Reinforcement Learning in Strategy-Based and Atari Games: A Review of Google DeepMind's Innovations

Abdelrhman Shaheen*, Anas Badr*, Ali Abohendy*, Hatem Alsaadawy*, Nadine Alsayad*, Ehab H. El-Shazly*†
*Egypt Japan University of Science and Technology (E-JUST), Alexandria, Egypt
{abdelrhman.shaheen, anas.badr, ali.abohendy, hatem.alsaadawy, nadine.alsayad, ehab.elshazly}@ejust.edu.eg

†Radiation Engineering Department, The National Center for Radiation Research and Technology,
Egyptian Atomic Energy Authority, Egypt

***Abstract*—Reinforcement Learning (Rl) has been widely used in many applications, one of these applications is the field of gaming, which is considered a very good training ground for AI models. From the innovations of Google DeepMind in this field using the reinforcement learning algorithms, including model-based, model-free, and deep Q-network approaches, AlphaGo, AlphaGo Zero, and MuZero. AlphaGo, the initial model, integrates supervised learning, reinforcement learning to achieve master in the game of Go, surpassing the performance of professional human players. AlphaGo Zero refines this approach by eliminating the dependency on human gameplay data, instead employing self-play to enhance learning efficiency and model performance. MuZero further extends these advancements by learning the underlying dynamics of game environments without explicit knowledge of the rules, achieving adaptability across many games, including complex Atari games. In this paper, we reviewed the importance of studying the applications of reinforcement Learning in Atari and strategy-based games, by discussing these three models, the key innovations of each model,and how the training process was done; then showing the challenges that every model faced, how they encounterd them, and how they improved the performance of the model. We also highlighted the advancements in the field of gaming, including the advancment in the three models, like the MiniZero and multi-agent models, showing the future direction for these advancements, and new models from Google DeepMind.**

***Index Terms*—Deep Reinforcement Learning, Google DeepMind, AlphaGo, AlphaGo Zero, MuZero, Atari Games, Go, Chess, Shogi,**

I. Introduction

Artificial Intelligence (AI) has revolutionized the gaming industry, both as a tool for creating intelligent in-game opponents and as a testing environment for advancing AI research. Games serve as an ideal environment for training and evaluating AI systems because they provide well-defined rules, measurable objectives, and diverse challenges [1] [2]. From simple puzzles to complex strategy games, AI research in gaming has pushed the boundaries of machine learning and reinforcement learning [3] [4] [5] [6]. Also, the benefits from such employment helped game developers to realize the power of AI methods to analyze large volumes of player data and optimize game designs [2].

Atari games, in particular, with their retro visuals and straightforward mechanics, offer a challenging yet accessible benchmark for testing AI algorithms. The simplicity of Atari games hides complexity: they require strategies that involve planning, adaptability, and fast decision-making, making them a good testing environment for evaluating AI's ability to learn and generalize [7] [8]. The development of AI in games has been a long journey, starting with rule-based systems and evolving into more sophisticated machine learning models [3] [4]. However, early machine learning models faced challenges in decision-making within interactive game environments [9] [10]. Traditional supervised and unsupervised models depended on static datasets without the ability to interact dynamically with the environment. To overcome this problem, game developers and researchers began to employ reinforcement learning (RL) [11] [12] [13] [14] [15]. Years later, deep learning (DL) showed remarkable success in computer vision and video games [16] [17] [18]. The combination of RL and DL resulted in Deep Reinforcement Learning (DRL), enabling agents to learn directly from high-dimensional sensory data [19] [20]. The first notable use of DRL was in Atari games [21] [8].

One of the leading companies in applying DRL to games is Google DeepMind. This company is widely recognized for developing advanced AI models, including those for games. Prior to AlphaGo, DeepMind had already contributed substantially to DRL through Atari benchmarks and novel algorithmic developments [8] [7]. For sequential decision-making, DeepMind combined RL techniques with neural networks to create models capable of memory and reasoning, such as Neural Turing Machines (NTMs) [22]. They then introduced the Deep Q-Network (DQN) algorithm, which combined Q-learning [14] [24] with deep neural networks, allowing agents to approximate the Q-function from high-dimensional inputs [23]. The DQN represented a major breakthrough in deep RL, as it was the first algorithm to learn directly from raw pixels and achieve human-level performance on Atari games [8].

To further enhance learning efficiency, DeepMind introduced experience replay [25] [26], which stabilized training by breaking correlations in sequential data. They then developed asynchronous methods such as A3C, which improved stability and scalability in DRL [27] [28] [19]. These developments enabled DeepMind to build AlphaGo, the first AI model to defeat the world champion in the game of Go, a historic milestone in both AI and reinforcement learning [56].

Our paper is similar to Shao et al. [30], as we discussed the developments that Google DeepMind made in developing AI models for games and the advancements that they made over the last years to develop the models and the future directions of implementing DRL in games; how this implementation helps in developing real life applications. The main contribution in our paper is the comprehensive details of the four models AlphaGo, AlphaGo Zero, AlphaZero, and MuZero, focusing on the key innovations for each model, how the training process was done, challenges that each model faced and the improvements that were made, and the performance benchmarks. Studying each one of these models in detail helps in understanding how RL was developed in games reaching the current state, by which it is now used in real life applications. Also we discussed the advancements in these four AI models, reaching to the future directions.

II. Background

There are a lot of related work that reviewed the reinforcement learning in strategy-based and Atari games. Arulkumaran et al. [31] serves as a foundational reference that outlines the evolution and state-of-the-art developments in DRL up to its publication. It also offers insights into how combining deep learning with reinforcement learning has led to significant advancements in areas such as game playing, robotics, and autonomous decision-making systems. Zhao et al. [32] surveys how DRL combines capabilities of deep learning with the decision-making processes of reinforcement learning, enabling systems to make control decisions directly from input images. It also analyzes the development of AlphaGo, and examines the algorithms and techniques that contributed to AlphaGo's success, providing insights into the integration of DRL in complex decision-making tasks. Tang et al. [33] surveys how AlphaGo marked a significant milestone by defeating human champions in the game of Go, and its architecture and training process; then delves into AlphaGo Zero. Shao et al. [30] categorize DRL methods into three primary approaches: value-based, policy gradient, and model-based algorithms, offering a comparative analysis of their techniques and properties.

Earlier surveys such as Kaelbling et al. [5] and Sutton & Barto [6] established the theoretical foundations of reinforcement learning, while subsequent works extended these frameworks to deep learning contexts [15] [20]. The Arcade Learning Environment (ALE) benchmark standardized Atari game evaluation for RL research [7], which was later expanded with the Deep Q-Network (DQN) [8], asynchronous actor-critic methods (A3C) [27] [28], and multi-step improvements such as Rainbow DQN [34]. These contributions highlighted the importance of balancing stability, sample efficiency, and exploration.

More recent algorithmic innovations include deterministic policy gradient methods for continuous control (DDPG) [35], trust region optimization techniques (TRPO) [36], and proximal policy optimization (PPO) [37], which have become standard tools for modern DRL. Distributed approaches such as IMPALA [38] demonstrated scalability in large environments, while multi-agent settings were explored by Lowe et al. [39] and Zhang et al. [40].

In terms of applications, DeepMind extended DRL beyond Atari and Go. AlphaStar achieved grandmaster level in StarCraft II [41], while Jaderberg et al. [42] explored cooperative multi-agent environments in Capture the Flag. These large-scale experiments highlight the versatility of DRL in handling both perfect-information board games and complex imperfect-information environments.

Several works also emphasized the open challenges in DRL. Exploration– exploitation balance and intrinsic motivation were studied in [43] [44]. Sample efficiency and generalization remain critical hurdles [45] [46]. Multi-agent coordination [47], sparse reward settings [48], and transfer learning [49] continue to be active areas of research. Comprehensive surveys such as Li [50] and Francois-Lavet et al. [51] summarize these advancements and challenges.

Reinforcement Learning (RL) is a key area of machine learning that focuses on learning through interaction with the environment [53]. In RL, an agent takes actions (A) in specific states (S) with the goal of maximizing the rewards (R) received from the environment. The foundations of RL can be traced back to 1911, when Thorndike introduced the Law of Effect, suggesting that actions leading to favorable outcomes are more likely to be repeated, while those causing discomfort are less likely to recur [52].
RL emulates the human learning process of trial and error. The agent receives positive rewards for beneficial actions and negative rewards for detrimental ones, enabling it to refine its policy function (π) —a strategy that dictates the best action to take in each state. That's said, for a given agent in state **s**, if it takes action **a**, then the immediate reward **r** can be modeled as $r(s,a) = \mathbb{E}[r_t \mid s = s_{t-1}, a = a_{t-1}]$. So for a full episode of T steps, the cumulative reward R can be modeled as $R = \sum_{t=1}^{T} r_t$.

To formally capture the interaction between an agent and its environment, reinforcement learning problems are often expressed within the framework of a **Markov Decision Process (MDP)**. The MDP provides a mathematical structure for modeling sequential decision-making, allowing us to define states, actions, rewards, and transitions in a rigorous way.

A. Markov Decision Process (MDP)

In reinforcement learning, the environment is often modeled as a **Markov Decision Process (MDP)**, which is defined as a tuple (S, A, P, R, γ), where:

- S is the set of states,
- A is the set of actions,
- P is the transition probability function,
- R is the reward function, and
- γ is the discount factor.

The MDP framework is grounded in **sequential decision-making**, where the agent makes decisions at each time step based on its current state. This process adheres to the **Markov property**, which asserts that the future state and reward depend only on the present state and action, not on the history of past states and actions [53].

Formally, the Markov property is represented by:

$$P(s' \mid s, a) = \mathbb{P}[s_{t+1} = s' \mid s_t = s, a_t = a] \tag{1}$$

which denotes the probability of transitioning from state s to state s' when action a is taken.

The reward function R is similarly defined as:

$$R(s, a) = \mathbb{E}[r_t \mid s_{t-1} = s, a_{t-1} = a] \tag{2}$$

which represents the expected reward received after taking action a in state S.

B. Policy and Value Functions

In reinforcement learning, an agent's goal is to find the optimal **policy** (π) that the agent should follow to maximize cumulative rewards over time. To facilitate this process, we need to quantify the desirability of a given state, which is done through the **value function** $V(s)$. Value function estimates the expected cumulative reward an agent will receive starting from state s and continuing thereafter. In essence, the value function reflects how beneficial it is to be in a particular state, guiding the agent's decision-making process. The **state-value function** is then defined as:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \mid s_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \dots \mid s_t = s] \end{aligned} \tag{3}$$

where G_t is the cumulative reward from time step t onwards [53].

From here we can define another value function the **action-value function** under policy π, which is $Q^\pi(s, a)$, that estimates the expected cumulative reward from the state s and taking action a and then following policy π:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a] \\ &= \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \dots \mid s_t = s, a_t = a] \end{aligned} \tag{4}$$

where γ is the discount factor, which is a decimal value between 0 and 1 that detemines how much we care about immediate rewards versus future reward rewards [53].

We say that a policy π is better than another policy π' if the expected return of every state under π is greater than or equal to the expected return of every state under π', i.e., $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in S$. Eventually, there will be a policy (or policies) that are better than or equal to all other policies, this is called the **optimal policy** π^*. All optimal policies will then share the same optimal state-value function $V^*(s)$ and the same optimal action-value function $Q^*(s, a)$, which are defined as:

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ Q^*(s, a) &= \max_\pi Q^\pi(s, a) \end{aligned} \tag{5}$$

If we can estimate the optimal state-value (or action-value) function, then the optimal policy π^* can be obtained by selecting the action that maximizes the state-value (or action-value) function at each state, i.e., $\pi^*(s) = \arg\max_a Q^*(s, a)$ and that's the goal of reinforcement learning [53].

C. Reinforcement Learning Algorithms

There are multiple reinforcement learning algorithms that have been developed that falls under a lot of categories [54]. But, for the sake of this review, we will focus on the following algorithms that have been used by the Google DeepMind team in their reinforcement learning models.

*1) **Model-Based Algorithms: Dynamic Programming***

Dynamic programming (DP) algorithms can be applied when we have a perfect model of the environment, represented by the transition probability function $P(s', r \mid s, a)$. These algorithms rely on solving the Bellman equations (recursive form of equations 3 and 4) iteratively to compute optimal policies. The process alternates between two key steps: **policy evaluation** and **policy improvement**.

1.1 Policy Evaluation

Policy evaluation involves computing the value function $V^\pi(s)$ under a given policy π. This is achieved iteratively by updating the value of each state based on the Bellman equation:

$$V^\pi(s) = \sum_{a \in A} \pi(a \mid s) \sum_{s', r} P(s', r \mid s, a) \left[r + \gamma V^\pi(s') \right]. \tag{6}$$

Starting with an arbitrary initial value $V^\pi(s)$, the updates are repeated for all states until the value function converges to a stable estimate [53].

1.2 Policy Improvement

Once the value function $V^{\pi}(s)$ has been computed, the policy is improved by choosing the action a that maximizes the expected return for each state:

$$\pi'(s) = \arg\max_a \sum_{s',r} P(s', r \mid s, a) \left[r + \gamma V^{\pi}(s') \right]. \quad (7)$$

This step ensures that the updated policy π' is better than or equal to the previous policy π. The process alternates between policy evaluation and improvement until the policy converges to the optimal policy π^*, where no further improvement is possible. It can be visualized as:

$$\pi_0 \xrightarrow{\text{Eval}} V^{\pi_0} \xrightarrow{\text{Improve}} \pi_1 \xrightarrow{\text{Eval}} V^{\pi_1} \xrightarrow{\text{Improve}} \pi_2 \xrightarrow{\text{Eval}} \dots \xrightarrow{\text{Improve}} \pi^*.$$

1.3 Value Iteration

Value iteration simplifies the DP process by combining policy evaluation and policy improvement into a single step. Instead of evaluating a policy completely, it directly updates the value function using:

$$V^*(s) = \max_a \sum_{s',r} P(s', r \mid s, a) \left[r + \gamma V^*(s') \right]. \quad (8)$$

This method iteratively updates the value of each state until convergence and implicitly determines the optimal policy. Then the optimal policy can be obtained by selecting the action that maximizes the value function at each state, as

$$\pi^*(s) = \arg\max_a \sum_{s',r} P(s', r \mid s, a) \left[r + \gamma V^*(s') \right]. \quad (9)$$

Dynamic Programming's systematic approach to policy evaluation and improvement forms the foundation for the techniques that have been cruical in training systems like AlphaGo Zero and MuZero [53].

2) Model-Free Algorithms

2.1 Monte Carlo Algorithm (MC)

The Monte Carlo (MC) algorithm is a model-free reinforcement learning method that estimates the value of states or state-action pairs under a given policy by averaging the returns of multiple episodes. Unlike DP, MC does not require a perfect model of the environment and instead learns from sampled experiences [53].

The key steps in MC include:

- **Policy Evaluation:** Estimate the value of a state or state-action pair $Q(s, a)$ by averaging the returns observed in multiple episodes.
- **Policy Improvement:** Update the policy π to choose actions that maximize the estimated value $Q(s, a)$.

MC algorithms operate on complete episodes, requiring the agent to explore all state-action pairs sufficiently to ensure accurate value estimates. The updated policy is given by:

$$\pi(s) = \arg\max_a Q(s, a). \quad (10)$$

While both MC and DP alternate between policy evaluation and policy improvement, MC works with sampled data, making it suitable for environments where the dynamics are unknown or difficult to model.

This algorithm is particularly well-suited for environments that are *episodic*, where each episode ends in a terminal state after a finite number of steps.
Monte Carlo's reliance on episodic sampling and policy refinement has directly influenced the development of search-based methods like Monte Carlo Tree Search (MCTS), which was crucial in AlphaGo for evaluating potential moves during gameplay. The algorithm's adaptability to model-free settings has made it a cornerstone of modern reinforcement learning strategies.

2.2 Temporal Diffrence (TD)

Temporal Diffrence is another model free algorithm that's very similar To Monte Carlo, but instead of waiting for termination of the episode to give the return, it estimates the return based on the next state. The key idea behind TD is to update the value function based on the difference between the current estimate and the estimate of the next state. The TD return is then given by:

$$G_t = r_{t+1} + \gamma V(s_{t+1}) \quad (11)$$

that's the target (return value estimation) of state s at time t is the immediate reward r plus the discounted value of the next state s_{t+1}.
This here is called the TD(0) algorithm, which is the simplest form of TD that take only one future step into account. The update rule for TD(0) is:

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (12)$$

There are other temporal difference algorithms that works exactly like TD(0), but with more future steps, like TD(λ).

Another important variant of TD is the Q-learning algorithm, which is an off-policy TD algorithm that estimates the optimal action-value function Q^* by updating the current action value based on the optimal action value of the next state. The update rule for Q-learning is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \quad (13)$$

and after the algorithm converges, the optimal policy can be obtained by selecting the action that maximizes the action-value function at each state, as $\pi^*(s) = \arg\max_a Q^*(s, a)$ [53].

Temporal Difference methods, including Q-learning, play a crucial role in modern reinforcement learning by enabling model-free value function estimation and action selection without the need to terminate the episode. In systems like AlphaGo and MuZero, TD methods are used to update value functions efficiently and support complex decision-making processes without requiring a model of the environment [53].

3) ***Deep RL: Deep Q-Network (DQN)***

Deep Q-Networks (DQN) represent a significant leap forward in the integration of deep learning with reinforcement learning. DQN extends the traditional Q-learning algorithm by using a deep neural network to approximate the Q-value function, which is essential in environments with large state spaces where traditional tabular methods like Q-learning become infeasible [54].

In standard Q-learning, the action-value function $Q(s,a)$ is learned iteratively based on the Bellman equation, which updates the Q-values according to the reward received and the value of the next state. However, when dealing with complex, high-dimensional inputs such as images or unstructured data, a direct tabular representation of the Q-values is not practical. This is where DQN comes in: it uses a neural network, typically a convolutional neural network (CNN), to approximate $Q(s,a;\theta)$, where θ represents the parameters of the network.

The core ideas behind DQN are similar to those of traditional Q-learning, but with a few key innovations that address issues such as instability and high variance during training. The DQN algorithm introduces the following components:

- **Experience Replay:** To improve the stability of training and to break the correlation between consecutive experiences, DQN stores the agent's experiences in a replay buffer. Mini-batches of experiences are randomly sampled from this buffer to update the network, which helps in better generalization.
- **Target Network:** DQN uses two networks: the primary Q-network and a target Q-network. The target network is updated less frequently than the primary network and is used to calculate the target value in the Bellman update. This reduces the risk of oscillations and divergence during training.

The update rule for DQN is based on the Bellman equation for Q-learning, but with the neural network approximation:

$$\begin{aligned}&Q(s_t,a_t;\theta)=Q(s_t,a_t;\theta)+\\&\alpha\left[r_{t+1}+\gamma\max_{a'}Q(s_{t+1},a';\theta^-)-Q(s_t,a_t;\theta)\right]\end{aligned}\tag{14}$$

where θ^- represents the parameters of the target network. By training the network to minimize the difference between the predicted Q-values and the target Q-values, the agent learns an optimal policy over time. [55]

The DQN algorithm revolutionized reinforcement learning, especially in applications requiring decision-making in high-dimensional spaces. One of the most notable achievements of DQN was its success in mastering a variety of Atari 2600 games directly from raw pixel input, achieving human-level performance across multiple games. This breakthrough demonstrated the power of combining deep learning with reinforcement learning to solve complex, high-dimensional problems.

In subsequent improvements, such as Double DQN, Dueling DQN, and Prioritized Experience Replay, enhancements were made to further stabilize training and improve performance. However, the foundational concepts of using deep neural networks to approximate Q-values and leveraging experience replay and target networks remain core to the DQN framework.

III. Classification of Methods

In this section, we will be discussing the four models of Google DeepMind:

A. AlphaGo

1) Introduction

AlphaGo is a groundbreaking AI model that utilizes neural networks and tree search to play the game of Go, which is thought to be one of the most challenging classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves [56].

AlphaGo uses value networks for position evaluation and policy networks for taking actions, that combined with Monte Carlo simulation achieved a 99.8% winning rate, and beating the European human Go champion in 5 out 5 games.

2) Key Innovations

*Integration of Policy and Value Networks with MCTS AlphaGo combines policy and value networks in an MCTS framework to efficiently explore and evaluate the game tree. Each edge (s,a) in the search tree stores:

- Action value $Q(s,a)$: The average reward for taking action a from state s.
- Visit count $N(s,a)$: The number of times this action has been explored.
- Prior probability $P(s,a)$: The probability of selecting action a, provided by the policy network.

During the selection phase, actions are chosen to maximize:

$$a_t=\arg\max_a\left(Q(s,a)+u(s,a)\right)\tag{15}$$

where the exploration bonus $u(s,a)$ is defined as:

$$u(s,a)\propto\frac{P(s,a)}{1+N(s,a)}\tag{16}$$

When a simulation reaches a leaf node, its value is evaluated in two ways: 1. Value Network Evaluation: A forward pass through the value network predicts $v_\theta(s)$, the likelihood of winning. 2. Rollout Evaluation: A lightweight policy simulates the game to its conclusion, and the terminal result z is recorded.

These evaluations are combined using a mixing parameter λ:

$$V(s_L)=\lambda v_\theta(s_L)+(1-\lambda)z_L\tag{17}$$

The back propagation step updates the statistics of all nodes along the path from the root to the leaf.

It's also worth noting that the SL policy network performed better than the RL policy network and that's probably because humans select a diverse beam of promising moves, whereas RL optimizes for the single best move.

Conversely though, the value network that was derived from the RL policy performed better than the one derived from the SL policy.

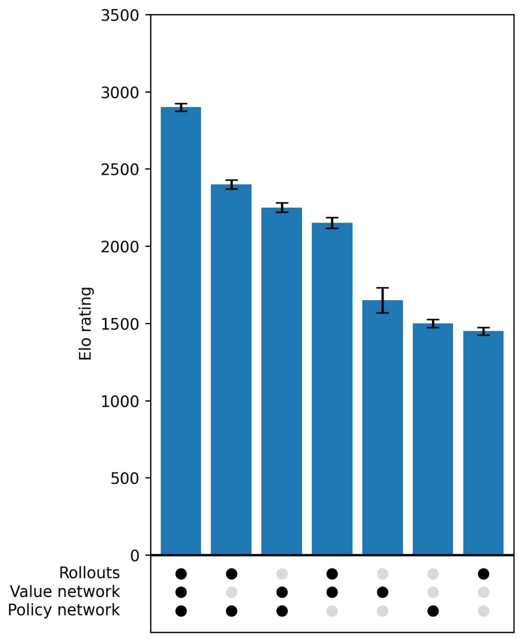

Fig. 1. Performance of AlphaGo, on a single machine, for different combinations of components.

3) Training Process

Supervised Learning for Policy Networks The policy network was initially trained using supervised learning on human expert games. The training data consisted of 30 million board positions sampled from professional games on the KGS Go Server. The goal was to maximize the likelihood of selecting the human move for each position:

$$\Delta\sigma \propto \nabla_\sigma \log p_\sigma(a|s) \tag{18}$$

where $p_\sigma(a|s)$ is the probability of selecting action a given state s.

This supervised learning approach achieved a move prediction accuracy of 57.0% on the test set, significantly outperforming prior methods. This stage provided a solid foundation for replicating human expertise.

Reinforcement Learning for Policy Networks The supervised learning network was further refined through reinforcement learning (RL). The weights of the RL policy network were initialized from the SL network. AlphaGo then engaged in self-play, where the RL policy network played against earlier versions of itself to iteratively improve.

The reward function used for RL was defined as:

$$r(s) = \begin{cases} +1 & \text{if win} \\ -1 & \text{if loss} \\ 0 & \text{otherwise (non-terminal states).} \end{cases} \tag{19}$$

At each time step t, the network updated its weights to maximize the expected reward using the policy gradient method:

$$\Delta\rho \propto z_t \nabla_\rho \log p_\rho(a_t|s_t) \tag{20}$$

where z_t is the final game outcome from the perspective of the current player.

This self-play strategy allowed AlphaGo to discover novel strategies beyond human knowledge. The RL policy network outperformed the SL network with an 80% win rate and achieved an 85% win rate against Pachi, a strong open-source Go program, without using MCTS.

Value Network Training The value network was designed to evaluate board positions by predicting the likelihood of winning from a given state. Unlike the policy network, it outputs a single scalar value $v_\theta(s)$ between -1 (loss) and $+1$ (win).

Training the value network on full games led to overfitting due to the strong correlation between successive positions in the same game. To mitigate this, a new dataset of 30 million distinct board positions was generated through self-play, ensuring that positions came from diverse contexts.

The value network was trained by minimizing the mean squared error (MSE) between its predictions $v_\theta(s)$ and the actual game outcomes z:

$$L(\theta) = \mathbb{E}_{(s,z)\sim D}\left[(v_\theta(s) - z)^2\right] \tag{21}$$

4) Challenges and Solutions

AlphaGo overcame several challenges:

- Overfitting: Training the value network on full games led to memorization. This was mitigated by generating a diverse self-play dataset.
- Scalability: Combining neural networks with MCTS required significant computational resources, addressed through parallel processing on GPUs and CPUs.
- Exploration vs. Exploitation: Balancing these in MCTS was achieved using the exploration bonus $u(s, a)$ and the policy network priors.

5) Performance Benchmarks

AlphaGo achieved the following milestones:

- 85% win rate against Pachi without using MCTS.
- 99.8% win rate against other Go programs in a tournament held to evaluate the performance of AlphaGo.
- Won 77%, 86%, and 99% of handicap games against Crazy Stone, Zen and Pachi, respectively.
- Victory against professional human players such as Fan Hui (5-0) and Lee Sedol (4-1), marking a significant breakthrough in AI.

B. AlphaGo Zero

1) Introduction

AlphaGo Zero represents a groundbreaking advancement in artificial intelligence and reinforcement learning. Unlike its predecessor, AlphaGo, which relied on human gameplay data for training, AlphaGo Zero learns entirely from self-play, employing deep neural networks and Monte Carlo Tree

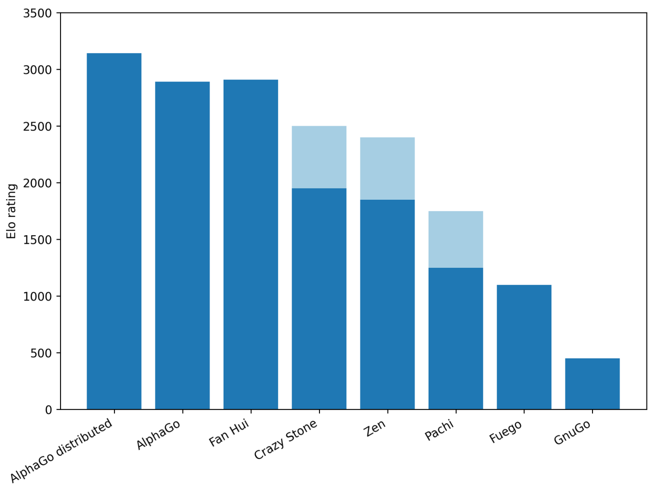

Fig. 2. Elo rating comparison between AlphaGo and other Go programs.

Search (MCTS). [57]

By starting with only the rules of the game and leveraging reinforcement learning, AlphaGo Zero achieved superhuman performance in Go, defeating the previous version of AlphaGo in a 100-0 match.

2) Key Innovations

AlphaGo Zero introduced several groundbreaking advancements over its predecessor, AlphaGo, streamlining and enhancing its architecture and training process:

1) Unified Neural Network f_θ: AlphaGo Zero replaced AlphaGo's dual-network setup—separate networks for policy and value—with a single neural network f_θ. This network outputs both the policy vector p and the value scalar v for a given game state, reprsented as

$$f_\theta(s) = (p, v) \tag{22}$$

This unified architecture simplifies the model and improves training efficiency.

2) Self-Play Training: Unlike AlphaGo, which relied on human games as training data, AlphaGo Zero was trained entirely through self-play. Starting from random moves, it learned by iteratively playing against itself, generating data and refining f_θ without any prior knowledge of human strategies. This removed biases inherent in human gameplay and allowed AlphaGo Zero to discover novel and highly effective strategies.
3) Removal of Rollouts: AlphaGo Zero eliminated the need for rollouts, which were computationally expensive simulations to the end of the game used by AlphaGo's MCTS. Instead, f_θ directly predicts the value v of a state, providing a more efficient and accurate estimation.
4) Superior Performance: By integrating these advancements, AlphaGo Zero defeated AlphaGo 100-0 in direct matches, demonstrating the superiority of its self-play training, unified architecture, and reliance on raw rules over pre-trained human data.

3) Training Process

Monte Carlo Tree Search (MCTS) as policy evaluation operator Intially the neural network f_θ is not very accurate in predicting the best move, as it is intiallised with random weights at first. To overcome this, AlphaGo Zero uses MCTS to explore the game tree and improve the policy.

At a given state S, MCTS expands simualtions of the best moved that are most likely to generate a win based on the initial policy $P(s, a)$ and the value V. MCTS iteratively selects moves that maximize the upper confidence bound (UCB) of the action value. UCB is designed to balanced exploration and exploitation. and it is defined as

$$UCB = Q(s, a) + U(s, a) \tag{23}$$

where

$$U(s, a) \propto \frac{p(s, a)}{1 + N(s, a)}$$

MCTS at the end of the search returns the policy vector π which is used to update the neural network f_θ by minimizing the cross-entropy loss between the predicted policy by f_θ and the MCTS policy.

Policy Iteration and self play The agent uses the MCTS to select the best move at each state and the game is played till the end in a process called self play. The agent then uses the outcome of the game, z game winner and π to update the neural network. This process is repeated for a large number of iterations.

Network Training Process

The neural network is updated after each self-play game by using the data collected during the game. This process involves several key steps:

1) Intilisation of the network: The neural network starts with random weights θ_0, as there is no prior knowledge about the game.
2) Generating Self-play Games: For each iteration $i \geq 1$ self-play games are generated. During the game, the neural network uses its current parameters θ_{i-1} to run MCTS and generate search probabilities π_t for each move at time step t.
3) Game Termination and scoring: A game ends when either both players pass, a resignation threshold is met, or the game exceeds a maximum length. The winner of the game is determined, and the final result z_t is recorded, providing feedback to the model.
4) Data Colletion: for each time step t, the training data (s_t, π_t, z_t) is stored, where s_t is the game state, π_t is the search probabilities, and z_t is the game outcome.
5) Network training process: after collecing data from self-play, The neural network f_θ is adjusted to minimize the error between the predicted value v and the self-play winner z, and to maximize the similarity between the search probabilities P and the MCTS probabilities. This

is done by using a loss function that combines the mean-squared error and the cross entropy losses repsectibly. The loss function is defined as

$$L = (z - v)^2 - \pi^T \log p + c||\theta||^2 \tag{24}$$

where c is the L2 regularization term.

4) Challenges and Solutions

Alpha Go Zero overcame several challanges:

1) Human knowledge Dependency: AlphaGo Zero eliminated the need for human gameplay data, relying solely on self-play to learn the game of Go. This allowed it to discover novel strategies that surpassed human expertise.
2) Compelxity of the dual network approach in alpha go: AlphaGo utilized separate neural networks for policy prediction p and value estimation V, increasing the computational complexity. AlphaGo Zero unified these into a **single network** that outputs both p and V, simplifying the architecture and improving training efficiency.
3) The need of handcrafted features: AlphaGo relied on handcrafted features, such as board symmetry and predefined game heuristics, for feature extraction. AlphaGo Zero eliminated the need for feature engineering by using **raw board states** as input, learning representations directly from the data.

5) Performance Benchmarks

Evaluating the performance of DeepMind's reinforcement learning (RL) agents for strategy games such as Go has primarily been conducted using Elo ratings and direct head-to-head matches against previous versions.

Table I summarizes the reported Elo ratings of the different versions of AlphaGo, illustrating the rapid progression in strength achieved through successive architectural improvements.

TABLE I
ELO RATINGS OF DIFFERENT ALPHAGO VERSIONS

Model	Elo rating
AlphaGo Zero	5185
AlphaGo Master	4858
AlphaGo Lee	3739
AlphaGo Fan	3144

Additionally, AlphaGo Zero defeated the then state-of-the-art AlphaGo Master in a 100-game match with a score of 89–11, demonstrating the effectiveness of self-play reinforcement learning without the need for Monte Carlo rollouts or human expert data.

C. Alpha Zero

1) Introduction

AlphaZero is a generalized reinforcement learning framework that extends the methodology of AlphaGo Zero beyond the domain of Go. It employs the same core components—self-play reinforcement learning, deep neural networks with policy and value heads, and Monte Carlo Tree Search (MCTS)—but is designed to operate across Shogi, Chess, and Go. With only minimal adjustments to the input and output representations to account for game-specific rules, AlphaZero achieved superhuman performance in Go, Chess, and Shogi, thereby demonstrating the domain-independence and general applicability of the approach.

2) Key Innovations

AlphaZero introduced several advancements over AlphaGo Zero, extending its applicability and improving efficiency:

1) Generalization Across Games: Unlike AlphaGo Zero, which was designed exclusively for Go, AlphaZero applied the same reinforcement learning framework to *Go, Chess, and Shogi*. With only minimal modifications to the input and output representations to encode different rules and move sets, AlphaZero demonstrated the domain-independence of the algorithm.
2) Game-Agnostic Training Procedure: AlphaZero established the training process as a fully general loop applicable to multiple strategy games. Self-play guided by MCTS generated training examples (s_t, π_t, z), and the neural network was updated using the same loss function as in AlphaGo Zero. No game-specific heuristics or specialized adjustments were required, confirming the general-purpose nature of the approach.
3) Improved Efficiency: AlphaZero achieved strong performance with approximately 800 MCTS simulations per move, compared to the $\sim$1,600 simulations typically used by AlphaGo Zero. This reduction highlights the efficiency of its policy–value network in guiding search and reduced the computational cost of training and inference.
4) Cross-Domain Superiority: Beyond outperforming AlphaGo Zero in Go, AlphaZero achieved superhuman results in Chess and Shogi, decisively defeating *Stockfish* and *Elmo*, the strongest domain-specific engines at the time. These results established AlphaZero as the first general reinforcement learning system to surpass highly optimized, handcrafted programs across multiple complex strategy games.

3) Performance Benchmarks

Chess AlphaZero was evaluated against Stockfish, the strongest chess engine at the time. After only 24 hours of training from random play, AlphaZero achieved a dominant performance.
In a 100-game match under one-minute per move time controls, it recorded the following results:

1) 28 wins, 72 draws, and 0 losses as White
2) 25 wins, 73 draws, and 2 losses as Black

These results demonstrated that AlphaZero could surpass a hand-crafted, search-intensive engine within a single day of training.

Shogi In Shogi, AlphaZero was trained for 24 hours and then matched against Elmo, the world champion program.

In a 100-game match under one-minute per move controls, AlphaZero achieved 90 wins, 8 draws, and only 2 losses, a result that established it as the strongest known Shogi player, human or machine, at the time.

Go

AlphaZero was compared directly with AlphaGo Zero, its predecessor and the strongest Go-playing system prior to this work. Despite the short training duration of 24 hours, AlphaZero won 60 games to 40 in a 100-game match. This result demonstrated that AlphaZero not only generalized across domains but also improved upon the performance of domain-specialized systems.

D. MuZero

1) Introduction

Through the development of AlphaZero, a general model for board games with superhuman ability has been achieved in three games: Go, chess, and Shogi. It could achieve these results without the need for human gameplay data or history, instead using self-play in an enclosed environment. However, the model still relied on a simulator that could perfectly replicate the behavior, which might not translate well to real-world applications, where modeling the system might not be feasible. MuZero was developed to address this challenge by developing a model-based RL approach that could learn without explicitly modeling the real environment. This allowed for the same general approach used in AlphaZero to be used in Atari environments where reconstructing the environment is costly. Essentially, MuZero was deployed to all the games with no prior knowledge of them or specific optimization and managed to show state-of-the-art results in almost all of them.

2) MuZero Algorithm

The model takes in an input of observations $o_1, \ldots, o_t$ that are then fed to a representation network h, which reduces the dimensions of the input and produces a root hidden state s_0. Internally, the model mirrors an MDP, with each state representing a node with edges connecting it to the future states depending on available actions. Unlike traditional RL approaches, this hidden state is not constrained to contain the information necessary to reproduce the entire future observations. Instead, the hidden states are only optimized for predicting information that is related to planning. At every time step, the model predicts the policy, the immediate reward, and the value function. The output of the state-action pair is then used by the dynamics function to produce future states. Similar to AlphaZero, a Monte Carlo tree search is used to find the best action policy given an input space. This is used to train the model by comparing the MCTS policy with the predictor function policy. Also, after a few training runs, the model ceases to use illegal moves, and the predicted actions map to the real action space. This eliminates the need for a simulator, as the model internalizes the environment characteristics it deems necessary for planning and acting, which generally converges to reality through training. The value function at the final step is compared against the game result in board games, i.e., win, loss, or a draw.

3) Loss function and learning equations

$$s_0 = h_\theta(o_1, \ldots, o_t) \tag{25}$$

$$r_k, s_k = g_\theta(s_{k-1}, a_k) \tag{26}$$

$$p_k, v_k = f_\theta(s_k) \tag{27}$$

$$\begin{bmatrix} p_k \\ v_k \\ r_k \end{bmatrix} = \mu_\theta(o_1, \ldots, o_t, a_1, \ldots, a_k) \tag{28}$$

$$\nu_t, \pi_t = \text{MCTS}(s_0^t, \mu_\theta) \tag{29}$$

$$a_t \sim \pi_t \tag{30}$$

$$p_k^t, v_k^t, r_k^t = \mu_\theta(o_1, \ldots, o_t, a_{t+1}, \ldots, a_{t+k}) \tag{31}$$

$$z_t = \begin{cases} u_T, & \text{for games} \\ u_{t+1} + \gamma u_{t+2} + \ldots \\ \quad +\gamma^{n-1} u_{t+n} + \gamma^n \nu_{t+n}, & \text{for general MDPs} \end{cases}$$

$$l_t(\theta) = \sum_{k=0}^{K} \big[l_r(u_{t+k}, r_k^t) + l_v(z_{t+k}, v_k^t) + l_p(\pi_{t+k}, p_k^t) \big] + c\|\theta\|^2 \tag{32}$$

$$l_r(u, r) = \begin{cases} 0, & \text{for games} \\ \phi(u)^T \log r, & \text{for general MDPs} \end{cases} \tag{33}$$

$$l_v(z, q) = \begin{cases} (z-q)^2, & \text{for games} \\ \phi(z)^T \log q, & \text{for general MDPs} \end{cases} \tag{34}$$

$$l_p(\pi, p) = \pi^T \log p \tag{35}$$

4) MCTS

MuZero uses the same approach developed in AlphaZero to find the optimum action given an internal state. MCTS is used where states are the nodes, and the edges store visit count, mean value, policy, and reward. The search is done in a three-phase setup: selection, expansion, and backup. The simulation starts with a root state, and an action is chosen based on the state-transition reward table. Then, after the end of the tree, a new node is created using the output of the dynamics function as a value, and the data from the prediction function is stored in the edge connecting it to the previous state. Finally, the simulation ends, and the updated trajectory is added to the state-transition reward table. In two-player zero-sum games, board games, for example, the value function is bounded between 0 and 1, which is helpful to use value estimation and probability using the pUCT rule. However, many other environments have unbounded values, so MuZero rescales the value to the maximum value observed by the model up to this training step, ensuring no environment-specific data is needed. [59]

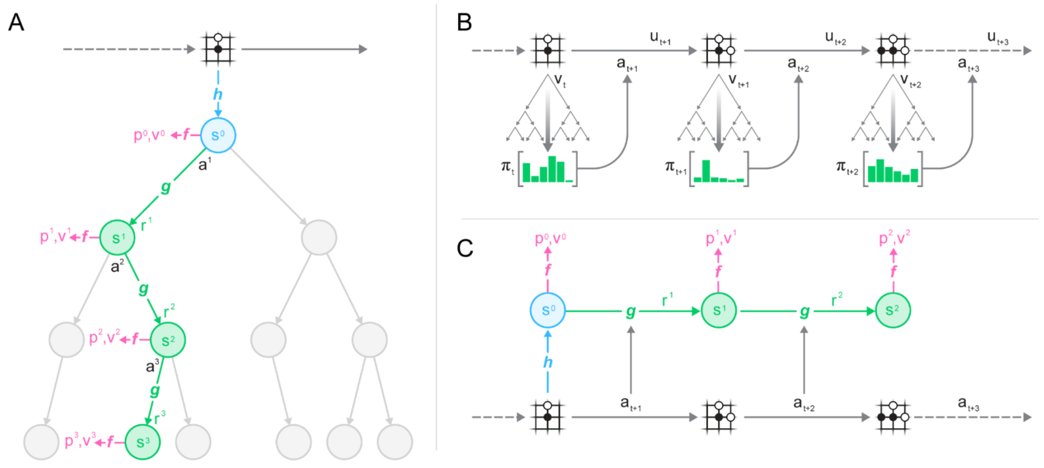

Fig. 3. (A) Represents the progression of the model through its MDP, while (B) Represents MuZero acting as an environment with MCTS as feedback, and (C) Represents a diagram of training MuZero's model.

5) *Results*

The MuZero model demonstrated significant improvements across various test cases, achieving state-of-the-art performance in several scenarios. Key findings include:

Board Games

- When tested on the three board games AlphaZero was trained for (Go, chess, and shogi):
  - MuZero matched AlphaZero's performance **without any prior knowledge** of the games' rules.
  - It achieved this with **reduced computational cost** due to fewer residual blocks in the representation function.

Atari Games

- MuZero was tested on 60 Atari games, competing against both human players and state-of-the-art models (model-based and model-free). Results showed:
  - **Starting from regular positions:** MuZero outperformed competitors in **46 out of 60 games**.
  - **Starting from random positions:** MuZero maintained its lead in **37 out of 60 games**, though its performance was reduced.
- The computational efficiency and generalization of MuZero highlight its effectiveness in complex, unstructured environments.

Limitations

- Despite its strengths, MuZero struggled in certain games, such as:
  - *Montezuma's Revenge* and *Pitfall*, which require long-term planning and strategy.
- General challenges:
  - Long-term dependencies remain difficult for MuZero, as is the case for RL models in general.
  - Limited input space and lack of combinatorial inputs in Atari games could introduce scalability issues for broader applications. [59]

IV. Real-World Applications

The true value of the deep reinforcement learning paradigm pioneered by the DeepMind models is demonstrated by its successful translation into high-impact real-world applications. These models excel in domains that can be framed as sequential decision-making problems with clear objectives, often discovering novel strategies that surpass human-designed solutions. The following case studies illustrate how algorithms developed for games are now driving innovation across mathematics, computer science, systems engineering, and biology.

A. *Matrix Multiplication: Alpha Tensor*

A prime example of this translation is **Alpha Tensor**, a deep RL model that adapts the AlphaZero algorithm to the fundamental problem of discovering efficient algorithms for matrix multiplication [60]. Alpha Tensor frames the problem as a single-player game, the *Tensor Game*. The state of the game is a three-dimensional tensor, and actions correspond to updating this tensor. The model is rewarded for finding a path that factorizes the initial matrix multiplication tensor into the zero tensor in the fewest steps, corresponding to the lowest *rank* of the multiplication algorithm.

Searching a space of predefined factor entries, the model discovers the optimal multiplication algorithm from scratch. Through this process, Alpha Tensor discovered algorithms that matched or surpassed the best human-developed ones. A notable achievement was for 4×4 matrices, where it discovered a rank-47 algorithm, an improvement over the long-standing best-known rank-49 (Strassen's) algorithm. Beyond finding a single optimal solution, Alpha Tensor generated a vast database of distinct, non-equivalent multiplication algorithms, providing a valuable resource for further mathematical research. Furthermore, the model's reward function can be adjusted to optimize for specific hardware capabilities, demonstrating its flexibility in generating hardware-aware algorithms that minimize latency or energy consumption.

B. *Sorting Algorithms: AlphaDev*

Sorting is one of the most common subroutines in programming, and as the demand for computation increases, the use of such fundamental algorithms grows. As such, optimizing them is critical for computational efficiency. **AlphaDev** adapts the AlphaZero algorithm to this task by modeling the sequence of low-level CPU instructions (assembly code) for a sorting function as a single-player game, the *AssemblyGame* [61]. The goal of the game is to find a correct program that minimizes latency (a lower "score" is better).

The model searches the game space for the shortest, fastest, and correct algorithm. For fixed-length sorting (e.g., sorting lists of length 3, 4, and 5), it discovered new algorithms that outperformed the best-known human-designed solutions in the standard C++ library. Even for the more complex variable-length sort, AlphaDev generated enhancements. Notably, the improved sorting libraries discovered by AlphaDev have been integrated into the standard C++ `libc++` library, providing efficiency gains for millions of developers and applications worldwide.

C. *Compression Optimization: MuZero RC*

A notable implementation of MuZero has been in collaboration with YouTube, where it was used to optimize video compression within the open-source VP9 codec. This implementation, called **MuZero Rate-Controller (MuZero-RC)**, adapted MuZero's ability to predict and plan to the complex, practical task of video streaming [62].

By optimizing the encoding process, MuZero-RC achieved an average bitrate reduction of 4% without degrading video quality. This improvement directly impacts the efficiency of video streaming services such as YouTube, leading to faster loading times and reduced data usage for users. This application exemplifies how the model-based reinforcement learning principles behind MuZero can address practical real-world challenges outside of games, making computer systems more efficient and less resource-intensive.

D. *Protein Folding: AlphaFold*

AlphaFold addresses one of the most challenging problems in biology: predicting the three-dimensional structure of a protein from its amino acid sequence [63]. While primarily a feat of supervised learning, AlphaFold's training was refined using reinforcement learning to improve the accuracy of its predicted protein structures. The model operates on a feedback loop where it is rewarded for generating structures that match known experimental data.

This process of iterative refinement, guided by a reward signal, aligns with the core RL principles explored in this paper. The architecture of AlphaFold includes deep neural networks that analyze both the sequential and spatial relationships between amino acids. By training on extensive datasets of known protein structures, AlphaFold has achieved unprecedented accuracy, often rivaling experimental methods such as X-ray crystallography. This breakthrough has had a transformative impact on biological research and drug discovery.

V. Future Directions

The Alpha series demonstrated a remarkable capacity for mastering environments through self-play, a feat extended by MuZero, which learned an internal model without a simulator. Despite their success, these models are fundamentally limited to single-player, deterministic games. The next critical step in their evolution is to expand their scope to more complex and realistic settings. Furthermore, the core environment discovery algorithm must be enhanced to operate effectively in wider state and action spaces. This section discusses pioneering advancements in these directions, including multi-agent generalization, algorithmic improvements for exploration, and handling stochasticity.

A. *Model-based Multi-Agent Reinforcement Learning*

Traditional RL models, including the Alpha series, focus on optimizing a strategy in a stationary environment. However, many real-world applications require collaboration or competition between multiple agents. Deploying multiple agents introduces fundamental challenges: the environment becomes non-stationary from any single agent's perspective, partial observability is common, and the problems of coordination, reward assignment, and scalability become paramount. These challenges make MARL significantly more complex and sample inefficient.

A key future direction is to extend the model-based planning principles of MuZero to these multi-agent settings. Model-based methods have demonstrated higher sample efficiency in complex single-agent environments, a trait that is critically needed to tackle the exponential complexity of MARL [64]. The goal is to develop agents that can not only predict environment dynamics but also model the strategies of other agents, enabling sophisticated planning in competitive or cooperative spaces.

This remains a formidable open challenge. Projects like AlphaStar [66] show that superhuman performance in complex multi-agent games is possible, but they rely on immense computational resources and specialized architectures. Future research must focus on making such planning efficient and generalizable. For instance, recent work explores using deep networks to model diverse agent behaviors and improve coordination, as seen in approaches that learn diverse Q-vectors [65] or use centralized training for decentralized execution, pushing the boundaries of what is possible in cooperative tasks like multi-robot coordination.

B. *Algorithmic Enhancements for Exploration and Efficiency*

A parallel direction for advancement lies in enhancing the core algorithms themselves to improve their sample efficiency, exploration capabilities, and computational footprint. The monumental computational cost of training models like MuZero from scratch remains a significant barrier to wider application and experimentation.

The **MiniZero** framework [67] provides a comparative analysis of algorithms like AlphaZero and MuZero, including variants that incorporate stochastic planning techniques. A key innovation in this space is the integration of **Gumbel noise** into the Monte Carlo Tree Search (MCTS) process [68]. This approach replaces deterministic action selection with a probabilistic one, leading to more robust exploration and the discovery of novel strategies that might be missed by the standard upper confidence bound heuristic. By improving the planning process itself, these methods aim to achieve stronger performance with fewer simulations and reduced overall computation, making the powerful model-based RL paradigm more accessible and applicable to problems with vast state spaces.

C. *Robustness in Stochastic and Partially Observable Environments*

Finally, a major frontier is equipping these models to handle the inherent uncertainty and randomness of real-world environments. The successes of the Alpha series and MuZero have largely been confined to deterministic and fully observable settings, which are rare outside of perfect simulators and perfect information games.

Future iterations must develop a robustness to stochastic transitions and rewards. This involves creating agents that can maintain performance even when the outcome of an action is not guaranteed or when the reward signal is noisy. Furthermore, models need to effectively address partial observability, where the agent does not have access to the full state of the environment. This requires moving beyond the Markov assumption and learning to maintain and update a belief state over time. Promising research directions include blending the planning strengths of MuZero with techniques from robust reinforcement learning [69] and leveraging recurrent architectures to better infer the true state of a partially observed world. Achieving this would represent a final crucial step in transitioning these game-playing algorithms into robust decision-making engines for real-world applications.

Conclusion

Games as an environment for Reinforcement learning, have proven to be very helpful as a sandbox. Their modular nature enables experimentation for different scenarios from the deterministic board games to visually complex and endless atari games. Google's DeepMind utilized this in developing and enhancing their models starting with AlphaGo that required human gameplay as well as knowledge of the game rules. Incrementally, they started stripping down game specific data and generalizing the models. AlphaGoZero removed the need for human gameplay and AlphaZero generalized the approach to multiple board games. Subsequently, MuZero removed any knowledge requirements of games and was able to achieve break-through results in tens of games surpassing all previous models. These advancements were translated to real-life applications seen in MuZero's optimization of the YouTube compression algorithm, which was already highly optimized using traditional techniques. The well defined nature of the problem helped in achieving this result. Also, AlphaFold used reinforcement learning in combination with supervised learning and biology insights to simulate protein structures. While these uses are impressive, especially coming from models primarily trained to play simple games, they are still limited in scope. There are many possible holdbacks mainly the training cost, scalability, and stochastic environments. These models are very expensive to train despite the limited action and state spaces. This cost would only increase at more complex environments, taking us to the second issue: scalability. In many real applications, the actions aren't mutually exclusive. This would make the MCTS exponentially more expensive and would further increase the training cost. Finally, while these models have been tested in deterministic environments, stochastic scenarios might cause trouble for their training and inference.

References

[1] B. F. Skinner, *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century, 1938.

[2] N. Y. Georgios and T. Julian, Artificial Intelligence and Games. New York: Springer, 2018.

[3] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Dev.*, vol. 3, no. 3, pp. 210–229, 1959.

[4] M. Minsky, "Steps Toward Artificial Intelligence," *Proc. IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[5] L. P. Kaelbling, M. L. Littman, and A. W. Moore,"Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[6] R. S. Sutton and A. G. Barto,*Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018.

[7] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling,"The Arcade Learning Environment: An evaluation platform for general agents,"*Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[8] V. Mnih et al.,"Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[10] R. A. Howard, *Dynamic Programming and Markov Processes*. Wiley, 1960.

[11] R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[12] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst., Man, Cybern.*, vol. 13, no. 5, pp. 834–846, 1983.

[13] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[14] C.J.C.H.Watkins, P. Dayan, Q-learning. Mach Learn 8, 279–292 (1992).

[15] P. Dayan, "The convergence of TD(λ) for general λ," *Machine Learning*, vol. 8, no. 3–4, pp. 341–362, 1992.

[16] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[18] N. Justesen, P. Bontrager, J. Togelius, S. Risi, (2019). Deep learning for video game playing. arXiv.

[19] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Trans. Autom. Control*, vol. 42, no. 5, pp. 674–690, 1997.

[20] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning,"

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, (2013). Playing Atari with deep reinforcement learning. arXiv.

[22] A. Graves, G. Wayne, I. Danihelka, (2014). Neural Turing Machines. arXiv.

[23] DeepMind, (2015, February 12), Deep reinforcement learning.
[24] C. J. C. H. Watkins, *Learning from Delayed Rewards*, PhD thesis, Univ. of Cambridge, 1989.
[25] T. Schaul, J. Quan, I. Antonoglou, D. Silver, (2015). Prioritized Experience Replay. arXiv.
[26] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3–4, pp. 293–321, 1992.
[27] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, (2016). Asynchronous Methods for Deep Reinforcement Learning. arXiv.
[28] V. R. Konda and J. N. Tsitsiklis, "Actor-Critic Algorithms," in *Advances in Neural Information Processing Systems*, vol. 12, 2000.
[29] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.
[30] K. Shao, Z. Tang, Y. Zhu, N. Li, D. Zhao, (2019). A survey of deep reinforcement learning in video games. arXiv.
[31] A. Kailash, P. D. Marc, B. Miles, and A. B. Anil, (2017). Deep Reinforcement Learning: A Brief Survey. IEEE Signal Processing Magazine, vol. 34, pp. 26–38, 2017. arXiv.
[32] D. Zhao, K. Shao, Y. Zhu, D. Li, Y. Chen, H. Wang, D. Liu, T. Zhou, and C. Wang, "Review of deep reinforcement learning and discussions on the development of computer Go," Control Theory and Applications, vol. 33, no. 6, pp. 701–717, 2016 arXiv.
[33] Z. Tang, K. Shao, D. Zhao, and Y. Zhu, "Recent progress of deep reinforcement learning: from AlphaGo to AlphaGo Zero," Control Theory and Applications, vol. 34, no. 12, pp. 1529–1546, 2017.
[34] M. Hessel et al., "Rainbow: Combining improvements in deep reinforcement learning,"in *Proc. AAAI*, 2018, pp. 3215–3222.
[35] T. P. Lillicrap et al.,"Continuous control with deep reinforcement learning," arXiv:1509.02971, 2015.
[36] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz,"Trust region policy optimization," in *Proc. ICML*, 2015.
[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov,"Proximal Policy Optimization Algorithms," arXiv:1707.06347, 2017.
[38] L. Espeholt et al.,"IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proc. ICML*, 2018.
[39] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments,"in *Proc. NeurIPS*, 2017.
[40] K. Zhang, Z. Yang, and T. Başar,"Multi-agent reinforcement learning: A selective overview of theories and algorithms,"*Handbook of Reinforcement Learning and Control*, Springer, 2019.
[41] O. Vinyals et al.,"Grandmaster level in StarCraft II using multi-agent reinforcement learning,"*Nature*, vol. 575, pp. 350–354, 2019.
[42] M. Jaderberg et al.,"Human-level performance in 3D multiplayer games with population-based reinforcement learning,"*Science*, vol. 364, no. 6443, pp. 859–865, 2019.
[43] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos,"Unifying count-based exploration and intrinsic motivation,"in *Proc. NeurIPS*, 2016.
[44] D. Pathak, P. Agrawal, A. Efros, and T. Darrell,"Curiosity-driven exploration by self-supervised prediction,"in *Proc. CVPR*, 2017.
[45] C. Zhang, O. Vinyals, R. Munos, and S. Bengio,"A study on overfitting in deep reinforcement learning," arXiv:1804.06893, 2018.
[46] R. Kirk, C. Zhang, J. Parker-Holder, Y. Lu, A. Gleave,and J. Foerster, "Survey of generalisation in deep reinforcement learning,"*Artificial Intelligence Review*, vol. 56, pp. 3003–3037, 2023.
[47] K. Zhang, Z. Yang, and T. Başar,"Multi-agent reinforcement learning: A selective overview of theories and algorithms,"*Handbook of Reinforcement Learning and Control*, Springer, 2019.
[48] A. Ecoffet, J. Huizinga, J. Lehman, K. Stanley, and J. Clune,"First return, then explore," *Nature*, vol. 590, pp. 580–586, 2021.
[49] M. E. Taylor and P. Stone,"Transfer learning for reinforcement learning domains: A survey,"*Journal of Machine Learning Research*, vol. 10, pp. 1633–1685, 2009.
[50] Y. Li, "Deep reinforcement learning: An overview,"*arXiv:1701.07274*, updated 2018.
[51] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare,J. Pineau, and D. Precup, "An introduction to deep reinforcement learning,"*Foundations and Trends in Machine Learning*, vol. 11, no. 3–4,pp. 219–354, 2018.
[52] L. Thorndike and D. Bruce, Animal Intelligence. Routledge, 2017.
[53] R. S. Sutton and A. Barto, Reinforcement learning : an introduction. Cambridge, Ma ; London: The Mit Press, 2018.
[54] A. Kumar Shakya, G. Pillai, and S. Chakrabarty, "Reinforcement Learning Algorithms: A brief survey," Expert Systems with Applications, vol. 231, p. 120495, May 2023
[55] Mnih, Volodymyr, et al. "Human-Level Control through Deep Reinforcement Learning." Nature, vol. 518, no. 7540, Feb. 2015, pp. 529–533.
[56] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: https://doi.org/10.1038/nature16961.
[57] Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, et al. 2017. "Mastering the Game of Go without Human Knowledge." Nature 550 (7676): 354–59. https://doi.org/10.1038/nature24270.
[58] David Silver et al. ,A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science 362,1140-1144(2018). doi:10.1126/science.aar6404
[59] J. Schrittwieser et al., "Mastering Atari, go, chess and shogi by planning with a learned model," Nature, vol. 588, no. 7839, pp. 604–609, Dec. 2020. doi:10.1038/s41586-020-03051-4
[60] A. Fawzi et al., "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, Oct. 2022, doi: 10.1038/s41586-022-05172-4.
[61] D. J. Mankowitz et al., "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, Jun. 2023, doi: 10.1038/s41586-023-06004-9.
[62] DeepMind, "MuZero's first step from research into the real world," DeepMind, Feb. 11, 2022. [Online]. Available: https://www.deepmind.com/blog/muzeros-first-step-from-research-into-the-real-world. [Accessed: Oct. 1, 2023].
[63] J. Jumper et al., "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021, doi: 10.1038/s41586-021-03819-2.
[64] X. Wang, Z. Zhang, W. Zhang, "Model-based Multi-agent Reinforcement Learning: Recent Progress and Prospects," *arXiv preprint arXiv:2303.07883*, 2023.
[65] Z. Luo, Z. Chen, J. Welsh, "Multi-agent Reinforcement Learning with Deep Networks for Diverse Q-Vectors," *IEEE Transactions on Robotics*, vol. 39, no. 2, 2023.
[66] O. Vinyals et al., "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.
[67] T.-R. Wu et al., "MiniZero: Comparative Analysis of AlphaZero and MuZero in Go, Othello, and Atari Games," *arXiv preprint arXiv:2304.06826*, 2023.
[68] I. Danihelka et al., "Policy improvement by planning with gumbel," in *International Conference on Learning Representations (ICLR)*, 2022.
[69] T. Oikarinen et al., "Robust Reinforcement Learning via Adversarial Kernel Approximation," in *International Conference on Learning Representations (ICLR)*, 2023.