has a single A2A missile and a single A2L missile. Firing actions without a suitable target or without available ammunition are not executed but are accounted for a negative reward.

D. Rewards

In the planning scenario, a positive reward is granted if the attacker aircraft reaches the target. If the aircraft exits the map or is shot down this leads to a negative reward. Additionally each action is accompanied by a small negative reward

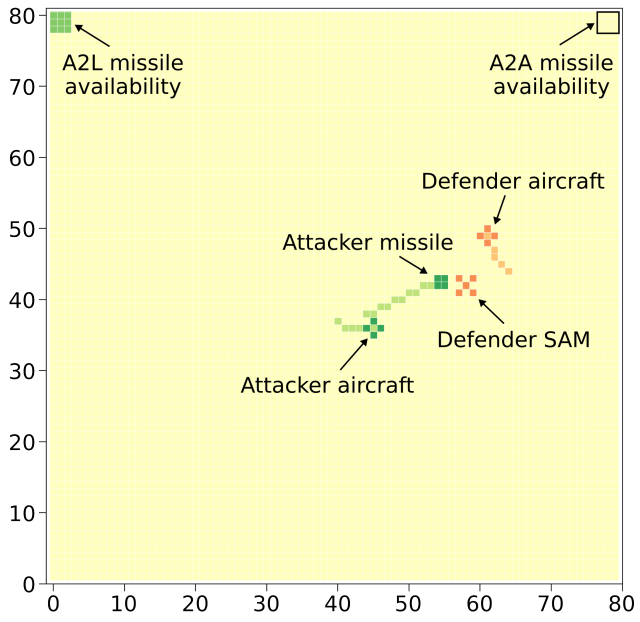

Fig. 5 Observation encoding in a combat scenario (attacker units are encoded in green, defender in orange).

to ensure an efficient trajectory to the target. In summary for the planing scenario we have:

$$r = \begin{cases} +10 & \text{if reaching the target}\,, \\ -10 & \text{if exiting the map or being shot}\,, \\ -0.01 & \text{otherwise}\,. \end{cases} \tag{1}$$

For the combat scenario we have:

$$r = \begin{cases} +8 & \text{if destroying the SAM battery}\,, \\ +2 & \text{if destroying the defender aircraft}\,, \\ -5 & \text{if exiting the map limits}\,, \\ -5 & \text{if being shot}\,, \\ -5 & \text{if firing without an active target}\,, \\ -1 & \text{if firing a A2A or A2L missile}\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{2}$$

E. Algorithms

For PPO and DQN, we used the version provided by the Python library RLlib [45]. Both algorithms include the latest enhancements reported in the literature and provide some basic recovery behaviour in case of environment failures. For AlphaZero, we implemented a custom Python version based on [19] that we adapted to satisfy the requirement of a wargaming environment. This includes problem-specific parallelisation and fault-tolerance of parallel processes with respect to environment failures. The followings are the main characteristics of our AlphaZero implementation. There are *n player* processes that play episodes using the latest trained policy. The players select the best action by building an MCTS tree with a given number of expansions (50 in the planning scenario, 100 in the combat one). The samples collected by the players are stored in a replay buffer. At the same time, a *learner* process retrieves batches of samples from the replay buffer and trains a NN with two heads: one for the policy and the other for the value. As the policy improves, the episodes generated by the player processes achieve higher rewards, which in turn improves the learner process, until convergence is reached for the environment at hand. During evaluation, only the policy head is used to select actions. This has the advantage of reducing the time spent during inference while leveraging multiprocessing during training.

F. Policy Encoding

For the planning scenario, a feed-forward NN with two hidden layers seems sufficient to encode the policy. For the combat scenario, we consider a convolutional neural network (CNN) that offers more power to learn geometric features of the map. Fig. 6 details the structure of the NN for the combat scenario.

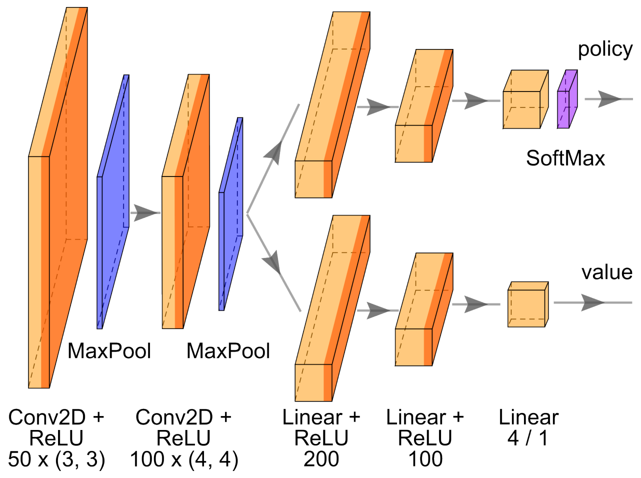

Fig. 6 Exemplary NN structure for the policy in the combat scenario within the AlphaZero algorithm.

G. Connecting CMO and Python

CMO provides an API based on the LUA language, allowing RL agents to interact with the simulation engine via a TCP/IP connection. LUA's design goals have been to achieve a lightweight, intuitive and human-like interaction. To use the simulator as an RL environment, we additionally designed an abstraction library called `cmoclient`, which allows us to spawn a CMO simulator instance easily and to issue commands in a clean Python syntax. The scenarios are then implemented in separate Python classes that interact with CMO through the `cmoclient` library. Each class runs an instance of the simulator with a predefined *scenario file*, connects to the LUA API TCP/IP port of the simulator, sets up the simulation and waits for the RL training to start. With our hardware (see next Sec. VI.H for details), it could be possible to run up to 40 simulations concurrently. In case of a failure in the communication between CMO and `cmoclient` the processes are killed, and a new instance of the engine is started thereafter. The recovery procedure lasts around 30 sec, which constitutes a significant workflow bottleneck. The `war-sim` simulator prevents this issue offering crash-free execution and far lower latency times. In our scenarios this entails far higher training stability and better overall training performance.

H. Setup

We begin our experiments with `war-sim`. We investigate the CMO variants only in case that training has been successful within `war-sim`. In our experiments hyper-parameters have been manually tuned on the `war-sim` variants and then transferred to CMO. We used Command Professional Edition 2.3.1 on a dedicated server with a 64-core processor (AMD EPYC 7763), 256GB RAM, and NVIDIA GeForce RTX 3090 (24GB).

I. Results

We begin with a summary of training results for the combat and planning scenarios within the `war-sim` and CMO environments. We report training results in both environments to showcase the strong impact of environment features to the learning process. Tab. 3 provides a high-level overview of our findings. Technical details follow below.

	Planning Scenario		Combat Scenario	
	`war-sim`	CMO	`war-sim`	CMO
PPO	No	No	No	-
DQN	No	No	Yes	No
AlphaZero	Yes	Partial	Yes	No

Table 3 Convergence of learning algorithms in different experimental setups.

Overall, we observe better learning performances of the tested algorithms across all scenarios on the `war-sim` environment. Despite the small differences between `war-sim` and CMO in terms of scenario dynamics (for the purpose

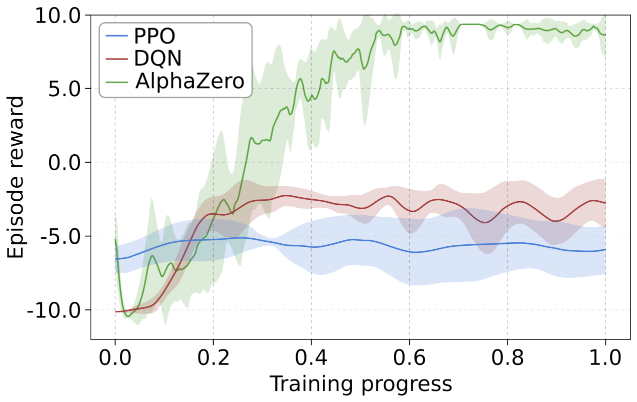

Fig. 7 Training progress for the planning scenario (`war-sim`).

of our experiments), the impact of environment meta-features such as latency and stability is substantial.

Within the `war-sim` simulator, PPO shows the strongest performance on control tasks and simultaneously the weakest performance on combat and planning tasks. See Tabs. 2 and 3 for a comparison. Design features of the planning and combat scenarios require a high level of initial exploration; a lack thereof (that can hardly be mitigated by tuning of hyper-parameters in PPO) could explain PPO's rather poor performance in those cases. We observed some successful PPO training runs, but their frequency was not sufficient to achieve consistent convergence. Overall, this is consistent with performance reports in the literature. Within `war-sim`, DQN showed similarly poor performance on the planning scenario, but a much better performance on the combat scenario. As expected, the planning-based AlphaZero algorithm showed the strongest convergence properties and highest rewards in the planning and combat scenarios in `war-sim`, identifying near optimal policies in both cases. When tested on CMO, all algorithms ran into difficulties. These difficulties include failure of convergence in scenarios where convergence has been achieved in `war-sim` (e.g., DQN on the combat scenario), but also full algorithm execution failures. In the planning scenario, the PPO and DQN algorithms ran successfully, but none of them reached convergence. Similarly, in the combat scenario DQN run successfully but with no convergence. For the combat scenario we forgo running PPO since it did not converge in the `war-sim` case. When applied to CMO, AlphaZero's performance deteriorates due to crashes during saving/restoration of states. We observed a significant number of errors in the CMO action execution and state loading processes; the origin of those errors is beyond the access of the LUA interface user. For AlphaZero those errors entailed that the learning process succeeded only partially in case of the planning scenario and failed for the combat scenario. Given that AlphaZero trained successfully within `war-sim`, we believe that a successful training can also be attained in CMO if save/restore errors did not occur.

Fig. 7 shows a comparison of the learning progress of the three algorithms for the planning scenario within the `war-sim` simulator. We report mean, tenth and ninetieth percentiles over five independent training runs. To provide additional context on how those training graphs are achieved, Fig. 8 shows histograms of the following performance

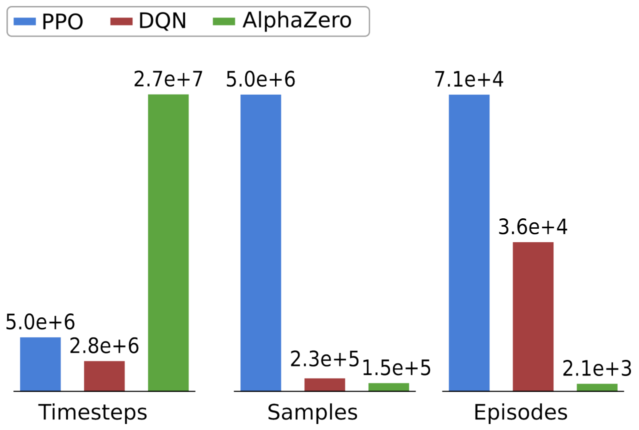

Fig. 8 Data requirements for the PPO, DQN, and AlphaZero algorithms in the planning scenario (`war-sim`).

metrics: 1) *timesteps*, defined as the total number of calls to the simulation environment; 2) *samples*, defined as the total number of collected training samples provided to the NN; 3) *episodes*, defined as the total number of complete episodes played. PPO executes the largest number of episodes and collects the large number of samples. DQN executes approximately half of PPO's episodes but collects an OOM less samples. AlphaZero executes an OOM less episodes, with an OOM less samples per episodes, but requires an OOM larger number of simulator steps.

To illustrate the outcomes of the training process, Fig. 9 shows a heat-map of the valuation function for the planning scenario obtained with AlphaZero. Brighter cell colour indicates higher levels of the value function. The valuation function guides the agent along the shortest path that circumvents the radar to the mission target. As the agent approaches the target, the cell's colours become brighter, while the cells just below the starting point are dark, corresponding to a path that likely leads to no return (i.e., shot down by the SAM battery). Finally, Fig. 10 shows a near optimal trajectory chosen by the AlphaZero algorithm after full training.

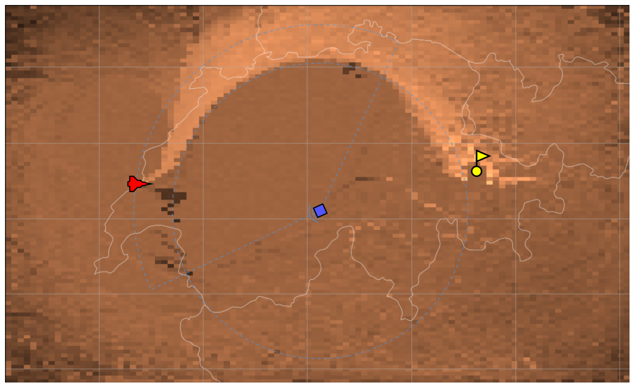

Fig. 9 Value map for the planning scenario after full training on AlphaZero (`war-sim`).

A similar analysis has been performed for the combat scenario. As before we report mean, tenth and ninetieth percentiles over five independent training runs. The reward comparison in Fig. 11 shows that, even though AlphaZero

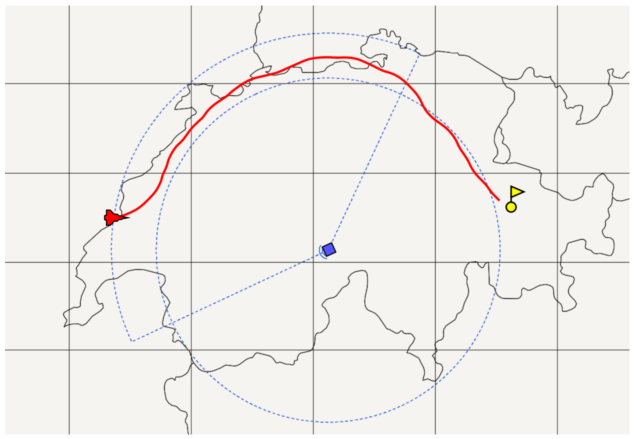

Fig. 10 Generated trajectory for the planning scenario after full training on AlphaZero (`war-sim`).

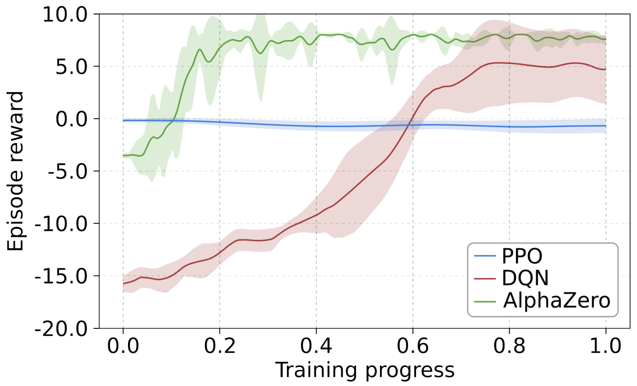

Fig. 11 Training progress in the combat scenario (`war-sim`).

shows best overall performance, DQN can reach good reward levels at an OOM computational effort below AlphaZero. The histograms in Fig. 12 show that data requirements in combat and planning scenarios are similar, indicating that those data consumption properties are rather driven by the choice of algorithm than the specific scenario.

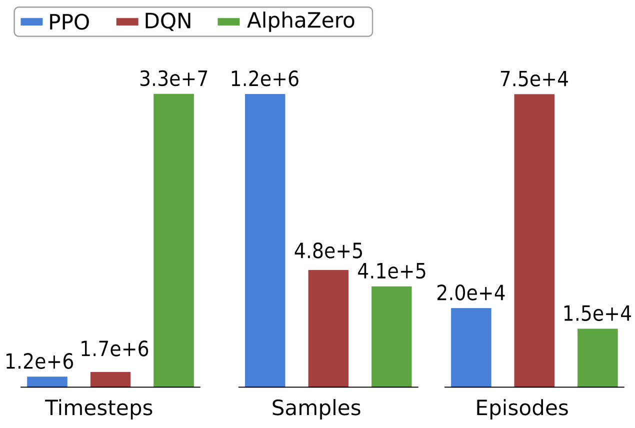

Fig. 12 Data requirements for the PPO, DQN and AlphaZero algorithms in the combat scenario (`war-sim`).

The optimal Blue and Red team strategies are not obvious in the combat scenario. The training of AlphaZero allows us to identify a realistic and near-optimal policy for the Red team that avoids the Blue teams' defence measures. The heat-map in Fig. 13 illustrates the outcome of training showing where the Red team should fire the first missile. Brighter cell colour indicates higher expected reward from launching the missile (towards SAM battery or defender). Fig. 14 shows a summary of an episode, where the Red team successfully accomplishes its mission. In this episode, the attacker first targets the defender aircraft. The latter replies with return fire but misses the attacker. Finally, the attacker fires another missile at the SAM battery to destroy it. In this case both targets are destroyed, and the maximum reward is attained.

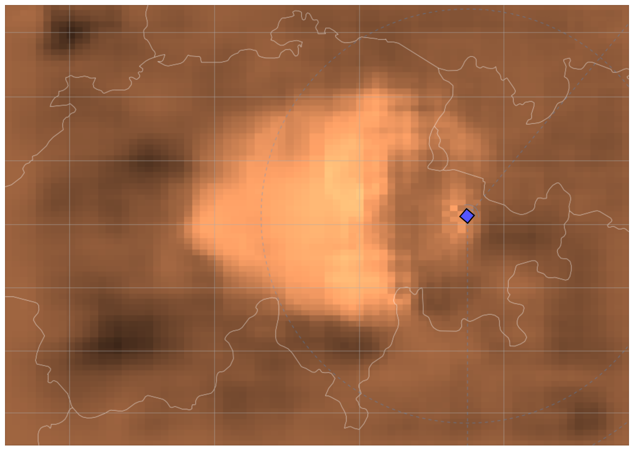

Fig. 13 Value map for the combat environment after full training on AlphaZero (`war-sim`).

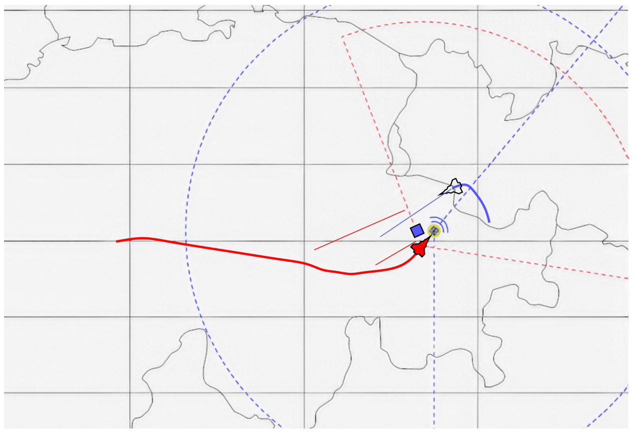

Fig. 14 An episode of the combat scenario after a full training on AlphaZero (`war-sim`).

VII. Conclusions

We investigated the application of RL to wargaming. From an RL perspective, wargames are complex environments posing several challenges because the state space and the branching factors are large, and even in simple scenarios a straightforward application of RL algorithms fail. We considered a popular wargaming simulator, CMO, along with its lightweight twin, `war-sim`, used to speed up the experiments. An extensive analysis of the features of existing Deep-RL algorithms led us to select PPO, DQN, and AlphaZero. Similarly, we proposed three demonstrative scenarios. With `war-sim`, AlphaZero succeeded in solving all the scenarios, while the same is not true for the CMO case. The complexity of CMO and its interface to Python, lead to crashes that prevents a stable learning. Still, in a *combat* scenario, we demonstrated that, when the environment is stable and a tailored algorithm implementation is used, RL can be employed to investigate CoAs. This can be translated to bigger and non-trivial scenarios to found unexpected and innovative solutions.

The results of our experiments also allow us to make general considerations on the behaviour of the PPO, DQN and AlphaZero algorithms when applied to wargaming. A model-based algorithm like AlphaZero tends to create few samples of better quality in a limited number of episodes, while the model-free PPO and DQN collect a lot of samples by playing many more episodes. This has been exemplified in a *planning* scenario, where MCTS prevailed in performance but a computational cost that is one OOM greater than PPO and MCTS: while this can be convenient with fast environments, it may not always be a viable option in slower simulators like CMO. DQN has a good trade-off between applicability and performance: being model-free, it does not require the slow calls for saving and restoring the game state, and at the same time shows promising results, at least in scenarios where the reward signal is not too sparse.

We should note here that the central assumption empowering RL is that the environment's transition probabilities remain fixed during the training. For a wargame, this means that if a specific action is taken in a particular state, then statistically, the outcomes will always be the same (irrespective of the past evolution of the game). However, this assumption is usually not satisfied in multi-party scenarios, which poses a significant limitation when applying RL to wargaming. If a learning agent is faced with multiple adversaries, from the learner's perspective, its environment will appear non-stationary because of the unseen actions of other players. Multi-party conflicts are commonly formalised as Markov Games and are the subject of multi-agent RL [46]. Exploring this direction is a necessary future work. Besides that, we will also continue to investigate how RL algorithms can be applied to wargaming, possibly using different simulators and defining more complex scenarios.

Appendices

A. Game Tree Search

Search. Search techniques constitute the classical approach to combinatorial two-player games (like chess). The main idea is to evaluate all possible actions of the current player, all possible replies, all replies to replies, and so forth. This presumes discrete state and action spaces and perfect information. Given the tree's exponential size, this only works for simple games. More commonly, the game tree is pruned at a specific planning horizon, and a valuation function evaluates the emerging leaf states. Min-max is a simple technique that reduces computational effort. For each state, only the player's best action (maximising the value of the state, `max`) and the opponent's best response (minimising the value of the position, `min`) are followed through planning. A substantial improvement is α/β-search, which further reduces the search effort. Min-max variations only work for small to moderate-size branching factors, and the performance depends strongly on the accuracy of the valuation function. They constituted the state-of-the-art method for many adversarial planning problems until the emergence of deep RL. The strength of deep RL compared to α/β-search with NN valuation in chess remains debated.

Monte Carlo Methods. Games with large branching factors constitute a major challenge for non-adaptive search methods. Stochastic environments and imperfect information add to this challenge since taking a single action in a given state yields varying outcomes, which inflates the effective branching factor. Although search algorithms can be extended to operate under randomness (see, e.g., the *Expect-min-max* algorithm of [47] and heuristic search methods as in [48, 49]) the increase in computational effort makes them often impractical. To avoid costly evaluations of expected values, Monte Carlo search methods have been introduced that provide estimates from a limited number of random samples. In a nutshell, the state is evaluated by trying every possible action several times and, recursively, from each generated state every possible action until the planning horizon is reached. This search can yield near-optimal policies independent of the state space size [50]. Yet, in the case of complex games, the number of tries and the planning horizon need to be so large that computation becomes impractical.

B. Reinforcement Learning

The primary conceptual step behind RL is the transition from artificial agents that play a fixed strategy to learning agents with adaptive strategies. To improve strategies, agents must take apparently sub-optimal actions to explore new CoAs. The trade-off between exploring new actions and exploring known lucrative actions is most evident in bandit algorithms.

Multi-Armed Bandits. The k-armed bandit, whose name originates from a gambler choosing from k slot machines, is a prototypical example of RL. It describes a simplified scenario where actions (called *arms*) do not influence the environment [31]. Consequently, instead of planning ahead, the k-armed bandit only optimises the immediate rewards. If applied to a wargame, this limitation means that the algorithm focuses on immediate results (e.g., maximising the number of destroyed enemy units) but ignores their long-term consequences (such as missing overall operations targets because insufficient ammunition is available). Formally, at each time step $t = 1, \ldots, T$, the agent chooses an action $a \in \{a_1, \ldots, a_k\}$ and receives a random reward $R_a = X_{a,t}$, where the random variables $X_{a,t}$ are independent with respect to a and independently and identically distributed with respect to t. Running through many episodes, the agent gathers the reward statistics of each arm. The exploitation-exploration dilemma manifests itself in the question of whether the agent should choose an arm that has been lucrative so far or try arms with high statistical uncertainty in the hope of unveiling even higher rewards. The UCB1 policy is employed frequently in the control of exploration and exploitation. It maximises a trade-off between expected reward and confidence

$$UCB1_a = \bar{R}_a + wc_{n,n_a} \text{ with } c_{n,l} = \sqrt{\frac{2\ln(n)}{l}}\,, \tag{3}$$

where n_a is the number of times a has been played so far. The average reward $\bar{R}_a$ emphasises exploitation of the currently best action, while the c_{n,n_a} encourages the exploration of high-variance actions. Their relative weight is determined by a problem-specific hyper-parameter w. For large classes of reward distributions, the UCB1 policy is nearly optimal.

Monte Carlo Methods. Monte Carlo methods approach the RL problem by taking random samples of actions and environment transitions throughout many learning episodes [41]. Monte Carlo methods can operate in a model-free setting, learning solely from experiences or through targeted environment interaction in the model-based case. A recurrent concept is that a table is maintained during the training process that contains estimates of the Q-value of state-action pairs (s_t, a_t) in view of the episode's expected reward. After completing each training episode, the table is updated following the standard Monte Carlo updating rule:

$$\hat{Q}_{new}(s_t, a_t) \leftarrow \hat{Q}_{old}(s_t, a_t) + \alpha \left[\sum_{k=t}^{T} \gamma^{k-t} R_{a_k,k} - \hat{Q}_{old}(s_t, a_t) \right], \tag{4}$$

where the summation in the right-hand side stands for the new Monte Carlo estimate and the step-size parameter α weights the relevance of the currently held estimate versus the new one. A main disadvantage of this rule is that updates can only be executed once an episode is complete. If Q-values could be updated during the episode, this would improve the selection of subsequent actions during the ongoing episode and boost algorithm efficiency. Temporal-difference methods update Q-values immediately after an action is executed. This implies that the episode's remaining return $\sum_{k=t}^{T} R_{a_k,k}$ must be estimated before the episode is complete. A common choice is to use a Bellman estimate, which is, e.g., employed in the celebrated Q-learning algorithm [32].

Q-learning. The Q-table is updated after each decision at each time step t. In Q-learning, the agent proceeds by an iteration of the following steps. *i)* The agent chooses the best action a_t from the table, *ii)* the agent observes the reward $R_{a_t,t}$, and *iii)* updates the estimates in the Q-table according to the modified updating rule:

$$\hat{Q}_{new}(s_t, a_t) \leftarrow \hat{Q}_{old}(s_t, a_t) + \alpha \left[R_{a_t,t} + \gamma \max_a \hat{Q}_{old}(s_{t+1}, a) - \hat{Q}_{old}(s_t, a_t) \right] . \tag{5}$$

The updating rule contains two distinct types of estimates. First, Q is estimated via a Monte Carlo average $\hat{Q}_{new/old}$. Second, the term $R_{a_t,t} + \gamma \max_a \hat{Q}_{old}(s_{t+1}, a)$ is a (Bellman-type) estimate of the total expected return for the ongoing episode.

Monte Carlo Tree Search. MCTS is a class of model-based RL algorithms for approximating optimal policies in complex environments. For any RL environment, near-optimal policies can be constructed by a combination of Monte Carlo sampling and search [50], but for complex wargames, the size of the simulation often needs to be so large that the computation becomes impractical. To increase efficiency, MCTS creates a problem-specific, restricted and asymmetric decision tree instead of "brute force Monte Carlo" [33, 51]. In MCTS, each node holds the performance statistics of branches starting from this node, and tree growth is guided by a dedicated tree policy (that balances between the incorporation of new nodes and the simulation of existing lines). The learning process incorporates iterative improvements between tree policy and state valuation [52]: 1) *selection*, that is MCTS identifies the most relevant leaf node from the current planning tree; 2) *expansion*, that is a new leaf node is added to the tree at the selected node; 3) *simulation*, that is a valuation is executed at the new node; and 4) *back-propagation*, that is the simulation result improves the tree policy by updating the reward statistics of all tree nodes. Frequently, individual nodes employ the UCB1 bandit policy to guide the construction of the planning tree [33]. The resulting algorithm, called UCT, is guaranteed to identify the optimal policy if enough resources are granted.

C. Deep Reinforcement Learning

Although many classical RL methods have convergence guarantees, the computational effort to achieve convergence in games is often horrendous. Even the maintenance of a table of state value estimates is impossible for complex games (with approximately 10^{47} states for chess and 10^{170} for Go). One of the main aims of introducing deep learning into RL is to reduce computational effort by using the generalisation capability of abstract representations. Roughly speaking, instead of evaluating a large number of nearby states (by expensive simulation or ad-hoc valuation), one might evaluate a (deep) NN that has been trained on a relatively small number of accurate samples. The price to pay is that convergence guarantees are lost as NN training can get stuck in a poor local optimum. An important challenge in deep RL is to design the training process to ensure that the agent's performance continues to increase.

Deep Q-learning. Deep Q-network (DQN) is a model-free RL algorithm that extends classical Q-learning representing Q-values by a (deep) NN. This allows us to learn abstract state representations and to extract the state's most relevant features in view of subsequent rewards. While the size of state and action spaces might be gigantic, their NN representation often remains manageable because many details of the state are irrelevant in view of forthcoming rewards. Instead of maintaining and updating a Q-table, DQN solely operates on the NN parameters, updating them intra-episode (temporal-difference learning). The updating rule is gradient-based, but it takes inspiration from the Bellman targets $R_{a_t,t} + \gamma \max_a \hat{Q}_{old}(s_{t+1}, a)$ of Q-learning to estimate the impact of a new sample to $\hat{Q}$,

$$w_{new} \leftarrow w_{old} + \alpha \nabla_w \hat{Q}_{old}(s_{t+1}, a) \cdot \left[R_{a_t,t} + \gamma \max_a \hat{Q}_{old}(s_{t+1}, a) - \hat{Q}_{old}(s_t, a_t) \right] . \tag{6}$$

Unlike supervised learning, where learning targets are fixed, the Bellman targets depend on the NN parameters. This would imply that the NN is trained on a sequence of targets that change in each time step (resulting in instability and divergence [53]). To address this shortcoming, in DQN, two instances of the NN are used, each equipped with its own set of parameters. The *value NN* provides estimates of $\hat{Q}_{old}$ and is trained at each time step on the Bellman targets. The *target NN* provides the action values for the computation of Bellman targets. The parameters of the target NN are updated asynchronously, setting them to the parameters of the value NN after a given number of episodes. DQN employs a replay memory buffer [54], which stores all experienced transitions $(s_t, a_t, r_{t+1}, s_{t+1})$. This allows batched gradient descent algorithms to be used for NN training. Removing the dependence of samples on the current weights and experience replay remediates the mentioned instability. Many variations of the DQN design have appeared in the literature, where recent architectures outperform the original design in terms of stability and sample efficiency [53].

Proximal Policy Gradient. Policy gradient algorithms are model-free Monte-Carlo methods that operate on a set of parametrised policies rather than on Q-values directly. They compute Monte Carlo estimates of policy gradients and subsequently update the policy parameters using stochastic gradient descent. The prototypical policy gradient method, REINFORCE [41], uses a gradient estimator of the form:

$$\hat{\mathbb{E}}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t | s_t) \sum_{k=t}^{T} R_{a_k,k} \right] , \tag{7}$$

where π_θ denotes the policy (parametrised by θ) and $\hat{\mathbb{E}}_{\pi_\theta}[\cdot]$ is a Monte Carlo estimate of expected return when following π_θ. REINFORCE updates parameters retrospectively for each step t once the episode is complete:

$$\theta_{new} \leftarrow \theta_{old} + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{k=t}^{T} R_{a_k,k} \,. \tag{8}$$

As for tabular Monte Carlo methods, temporal difference techniques are available to update parameters intra-episode. In the updating rule, this leads to a replacement of the episode's remaining return by an estimator of advantage $\hat{A}_t$. Nonetheless, temporal difference policy gradient methods have relatively poor data efficiency, and their sensitivity to step size (in gradient descent) remains an important source of instability. Performing multiple steps of optimisation on the REINFORCE loss function $\text{Loss}^{PG} = \hat{\mathbb{E}}_{\pi_\theta}\left[\log \pi_\theta(a_t|s_t) \sum_{k=t}^{T} R_{a_k,k}\right]$ on the same trajectory empirically leads to destructively large policy updates. *Trust Region Policy Optimisation* (TRPO [55]) employs a surrogate loss function:

$$\text{Loss}^{TRPO} = \hat{\mathbb{E}}_{\pi_\theta}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] , \tag{9}$$

which is optimised subject to a constraint on the discrepancy of new and old policies (based on their expected KL-divergence) to avoid overly large gradients. PPO [56] employs a clipping procedure on the policy ratio in TRPO's objective function to ensure conservative parameter updates. This simplifies the algorithm by removing the KL penalty and the need to make adaptive updates.

AlphaZero. AlphaZero is an MCTS variant [57, 58], where the original UCT design is enhanced by using NNs. Typically, the NNs serve the purpose of *1)* guiding the tree construction and *2)* providing accurate evaluation of leaf nodes. Employing NNs on top of UCT has two main advantages. First NNs provide faster state valuation as compared to UCT evaluation. Second, they provide generalisation capability: to evaluate "similar" states UCT can only execute costly simulations, whereas a trained NN might "recognise" the similarity for valuation. The AlphaZero training process contains an alternation of mutual improvements of the MCTS and the NN components. The algorithm starts (either by supervising pre-training or) by an execution of the UCT algorithm. Once enough samples are available, NN representations of tree policy and a valuation function will be trained on UCT targets. Specifically, concerning point *1)* a tree-policy NN is trained to imitate the average UCT action at the root using:

$$\text{Loss}_{TPT} = -\sum_a \frac{n_a}{n} \log \pi^{NN}(a|s) \,, \tag{10}$$

where n_a is the number of times a has been played from s and n is the total number of simulations. In the subsequent iteration, an NN term is added to the UCB1 tree policy, guiding tree search towards stronger courses of action,

$$NUCB1_a = UCB1_a + w\frac{\pi^{NN}(a|s)}{n_a + 1}\,, \tag{11}$$

where $w \sim \sqrt{n}$ is a hyper-parameter that weights the contributions of $UCB1$ and the NN. Concerning point *2)*, the value network reduces search depth and avoids inaccurate rollout-based value estimation. As before tree search provides value samples z by collecting rewards over training episodes. The value NN is trained to predict the obtained values via $\text{Loss}_V = -(z - V^{NN}(s))^2$. To regularise prediction and accelerate training, it is common to cover *1)* and *2)* simultaneously by a multitask network, whose loss is simply the sum of Loss_V and Loss_{TPT}.

D. On the Choice of Algorithm for Wargames

The choice of algorithm can be delicate due to large variations in types of wargames, varying objectives, and the large number of available algorithms. Training agents that combine real-time control of individual units (troop level) and global mission planning (commander level) poses a challenge because these tasks lead to different system requirements, choices of algorithms, training settings, among others. A rough orientation is as follows.

Search. Traditional search techniques apply to discrete games with a clear underlying tree structure. Prerequisites are the availability of computational resources for investigating a meaningful portion of the game tree and a sufficiently accurate function for evaluating leaf nodes. Indicators of game tree size (such as the size of state and action spaces and branching factors) should be of moderate size. In the case of chess (branching factor $\sim$ 30) α/β-search variations easily achieve superhuman strengths, but the valuation function is the result of decades of research [59]. α/β-search performs strongly on the Swiss wargame *New Techno War*[¶]. Increasing branching factors typically lead to a deterioration of search, e.g., with only moderate performance for Go (branching factor $\sim$ 361). Monte Carlo search techniques address situations when deterministic search is unfeasible, e.g., in the presence of large branching factors (that might emerge from the "fog of war"). Compared to deterministic search, Monte Carlo methods are typically less accurate but can cope with larger complexity.

To apply search techniques in the presence of continuous variables some form of discretisation is required. If this is achieved by enforcing that variables are located on specified grids, the grids' mesh sizes will be delicate hyper-parameters. If a grid is too coarse, the methods might miss out reward signals; if it is too fine, the computational effort becomes unmanageable.

Traditional search algorithms might be combined with deep learning components (NNs) to represent the acquired

[¶]`https://deftech.ch/wargaming`.

knowledge (valuation function) and trained using deep RL, see below. For chess, combining α/β-search with NNs [60] leads to a significant performance boost [61]. The performance comparison to AlphaZero remains debated.

Supervised Learning. Supervised learning systems are limited by the availability and the quality of data; in (the practically rare) case that sufficient data is available, supervised learning copes well even with very complex wargames. Supervised learning agents achieve the level of a strong human player in games like Go and Starcraft 2 [21, 62]. As a rule of thumb, supervised learning systems are easier to train and are less resource-consuming than tabula rasa deep RL. If appropriate training data is available it is advisable to attempt to train a supervised deep learning system. Deep RL might be applied on top of the pre-trained system to increase performance.

Classical Reinforcement Learning. Classical RL techniques often have strong convergence guarantees. Their main drawback is that the computational effort often becomes impractical even for simple wargames. Tabular RL methods (including Monte Carlo methods and Q-learning in particular) require discrete state and action spaces. As for search, discretisation introduces hyper-parameters that are hard to control. More broadly, solving wargames exactly is often unfeasible because it requires a statistical assessment of the entire game tree. In similar vein, the convergence of general approximate solution methods (such as value iteration or gradient-based optimisation) can be slow. Approximate solution methods emerge naturally in the context of continuous variables. Policy gradient methods assume a (continuous) policy distribution, updating it using gradient methods as information becomes available. This avoids updates of large action-value tables but requires appropriate priors and learning schemes for policies. Appropriate prior construction can often benefit learning [31].

Deep Reinforcement Learning. Introducing deep learning into RL algorithms serves at least two purposes. First, deep learning provides abstraction and generalisation capabilities. In the case of states, abstraction allows the agent to focus on the state's relevant content in view of the subsequent rewards. In the case of actions abstraction means that actions can be grouped into sub-routines that implement relevant operations instead of a grid-type discretisation. Second, costly simulations might be replaced by the evaluation of a trained function approximator (a NN), which allows to exploit state similarities and abstract symmetries instead of brute force computation. As for classical RL derivatives of tabular methods fit with small/discrete games, whereas policy-gradient methods offer a practical way to deal with large/continuous action spaces. Behind this stands the fact that policy methods optimise entire probability distributions rather than recording statistics of action-value performance. Regarding DQN it should be said that while it performs well on the Arcade Learning Environments (Atari games with discrete action spaces), it fails on many simple problems for reasons that are often hard to retrace. On continuous control benchmarks such as those in OpenAI Gym [63], PPO-variants often show better performance and better learning stability [56]. Another interesting benchmark is that PPO-trained agents outperform human experts in time-critical control problems such as drone racing [16] or car racing simulations [17]. For planning problems, planning-based methods (such as combined NN and search algorithms) learn stronger policies faster and achieve higher performance scores than the best model-free approaches. It is a common empirical observation that the performance of Deep Q-Network (DQN) and Proximal Policy Optimisation (PPO) deteriorates quickly on complex planning problems (such as Go or chess), particularly when rewards are sparse. Nowadays, neural MCTS variants outperform model-free methods on numerous benchmarks, see the performance statistics in the planning track of the General Video Game AI competition [18, 64], and even control tasks such as the Arcade Learning Environments [65]. Neural MCTS variants also commonly have higher generalisation capability than model-free methods, i.e., they perform better on previously unseen scenarios (given some high-level description, such as the rules of the wargame and the mission targets), where "almost every general game playing program today uses some version of MCTS" [66]. Last but not least, NN systems can be combined with memory cells to transport contextual information through consecutive sequences of states making it available for strategic planning of CoAs. In the context of wargames the Long-Short-Term-Memory (LSTM) architecture of [67] stands behind the tremendous success of artificial agents in the games Dota 2 [20] and Starcraft 2 [21]. LSTM architectures strike an efficient balance between discrete multi-period planning and continuous control without significant look-ahead, see [20, 21]. For modern wargames that combine control and strategic planning, this makes LSTM architectures a natural choice. An additional advantage is that LSTM and NN architectures can be combined in a modular way, allowing for the creation of learning hierarchies and individual training and assessment of components.

Training, Subgoals and Hierarchies. If no data is available for supervised pre-training reward shaping techniques can be an alternative method to facilitate the learning process. To guide the agent towards and overall mission target pre-specified sub-goals might be introduced. Sub-goals and reward signals might be provided based on human knowledge (e.g., imposing a penalty for losing a unit), or learned automatically [68, 69]. In real-world operations, the complexity of defence scenarios is managed by a hierarchical organisation of the decision-making process. Low-level manoeuvring decisions are made by individual units, while abstract mission planning objectives are set at higher hierarchy levels. This provides higher efficiency in training and execution by exploiting operational similarities between individual actors. Mimicking real-world hierarchies with hierarchical deep RL system decomposes the overall policy into problem-specific sub-policies [70]. These sub-policies are equipped with task-specific architectures and trained in specific ways, providing a similar gain in efficiency in wargames [21, 22]. The AlphaStar agent employs a hierarchy of NNs, where low-level networks take responsibility of unit control, whereas an LSTM module at higher hierarchy level plans CoAs to achieve overall mission targets. For complex wargames, tabula rasa learning without any form of human guidance, be it in the form of demonstrations or reward shaping, is currently elusive.

Software Architecture. The complexity of software architecture is an important consideration when choosing an algorithm. While many model-free algorithms are available in open-source repositories (DQN and PPO are, e.g., readily implemented within the RLlib package for Python [45] and within standardised RL libraries (e.g., Gymnasium [42]), model-based systems will often demand for a custom development effort. This affects not only the development of the algorithm core but also suitable parallelisation, dedicated error-catching mechanisms and handling many concurrent simulation processes. The latter points add significantly to the effort of creating model-based deep RL systems for wargames. Overall, this makes model-free approaches the primary practical choice for many wargames. Model-based systems target higher reward performance at the cost of complex infrastructure and considerable custom development effort. Given a computation and time budget, we have observed that model-free methods can yield better results because latency in environment transitions and instantiation erodes the advantage of efficient learning. Another point is that model-free agents are fast in execution (often, decisions are made at superhuman frequencies), while planning agents are several orders of magnitude slower (e.g., requiring seconds to select an action).

References

[1] UK Ministry of Defense, *Wargaming Handbook - Development, Concepts and Doctrine Centre*, 2017. URL `https://assets.publishing.service.gov.uk/media/5a82e90d40f0b6230269d575/doctrine_uk_wargaming_handbook.pdf`, (accessed July 4, 2025).

[2] Rinaudo, C. H., Leonard, W. B., Hopson, J. E., Coumbe, T. R., Pettitt, J. A., and Darken, C., "Applying Deep Reinforcement Learning to Train AI Agents in a Wargaming Framework," *SoutheastCon 2024*, 2024, pp. 1131–1136. https://doi.org/10.1109/SoutheastCon52093.2024.10500249.

[3] Castro, G., Heradio, R., and Cerrada, C., "Automated Support for Battle Decision Making: A Systematic Literature Review," *Military Operations Research*, Vol. 27, 2022, pp. 5–23. URL https://www.jstor.org/stable/27182470.

[4] Davis, P., "Distributed interactive simulation in the evolution of DoD warfare modeling and simulation," *Proceedings of the IEEE*, Vol. 83, No. 8, 1995, pp. 1138–1155. https://doi.org/10.1109/5.400454.

[5] Rashid, A. B., Kausik, A. K., Al Hassan Sunny, A., and Bappy, M. H., "Artificial Intelligence in the Military: An Overview of the Capabilities, Applications, and Challenges," *International Journal of Intelligent Systems*, Vol. 2023, No. 1, 2023, p. 8676366. https://doi.org/10.1155/2023/8676366.

[6] Pournelle, P., "The need for cooperation between wargaming and modeling & simulation for examining Cyber, Space, Electronic Warfare, and other topics," *The Journal of Defense Modeling and Simulation*, Vol. 21, No. 4, 2024, pp. 359–362. https://doi.org/10.1177/15485129221118100.

[7] Lin-Greenberg, E., Pauly, R. B., and Schneider, J. G., "Wargaming for International Relations research," *European Journal of International Relations*, Vol. 28, No. 1, 2022, pp. 83–109. https://doi.org/10.1177/13540661211064090.

[8] Davis, P. K., and Bracken, P., "Artificial intelligence for wargaming and modeling," *The Journal of Defense Modeling and Simulation*, Vol. 22, No. 1, 2025, pp. 25–40. https://doi.org/10.1177/15485129211073126.

[9] Goodman, J., Risi, S., and Lucas, S., "AI and Wargaming," arXiv, 2020. URL https://arxiv.org/abs/2009.08922.

[10] BreakingDefense, 2020. URL https://breakingdefense.com/2020/08/darpa-wants-wargame-ai-to-never-fight-fair, (accessed July 4, 2025).

[11] Dimitriu, A., Michaletzky, T. V., Remeli, V., and Tihanyi, V. R., "A Reinforcement Learning Approach to Military Simulations in Command: Modern Operations," *IEEE Access*, Vol. 12, 2024, pp. 77501–77513. https://doi.org/10.1109/ACCESS.2024.3406148.

[12] Andersen, P.-A., Goodwin, M., and Granmo, O.-C., "Towards safe and sustainable reinforcement learning for real-time strategy games," *Information Sciences*, Vol. 679, 2024, p. 120980. https://doi.org/10.1016/j.ins.2024.120980.

[13] Hastie, T., Tibshirani, R., and Friedman, J., *The Elements of Statistical Learning*, Springer New York, 2009. URL https://link.springer.com/book/10.1007/978-0-387-84858-7.

[14] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A., "Playing Atari with Deep Reinforcement Learning," arXiv, 2013. URL https://arxiv.org/abs/1312.5602.

[15] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D., "Human-level control through deep reinforcement learning," *Nature*, Vol. 518, No. 7540, 2015, pp. 529–533. https://doi.org/10.1038/nature14236.

[16] Kaufmann, E., Bauersfeld, L., Loquercio, A., Müller, M., Koltun, V., and Scaramuzza, D., "Champion-level drone racing using deep reinforcement learning," *Nature*, Vol. 620, No. 7976, 2023, p. 982–987. https://doi.org/10.1038/s41586-023-06419-4.

[17] Wurman, P. R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T. J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F., Gilpin, L., Khandelwal, P., Kompella, V., Lin, H., MacAlpine, P., Oller, D., Seno, T., Sherstan, C., Thomure, M. D., Aghabozorgi, H., Barrett, L., Douglas, R., Whitehead, D., Dürr, P., Stone, P., Spranger, M., and Kitano, H., "Outracing champion Gran Turismo drivers with deep reinforcement learning," *Nature*, Vol. 602, No. 7896, 2022, p. 223–228. https://doi.org/10.1038/s41586-021-04357-7.

[18] Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., and Lucas, S., "General Video Game AI: Competition, Challenges and Opportunities," *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30, No. 1, 2016. https://doi.org/10.1609/aaai.v30i1.9869.

[19] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D., "A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play," *Science*, Vol. 362, No. 6419, 2018, pp. 1140–1144. https://doi.org/10.1126/science.aar6404.

[20] Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., d. O. Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S., "Dota 2 with Large Scale Deep Reinforcement Learning," arXiv, 2019. URL https://arxiv.org/abs/1912.06680.

[21] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D., "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, Vol. 575, No. 7782, 2019, pp. 350–354. https://doi.org/10.1038/s41586-019-1724-z.

[22] Selmonaj, A., Szehr, O., Del Rio, G., Antonucci, A., Schneider, A., and Rüegsegger, M., "Hierarchical Multi-Agent Reinforcement Learning for Air Combat Maneuvering," *Proceedings of the 2023 International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2023, pp. 1031–1038. https://doi.org/10.1109/ICMLA58977.2023.00153.

[23] Zhu, J., Kuang, M., Zhou, W., Shi, H., Zhu, J., and Han, X., "Mastering air combat game with deep reinforcement learning," *Defence Technology*, Vol. 34, 2024, pp. 295–312. https://doi.org/10.1016/j.dt.2023.08.019.

[24] Jeong, H., Hassani, H., Morari, M., Lee, D. D., and Pappas, G. J., "Deep Reinforcement Learning for Active Target Tracking," *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 1825–1831. https://doi.org/10.1109/ICRA48506.2021.9561258.

[25] Xia, J., Luo, Y., Liu, Z., Zhang, Y., Shi, H., and Liu, Z., "Cooperative multi-target hunting by unmanned surface vehicles based on multi-agent reinforcement learning," *Defence Technology*, Vol. 29, 2023, pp. 80–94. https://doi.org/10.1016/j.dt.2022.09.014.

[26] Shang, T., Han, K., Ma, J., and Mao, M., "Research on Self-Gaming Training Method of Wargame Based on Deep Reinforcement Learning," *Proceedings of the 2019 International Conference on Artificial Intelligence and Computer Science*, Association for Computing Machinery, New York, NY, USA, 2019, p. 251–254. https://doi.org/10.1145/3349341.3349411.

[27] Shang, T., Ma, J., Han, K., and Yu, Y., "Research on Game Problem Under Incomplete Information Condition Based on Deep Reinforcement Learning," *Proceedings of the 2019 International Conference on Artificial Intelligence and Computer Science*, Association for Computing Machinery, New York, NY, USA, 2019, p. 255–257. https://doi.org/10.1145/3349341.3349412.

[28] Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., and Hester, T., "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis," *Machine Learning*, Vol. 110, No. 9, 2021, pp. 2419–2468. https://doi.org/10.1007/s10994-021-05961-4.

[29] Albert, M. H., Nowakowski, R. J., and Wolfe, D., *Lessons in Play: An Introduction to Combinatorial Game Theory*, CRC press, 2007.

[30] Fudenberg, D., and Tirole, J., *Game Theory*, MIT press, 1993. URL https://mitpress.mit.edu/9780262061414/game-theory/.

[31] Sutton, R. S., and Barto, A. G., *Reinforcement learning: An introduction*, MIT press, 2018.

[32] Watkins, C. J. C. H., and Dayan, P., "Technical Note: Q-learning," *Machine Learning*, 1992, pp. 279–292. https://doi.org/10.1023/A:1022676722315.

[33] Kocsis, L., and Szepesvári, C., "Bandit Based Monte-Carlo Planning," *17th European Conference on Machine Learning (ECML 2006)*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 282–293. https://doi.org/10.1007/11871842_29.

[34] Dong, L., Li, N., Yuan, H., and Gong, G., "Accelerating wargaming reinforcement learning by dynamic multi-demonstrator ensemble," *Information Sciences*, Vol. 648, 2023, p. 119534. https://doi.org/10.1016/j.ins.2023.119534.

[35] Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., *Building expert systems*, Addison-Wesley Longman Publishing Co., Inc., USA, 1983. URL https://dl.acm.org/doi/abs/10.5555/6123.

[36] Balduzzi, D., Garnelo, M., Bachrach, Y., Czarnecki, W., Perolat, J., Jaderberg, M., and Graepel, T., "Open-ended Learning in Symmetric Zero-sum Games," *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97, PMLR, 2019, pp. 434–443. https://doi.org/10.48550/arXiv.1901.08106.

[37] Hester, T., and Stone, P., "TEXPLORE: real-time sample-efficient reinforcement learning for robots," *Machine Learning*, Vol. 90, No. 3, 2012, pp. 385–429. https://doi.org/10.1007/s10994-012-5322-7.

[38] Hung, C.-C., Lillicrap, T., Abramson, J., Wu, Y., Mirza, M., Carnevale, F., Ahuja, A., and Wayne, G., "Optimizing Agent Behavior over Long Time Scales by Transporting Value," *Nature Communications*, Vol. 10, 2019, p. 5223. https://doi.org/10.1038/s41467-019-13073-w.

[39] Sommer, M., Rüegsegger, M., Del Rio, G., and Szehr, O., "Deep Self-optimizing Artificial Intelligence for Tactical Analysis, Training and Optimization," , 2021. STO-MP-SAS-OCS-ORA-2021, Nato, Science and Technology Organization.

[40] Melnikov, A. A., Makmal, A., and Briegel, H. J., "Benchmarking Projective Simulation in Navigation Problems," *IEEE Access*, Vol. 6, 2018, pp. 64639–64648. https://doi.org/10.1109/ACCESS.2018.2876494.

[41] Sutton, R. S., "Learning to predict by the methods of temporal differences," *Machine learning*, Vol. 3, No. 1, 1988, pp. 9–44. https://doi.org/10.1007/BF00115009.

[42] Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G. d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Shen, A. T. J., and Younis, O. G., "Gymnasium," , Mar. 2023. URL https://zenodo.org/record/8127025, (accessed July 4, 2025).

[43] Met Office, *Cartopy: a cartographic Python library with a Matplotlib interface*, 2010 - 2017. URL http://scitools.org.uk/cartopy, (accessed July 4, 2025).

[44] Karney, C. F. F., "Algorithms for geodesics," *Journal of Geodesy*, Vol. 87, No. 1, 2013, pp. 43–55. https://doi.org/10.1007/s00190-012-0578-z.

[45] Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I., "RLlib: Abstractions for Distributed Reinforcement Learning," *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 80, PMLR, 2018, pp. 3059–3068. URL https://proceedings.mlr.press/v80/liang18b.html.

[46] Gronauer, S., and Diepold, K., "Multi-agent deep reinforcement learning: a survey," *Artificial Intelligence Review*, Vol. 55, No. 2, 2021, pp. 895–943. https://doi.org/10.1007/s10462-021-09996-w.

[47] Michie, D., "Game-Playing and Game-Learning Automata," *Advances in Programming and Non-Numerical Computation*, Pergamon, 1966, pp. 183–200. https://doi.org/10.1016/B978-0-08-011356-2.50011-2.

[48] Dean, T., Kaelbling, L. P., Kirman, J., and Nicholson, A., "Planning under time constraints in stochastic domains," *Artificial Intelligence*, Vol. 76, No. 1-2, 1995, pp. 35–74. https://doi.org/10.1016/0004-3702(94)00086-G.

[49] Hansen, E. A., and Zilberstein, S., "LAO*: A heuristic search algorithm that finds solutions with loops," *Artificial Intelligence*, Vol. 129, No. 1-2, 2001, pp. 35–62. https://doi.org/10.1016/S0004-3702(01)00106-0.

[50] Kearns, M., Mansour, Y., and Ng, A. Y., "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes," *Machine Learning*, Vol. 49, No. 2/3, 2002, pp. 193–208. https://doi.org/10.1023/A:1017932429737.

[51] Chang, H. S., Fu, M. C., Hu, J., and Marcus, S. I., "An Adaptive Sampling Algorithm for Solving Markov Decision Processes," *Operations Research*, Vol. 53, No. 1, 2005, pp. 126–139. https://doi.org/10.1287/opre.1040.0145.

[52] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S., "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, 2012, pp. 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810.

[53] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D., "Rainbow: Combining improvements in deep reinforcement learning," *32nd AAAI Conference on Artificial Intelligence*, AAAI Press, 2018, pp. 3215–3222. URL https://dl.acm.org/doi/10.5555/3504035.3504428.

[54] Lin, L.-J., "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, Vol. 8, No. 3-4, 1992, pp. 293–321. https://doi.org/10.1007/BF00992699.

[55] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P., "Trust Region Policy Optimization," *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and D. Blei, PMLR, Lille, France, 2015, pp. 1889–1897. URL https://proceedings.mlr.press/v37/schulman15.html.

[56] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O., "Proximal Policy Optimization Algorithms," ArXiv, 2017. https://doi.org/10.48550/arXiv.1707.06347.

[57] Anthony, T., Tian, Z., and Barber, D., "Thinking fast and slow with deep learning and tree search," *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Curran Associates Inc., 2017, p. 5366–5376. URL https://dl.acm.org/doi/10.5555/3295222.3295288.

[58] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., and Bolton, A., "Mastering the game of Go without human knowledge," *Nature*, Vol. 550, No. 7676, 2017, pp. 354–359. https://doi.org/10.1038/nature24270.

[59] Newborn, M., *Computer Chess*, ACM Monograph Series, 2014.

[60] Stockfish, N., 2020. URL https://stockfishchess.org/blog/2020/introducing-nnue-evaluation, (accessed July 4, 2025).

[61] Klein, D., "Neural Networks for Chess," arXiv, 2022. https://doi.org/10.48550/arXiv.2209.01506.

[62] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D., "Mastering the game of Go with deep neural networks and tree search," *Nature*, Vol. 529, 2016, pp. 484–503. https://doi.org/10.1038/nature16961.

[63] Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P., "Benchmarking deep reinforcement learning for continuous control," *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, JMLR.org, 2016, p. 1329–1338. URL https://dl.acm.org/doi/10.5555/3045390.3045531.

[64] Torrado, R. R., Bontrager, P., Togelius, J., Liu, J., and Perez-Liebana, D., "Deep Reinforcement Learning for General Video Game AI," *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE Press, 2018, p. 1–8. https://doi.org/10.1109/CIG.2018.8490422.

[65] Guo, X., Singh, S., Lee, H., Lewis, R., and Wang, X., "Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning," *Advances in Neural Information Processing Systems*, Vol. 27, Curran Associates, Inc., 2014. URL https://papers.nips.cc/paper_files/paper/2014/hash/88bf0c64edabeeb913c378227beef8f9-Abstract.html.

[66] Genesereth, M., and Björnsson, Y., "The International General Game Playing Competition," *AI Magazine*, Vol. 34, No. 2, 2013, p. 107. https://doi.org/10.1609/aimag.v34i2.2475.

[67] Hochreiter, S., and Schmidhuber, J., "Long Short-Term Memory," *Neural Computation*, Vol. 9, No. 8, 1997, p. 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735.

[68] Schmidhuber, J., "Learning to generate subgoals for action sequences," *IJCNN-91-Seattle International Joint Conference on Neural Networks*, Vol. 2, 1991, p. 453. https://doi.org/10.1109/IJCNN.1991.155375.

[69] Schmidhuber, J., and Wahnsiedler, R., "Planning Simple Trajectories Using Neural Subgoal Generators," *From Animals to Animats 2*, The MIT Press, 1993, p. 195–201. https://doi.org/10.7551/mitpress/3116.003.0027.

[70] Bakker, B., and Schmidhuber, J., “Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization,” *Proceedings of the 8th Conference on Intelligent Autonomous Systems*, 2004, pp. 438–445.