universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Learning When to Plan

verfasst von | submitted by

Diego Fernando Monge Pimentel

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 645
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Data Science
Betreut von | Supervisor:	Assoz. Prof. Dipl.-Ing. Dr.techn. Sebastian Tschiatschek BSc

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Assoz. Prof. Dipl.-Ing. Dr.techn. Sebastian Tschiatschek, BSc, and Timo Klein, B.Sc. M.Sc., for their invaluable guidance and support throughout this work. Their insights were crucial to its development. I also wish to thank my family for their continuous encouragement during my academic journey, as well as my friends for their unwavering support.

Abstract

Reinforcement Learning (RL) algorithms have gained significant attention due to their impressive performance and wide applicability across various domains. A prominent algorithm focused on games, AlphaZero, combines neural networks with Monte Carlo Tree Search (MCTS) to accelerate learning. However, AlphaZero's fixed search budget (i.e., the number of MCTS search traces) can lead to inefficiencies by dedicating too much planning to predictable outcomes or insufficient planning to uncertain ones.

In this thesis, we propose modifications to the AlphaZero algorithm to dynamically adjust the search budget based on the estimated uncertainty in its components: the neural network and MCTS. To estimate the first, we extend AlphaZero using principles from distributional reinforcement learning, taking the variance in quantiles as uncertainty. MCTS uncertainty is estimated based on the depth and size of subtrees. These uncertainties are then leveraged by a Deep Q-Network (DQN) agent to adjust the search budget at each step.

Our proposed approach is evaluated on the CartPole and MinAtar environments using AlphaZero and DQN as baselines. The findings show that dynamically allocating the number of search traces based on uncertainty enhances algorithm efficiency, as evidenced by an improved reward-per-trace ratio. Furthermore, even though our modifications introduced additional computation costs, the overall run-time was reduced in certain environments while maintaining or exceeding baseline performance in terms of total reward.

Kurzfassung

Verstärkendes Lernen (Reinforcement Learning, RL) Algorithmen haben aufgrund ihrer beeindruckenden Leistung und breiten Anwendbarkeit in verschiedenen Bereichen große Aufmerksamkeit erlangt. Ein prominenter Algorithmus, der sich auf Spiele konzentriert, AlphaZero, kombiniert neuronale Netzwerke mit Monte Carlo Tree Search (MCTS), um das Lernen zu beschleunigen. Allerdings kann das feste Suchbudget von AlphaZero (d.h. die Anzahl der MCTS-Suchspuren) zu Ineffizienzen führen, indem zu viel Planung auf vorhersehbare Ergebnisse und zu wenig auf unsichere Ergebnisse verwendet wird.

In dieser Arbeit schlagen wir Änderungen am AlphaZero-Algorithmus vor, um das Suchbudget dynamisch basierend auf der geschätzten Unsicherheit in seinen Komponenten – dem neuronalen Netzwerk und MCTS – anzupassen. Um die Unsicherheit des neuronalen Netzwerks zu schätzen, erweitern wir AlphaZero unter Verwendung von Prinzipien aus dem distributionellen Verstärkungslernen, wobei die Varianz in den Quantilen als Unsicherheit betrachtet wird. Die Unsicherheit von MCTS wird basierend auf der Tiefe und Größe von Teilbäumen geschätzt. Diese Unsicherheiten werden dann von einem Deep Q-Network (DQN)-Agenten genutzt, um das Suchbudget bei jedem Schritt anzupassen.

Unser vorgeschlagener Ansatz wird an den CartPole- und MinAtar-Umgebungen unter Verwendung von AlphaZero und DQN als Baselines evaluiert. Die Ergebnisse zeigen, dass die dynamische Zuweisung der Anzahl von Suchspuren basierend auf Unsicherheit die Effizienz des Algorithmus verbessert, was durch ein verbessertes Verhältnis von Belohnung pro Suchspur belegt wird. Darüber hinaus, obwohl unsere Änderungen zusätzliche Rechenkosten eingeführt haben, wurde die Gesamt-Laufzeit in bestimmten Umgebungen reduziert, während die Baseline-Leistung in Bezug auf die Gesamtbelohnung beibehalten oder sogar übertroffen wurde.

Contents

Acknowledgements i

Abstract iii

Kurzfassung v

List of Tables ix

List of Figures xi

1. Introduction 1

2. Background 5
2.1. Reinforcement Learning (RL) . . . 5
2.1.1. Distributional RL (DRL) . . . 7
2.1.2. Deep Q-network (DQN) algorithm . . . 8
2.1.3. Double DQN (DDQN) . . . 8
2.1.4. QR-DQN . . . 9
2.2. Uncertainty . . . 10
2.3. Monte Carlo Tree Search (MCTS) . . . 13
2.4. AlphaZero . . . 15

3. Related work 19
3.1. Alphazero and MuZero . . . 19
3.2. Neural Network uncertainty estimation . . . 20
3.3. MCTS uncertainty estimation . . . 21

4. Methodology 25
4.1. Uncertainty estimation . . . 25
4.1.1. MCTS uncertainty . . . 28
4.2. Dynamic selection of number of traces . . . 29

5. Environments 33
5.1. CartPole . . . 33
5.2. MinAtar . . . 33
5.2.1. Games . . . 34

6. Experiments 35
6.1. Setups . . . 35

6.2. Results 36
6.2.1. Evaluation 36

7. Discussion **43**
7.1. Evaluation of results 43
7.2. Limitations 44

8. Conclusion **45**
8.1. Future works 45

Bibliography **47**

A. Appendix **51**
A.1. DQN results 51
A.2. Hyperparameters 52
A.3. Environments details 52

List of Tables

A.1. Hyperparameters description. 53
A.2. Experiments hyperparams. If not specified, they apply to all setups. "Q/U" refers to QR_AZ and UA_AZ. 54

List of Figures

2.1. Reinforcement learning (RL) basic cycle. The agent interacts with the environment $\mathcal{M}$ at time t by performing action a_t, chosen by the policy $\pi(a|s)$. The environment changes to state s_{t+1} and emits it along with the reward r_{t+1} to the agent. From [35] 5

2.2. Illustration of the four phases of MCTS. In the **Selection** phase, guided by the tree policy π, the tree is navigated until a leaf node is reached. The **Expansion** phase involves choosing an action and adding the resulting state to the tree. In the **Simulation** phase, the value of the newly added node is estimated through a simulation guided by the simulation policy $\pi_{simulation}$. Finally, the **Backup** phase propagates the estimated value upwards, updating all visited nodes along the trajectory. Illustration based on Browne et al. [6]. 14

2.3. MCTS phases for AlphaZero. The introduced **Evaluation** phase replaces the original **Simulation** phase. It directly evaluates the newly added state during **Expansion** phase using the DNN f_θ instead of running simulations. The predicted value v is propagated upwards on the **Backup** phase as before. 16

2.4. **a. Self-play:** an episode of the game is run from empty board s_1 until the end of game s_T. For each state s_t, an MCTS tree is built using the latest DNN f_θ. The tree outputs a search probability π_t, which is used to choose the action $a_t \sim \pi_t$. On the final state, s_T, the game outcome Z is computed. **b. (Deep) Neural network training:** for each state used on the self-play episode s_t, the neural network is evaluated with it as input. The DNN outputs a probability distribution over moves $\mathbf{p}_t$ and outcome v_t. The DNN parameters θ are updated by using the loss defined on equation 2.21. The updated DNN f_θ is used on the next self-play episode. Figure taken from Silver et al. [32] 17

4.1. Illustration of the modifications on the DNN θ output heads. Here, J denotes $(K-1)*N$. For an input state s, action space $\mathcal{A}$ of $K = |\mathcal{A}|$ actions and N quantiles, the original DNN (left) has a value head outputting a single value v and a policy head predicting action probabilities $\mathbf{p}_{a_i}$ for each action a_i. In the modified DNN (right), the value head output $\hat{Z}^v$ contains the quantile values z_i^v of the return (v) distribution, while the policy head output $\hat{Z}^{\mathbf{P}}$ contains $K * N$ quantile values $z_i^{\mathbf{P}}$, representing the probability distributions of actions probabilities $\mathbf{p}_a$ for all actions a concatenated together. For AlphaZero-related calculations, $\hat{Z}^v$ is aggregated via its mean, and $\hat{Z}^{\mathbf{P}}$ by applying softmax to the mean quantile values across actions. 27

4.2. Dynamic trace allocation on MCTS search. Key changes are highlighted in blue. At step k with state s_k, the search (right) begins with a warm-up of w traces. A context $\tilde{s}$ is derived from uncertainty and statistics after the warm-up. The DQN agent uses this context to allocate additional traces m_k. The search proceeds until $w + m_k$ traces are reached, yielding the policy π_k, which is used to advance to the next state s_{k+1}. Additionally, a final context $\tilde{s}'$ and a reward $\tilde{r}$ for the allocation efficiency are generated. The experience tuple $(\tilde{s}, m_k, \tilde{s}', \tilde{r})$ is stored in buffer B for DQN agent's DNN training. 31

6.1. CartPole environment results. Our approach, UA_AZ, achieves comparable performance to the baselines AZ and DQN while utilizing fewer traces, as shown in the "Cumulative traces" and "Step traces" subplots. Due to the simplicity of the environment, all agents converge to similar maximum rewards, as shown in "Normalized Reward." Overall, UA_AZ's efficient use of traces by UA_AZ is reflected in the "Reward-per-Trace" subplot. . 37

6.2. Asterix environment results. The Step Rewards subplot highlights the faster convergence of AlphaZero-based agents compared to DQN. The Normalized Reward subplot demonstrates that these agents achieve at least four times the reward of DQN, even when the latter runs five times longer. While UA_AZ slightly underperforms the AZ baseline regarding Step Reward, the Reward-per-Trace subplot reveals its superior efficiency, confirming the effectiveness of dynamic trace allocation. Overall, the Cumulative Traces subplot highlights significant trace savings achieved by UA_AZ. 37

6.3. Breakout environment results. The Step Rewards subplot demonstrates the faster convergence of AlphaZero-based agents compared to DQN. The Normalized Reward subplot shows these agents achieving at least 2.5 times the reward of DQN, even when the latter runs five times longer. While QR_AZ slightly outperforms UA_AZ in step rewards, UA_AZ exhibits greater trace efficiency, as highlighted in the Reward-per-Trace subplot. The Cumulative Traces subplot confirms significant trace savings overall. . 38

6.4. Seaquest environment results. This environment is the most challenging due to its larger state space and sparse rewards. As a result, trace usage remained largely unchanged, reflecting this complexity. The Step Rewards subplot highlights the significantly faster convergence of AlphaZero-based agents compared to DQN. Additionally, the Normalized Reward subplot shows these agents achieving at least 2.5 times the reward of DQN, even when the latter runs five times longer. Overall, the Cumulative Traces subplot indicates small trace savings. 38
6.5. SpaceInvaders environment results. The Step Rewards subplot highlights the faster convergence of AlphaZero-based agents compared to DQN. The Normalized Reward subplot shows that all agents achieve similar rewards to DQN, even when it runs five times longer. The Reward-per-Trace subplot demonstrates the efficient trace utilization of UA_AZ. Overall, the Cumulative Traces subplot indicates significant trace savings. 39
6.6. Comparison of performance across different numbers of quantiles in the Breakout environment. Using 20 quantiles yields competitive step rewards while reducing the cumulative number of traces. All setups were run with 5 seeds and identical hyperparameters. 40
6.7. Step rewards corresponding to two UA_AZ agents over Breakout environment, using update main target network frequency 100 and 1000 steps. 41
6.8. Average total run-time for Breakout and Space Invaders over 1,000,000 steps and five seeds. 41

A.1. Full runs of DQN agent. Presented first 2,000,000 steps on CartPole environment, and 5,000,000 steps on MinAtar ones 51

1. Introduction

Reinforcement learning (RL) has shown great performance in a wide range of fields [21]. Specifically, on games, which usually require making decisions over enormous options (e.g., GO, Chess, etc.), various approaches were developed to master them efficiently. AlphaZero [34] is widely recognized as one of the most popular and exceptionally high-performing approaches by outperforming human learning entirely through self-play.

AlphaZero is the result of iteratively released versions of game-playing algorithms aimed at reducing the reliance on human inputs. It enables learning games with minimum prior information, thereby avoiding the biases introduced by earlier approaches that relied heavily on human-generated data.

In an earlier release within its family, AlphaGo [32], learning is directly derived from human expert games, requiring a large number of games and resulting in inherent biases. Its successor, AlphaGo Zero [33], removes the need for human knowledge by relying solely on the given game rules optimized for Go. Representing a more generalized evolution, AlphaZero [34] expanded upon AlphaGo Zero to also master Chess and Shogi, removing the need for game-specific optimizations.

AlphaZero relies on the joint work of two main components: a deep Neural Network (DNN) for quick estimation of game state outcome and policy (further moves probabilities) and the Monte Carlo Tree Search (MCTS) algorithm for planning.

At each step of the game, an MCTS tree rooted in the current game state is built by iteratively expanding nodes representing future valid states within a fixed budget (i.e., number of search traces). This search is performed within an isolated instance of the game, ensuring that the real game state remains unaffected. Each added node is evaluated by the DNN instead of running random simulations from it, as with vanilla MCTS. Thus, DNN guides the MCTS algorithm by providing an estimate of the policy and outcome. The resulting tree captures the potential trajectories or traces starting from the current (real) game state. Based on the collected search data, the most promising action is selected for execution in the real game. Both components continuously improve as immediate rewards from MCTS steps are collected and used to train the DNN, and the updated DNN weights then guide subsequent MCTS searches.

The interaction of the two main components within AlphaZero is reminiscent of a psychological concept: the dual process theory of human cognition [10]. It defines two cognitive systems that interact in human thinking: intuitive and planning systems. The first operates fast with hardly any reasoning, whereas the latter needs to reason carefully.

AlphaZero's DNN works similarly to the intuitive system by directly predicting estimates of policy and outcomes, eliminating the need for explicit simulations. On the other hand,

the MCTS is analogous to the planning system since, for each step, states would be evaluated by explicitly reasoning over future outcomes.

A characteristic of human thinking is that knowledge reduces planning. In contrast, the original AlphaZero doesn't use its acquired knowledge within the DNN and MCTS to dynamically reduce MCTS's load. Therefore, the planning process remains expensive.

In AlphaZero, planning plays a central role in both the training phase, where the agent learns through self-play, and the inference phase, where it makes decisions in real-time games. The expensive and recurring nature of planning, intensively used in both phases, combined with the lack of a smart budget allocation strategy for building the MCTS trees, presents an opportunity for a more directed use of computation during planning. This would result in a more efficient use of resources due to a considerable reduction of planning in both the training and inference stages.

Quantifying uncertainty when working with DNNs and MCTS provides an avenue for a more efficient allocation of the search budget (i.e., the maximum number of MCTS tree traces) by dynamically adjusting it based on the estimated uncertainty. This approach promotes informed decision-making by reducing unnecessary investigation in low-uncertainty scenarios and encouraging deeper investigation in uncertain ones. Consequently, it enhances algorithm speed and efficiency. To achieve these potential improvements, the present work addresses the following research questions:

RQ1: How can we account for online, multi-step planning uncertainty?

Various studies have been developed to estimate and use the uncertainty of AlphaZero components individually. For DNNs, there are multiple strategies for estimating its uncertainty –mainly the epistemic component– applied in different RL algorithms [28] [8]. On the other hand, uncertainty estimation in MCTS trees is a less explored topic with few published works [27] [18].

While the individual estimation and use of the uncertainty in DNNs and MCTS trees have demonstrated promising outcomes in different algorithms, their combined impact remains unexplored. Our work aims to address this gap by simultaneously estimating and using the uncertainties from both DNN (epistemic) and MCTS in AlphaZero.

As in Clements et al. [8], the estimation of a DNNs' uncertainty is done by using the Distributional RL framework [9], specifically by framing the estimation of return distribution quantiles as a Bayesian inference problem. Epistemic uncertainty is estimated based on the variance of quantiles.

On the other hand, following the strategy introduced by Moerland et al. [27], uncertainty for the MCTS will be guided by the states' number of sub-trees. This number, weighted by the state's visitation count, is stored and updated in the tree.

RQ2: How to use uncertainty estimates to dynamically adjust the number of traces?

To the best of our knowledge, none of the listed research takes into account the DNNs and tree uncertainty at the same time. The measured uncertainties can give insights into the performance of the tree building in real time and potentially provide a good criterion for making decisions regarding budget allocation.

At each game step search, the maximum number of traces for the MCTS tree is dynamically determined by a DQN agent. During a warm-up phase, a fixed set of traces are generated. After this phase, the agent uses performance statistics and uncertainty information to decide the number of extra traces to generate, thereby modifying the total trace count.

The proposed approach viability is verified in CartPole [5], a basic environment. As a more challenging benchmark, we utilize MinAtar [38], a miniaturized version of five Atari 2600 games.

The metrics analyzed include immediate and cumulative trace usage and runtime, measured in wall-clock time per agent step. These are evaluated in relation to the obtained rewards, with the expected outcome being a reduction in these metrics while quickly converging to similar favorable rewards. The baseline models utilized are the original AlphaZero and Deep Q-Networks [24]. Experiments are conducted to evaluate the impact of individual estimated uncertainties by isolating each uncertainty factor.

This thesis is structured as follows: Chapter 2 provides the background of key concepts, including a brief introduction to RL, uncertainty definition and quantification, and a review of MCTS and the AlphaZero algorithm. Next, a review of related literature to contextualize our approach is presented in Chapter 3. In the following, Chapter 4, details our methodology, covering the estimation of the uncertainty in the DNN and MCTS, along with defining the dynamic MCTS' trace budget allocation. Moving into the evaluation of our approach, the evaluation environments are detailed in Chapter 5 followed by the experiment setups, their outcomes, and analyses in Chapter 6, and the overall discussion and limitations in Chapter 7. Finally, Chapter 8 summarizes the thesis and suggests future work directions.

2. Background

This chapter introduces the foundational concepts underpinning the proposed method. First, a brief introduction to RL and the concept of distributional RL will be provided, followed by RL algorithms, including Deep Q-network (DQN), Double DQN, and Quantile Regression DQN. Then, a brief introduction to uncertainty and methods for estimating it in DNNs is presented. Next, the MCTS planning algorithm is detailed. Finally, we introduce the AlphaZero algorithm.

2.1. Reinforcement Learning (RL)

Reinforcement Learning aims to learn the optimal behavior of an agent that maximizes the rewards from the interaction with an environment. This interaction involves the agent performing actions — selected by a policy π — that change the environment's state s_t. The environment then notifies the agent about the new state s_{t+1} and a reward for the transition r_{t+1}. The ultimate objective of RL is to learn the optimal policy that maximizes the discounted cumulative reward denoted as return $R = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ [11] (discount factor γ is described next). If the environment is episodic (i.e., the environment is reset after episodes of some length T), the number of steps k is limited to T. Figure 2.1 illustrates the process.

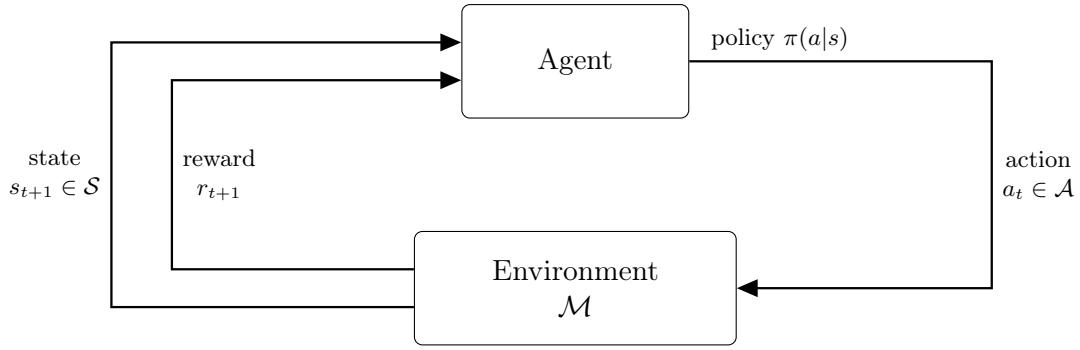

Figure 2.1.: Reinforcement learning (RL) basic cycle. The agent interacts with the environment $\mathcal{M}$ at time t by performing action a_t, chosen by the policy $\pi(a|s)$. The environment changes to state s_{t+1} and emits it along with the reward r_{t+1} to the agent. From [35]

Typically, the environment is formulated as a Markov Decision Process (MDP) [35]. Formally, an MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ denote the state and

action spaces, respectively. The transition function $\mathcal{T}(s'|s,a) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ defines the dynamics of state transitions, while the reward function $\mathcal{R}(s,a,s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ assigns rewards to given transition. The discount factor $\gamma \in [0,1]$ weights future rewards.

MDPs have two key properties: (i) the future outcomes solely depend on the current state, eliminating the need for the agent to access historical information and (ii) the environment is fully observable, enabling the agent to access all relevant information about the environment's state. These properties simplify the agent's decision-making process by providing enough information.

Formally, the policy π is defined as:

$$\pi(a|s) : \mathcal{S} \times \mathcal{A} \to [0,1]$$

It maps the probability of an action a based on the current state s, serving as the primary decision-making criteria for the agent. As mentioned earlier, the objective of RL is to find an optimal policy π^* that maximizes the return R.

$$\pi^* = \max_{\pi} \mathbb{E}\big[R \,\big|\, \pi\big]$$

Two important concepts that aim to estimate the performance of interacting from a specific state are the V-value and Q-value functions.

The V-value function $V^\pi(s) : \mathcal{S} \to \mathbb{R}$ estimates the expected return by starting from state s and following the policy π. It is formally defined as [11]:

$$V^\pi(s) = \mathbb{E}\left[R \,\middle|\, s_t = s, \pi\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, \pi\right] \tag{2.1}$$

The optimal expected return based on the V-value function is defined as [11]:

$$V^*(s) = \max_{\pi} V^\pi(s) \qquad \forall s \in \mathcal{S} \tag{2.2}$$

The optimal policy π^* cannot be directly deduced from the optimal state-value function V^* because the agent requires access to the transition function $\mathcal{T}$ to directly select actions that maximize the value function [1].

The Q-Value function, also known as state-action-value, $Q^\pi(s,a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ differs from the V-value function by specifying an initial action a performed on state s and following the policy π thereafter. It is defined as [11]:

$$Q^\pi(s,a) = \mathbb{E}\left[R \,\middle|\, s_t = s, a_t = a, \pi\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a, \pi\right] \tag{2.3}$$

Similarly, the optimal expected return based on the Q-value function is defined as [11]:

$$Q^*(s,a) = \max_{\pi} Q^\pi(s,a) \tag{2.4}$$

In contrast to the V-value function, the optimal formulation of Q-value one, $Q^*(s,a)$, can directly point out to the optimal policy $\pi^*(s)$ by [11]:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q^*(s,a) \tag{2.5}$$

Additionally, the Advantage function is commonly used to characterize the effectiveness of an action a relative to the expected return achieved by adhering directly to the policy π. It is defined by integrating the aforementioned functions as [11]:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.6}$$

While interacting with the environment, the agent faces the so-called exploitation-exploration dilemma. Exploitation refers to the strategy of utilizing acquired knowledge to consistently select actions with the highest known rewards. Despite this strategy appearing beneficial for leveraging gained experience, it carries the drawback of potentially overlooking initially sub-optimal trajectories that could yield better long-term outcomes. On the other hand, exploration encourages the investigation of suboptimal trajectories with the hope that they may eventually lead to favorable rewards.

Hence, it is crucial for the policy π to strike an appropriate exploitation-exploration tradeoff. The right balance allows for efficient decision-making by leveraging acquired knowledge while also affording the agent some freedom to discover the environment.

2.1.1. Distributional RL (DRL)

DRL aims to learn the complete distribution over the return R (or Z^π as on [3]) rather than its expected value, diverging from conventional RL. This approach captures more information about R, enabling risk sensitivity exploration and uncertainty awareness strategies and even enhancing sample efficiency by providing richer return signals.

The distribution is denoted as the *value distribution* and referenced as the random variable Z. The environment remains formulated as an MDP, with the reward function $\mathcal{R}(s,a,s')$ treated explicitly as a random variable denoted $\mathsf{R}(s,a)$.

By using the distributional Bellman operator T^π under policy π [3], the distribution is characterized considering the joint interaction of three other random variables: immediate reward, next state s', and its random return $Z(s',a')$ as follows:

$$\begin{aligned} T^\pi Z(s,a) &:\overset{D}{=} \mathsf{R}(s,a) + \gamma Z(s',a') \\ s' &\sim \mathcal{T}(\cdot|s,a), a' \sim \pi(\cdot|s') \end{aligned} \tag{2.7}$$

where $A :\overset{D}{=} B$ denotes that random variable A is distributed by the same probability law as B.

Various representations of Z have been introduced. For instance, Bellemare et al. [3] introduced the c51 algorithm, which parameterizes Z as a categorical distribution over a fixed set of 51 atoms (locations uniformly spaced over the predetermined support).

Dabney et al. [9], on the other hand, proposed the Quantile Regression DQN (QR-DQN) algorithm, which parameterizes Z as a modification of c51 by fixing the probabilities and learning the locations (i.e., quantile values).

In this thesis, we adopt concepts from QR-DQN as suggested by Clements et al. [8]. Section 2.1.4 covers it in greater detail.

2.1.2. Deep Q-network (DQN) algorithm

Mnih et al. [25] introduced this off-policy algorithm aimed to approximate the optimal action-value function in RL using a particular DNN named Deep Q-Network, denoted as θ_Q^ω, where ω is the set of network weights.

The DQN takes the environment's state s as input and predicts Q-values for all possible actions a_i, $Q^\omega(s, a_i)$. Action selection is based on choosing the action associated with the highest Q-value.

To address instability in learning, Mnih et al. [25] proposed two key components: experience replay and target networks. Experience replay mitigates the instability caused by the correlation between consecutive samples, while target networks address the issue of frequently changing target values.

The experience replay (or replay buffer) B stores interactions as experiences $e_t = (s_t, a_t, r_t, s_{t+1}, d_t)$[1] consisting of state, action, reward, next state, and done flag. Proper selection buffer and batch size are necessary to balance the longevity of stored experiences.

The target network θ_Q^ϕ is a DQN that mirrors the architecture of the main DQN θ_Q^ω initialized with the same network weights, which are then synchronized periodically after a fixed number of steps. Choosing an appropriate synchronization frequency is critical to avoid instability from rapid updates and slow training from outdated target networks.

The algorithm is based on Temporal Difference (TD) learning, with the target Q-value defined as:

$$Q_{tar_{dqn}}(s, a) = r + \gamma \max_{a'} Q^\phi(s', a') \tag{2.8}$$

where γ is the discount factor that tradeoffs the agent horizon. The loss function to minimize is:

$$\mathcal{L}_{DQN} = \mathbb{E}_{(s,a,r,s',d)\sim U(B)}\left[(Q_{tar_{dqn}}(s, a) - Q^\omega(s, a))^2\right] \tag{2.9}$$

2.1.3. Double DQN (DDQN)

Van Hasselt et al. [37] introduced Double DQN, a modification of DQN to address the bias and improve its performance. They state that using inaccurate values to estimate the maximum expected value leads to positively biased estimations. That bias is particularly problematic in the DQN framework, where the estimation of Q-values is performed by a continuously learning network, making estimation errors likely.

[1] For readability, the subscripts t and $t+1$ will be replaced with no subscript and a prime symbol ($'$), respectively.

Van Hasselt et al. [37] decomposed the equation 2.8 into evaluation and action selection as:

$$Q_{tar_{dqn}}(s,a) = r + \gamma Q^{\phi}(s', \max_{a'} Q^{\phi}(s', a')) \tag{2.10}$$

Here, the same network (θ_Q^{ϕ}) handles both evaluation and selection, introducing biased estimations. To mitigate this, Van Hasselt et al. [37] proposed using the target network θ_Q^{ϕ} to evaluate the best action selected by the main network θ_Q^{ω}, modifying the target equation 2.10 as:

$$Q_{tar_{ddqn}}(s,a) = r + \gamma Q^{\phi}(s', \max_{a'} Q^{\omega}(s', a')) \tag{2.11}$$

The learning process remains unchanged, except for swapping the target estimate $Q_{tar_{dqn}}$ with $Q_{tar_{ddqn}}$ in the loss equation 2.9.

2.1.4. QR-DQN

Dabney et al. [9] introduced QR-DQN (Quantile Regression DQN), a DRL algorithm that models the distribution of return Z^{π} (i.e., its probability law Z) using quantile regression, providing more accurate estimates and greater flexibility than previous methods [3].

Given a fixed number of quantiles N and $i = 1..N$, parametrization of the distribution Z involves fixing the probabilities $q_i = 1/N$ and varying their corresponding locations (or quantile values) z_i, expecting $z_1 <= .. <= z_N$. The cumulative probabilities along the Z's cumulative distribution function (CDF) F_Z are denoted as $\tau_i, .., \tau_N$, where $\tau_i = i/N$ and their corresponding *quantile midpoints* $\hat{\tau}_i = \frac{\tau_{i-1}+\tau_i}{2}$. The locations are then defined as $z_i = F_Z^{-1}(\hat{\tau}_i)$.

Dabney et al. [9] utilize this representation to estimate the locations (quantiles values) of Z. The learning process aims to minimize the Wasserstein distance [3], which quantifies the difference between the cumulative distribution functions of the Bellman updated distribution $T^{\pi}Z$ and the current distribution Z, through quantile regression.

Quantile regression employs a convex function, the QR loss, to penalize overestimation errors with weight τ and underestimation with $1 - \tau$, encouraging the expected sorting behavior $z_1 <= .. <= z_N$. The QR loss corresponding to predicted locations $\hat{Z}$ is defined as:

$$\mathcal{L}_{QR}(\hat{Z}) = \sum_{i=1}^{N} \mathbb{E}_{Z' \sim Z}\big[\rho_{\tau_i}(Z' - \hat{Z})\big] \tag{2.12}$$

where:

$$\rho_{\tau}(u) = |\tau - \delta_{\{u<0\}}|u \tag{2.13}$$

In this context, δ_a denotes a Dirac at $a \in \mathbb{R}$.

To address the non-smoothness of the QR loss at zero, Dabney et al. [9] proposed the quantile Huber Loss, which confines the square loss to an interval $\big[-\kappa, \kappa\big]$ around zero and works as QR loss out of it. The Huber loss is defined as [13]:

$$\mathcal{L}_\kappa(u) = \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \leq \kappa \\ \kappa(|u| - \frac{1}{2}\kappa), & \text{otherwise} \end{cases} \tag{2.14}$$

and introducing Huber Loss into Equation 2.13, the quantile Huber loss is defined as:

$$\rho_\tau^\kappa(u) = |\tau - \delta_{\{u<0\}}|\mathcal{L}_\kappa(u) \tag{2.15}$$

finally, the loss from Equation 2.12 is updated by swapping function ρ_τ for ρ_τ^κ as:

$$\mathcal{L}_{QR}(\hat{Z}) = \sum_{i=1}^{N} \mathbb{E}_{Z' \sim Z}\left[\rho_{\tau_i}^\kappa(Z' - \hat{Z})\right] \tag{2.16}$$

Additionally, instead of sampling Z' values from the return distribution Z, the QR loss is extended by using deterministically estimated targets. The updated general version is defined as:

$$\mathcal{L}_{QR}(\hat{Z}, Z') = \sum_{i=1}^{N} \mathbb{E}_j\left[\rho_{\tau_i}^\kappa(Z'_j - \hat{Z})\right] \tag{2.17}$$

Here, Z'_j represents the target value for the j-th quantile, so the expectation is taken over all target quantiles rather than random samples from Z. In QR-DQN, the target value Z' is computed deterministically using the Bellman equation (see Equation 2.7).

Dabney et al. [9] proposed the QR-DQN algorithm as a variant of DQN architecture on Mnih et al. [25] work, introducing a return distribution representation and the presented custom quantile regression loss, modified by the quantile Huber loss, with $\kappa = 1$.

2.2. Uncertainty

The uncertainty in deep learning networks (DNNs) can arise from two sources [14]:

- *Epistemic uncertainty* stems from a lack of knowledge. It can be reduced through the process of learning about the environment (i.e., input source).
- *Aleatoric uncertainty* origins from inherent randomness in the environment. Unlike epistemic uncertainty, it is not reducible.

Gawlikowski et al. [12] state that uncertainty in DNNs can be attributed to five primary factors:

Real world environments' inherent variability and high sensibility to different external factors depending on their nature, such as temperature, illumination, and others, can significantly affect the performance of NNs, given their sensitivity to data distribution shifts.

The process of data collection can introduce another significant factor, as the collected data may be biased by the measurement methods utilized. Inappropriate measurement techniques that fail to capture sufficient and reliable information about the environment

can lead to false labeling, added noise, and inaccurate representations of the environment. As NNs rely on the provided data for learning, the *error and noise in measurement systems* introduces uncertainty into their predictions.

Another crucial factor lies in the *errors within the NN model structure*. Design decisions directly impact performance, with the complexity of the NN potentially leading to undesired behaviors such as overfitting or underfitting the training data. Hence, it is crucial to carefully define a suitable architecture that aligns the characteristics of the environment and data being used.

Errors in the NN training procedure constitute another significant factor. The training process offers customization through various parameters, such as batch size, learning rate, etc. Additionally, stochastic decisions such as batch generation, weight initialization, and data augmentation are made. This flexibility leads to different sets of trained NN parameters, making consistency across different training setups unlikely.

Another factor that can impact prediction accuracy is imbalanced data, where the training process may overfit and yield poor results for the class with fewer samples. The use of different strategies, such as data augmentation, can lead to varied outcomes.

Errors stemming from unknown data present another significant factor. Particularly in classification tasks, NNs often struggle with knowledge transfer, especially when dealing with unknown data. When an unknown class is encountered, the NN is unable to accurately identify it as such. Instead, it may assign one of the trained data labels, leading to the generating of false labels that are difficult to trace.

In the context of this work, the second factor is dismissed due to the use of artificial environments, offering bounded and reliable information. Furthermore, the last factor is irrelevant as our (D)NN task is not classification.

Gawlikowski et al. [12] describe four general methods of uncertainty quantification: by using single deterministic networks, Bayesian methods, ensemble methods, and test-time data augmentation.

Single deterministic networks - This group of approaches for quantifying uncertainty aims to compute the prediction uncertainty based on a single forward pass through a (deterministic) NN. These approaches can be categorized as internal and external uncertainty quantification methods.

Internal methods modify the NN itself to output the distribution over the predictions instead of a single prediction. Implementation complexity varies, but generally, only the loss function and network final output require modifications. Internal methods have the advantage of not requiring external components, but this also represents a drawback since already trained networks need to be retrained with the introduced modifications of the architecture. Additionally, these methods are unable to output the prediction and uncertainty estimations separately.

External methods do not affect models' predictions and quantify uncertainty separately. Typically, a second NN is used to predict the uncertainty of the original network. Implementation complexity depends on the specific approach chosen but is generally low. Since these methods are external, they can be applied to already-trained networks.

Single deterministic methods are generally computationally efficient. Internal methods do not incur additional training costs, while external may, depending on the complexity of the external component. Nevertheless, single deterministic methods are generally more efficient than the other approaches to be discussed. However, these methods are sensitive to the network architecture, training procedure, and data.

Bayesian methods - These methods are characterized by utilizing multiple instances of the model, with parameters explicitly modeled as random variables. Due to the varied parametrization, stochastic predictions are expected. However, a drawback of these methods is their expensiveness since they require handling multiple model instances.

Neural network parameters are modeled as probability distributions rather than fixed values to capture the uncertainty in parameter values. To learn this distribution, Bayes' theorem is applied, which relates the posterior distribution of the parameters given the data sample $p(\theta|x, y)$ to the prior distribution $p(\theta)$. The computation of the posterior distribution is complex, therefore approximate inference techniques are used.

Gawlikowski et al. [12] group the methods into three categories based on how the posterior distribution over the parameters is inferred to approximate Bayesian inference;

- Variational inference: the posterior distribution is approximated by optimizing a set of track table distributions $q(\theta)$. The goal is the minimization of the difference, measured by Kullback-Leibler(KL) divergence [17], between $q(\theta)$ and the target distribution.

- Sampling approach: also known as Monte Carlo methods, these approaches use a set of samples (drawn from the posterior distribution) and offer the advantage that the representation is not restricted by the type of distribution. Therefore, distributions can be obtained non-parametrically.

- Laplace Approximations: these methods directly approximate the logarithm of the posterior distribution with a multivariate normal distribution.

Ensemble methods - These methods aim to improve generalization by combining predictions from multiple models, known as ensemble members. These members have independent architecture but use the same data. Evaluating the combined ensemble's predictions can provide an estimate of the model's uncertainty. To achieve effective results, it is important to maximize the diversity among the ensemble members. Common techniques to achieve it include random initialization and data shuffling, bagging and boosting of data, and data augmentation.

Although ensemble methods can be relatively easy to implement and even parallelize, they do come with a significant computational cost. Depending on the task at hand, this drawback can render the use of ensemble methods infeasible.

Test-time data augmentation - These methods are inspired by ensemble methods and adversial examples [2]. They involve creating multiple augmented test samples using some data augmentation techniques, testing them, and obtaining a predictive distribution.

This captures the uncertainty by leveraging the introduced variety of samples. These methods are easier compared to the others discussed since they don't require modifying the model or collecting extra data. However, it is crucial to find a suitable augmentation technique for the task and data to avoid generating data outside the target distribution. These methods are commonly used in tasks where data collection is challenging or costly, such as medical imaging.

In the present work, we incorporate Bayesian methods with sampling-based approximations to estimate uncertainty in DNN, building on the framework proposed by Clements et al. [8].

2.3. Monte Carlo Tree Search (MCTS)

MCTS is a planning algorithm for decision-making that takes random samples in the action space and builds a search tree guided by statistics captured on the nodes (e.g., visitations counts, estimated value, etc.) [6].

In detail, the root of the tree corresponds to the current state. Each node in the tree represents a state $s \in \mathcal{S}$. Edges in the tree represent an action $a \in \mathcal{A}$ taken from the source node s to reach the target s' child node, forming the so-called state-action pairs (s, a). These edges store various statistics, such as the visitations $n(s, a)$, the cumulative reward $W(s, a)$, and the averaged action value estimate $Q(s, a) = W(s, a)/n(s, a)$ [6]. Additional custom statistics can also be stored as required.

At each step, the tree is traversed by selecting paths, also referred to as traces or trajectories, based on the tree policy π and a specific selection rule. Upon reaching a leaf node, the tree is expanded by adding a new node. From it, a simulation (random, by default) is run to obtain a value estimate of the expanded node. Finally, the estimated value is propagated upwards along the path, updating the statistics of all implicated nodes. Upon reaching a specified budget, the accumulated statistics in the root node are used to make the final decision for that particular state. Figure 2.2 illustrates the MCTS algorithm.

Vanilla MCTS consists of four phases:

1. Selection:
   A tree policy π guides the tree traversal, selecting the action to be taken from the current node based on the action values associated with the potential edges. This process continues until it reaches a leaf node, which is defined as a non-terminal node with unvisited actions. A commonly used selection rule combines ϵ-greedy, and Upper Confidence Bound applied to trees (UCT) [15].

   UCT aims to strike an exploration-exploitation balance based on the available statistics. Its formula is given as follows [34]:

$$UCT(s, a) = Q(s, a) + C_{uct} \sqrt{\frac{log\, n(s)}{n(s, a)}} \tag{2.18}$$

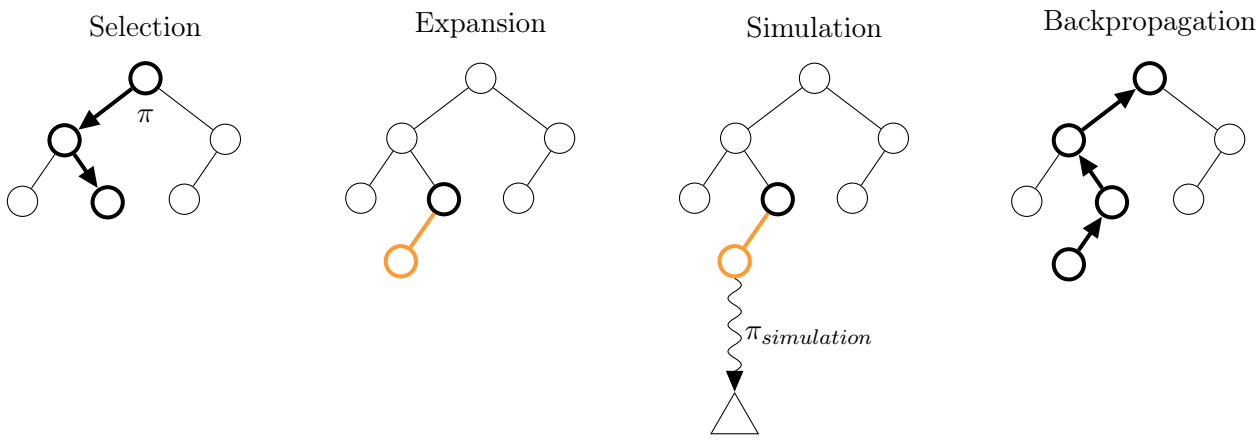

Figure 2.2.: Illustration of the four phases of MCTS. In the **Selection** phase, guided by the tree policy π, the tree is navigated until a leaf node is reached. The **Expansion** phase involves choosing an action and adding the resulting state to the tree. In the **Simulation** phase, the value of the newly added node is estimated through a simulation guided by the simulation policy $\pi_{simulation}$. Finally, the **Backup** phase propagates the estimated value upwards, updating all visited nodes along the trajectory. Illustration based on Browne et al. [6].

The first component of the formula, Q-value $Q(s,a)$, encourages exploitation, while the second component encourages exploration. The exploration rate $C_{uct} > 0$ regulates the level of exploration, and $n(s)$ represents the total visit count of state s: $n(s) = \sum_a n(s,a)$.

Equation 2.18 is evaluated on the valid actions a for the current state (node) s. Finally, the action with the highest value is selected.

2. Expansion
   When a leaf node is reached, the tree is expanded by randomly selecting a valid action and adding the resulting state as a new leaf node s_L.

3. Simulation
   From the newly added node, a simulation is run following a simulation policy $\pi_{simulation}$. By default, this policy involves sampling actions uniformly at random. Once completed, the resulting accumulated reward is assigned as the value of the leaf s_L.

4. Backup
   In this final phase, the estimated value of the added node is propagated back, or "backed up," through the edges and nodes visited on the current iteration, updating their respective statistics.

The presented four phases are executed iteratively until a predefined budget is consumed, such as time, memory, or a fixed number of executions N (traces). When the budget is reached, the search is considered complete, and the action of the root node is chosen based on some criteria, often by selecting the edge that has been visited the most, which is considered more robust [6].

2.4. AlphaZero

Silver et al. [34] introduced AlphaZero as an iteration of the program that aims to learn to play challenging games with minimal prior information that works with a fully defined environment.

Conceptually, AlphaZero aims to learn a policy that maximizes the game rewards, and it achieves this through two key components: a DNN, which estimates the policy and value of a given game state, and the MCTS planning algorithm, responsible for decision-making at each game step.

In detail, the DNN f_θ with parameters θ takes the board representation as state s, also named position, as input. It produces two outputs, denoted as:

$$(\mathbf{p}, v) = f_\theta(s) \tag{2.19}$$

where the output $\mathbf{p}$ represents probabilities for moves, indicating the likelihood of selecting each valid action (or move), while v is the estimated outcome of the game from state s.

The MCTS algorithm employed here is extended by storing an additional statistic on state-action pairs (s, a), denoted as $P(s, a)$, representing the prior probability of selecting a in s [34]. Furthermore, selection's UCT definition (equation 2.18) is updated to weight the exploration component by the corresponding prior probability $P(s, a)$ as follows [34]:

$$UCT(s, a) = Q(s, a) + C_{uct}\, P(s, a)\, \frac{\sqrt{n(s)}}{n(s, a) + 1} \tag{2.20}$$

Moreover, the simulation phase undergoes replacement by direct evaluation of the added leaf node using the DNN f_θ. Figure 2.3 illustrates this modification.

AlphaZero's training relies exclusively on self-play RL, which consists of an iterative process involving:

- Initialization: At the very beginning, the DNN f_θ is initialized (generally, with random weights).
- Self-play: AlphaZero plays against itself, guided solely by its predefined components that continuously refine. At each step of a game episode, an MCTS tree is built using the latest version of the DNN f_θ for evaluation. The built tree yields a search probability π_t to autonomously select action a_t reaching the next state s_{t+1}. Upon reaching the final step s_T, the final game score Z is calculated according to the game rules.

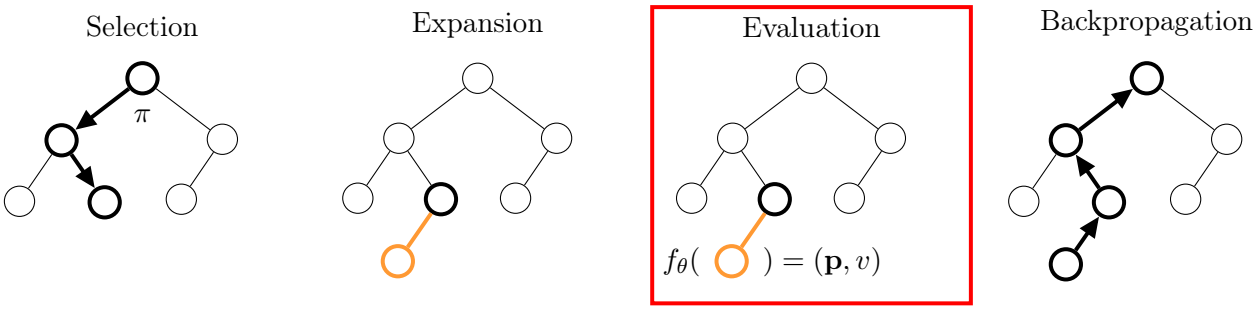

Figure 2.3.: MCTS phases for AlphaZero. The introduced **Evaluation** phase replaces the original **Simulation** phase. It directly evaluates the newly added state during **Expansion** phase using the DNN f_θ instead of running simulations. The predicted value v is propagated upwards on the **Backup** phase as before.

- DNN training: After the self-play episode is completed, the DNN is trained using the collection of states s_t, search probabilities π_t, and game score Z obtained via self-play. Specifically, each state s_t is used as input to the network f_θ, which outputs a policy vector $\mathbf{p}_t$ and a predicted outcome v_t. The DNN parameters θ are updated to maximize the similarity between the MCTS policy vector $\mathbf{p}_t$ and DNN's policy vector π_t while minimizing the error between the predicted outcome v_t and the game episode outcome Z. The loss function is defined as:

$$\mathcal{L}_{AZ} = (Z - v)^2 - \boldsymbol{\pi}^\mathsf{T} log\mathbf{p} + c||\theta||^2 \tag{2.21}$$

  where parameter c regulates the DNN's $L2$ weight regularizer.

Through this iterative training process, both the DNN and the MCTS continually improve. Figure 2.4 presents the self-reinforcement learning loop.

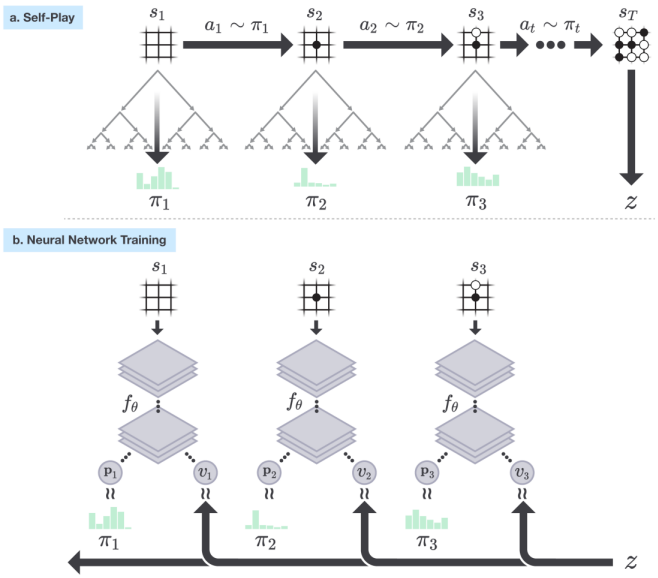

Figure 2.4.: **a. Self-play:** an episode of the game is run from empty board s_1 until the end of game s_T. For each state s_t, an MCTS tree is built using the latest DNN f_θ. The tree outputs a search probability π_t, which is used to choose the action $a_t \sim \pi_t$. On the final state, s_T, the game outcome Z is computed. **b. (Deep) Neural network training:** for each state used on the self-play episode s_t, the neural network is evaluated with it as input. The DNN outputs a probability distribution over moves $\mathbf{p}_t$ and outcome v_t. The DNN parameters θ are updated by using the loss defined on equation 2.21. The updated DNN f_θ is used on the next self-play episode. Figure taken from Silver et al. [32]

3. Related work

This chapter provides an overview of relevant related work. Firstly, we review the AlphaZero algorithm and its more recent version, MuZero. Subsequently, we discuss relevant approaches for estimating uncertainty in neural networks and MCTS.

3.1. Alphazero and MuZero

AlphaGo [32], the pioneering work by the DeepMind team, is an algorithm that masters the challenging game of Go, surpassing human professionals. It employs a combination of MCTS and two separated DNNs to model the policy, which selects the next action, and the value function, which predicts the score. AlphaGo's learning involves analyzing and training from expert gameplay data. Building on its success, subsequent variations emerged, including AlphaZero [33] and MuZero [31].

The AlphaZero algorithm was designed to achieve proficiency in two-player zero-sum games with given rules. This is accomplished through a process of self-play, wherein it generates samples of game episodes using a combination of a DNN and MCTS. Both components are continuously refined by automatically fine-tuning the DNN parameters based on the latest self-play episode's search probabilities and score. The MCTS subsequently employs the enhanced DNN to generate new episodes. The learning process in AlphaGo involves using pre-trained DNNs trained on expert gameplay. In contrast, AlphaZero represents a significant improvement by eliminating the need for pre-training. This approach reduces biases introduced by human expert gameplay and offers a more cost-effective and reliable solution by avoiding the requirement for an extensive number of expert gameplay samples. AlphaZero is categorized as a model-free RL algorithm since it does not aim to learn the environment dynamics or rules (i.e., it doesn't build a model); instead, it leverages these dynamics directly, supporting the planning when learning.

MuZero, in contrast, is a model-based RL algorithm as it learns an internal model of the environment's dynamics, allowing it to predict outcomes and rewards without needing explicitly defined rules. To do so, it solely models the essential information to the agent rather than the entire environment. MuZero's model utilizes three crucial learnable components: representation, dynamics, and prediction functions. The representation function efficiently encodes the typically large input (state), while the prediction function derives the policy and value from a given state. The dynamics function calculates the reward and next state. Through learning these components, the MuZero algorithm can master environments where the model is not given.

While AlphaZero and MuZero have delivered outstanding results with reduced information requirements, they do demand significant resources and time. The training phase

involves generating a large quantity of self-play episodes, which requires constructing a substantial number of MCTS trees within a fixed budget. This budget—namely, the number of MCTS tree traces—can be excessive in scenarios where the agent has high certainty about its moves. Consequently, there is potential for improvement by incorporating uncertainty into budget allocation.

To address this challenge, we propose an adaptive search trace budget allocation method that enhances the efficiency of the planning process. Our approach dynamically adjusts the search trace budget based on uncertainty estimates gathered during tree construction. In the following sections, we review existing methods for deriving these estimates from both the DNN and MCTS components.

The budget allocation optimization ensures more effective resource utilization, with the primary goal of improving the AlphaZero algorithm in this work.

3.2. Neural Network uncertainty estimation

Uncertainty in neural networks is classified into two types: aleatoric and epistemic. Aleatoric uncertainty stems from inherent randomness in the input data (i.e., variability or noise) and is not reducible. In contrast, Epistemic uncertainty arises from a lack of knowledge and can be reduced through learning [14].

Various approaches have introduced diverse techniques for estimating uncertainty in neural networks. Some aim to assess both aleatoric and epistemic uncertainty, while others focus specifically on the latter.

Moskovitz et al. [28] analyzes the significance of optimism and pessimism in exploration within the context of RL. Optimism refers to the belief that exploration can yield favorable outcomes, while pessimism reflects the concern that overestimating expected rewards might occur, leading the agent to be cautious and reduce exploration. They hypothesize that the level of optimism must be dynamically adapted since it depends on specific tasks and the current learning phase. Thus, they propose a dynamic adaptation of the optimism level to govern exploration in RL.

They introduce the Tactical Optimism and Pessimism (TOP) framework, which builds on the actor-critic architecture to guide decision-making based on uncertainty estimates, learning when to adopt an optimistic or pessimistic outlook. In this approach, the optimism/pessimism choice is framed as a multi-armed bandit problem [19][7], with each arm representing a fixed level of optimism. In the actor-critic framework, the actor proposes actions, and the critic evaluates them, with both being updated based on the selected level of optimism.

The authors model the aleatoric uncertainty by learning the full return distribution using a parametric neural network [4] rather than the solely expected return, with aleatoric uncertainty represented by the spread of this distribution. Epistemic uncertainty is estimated as a Gaussian distribution, where the mean and standard deviation are derived from the aforementioned return distribution across multiple critics. Two return distributions are generated using the neural network with distinct parameter sets to assess these uncertainties. A belief distribution is formed utilizing the quantile representation

of these distributions, balancing their mean and standard deviation quantile estimates with an optimism parameter: positive values induce optimism, while negative values pessimism. The actor and critics are updated based on this belief distribution, which varies the exploration level at each training step. The optimism parameter is dynamically adjusted as a multi-armed bandit, updating based on outcomes from each episode.

While Moskovitz et al. [28] 's method offers a promising direction for adaptive exploration, their approach is not directly applicable to the AlphaZero setup, as it is not framed within an actor-critic architecture.

Clements et al. [8] also base the estimations on the full return distribution [4] and estimation of it via quantiles [9]. They define the uncertainties as follows: epistemic is represented as the average of the variance of quantiles, while aleatoric uncertainty is defined as the variance of the expected value of the quantiles. Multiple samples of parameters (features) are required to get these values. Since this is computationally expensive, an approximation is proposed by framing quantile learning as a Bayesian inference problem. In detail, the authors use only two neural networks with parameterizations sampled via randomized maximum a posteriori (MAP) sampling [30]. The approximations are then used on an updated version of the Deep Q Networks (DQN) [25], named Uncertainty Aware DQN (UA-DQN).

We refer the reader to Lockwood and Si [22] for a comprehensive compilation of various approaches to estimate uncertainty in neural networks.

In this work, we extend the approach of Clements et al. [8] to estimate epistemic uncertainty in AlphaZero's DNN. Unlike their method, we only require the estimated value for trace budget allocation without influencing the MCTS selection phase as in their UA-DQN framework. Their approach, designed for DQN, uses a DNN that outputs the return estimate as a distribution. In contrast, AlphaZero DNNs include an additional output head for prior predictions. To address this, we adapt their method to also capture uncertainty from the prior head. Furthermore, we extend the loss functions from their approach and incorporate them into the AlphaZero framework to balance the learning dynamics between this distributional RL adaptation and the original AlphaZero.

3.3. MCTS uncertainty estimation

Various approaches have been proposed to gain insight into the confidence level during the construction of a search tree. They include measuring the variance of the action score compared to a reference score [18], considering the variance in transitions [16], and analyzing the morphological characteristics of the search tree [27].

Lan et al. [18] define uncertainty as the confidence that the current best action remains unchanged after a fixed number of simulations. Three approaches were presented to predict this uncertainty: The first two methods focus on utilizing information from the current state only. One approach involves using the original neural network, while the other involves training an external neural network. A third method incorporates information from the entire tree by utilizing an external neural network. The estimated uncertainty is then applied in a Monte Carlo Tree Search (MCTS) variant called Dynamic Simulation

MCTS (DS-MCTS), which determines whether to terminate the search based on the uncertainty value and a predefined threshold. This approach, however, has limitations. The most promising versions require pre-training an auxiliary DNN, which needs self-play data generated with specific parameters like maximum trace count. This incurs additional computation, and the auxiliary DNN, not benefiting from AlphaZero learning, depends solely on its ability to generalize. Furthermore, the DS-MCTS is sensitive to hyperparameters, as maximum trace counts and thresholds must be manually defined.

Another approach, which focuses on tracking the discrepancy between the MCTS tree and a reference, is presented by Kohankhaki [16]. It highlights the performance degradation when perfect simulation models are unavailable. The imperfect model, called the dynamics, is the environment model accessible to the agent. The author proposes an iterative improvement approach for this model. The authors introduce a model called Uncertainty Adapted MCTS (UA-MCTS), where uncertainty is modeled as the discrepancy between the environment and the imperfect model. Their main idea is to make the MCTS more conservative in states with high uncertainty. To estimate uncertainty, they utilize the difference between the actual next state and the predicted next state generated by the imperfect model. Hence, the uncertainty model is constructed through a learning process that involves buffering transitions using vanilla MCTS and comparing. Modifications to MCTS prioritize nodes with lower uncertainty in the selection, expansion, and backpropagation phases. Simulation-related changes are inapplicable to AlphaZero, which omits rollouts. Moreover, the approach proposed by Kohankhaki [16] requires training an imperfect model, which proves costly and is not justified in the context of a well-defined environment like on AlphaZero setup. Therefore, estimating uncertainty using this approach would impose unwarranted overhead on the AlphaZero setup.

A distinct approach is presented by Moerland et al. [27]. They identify the first type of uncertainty, inherent to the MCTS selection rule, derived from the number of visits to a state. In addition, they identify a second type of uncertainty induced by size variations of subtrees under a state, which is not inherently accounted for in the algorithm. To address this, they present an uncertainty approximation method based on the morphology of the trees in MCTS. They observe that significant variations in subtree sizes between different actions can lead to poor performance in the MCTS search. The paper introduces a methodology called MCTS-T, which tracks and backs up subtree sizes for each state in the tree. These values are then incorporated into the MCTS selection phase through the Upper Confidence Bound (UCB) formula to enhance exploration. The authors note that choosing an action with more subtrees leads to sampling — and then exploring — from a set of highly uncertain traces, while choosing an action with fewer subtrees provides more informative samples. Additionally, Moerland et al. [27] propose an extension called MCTS-T+ to handle loop scenarios where a state appears multiple times in a trace. While this extension demonstrates favorable results by avoiding redundant subtree construction, it does not apply to the environments considered in this study, as they do not contain such loops.

In this work, we adopt the core concept of estimating uncertainty by tracking subtree sizes (MCTS-T) during MCTS tree construction. Unlike the original MCTS-T approach,

we leave the selection phase unmodified, focusing solely on estimating MCTS uncertainty to enable further optimization of trace allocation without significant alterations to the AlphaZero MCTS component.

4. Methodology

This chapter describes the methodology used to address the research questions, including the strategy to estimate the uncertainty on AlphaZero. Subsequently, it outlines the strategy to dynamically allocate the MCTS max traces budget.

4.1. Uncertainty estimation

As detailed before, our study's subject, the AlphaZero algorithm, consists of the integration of DNN and MCTS trees. We consider the uncertainty stemming from both components following the strategies listed herein.

DNN uncertainty

Following Clements et al. [8], uncertainty estimation in the DNN is achieved by framing the problem within a Distributional RL (DRL) approach, where DNN predicts entire distributions rather than single values. Their approach, originally applied to return, models the full return distribution $Z^\pi(s,a)$ rather than the expected return. See 2.1.1 for further details on DRL.

As proposed by Dabney et al. [9] on their QR-DQN algorithm, the return distribution $Z^\pi(s,a)$ is estimated by a probability distribution $Z(s,a)$ framed into N quantiles. With fixed probabilities $q_i = 1/N$ ($i \in [1,N]$), the CDF F_Z is defined by cumulative probabilities $\tau_i = i/N$ and *quantile midpoints* $\hat{\tau}_i = \frac{\tau_{i-1}+\tau_i}{2}$. Dabney et al. [9] estimate the quantile midpoints' values, or *locations*, $z_i = F_Z^{-1}(\hat{\tau}_i)$ by minimizing the custom QR loss formulated in Equation 2.16. Further details on QR-DQN are in 2.1.4.

Clements et al. [8] suggest learning these quantile values using Bayesian inference. They propose to use a DNN with parameter set θ to learn the quantile values. It has N outputs $y_i(\theta,s,a)$, each corresponding the the quantile value z_i. Consider a state s, action a, policy π, and a dataset $D = (\tilde{r}_1, ..., \tilde{r}_K)$ containing K observed return samples from the probability distribution of return $Z^\pi(s,a)$. The likelihood to be maximized is defined as:

$$
\begin{aligned}
P(D|\theta) &= \prod_{j=1}^{K}\prod_{i=1}^{N} f_{\tau_i}(\tilde{r}_j - y_i(\theta,s,a)) \\
where \quad f_\tau(u) &= \frac{\tau(1-\tau)}{\sigma_D} exp\big(-\frac{\rho_\tau(u)}{\sigma_D}\big)
\end{aligned}
\tag{4.1}
$$

where σ_D scale is a parameter, and ρ_τ a function that evaluates the error u, weighting it asymmetrically according to the quantile level τ, as defined in Equation 2.13.

Maximizing equation 4.1 is equivalent to minimizing the custom quantile regression loss defined in Equation 2.16.

To draw samples from the posterior distribution $P(\theta|D)$, Clements et al. [8] assume that the normal prior on θ is centered around 0, and use approximate MAP sampling [30].

Clements et al. [8] formulate the uncertainties on the return distribution $Z^\pi(s,a)$ as follows:

Epistemic uncertainty is quantified as the variance in predicted quantile values across sampled parameter sets θ from the posterior distribution $P(\theta|D)$, with overall aggregation as the mean variance across quantiles. This is expressed as:

$$\sigma^2_{epistemic} = \mathbb{E}_{i\sim\mathcal{U}\{1,N\}}\big[\,\mathrm{var}_{\theta\sim P(\theta|D)}(y_i(\theta,s,a))\big] \tag{4.2}$$

Aleatoric uncertainty is defined as the variance of the expected quantile values within the posterior distribution of θ, isolating it from the epistemic uncertainty that arises from the distribution over θ. Formally:

$$\sigma^2_{aleatoric} = \mathrm{var}_{i\sim\mathcal{U}\{1,N\}}\big[\mathbb{E}_{\theta\sim P(\theta|D)}(y_i(\theta,s,a))\big] \tag{4.3}$$

Where $\mathcal{U}\{1,N\}$ is the uniform distribution over the integers 1 to N.

Furthermore, to avoid the need for a large number of samples from the posterior distribution $P(\theta|D)$, they propose approximations to the uncertainties by using only two samples (θ_A and θ_B) as follows:

$$\tilde{\sigma}^2_{epistemic} = \frac{1}{2}\mathbb{E}_{i\sim\mathcal{U}\{1,N\}}\big[(y_i(\theta_A,s,a)) - (y_i(\theta_B,s,a))\big]^2 \tag{4.4}$$

$$\tilde{\sigma}^2_{aleatoric} = cov_{i\sim\mathcal{U}\{1,N\}}\big(y_i(\theta_A,s,a), y_i(\theta_B,s,a)\big) \tag{4.5}$$

We extend the original AlphaZero DNN outputs shape — an array of probabilities for the policy head output $\mathbf{p}$ and a single scalar for the value head output v — to incorporate the specified number of quantiles. This modification enables the computation of quantile values for the return distribution for the value head, denoted as $\hat{Z}^v$, and action-specific probabilities for the policy head, denoted as $\hat{Z}^{\mathbf{p}}$, both essential for integrating quantile regression. The policy head output now consists of $K \times N$ quantile values, where $K = |\mathcal{A}|$ is the number of actions in the action space $\mathcal{A}$ and N is the number of quantiles, while the value head output consists of N quantile values. Figure 4.1 illustrates the modifications.

Moreover, we require two sampled DNNs, referred to as posteriors (θ_A and θ_B). Following Clements et al. [8], we end up with three DNNs (models) of identical architecture, each with the described output heads configuration and independent initialization. This setup aims to maintain one main model solely for predictions while the other two function as posteriors.

Additionally, following Clements et al. [8] work, we track copies of the posterior's initial weights, referred to as anchors, which serve as a reference for penalizing deviations during training. This acts as a regularizer, promoting weight stability and discouraging significant shifts.

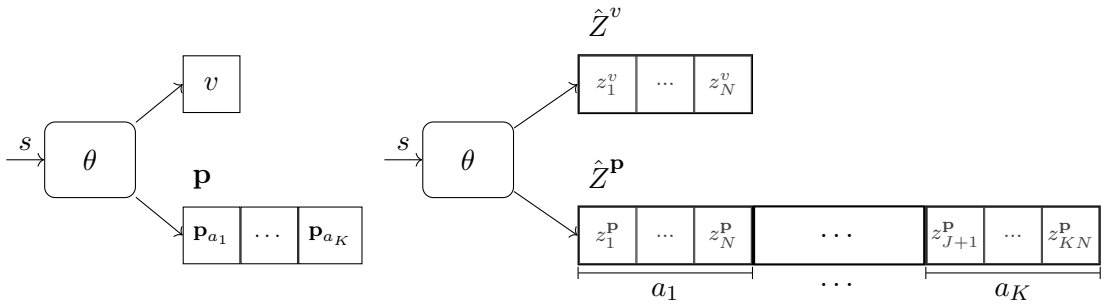

Figure 4.1.: Illustration of the modifications on the DNN θ output heads. Here, J denotes $(K-1)*N$. For an input state s, action space $\mathcal{A}$ of $K = |\mathcal{A}|$ actions and N quantiles, the original DNN (left) has a value head outputting a single value v and a policy head predicting action probabilities $\mathbf{p}_{a_i}$ for each action a_i. In the modified DNN (right), the value head output $\hat{Z}^v$ contains the quantile values z_i^v of the return (v) distribution, while the policy head output $\hat{Z}^{\mathbf{p}}$ contains $K*N$ quantile values $z_i^{\mathbf{p}}$, representing the probability distributions of actions probabilities $\mathbf{p}_a$ for all actions a concatenated together. For AlphaZero-related calculations, $\hat{Z}^v$ is aggregated via its mean, and $\hat{Z}^{\mathbf{p}}$ by applying softmax to the mean quantile values across actions.

To ensure compatibility with AlphaZero, the DNN output heads predictions are aggregated for AlphaZero-related calculations. For the main model's value head output, $\hat{Z}^v$, we obtain a single scalar by taking its mean. For the policy head output, $\hat{Z}^{\mathbf{p}}$, we compute the mean of the quantile values for each action, followed by a softmax operation across these means to yield valid probability values.

Since AlphaZero's DNN has two output heads, we compute the quantile regression loss separately for both the value and the policy head. Algorithm 3 details the calculation for the value head, where the loss is computed by using TD update as the target for the quantile regression loss function. For the policy head, however, TD updates are unsuitable because the predictions represent the locations (i.e., quantile values) of a probability distribution over actions rather than over the value (return). Instead, we directly apply the quantile regression loss to the predicted locations for the chosen action from both the network and the target network, as shown in Algorithm 4. Specifically, we extract the N-dimensional array of predicted locations corresponding to the chosen action from each network's output and apply the referred loss.

Each head loss is computed for every network instance (main model, posterior 1, and posterior 2) and summed to produce the cumulative quantile losses $\mathcal{L}_{QR}^v$ and $\mathcal{L}_{QR}^{\mathbf{p}}$. The overall quantile loss $\mathcal{L}_Q$ is defined as:

$$\mathcal{L}_Q = \alpha_Q * \mathcal{L}_{QR}^{\mathbf{p}} + (1 - \alpha_Q) * \mathcal{L}_{QR}^v \tag{4.6}$$

where α_Q servers as a weighting parameter between the cumulative quantile losses from value heads $\mathcal{L}_{QR}^v$ and the prior heads $\mathcal{L}_{QR}^{\mathbf{p}}$.

Algorithm 1: Quantile Regression Loss for Value Head

Input: Predicted quantile locations of value head from network θ and target network θ_T: $\hat{Z}^v_\theta$, $\hat{Z}^v_{\theta_T}$; sample's reward r, done flag d, and discount factor γ; and number of quantiles N

Output: Quantile regression loss for value head $\mathcal{L}^v_{QR}$

`/* Compute target value using TD update */`

$Z' = \mathrm{r} + (1 - \mathrm{d}) \cdot \gamma \cdot \hat{\mathrm{Z}}^v_{\theta_T}$

`/* Calculate loss using Equation 2.17 with target value` Z' `*/`

$\mathcal{L}^v_{QR} = \sum_{i=1}^{N} \mathbb{E}_j \left[\rho^\kappa_{\tau_i} (Z'_j - \hat{Z}^v_\theta) \right]$

return $\mathcal{L}^v_{QR}$

Algorithm 2: Quantile Regression Loss for Policy Head

Input: Probabilities' locations predictions from network θ and target network θ_T: $\hat{Z}^{\mathbf{p}}_\theta$, $\hat{Z}^{\mathbf{p}}_{\theta_T}$; sample's action a, and number of quantiles N

Output: Quantile regression loss for policy head $\mathcal{L}^{\mathbf{p}}_{QR}$

`/* Extract predicted locations for chosen action` a `from` $\hat{Z}^{\mathtt{p}}_\theta$ `and` $\hat{Z}^{\mathtt{p}}_{\theta_T}$ `*/`

$\hat{Z} = \hat{Z}^{\mathbf{p}}_\theta[a]$

$Z' = \hat{Z}^{\mathbf{p}}_{\theta_T}[a]$

`/* Calculate loss using Equation 2.17 */`

$\mathcal{L}^{\mathbf{p}}_{QR} = \sum_{i=1}^{N} \mathbb{E}_j \left[\rho^\kappa_{\tau_i} (Z'_j - \hat{Z}) \right]$

return $\mathcal{L}^{\boldsymbol{p}}_{QR}$

The loss function is updated to incorporate multiple components, including the computed composed QR Loss $\mathcal{L}_Q$, anchor loss $\mathcal{L}_A$, which quantifies the disparity between posterior and anchor parameter weights, and AlphaZero loss $\mathcal{L}_{AZ}$ as follows:

$$L = \alpha_{AZ} * \mathcal{L}_{AZ} + (1 - \alpha_{AZ}) * (\beta_Q * (\mathcal{L}_Q + \mathcal{L}_A)) \tag{4.7}$$

where α_{AZ} denotes a weighting parameter, and β_Q scales the losses associated with uncertainty estimation.

4.1.1. MCTS uncertainty

Following Moerland et al. [27], we track the so-called remaining uncertainty $\sigma_\tau(s) \in [0, 1]$ which represents the state s attached subtrees' depth and relates it to its exploration. For each state s in the tree, σ_τ is recursively estimated and backed up, with $\sigma_\tau(s) = 1$ indicating a completely unexplored subtree below s, and $\sigma_\tau(s) = 0$ indicating a fully explored subtree.

In the MCTS expansion phase, the added leaf node s_L's remaining uncertainty is initialized as:

$$\sigma_\tau(s_L) = \begin{cases} 0, & \text{if } s_L \text{ is terminal} \\ 1, & \text{otherwise} \end{cases} \tag{4.8}$$

The remaining uncertainty is then recursively propagated through the trace on the MCTS backpropagation phase, following the update rule in Equation 4.11.

Calculating the remaining uncertainty $\sigma_\tau(s)$ of state s involves considering the successor states s'' reached by performing each action a on the action space $\mathcal{A}$.

As Moerland et al. [27] highlight, to maintain the MCTS strategy of favoring the most promising (frequently visited) actions, their approach weights each successor's remaining uncertainty by its visitation count $n(s, a)$. To handle untried actions (i.e., those without visitations), they propose the weight function $m(s, a)$ defined as follows:

$$m(s,a) = \begin{cases} n(s,a), & \text{if } n(s,a) \geq 1 \\ 1, & \text{otherwise} \end{cases} \tag{4.9}$$

Additionally, since untried actions lack an inherent measure of remaining uncertainty, they suggest assigning these actions' virtual state the maximum remaining uncertainty (1). This modified remaining uncertainty of successor states, $\sigma^*_\tau(s'')$, is defined as follows:

$$\sigma^*_\tau(s'') = \begin{cases} \sigma_\tau(s''), & \text{if } n(s,a) \geq 1 \\ 1, & \text{otherwise} \end{cases} \tag{4.10}$$

Finally, the estimated remaining uncertainty of an state s with successor states $s'' \sim \mathcal{T}(\cdot|s, a)$ for $a \in \mathcal{A}$ is updated as follows:

$$\sigma_\tau(s) = \frac{\sum_a m(s,a) * \sigma^*_\tau(s'')}{\sum_a m(s,a)} \tag{4.11}$$

4.2. Dynamic selection of number of traces

We frame the selection of the number of traces in the MCTS tree search process in two stages. Initially, a fixed number of warm-up traces are allocated to assess preliminary performance. In the subsequent selection phase, a basic DQN agent determines the number of additional traces to allocate based on the observed context after the first stage. Figure 4.2 illustrates the modifications on the MCTS search.

To formalize this approach, we define its dynamics as a meta-MDP $(\tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \tilde{\mathcal{T}}, \tilde{\mathcal{R}}, \tilde{\gamma})$ built upon the base MDP of AlphaZero MCTS planning $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, as follows:

- Meta-state space $\tilde{\mathcal{S}}$: represents the context after running a certain number of traces, encapsulating performance and uncertainty. It includes the metrics obtained from executing traces in the MCTS (base MDP). In our work, specifically, the meta-state $\tilde{s} \in \tilde{\mathcal{S}}$ contains five metrics: MCTS uncertainty, DNN epistemic uncertainty, the average number of loss states (i.e., terminal states with unfavorable outcomes, such as death in games), and average achieved rewards within run traces and entropy over the action visitation counts at MCTS tree root.

- Meta-action space $\tilde{\mathcal{A}}$: represents the choices of the additional number of traces to allocate, influencing the MCTS' ability to make informed decisions with balanced computational cost. $\tilde{\mathcal{A}}$ consists of a discrete set of possible extra trace counts $\tilde{a}$.
- Meta-Transition Function $\tilde{\mathcal{T}}$: describes how the meta-state $\tilde{s}$ transitions to a new meta-state $\tilde{s}'$ after using the specific number of extra traces $\tilde{a}$ chosen by the (DQN) agent. The transition is affected by the stochasticity of the base MDP. Thus, the function is defined as: $\tilde{\mathcal{T}}(\tilde{s}, \tilde{a}, \tilde{s}') = \tilde{\mathcal{T}}(\tilde{s}, \tilde{a}) + \eta$ where η represents the stochasticity in the response of the base MDP to the allocated extra traces.
- Meta-Reward function $\tilde{\mathcal{R}}$: evaluates the efficiency of the allocated extra traces, emphasizing performance improvement relative to the initial warm-up traces results. This function rewards effective performance with fewer traces and penalizes inefficiency by allocating a larger number of traces. Equation 4.12 defines the proposed meta-reward function.
- Meta-discount factor $\tilde{\gamma}$: specifies how much the agent values future rewards relative to immediate ones.

With the meta-MDP framework established, we now define the custom meta-reward function, R_k^{traces}, which evaluates the efficiency of the allocated extra traces by measuring the performance improvement in average reward relative to the warm-up traces results, adjusted for the cost of extra traces. Let the rewards obtained by all traces represented as $\{r_1, r_2, \ldots, r_w, \ldots, r_n\}$, where n is the total number of traces (including the extra traces, m_k) and w the number of warm-up traces. The reward R_k^{traces} at some search step k is defined as:

$$R_k^{\text{traces}} = \frac{\frac{1}{n}\sum_{i=1}^{n} r_i - \frac{1}{w}\sum_{i=1}^{w} r_i}{\frac{m_k}{T}} \tag{4.12}$$

The parameters, including warm-up traces w, possible values of additional traces m_k, and maximum total traces T, might vary depending on the environment.

The DQN agent aims to learn a policy $\tilde{\pi} : \tilde{S} \to \tilde{A}$ that maximizes the reward over time by mapping the observed meta-states (context) to optimal meta-actions (extra traces to allocate). This DQN agent consists of a simple DNN with linear layers, which takes the meta-state $\tilde{s}$ as input and outputs a vector of estimated Q-values $Q(\tilde{s}, \tilde{a})$ for each action $\tilde{a}$ (representing the number of extra traces). These Q-values estimate the expected cumulative reward for each choice of extra traces given the current context $\tilde{s}$. The agent selects actions by choosing the one with the highest estimated Q-value, $\tilde{a}^*$.

During training, experiences - including the meta-state $\tilde{s}$, selected action $\tilde{a}^*$, obtained reward (as defined in Equation 4.12), and resulting next meta-state $\tilde{s}'$ - are stored in a replay buffer, with all experiences marked as ongoing (done=False). The agent follows the standard DQN learning methodology, sampling from the replay buffer and using a target network and Double DQN (DDQN) to stabilize learning. The objective is to approximate the optimal action-value function by minimizing the TD error, as in Equation 2.9.

Table A.2 details hyper-parameters adjusted based on environment complexity.

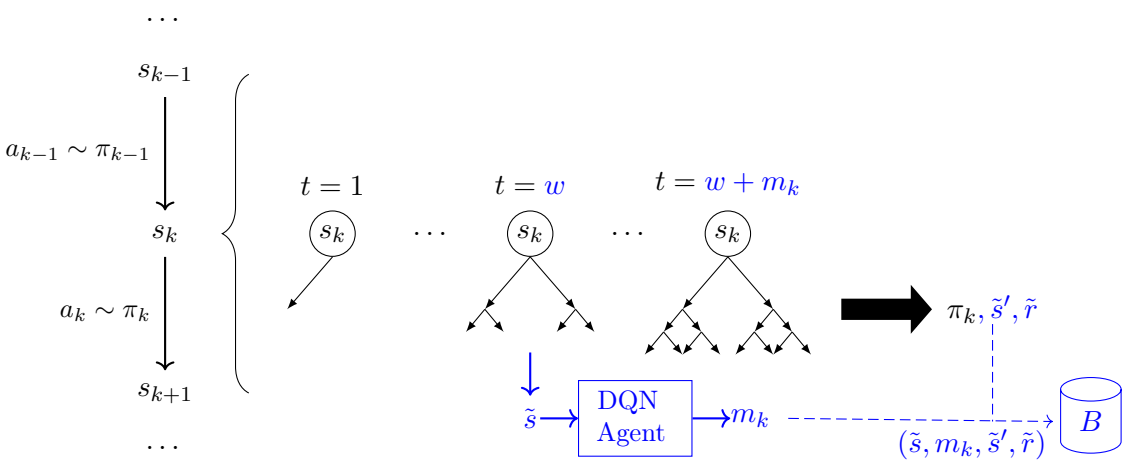

Figure 4.2.: Dynamic trace allocation on MCTS search. Key changes are highlighted in blue. At step k with state s_k, the search (right) begins with a warm-up of w traces. A context $\tilde{s}$ is derived from uncertainty and statistics after the warm-up. The DQN agent uses this context to allocate additional traces m_k. The search proceeds until $w + m_k$ traces are reached, yielding the policy π_k, which is used to advance to the next state s_{k+1}. Additionally, a final context $\tilde{s}'$ and a reward $\tilde{r}$ for the allocation efficiency are generated. The experience tuple $(\tilde{s}, m_k, \tilde{s}', \tilde{r})$ is stored in buffer B for DQN agent's DNN training.

5. Environments

This chapter briefly introduces the environments used in this work, with further details in Appendix A.3. We used the wrapped version of them available in the Gymnasium library [36]. First, to verify the proposed method, we used CartPole as a toy environment since it's simple and a common choice for the task. Then, a more challenging set of environments was used: MinAtar [38].

5.1. CartPole

Cartopole is a classic RL environment broadly used because of its simplicity and bounded actions and rewards. It consists of a pole attached to a cart that can be moved freely (without any friction or other physics actions) on a plain horizontal surface. While moving, the pole acts like a pendulum, which the agent would try to keep balanced. The agent is limited to producing movement step by step by applying forces. The velocity of the movements and the resistance are affected by the angle of the pole and its center of gravity, respectively. The state is represented as a vector.

5.2. MinAtar

MinAtar is a set of environments inspired by Atari 2600 games, simplified to make them more suitable for RL tasks. Young and Tian [38] detail their main improvements compared to Arcade Learning Environment ALE [23] (a standard implementation of Atari 2600 games):

- Reduced complexity: compared to ALE, whose frame information consists of matrices of 160x210 merging all game components into a single matrix, they separate each component into a channel as well as reduce the screen size, resulting in an observation space represented as a 10x10xn matrix, where n is the number of channels.

- Action space is limited to up to 6, game dependent.

- Added stochasticity: ALE implementation is fully deterministic, making it tricky for the agent to not fall into a repetitive, well-known path. MinAtar added sticky actions which with a probability of 0.1 perform the agent's last action instead of the current one.

5.2.1. Games

The MinAtar library provides five Atari 2600 simplified games. We used four of them, which are detailed below.

Asterix
The player moves around the screen along the four cardinal directions. Its goal is to collect a treasure while avoiding contact with enemies. Both treasures and enemies are spawned from the sides. The speed and spawn rate of enemies and treasure are periodically increased. The state is represented as a 10x10x4 matrix.

Breakout
The player moves a paddle at the bottom of the screen to bounce a ball, making it travel along diagonals. The bounce direction depends on the paddle side hit. There are also –initially– 3 rows of bricks on the top of the screen and walls on the borders. Whenever the ball hits a brick or a wall, it's reflected.

The goal is to make the ball break the most bricks as possible. When all bricks are cleared, a new set of 3 rows are added. The state is represented as a 10x10x4 matrix.

Seaquest
The agent oversees a submarine's movements across the sea, restricted to cardinal directions. The submarine encounters two primary elements: enemies (other submarines or fish) and divers. Enemies, particularly submarines, threaten by firing bullets, prompting the agent to avoid or counterattack. When enemies are shot, they disappear, and the player receives a reward. Divers can be collected onboard, with a maximum capacity of six. Oxygen levels within the submarine diminish over time but can be replenished upon surfacing (reaching the top of the screen) with at least one diver onboard. Upon resurfacing, if fewer than six divers are present, one is removed; if all six are aboard, they are all disembarked, earning an additional reward. After surfacing, enemy spawn rates and speed increase, intensifying the challenge. The state is represented as a 10x10x10 matrix.

Space Invaders
The player controls a mobile cannon at the bottom of the screen to shoot bullets upward at a cluster of aliens moving across the screen. When the cluster reaches the edges, its direction is switched and moves down. The aliens can also fire back to the player, and their speed is inversely proportional to the number of them. When the player removes all the aliens by shooting them, a new slightly faster cluster spawns.

The goal is to destroy the most possible aliens while avoiding contact with them or getting hit by their bullets.

The state is represented as a 10x10x6 matrix.

6. Experiments

This chapter presents the experiments conducted to evaluate our approach against the baselines, vanilla AlphaZero and DQN. The Setups section describes the experimental configurations, including architecture details, while the Results section provides results plots and their corresponding analysis, highlighting the effectiveness of the proposed approach.

6.1. Setups

Four experiments were conducted for each environment: a standard DQN agent (using DDQN) for reference, and three variations of AlphaZero—regular AlphaZero AZ, AlphaZero with Quantile Regression to account for uncertainty QR_AZ, and AlphaZero with dynamic trace allocation UA_AZ –extended from QR_AZ –. All AlphaZero variations shared the same maximum number of traces, with the first two utilizing the total trace limit, while the latter served as an upper bound.

Cartpole We used the Cartpole environment as a toy test of our approach, running 250,000 steps for the AlphaZero-based agents and 2,000,000 for the DQN agent.

MinAtar Our approach was evaluated on the Asterix, Breakout, Space Invaders, and Seaquest environments. We employed version 1 of these games, which reduces the action space to the minimum required for each game.

A fixed maximum episode length of 1,200 steps was imposed to standardize experiments due to the absence of upper bounds on episode duration in these environments. Upon reaching this limit, the environment flags the episode as truncated, signaling termination.

Given that the environment states are represented as 10x10 boolean matrices across several channels, we flattened the states to conserve computational resources, using only linear layers in the DNNs on the AlphaZero-based approaches.

Each variation was run for 1,000,000 steps. Due to the slower learning rate, the DQN variant was run for 5,000,000 steps to achieve comparable rewards.

Model architectures DQN experiment utilizes a single convolutional layer followed by a fully connected layer and an output linear layer, which outputs the Q-values for each action.

The AZ's model consists of a sequence of hidden linear layers followed by two output heads: one for the policy, representing the action probability distribution, and one for the value, predicting the expected return value.

Both QR_AZ's and UA_AZ's models share this structure but include an additional Dropout layer. Their output heads are extended to account for quantiles, where the policy head outputs the quantile values for each action probability, and the value head provides the value distribution.

The internal DQN agent of UA_AZ includes a simple neural network with two hidden layers. This network takes a 5-dimensional input (representing the context) and outputs the predicted reward for each possible number of additional traces.

Further architectural details are provided in Table A.2.

6.2. Results

Figures 6.1 - 6.5 illustrate key performance metrics across five plots for each environment, covering the first 250,000 steps in CartPole and 1,000,000 steps in MinAtar games.

The "Step Reward" plot tracks the evolution of step rewards, while "Step Traces" tracks the total number of traces used in MCTS searches.

To enable cross-environment comparison, rewards at each step t are normalized considering the final reward obtained after a full run by a DQN agent (r_{DQN}) and a random agent (r_{RAN}) as the upper and lower bounds, respectively. Thus, the normalized reward at step t, $\hat{r}_t$, is defined as:

$$\hat{r}_t = \frac{r_t - r_{RAN}}{r_{DQN} - r_{RAN}} \tag{6.1}$$

The "Normalized Reward" plot captures the evolution of this normalized reward $\hat{r}_t$ over all t.

The "Reward-per-Trace" plot further assesses efficiency by showing the ratio of normalized step reward and number of allocated traces, defined as:

$$r_t^{trace} = \frac{\hat{r}_t}{T_t} \tag{6.2}$$

where T_t is the total number of traces used on step t.

Finally, the "Cumulative Traces" plot tracks the cumulative number of MCTS search traces allocated over time. This plot shows data for AZ and UA_AZ. The QR_AZ trace usage is identical to that of AZ, and therefore, it is not separately shown in the plot.

Carpole plots' results correspond to 10 seeds, whereas MinAtar games' to 5. Lines represent the median of runs, while shaded areas are the standard error.

The DQN results are truncated for consistency, with full-run available in the Appendix A.1.

6.2.1. Evaluation

This section provides an evaluation of the experimental outcomes.

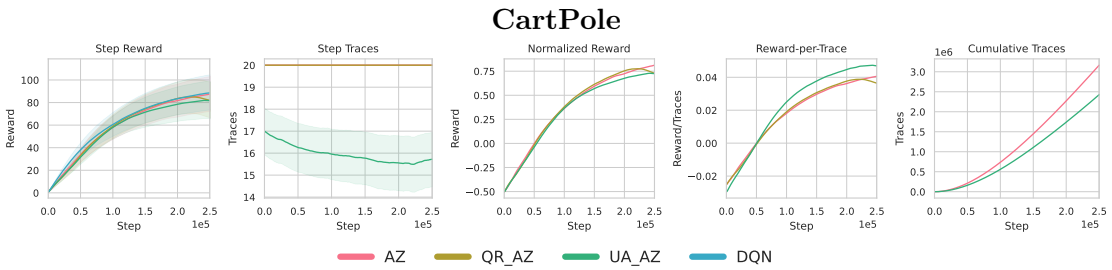

Figure 6.1.: CartPole environment results. Our approach, UA_AZ, achieves comparable performance to the baselines AZ and DQN while utilizing fewer traces, as shown in the "Cumulative traces" and "Step traces" subplots. Due to the simplicity of the environment, all agents converge to similar maximum rewards, as shown in "Normalized Reward." Overall, UA_AZ's efficient use of traces by UA_AZ is reflected in the "Reward-per-Trace" subplot.

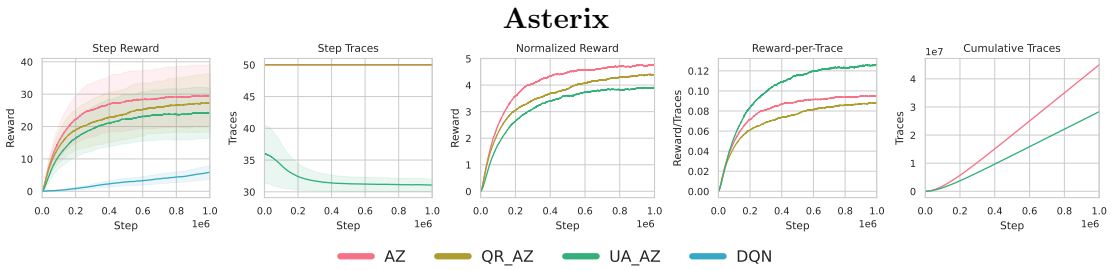

Figure 6.2.: Asterix environment results. The Step Rewards subplot highlights the faster convergence of AlphaZero-based agents compared to DQN. The Normalized Reward subplot demonstrates that these agents achieve at least four times the reward of DQN, even when the latter runs five times longer. While UA_AZ slightly underperforms the AZ baseline regarding Step Reward, the Reward-per-Trace subplot reveals its superior efficiency, confirming the effectiveness of dynamic trace allocation. Overall, the Cumulative Traces subplot highlights significant trace savings achieved by UA_AZ.

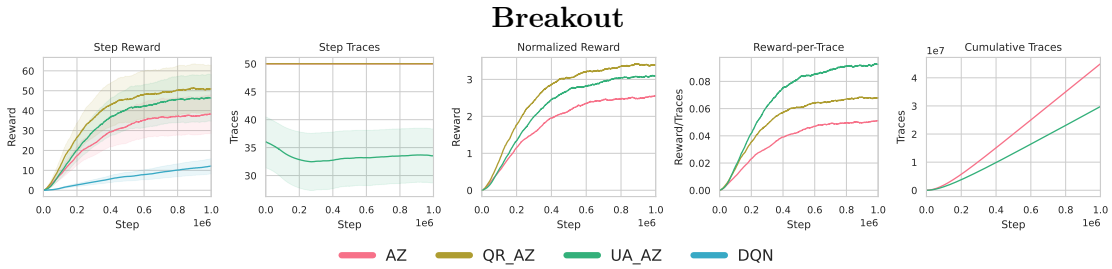

Figure 6.3.: Breakout environment results. The Step Rewards subplot demonstrates the faster convergence of AlphaZero-based agents compared to DQN. The Normalized Reward subplot shows these agents achieving at least 2.5 times the reward of DQN, even when the latter runs five times longer. While QR_AZ slightly outperforms UA_AZ in step rewards, UA_AZ exhibits greater trace efficiency, as highlighted in the Reward-per-Trace subplot. The Cumulative Traces subplot confirms significant trace savings overall.

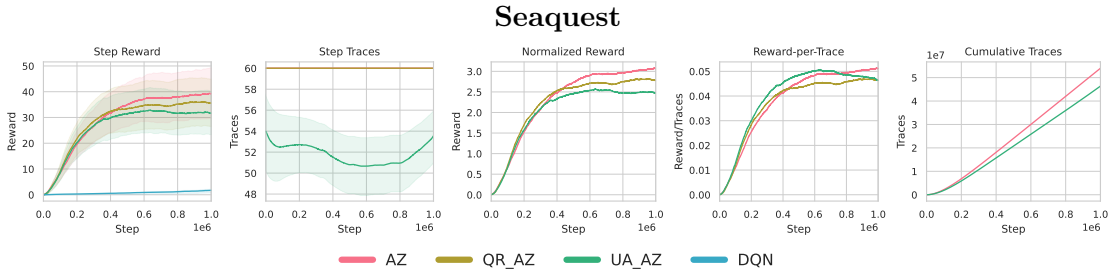

Figure 6.4.: Seaquest environment results. This environment is the most challenging due to its larger state space and sparse rewards. As a result, trace usage remained largely unchanged, reflecting this complexity. The Step Rewards subplot highlights the significantly faster convergence of AlphaZero-based agents compared to DQN. Additionally, the Normalized Reward subplot shows these agents achieving at least 2.5 times the reward of DQN, even when the latter runs five times longer. Overall, the Cumulative Traces subplot indicates small trace savings.

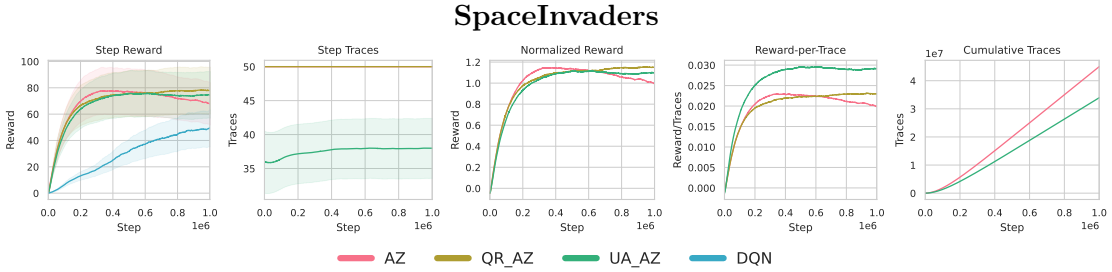

Figure 6.5.: SpaceInvaders environment results. The Step Rewards subplot highlights the faster convergence of AlphaZero-based agents compared to DQN. The Normalized Reward subplot shows that all agents achieve similar rewards to DQN, even when it runs five times longer. The Reward-per-Trace subplot demonstrates the efficient trace utilization of UA_AZ. Overall, the Cumulative Traces subplot indicates significant trace savings.

Trace efficiency The proposed dynamic trace allocation method improves the efficiency of trace usage across different environments. While the reduction of traces can lead to a performance decline (due to less search depth) when evaluated solely based on step rewards, most evaluated environments benefit from effective trace allocation. This trend is illustrated in Figures 6.1 - 6.5 Reward-per-Trace subplots, where the ratio of normalized step rewards to the number of traces is compared. Notably, the UA_AZ approach demonstrates more efficient use of traces than the base AlphaZero AZ.

A higher baseline number of traces (60) was used for the Seaquest environment, characterized by sparse rewards and more complex state space. As shown in Figure 6.4, Step Traces subplot, the UA_AZ allocation stabilized around 50, which was experimentally validated, as further reduction of traces in AZ leads to a significant performance drop.

Sample efficiency Figures 6.1 - 6.5 Normalized Reward subplot demonstrate that AlphaZero-based agents (AZ, QR_AZ, and UA_AZ) are more sample efficient than DQN, as evidenced by their higher normalized step rewards on a fraction of steps. Given the additional planning and complexity of DNN architecture in AZ-based approaches, this aligns with expectations.

Hyper-parameters importance Tuning revealed significant sensitivity to some key parameters. The number of quantiles is essential, as increasing tends to improve the informativeness of the prior and values outputs distributions, though at the cost of increased DNN complexity. Based on empirical results, we set 20 quantiles across all environments. Figure 6.6 compares performance across different numbers of quantiles in the Breakout environment.

In UA_AZ, where the basic DQN agent is trained, the epsilon parameter governing the epsilon-greedy exploration is critical to learning performance. We start with full exploration and gradually reduce epsilon to 0.01, shifting toward exploitation. Another

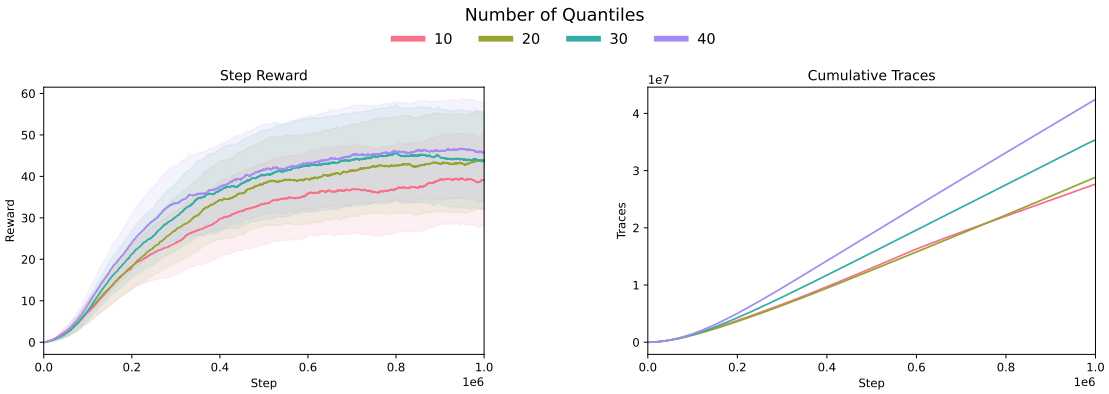

Figure 6.6.: Comparison of performance across different numbers of quantiles in the Breakout environment. Using 20 quantiles yields competitive step rewards while reducing the cumulative number of traces. All setups were run with 5 seeds and identical hyperparameters.

critical factor is the number of warm-up traces, which must be large enough for MCTS to explore the entire action space.

A fourth crucial hyperparameter is the frequency at which the target networks are updated. This applies to the DQN agent's target network in UA_AZ and the main target network used on QR_AZ and UA_AZ. Figure 6.7 highlights how different main target network update frequencies can affect the performance, even when all other parameters remain constant. Early in the training process, during shorter episodes, the main DNN is updated more frequently but with small gradient changes due to limited samples. As episodes lengthen, on later steps, updates occur less often with more substantial gradient adjustments. Consequently, in the early steps, the difference between the syncing frequency of 100 and 1000 is minimal. However, in later steps, less frequent syncs allow the target network to diverge more from the main DNN, which helps to make significant gradient updates.

Time comparison Figure 6.8 presents the average run-time in environments where our proposed agent equals or surpasses the baseline performance in terms of step reward. The QR_AZ agent exhibits significant computational overhead introduced by changes for DNN uncertainty estimation surpassing other run-times. In contrast, the proposed variant UA_AZ leverages these changes to allocate traces dynamically. In Breakout, UA_AZ is faster and outperforms other AZ-based baselines in terms of reward. In Space Invaders, it achieves similar rewards to the baselines while still producing considerable improvements in run-time. As expected, the DQN agent is significantly faster due to less computation, but it underperforms compared to others in terms of reward for the given number of steps.

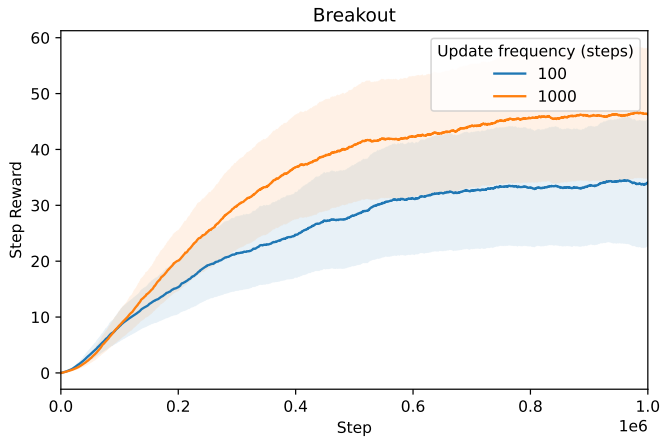

Figure 6.7.: Step rewards corresponding to two UA_AZ agents over Breakout environment, using update main target network frequency 100 and 1000 steps.

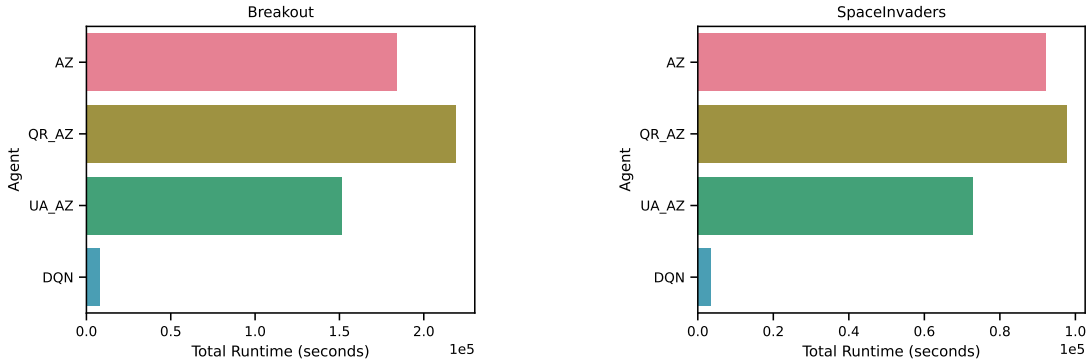

Figure 6.8.: Average total run-time for Breakout and Space Invaders over 1,000,000 steps and five seeds.

7. Discussion

This chapter provides a high-level evaluation of the experimental outcomes. Additionally, the key limitations encountered during the study are discussed, providing insight into constraints and directions for future improvements.

7.1. Evaluation of results

Based on the experiments, our approach, AZ_UA, has demonstrated effectiveness in trace usage. It dynamically adjusts the number of traces per search based on uncertainty, optimizing the depth of the search to balance thoroughness in uncertain regions and efficiency in predictable ones, without significantly affecting overall rewards. Although our approach increases the complexity of the AlphaZero DNN architecture and incorporates an additional DQN agent, it achieves a notable speed-up compared to the baseline.

By leveraging uncertainty as a criterion for planning, AZ_UA reduces computational costs, a critical consideration in reinforcement learning problems where planning is resource-intensive. Its ability to dynamically allocate traces based on available information highlights its potential for efficient resource utilization. This approach aligns with principles reminiscent of AlphaZero and dual-process theories of human cognition, which emphasize the strategic use of knowledge to streamline planning. Here, knowledge of the relationship between context (mainly uncertainty) and trace requirements effectively reduces the computational demands of the planning process.

Dynamic allocation method Before arriving at the proposed dynamic allocation based on a DQN agent, we explored framing the problem as a Contextual Multi-Armed Bandit (CMAB). Similar to the presented approach, we designed a context based on the tree's state, which the bandit uses to choose an arm (i.e., the number of traces to extend). We evaluated two algorithms, LinUCB [20] and NeuralUCB [39]. LinUCB offers an efficient closed-form computation of confidence intervals for the payoffs (or rewards) by learning an optimal vector, θ_a^*, which linearly relates the expected payoff $r_{t,a}$ and feature vector $x_{t,a}$, for each time step t. The algorithm calculates the UCB for each action a and selects the one with the highest UCB. This approach assumes a linear relation between the payoffs and the contexts.

On the other hand, NeuralUCB uses a DNN to estimate the rewards directly, offering more flexibility by not depending on the linearity assumption. However, it requires observing the context and reward for all actions at each time step t, which is unavailable on our setup. To work around this, we tried one-hot encoding the context for each action and assigning zero rewards for the non-current ones.

Both methods struggled with trace allocation. Even with extended contexts, including features such as advantage and total variance difference (TVD) in MTCS priors, results were inferior to the presented method.

7.2. Limitations

Informativeness of uncertainty methods Clements et al. [8] highlighted that epistemic uncertainty tends to increase as episodes advance when the agent faces new states, which aligns with our findings. This trend is counterintuitive, as a DNN is expected to improve over time, becoming increasingly robust to new states. Given that this metric influences trace allocation, this unexpected behavior might negatively impact the learning process. Moreover, this approach introduces computational overhead using more complex architectures that output distributions instead of single values, requiring two additional models for uncertainty estimation and introducing an extra post-processing step to fit the outputs to the AlphaZero loss function.

The MCTS uncertainty metric proposed by Moerland et al. [26] also has limitations, remaining static until a terminal state is reached. Consequently, this metric provides limited informativeness during the initial steps of an episode, which could lead to sub-optimal decisions.

Implementation The Gymnasium library does not offer an efficient method to clone the environments, which is essential for AlphaZero's planning phase, where isolated environment instances are required. We use Python's deep_copy function as a workaround, which incurs considerable overhead. In MinAtar environments, particularly, cloning accounts for approximately 20% of the total execution time.

Additionally, some experimental parameters were chosen empirically. MinAtar environments, for instance, do not have a fixed episode length. To standardize the experiments, we capped the episodes at step 1,200, after which a truncated flag indicates the episode's end. Similarly, the maximum number of traces was empirically chosen. Increasing these parameters could improve results by enabling the model to encounter new states and allowing further exploration, respectively.

We flatten the observation space to reduce training time further, avoiding convolutional layers in AlphaZero-based experiments. This optimization slightly improved runtime without compromising performance. However, this might not hold for more complex environments.

8. Conclusion

In conclusion, this thesis presents a novel approach for optimizing MCTS trace allocation in the AlphaZero algorithm by dynamically adjusting the budget based on DNN and MCTS uncertainties. We adapted a technique [8] to estimate epistemic uncertainty in DNNs by incorporating a DRL approach and extended MCTS nodes to track sub-trees depth, as on [27]. Trace allocation follows a two-phase process: initial warm-up traces are used to get insights, followed by a DQN agent that decides the number of additional traces based on estimated uncertainties and collected statistics after the first phase.

The results empirically confirm that our approach improves trace allocation efficiency, outperforming baselines in the reward-per-trace ratio in most experiments. However, the introduced computational overhead, particularly from the uncertainty estimation in DNN, leads to a slight increase in runtime in specific experiments.

8.1. Future works

Although the proposed method has demonstrated improvements in trace allocation efficiency, further refinement could address some limitations. Below, we outline three key directions.

Uncertainty estimation, especially on DNNs, is a broad and evolving field with diverse approaches. Given the significant overhead identified by the adapted method in our work, exploring alternatives may lead to more efficient solutions.

Incorporating uncertainties estimations on RL, such as information-directed exploration [8] [29], has demonstrated effective exploration-exploitation trade-offs. Extending our approach with techniques like these could enhance the overall learning process.

Finally, alternative dynamic trace allocation strategies should be considered. While our approach uses a warm-up stage, alternative approaches like early stopping traces or adjusting allocation frequency could reduce overhead. Additionally, exploring more complex decision-making frameworks, such as more complex CMAB, may be worthwhile.

Bibliography

[1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[2] Murat Seckin Ayhan and Philipp Berens. Test-time data augmentation for estimation of heteroscedastic aleatoric uncertainty in deep neural networks. In *Medical Imaging with Deep Learning*, 2022.

[3] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[4] Marc G. Bellemare, Will Dabney, and Mark Rowland. *Distributional Reinforcement Learning*. MIT Press, 2023. `http://www.distributional-rl.org`.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. doi: 10.1109/TCIAIG.2012.2186810.

[7] Nicolo Cesa-Bianchi and Gábor Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006.

[8] William R Clements, Bastien Van Delft, Benoît-Marie Robaglia, Reda Bahi Slaoui, and Sébastien Toth. Estimating risk and uncertainty in deep reinforcement learning. *arXiv preprint arXiv:1905.09638*, 2019.

[9] Will Dabney, Mark Rowland, Marc Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[10] Kahneman Daniel. *Thinking, fast and slow*. 2017.

[11] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Joelle Pineau, et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[12] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. A Survey of Uncertainty in Deep Neural Networks. *arXiv:2107.03342 [cs, stat]*, July 2021.

[13] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics: Methodology and distribution*, pages 492–518. Springer, 1992.

[14] Eyke Hüllermeier and Willem Waegeman. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine Learning*, 110: 457–506, 2021.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[16] Farnaz Kohankhaki. Monte carlo tree search and model uncertainty, 2022.

[17] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[18] Li-Cheng Lan, Ti-Rong Wu, I-Chen Wu, and Cho-Jui Hsieh. Learning to stop: Dynamic simulation monte-carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 259–267, 2021.

[19] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[20] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[21] Yuxi Li. Deep reinforcement learning: An overview, 2018.

[22] Owen Lockwood and Mei Si. A review of uncertainty for deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, pages 155–162, 2022.

[23] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[26] Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. A0c: Alpha zero in continuous action space. *arXiv preprint arXiv:1805.09613*, 2018.

[27] Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. The second type of uncertainty in monte carlo tree search. *arXiv preprint arXiv:2005.09645*, 2020.

[28] Ted Moskovitz, Jack Parker-Holder, Aldo Pacchiano, Michael Arbel, and Michael I. Jordan. Tactical Optimism and Pessimism for Deep Reinforcement Learning. *arXiv:2102.03765 [cs]*, January 2022.

[29] Nikolay Nikolov, Johannes Kirschner, Felix Berkenkamp, and Andreas Krause. Information-directed exploration for deep reinforcement learning. *arXiv preprint arXiv:1812.07544*, 2018.

[30] Tim Pearce, Felix Leibfried, and Alexandra Brintrup. Uncertainty in neural networks: Approximately bayesian ensembling. In *International conference on artificial intelligence and statistics*, pages 234–244. PMLR, 2020.

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

[32] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature16961.

[33] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[34] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362

(6419):1140–1144, December 2018. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.aar6404.

[35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[36] 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, March 2023. URL `https://zenodo.org/record/8127025`.

[37] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[38] Kenny Young and Tian Tian. Minatar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments. *arXiv preprint arXiv:1903.03176*, 2019.

[39] Dongruo Zhou, Lihong Li, and Quanquan Gu. Neuralucb: Contextual bandits with neural network-based exploration.

A. Appendix

A.1. DQN results

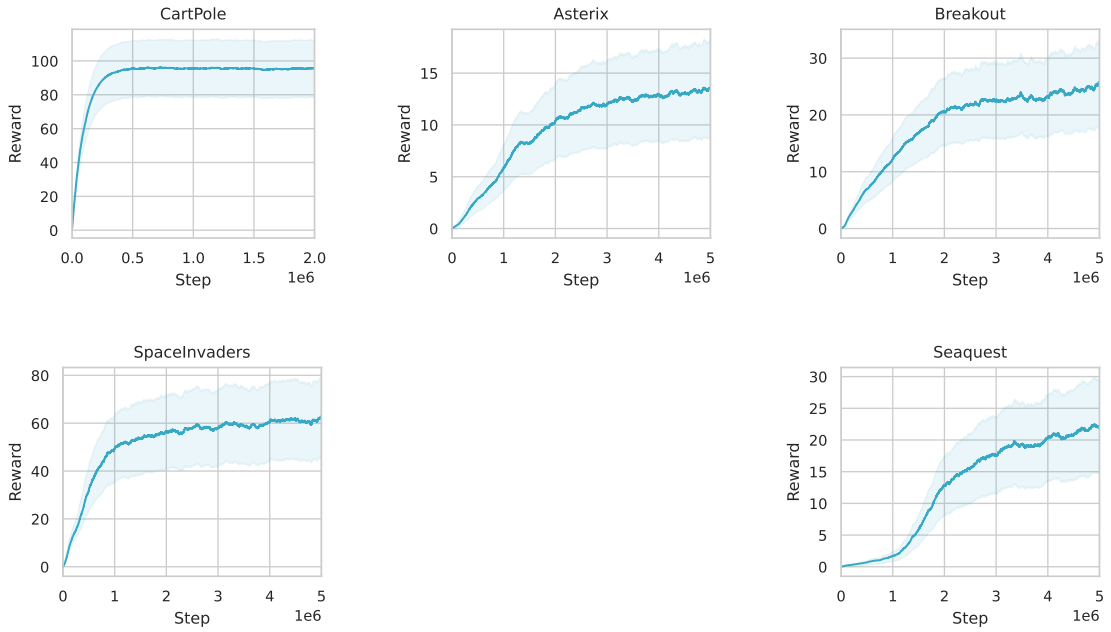

Figure A.1.: Full runs of DQN agent. Presented first 2,000,000 steps on CartPole environment, and 5,000,000 steps on MinAtar ones

A.2. Hyperparameters

This section presents the hyperparameters used in the experiments. Table A.1 provides their description, while Table A.2 their fine-tuned values.

A.3. Environments details

This section details the environment specifications, including action and observation spaces, rewards, and termination criteria.

CartPole

A pole on a frictionless cart acts as a pendulum, which the agent balances by applying stepwise forces, with speed and resistance affected by the pole's angle and center of gravity. Towers et al. [36] implementation in Gymnasium is described below.

Action space: 0,1 representing pushing the cart to the left or right.

Observation space: consists of four values: cart position, cart velocity, pole angle, and pole angular velocity.

Rewards: +1 point is given for each step made until termination (inclusive). This work uses the library's v0 version of Cartpole, where the max reward (i.e., max episode length) is 200.

Termination: episode ends when any of the following cases: pole angle is greater than +/- 12 degrees, cart position reaches the edge limit (+/- 2.4) or episode length reaches the threshold.

Asterix

The player navigates in four cardinal directions to collect treasures and avoid increasingly frequent and faster-spawning enemies from the screen edges.

Action space: {0,1,2,4,5} representing none, left, up, right, and down.

Observation space: matrix of booleans of shape 10x10x4. The (4) channels are player, enemy, trail, and gold.

Rewards: +1 point is given for picking up treasure.

Termination: episode ends when the player makes contact with an enemy.

Breakout

The player controls a paddle to bounce a ball diagonally, breaking rows of bricks at the top and reflecting off walls on the sides.

Action space: {0,1,2} representing none, left, and right.

Observation space: matrix of booleans of shape 10x10x4. The (4) channels are paddle, ball, trail, and brick.

Rewards: +1 point is given for each broken brick by the ball.

Hyperparameter	**Description**
Model Architecture	
Hidden layers	Number of neurons on hidden layers
Dropout	Dropout ratio after last hidden layer
Optimizer	
Learning rate	Learning rate
Type	Optimizer algorithm
Adam epsilon	Adam optimizer parameter to prevent division by zero
Buffer	
Max size	Main buffer maximum capacity. Buffer overwrites oldest entries once full.
Batch size	Sampling batch size
Ensemble model (QR_AZ / UA_AZ)	
Update target frequency	Number of steps for syncing target network parameters weights with main networks.
Update tau τ_{pk}	Polyak weight update parameter that adjusts the blending of target and main network weights during the update of the target network.
κ	Huber loss parameter that states the interval $[-\kappa, \kappa]$ where square loss is used. Eq. 2.14.
β_Q	Quantile regression loss scale. Scales the uncertainty estimate related losses (quantile regression and anchor loss). Eq. 4.7.
α_{AZ}	Weighting parameter to combine AZ and additional losses. Eq. 4.7.
Num. Quantiles	Number of quantiles to estimate.
Alpha entropy	Entropy bonus parameter for AlphaZero loss
α_Q	Quantile regression loss weight. It's the weighting factor for combining QRL from value and pi heads. Eq.4.6.
MCTS	
C_{uct}	Exploration rate C_{uct}
Epsilon greedy	Epsilon greedy parameter that determines the probability to choose random actions
Gamma γ_{mcts}	MCTS backpropagation discount factor.
Normalization temperature	Normalization temperature for calculating action priors from MCTS tree.
Max traces	Maximum number of traces
DQN Agent (UA_AZ) and DQN	
Hidden dims	Number of neurons on hidden layers of DQN/DQN Agent's DNN
Possible extra traces	Set of valid number of extra traces to allocate
Warm up traces	Fixed number of traces to run to get an insight of performance
Train frequency	Number of steps for training DQN Agent's DNN
Warm up exploration (steps)	Number of steps whiting learning rate linearly decays to Final exploration rate
Update target frequency	Number of steps for syncing target network parameters weights with training networks.
Final exploration rate	Learning rate value to use after warm-up exploration phase
Loss function	Loss function used for training DNN
Buffer max size	DQN Agent's buffer maximum capacity.
Buffer batch size	Sampling batch size
Gamma	Discount factor for TD update

Table A.1.: Hyperparameters description.

Hyperparameter	MinAtar	CartPole
Model Architecture		
Hidden layers	[128,256]	[32,64,128](Q/U); [64,64,64](AZ)
Dropout	0.2 (Q/U); None (AZ)	None
Optimizer		
Learning rate	1e-3	1e-3 (Q/U); 1e-2 (AZ)
Type	Adam (Q/U); RMS (AZ)	Adam (Q/U); RMS (AZ)
Adam epsilon	1e-8	1e-8
Buffer		
Max size	10,000 (Q/U); 5,000 (AZ)	2,000
Batch size	32	32
Ensemble model (Q/U)		
Update target frequency	1,000	1,000
Update tau τ_{pk}	0.8	0.8
κ	10	10
β_Q	1	1
α_{AZ}	0.2	0.5
Num. Quantiles	20	20
Alpha entropy	0.5	0.5
α_Q	0.8	0.8
MCTS		
C_{UCT}	1.5	1.5 (Q/U); 1.6 (AZ)
Epsilon greedy	0	0.1
Gamma γ_{mcts}	1	1
Normalization temperature	2.5	1.0
Max traces	Sea.: 60; others: 50	20
DQN Agent (UA_AZ)		
Hidden dims	[64,128]	[32,64]
Possible extra traces	Sea.:[10,20,30]; others: [10,20,30,40]	[5,10,15]
Warm up traces	Sea.:30; others: 10	5
Train frequency	1	1
Warm up exploration (steps)	100,000	10,000
Update target frequency	5,000	500
Final exploration rate	0.01	0.01
Loss function	Huber	Huber
Buffer max size	2,000	2,000
Buffer batch size	32	32
Gamma	0.9	0.9
DQN		
Hidden dims	Conv2d out 16, linear layer 128	[100,100] (linear layers)
Train frequency	1	1
Warm up exploration steps	1,000	2,000
Update target frequency	1,000	50
Final exploration rate	0.03	0.01
Loss function	MSE	MSE
Buffer max size	100,000	50,000
Buffer batch size	32	32
Gamma	0.99	0.99

Table A.2.: Experiments hyperparams. If not specified, they apply to all setups. "Q/U" refers to QR_AZ and UA_AZ.

Termination: episode ends when the ball hits the screen bottom.

Seaquest

The agent navigates a submarine to avoid or attack enemies, collect divers, and manage oxygen levels, which can be replenished by surfacing.

Action space: {0,1,2,3,4,5} representing none, left, up, right, down, and fire.

Observation space: matrix of booleans of shape 10x10x10. The (10) channels are sub_front, sub_back, friendly_bullet, trail, enemy_bullet, enemy_fish, enemy_sub, oxygen_guage, diver_guage, and diver.

Rewards: +1 point is given when an enemy is shot. A reward equivalent to the number of oxygen bars is given when surfacing full of divers.

Termination: episode ends when an enemy or bullet hits the submarine, oxygen runs out, or the submarine surfaces with no divers onboard.

Space Invaders

The player controls a cannon to shoot upward at a moving cluster of aliens that reverses direction at screen edges and fires back; as aliens are cleared, they speed up, and a new, faster cluster appears once all are eliminated.

Action space: {0,1,2,3} representing none, left, right, and fire.

Observation space: matrix of booleans of shape 10x10x6. The (6) channels are cannon, aliens position and direction (3), and bullets position (2).

Rewards: +1 point is given when an alien is shot.

Termination: episode ends when an alien or bullet hits the cannon.