Monte Carlo Tree Search in the Presence of Model Uncertainty

by

Kiarash Aghakasiri

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Monte Carlo Tree Search (MCTS) is a extremely successful search-based framework for decision making. With an accurate simulator of the environment's dynamics, it can achieve great performance in many games and non-games applications. However, without a perfect simulator, the performance degradation is so high that it can make the framework almost useless. Therefore, we propose two methods to improve the performance of MCTS in such a scenario: Deep Q-Network MCTS (DQMCTS) and Uncertainty Adapted MCTS (UAMCTS). In the former, we use the model-free algorithm DQN to evaluate the leaf nodes in the search tree. Although this approach shows promising improvement over baseline MCTS, our results show that there is still more room for improvement. In UAMCTS, we take a more fundamental approach and change the behavior of MCTS's components to directly take the model incorrectness into account. Our results show that with an accurate measure of model incorrectness, UAMCTS can achieve the performance of MCTS with a perfect simulator in some cases. Even with a poor measure of model error, UAMCTS can still outperform plain MCTS with an imperfect simulator.

Preface

Part of this thesis has been submitted to IJCAI-ECAI 2022 co-authored with Farnaz Kohankhaki, Martin Müller, and Ting-Han Wei. The proposed methods were developed in close collaboration with Farnaz Kohankhaki.

DQ-Expansion, UA-Selection, and UA-Backpropagation were implemented by myself. The other methods: DQ-Simulation, UA-Expansion, and UA-Simulation, were developed by Farnaz Kohankhaki. We discussed our results and combined these methods and created the final agents UA-MCTS and DQ-MCTS.

To my parents and my sister

For being the most supportive family that I could ever imagine

Most of the good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.

– Linus Torvalds, Software Engineer, Inventor of the Linux operating system

Acknowledgements

First and foremost, I want to thank my supervisor, Martin Müller, for giving me the opportunity to work with him. I remember whenever I was worried or stuck in a problem he kindly guided me through it. His words were always calm and encouraging and taught me how to stay positive and patient in research and life. I can neither thank him enough nor describe his guidance through all stages of my studies.

I would also like to thank Ting for his accessible help and support, Chao for his constructive comments, and Huawei Edmonton for their funding without which I could not finish my thesis. I would like to acknowledge my committee members, Martha White and Ryan Hayward, for taking time out of their busy schedules and reading my thesis.

Many thanks to my dear friends, Bedir, Farnaz, Shivam, Hamza, Mohsen, Aidan, Nikoo, Mahtab, and Sina who helped me pass through this tough pandemic situation and get where I am right now.

Last but not least, I wanted to thank three people for always being there for me: My mother, whose hair turned white so that I could light the way, my father, who burned with love to warm my heart and enlighten my way, my sister, who is also my best friend and I could not imagine my life without her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Monte Carlo Tree Search (MCTS) framework [5] approaches sequential decision-making problems by selective lookahead search. It manages the balance of exploration and exploitation with techniques such as UCT [21]. A well-known combination of MCTS with machine learning is the famous AlphaGo program [39]. MCTS has also shown great results in other multiplayer games such as Chess [13], Othello [31], Shogi [36], Blockus Duo [37], and Hex [2, 10]. The method has also been used for single player games such as Sudoku [8] and Solitaire [7] and real time games such as Ms. Pac-Man [48, 49]. MCTS has shown further great successes in non-game applications such as security systems [47], mixed integer programming [34], scheduling [29], and physics simulations [27].

In all these applications, a perfect simulation model of the problem domain is available, in which search steps can be efficiently performed. However, in many practical applications, only an imperfect model is available to the agent. Yet such a model can still be useful. The main goal of this thesis is to improve MCTS for this setting.

Model uncertainty or model error has been studied a lot in the field of deep learning. There is much research on how to capture model uncertainty using Bayesian techniques [17]. Examples are model ensembles [24], Monte-Carlo dropout [15], and heteroscedastic regression [32]. Another research area that utilizes imperfect models is model-based reinforcement learning (MBRL).

In MBRL, the agent can build its own model through interactions with

the environment, or it can make use of a given model. The model, when used for lookahead search, can either be for planning or for producing more accurate training targets [41]. It can also be used to generate simulated training samples for better sample efficiency [44]. If the model is learned, it may be inaccurate for many reasons, including stochasticity of the environment, insufficient training, insufficient model capacity, and non-stationary environments. Consequently, there is a rich body of research on *uncertainty* in MBRL.

While previous approaches to using search with imperfect models exist [50, 51], surprisingly, to the best of our knowledge, there is no prior work that directly adapts MCTS to deal with model uncertainty. In this thesis, we propose two approaches to this problem.

1.1 Contributions

Here is the list of our contributions:

- We empirically show a significant drop in MCTS performance under model error. (Chapter 3)
- We introduce our first adaptation, Deep Q-network MCTS (DQMCTS), which uses a learned DQN value function as a heuristic to deal with model uncertainty. (Chapter 4)
- We introduce our second adaptation, Uncertainty-Adapted MCTS (UAMCTS), which uses a method to capture the uncertainty and then diverts the search from the uncertain parts. (Chapter 5)
- We slightly change three deterministic MinAtar environments [54] to study search with a corrupt model (Chapter 3) and compare the performance of our proposed methods, DQMCTS and UAMCTS with two MCTS baselines. (Sections 4.3, 5.5)

Chapter 2

Background

In this chapter we explain the background needed to understand our approach.

2.1 Markov Decision Processes (MDP)

MDPs are a formalization of online sequential decision making in which making a decision affects future situations [44]. In this work we focus on *deterministic* MDPs which can be be specified by a tuple $\langle \mathcal{S}, \mathcal{A}, M, R, \gamma \rangle$. $\mathcal{S}$ and $\mathcal{A}$ are finite state and action spaces respectively. R is a deterministic function mapping a pair (s, a) with $s \in \mathcal{S}, a \in \mathcal{A}$ to a scalar reward. At each time step t, the agent observes a state $s_t \in \mathcal{S}$ and a scalar reward r_t from the environment. Then at the same time step the agent chooses an action $a_t \in \mathcal{A}$ to take, which affects the next time step's state, $s_{t+1} \in \mathcal{S}$ and reward, r_{t+1}. The agent follows a trajectory $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots$

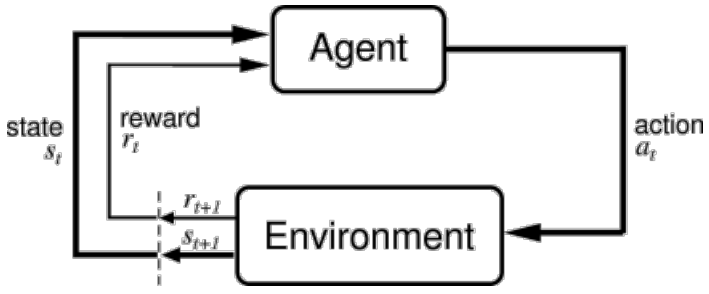

Figure 2.1: Agent and environment interaction loop [44]

The agent and environment interaction loop are shown in Figure 2.1. M is the transition dynamics, a deterministic function $\mathcal{S} \times \mathcal{A} \to \mathcal{S}$ showing the

next state given a state and action pair, $M(s_t, a_t) = s_{t+1}$. $0 \leq \gamma \leq 1$ is the discount rate which determines the importance of the future rewards. The goal of an agent acting in an MDP is to maximise its reward. Reinforcement Learning and Search methods are two closely related approaches to solve an MDP.

2.2 Reinforcement Learning (RL)

In Reinforcement Learning, the agent interacts with the environment to learn how to act. The environment can be formulated by an MDP. The goal of an agent is to maximize the sum of rewards G_t, which we call *return*, described in Equation 2.1.

$$G_t \doteq \sum_{k=0}^{\infty} r_{t+k+1} \tag{2.1}$$

In episodic tasks, the environment terminates after a finite number of time steps so the return is bounded. In continuing tasks, the environment doesn't terminate. With an infinite number of time steps, the sum in Equation 2.1 is not bounded. Thus, we should use the discounted sum of rewards, shown in Equation 2.2 ($0 \leq \gamma < 1$).

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

In the following subsections we introduce the concepts of policy and value function and two types of general RL algorithms, model-free and model-based methods.

2.2.1 Policy and Value Function

A Policy determines the behaviour of the agent and the corresponding value function estimates the return following the policy. A *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a mapping from states to actions. $\pi(a|s)$ is the probability of choosing action a in state s when following π.

The value function of state s under policy π, $v_\pi(s)$, is the expected return from state s when following policy π. This is called the *state value function* shown in Equation 2.3.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s\right] \tag{2.3}$$

Similarly, a *state-action value function*, $q_\pi(s, a)$, is the expected return of taking action a in state s and following policy π from there. (Equation 2.4).

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s, A_t = a\right] \tag{2.4}$$

To calculate the value function for a policy π we use a fundamental recursive relation called the *Bellman* equation shown in Equation 2.5 [44].

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\Big[R(s, a) + \gamma v_\pi\big(M(s, a)\big)\Big], \text{ for all } s \in \mathcal{S} \tag{2.5}$$

We can write a similar equation for state-action value functions (Equation 2.6).

$$q_\pi(s, a) = R(s, a) + \gamma \sum_{a' \in \mathcal{A}} \pi\big(a'|M(s, a)\big) q_\pi\big(M(s, a), a'\big), \text{ for all } s \in \mathcal{S}, a \in \mathcal{A} \tag{2.6}$$

Next we introduce optimal policies and value functions. A policy π is better than or equal to policy π' if and only if for all states $s \in \mathcal{S}$, $v_\pi(s) \geq v_{\pi'}(s)$. A policy for which its value function is greater than or equal to all other policies for all states is called an *optimal policy* π^* [44]. The value function corresponding to an optimal policy is called the *optimal value function* (Equation 2.7). There is at least one such policy. The optimal value function is unique, but more than one optimal policy might exist for an MDP.

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s) \\ q_*(s, a) &= \max_\pi q_\pi(s, a) \end{aligned} \tag{2.7}$$

If the policy in the Bellman equation is optimal, we obtain the *Bellman optimality equations* (Equation 2.8). They can be used to directly calculate the optimal value functions [44].

$$
\begin{aligned}
v_*(s) &= \max_a \Big[R(s,a) + \gamma v_*\big(M(s,a)\big) \Big], \text{ for all } s \in \mathcal{S} \\
q_*(s,a) &= R(s,a) + \gamma \max_{a'} q_*\big(M(s,a), a'\big), \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}
\end{aligned} \tag{2.8}
$$

2.2.2 Model-Free RL

In model-free methods, the agent does not have any access to the dynamics of the environment M. The only way for the agent to learn is through interactions with the environment. One group of learning methods is called Temporal-Difference (TD) learning [44]. TD methods use the Bellman equations but instead of having access to the summation over all states, they use individual transition samples to incrementally update the value function.

One famous TD method that uses Bellman optimality equations (Equations 2.7 and 2.8) to calculate the optimal policy is *Q-learning* [52]. A Q-learning agent starts with a set of random values for all state-actions and then updates the state-action value function with each transition sample (s, a, r, s') using Equation 2.9. To interact with the environment the agent chooses actions according to an ϵ-greedy policy coming from $q(s,a)$. A greedy policy derived from a value function always chooses an action with the highest value on the given state. An ϵ-greedy policy chooses a random action with probability ϵ and a greedy action with probability $1-\epsilon$. Usually, the value of ϵ is high initially, which encourages exploration in the early episodes, and over time it decreases to exploit the better states.

$$
q(s,a) \leftarrow q(s,a) + \alpha[r + \gamma \max_{a'} q(s',a') - q(s,a)] \tag{2.9}
$$

In Q-learning the agent uses a table of $q(s,a)$ values to learn and update the value function. Although using a table results in an accurate value function there are two downsides to it. First, in real applications, the state space S can be immensely big, making it impossible to fit all the values in a table. Secondly,

a large state space S requires generalizing between similar states because the agent might not be able to visit all the states in a reasonable time. Hence, a more practical idea it to use function approximation methods to estimate the value function. *Deep Q-Network (DQN)* is the function approximation version of Q-learning [28]. A DQN agent approximates the value function with a deep neural network and trains the neural network to minimize the TD loss (Equation 2.10) using an optimization technique such as gradient descent.

$$TDLoss(s, a, s', r) \doteq \big(r + \gamma \max_{a'} q_t(s', a') - q(s, a)\big)^2 \tag{2.10}$$

The target used in TD loss is not independent of the learning network which makes DQN a *semi-gradient* method. To make the method more similar to true gradient methods (target independent from the learner), DQN uses a target network q_t which is being updated with the learning network q after every fixed number of interactions f_t. DQN stores the environment interactions in a transition buffer B and at each training step the q network is being trained on a batch of data from B.

Algorithm 1 shows the pseudo code for DQN. $\pi_{q,\epsilon}(s)$ is an ϵ-greedy policy using values from network $q(s, a)$ which returns an action a for state s. The environment has two functions: The *Start* function initializes the environment and returns the initial state, and the *Step* function gets an action from the agent's policy and returns the next state s' and reward r. B_s is the size of the mini-batch used at each training step. The function $Sample(B, B_s)$ returns a random mini-batch of size B_s from B . λ is the step size for training the q network. N_F is the number of frames or number of interactions with the environment.

2.2.3 Model-Based RL

In model-based reinforcement learning (MBRL) methods, the agent uses a model $\hat{M}$ of the environment's dynamics. Given this model, the agent does not need to only rely on direct environment interactions to learn a good policy. A model can be used in dynamic programming methods or real-time search

Algorithm 1 DQN Algorithm, from [28]

Parameters: ϵ for the ϵ-greedy policy. f_t is the update frequency of q_t. B_s is the batch size. λ is the step size for q. N_F is the number of frames for training.

Initialize Buffer: $B \leftarrow \{\}$

Initialize Interaction Counter: $i \leftarrow 0$

Randomly Initialize θ

$\theta \leftarrow \theta_t$

$s \leftarrow$ Env.Start()

for N_F steps **do**

$a \leftarrow \pi_{q,\epsilon}(s)$

$r, s' \leftarrow$ Env.Step(a)

Store transition (s, a, r, s') in buffer B

$(S, A, R, S') \leftarrow Sample(B, B_s)$

if S' is terminal state **then**

$y = R$

else

$y = R + \gamma \max_{A'} q_t(S', A' \mid \theta_t)$

$loss \leftarrow \big(y - q(S, A \mid \theta)\big)^2$

Gradient Descent Update: $\theta \leftarrow \theta - \lambda . \dfrac{\partial loss}{\partial \theta}$

$i \leftarrow i + 1$

Update Target:

if $i = f_t$ **then**

$\theta_t \leftarrow \theta$

$i \leftarrow 0$

methods. In this thesis, we use the Monte Carlo Tree Search algorithm which we explain in the next section. In this section, we discuss MBRL in general and the use of different types of models.

If $\hat{M} = M$, the agent has access to the true underlying dynamics of the environment. In such cases, the agent can achieve extraordinary performances such as *AlphaGo* [39]. In many cases, the agent does not have access to the M function. In these situations the agent either has access to an imperfect model ($\hat{M} \neq M$) or it has no initial knowledge of the dynamics and has to learn a model from scratch. One simple model-based method uses experience replay such as in the *Dyna* structure [43]. A Dyna agent stores trajectories of real experience in a buffer and later retrains its policy using the stored transitions. Another approach to create a model is using parametric models such as neural networks. Hasselt et al. [16] compare the performance of experience replay and parametric models on Atari games [4].

There are many ways to train a model. We briefly explain some of them here. *Transition models* are most commonly used in MBRL. Such models approximate the dynamics M of the environment: given a state and an action they predict the next state. They can be used to generate "artificial" transitions for model-free updates [14], or to simulate rollouts for heuristic search methods [11]. In a stochastic environment, transition models can be further classified into expectation, sample and distribution models. *Distribution models* return a distribution over the next state's feature vector, *sample models* return a sample of the next state's feature vector, and *expectation models* return the expected feature vector of the next state given a state and an action. Sutton et al. [45] compare expectation and distribution models when using linear function approximation for the value function.

Another use of models are *backward transition models*. Such models take a state and an action as input and predict what the previous state would be if the agent took the given action and ended up in the given state. Chelu et al. [12] investigate the use of backward models for credit assignment.

The last type of models that we explain briefly are abstract models. They first transfer the state into an abstract latent space and then predict values,

instant rewards, or next states from the latent space [38, 40].

In this work, we focus on forward transition models which we explain in more detail in Section 2.3.

2.3 Model Uncertainty

Learning the forward transition model can be expressed as a regression task. An agent can use any regression method to train the model. One easy way is to train a neural network using the least squared error as the loss function. Assume $\hat{M}_\theta(s, a)$ is the model's prediction given s and a, and θ represents model parameters which are the weights of a neural network. Given a buffer of transitions B, the goal is to learn parameters θ that minimize the loss

$$L(\theta) = \sum_{s,a,s' \in B} (\hat{M}_\theta(s, a) - s')^2. \tag{2.11}$$

When the agent uses a learned model, uncertainties in the model's prediction may be due to three different factors [1]. The first is the stochasticity of the environment. Since we are learning the expectation over the next state's feature vector, if the environment is stochastic there is an inevitable uncertainty in the model's predictions. This type of uncertainty comes from the nature of the environment and is irreducible. Our deterministic environments does not have this type of uncertainty.

Another type of uncertainty is caused by insufficient capacity in the model's structure, which makes it unable to learn the true dynamics in principle. This type of uncertainty can be reduced by using a more powerful model, such as a deeper or larger network, or a more expressive activation function.

The last type of uncertainty comes from insufficient data coverage or training. Training a model takes both time and data. If the data does not represent the whole state space or if the model is not trained sufficiently with the data, it leads to an incorrect model. This type of uncertainty can be reduced by gathering more comprehensive data and increasing the training time.

In the rest of this section we discuss a few related works on model uncertainty.

CMAX and *CMAX++* are search algorithms specifically designed to deal with uncertainty [50, 51]. They work by deleting the imperfect parts of the model from search completely. To find a solution, there needs to be at least one viable path to a goal which does not involve states with uncertainty. As in our work, such states are identified by comparing against the real environment during interactions.

Many techniques quantify and use uncertainty in the context of MBRL. Lütjens et al. [26] capture uncertainty using ensembles of LSTM (a type of recurrent neural network) and Monte Carlo dropout, and change the behaviour of their agent to act more cautiously in the uncertain parts by introducing a cost function for model predictive control (MPC).

Multiple previous approaches do use uncertainty, but not as a component of an explicit search. Selective MVE [1], AdaMVE [53], and STEVE [6] modify the model value expansion (MVE), a model based algorithm based on model rollouts, by taking model uncertainty into consideration and giving less weight to uncertain rollouts. Jafferjee et al. [18] investigate the effect of model updates in both forward and backward directions with an imperfect model. Lai et al. [23] use forward and backward models in model-based policy optimization [19], a model based actor-critic method, in order to reduce accumulative model error while maintaining a similar update depth. Talvitie [46] designs a way to learn the model which reduces accumulative model error in deep lookahead.

2.4 Monte Carlo Tree Search

Beside RL, another way to find a solution in an MDP is to use Monte Carlo Tree Search (MCTS). MCTS is a search method which finds a desirable policy by building a search tree and using random sampling [5]. MCTS has been used in many successful works such as mastering the games of Go [39, 40], Hex [2, 10, 35], Othello [33], and Chinese Checkers [30]. It has been used in non-game applications such as Function Approximation [25], Physics Simulations [27], Mixed Integer Programming [34], and Mathematical Expression Generation [9].

The MCTS process builds a tree selectively and incrementally. In each MCTS iteration, a tree policy chooses which leaf node of the tree needs to be visited next. The goal of the tree policy is to balance between exploration of the less visited nodes and exploitation of the higher value nodes. The idea is to expand the nodes that are more promising deeply, but also explore other nodes to reduce their uncertainty. After reaching a leaf node, the search expands the selected node or runs simulations from the selected node. The first time we visit the selected node, one or more simulations are run to evaluate its value using the rollout policy. The simplest rollout policy is uniform random. If the selected node has been visited before, we expand it and add its children to the tree, then run one or more simulations from one of the children. New evaluations are backpropagated to all their ancestor nodes to update their values and visit counts. Algorithm 2 shows the pseudo code for an MCTS agent.

Algorithm 2 MCTS Framework

function MCTS(s_0)
    create a root node v_0 with state s_0
    for N_I **do**
        $v_s \leftarrow$ SELECT(v_0)
        if $N(v_s) > 0$ **then**
            $v_s \leftarrow$ EXPAND(v_s)
        $value \leftarrow$ SIMULATE($S(v_s)$)
        BACKPROPAGATE(v_s, $value$)
    $v_{best} \leftarrow$ choose the most visited child of v_0
    return $action(v_{best})$

After a fixed number of iterations N_I, the agent chooses an action at the root leading to the child with the highest number of visits. Sections 2.4.1-2.4.4 explain these four components in more detail.

2.4.1 MCTS Selection

The *Select* function starts from the root of the tree and repeatedly chooses a child node with best UCT value until it reaches a leaf node. It then returns that leaf node. For node v, $Q(v)$ is the sum of rewards observed from v and $N(v)$ is the number of times v has been visited. $Par(v)$ is the parent of node v for which the best child is selected. The UCT value is defined as follows:

$$UCT(v) = \frac{Q(v)}{N(v)} + c\sqrt{\frac{\ln\left(N(Par(v))\right)}{N(v)}}$$

The first term is the exploitation term and the second term is the exploration term. c is the exploration constant. Figure 2.2 shows an example of choosing between two child nodes with different values and number of visits. A greedy policy would explore the upper node with its higher value, but UCT chooses the lower value node due to its low number of visits.

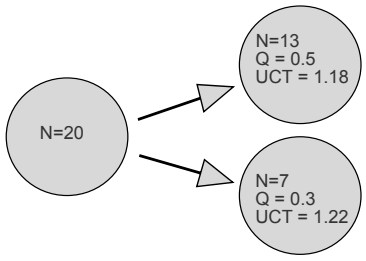

Figure 2.2: An example for selecting a child node in MCTS Selection ($c = \sqrt{2}$).

Algorithm 3 shows the pseudo code for the select function.

Algorithm 3 Selection Algorithm

function SELECT(v)

while v is expanded **do**

$$v \leftarrow \underset{v_i \in Ch(v)}{\operatorname{argmax}} \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\ln N(v)}{N(v_i)}}.$$

return v

2.4.2 MCTS Expansion

The *Expand* function adds all children of a node to the tree using the model of the environment $\hat{M}$. Expansion is responsible for growing the search tree. $S(v)$ is the state corresponding to node v and $R(v)$ is the reward when entering this node. Algorithm 4 shows the pseudo code for the expand function.

Algorithm 4 Expansion Algorithm

function EXPAND(v)

  for $a_i \in \mathcal{A}$ **do**

    $s_i, r_i \leftarrow \hat{M}(S(v), a_i)$

    create a node v_i with state s_i and reward r_i

    $N(v_i) \leftarrow 0$

    $Q(v_i) \leftarrow 0$

  return a random child of v

2.4.3 MCTS Simulation

When the search reaches a leaf node for the first time, it needs to estimate its value. The simulate function estimates a leaf node's value by doing N_S random rollouts ($N_S \geq 1$) from that node until a depth D_S, and returns the average of the rollout estimates. Algorithm 5 shows the pseudo code for simulate.

2.4.4 MCTS Backpropagation

This last step of the search loop updates the values and visit counts of nodes along the search path. The *backpropagate* function starts from a leaf node and updates the values and visit counts of its ancestors until it reaches the root of the tree. Let $Par(v)$ be the parent of node v. We define $Par(root) \doteq NULL$. The pseudo code for the backpropagation process is shown in Algorithm 6.

Algorithm 5 Simulation Algorithm

function SIMULATE($s, depth$)

  for $i \leftarrow 1$ to N_S **do**

    $g_i \leftarrow$ ROLLOUT(s)

    $\alpha_i \leftarrow 1/\mathrm{N}_S$

  return $\sum_{i=1}^{N_S} \alpha_i \cdot g_i$

function ROLLOUT(s)

  $count \leftarrow 0$

  $rewards \leftarrow 0$

  $discount \leftarrow 1$

  while s is not terminal and $count < D_S$ **do**

    choose a random action a from $\mathcal{A}$

    $s, r \leftarrow \hat{M}(s, a)$

    $count \leftarrow count + 1$

    $rewards \leftarrow rewards + discount \cdot r$

    $discount \leftarrow discount \cdot \gamma$

  return $rewards$

Algorithm 6 Backpropagation Algorithm

function BACKPROPAGATE(v, $value$)

  while v is not NULL **do**

    $N(v) \leftarrow N(v) + 1$

    $Q(v) \leftarrow Q(v) + value$

    $value \leftarrow value \cdot \gamma + R(v)$

    $v \leftarrow Par(v)$

Chapter 3

MCTS Performance Drop Under Model Corruption

In this chapter we investigate the performance drop in MCTS when using an imperfect model. We experimented on the deterministic environments from the MinAtar testbed [54]: Space Invaders, Freeway, and Breakout. In order to investigate the effect of an imperfect model on the performance of MCTS, we modified the true dynamics M of each environment. Each game is slightly different from its original version but the agent only has access to the original game $\hat{M}$. This type of model corruption is motivated by robotics tasks. For instance, in Vemula et al. [51] the simulator knows only the perfect dynamics, but one of the robot's arms is broken, which slightly changes the real dynamics. We introduce our modified environments in this chapter and this type of environment corruption is used in the rest of this thesis (Chapters 4 and 5).

3.1 Space Invaders

A snapshot of the Space Invaders environment is shown in Figure 3.1. The goal of the game is to eliminate the enemies and avoid the shots from them. The gray square at the bottom is the agent and the green rectangle (4×6) at the top are the enemies. The size of the screen is 10×10. At each step the agent has four actions {left, right, none, fire}. The pink and white squares show the agent's and the enemies' bullets respectively. If the agent's bullet hits one of the 24 enemies, it is eliminated and the agent receives a reward of

+1. If the enemies' bullet hits the agent or if all the enemies are eliminated, the game terminates. The enemies move from left to right and drop down when they reach the side of the screen and change direction to the other side. In our modification to this game, at columns 2, 3, 4, 5, and 6 the agent's "fire" action does not work and is equal to the "none" action. The modification is included in the true model M. However the corrupted model $\hat{M}$ is not aware of this modification and only includes the dynamics from the original game where "fire" always works. To investigate the performance drop, we

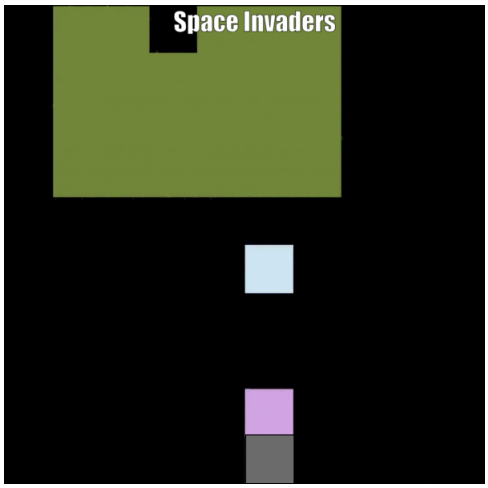

Figure 3.1: A snapshot of the Space Invaders environment. The gray square and the green rectangle represent the agent and the enemies respectively. The pink and white rectangles are the agent's and enemies' bullets respectively.

used two independent agents: One has access to M and one only has access to $\hat{M}$. The former is labeled "True" and the latter is labeled "Corrupted" in the following figures. We investigated the performance for different simulation depths $D_S \in \{0, 5, 10, 20, 50\}$ and number of iterations $N_I \in \{4, 10, 25, 50, 100\}$ in two separate experiments. In the former $N_I = 10$ and in the latter $D_S = 10$, and in both cases $N_S = 10$. In this experiment and all the following experiments in this work we performed 30 independent runs and the error bars show the standard deviation.

Figure 3.2 shows the results of these experiments. As expected, for each simulation depth D_S, the agent using $\hat{M}$ had a lower performance than the agent using M. Increasing D_S improved the performance of the corrupted

agent until $D_S = 20$. Although the true agent's performance improved a lot from $D_S = 20$ to $D_S = 50$, the corrupted agent did not have much improvement due to the model error. This suggests that deeper rollouts with an imperfect model do not improve the performance. Similarly, the true agent's performance improved with more number of iterations but the corrupted agent did not improve after $N_I = 10$.

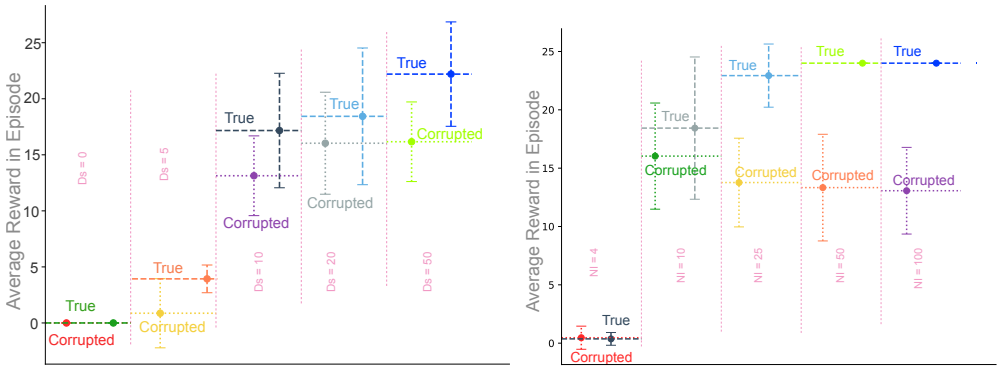

Figure 3.2: The performance drop of MCTS under model corruption in the Space Invaders environment for different simulation depths D_S and number of iterations N_I.

3.2 Freeway

The Freeway environment is shown in Figure 3.3. The goal of the game is to reach the top of the 10×10 screen without getting hit by any of the moving objects. The gray square at the bottom of the screen is the agent. At each step it has three actions {up, down, none}. In each row, a 2×1 rectangle represents a moving object. The green square is the head of the object. The objects move in their own row with a fixed speed in the direction in which their head is pointing. When a moving object reaches the border of the screen it reappears on the other side. If the agent reaches the top of the screen, the game terminates with a reward of +1. If the agent hits one of the objects, the game terminates with a reward of 0. In our modification to this game, at rows 1, 2, 3, 5, 6, and 7 the effect of the "none" action is equal to the "up" action. The agent cannot stop in these rows and has to plan ahead to either

pass them quickly or step back. This modification is only added to the true model M but not to $\hat{M}$.

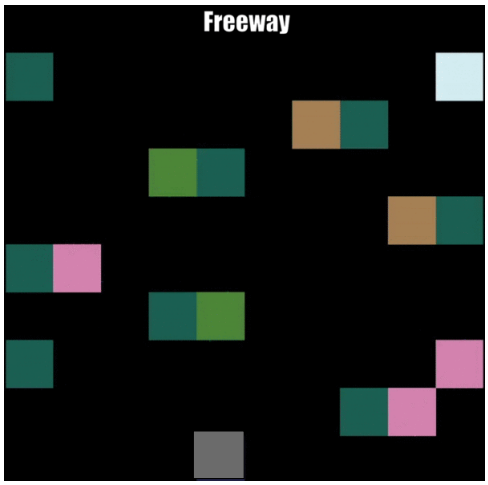

Figure 3.3: A snapshot of the Freeway environment. The gray square is the agent and other rectangles are the moving objects.

We tested two MCTS agents, one using M and one using $\hat{M}$. The modifications make this environment much more difficult to solve and it needs deeper search than Space Invaders. While in Space Invaders the agent cannot shoot in some states, nothing harmful happens even if the agent continues shooting from those states. In Freeway, the agent now has to move in many states, and these states are not avoidable because the agent has to pass them in order to reach the top and get a reward. Thus, the agent needs to plan ahead and start going up when the path is free until the next position that the agent can stop in.

We investigated the performance for different simulation depths $D_S \in \{0, 5, 10, 20, 50\}$ and number of iterations $N_I \in \{3, 20, 50, 100, 200\}$. In the former we used more iterations in this more complex environment, $N_I = 100$ and the latter $D_S = 50$. In both cases $N_S = 10$.

Figure 3.4 shows the results. Due to model error the increased simulation depth couldn't help the agent after $D_S = 5$ and even made the performance worse. Increasing the number of iterations also did not improve the performance of the corrupted agent after $N_I = 100$.

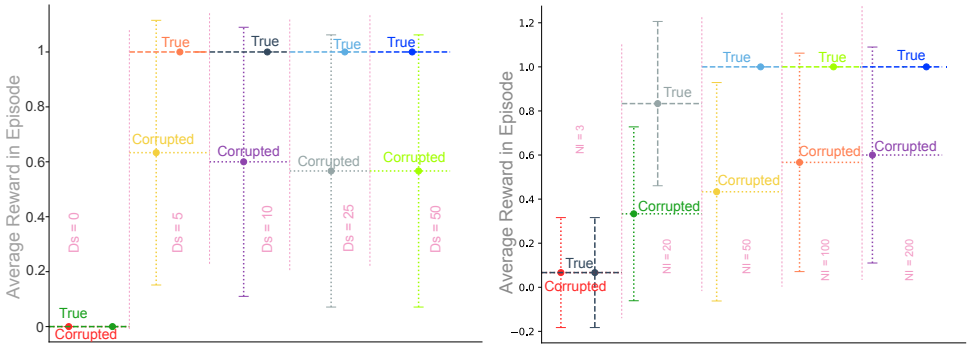

Figure 3.4: The performance drop of MCTS under model corruption in the Freeway Environment for different simulation depths D_S and number of iterations N_I.

3.3 Breakout

The breakout environment is shown in Figure 3.5. The goal of the game is to hit the bricks at the top of the screen with a moving ball. The gray square at the bottom of the screen is the agent. There is a 3×10 brick wall at the top of the screen shown in white. The two connected green and pink squares are the moving ball. The green square shows the head. At each step, the agent has three actions {left, right, none}. When the ball hits the agent it either bounces back at the same angle, or reflects at the exact opposite angle, depending on the agent's movement. If the agent moves towards the ball when hitting it, the ball bounces back at the same angle, but if the agent stands still when hitting it, the ball gets reflected at the opposite angle, like a mirror. When the ball hits any of the bricks, it reflects and that brick disappears with a reward of +1. If the ball reaches the bottom of the screen or if all the bricks disappear, the game terminates. Our modification to this game is that when the agent is at columns 2 or 4 the reflection does not work, the ball can go through the agent in those positions, and the game is over. To avoid this, the agent has to plan ahead and try to control the direction of the ball. This modification also makes the problem much more difficult, even harder than Freeway. In this environment it is much harder to avoid the "bad" states and it needs more thorough search. The agent needs to hit the ball in a direction such that after many bounces, the ball doesn't land on positions 2 or 4 (20% of positions).

Otherwise the agent loses.

For the first set of experiments, we again used a deeper search, setting N_I and N_S equal to 100 and 10 respectively and $D_S \in \{0, 5, 10, 25, 50\}$. For the second part of the experiments we used $D_S = 50$, $N_S = 10$, and $N_I \in \{3, 20, 50, 100\}$.

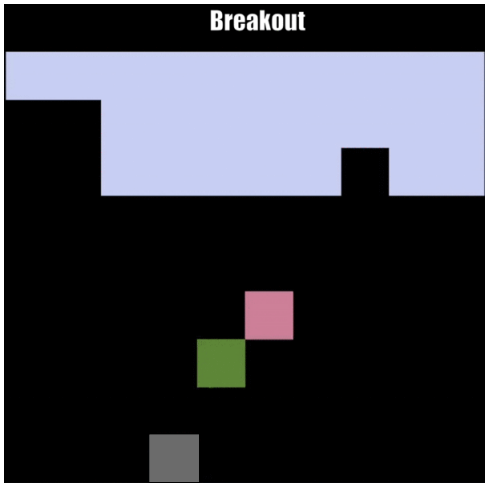

Figure 3.5: A snapshot of the Breakout environment. The gray square and the white rectangle represent the agent and the bricks respectively. The pink and green squares represent the moving ball.

Figure 3.6 shows the results. Again, the agent with the corrupted model has much lower performance than the agent with the true model. The performance drop in this environment is more than in the other two environments, which shows the difficulty of handling it without access to the true dynamics. Similar to Freeway, the increased simulation depth couldn't help the agent after $D_S = 10$ and even made the performance worse. Also, increasing the number of iteration did not seem to improve the performance of the corrupted agent at all.

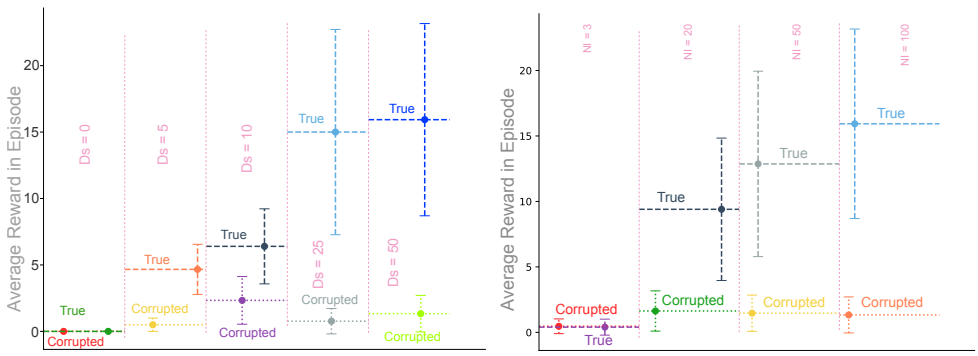

Figure 3.6: The performance drop of MCTS under model corruption in the Breakout Environment for different simulation depths D_S and number of iterations N_I.

3.4 Summary

In this chapter we talked about the performance drop in MCTS when using an imperfect model. We explained our modifications to the MinAtar environments and the type of model error we used for the rest of the experiments. The results show a significant drop in the performance of MCTS when searching using an imperfect model. They also suggest that increasing the simulation depth in an imperfect model not only is not helpful after some point but it can also be harmful and reduce the performance.

Chapter 4

Combining DQN and MCTS

With an accurate model, model-based methods can be much more sample efficient than model-free methods [42]. However, in most cases their performance is highly sensitive to the quality of the model. Their performance drops heavily when using an inaccurate model. In a multi-step trajectory, the error compounds at each step which results in a significant error for longer trajectories [53]. Research to address this issue includes learning the multi-step model directly [3] or using the model's predictions in the training to correct them [46].

As mentioned in Section 2.4, MCTS is a very successful model-based search method that relies on deep simulations. Due to compounding error, having an imperfect model causes erroneous return predictions from simulations which results in a low performance even after many iterations. In this chapter, we address the issue of model error in MCTS by proposing a new method called Deep Q-network MCTS (DQMCTS).

DQMCTS combines the model-free algorithm DQN with MCTS to deal with imperfect models. DQN solely uses the interactions between agent and environment to train a value network. Thus, we can use its value function safely even when the model is not accurate. The idea consists of taking advantage of DQN values in MCTS in two ways: During simulations, and during expansion. We call the former method DQ-Simulation and the latter DQ-Expansion.

4.1 DQ-Simulation

Deep rollouts are one of the fundamental aspects of MCTS. MCTS utilizes deep rollouts to evaluate its leaf nodes and backpropagates those evaluations to tree nodes higher up. Using an imperfect model misleads these rollouts and can drop the performance significantly. To compensate for incorrect rollouts, we investigate using a learned DQN value function to evaluate leaf nodes, or rolling out to a fixed depth and evaluating the endpoints with DQN values. This method gives insights in the choice of DQN value function and of a good depth of the rollouts. For more details see [22].

4.2 DQ-Expansion

During the expansion process in MCTS, all children of a specific node are added to the search tree. In the basic framework, the initial value of the newly added nodes is zero. This initial value gets updated later with rollout results or backpropagated values. A better idea is to use a heuristic to initialize the children's values. Creating a hand designed heuristic function might not be simple and may require profound knowledge of the environment.

An initial value of zero puts all the pressure of learning on the rollouts which as we mentioned can be considerably misleading with an imperfect model. In DQMCTS, we suggest to use a learned DQN value function as a heuristic to overcome the model's imperfections and improving the performance. Algorithm 7 shows the modified expand function. The term written in red is the difference between basic MCTS and DQMCTS. In DQ-Expansion, when we add a new child v_i to the search tree, its initial value $Q(v_i)$ comes from the learned DQN value function q_{dqn}.

4.3 Experiments

In this section we first explain the details of our experimental design, and then show and discuss the results. We experimented with the new methods on the modified version of Space Invaders, Freeway, and Breakout from the MinAtar

Algorithm 7 DQ-Expansion Algorithm

function DQ-EXPAND(v, q_{dqn})

  for $a_i \in \mathcal{A}$ **do**

    $s_i, r_i \leftarrow \hat{M}(S(v), a_i)$

    create a node v_i with state s_i and reward r_i

    $N(v_i) \leftarrow 0$

    $Q(v_i) \leftarrow \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} q_{dqn}\big(S(v), a'\big)$

  return a random child of v

testbed [54]. See chapter 3 for more details on these environments.

The DQMCTS experiments for each environment proceed in two stages. First, the agent trains a DQN agent on the real environment for a fixed number of episodes and stores its learned value function q_{dqn}. In the second stage, the agent plays in the environment using DQMCTS and the stored value function q_{dqn} from the first stage. We compared the performance of DQMCTS with two MCTS baselines: An MCTS agent with access to the true dynamics M and an MCTS agent which only has access to the corrupted model $\hat{M}$. In the following plots, the former is labeled with "True MCTS" and the latter is labeled with "Corrupted MCTS". To have fair experiments, we optimized the exploration rate c separately for each of the experiments over a set of four values $c \in \{0.5, 1, \sqrt{2}, 2\}$. For reproducibility, the chosen value for c, N_I, N_S, and D_S are mentioned in each environment later on.

We study the effect of the learned value function in DQMCTS on snapshots from different stages of learning. To investigate the effect of rollouts with an imperfect model we also experimented with different simulation depths D_S which we mentioned for each agent in their section.

4.3.1 DQN Details

The DQN used in all the environments had similar parameters which we explain in more detail in this section. For the q_{dqn} network, we used a fully connected neural network with two hidden layers each containing 64 units.

We used the RMSProp optimizer with step size $\lambda = 0.00025$ to train the network. The training batch size B_s is 32 and the target network update frequency f_t is every 1000 interactions. To have more robust convergence, we started with $\epsilon = 1$ and linearly reduced its value to 0.1 over the course of 100000 interactions. After that the ϵ value remains at 0.1 for the rest of the training. We used a buffer of size 100000 to store the transitions. When the buffer is full the oldest transition is replaced with the new one.

4.3.2 Space Invaders

The right plot in Figure 4.1 shows the learning curve of training the dqn agent. We test the quality of the learned value function q_{dqn} at different episodes using $\epsilon = 0$. The left plot in Figure 4.1 shows the evaluation of value functions at different stages of training. Based on this evaluation we picked three learned q_{dqn} at different levels of training to use in DQMCTS: after episodes 3000, 7000, and 20000. These value functions are shown in red in Figure 4.1.

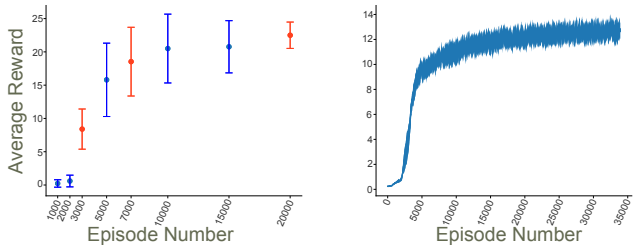

Figure 4.1: Left plot: performance of value functions at different training stages using $\epsilon = 0$. Right plot: learning curve of DQN on Space Invaders.

We compared the performance of DQ-Expansion and the combination of both DQ-Expansion and DQ-Simulation (DQMCTS) with MCTS baselines using 4 different simulation depths, $D_S \in \{0, 5, 10, 20\}$, and number of iterations and simulations $N_I = N_S = 10$. Figure 4.2 shows the comparison plots. We observe that while DQ-Expansion always has higher performance than the corrupted MCTS baseline, DQN outperforms all of them. This is because even if we evaluate the leaf nodes using the DQN, the search tree itself is built with incorrect transitions. This encouraged us to develop another idea to weigh

different parts of the search tree which we explain in Chapter 5. Another observation is that increasing the simulation depth D_S reduces the DQMCTS performance. The reason is that an unrealistic rollout might lead to a state with good DQN values that can happen during DQ-Simulation. This effect falsely evaluates a leaf node to be much better than it actually is. Thus, when using DQMCTS we suggest to completely eliminate the simulation steps and only use the q_{dqn} to evaluate leaf nodes as in AlphaZero [40]. However, in the case that we only use DQ-Expansion, increasing D_S improves the performance because in DQ-Expansion the q_{dqn} values only give an initial direction to the search. Table 4.1 shows the best parameter c for each of the algorithms and for different simulation depths D_S.

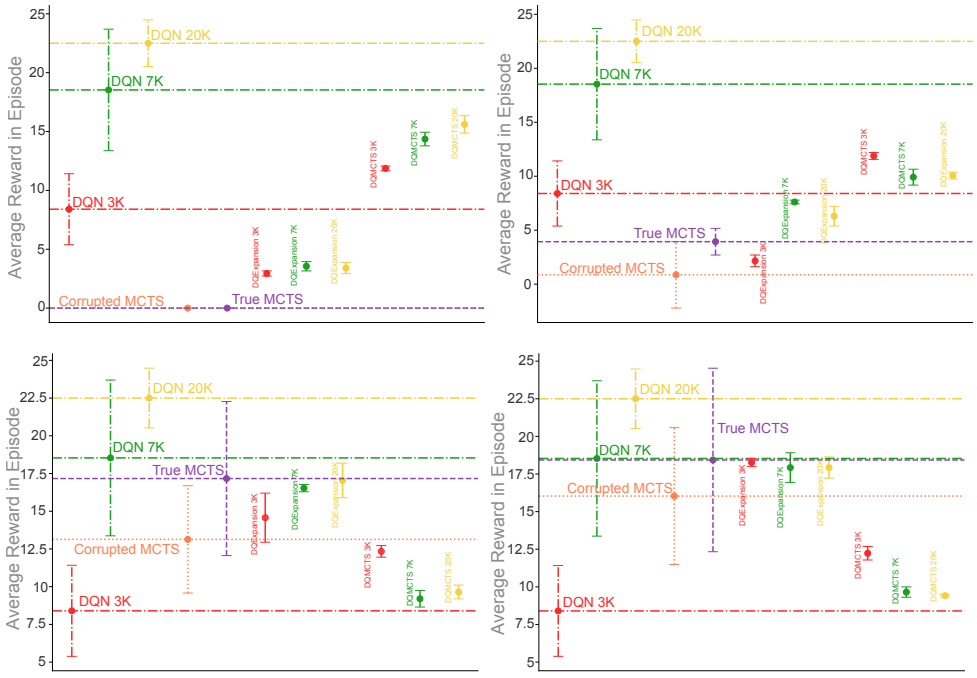

Figure 4.2: Comparison of DQMCTS performance with MCTS baselines and DQN in Space Invaders. From left to right and top to bottom the simulation depth D_S is equal to 0, 5, 10, 20 respectively.

4.3.3 Freeway

Figure 4.3 shows the training and evaluation of the DQN agent at different stages of training. Based on this evaluation we again picked three learned q_{dqn}

	$D_S = 0$	$D_S = 5$	$D_S = 10$	$D_S = 20$
True MCTS	2	1	0.5	0.5
Corrupted MCTS	2	1	2	2
DQExpansion 3K	0.5	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{2}$
DQExpansion 7K	$\sqrt{2}$	2	0.5	2
DQExpansion 20K	2	$\sqrt{2}$	2	0.5
DQMCTS 3K	$\sqrt{2}$	2	2	$\sqrt{2}$
DQMCTS 7K	0.5	2	0.5	2
DQMCTS 20K	$\sqrt{2}$	$\sqrt{2}$	2	1

Table 4.1: Best parameter c for each (algorithm, D_S) pair in Space Invaders.

at different levels of training to use in DQMCTS: after episode 7000, after episode 10000, after episode 20000. The DQN results show that this task was too difficult for the DQN agent to solve and the agent did not even get close to perfect play (average reward of 1). Next, we evaluate whether such a poorly learned value function can still be useful in DQMCTS.

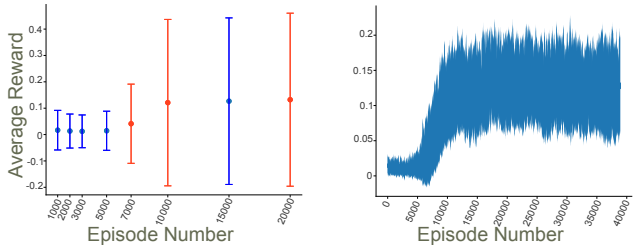

Figure 4.3: Left plot: performance of value functions at different training stages using $\epsilon = 0$. Right plot: learning curve of DQN on Freeway.

We compare the performance of DQMCTS (combination of DQ-Expansion and DQ-Simulation) and DQ-Expansion with the same MCTS baselines at 5 different simulations depths, $D_S = [0, 5, 10, 25, 50]$. As mentioned in Chapter 3, this environment is harder than Space Invaders and needs more thorough search, so we used $N_I = 100$ and $N_S = 10$. Figure 4.4 shows the comparison plots. The results show a significant improvement over the corrupted MCTS baseline and the DQN agent in all cases. Another observation from Figure 4.4 is that DQMCTS has lower standard error than DQN and the corrupted MCTS baseline which makes it a more robust algorithm. Table 4.2 shows the best parameter c for each of the algorithms at different simulation depths D_S.

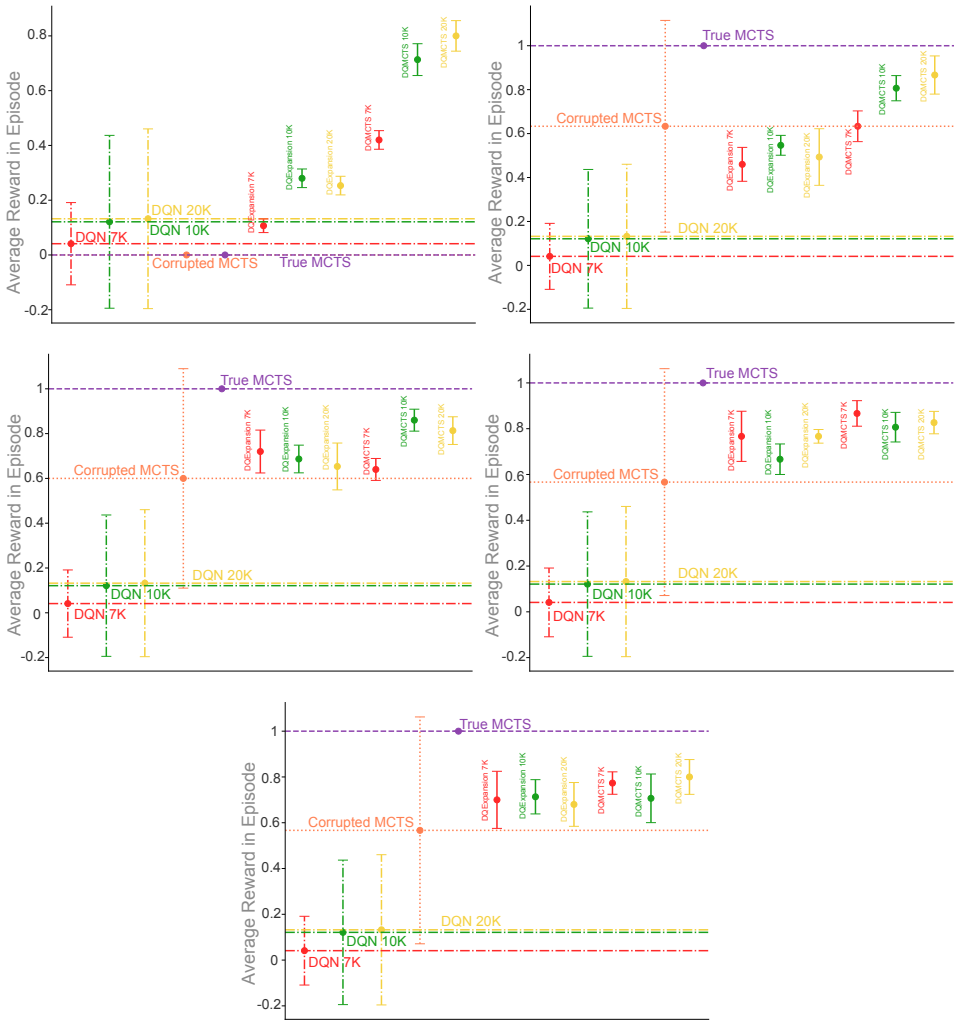

Figure 4.4: Comparison of DQMCTS performance with MCTS baselines and DQN in Freeway. From left to right and top to bottom the simulation depth D_S is equal to 0, 5, 10, 25, 50 respectively.

	$D_S = 0$	$D_S = 5$	$D_S = 10$	$D_S = 25$	$D_S = 50$
True MCTS	2	2	2	2	$\sqrt{2}$
Corrupted MCTS	2	2	$\sqrt{2}$	1	$\sqrt{2}$
DQExpansion 7K	2	2	$\sqrt{2}$	$\sqrt{2}$	2
DQExpansion 10K	2	$\sqrt{2}$	2	$\sqrt{2}$	2
DQExpansion 20K	2	1	2	2	$\sqrt{2}$
DQMCTS 7K	2	$\sqrt{2}$	1	2	0.5
DQMCTS 10K	2	$\sqrt{2}$	0.5	$\sqrt{2}$	2
DQMCTS 20K	$\sqrt{2}$	1	2	2	2

Table 4.2: Best parameter c for each (algorithm, D_S) in Freeway.

	$D_S = 0$	$D_S = 5$	$D_S = 10$	$D_S = 25$	$D_S = 50$
True MCTS	2	$\sqrt{2}$	1	2	2
Corrupted MCTS	2	$\sqrt{2}$	2	0.5	0.5
DQExpansion 7K	2	2	1	2	0.5
DQExpansion 10K	0.5	0.5	0.5	1	1
DQExpansion 20K	$\sqrt{2}$	1	0.5	0.5	1
DQMCTS 7K	1	$\sqrt{2}$	1	0.5	2
DQMCTS 10K	1	1	0.5	2	2
DQMCTS 20K	2	2	0.5	$\sqrt{2}$	0.5

Table 4.3: Best parameter c for each (algorithm, D_S) in Breakout

4.3.4 Breakout

Similar to Freeway, we picked three learned q_{dqn} after episodes 7000, 10000, and 20000 (Figure 4.5).

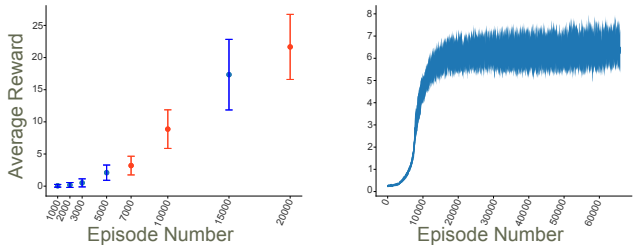

Figure 4.5: Left plot: performance of value functions at different training stages using $\epsilon = 0$. Right plot: learning curve of DQN on Breakout.

Figure 4.6 shows the performance of DQMCTS with MCTS and DQN baselines. We compared with the same five simulations depths, $D_S = [0, 5, 10, 25, 50]$, with $N_I = 100$ and $N_S = 10$. We observe that DQMCTS and DQ-Expansion outperformed the corrupted MCTS baseline, however DQN still outperformed them both. This again suggests that evaluating the leaf nodes is not enough to deal with model corruption and it needs further measures to properly make use of the built search tree. The Breakout and Space Invaders results motivated our UA-MCTS method explained in Chapter 5. Table 4.3 shows the best parameter c for each of the algorithms with different simulation depths D_S.

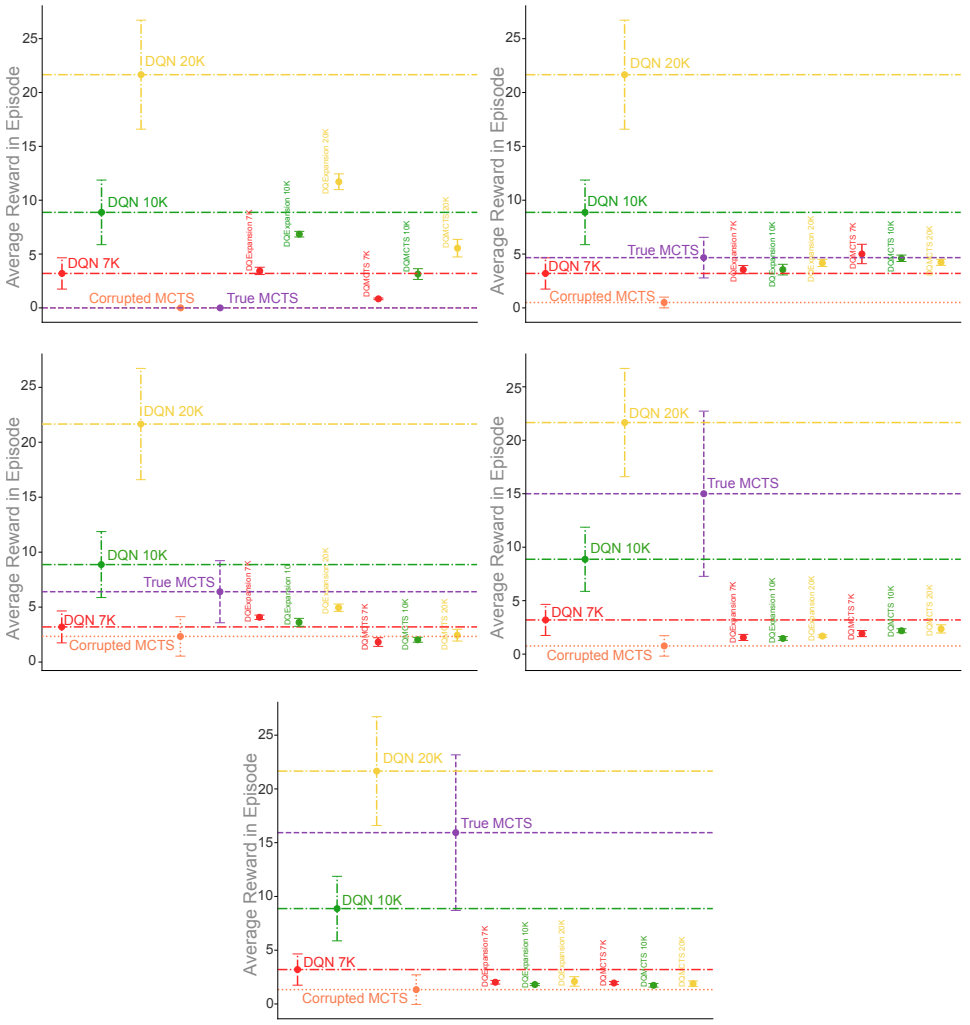

Figure 4.6: Comparison of DQMCTS performance with MCTS baselines and DQN in Breakout. From left to right and top to bottom the simulation depth D_S is equal to 0, 5, 10, 25, 50 respectively.

4.4 Summary

To summarize, in this chapter we introduced a new method DQMCTS to deal with model imperfection using a model free algorithm DQN. DQMCTS uses a DQN learned value function to evaluate leaf nodes in the search tree. We compared our method's performance with DQN and MCTS baselines in three MinAtar environments: Space Invaders, Freeway, and Breakout. DQMCTS outperformed the MCTS baseline in all cases, and even outperformed the DQN baseline in Freeway. However, DQN outperformed our method in the other two environments, which suggests that evaluating the leaf nodes is not enough to deal with model corruption in all cases. Therefore, we also change the way MCTS builds and traverses the search tree in Chapter 5. [1]

[1]Implementation of these ideas can be found in: `https://github.com/ualberta-mueller-group/imperfect_model_code`

Chapter 5

Uncertainty-Adapted MCTS

In Chapter 4, we used a learned DQN value function to evaluate the leaf nodes of the search tree in order to deal with model uncertainty. That approach showed improvements over the MCTS baseline with the corrupted model. However, there was a significant gap between the MCTS baseline with the true model and our method. This motivates our new method, Uncertainty-Adapted MCTS (UA-MCTS) which changes the behavior of MCTS more fundamentally.

Assuming a measure of model uncertainty is available, we modify the behavior of MCTS. The choice of uncertainty measure is independent of UA-MCTS. It can be a simple euclidean distance between the model predictions and the true next state or a more complicated similarity measure between their representations. It can also be learned by environment interactions or can be given by a domain expert.

Given an uncertainty model, there are two general behaviors one might follow. The first behavior is the exploratory approach: The agent tries to explore the uncertain parts of the model in order to see what it might achieve there. The upside to this approach is that the agent might find a policy that performs better than any policy that could have been found when trusting the model. The downside is that taking exploratory actions may be very costly for the agent, for example in robotics tasks or in real environments.

The other approach is the risk-avoiding approach: In this approach, the agent tries to avoid the uncertain parts of the model and exploits the certain-

ties. This approach can be seen as safer because the agent does not take any risk by exploring uncertain states. The downside of this approach is that in the case that the optimal policy is not available in the model, the agent is very unlikely to find it. However it will achieve a sub-optimal policy if one exists in the model.

In this work, we propose Uncertainty-Adapted MCTS (UA-MCTS) based on the exploitatory attitude. Our goal for the agent is to spend less search in the uncertain parts, without completely excluding them.

We modified each of the four main components of MCTS: selection, expansion, simulation, and backpropagation in UA-MCTS, by taking uncertainty into account. Since the specific choice of uncertainty measure is independent of the algorithm, we leave it unspecified while explaining the ideas, and will test concrete choices in experiments. During the expansion phase in UA-MCTS, the agent stores the uncertainty of the new transitions as an attribute $\hat{U}$ in the nodes. Since the model is fixed and the environment is static, this model error does not change over time. Each node has a measure of its transition uncertainty. We will explain how to use it in different parts of UA-MCTS in the rest of this chapter.

5.1 UA-Expansion

During UA-Expansion we exclude one of the new child nodes from getting added to the search tree with probability τ. With probability $1 - \tau$, UA-Expansion is exactly the same as regular Expansion. The probability of a child being excluded is proportional to its uncertainty, so the child with the highest uncertainty is the most likely to get excluded from the search tree. In the case that all children have zero uncertainty, we do not exclude any of them. This idea reduces the branching factor of the search tree and leaves more time to search the more certain parts of the model. For more details see [22].

5.2 UA-Simulation

MCTS gives all rollouts the same credit by taking the unweighted average of all rollout results in a state. UA-MCTS changes this behavior and gives more weight to the more certain rollouts. The UA-Simulation approach first calculates the sum over the uncertainty of all transitions in a rollout to measure its uncertainty as a whole, then takes a weighted average between different rollouts using the softmax of the negative uncertainty as the weights. For more details on this method see [22].

5.3 UA-Selection

As we discussed in Chapter 2, selection in MCTS starts from the root of the search tree and decides which child is the most promising to follow until it reaches a leaf node. The child of a node with highest UCT value is chosen (Equation 5.1).

$$UCT(v) = \frac{Q(v)}{N(v)} + C\sqrt{\frac{\ln\Big(N(Par(v))\Big)}{N(v)}} \tag{5.1}$$

Intuitively, our goal is that the selection should also take the uncertainty of these children into account, in a way that more uncertain children are chosen less often. We change the UCT formula in a way that favors more certain children over uncertain ones.

We formalize this intuition by adding a new term to the standard UCT formula. This new part, $(1 - \alpha_v)$ in Equation 5.3, is bounded between 0 and 1. It is smaller when there is more uncertainty which results in a lower UCT value for this child.

$$UA - UCT(v) = \frac{Q(v)}{N(v)} + C\sqrt{\frac{\ln\Big(N(Par(v))\Big)}{N(v)}} \times (1 - \alpha_v) \tag{5.2}$$

$$\alpha_v \doteq \frac{e^{\hat{U}(v)/\tau}}{\sum\limits_{v_j \in Ch(Par(v))} e^{\hat{U}(v_j)/\tau}} \tag{5.3}$$

Equation 5.3 shows the definition of α_v which is a *softmax* function over the uncertainties of all the siblings of node v. τ is the temperature parameter of the *softmax* which decides how much we should pay attention to the uncertainty values. We will tune this τ parameter in our experiments in Section 5.5. Algorithm 8 shows the pseudo code of UA-Selection. The red parts are the parts added to normal MCTS selection.

Algorithm 8 Uncertainty Adapted Selection Algorithm.

Parameter: temperature τ for softmax

function UA-SELECT(v)

while v is expanded **do**

for $v_i \in Ch(v)$ **do**

$$\alpha_i \leftarrow \frac{e^{\hat{U}(v_i)/\tau}}{\sum_{v_j \in Ch(v)} e^{\hat{U}(v_j)/\tau}}$$

$$v \leftarrow \underset{v_i \in Ch(v)}{\text{argmax}} \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\ln N(v)}{N(v_i)}} \cdot (1 - \alpha_i)$$

return v

5.4 UA-Backpropagation

Giving more credit to certain transitions while selecting a node is good but it is not enough. In UA-Backpropagation we modify MCTS backpropgation in such a way that more certain children have more impact on the value of their parents. In MCTS, the backpropagate function gives an equal weight of 1 to all transitions, so the children chosen more often have more influence on the value of their parent. Let $\mu(v)$ be the average reward seen from node v ($\mu(v) = \frac{Q(v)}{N(v)}$ in MCTS). Equation 5.4 shows $\mu(v)$ in terms of its children values in normal MCTS.

$$\mu(v) = \frac{\sum_{v_i \in Ch(v)} Q(v_i) \times N(v_i)}{\sum_{v_i \in Ch(v)} N(v_i)} \tag{5.4}$$

The goal of UA-Backpropagate is to give more weight to the more certain

children while updating the value of their parent. Equation 5.5 shows the weighted average reward computation using UA-Backpropagation. The new parameter α_i has an inverse relationship with the uncertainty $\hat{U}$ of node v_i (shown in Equation 5.6).

$$\mu(v) = \frac{\sum_{v_i \in Ch(v)} Q(v_i) \times N(v_i) \times \alpha_i}{\sum_{v_i \in Ch(v)} N(v_i) \times \alpha_i} \tag{5.5}$$

$$\alpha_i = \frac{e^{-\hat{U}(v_i)}}{\sum_{v_j \in Ch(Par(v_i))} e^{-\hat{U}(v_j)}} \tag{5.6}$$

During regular MCTS backpropagation, the backpropagated value is added to the parent values and the number of visits of the parent is incremented by one. In UA-Backpropagate, we first calculate the coefficient α_i in Equation 5.6 and update the parent value with the child i's value weighted by α_i. Algorithm 9 shows the pseudo code for this method.

Using UA-Backpropagate changes the $Q(v)$ stored in each parent node. In order to reach the exact value as in Equation 5.5, we need to slightly change the exploitatory term UCT formula in selection. The UA-Select function in Algorithm 9 shows this change. UA-Selection changes the exploration term of UCT and UA-Backpropagation changed the exploitation term of UCT. Thus, we can use both ideas at the same time.

5.5 Experiments

In this section we explain our experimental designs and results. We experimented with UA-MCTS on modified versions of three MinAtar environments, as discussed in Chapter 3: Space Invaders, Freeway, and Breakout. In order to investigate the effect of each modification in UA-MCTS, for each agent, we combined one component of UA-MCTS (e.g. UA-Selection) with normal MCTS and compared the results with MCTS baselines. UA-SB is the combination of both UA-Selection and UA-Backpropagation, and UA-MCTS is the combination of all four components.

Algorithm 9 Uncertainty Adapted Backpropagation Algorithm.

Parameter: temperature τ for softmax

function UA-BACKPROPAGATE(v, $value$)

while v is not NULL **do**

$N(v) \leftarrow N(v) + 1$

$$\alpha = \frac{e^{-\hat{U}(v)/\tau}}{\sum_{n \in Ch(Par(v))} e^{-\hat{U}(n)/\tau}}$$

$Q(v) \leftarrow Q(v) + \alpha \cdot value$

$value \leftarrow value \cdot \gamma + R(v)$

$v \leftarrow Par(v)$

function UA-SELECT(v)

while v is expanded **do**

for $v_i \in Ch(v)$ **do**

for $v_j \in Ch(v_i)$ **do**

$$\alpha_j = \frac{e^{-\hat{U}(v_j)/\tau}}{\sum_{n \in Ch(v_i)} e^{-\hat{U}(n)/\tau}}$$

$$d_i = \sum_{v_j \in Ch(v_i)} N(v_j) \cdot \alpha_j$$

$$v \leftarrow \underset{v_i \in Ch(v)}{\operatorname{argmax}} \frac{Q(v_i)}{d_i} + c\sqrt{\frac{\ln N(v)}{N(v_i)}}$$

return v

We experimented with two scenarios: offline and online. In the offline scenario, we assume that the agent has access to the true uncertainty of the model $\hat{U} = U$. This way we can investigate the performance of our ideas without the additional complication and errors from learning the uncertainty.

The online scenario consists of two stages. First, a normal unmodified MCTS interacts with the real environment and gathers a buffer B of transitions. When B is full we stop the MCTS agent and train the measure of uncertainty $\hat{U}$ for 10000 training steps. In the second stage, we use the trained $\hat{U}$ with UA-MCTS to search in the model and then play in the environment.

For each experiment, we optimize parameters c and τ over a set of values $c \in \{0.5, 1, \sqrt{2}, 2\}$, $\tau \in \{0.1, 0.5, 0.9\}$. Other hyper-parameters such as N_I, N_S, D_S are specified in each experiment separately. To study the effect of the learned uncertainty $\hat{U}$ on the performance of UA-MCTS we used three different buffer sizes, 1000, 3000, and 7000, for each experiment.

5.5.1 Measure of Uncertainty (U)

Any measure of uncertainty can be chosen in the UA-MCTS framework. In the following experiments we used a simple euclidean distance between the true and predicted state representations as the model uncertainty. Thus for a given state action pair (s, a) the uncertainty $U(s, a)$ is defined in Equation 5.7.

$$U(s,a) = \left(\hat{M}(s,a) - M(s,a)\right)^2 \tag{5.7}$$

In the offline scenario the agent has access to the function $U(s, a)$ for any $s \in \mathcal{S}$, $a \in \mathcal{A}$. In the online scenario we used a neural network with two fully connected layers (64 units in each layer) to approximate U. The neural network is trained over the gathered buffer B from MCTS's interaction with the environment for 10000 steps using the loss function in Equation 5.8. We used the Adam optimizer [20] and a step size of 0.001 with batch size of 32.

$$L = \sum_{s,a,U(s,a)\in B} \left(\hat{U}(s,a) - U(s,a)\right)^2 \tag{5.8}$$

5.5.2 Space Invaders

Offline Scenario

The results are shown in Figure 5.1. In this experiment, $N_I = 10$, $D_S = 20$, and $N_S = 10$. The combination of UA-Selection and UA-Backpropagation outperformed their separate versions and both MCTS baselines. UA-MCTS, the combination of all four components, almost reached perfect play which has an average reward of 24. Table 5.5.2 shows the best parameters c and τ for each of the experiments.

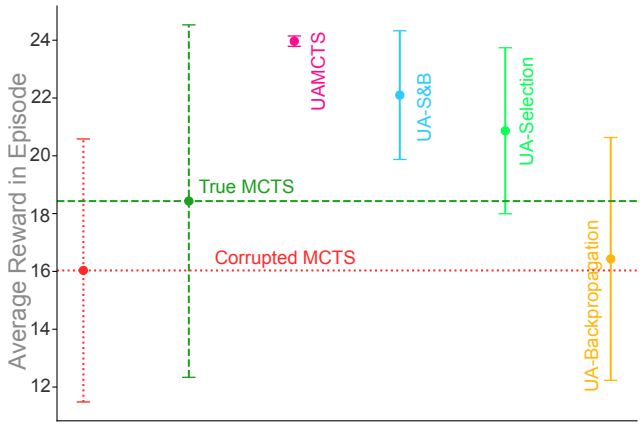

Figure 5.1: Comparison of UA-MCTS and its components with MCTS baselines in Space Invaders.

	C	τ
True MCTS	0.5	NA
Corrupted MCTS	2	NA
UA-MCTS	1	0.1
UA-S&B	$\sqrt{2}$	0.1
UA-Selection	$\sqrt{2}$	0.1
UA-Backpropagation	$\sqrt{2}$	0.9

Table 5.1: Best parameters c and τ for each experiment in Space Invaders for the offline scenario.

Online Scenario

Figure 5.2 shows the performance of UA-MCTS with different buffer sizes for training $\hat{U}$. N_I, D_S, and N_S are the same as in the offline scenario. UA-MCTS and its components had a much better performance when having access to the U function but their performance with a learned $\hat{U}$ function is still better than both MCTS baselines. With less data to train $\hat{U}$, the performance of UA-MCTS slightly drops but still outperforms the MCTS baselines. Table 5.5.2 shows the best c and τ parameters for each method.

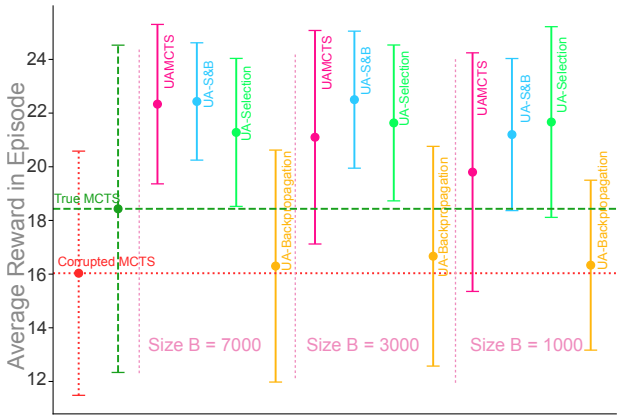

Figure 5.2: Comparison of UA-MCTS and its components with MCTS baselines in Space Invaders in the online scenario.

	Size B = 7000		Size B = 3000		Size B = 1000	
	C	τ	C	τ	C	τ
UA-MCTS	0.5	0.1	1	0.1	1	0.1
UA-S&B	1	0.1	$\sqrt{2}$	0.1	$\sqrt{2}$	0.1
UA-Selection	2	0.1	2	0.1	1	0.1
UA-Backpropagation	2	0.5	$\sqrt{2}$	0.5	$\sqrt{2}$	0.9

Table 5.2: Best parameters c and τ for each experiment in Space Invaders for the online scenario.

5.5.3 Freeway

Offline Scenario

Figure 5.3 shows the results for the offline scenario in the Freeway environment. All the component methods improved over the MCTS with the corrupted model but UA-MCTS outperformed them all and almost reached the performance of MCTS with the true model. For freeway we picked N_I, D_S, and N_S to be 100, 50, and 10 respectively due to the difficulty of the environment.

Online Scenario

Figure 5.4 shows the performance of UA-MCTS in the online scenario (N_I, D_S, and N_S are same as in the offline scenario). The performance of all UA-MCTS versions with the learned $\hat{U}$ functions is better than the MCTS baselines.

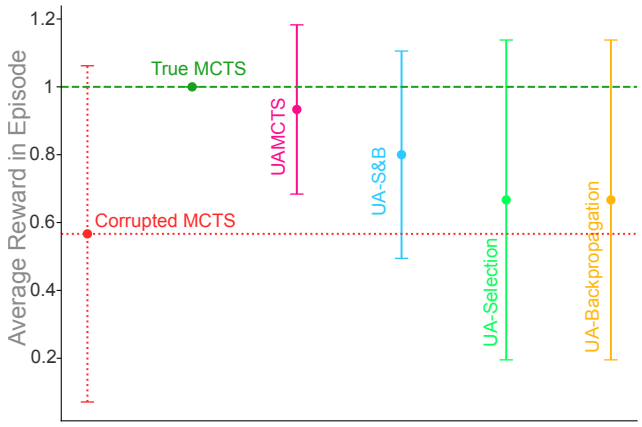

Figure 5.3: Comparison of UA-MCTS and its components with MCTS baselines in Freeway.

	C	τ
True MCTS	$\sqrt{2}$	NA
Corrupted MCTS	$\sqrt{2}$	NA
UA-MCTS	2	0.1
UA-S&B	0.5	0.1
UA-Selection	1	0.1
UA-Backpropagation	2	0.5

Table 5.3: Best parameters c and τ for each experiment in Freeway for the offline scenario

However, with the true U function (Figure 5.3), UA-MCTS performed much better. Table 5.5.3 shows the best c and τ parameters for each method.

5.5.4 Breakout

Offline Scenario

Figure 5.5 shows the results for the offline scenario in the Breakout environment. In this experiment, $N_I = 100$, $D_S = 50$, and $N_S = 10$. With the perfect uncertainty model, $\hat{U} = U$, the UA-MCTS agent achieves the True MCTS performance.

Online Scenario

Figure 5.6 shows the performance of UA-MCTS and its individual components for the online scenario. Table 5.5.4 shows the parameters for each of these

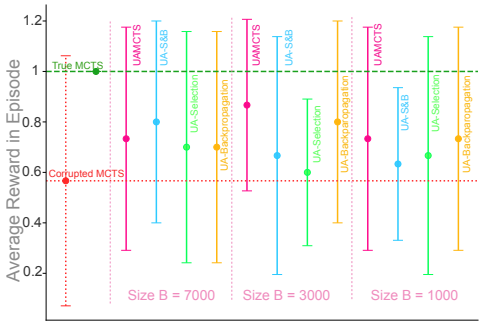

Figure 5.4: Comparison of UA-MCTS and its components with MCTS baselines in Freeway in the online scenario.

	Size B = 7000		Size B = 3000		Size B = 1000	
	C	τ	C	τ	C	τ
UA-MCTS	1	0.1	$\sqrt{2}$	0.1	2	0.1
UA-S&B	1	0.1	0.5	0.1	0.5	0.1
UA-Selection	2	0.1	0.5	0.1	1	0.1
UA-Backpropagation	$\sqrt{2}$	0.9	$\sqrt{2}$	0.9	$\sqrt{2}$	0.9

Table 5.4: Best parameters c and τ for each experiment in Freeway for the online scenario.

experiments. Our method outperformed the corrupted MCTS baseline but again could not achieve its own offline performance. Compared to Freeway and Space Invaders, there is a larger drop in the performance of offline and online scenarios in Breakout. The red horizontal line in Figure 5.6 shows that MCTS performs quite poorly with a corrupted model in this environment, with an average reward of 1.33. In order to train the uncertainty measure $\hat{U}$ we used a buffer gathered by the MCTS agent and due to the poor performance of the MCTS baseline we can deduct that the gathered buffer is not sufficient to train

	C	τ
True MCTS	2	NA
Corrupted MCTS	0.5	NA
UA-MCTS	$\sqrt{2}$	0.1
UA-S&B	2	0.1
UA-Selection	$\sqrt{2}$	0.1
UA-Backpropagation	2	0.5

Table 5.5: Best parameters c and τ for each experiment in Breakout for the offline scenario.

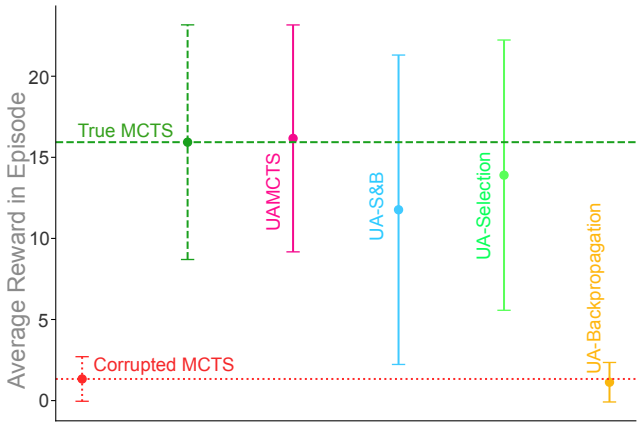

Figure 5.5: Comparison of UA-MCTS and its components with MCTS baselines in Breakout.

a good model $\hat{U}$. Since the buffer sizes are similar for each environment, we investigated this hypothesis, by checking the percentage of unique data in the gathered buffers. For Space Invaders and Freeway more than 90% of the data was unique on average but for Breakout the average of unique data in buffers was less than 20% which validates our hypothesis. However, even with such a poorly trained $\hat{U}$, UA-MCTS managed to outperform the MCTS baseline. This phenomenon might occur in any environment in which model corruption is in a way that significantly drops MCTS performance and causes an insufficient buffer content. As a future research direction, we want to constantly gather a buffer and train $\hat{U}$ while UA-MCTS interacts with the environment.

	Size B = 7000		Size B = 3000		Size B = 1000	
	C	τ	C	τ	C	τ
UA-MCTS	1	0.1	$\sqrt{2}$	0.1	2	0.9
UA-S&B	2	0.1	2	0.1	2	0.1
UA-Selection	$\sqrt{2}$	0.1	1	0.1	0.5	0.1
UA-Backpropagation	$\sqrt{2}$	0.1	$\sqrt{2}$	0.5	$\sqrt{2}$	0.5

Table 5.6: Best parameters c and τ for each experiment in Breakout for online scenario.

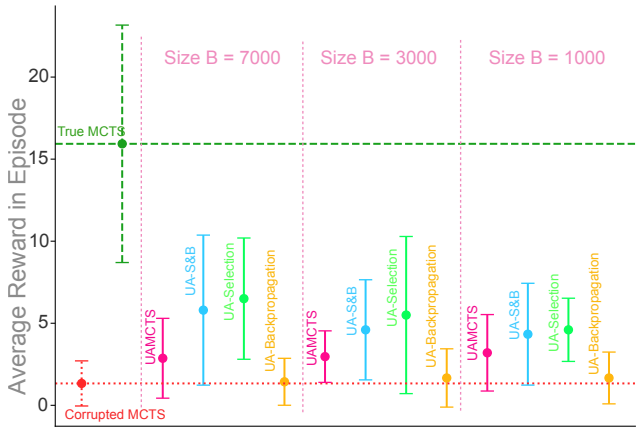

Figure 5.6: Comparison of UA-MCTS and its components with MCTS baselines in Breakout in the online scenario.

5.6 Scaling Experiments

In this section, we investigate the online and offline performance of UA-MCTS agents with different numbers of iterations. Figures 5.7-5.9 demonstrate the experimental results. In the offline scenario in which the uncertainty model is perfect, with more iterations, the performance improved, and even in the Space Invaders environment, it reached perfect play. For the online scenario, in Freeway and Space Invaders environment the performance improved with more iterations but could not achieve the offline performance due to the error in the uncertainty model. Since the learned uncertainty in the Breakout environment was insufficient, the online agent performed poorly with both lower and higher number of iterations, which shows the importance of the learned measure of the uncertainty.

5.7 Summary

In this chapter we introduced a new method, UA-MCTS, to deal with model uncertainty. UA-MCTS has the four main components of MCTS but their behavior is different due to taking model uncertainty into account. We showed the performance of UA-MCTS when having access to the true uncertainty function U in the offline scenario. UA-MCTS achieved the performance of

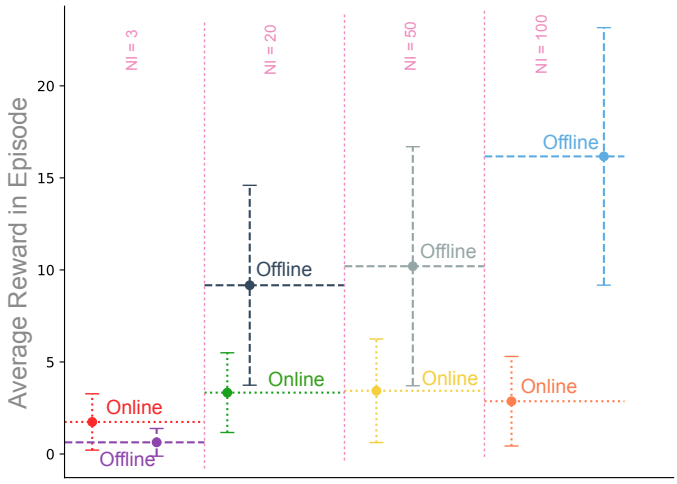

Figure 5.7: Comparison of UA-MCTS offline and online scenario in Breakout with different number of iterations.

the MCTS baseline with the true model or even outperformed it in the Space Invaders environment. We also investigate UA-MCTS's performance with a learned uncertainty function $\hat{U}$ for three different sizes of training buffer in the online scenario. We showed that UA-MCTS cannot achieve its true potential with $\hat{U}$ insufficiently trained, but still outperforms the MCTS baseline.

[1]

[1]Implementation of these ideas can be found in: `https://github.com/ualberta-mueller-group/imperfect_model_code`

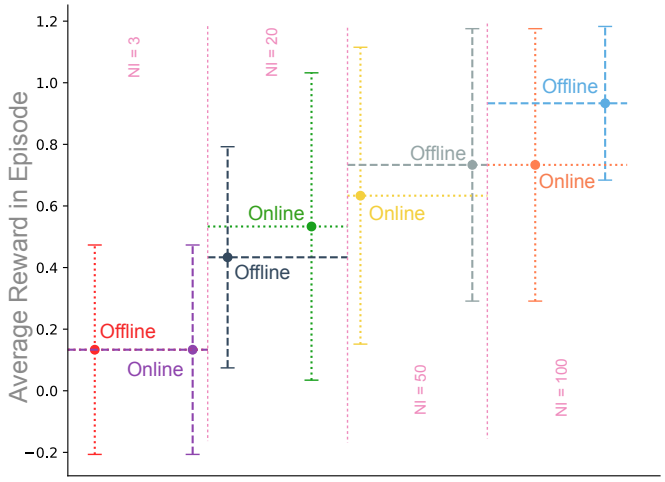

Figure 5.8: Comparison of UA-MCTS offline and online scenario in Freeway with different number of iterations.

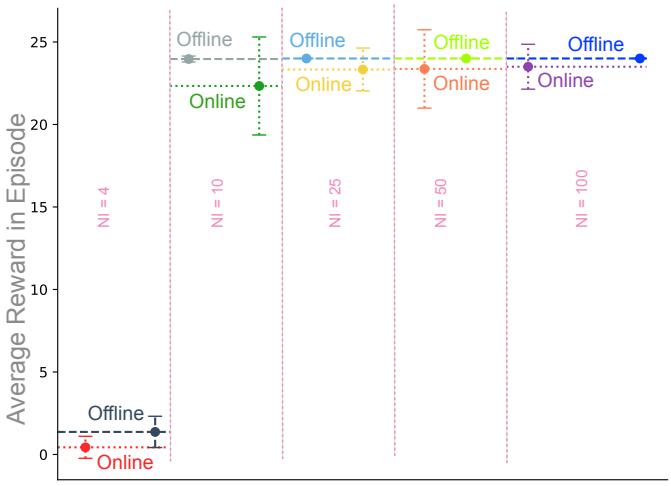

Figure 5.9: Comparison of UA-MCTS offline and online scenario in Space Invaders with different number of iterations.

Chapter 6

Conclusion and Future Work

In this thesis, we investigated the effect of model inaccuracies on the performance of MCTS. We modified three deterministic MinAtar environments in a specific way to induce model error motivated by robotic tasks. We show that deeper simulations in a wrong model do not improve the performance and in some cases can also reduce the performance of MCTS.

To deal with this error we first proposed the DQMCTS method, which uses a learned DQN value function as a heuristic to evaluate the leaf nodes of the search tree. We empirically show that DQMCTS outperforms MCTS baselines and performs best with lower simulation depth. However, there is still room for improvement.

Our second method UA-MCTS deals with model error by changing the behavior of all four components of MCTS, to focus the search more on certain parts of the model. We empirically show that with an accurate measure U of uncertainty, UA-MCTS can achieve the performance of a MCTS agent that has access to the true dynamics. Even with a poorly trained uncertainty measure $\hat{U}$, UA-MCTS still performs better than the MCTS baseline with a corrupted model.

There are several future directions that are worthwhile investigating:

- In our work we did not focus on the amount of model error. Investigating the amount of model error on the performance of MCTS, DQMCTS, and UA-MCTS is an interesting question to explore.

- The precision of the uncertainty model has a lot of influence on the per-

formance of UA-MCTS. Thus, investigating different uncertainty measures or finding ways to train uncertainty more accurately is another future step for our work.

- Creating a more general and online framework that can learn the uncertainty and value functions while interacting with the environment and uses them in a combination of UA-MCTS and DQMCTS is the most straight forward follow up to our work which we would very much like to pursue.

References

[1] Z. Abbas, S. Sokota, E. Talvitie, and M. White, "Selective Dyna-style planning under limited model capacity," in *International Conference on Machine Learning*, PMLR, 2020, pp. 1–10.

[2] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo tree search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, 2010.

[3] K. Asadi, K. Evan, D. Misra, and M. L. Littman, "Towards a Simple Approach to Multi-step Model-based Reinforcement Learning," *NeurIPS 2019 Deep Reinforcement Learning Workshop*, 2019.

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

[5] 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*, vol. 4, no. 1, pp. 1–43, 2012.

[6] J. Buckman, D. Hafner, G. Tucker, E. Brevdo, and H. Lee, "Sample-efficient Reinforcement Learning with stochastic ensemble value expansion," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, Montréal, Canada: Curran Associates Inc., 2018, pp. 8234–8244.

[7] T. Cazenave, "Reflexive Monte-Carlo search," in *Computer Games Workshop*, Citeseer, 2007, pp. 165–173.

[8] ——, "Nested Monte-Carlo search," in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009, pp. 456–461.

[9] ——, "Nested Monte-Carlo expression discovery," in *ECAI*, IOS Press, 2010, pp. 1057–1058.

[10] T. Cazenave and A. Saffidine, "Monte-Carlo Hex," in *Proc. Board Games Studies Colloq., Paris, France*, Citeseer, 2010.

[11] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo tree search: A new framework for game AI.," *AIIDE*, vol. 8, pp. 216–217, 2008.

[12] V. Chelu, D. Precup, and H. P. van Hasselt, "Forethought and hindsight in credit assignment," *Advances in Neural Information Processing Systems*, vol. 33, pp. 2270–2281, 2020.

[13] M. Enzenberger, M. Müller, B. Arneson, and R. Segal, "Fuego—an open-source framework for board games and Go engine based on Monte Carlo tree search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 259–270, 2010.

[14] V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine, "Model-based value expansion for efficient model-free Reinforcement Learning," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

[15] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *International Conference on Machine Learning*, PMLR, 2016, pp. 1050–1059.

[16] H. P. van Hasselt, M. Hessel, and J. Aslanides, "When to use parametric models in Reinforcement Learning?" *Advances in Neural Information Processing Systems*, vol. 32, pp. 14 322–14 333, 2019.

[17] G. E. Hinton and D. Van Camp, "Keeping the neural networks simple by minimizing the description length of the weights," in *Proceedings of the sixth annual conference on Computational Learning Theory*, 1993, pp. 5–13.

[18] T. Jafferjee, E. Imani, E. Talvitie, M. White, and M. Bowling, "Hallucinating value: A pitfall of dyna-style planning with imperfect environment models," *arXiv preprint arXiv:2006.04363*, 2020.

[19] M. Janner, J. Fu, M. Zhang, and S. Levine, "When to trust your model: Model-based policy optimization," *Advances in Neural Information Processing Systems*, vol. 32, pp. 12 519–12 530, 2019.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR*, 2015.

[21] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved Monte-Carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[22] F. Kohankhaki, "Monte Carlo tree search in the presence of model uncertainty," MSc Thesis, University of Alberta, 2022.

[23] H. Lai, J. Shen, W. Zhang, and Y. Yu, "Bidirectional model-based policy optimization," in *International Conference on Machine Learning*, PMLR, 2020, pp. 5618–5627.

[24] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[25] J. Lu, X. Wang, D. Wang, and Y. Wang, "Parallel Monte Carlo tree search in perfect information game with chance," in *Chinese Control and Decision Conference (CCDC)*, IEEE, 2016, pp. 5050–5053.

[26] B. Lütjens, M. Everett, and J. P. How, "Safe Reinforcement Learning with model uncertainty estimates," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 8662–8668.

[27] C. Mansley, A. Weinstein, and M. Littman, "Sample-based planning for continuous action Markov Decision Processes," in *Twenty-First International Conference on Automated Planning and Scheduling*, 2011.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep Reinforcement Learning," in *NIPS Deep Learning Workshop*, 2013.

[29] H. Nakhost and M. Müller, "Monte-Carlo exploration for deterministic planning," in *Twenty-First International Joint Conference on Artificial Intelligence*, Citeseer, 2009.

[30] J. P. A. Nijssen and M. H. Winands, "Enhancements for multi-player Monte-Carlo tree search," in *International Conference on Computers and Games*, Springer, 2010, pp. 238–249.

[31] J. Nijssen, "Playing Othello using Monte Carlo," *Strategies*, pp. 1–9, 2007.

[32] D. A. Nix and A. S. Weigend, "Estimating the mean and variance of the target probability distribution," in *Proceedings of IEEE International Conference on Neural Networks (ICNN)*, IEEE, vol. 1, 1994, pp. 55–60.

[33] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani, "An Othello evaluation function based on Temporal Difference Learning using probability of winning," in *IEEE Symposium On Computational Intelligence and Games*, IEEE, 2008, pp. 205–211.

[34] A. Sabharwal, H. Samulowitz, and C. Reddy, "Guiding combinatorial optimization with UCT," in *International conference on integration of artificial intelligence (AI) and operations research (OR) techniques in constraint programming*, Springer, 2012, pp. 356–361.

[35] A. Saffidine, "Utilisation d'UCT au Hex," *Ecole Normale Super. Lyon, France, Tech. Rep*, 2008.

[36] Y. Sato, D. Takahashi, and R. Grimbergen, "A Shogi program based on Monte-Carlo tree search," *ICGA Journal*, vol. 33, no. 2, pp. 80–92, 2010.

[37] K. Shibahara and Y. Kotani, "Combining final score with winning percentage by sigmoid function in Monte-Carlo simulations," in *IEEE Symposium On Computational Intelligence and Games*, IEEE, 2008, pp. 183–190.

[38] D. Silver, H. Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto, *et al.*, "The Predictron: End-to-end learning and planning," in *International Conference on Machine Learning*, PMLR, 2017, pp. 3191–3199.

[39] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[40] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[41] D. Silver, R. S. Sutton, and M. Müller, "Sample-based learning and search with permanent and transient memories," in *Proceedings of the 25th International Conference on Machine learning*, 2008, pp. 968–975.

[42] W. Sun, N. Jiang, A. Krishnamurthy, A. Agarwal, and J. Langford, "Model-based RL in contextual decision processes: Pac bounds and exponential improvements over model-free approaches," in *Conference on Learning Theory*, PMLR, 2019, pp. 2898–2933.

[43] R. S. Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," in *Machine Learning proceedings*, Elsevier, 1990, pp. 216–224.

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

[45] R. S. Sutton, C. Szepesvári, A. Geramifard, and M. Bowling, "Dyna-style planning with linear function approximation and prioritized sweeping," in *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'08, AUAI Press, 2008, pp. 528–536, ISBN: 0974903949.

[46] E. Talvitie, "Self-correcting models for model-based Reinforcement Learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[47] Y. Tanabe, K. Yoshizoe, and H. Imai, "A study on security evaluation methodology for image-based biometrics authentication systems," in *IEEE 3rd International Conference on Biometrics: Theory, Applications, and Systems*, IEEE, 2009, pp. 1–6.

[48] B. K.-B. Tong, C. M. Ma, and C. W. Sung, "A Monte-Carlo approach for the endgame of Ms. Pac-Man," in *IEEE Conference on Computational Intelligence and Games (CIG'11)*, IEEE, 2011, pp. 9–15.

[49] B. K.-B. Tong and C. W. Sung, "A Monte-Carlo approach for ghost avoidance in the Ms. Pac-Man game," in *2nd International IEEE Consumer Electronics Society's Games Innovations Conference*, IEEE, 2010, pp. 1–8.

[50] A. Vemula, J. A. Bagnell, and M. Likhachev, "CMAX++: Leveraging experience in planning and execution using inaccurate models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 6147–6155.

[51] A. Vemula, Y. Oza, J. Bagnell, and M. Likhachev, "Planning and Execution using Inaccurate Models with Provable Guarantees," in *Proceedings of Robotics: Science and Systems*, Corvalis, Oregon, USA, Jul. 2020.

[52] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[53] C. Xiao, Y. Wu, C. Ma, D. Schuurmans, and M. Müller, "Learning to combat compounding-error in model-based Reinforcement Learning," *NeurIPS 2019 Deep Reinforcement Learning Workshop*, 2019.

[54] K. Young and T. Tian, "MinAtar: An Atari-inspired testbed for thorough and reproducible Reinforcement Learning experiments," *arXiv preprint arXiv:1903.03176*, 2019.