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

Abdelrhman Shaheen
Computer Science Engineering Undergraduate Student
Egypt Japan University of Science and Technology
Alexandria, Egypt
abdelrhman.shaheen@ejust.edu.eg

Anas Badr
Computer Science Engineering Undergraduate Student
Egypt Japan University of Science and Technology
Alexandria, Egypt
anas.badr@ejust.edu.eg

Ali Abohendy
Computer Science Engineering Undergraduate Student
Egypt Japan University of Science and Technology
Alexandria, Egypt
ali.abohendy@ejust.edu.eg

Hatem Alsaadawy
Computer Science Engineering Undergraduate Student
Egypt Japan University of Science and Technology
Alexandria, Egypt
hatem.alsaadawy@ejust.edu.eg

Nadine Alsayad
Computer Science Engineering Undergraduate Student
Egypt Japan University of Science and Technology
Alexandria, Egypt
nadine.alsayad@ejust.edu.eg

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. From simple puzzles to complex strategy games, AI research in gaming has pushed the boundaries of machine learning and reinforcement learning. Also The benfits from such employment helped game developers to realize the power of AI methods to analyze large volumes of player data and optimize game designs. [1]
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 that they require strategies that involve planning, adaptability, and fast decision-making, making them also a good testing environment for evaluating AI's ability to

learn and generalize. The development of AI in games has been a long journey, starting with rule-based systems and evolving into more sophisticated machine learning models. However, the machine learning models had a few challenges, from these challenges is that the games employing AI involves decision making in the game eivronment. Machine learning models are unable to interact with the decisions that the user make because it depends on learning from datasets and have no interaction with the environment. To overcome such problem, game developers started to employ reinforcement learning (RL) in developing games. Years later, deep learning (DL) was developed and shows remarkable results in video games [2]. The combination between both reinforcement learning and deep learning resulted in Deep Reinforcement Learning (DRL). The first employment of DRL was by game developers in atari game [3]. One of the famous companies that employed DRL in developing AI models is Google DeepMind. This company is known for developing AI models, including games. Google DeepMind passed through a long journey in developing AI models for games. Prior to the first DRL game model they develop, which is AlphaGo, Google DeepMind gave a lot of contributions in developing DRL, by which these contributions were first employed in Atari games.

For the employment of DRL in games to be efficient, solving tasks in games need to be sequential, so Google DeepMind combined RL-like techniques with neural networks to create models capable of learning algorithms and solving tasks that require memory and reasoning, which is the Neural Turing Machines (NTMs) [4]. They then introduced the Deep Q-network (DQN) algorithm, which is combine deep learning with Q-learning and RL algorithm. Q-learning is a model in reinforcement learning which use the Q-network, which is is a type of neural network to approximate the Q-function, which predicts the value of taking a particular action in a given state [5]. The DQN algorithm was the first algorithm that was able to learn directly from high-dimensional sensory input, the data that have a large number of features or dimensions [6].

To enhance the speed of learning in reinforcement learning agents, Google DeepMind introduced the concept of experience replay, which is a technique that randomly samples previous experiences from the agent's memory to break the correlation between experiences and stabilize the learning process [7]. They then developed asynchronus methods for DRL, which is the Actor-Critic (A3C) model. This model showed faster and more stable training and showed a remarkable performance in Atari games [8]. By the usage of these algorithms, Google DeepMind was able to develop the first AI model that was able to beat the world champion in the game of Go, which is AlphaGo in 2016.

The paper is organized as follows: Section II presents the related work that surveys the development of DRL in games and the contribution that we added to the previous surveys. Section III presents the background information about the development of DRL in games. Section IV presents the first AI model that Google DeepMind developed, which is AlphaGo. Section V presents AlphaGo Zero. Section VI presents MuZero. Section VII presents the advancements that were made in developing AI models for games. Section VIII presents the future directions that AI models for games will take, and their applications in real life.

II. Related Work

There are a lot of related work that reviewed the reinforcement learning in strategy-based and atari games. Arulkumaran et al [9] make a brief introduction of DRL, covering central algorithms and presenting a range of visual RL domains Zhao et al. [10] and Tang et al. [11] survey the development of DRL research, and focus on AlphaGo and AlphaGo Zero models. Shao et al. [12] review the development of DRL in game AI, from 2D to 3D, and from single-agent to multi-agent, and discuss the real-time strategy games.

In this paper, we will discuss the development that Google DeepMind made in developing AI models for games and the advancments. The main contribution will be the comprehensive details of the three models AlphaGo, AlphaGo Zero, and MuZero, focusing on the challenges that each model faced and the improvements that were made. Also we will discuss the advancments in these three AI models, reaching to the future directions.

III. Background

Reinforcement Learning (RL) is a key area of machine learning that focuses on learning through interaction with the environment. 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 [13].

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—a strategy that dictates the best action to take in each state. That's said, for a give agent in state u, if it takes action u, then the immediate reward r can be modeled as $r(x, u) = \mathbb{E}[r_t \mid x = x_{t-1}, u = u_{t-1}]$.

So for a full episode of T steps, the cumulative reward R can be modeled as $R = \sum_{t=1}^{T} r_t$.

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.

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 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:

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

where G_t is the cumulative reward from time step t onwards. 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:

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

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 [14].

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 [14].

C. Reinforcement Learning Algorithms

There are multiple reinforcement learning algorithms that have been developed that falls under a lot of categories. 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.

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:

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

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:

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

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:

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

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

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

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.

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.

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:

$$\pi(s) = \arg\max_a Q(s, a). \tag{10}$$

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:

$$G_t = r_t + \gamma V(s_{t+1}) \tag{11}$$

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:

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

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:

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

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 $\pi^*(s) = \arg\max_a Q^*(s, a)$.

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.

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.

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. [16]

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.

IV. AlphaGo

A. 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 [18].

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.

B. 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.

C. Training Process

1) 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.

2) 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.

3) 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:

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

D. 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.

E. 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.

V. AlphaGo Zero

A. 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 Search (MCTS). [19]

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.

B. 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.

C. Training Process

1) 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.

2) Policy Iteration and self play

The agent plays games against itself using the predicted policy $P(s,a)$. 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.

3) 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 winnder 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.

D. 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.

E. Performance Benchmarks

AlphaGo Zero introduced a significant improvement in neural network architecture by employing a unified residual network (ResNet) design for its f_θ model. This replaced the separate CNN-based architectures previously used in AlphaGo, which consisted of distinct networks for policy prediction and value estimation.

The superiority of this approach is evident in the Elo rating comparison shown in fig.1. The ”dual-res” architecture, utilized in AlphaGo Zero, achieved the highest Elo rating, significantly outperforming other architectures like ”dual-conv” and ”sep-conv” used in earlier versions of AlphaGo.

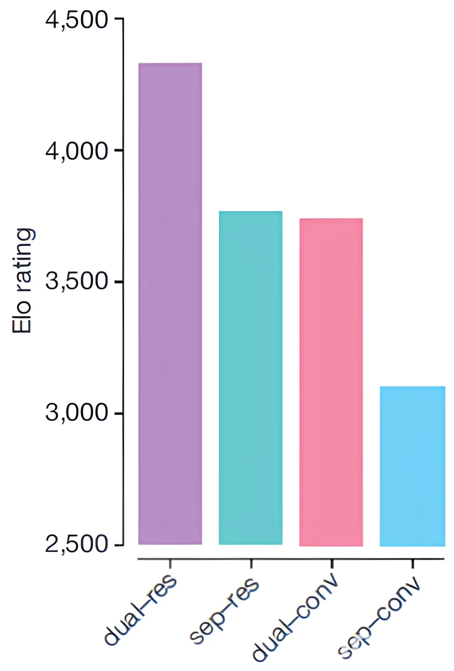

Fig. 1. Elo rating comparison of different neural network architectures.

VI. MuZero

A. 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.

B. 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.

C. Loss function and learning equations

$$s_0 = h_\theta(o_1, \ldots, o_t) \tag{25}$$
$$r_k, s_k = g_\theta(s_{k-1}, a_k) \tag{26}$$
$$p_k, v_k = f_\theta(s_k) \tag{27}$$

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

$$\nu_t, \pi_t = \text{MCTS}(s_0^t, \mu_\theta) \tag{29}$$
$$a_t \sim \pi_t \tag{30}$$

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

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

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

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

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

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

D. 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. [20]

E. Results

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

1) 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.

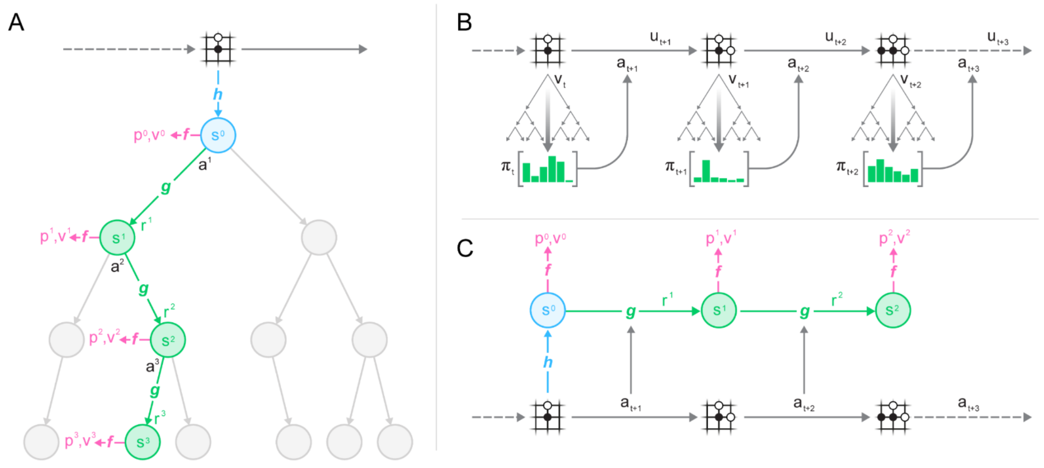

Fig. 2. (A) Represents the progression of the model through its MDP, while (B) Represents MuZero acting as an environment with MCTS as feedback, and (C) Represents a diagram of training MuZero's model.

2) 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.

3) 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. [20]

VII. Advancements

The evolution of AI in gaming, particularly through the development of AlphaGo, AlphaGo Zero, and MuZero, highlights remarkable advancements in reinforcement learning and artificial intelligence. AlphaGo, the pioneering model, combined supervised learning and reinforcement learning to master the complex game of Go, setting the stage for AI to exceed human capabilities in well-defined strategic games. Building on, AlphaGo Zero eliminated the reliance on human data, introducing a fully self-supervised approach that demonstrated greater efficiency and performance by learning solely through self-play. MuZero took this innovation further by generalizing beyond specific games like Go, Chess, and Shogi, employing model-based reinforcement learning to predict dynamics without explicitly knowing the rules of the environment. Completing on these three models, here are some of the advancements that developed from them: AlphaZero and MiniZero; and one of the most used in generating AI models, Multi-agent models.

AlphaZero

While AlphaGo Zero was an impressive feat, designed specifically to master the ancient game of Go through self-play, AlphaZero developes it by generalizing its learning framework to include multiple complex games: chess, shogi (Japanese chess), and Go. The key advancement is in its ability to apply the same algorithm across different games without requiring game-specific adjustments. AlphaZero's neural network is trained through self-play, predicting the move probabilities and game outcomes for various positions. This prediction is then used to guide the MCTS, which explores potential future moves and outcomes to determine the best action. Through iterative self-play and continuous refinement of the neural network, AlphaZero efficiently learns and improves its strategies across different games [21]. Another significant improvement is in AlphaZero's generalized algorithm, is that it does not need to be fine-tuned for each specific game. This was a departure from AlphaGo Zero's Go-specific architecture, making AlphaZero a more versatile AI system.
AlphaZero's architecture integrates a single neural network that evaluates both the best moves and the likelihood of winning from any given position, streamlining the learning process by eliminating the need for separate policy and value networks used in earlier systems. This innovation not only enhances computational efficiency but also enables AlphaZero to adopt unconventional and creative playing styles that diverge from established human strategies.

MiniZero

MiniZero is a a zero-knowledge learning framework that supports four state-of-the-art algorithms, including AlphaZero, MuZero, Gumbel AlphaZero, and Gumbel MuZero [22]. Gumbel AlphaZero and Gumbel MuZero are variants of the AlphaZero and MuZero algorithms that incorporate Gumbel noise into their decision-making process to improve exploration and planning efficiency in reinforcement learning tasks. Gumbel noise is a type of stochastic noise sampled from the Gumbel distribution, commonly used in decision-making and optimization problems.
MiniZero is a simplified version of the original MuZero algorithm, which is designed to be have a more simplified architecture reducing the complexity of the neural network used to model environment dynamics, making it easier to implement and experiment with. This simplification allows MiniZero to perform well in smaller environments with fewer states and actions, offering faster training times and requiring fewer computational power compared to MuZero.

Multi-agent models

Multi-agent models in reinforcement learning (MARL) represent an extension of traditional single-agent reinforcement learning. In these models, multiple agents are simultaneously interacting, either competitively or cooperatively, making decisions that impact both their own outcomes and those of other agents. The complexity in multi-agent systems arises from the dynamic nature of the environment, where the actions of each agent can alter the environment and the states of other agents. Unlike in single-agent environments, where the agent learns by interacting with a static world, multi-agent systems require agents to learn not only from their direct experiences but also from the behaviors of other agents, leading to a more complex learning process. Agents must adapt their strategies based on what they perceive other agents are doing, and this leads to problems such as strategic coordination, deception, negotiation, and competitive dynamics. In competitive scenarios, agents might attempt to outwit one another, while in cooperative scenarios, they must synchronize their actions to achieve a common goal [23].
AlphaGo and AlphaGo Zero are not designed to handle multi-agent environments. The core reason lies in their foundational design, which assumes a single agent interacting with a static environment. AlphaGo and AlphaGo Zero both rely on model-based reinforcement learning and self-play, where a single agent learns by interacting with itself or a fixed opponent, refining its strategy over time. However, these models are not built to adapt to the dynamic nature of multi-agent environments, where the state of the world constantly changes due to the actions of other agents. In AlphaGo and AlphaGo Zero, the environment is well-defined, and the agent's objective is to optimize its moves based on a fixed set of rules. The agents in these models do not need to account for the actions of other agents in real-time or consider competing strategies, which are essential in multi-agent systems. Additionally, AlphaGo and AlphaGo Zero are not designed to handle cooperation or negotiation, which are key aspects of multi-agent environments. On the other hand, MuZero offers a more flexible framework that can be adapted to multi-agent environments. Unlike AlphaGo and AlphaGo Zero, MuZero operates by learning the dynamics of the environment through its interactions, rather than relying on a fixed model of the world. This approach allows MuZero to adapt to various types of environments, whether single-agent or multi-agent, by learning to predict the consequences of actions without needing explicit knowledge of the environment's rules. The key advantage of MuZero in multi-agent settings is its ability to plan and make decisions without needing to model the entire system upfront. In multi-agent environments, this ability becomes essential, as MuZero can dynamically adjust its strategy based on the observed behavior of other agents. By learning not just the immediate outcomes but also the strategic implications of others' actions, MuZero can navigate both competitive and cooperative settings.

VIII. Future Directions

As mentioned earlier in the paper, The development of the AI models and systems in the field of gaming represent a good training set for the models to study the environment, address the challenges, modify the models, and achieve good results in this field, to judge whether this model is able to be implemented in real world, and how it can be implemented. The main purpose from such models, Google DeepMind, through the previous years, had been training the models to play games, and the main goal was to implement the models of reinforcement learning in real life, and benefit from them. DeepMind already started in this implementation with MuZero, and developing other models to be able to be implemented in real life directly.

MuZero's first step from research into the real world

One of the notable implementations of MuZero has been in collaboration with YouTube, where it was used to optimize video compression within the open-source VP9 codec. This involved adapting MuZero's ability to predict and plan, which it had previously demonstrated in games, to a complex and practical task of video streaming. By optimizing the encoding process, as shown in fig. 3, MuZero achieved an average bitrate reduction of 4% without degrading video quality [24]. This improvement directly impacts the efficiency of video streaming services such as YouTube, Twitch, and Google Meet, leading to faster loading times and reduced data usage for users. This implementation is called MuZero Rate-Controller (MuZero-RC). Beyond video compression, this initial application of MuZero outside of game research settings exemplifies how reinforcement learning agents can address practical real-world challenges. By designing agents with new capabilities to enhance products across different sectors, computer systems can be more efficient, less resource-intensive, and increasingly automated [24].

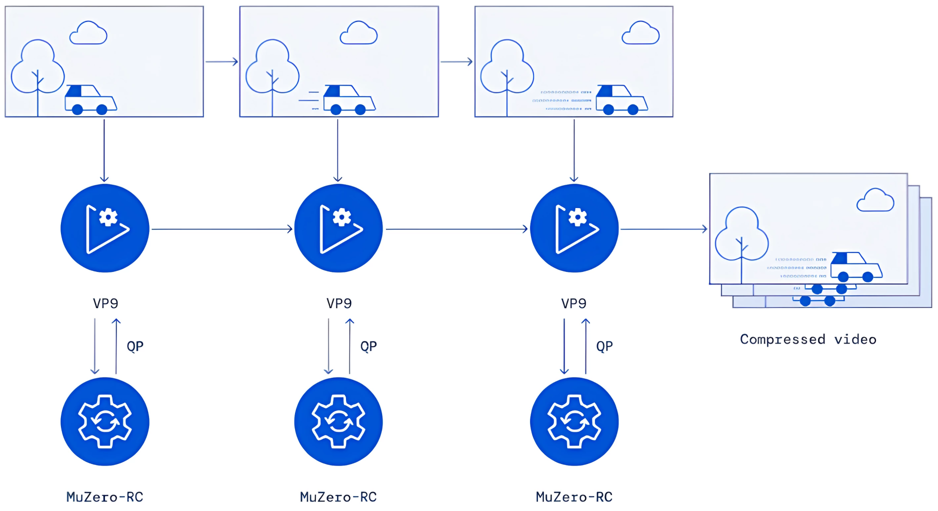

Fig. 3. MuZero Rate-Controller (MuZero-RC) optimizing the encoding process in video streaming.

AlphaFold

AlphaFold is a model developed by DeepMind that addresses one of the challenging problems in biology, which is predicting the three-dimensional structures of proteins from their amino acid sequences. AlphaFold employs advanced deep learning techniques, prominently featuring reinforcement learning, to enhance its predictive capabilities.The model operates on a feedback loop where it generates predictions about protein structures and receives rewards based on the accuracy of these predictions compared to experimentally determined structures. This process allows AlphaFold to iteratively refine its models, optimizing them to better reflect the complexities of protein folding dynamics. The architecture of AlphaFold includes deep neural networks that analyze both the sequential and spatial relationships between amino acids, enabling it to capture intricate patterns for protein conformation. By training on extensive datasets of known protein structures, AlphaFold has achieved unprecedented accuracy, often rivaling experimental methods such as X-ray crystallography and cryo-electron microscopy [25].

As shown form the previous models, how the employment of reinforcementl learning changed starting from making AI systems which play atari and strategy-based games, reaching to help in human biology and create protein structures, the enployment of reinforcement learning in games still has a long journey to be developed which helps in both real life and gaming. Google DeepMind is still working on other models which are able to be implemented in real life applications. They also developed models which use the multi-agent models in games, like AlphaStar, which is a model to play StarCraft II; but still didn't apply them in real life applications, which is a good future direction to be developed.

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

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

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

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

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

[6] DeepMind, (2015, February 12), Deep reinforcement learning.

[7] T. Schaul, J. Quan, I. Antonoglou, D. Silver, (2015). Prioritized Experience Replay. arXiv.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] K. Shao, Z. Tang, Y. Zhu, N. Li, D. Zhao, (2019). A survey of deep reinforcement learning in video games. arXiv.

[13] L. Thorndike and D. Bruce, Animal Intelligence. Routledge, 2017.

[14] R. S. Sutton and A. Barto, Reinforcement learning : an introduction. Cambridge, Ma ; London: The Mit Press, 2018.

[15] A. Kumar Shakya, G. Pillai, and S. Chakrabarty, “Reinforcement Learning Algorithms: A brief survey,” Expert Systems with Applications, vol. 231, p. 120495, May 2023

[16] Mnih, Volodymyr, et al. “Human-Level Control through Deep Reinforcement Learning.” Nature, vol. 518, no. 7540, Feb. 2015, pp. 529–533.

[17] 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.

[18] 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.

[19] 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.

[20] 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

[21] DeepMind, ”AlphaZero: Shedding New Light on Chess, Shogi, and Go,” DeepMind, 06-Dec-2018.

[22] T.-R. Wu, H. Guei, P.-C. Peng, P.-W. Huang, T. H. Wei, C.-C. Shih, Y.-J. Tsai, (2023). MiniZero: Comparative analysis of AlphaZero and MuZero on Go, Othello, and Atari games. arXiv.

[23] K. Zhang, Z. Yang, T. Başar, (2021). Multi-agent reinforcement learning: A selective overview of theories and algorithms. arXiv preprint arXiv:2103.04994

[24] ”MuZero’s first step from research into the real world,” DeepMind, Feb. 11, 2022.

[25] Jumper, J et al. Highly accurate protein structure prediction with AlphaFold. Nature (2021)