NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Review of the MuZero Algorithm with Implementation on Quoridor

Dimitrios A. Mystriotis

SUPERVISOR: Stamatopoulos Panagiotis, Assistant Professor

ATHENS

MARCH 2023

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επισκόπηση του Αλγορίθμου MuZero και η Εφαρμογή του στο Quoridor

Δημήτριος Α. Μυστριώτης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Σταματόπουλος Παναγιώτης, Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2023

BSc THESIS

Review of the MuZero Algorithm with Implementation on Quoridor

Dimitrios A. Mystriotis

S.N.: 1115201900130

SUPERVISOR: Stamatopoulos Panagiotis, Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επισκόπηση του Αλγορίθμου MuZero και η Εφαρμογή του στο Quoridor

Δημήτριος Α. Μυστριώτης

Α.Μ.: 1115201900130

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Σταματόπουλος Παναγιώτης, Επίκουρος Καθηγητής

ABSTRACT

This thesis discusses the development of the MuZero algorithm by DeepMind and its application in the game of Quoridor. The algorithm is a deep reinforcement learning algorithm that expands on previous algorithms to achieve exceptional performance in learning and planning. The key difference from its predecessors is the ability to operate in complex environments without any prior knowledge. All knowledge of game rules and dynamics is learned through interactions with the environment. The algorithm is trained through self-play, where it learns by playing games against itself, and uses the generated data to improve its performance. The thesis also discusses the environment of Quoridor, a competitive two-player strategy board game, and the application of the MuZero algorithm to it.

SUBJECT AREA: Machine Learning

KEYWORDS: Reinforcement learning, deep learning, neural networks, Markov decision process, Monte Carlo tree search, deep reinforcement learning, board games

ΠΕΡΙΛΗΨΗ

Αυτή η πτυχιακή εργασία πραγματεύεται την ανάπτυξη του αλγορίθμου MuZero από την DeepMind και την εφαρμογή του στο παιχνίδι του Quoridor. Ο αλγόριθμος είναι ένας αλγόριθμος βαθιάς ενισχυτικής μάθησης που επεκτείνει προηγούμενους αλγόριθμους επιτυγχάνοντας εξαιρετική απόδοση στη μάθηση και στον σχεδιασμό. Η βασική διαφορά με τους προγόνους του είναι η ικανότητα λειτουργίας σε πολύπλοκα περιβάλλοντα χωρίς προηγούμενη γνώση. Όλη η γνώση των κανόνων και της δυναμικής του παιχνιδιού μαθαίνεται μέσω των αλληλεπιδράσεων με το περιβάλλον. Ο αλγόριθμος εκπαιδεύεται μέσω του self-play, όπου μαθαίνει παίζοντας παιχνίδια εναντίον του εαυτού του και χρησιμοποιεί τα δεδομένα που δημιουργούνται για να βελτιώσει την απόδοσή του. Η πτυχιακή εργασία εξετάζει επίσης το περιβάλλον του Quoridor, ενός ανταγωνιστικού επιτραπέζιου παιχνιδιού στρατηγικής δύο παικτών, και την εφαρμογή του αλγορίθμου MuZero σε αυτό.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μηχανική Μάθηση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ενισχυτική μάθηση, βαθιά μάθηση, νευρωνικά δίκτυα, Μαρκοβιανή διαδικασία απόφασης, αλγόριθμος αναζήτησης δέντρου Monte Carlo, βαθιά ενισχυτική μάθηση, επιτραπέζια παιχνίδια

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Panagiotis Stamatopoulos, for his guidance and support throughout this project. His input and insights have been invaluable in shaping this thesis. I would also like to thank my family for their encouragement and advice during this time. Their support and feedback have been a key source of motivation. My friends and colleagues, especially Akis Lionis and Christos Tolis, who have also been a great source of support and inspiration, have my deepest gratitude. Finally, I would like to express my gratitude to the DeepMind team for developing the MuZero algorithm and making it available to the public. Their work has inspired this project and many others in the field of artificial intelligence.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In the field of artificial intelligence, researchers have been relentlessly searching for algorithms that can excel at making complex decisions in uncertain settings. One exciting development is the MuZero algorithm [10]. MuZero combines reinforcement learning, planning, and model-based techniques, providing a novel approach to learning and planning in environments where the underlying dynamics are unknown.

MuZero is a remarkable algorithm developed by DeepMind in 2019. Unlike its predecessor, AlphaZero [12], MuZero doesn't require explicit knowledge of the environment it's operating in. Instead, it can learn to navigate those environments on its own, without prior knowledge of their rules or dynamics. This makes MuZero a highly versatile tool that can be applied to a wide range of domains, from classic board games to complex video games and even real-world scenarios like robotics and autonomous decision-making. MuZero's ability to adapt and learn on the fly sets it apart from its predecessors and opens up new possibilities for artificial intelligence.

At its heart, MuZero uses a mix of deep neural networks and reinforcement learning to reach its goals. Through a process of self-improvement driven by reinforcement learning, MuZero learns to simulate and predict future states, anticipate rewards, and plan the best actions. What makes MuZero special is its ability to learn a model of the environment's dynamics just from interaction data, without needing to know the underlying rules - a feature that's very useful in areas where that knowledge is hard to get or doesn't exist.

2. BACKGROUND AND RELATED WORK

2.1 Neural Networks

An Artificial Neural Network (ANN) is a computing model that attempts to mimic the information processing of the human brain. It is a non-programmable, brain-style information processing system with adaptive, self-organizing, and real-time learning features [18]. An artificial neural network consists of a large number of perceptrons, commonly referred to as nodes, or neurons, connected to each other. Each of these nodes represents a specific output function called the activation function [18]. These nodes are linked via weighted connections, which influence the signal transmissions between neurons and are responsible for the network's ability to retain information.

The overall behavior of an artificial neural network depends on how the neurons are connected, the weight values, and the activation function. They can identify trends and patterns in data, and make forecasts and predictions. ANNs are developed for applications such as data classification or pattern recognition through the learning process, where adjustments to the synaptic relationships between neurons occur, similar to the learning process in the human brain [1]. ANN modeling involves the use of interconnected processing units that mimic neurons, and optimization techniques like gradient descent to update weights in the models for better performance.

ANNs have found applications in various fields, including informatics, medicine, economy, control, transportation, and psychology [18]. They also exhibit potential advantages in tasks that involve pattern recognition, prediction, classification, and optimization in various other domains of science and technology. In particular, they can prove useful in applications where their self-learning capabilities are needed to process changing conditions and new information, achieved by processing complex data to automatically extract meaningful features.

2.1.1 Neural Network Architecture

Neural networks are composed of individual neurons which perform basic computations. Each neuron follows a two-step process [2]: in the first step, it calculates a real value by combining input values with connection weights, usually through a weighted linear operation. In the second step, an activation function is applied to this calculated value, producing a scalar output. This output becomes the input for the next layer of neurons.

In an ANN, the neurons are organized into layers [18], with the input layer receiving external data, the output layer producing the final result, and hidden layers handling intermediate information. Neurons in a layer receive connections from all neurons in the previous layer, with each connection weighted to determine its impact. As stated earlier, computation within a layer involves summing the outputs from the previous layer to each node of the current layer, scaling them by the weights of each connection and applying an activa-

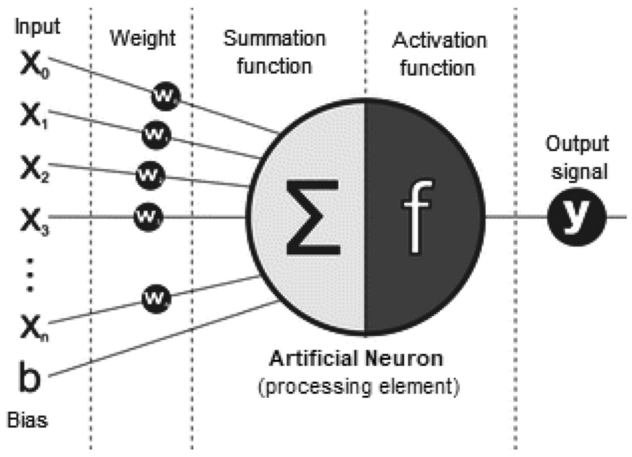


Figure 2.1: Schematic representation of the mathematical model of an artificial neuron [9].

tion function. This sequential computation progresses from the input to the output layers, resulting in the network's prediction or decision, as detailed below:

1. Input layer: The input layer of a neural network receives the initial data or signals from external sources. Each neuron in this layer represents a feature or input that is fed into the network.
2. Hidden layers: The hidden layers are located between the input and output layers and are responsible for processing the input data. These layers perform complex computations and extract relevant features from the input data through a series of weighted connections and activation functions.
3. Output layer: The output layer of a neural network produces the final prediction or output based on the processed information from the hidden layers. The number of neurons in the output layer depends on the type of task the neural network is designed to perform.

Neurons in the network work together in groups to process and change the input data, which then helps produce the output. The way these neuron groups are set up and connected enhances the network's ability to learn and make predictions based on the input. Each layer examines the data and represents different levels of detail, allowing the network to carry out complex tasks like classification or regression.

To train a network, a data set is used called the train set, which contains inputs and their expected outputs as pairs. During training, the weights of the node connections are adjusted to minimize the error between the network's prediction of output and the expected output from the training set. Activation functions, as will be discussed later, help introduce non-linearity into the network, allowing it to learn complex patterns and relationships in the data.

Overall, the architecture of a neural network is designed to capture and process information by learning patterns based on its training data and making predictions or decisions.

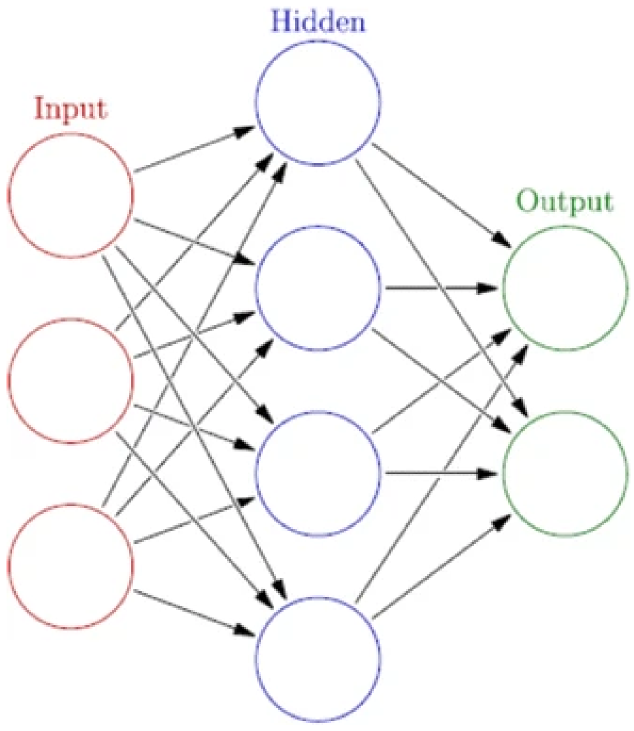


Figure 2.2: Visualization of an Artificial Neural Network [18].

2.1.2 Activation Functions

The activation function [2] is crucial in the operation of individual neurons within a neural network, as it directly determines the output of each neuron. The activation function introduces non-linearity and adaptability to the network's computations, shaping how individual neurons behave. Designing and implementing the activation function properly can significantly impact the network's learning ability, performance, and overall effectiveness in tasks like classification, regression, and pattern recognition.

Historically, various terms like transfer function and output function have been used to describe activation functions in the literature. However, the modern understanding is that the activation function plays a pivotal role in determining a neuron's state based on its inputs, weights, and biases.

The introduction of new and adjustable activation functions, like ReLU and Leaky ReLU, has significantly improved network performance by addressing issues like vanishing gradients [2]. By allowing the modification of activation functions during training, researchers have been able to explore fresh ways to enhance neural network capabilities and achieve better results through the use of trainable activation functions.

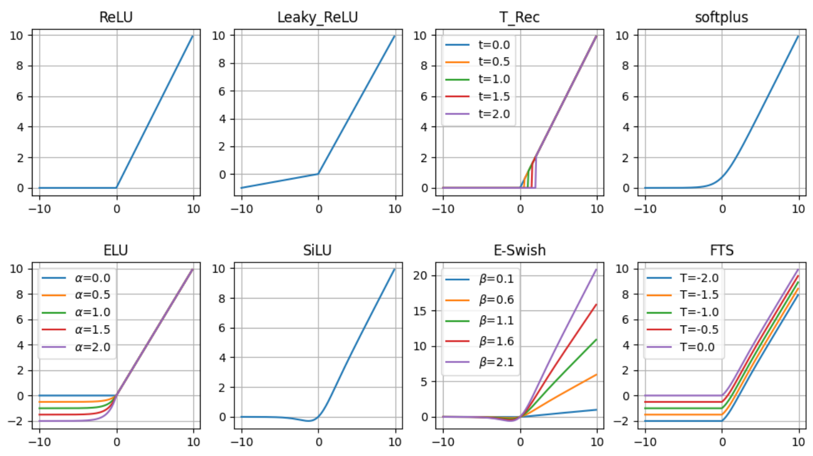


Figure 2.3: Examples of rectifier-based activation functions used in neural networks [2].

2.1.3 Deep Learning

Deep learning (DL) is a section of machine learning (ML) that uses ANNs with multiple layers to find and model patterns from complex and multi-dimensional data [5]. This approach has demonstrated the ability to create abstract representations using the training data, a process that is facilitated by the multiple layers in the network. Specifically, the use of multiple hidden layers allows the network to learn complex representations of the input data, which can then be used to make predictions or decisions.

DL expands on the idea of traditional neural networks, to mimic the human brain's ability, by creating a more complex and deeper network with more connections. This more closely resembles our current understanding of the human brain's structure and function, and it has been shown to be more effective in learning from complex data.

Deep learning methods have been particularly successful in various applications such as speech recognition, computer vision, pattern recognition, recommendation systems, and natural language processing [1]. Networks with multiple layers, specifically convolutional neural networks as it will be discussed in later sections, have shown especially excellent performance in such applications through the use of a DL structure.

In the next subsections, we will discuss some of the most common types of neural networks used in deep learning, categorized by the structure of their layers.

2.1.3.1 Feedforward Network

A Feedforward network [8] generally consists of a series of linear layers whose nodes calculate the result of its inputs and pass it to its successor nodes. Creating a graph from the connections of the nodes would result in an acyclical-directed graph starting at the input and ending at the output nodes. A node of the network is called a unit and applies an activation function to its inputs. Specifically, for any neuron j, let us denote the activation function as g_j, and its output as a_j. The output of the neuron is calculated as

$$a_j = g_j(w^T x) \tag{2.1}$$

where x is the input vector and w the weight vector. The input vector also contains an auxiliary value set to $+1$ with a weight ($w_{0,j}$) being applied to it. This allows for the output to be non-zero even if all other input values are zero.

2.1.3.2 Convolutional Network

Convolutional neural networks (CNNs) [8] apply a method called convolution to sets of pixels or an area of the input, more generally. The method of convolution is essentially the dot product of the kernel k and the values x of the local area it is applied. The kernel thus, acts as weights and can encode patterns important to the output.

The process happening at each layer of a convolutional network is, that the kernel passes over every point of the input image, this applies a filter to the point and nearby pixel within length l from the point. The addition of stride s at a value > 1 means the kernel will be moving s pixel each time, this also results in a decrease in the size of the output. Note that if $s > l + 1$ the convolution will be ignoring information. At the end of this process, the output is passed through a nonlinear activation function before being passed to the next layer.

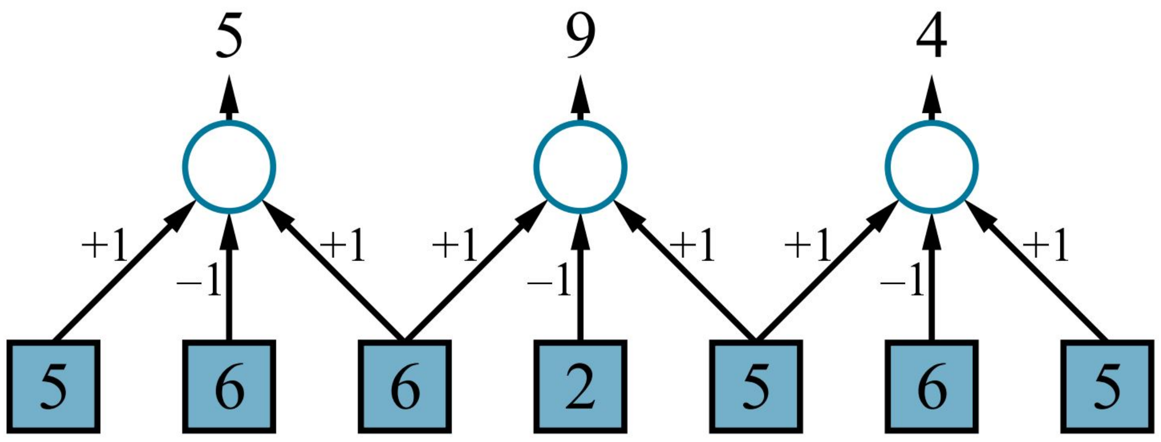


Figure 2.4: Demonstration of a one-dimensional convolution operation [8].

An example of the operation of a convolutional layer is shown in the figure above. The image, in this case, is a one-dimensional array of values, and the kernel has a size of $l = 3$, meaning it is applied to three values each time. The kernel with values $[+1, -1, +1]$ is applied to the input image, and the output is calculated by taking the dot product of the kernel and the input values. A stride of $s = 2$ is used meaning the kernel centers on every second value of the input. The application of the activation function is not shown in the figure, but it is a crucial step in the process.

2.1.3.3 Residual Network

Residual networks [4] are based on the idea of creating shortcuts for the input values and letting mapping that uses multiple stacked layers calculate the residual function $F(x, W)$. The base component of these networks is called a block, using one or multiple of them to create a network. The output of a residual block with the layer weights denoted as W is

$$y = F(x, W) + W_s x. \tag{2.2}$$

Where x, and y are the input and output respectively, $F(x, W)$ is the residual, and W_s is a linear projection by the shortcut connection to match the dimensions of the residual. The simplest form of a residual block would use a single linear layer with the same input and output shapes. Such a block would have a residual function of $F(x, W) = Wx$. The residual block's output would then be $y = Wx + x$, which is the sum of the linear layer's output and the input. The figure below is a diagram of the structure of a residual block with two layers instead of one.

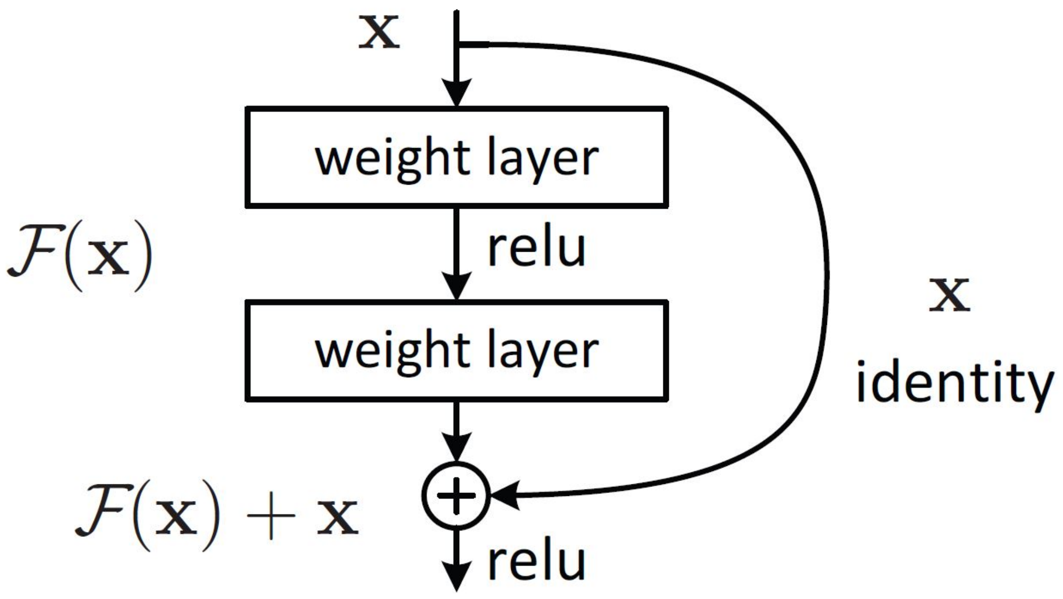

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2.5: Residual learning: a building block [4].

The examples given use linear layers, but the same principle can be applied to any type of layer, such as CNN. The residual block allows for the network to learn the residual function, which is easier to optimize than the original function.

2.2 Reinforcement Learning

The continuous development of deep neural network technologies, such as convolutional neural networks (CNN) and recurrent neural networks (RNN), has had a significant influence on deep learning [16]. These advancements have contributed to deep learning becoming more influential and impactful in various fields such as image recognition and text processing. For example, deep learning has demonstrated strong fitting and representation capabilities when dealing with high-dimensional data. Additionally, the use of deep learning techniques has further enhanced the capabilities of deep neural networks in tasks like data generation and translation. These advancements have paved the way for the integration of deep learning with reinforcement learning to achieve powerful end-to-end learning control capabilities.

Reinforcement learning (RL) [8] is an ML technique that allows agents to train through interactions with the environment and the rewards they receive. Such algorithms are unique in that they learn through a trial-and-error process, where they receive feedback on their actions and adjust their strategies accordingly. RL algorithms aim to maximize the cumulative reward an agent receives over time by learning the optimal policy for decision-making. Specifically, when dealing with an MDP, the agent does not attempt to predict the outcome given a chain of actions, but rather it performs actions with the goal of finding how to maximize the expected return.

The addition of deep neural networks into RL algorithms gives us a very promising technique that is called Deep Reinforcement Learning (DRL). DRL has already been applied to a variety of problems across different domains, showcasing its ability to achieve superhuman results in a variety of environments. The potential of these algorithms has been observed in applications such as robot control, games, natural language processing (NLP), autonomous driving, recommendation systems, and computer vision [16]. In addition to these domains, DRL has also been expanded into several subgroups of algorithms, namely hierarchical reinforcement learning, multiagent reinforcement learning, and imitation learning, among others. The integration of deep learning with reinforcement learning in DRL has enabled the processing of high-dimensional inputs and the approximation of values or policies to solve problems with excessive state spaces and continuous action spaces. By harnessing the representation capabilities of neural networks, DRL has demonstrated its ability to address a wide range of complex challenges across different problem domains.

Training agents with high-dimensional inputs has presented a significant challenge to a machine learning algorithm, a good example of such cases is games. When an agent receives inputs from a video game these inputs are typically high-dimensional, making decision-making complex and difficult. To address this challenge, researchers have leveraged deep neural networks as function approximators to optimize policies effectively. Deep neural networks have the capability to extract meaningful features automatically from the high-dimensional inputs, enabling the agent to make informed decisions based on these features [11].

By utilizing deep neural networks in policy optimization, researchers can train agents to update their policies end-to-end to maximize their return in the game environment. This learning mechanism involves updating the policy through deep reinforcement learning methods, which have shown great success in training agents to make decisions in various video games, ranging from classical Arcade games to first-person perspective games and multi-agent real-time strategy games [11]. Through the continuous optimization of policies using deep neural networks, agents are able to improve their decision-making processes and achieve superhuman performance in different game environments.

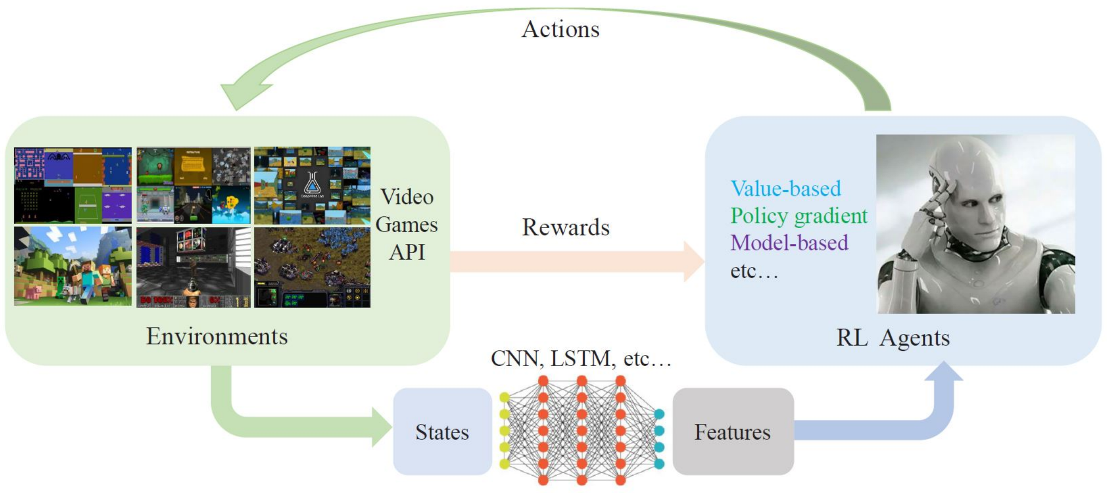


Figure 2.6: The framework diagram of the typical DRL for video games [11].

The deep learning model takes input from video games API and extracts meaningful features automatically. DRL agents produce actions based on these features and make the environments transfer to the next state. This structure is not excessive for video games but is found in DRL algorithms generally.

Overall, the collaboration between deep neural networks and reinforcement learning techniques enables agents to effectively navigate high-dimensional inputs, optimize policies for decision-making, and achieve excellent performance.

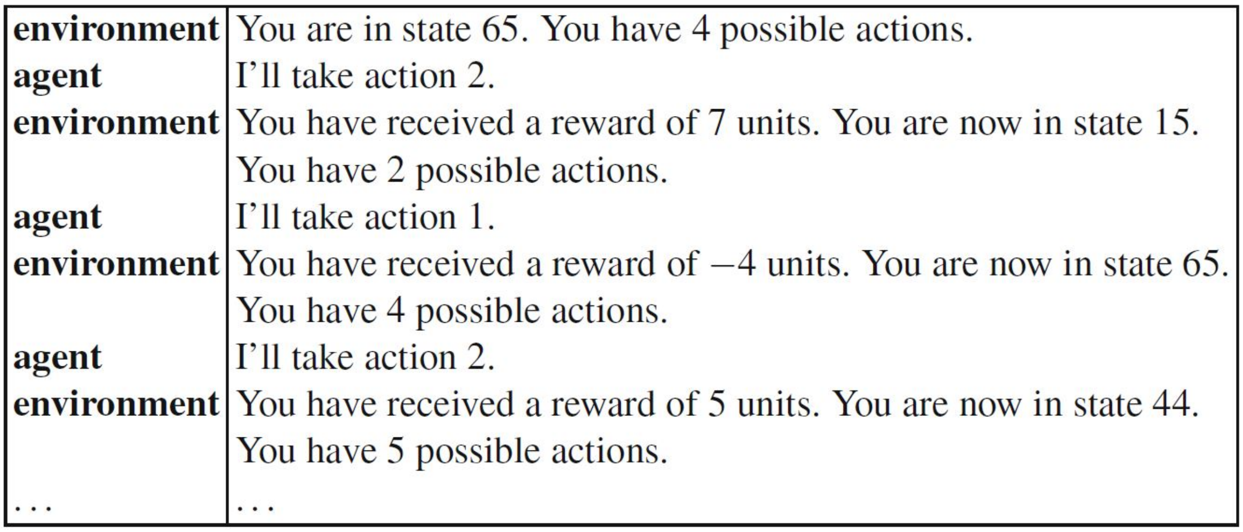
2.2.1 Markov decision process

A Markov decision process (MDP) is a formal framework for decision-making in sequential environments [14]. It is situated between supervised learning and unsupervised learning and is often used in reinforcement learning.

In an MDP, an environment is modeled as a set of states, and actions can be performed to control the system's state. The goal is to control the system in such a way that some performance criterion is maximized. MDPs are widely used in various domains, including planning problems, learning robot control, and game playing.

The basic components of an MDP are the state set (S), action set (A), transition function (T), and reward function (R). The transition function defines the probability of transitions from one state to another given a specific action, and the reward function assigns a reward to each state-action-state transition. These sets and functions together define the model of the MDP.

Typically, MDPs are represented as state transition graphs, where nodes represent states, and directed edges denote transitions. The MDP framework assumes the Markov property, meaning that the current state contains enough information to make optimal decisions, and the future state depends only on the current state and action.



environment	You are in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 7 units. You are now in state 15. You have 2 possible actions.
agent	I'll take action 1.
environment	You have received a reward of -4 units. You are now in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 5 units. You are now in state 44. You have 5 possible actions.
...	...

Figure 2.7: Example of interaction between an agent and its environment, from an RL perspective [14].

Markov decision processes provide a formal and intuitive way to model and solve sequential decision-making problems. Various algorithms, such as dynamic programming and reinforcement learning, can be used to compute optimal policies in MDPs and learn how to act in stochastic environments.

The optimal policy, denoted π^* in MDPs, is calculated by using the Bellman optimality equation. The value function of a state s under an optimal policy $V^{\pi^*}(s) = V^*(s)$ must be equal to the expected return for the best action in that state.

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{2.3}$$

The Bellman optimality equation states that the optimal value function of a state is equal to the maximum expected return over all possible actions in that state, taking into account the transitions to the next states and the rewards received. The optimal policy selects the action that maximizes the expected return based on the optimal value function.

To compute the optimal policy, algorithms such as value iteration or policy iteration can be used. These algorithms iterate between policy evaluation, which estimates the value function of a policy, and policy improvement, which updates the policy based on the estimated value function. Through these iterations, the algorithms converge to the optimal policy. Value iteration directly updates the value function at each iteration, while policy iteration updates the policy based on the value function.

```
**Require:** $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
  {POLICY EVALUATION}
  **repeat**
    $\Delta := 0$
    **for each** $s \in S$ **do**
      $v := V^{\pi}(s)$
      $V(s) := \sum_{s'} T(s,\pi(s),s')\left(R(s,\pi(s),s') + \gamma V(s')\right)$
      $\Delta := \max(\Delta, |v - V(s)|)$
  **until** $\Delta < \sigma$
  {POLICY IMPROVEMENT}
  policy-stable := true
  **for each** $s \in S$ **do**
    $b := \pi(s)$
    $\pi(s) := \arg\max_a \sum_{s'} T(s,a,s')\left(R(s,a,s') + \gamma \cdot V(s')\right)$
    **if** $b \neq \pi(s)$ **then** policy-stable := false
  **if** policy-stable **then** stop; **else** go to POLICY EVALUATION
```

Figure 2.8: Policy Iteration Howard (1960) [14].

The optimal policy can also be represented using the Q-function, which is the expected return starting from a state, taking a certain action, and following the optimal policy thereafter.

$$Q(s,a) = \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{2.4}$$

The Q-function can be used to select the optimal action in each state, without the need for the transition function. In case the transition and/or reward functions are unknown, ML algorithms can be used to learn the Q-function directly through interactions with the environment.

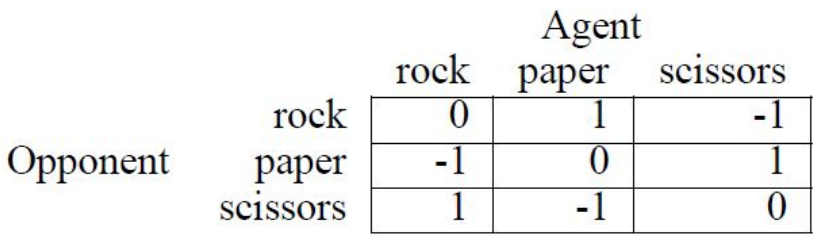
2.2.2 Markov Games

Markov games, also known as stochastic games, extend the framework of MDPs to include multiple agents with shared or opposing goals that interact with each other during play [7]. In a Markov game, same as in MDPs, there is a set of states and action sets for each agent. The state transitions depend on the current state and the actions taken by those agents. Each agent also has its own reward function and aims to maximize its expected sum of rewards.

Unlike MDPs where a single agent interacts with the environment, Markov games, as stated, involve multiple agents interacting with each other. This means that Markov games account for the presence of other agents with their own goals, thus changing the dynamics of the environment and the interactions between agents. The result is that the concept of optimal policies in Markov games differs from MDPs as well. While MDPs often have deterministic optimal policies, Markov games may require probabilistic ones. This is because the choice of action in Markov games depends on the uncertainty of an opponent's move and the need to adapt to their behavior.

In Markov games, the presence of multiple agents introduces interactions like cooperation or competition between them. One example of this can be seen in two-player zero-sum games, where two agents have diametrically opposed goals. In such games, each agent's optimal policy depends not only on its own actions but also on the opponent's actions. The strategies each agent chooses need to consider the potential moves and goals of the opponent.

A simple example is the game of rock-paper-scissors, where two agents compete against each other to win the game [7]. The optimal policy for each agent depends on the opponent's strategy, and the agents are required to adapt their policies to counter the opponent's moves or lose, see figure 2.9.



		Agent		
		rock	paper	scissors
Opponent	rock	0	1	-1
	paper	-1	0	1
	scissors	1	-1	0

Figure 2.9: The matrix game for "rock, paper, scissors." [7].

The reward matrix shows the rewards for an agent based on their and their opponent's actions. Where 1 is a victory, 0 is a draw, and -1 is a victory for the opponent.

Therefore, the presence of multiple agents in Markov games changes the dynamics of the interactions and competitions, requiring agents to consider the strategies and potential actions of their opponents when determining their own optimal policies.

In summary, Markov games provide a framework for modeling and reasoning about multi-agent environments. They allow for interactions between adaptive agents and offer insights into finding optimal policies in such complex settings.

The game addressed in this thesis, Quoridor, is a Markov game, and our algorithm is expected to handle such environments effectively.

2.2.3 Monte-Carlo tree search

Monte Carlo Tree Search (MCTS) is a computational method commonly used for search and planning tasks [15]. It combines the Monte Carlo search with an incremental tree structure. The core principle of MCTS is to identify optimal actions by sampling a given model, typically a MDP. By employing an internal mental simulator, MCTS algorithms can achieve notable performance improvements compared to traditional search techniques.

The MCTS planning process entails the iterative construction of an asymmetric search tree, guided by an exploratory action-selection policy that refines with each iteration. This process involves action selection, followed by expansion of the tree to the child nodes, playout simulation until reaching a terminal state, and finally the propagation of feedback upwards through the tree [15]. MCTS algorithms frequently leverage many robust search methods, informed playout policies, and domain-specific enhancements to enhance their efficacy.

MCTS has gained widespread acceptance in game playing, particularly for games like Go, due to its demonstrated success. MCTS methods can be conceptualized as mechanisms for dynamically altering the representation of the state space during computation, thereby enabling learning and planning. The MCTS framework incorporates elements from machine learning and reinforcement learning, such as direct-lookup tables and adaptive representations, to further improve its performance.

The use of an internal mental simulator [15] is a key catalyst that contributes to the performance of MCTS algorithms in decision-making processes by allowing them to simulate the given problem and incrementally build an asymmetric search tree. This simulator guides the algorithm in the most promising direction by using its action-selection policy, which is computed iteratively and improves with an increase in the number of iterations. By simulating possible scenarios and outcomes, MCTS algorithms can make informed decisions on which actions to take, leading to improved performance in decision-making processes. Moreover, employing a generative model within the internal simulator facilitates the creation of adaptive representations and offers efficient strategies for tasks characterized by large state spaces. A simulator in MCTS algorithms plays a crucial role in enhancing their performance by enabling them to play out and explore different possible actions, ultimately aiding in effective decision-making processes.

2.3 AlphaZero

AlphaZero is a reinforcement learning algorithm developed by researchers from DeepMind [12]. It is capable of mastering the games of Go, shogi (Japanese chess), and chess at a superhuman level. Unlike previous chess programs, AlphaZero does not rely on human input or pre-programmed strategies; it learns to play the games solely based on the basic rules of the games through deep reinforcement learning. The algorithm uses MCTS, in contrast with previous chess programs which relied on different search algorithms, to achieve greater scalability with thinking time.

AlphaZero outperformed both human and computer opponents in chess and shogi, showcasing its ability to achieve superior results in complex games with greater computational complexity and asymmetric rules. The algorithm utilized deep neural networks to evaluate positions and make moves, providing a powerful representation that combines neural network representations with a powerful, domain-independent search. AlphaZero's MCTS search was able to focus more selectively on the most promising variations, which is considered a more ”human-like” approach to search.

AlphaZero's performance using reinforcement learning with self-play games as training data showed significant progress in a short amount of time. It outperformed established chess engines like Stockfish and Elmo after only a few days of training. In 100 game matches against Stockfish, Elmo, and the previous version of AlphaGo Zero, AlphaZero achieved winning results in chess, shogi, and Go [12].

AlphaZero's search algorithm is a significant departure from traditional search algorithms used in computer chess programs. In AlphaZero the algorithm utilizes a general-purpose MCTS algorithm [12].

During each search, AlphaZero conducts a series of simulated games of self-play, starting from the root state and traversing the tree to the leaf state. At each state, a move is selected based on low visit count, high move probability, and high value according to the current neural network. This process continues until a leaf state is reached, and the search returns a probability distribution over moves.

One of the advantages of using MCTS in AlphaZero is its ability to combine deep neural network representations with a powerful, domain-independent search. The algorithm leverages non-linear function approximation through the deep neural network, which provides a more powerful representation compared to traditional linear function approximations used in typical chess programs. By averaging the approximation errors in MCTS, AlphaZero is able to effectively evaluate positions and make decisions in complex game scenarios.

In terms of training the algorithm, the parameters of the deep neural network in AlphaZero are trained through self-play reinforcement learning. This involves playing games by selecting moves using MCTS and scoring the terminal position to compute the game outcome. The neural network parameters are then updated based on minimizing error and maximizing the similarity of the policy vector to the search probabilities.

The algorithm's success in mastering these games without human intervention was an important milestone in AI research, showcasing the potential of reinforcement learning and neural networks to achieve superhuman performance in complex tasks. It also is a good showcase that human knowledge, such as a preexisting training data set, is not needed and that a model is able to generate the required data on its own.

2.4 Environments

It is important to discuss the environment in which the algorithm will be implemented. The environment is the context in which the agent operates and interacts with the world or, in many cases, a game. There are many factors that differentiate the environments, such as the action space, the observation space, the reward system, and the dynamics of the environment. Any machine learning algorithm, especially reinforcement learning algorithms, must be able to handle these differences and adapt to the environment in which it is implemented. The ability to handle different environments without major changes is of particular importance in the development of general-purpose algorithms.

The complexity of environments can vary greatly, from as simple as tic-tac-toe to complex video games. The complexity plays a key role in performance as well as the resources used to train the algorithm. Visually rich environments, such as Atari games prove a challenge for model-based algorithms, while games with a need for a long-term strategy, such as chess, prove a challenge for model-free algorithms [10].

2.4.1 Quoridor

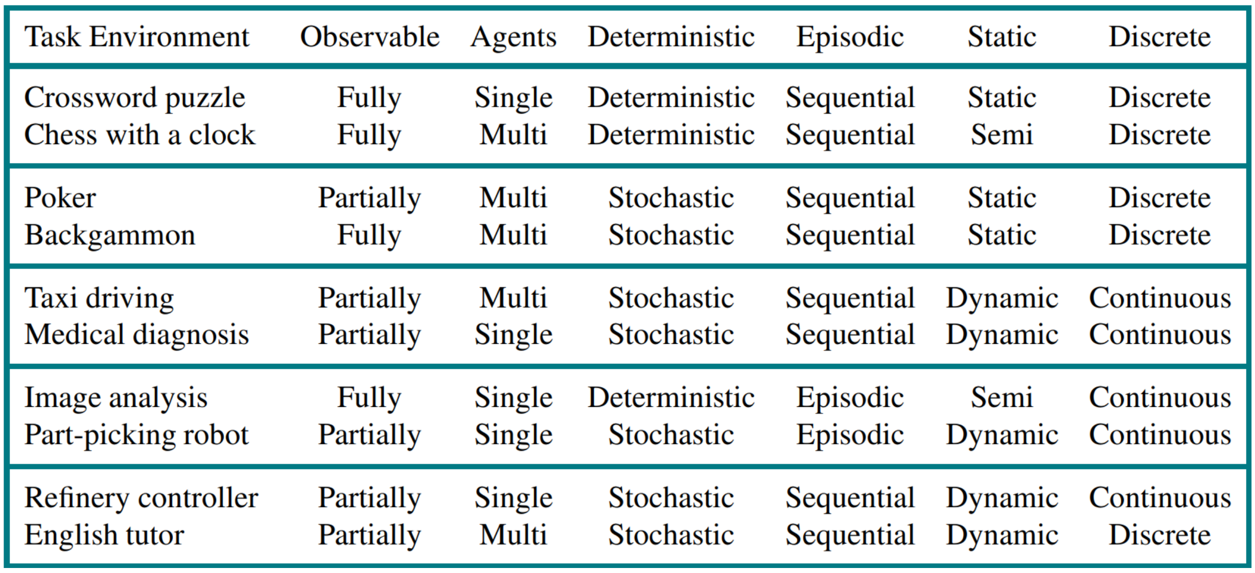
Quoridor is a competitive two-player strategy board game that was developed by Mirko Marchesi in 1997. The game is played on a 9x9 board, and the objective is to move your pawn to the opposite side of the board. Each player has a pawn and a set of 10 walls that can be placed on the board to block the opponent's path. The game is played in turns, and players can either move their pawns or place a wall on the board. This game is a Markov game, as it involves two players with opposing goals and interactions between them. The game's complexity and strategic nature make it an interesting environment for reinforcement learning algorithms. The key challenge in Quoridor is wall placement, which requires long-term planning and strategy to block the opponent's path and reach the goal. The game's strategic nature and the need for long-term planning makes it an interesting environment to evaluate a reinforcement learning algorithm.

2.5 Environment Classes

The amount of environments an agent can interact with is virtually unlimited, thus separating them into categories can help create or transfer algorithms that are designed to face similar challenges. Environments can be separated into different categories based on eight axes, which are: [8]

1. Fully Observable or Partially Observable
2. Single-agent or Multi-agent
3. Competitive or Collaborative
4. Deterministic or Stochastic
5. Episodic or Sequential
6. Static or Dynamic
7. Discrete or Continuous
8. Known or Unknown

Every environment will fall on one side of each of these axes, and the combination of these axes will define the environment and the challenges it presents to the agent. There are several examples of environments, in our case, Quoridor will be used as an example to illustrate these axes as adding rules to the game can change the environment completely.



Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.10: Examples of task environments and their characteristics [8].

Fully Observable or Partially Observable: In a fully observable environment, the agent has access to the complete state of the environment at each time step. In a partially observable environment, the agent has access to only a partial view of the environment. In this case, an internal state is useful to keep track of the unobserved parts. Quoridor is a fully observable environment, as the players can see the entire board and the positions of the pawns and walls. A variant of the game where walls block the view of the board, would turn the game into a partially observable environment.

In the case where there is no information about the environment, the environment is called unobservable. The agent has no access to the state of the environment, making it difficult (but not impossible) to navigate.

Single-agent or Multi-agent: In a single-agent environment, there is only one agent controlling all actions and interacting with the environment. A multi-agent environment has multiple agents interacting not only with the environment but with each other as well. Quoridor is a multi-agent environment, as it involves two players with opposing goals and interactions between them. To turn the game into a single-agent environment, the opponent would have to be removed, and the game would have to be modified to have a single player reach a goal, making it quite uninteresting.

Competitive or Collaborative: Multi-agent environment can be further separated into competitive or collaborative. In a competitive environment, the agents have opposing goals and compete against each other. Collaborative environments, on the other hand, have agents that share a goal, and they must work together to achieve it. Quoridor is a competitive environment, as the two players have opposing aims, each trying to reach the edge first. To turn it into a collaborative environment, the game can be turned into a team game, with two players on each side, that must work together to reach the goal. This category of games is a hybrid of competitive and collaborative, where cooperation with your team is required, but there are opposing goals with the enemy team.

Deterministic or Stochastic: Deterministic environments, the following state is completely determined by the current state and the action taken. In stochastic environments, the next state is not completely known by the current state and action, meaning the same chain of actions could have different outcomes. Stochastic environments have known probabilities of the next state, which separates them from non-deterministic environments when the chances of the results are unknown. Quoridor is a deterministic environment, as an action has exactly one expected result. Quoridor could become a stochastic environment by adding a chance of a wall falling or making the pawns move in a random direction using dice.

Episodic or Sequential: In episodic environments, the agent is called to take an action based on a single observation that is completely unrelated to other actions or states. In sequential environments, the agent must take a sequence of actions based on the observations at each state, where the current action affects the next observation. Quoridor is a sequential environment, as the players have to make a series of moves to play and win. Most games are sequential, but the evaluation of a game state is an episodic task, so the evaluation of the board state of Quoridor is episodic.

Static or Dynamic: In static environments, the environment does not change while the agent is making a decision. In dynamic environments, the environment is ever-changing and doesn't wait for an agent's action. On this axis, Quoridor is a static environment, as the board does not change while the players are making their moves. The game could be turned into a dynamic environment by adding a time limit for each move.

Discrete or Continuous: In discrete environments, the state and action spaces are finite, while in continuous environments, they are infinite. Quoridor is a discrete environment, as the board has a finite number of states it can be in and the actions are also limited. To transform Quoridor into a continuous environment, pawn movement could be made to any position within a circle around the pawn. Continuous environments are more difficult to handle, as they require more complex algorithms to handle the infinite state and action spaces.

Known or Unknown: In known environments, the agent has complete knowledge of the environment's dynamics and rules. In unknown environments, the agent has no knowledge of the environment's dynamics and rules and has to learn them from scratch. This is not a distinction as much of the environment as of the agent's knowledge of it. Quoridor is a known environment, as the rules and dynamics of the game should be known to the players. The game could be turned into an unknown if you don't read the rules. In this case, the agent we are going to be examining, MuZero, is designed to handle unknown environments, in contrast to previous algorithms like AlphaZero, which require knowledge of the environment's rules.

3. MUZERO

MuZero is a machine learning algorithm developed by DeepMind. It expands on previous work like AlphaGo and AlphaZero, by using an ANN to learn the environment dynamics from interacting with it [10]. The central innovation of this new approach is to allow the algorithm to evaluate the policy and value functions, and the expected reward, without any prior knowledge. The hidden states of the model are allowed to create representations of the observation and find game dynamics without constraints. With this method ”the agent can invent, internally, any dynamics that lead to accurate planning.” [10], giving the ability to operate in complex environments.

MuZero aims to solve the shortcomings of previous algorithms by using ANNs to predict the relevant quantities for planning: the policy, the value function, and the intermediate reward. In this way, the algorithm expands on previous value equivalent models and value prediction networks, which model an MDP in a hidden layer, by also modeling and predicting the policy [10].

3.1 MuZero Algorithm

MuZero is a deep reinforcement learning algorithm that combines AlphaZero’s MCTS search algorithm with deep neural networks to achieve exceptional performance in learning and planning. This is achieved through the combination of a representation function, a dynamics function, and a prediction function [10]. The latter two are used in each recurrent step of the search, to make predictions on the policy, value, and reward quantities. The representation function is used to encode the starting internal state (or root) of the MCTS. These functions use deep neural networks to compute their outputs.

While playing in an environment, at each step, the algorithm will perform a search to plan the next action in the MDP, which will use the predicted quantities of the dynamics function. Any search algorithm may be used here, but MuZero uses one similar to AlphaZero’s [10]. This powerful lookahead algorithm uses the predicted quantities to improve the policy and value function, after which it will be picked according to the new policy.

MuZero avoids the need for human-made data by executing a data generation process that involves self-play. This means MuZero learns by playing games against itself. This generates a large amount of game data which is used for training the model. During the process, MuZero selects actions based on its learned policy network. After each action selection, the search algorithm is used to explore the game tree and calculate possible future outcomes. Later this data, generated during self-play, is used as training data to update the weights of the policy and value networks. In this way, MuZero interacts with the environment and learns to predict the value of a game state and improve its results and strategy.

As MuZero is trained on the games generated by self-play, the set of games that were created is sampled and used as the training set. The parameters of the neural networks are trained to fit the set. With a regularly updated network, new games are generated with, hopefully, better results. This new experience is used to train the network again on the new data. The cycle continues, changing between self-play and training, improving the performance of the algorithm in the game.

3.1.1 Dynamics function

The dynamics function [10] is a recurrent process that computes an immediate reward and an internal state at each simulated step, given the previous hidden state and the action taken for the transition. This is used to predict the expected reward and state transition for a given state and action, similar to an MDP model. It uses the same architecture as the prediction function, specifically a deep neural network with residual blocks. During training, the network is fitted to predict the function outputs using the self-play data.

3.1.2 Prediction function

The prediction function [10] in the MuZero algorithm is responsible for computing the policy and value function based on the internal state (hidden state) of the model. It is used during play, at each step of the search, to determine the optimal policy and predict the value function. These predictions are essential for planning and decision-making in the MuZero algorithm. The prediction function is trained end-to-end along with the representation and dynamics functions to accurately estimate these quantities.

3.1.3 Representation function

The representation function [10] in the MuZero algorithm is responsible for encoding the observation of the environment into a hidden state. This hidden state is then used by the dynamics function and the prediction function to predict relevant future quantities for planning. The advantage to this approach is the reduction of the observation to a few features, thus condensing the important information. The representation function is initialized with past observations and does not have any special semantics beyond supporting future predictions [10].

3.1.4 Loss

The loss function of the MuZero algorithm [10] is used to train the neural networks to predict the policy, value, and reward quantities accurately. The loss function is a combination of the policy loss, value loss, and reward loss, and is used to update the weights of the neural networks during training. The loss at a time step t, with the current model weights θ, is defined as:

$$l_t(\theta) = \sum_{k=0}^{K} l_{policy}(\pi_{t+k}, p_t^k) + \sum_{k=0}^{K} l_{value}(z_{t+k}, \nu_t^k) + \sum_{k=1}^{K} l_{reward}(u_{t+k}, r_t^k) + c\|\theta\|^2 \tag{3.1}$$

which is the sum of the policy, value, and reward losses over K hypothetical future steps. Specifically, l_{policy} is the policy loss, with π_{t+k} being the search policy at future step $t + k$ and p_t^k the predicted policy at that step. l_{reward} is the reward loss, with u_{t+k} being the intermediate rewards and r_t^k the predicted reward. The intermediate reward is also used to calculate the value loss, with v_t the search value at time step t. l_{value} is the value loss, with $z_{t+k} = u_{t+k+1} + \gamma u_{t+k+2} + \ldots + \gamma^{n-1} u_{t+k+n} + \gamma^n v_{t+k+n}$ being the target value and ν_t^k the predicted value. Lastly, $\|\theta\|^2$ is the L2 regularization term, scaled by a constant c, which is used to prevent overfitting during training.

4. IMPLEMENTATION

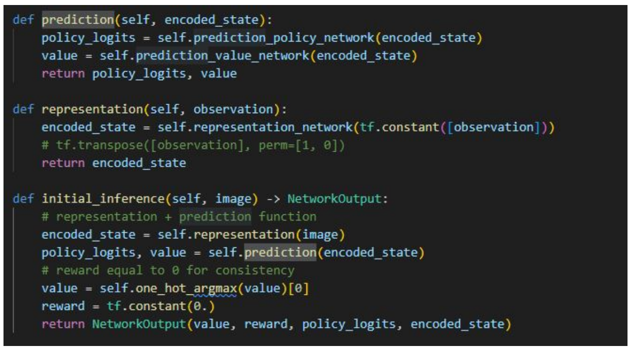
The MuZero algorithm was implemented based on the pseudocode provided in the original reference [10]. The pseudocode provided a structured and detailed outline of the algorithm's components and the interactions between them, but its interfaces (initial and recurrent) were hollow, and the policy, dynamics, and prediction functions were not defined. The implementation of the algorithm required the development of these components and their integration. To do this, the algorithm was implemented in Python, using the TensorFlow and Keras libraries for the neural network components. The implementation of these components was based on MuZero General [17], an open-source implementation of the MuZero algorithm.

The implementation uses as a base the process of generating self-play games, which are then used to train the model on the created data. To achieve this, the algorithm swaps between self-play and training, running N games of self-play, then training the model on the generated data for s_t, and repeating this process. The model is trained using the self-play data, which gets sampled each time and used as the training set. Self-play is executed on multiple threads, one for each game for a total of s_t which allows for the generation of a large amount of games in parallel. The threads are joined to train the model on a main thread. In this way, the storing of game data is facilitated, avoiding a conflict between the threads. Due to the algorithm intrensic parallelism for multiple games, it is advantageous to use a multi-threaded approach, thus speeding up the training process.

4.1 Interfaces

The initial and recurrent interfaces are the backbone of the MuZero algorithm, as they are used for both self-play and training. The initial interface is used at the start of each playout in the environment, to encode the observation of the environment into a hidden state. Specifically, as shown in figure 4.1, the initial interface uses the representation function to encode the image of the game state into a hidden state, after which the prediction function is used. The prediction function calculates the starting policy and value function using the encoded state. An ANN is used to evaluate each of the function outputs.

Similarly, the recurrent interface is used during the search to evaluate the value of future nodes of the search tree. The recurrent interface is used to compute the reward and the hidden state using the dynamics function, and in the same way as the initial interface, the prediction function is used to calculate the policy and value function. The dynamics function calculates its output using an ANN, in a similar way as the prediction and representation function, as shown in figure 4.2.

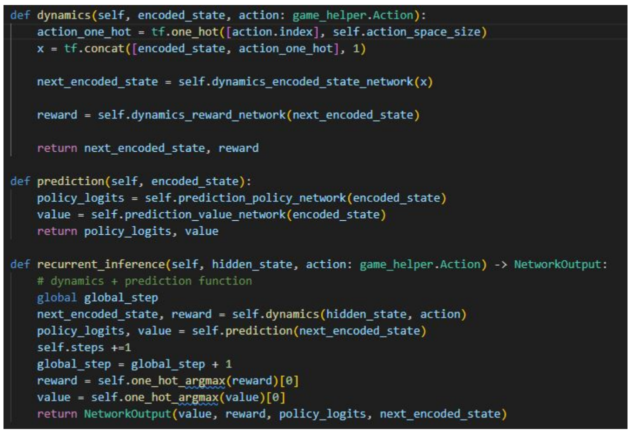


```python
def prediction(self, encoded_state):
    policy_logits = self.prediction_policy_network(encoded_state)
    value = self.prediction_value_network(encoded_state)
    return policy_logits, value

def representation(self, observation):
    encoded_state = self.representation_network(tf.constant([observation]))
    # tf.transpose([observation], perm=[1, 0])
    return encoded_state

def initial_inference(self, image) -> NetworkOutput:
    # representation + prediction function
    encoded_state = self.representation(image)
    policy_logits, value = self.prediction(encoded_state)
    # reward equal to 0 for consistency
    value = self.one_hot_argmax(value)[0]
    reward = tf.constant(0.)
    return NetworkOutput(value, reward, policy_logits, encoded_state)
```

Figure 4.1: The initial interface of the MuZero algorithm. Based on MuZero General [17].



```python
def dynamics(self, encoded_state, action: game_helper.Action):
    action_one_hot = tf.one_hot([action.index], self.action_space_size)
    x = tf.concat([encoded_state, action_one_hot], 1)

    next_encoded_state = self.dynamics_encoded_state_network(x)

    reward = self.dynamics_reward_network(next_encoded_state)

    return next_encoded_state, reward

def prediction(self, encoded_state):
    policy_logits = self.prediction_policy_network(encoded_state)
    value = self.prediction_value_network(encoded_state)
    return policy_logits, value

def recurrent_inference(self, hidden_state, action: game_helper.Action) -> NetworkOutput:
    # dynamics + prediction function
    global global_step
    next_encoded_state, reward = self.dynamics(hidden_state, action)
    policy_logits, value = self.prediction(next_encoded_state)
    self.steps +=1
    global_step = global_step + 1
    reward = self.one_hot_argmax(reward)[0]
    value = self.one_hot_argmax(value)[0]
    return NetworkOutput(value, reward, policy_logits, next_encoded_state)
```

Figure 4.2: The recurrent interface of the MuZero algorithm. Based on MuZero General [17].

4.2 Self-play

Self-play is the process of generating games by having the model play against itself. The model selects actions based on its learned policy network, and the search algorithm is used to explore the game tree and calculate possible future outcomes.

The first step is the initialization of a new game, a configuration object is used to allow for easy adaptation to different games. Immediately after, the game loop starts, where the current model is used to select an action. To facilitate action selection, the image of the current game state is extracted and passed to the initial interface, which is the representation function that encodes the image into an encoded hidden state, and the prediction function where a starting policy and value function is calculated using the encoded state [10].

Using the policy, the game tree is expanded by evaluating the child nodes of the current state and adding them to the tree. The visits of their root node are then updated using the initial evaluation and noise is added with the Dirichlet distribution. This process prepares the tree for the search algorithm.

The search algorithm is an MCTS algorithm, which is used to explore the game tree and calculate possible future outcomes. The algorithm is used to explore the tree, figuring out an optimal action path by selecting the most promising nodes and expanding them. The recurrent interface is used during the search to evaluate the value of future nodes.

Finally, to complete a loop, an action is selected based on the number of visits each child node has received, the choice is influenced by the temperature parameter, which is used to control the level of exploration. The temperature value fluctuates between 0 and 1 with 1 being always exploration and 0 always exploitation. The temperature parameter starts high and drops as the training continues, which allows the model to exploit the knowledge it has gained. The action chosen is executed in the game environment, after which the game data is saved and the loop repeats until the game ends or the max number of moves is reached. The game data from each thread is gathered and stored for the training of the model.

4.2.1 Search Algorithm

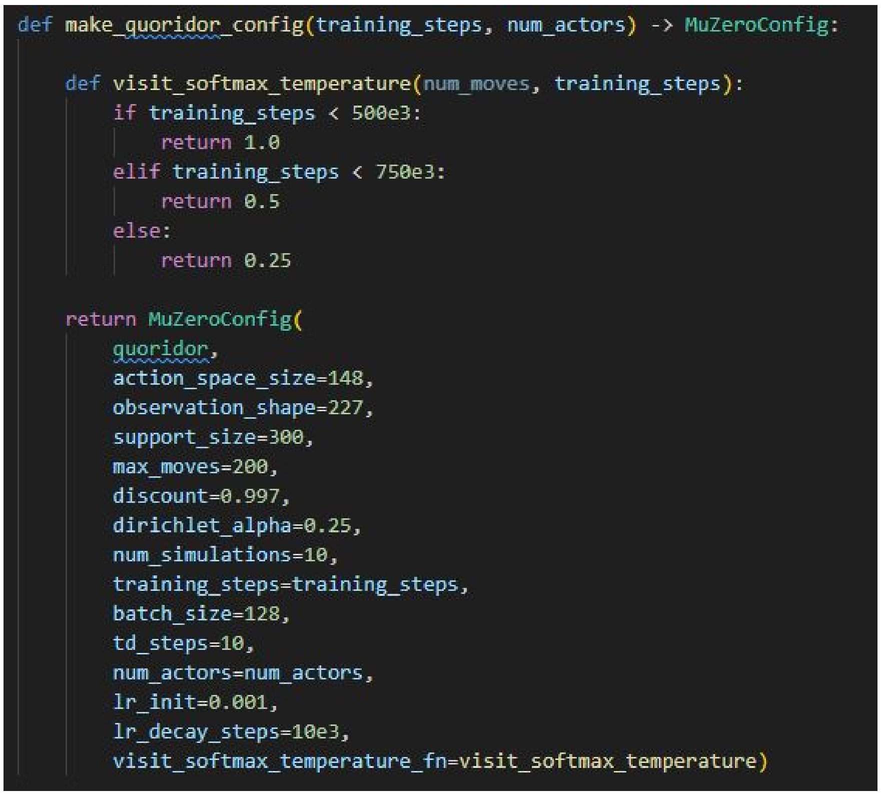
To explain in depth the search algorithm used in the pseudocode of the reference [10], the search starts by selecting the child node with the highest score, as determined by using the UCB formula. The selected node is then set as the current node and added to the search path. For the new node, the recurrent interface determines all the quantities needed for the search. More specifically, the reward and the hidden state are calculated using the dynamics function, and the policy and the value function are calculated using the prediction function. With these values, the search algorithm is able to expand the current node and continue the search by first backpropagating the value of the new node and then repeating all steps. This process is repeated for a set number of actions, which for all tests in this thesis is 10.

4.3 Training

The training of the network is done using batched samples of the data generated by self-play. The training process updates the weights of the neural networks to fit the set of games that are being used as the training set. This results in improved performance in future games, which are then used to train the network again. The training process is done in a main thread, where the self-play threads are joined to train the model on the generated data.

At the start of training, the current best model is initialized, as well as the optimizer, in this case Adam [6], using TensorFlow. After the initialization, the training loop starts, where the model is trained on the generated data. A batch is created by picking random games from those available and selecting a random position from each game. The game state, the steps taken after the position, and their expected value which is also the target value for the model, are put together to create a complete batch. The batch is then used to update the weights of the model.

The weight update is done by first calculating the loss [10] of the model using the batch, using the mean squared error (MSE) between the predicted value and the target value. The loss is averaged over the batch and the L2 regularizer is used along with a weight decay factor set to $\gamma = 10^{-4}$. To update the weights, the optimizer is used to scale the gradient according to the loss.



```python
def make_quoridor_config(training_steps, num_actors) -> MuZeroConfig:

    def visit_softmax_temperature(num_moves, training_steps):
        if training_steps < 500e3:
            return 1.0
        elif training_steps < 750e3:
            return 0.5
        else:
            return 0.25

    return MuZeroConfig(
        quoridor,
        action_space_size=148,
        observation_shape=227,
        support_size=300,
        max_moves=200,
        discount=0.997,
        dirichlet_alpha=0.25,
        num_simulations=10,
        training_steps=training_steps,
        batch_size=128,
        td_steps=10,
        num_actors=num_actors,
        lr_init=0.001,
        lr_decay_steps=10e3,
        visit_softmax_temperature_fn=visit_softmax_temperature)
```

Figure 4.3: Hyperparameters of the MuZero algorithm for the Quoridor game.

4.4 Network

The neural networks used to predict the policy, value, and reward quantities in the MuZero algorithm are implemented using TensorFlow. There are actually five individual networks used in the algorithm, one for the representation, two for the dynamics, and two for the prediction function, following the MuZero General structure [17]. The representation network is used to encode the observation of the environment into a hidden state, which is then used by the dynamics and prediction networks to predict relevant future quantities for planning.

These networks are used exclusively as parts of the interfaces, specifically the initial and recurrent interfaces. These interfaces are the backbone of the algorithm, as they are used for both self-play and training. To make the output creation easier the interfaces use one-hot vectors to represent the actions, and also the output of the networks are designed to match the input of the next network.

4.5 Environment

As described earlier, the environment used to test the algorithm is the game Quoridor. The game is a competitive two-player strategy board game. The game is played on a 9x9 board, and the objective is to move your pawn to the opposite side of the board.

Minor changes were made to the game to facilitate the implementation of its simulation. While a pawn can move in the four cardinal directions, in the original version there is a special case where its movement is blocked by another pawn with a wall behind it, it can move diagonally to the left or right. This rule allows for the pawn to avoid being boxed in by the opponent. Our version of the game has removed the ability to make such a move. This change causes a small portion of games to end prematurely, as a player lacks valid moves. These games are ignored in the training process, but they can also be classified as a tie, like in chess.

4.6 Evaluation

The evaluation of the algorithm is done by having it face a random agent, which selects a random action at each step. The model used is the latest model generated by the training process and it is evaluated after every 2000 training step with 10 games. The algorithm focuses on exploration in the early stages of training, picking actions based on the distribution of the visits, but as the training progresses, the algorithm starts to exploit the knowledge it has gained. This means that, in combination with the limited experience at the early stages of training, the algorithm will lose to the random agent. We can expect that as training progresses, the algorithm will start to win more games, as the number of training steps performed in the reference [10] is 1 million, significantly more then this evaluation.

The present evaluation was based on the scoring system of the environment, which is a score of 1 for a victory and 0 for a tie. The result is negative for a player if the reward gain is during the opponent's turn, meaning if the game end in victory in the opponent's turn the score is -1. The evaluation is done by calculating the average score of the games played, which is then used to determine the performance of the algorithm. The results are indecisive with the number of steps at this point of the training, as the algorithm has yet to understand the game, and its dynamics, and is still learning its strategies. We can expect that with more time to train, and better tuning of the hyperparameters, the algorithm will achieve better results.

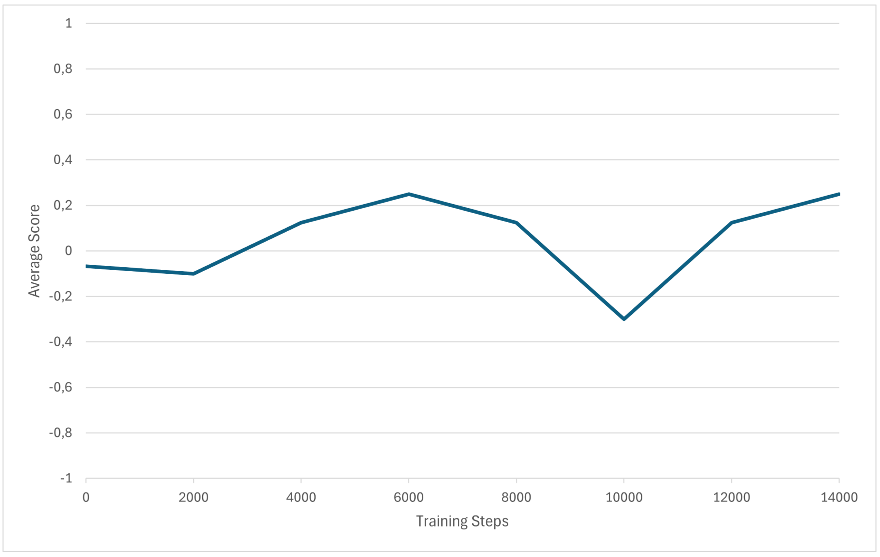


Figure 4.4: Win rate of MuZero's implementation
The win rate stays low, as it is at the very start of its training.

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The MuZero algorithm is a powerful deep reinforcement learning algorithm that combines an MCTS with deep neural networks to achieve exceptional performance in learning and planning. It has the versatility to be implemented and handle a wide range of environments, making it a general-purpose algorithm that can adapt to different challenges. The algorithm's ability to learn the dynamics of the environment without prior knowledge and to predict the policy, value, and reward quantities for planning makes it a robust and effective solution for complex tasks.

Preliminary results show that the algorithm has difficulty beating a random agent in the early stages of training, as it is still learning the game's dynamics and strategies. However, with more training and tuning of the hyperparameters, the algorithm is expected to achieve better results and outperform the random agent.

5.2 Future Work

Outside the scope of the original work by DeepMind, the algorithm implemented for this thesis requires further fine-tuning and optimization to achieve optimal performance. The training process can be improved by adjusting the hyperparameters. The training time could also be improved by using an accelerator like a GPU or TPU for the entire training process to speed it up. The implementation of the algorithm can be further optimized to handle larger environments and more complex games.

Their algorithm should also be tested on a wider range of environments to evaluate its performance and adaptability. Video game environments, such as those in the OpenAI Gym [3] or the newer version Gymnasium [13], could be used to test the algorithm's capabilities in more complex and visually rich environments. Also, more complex and life-like games, with multiple objectives and agents, could be used to test the algorithm's performance in more challenging environments.

ABBREVIATIONS - ACRONYMS

MDP	Markov Decision Processes
MTCS	Monte Carlo Tree Search
ML	Machine learning
DL	Deep learning
RL	Reinforcement learning
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
NLP	Natural Language Processing
MSE	Mean Squared Error

REFERENCES

[1] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, November 2018.

[2] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. A survey on modern trainable activation functions. *Neural Networks*, 138:14–32, June 2021.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv e-prints*, page arXiv:1606.01540, June 2016.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2016, pages 770–778, 2016.

[5] Weiming Hu, Xintong Li, Chen Li, Rui Li, Tao Jiang, Hongzan Sun, Xinyu Huang, Marcin Grzegorzek, and Xiaoyan Li. A state-of-the-art survey of artificial neural networks for whole-slide image analysis: From popular convolutional neural networks to potential visual transformers. *Computers in Biology and Medicine*, 161:107034, July 2023.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, December 2014.

[7] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 157–163. Morgan Kaufmann, San Francisco (CA), 1994.

[8] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[9] Iqbal H. Sarker. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2(6):420, August 2021.

[10] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[11] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *ArXiv*, abs/1912.10944, 2019.

[12] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[13] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium. `https://zenodo.org/record/8127025`, March 2023.

[14] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, page 3–42. Springer Berlin Heidelberg, 2012.

[15] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. On Monte Carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, December 2017.

[16] Xu Wang, Sen Wang, Xingxing Liang, Dawei Zhao, Jincai Huang, Xin Xu, Bin Dai, and Qiguang Miao. Deep reinforcement learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022:1–15, 2022.

[17] Aurèle Hainaut Werner Duvaud. Muzero general: Open reimplementation of muzero. `https://github.com/werner-duvaud/muzero-general`, 2019.

[18] Yu-chen Wu and Jun-wen Feng. Development and application of artificial neural network. *Wireless Personal Communications*, 102(2):1645–1656, December 2017.