Monte Carlo Tree Search and Model Uncertainty

by

Farnaz Kohankhaki

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 popular tree search framework for choosing actions in decision-making problems. MCTS is traditionally applied to applications in which a perfect simulation model is available. However, when the model is imperfect, the performance of MCTS drops heavily.

In this work, we introduce the Uncertainty Adapted MCTS (UA-MCTS) framework; an adaptation of the MCTS framework to model uncertainty. We define model uncertainty as the difference between the actual environment and the imperfect model. In UA-MCTS we modify each of the 4 steps selection, expansion, simulation, and backpropagation in MCTS so that they consider uncertainty. Although we provide a method to learn the uncertainty of the model, UA-MCTS is not restricted to our specific learning method.

In the Reinforcement Learning (RL) domain, we propose the DQ-MCTS framework. DQ-MCTS uses the learned values from DQN, a state of the art model-free RL method, to improve MCTS performance. Since DQN is a model-free method, the errors in the model do not affect the learned values. DQ-MCTS uses DQN learned values to initialize the newly added nodes in the expansion step and to evaluate the last states in the simulation step.

We experimentally evaluate UA-MCTS and DQ-MCTS on the deterministic domains from the MinAtar test suite. Our results demonstrate that UA-MCTS strongly improves MCTS in the presence of model error, and that DQ-MCTS can perform better than MCTS but not better than DQN.

Preface

The methods and results of herein were developed in close collaboration with Kiarash Aghakasiri.

To Pouneh Gorji and Arash Pourzarabi,
Wonderful, kind, and unforgettable souls.

Success is walking from failure to failure with no loss of enthusiasm.
(Like Research)

– Winston Churchill.

Acknowledgements

I would like to begin by thanking my supervisor Martin Müller for his guidance, kindness, and inspiration. This thesis would have never been possible without your endless support and patience.

I extend my gratitude to Huawei Technologies for providing financial support for this work. Additionally, I would like to thank Ting Han for his invaluable advice and constructive suggestions, and Chao Gao for his insights throughout this project.

To my friend and colleague, Kiarash Aghakasiri, it has been an absolute pleasure working alongside you on this project.

My mother, father, and brother, thank you for being with me through this stage of my life. None of my life's achievements would have been possible without your love and continuous encouragement. I am very fortunate to have you.

Everyone who was there for me, I love you all.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Monte Carlo Tree Search (MCTS) [6] framework is a search-based method for decision-making problems. MCTS is best known for its achievements in computer Go [19]. Moreover, MCTS has significant success in other game applications such as Hex [4, 17], Shogi [18], and Poker [15] and in non-game applications such as video parsing [3], harvest scheduling [14], robot controllers [7], cost evaluation of polynomials [11], function approximation [16], and the Travelling Salesman problem [5]. In all of these applications, MCTS has access to a perfect simulation model in which search steps can be efficiently performed. However, in many practical applications, a perfect model is not available and MCTS has only access to an imperfect model. Yet such a model can still be useful.

While previous approaches to using search with imperfect models exist [21, 22], surprisingly, to the best of our knowledge, there is no prior work that directly adapts MCTS to deal with imperfect models. In this thesis, we adapt MCTS to this setting.

When MCTS has an imperfect model, its performance drops heavily. We show this in our experiments in chapter 3. To overcome this, we propose Uncertainty Adapted MCTS (UA-MCTS) as an improved version of the MCTS framework that directly considers the uncertainty in imperfect models in the MCTS algorithm. We define uncertainty as a measure of the distance between the perfect model and the imperfect model that is available to the search algorithm. We propose a method to learn the uncertainty of an imperfect

model. But UA-MCTS is not restricted to this specific method for learning the uncertainty. UA-MCTS is a general framework and it can use any other method that captures the uncertainty.

The problem of imperfect models has been addressed in the Model-Based Reinforcement Learning (MBRL) literature. In MBRL, the model of the environment can be given but is imperfect, or it may not be given at all, in which case the agent needs to learn the model itself. Due to the compounding error phenomenon in imperfect models, MBRL agents can fail catastrophically [25]. Abbas et al. [1] proposed a method to use an imperfect model selectively.

Model-free RL methods do not use a model for learning, they only use real interaction with the environment. Therefore, the errors in the model do not affect the values learned by these methods. Such learned values can be used as a heuristic in the MCTS framework when the model is imperfect. In this thesis, we propose the DQ-MCTS framework which uses the learned values from a pretrained DQN [13] to improve MCTS performance when the model is imperfect.

In Chapter 2 we provide the background for our work. In Chapters 3 and 4, we present the details of our UA-MCTS and UA-DQN frameworks and also the details and results of our experiments. We test the performance of UA-MCTS and DQ-MCTS by running experiments using three deterministic MinAtar environments [26]. In Chapter 5 we provide a conclusion and directions for possible future work.

The methods and results in this work were developed in close collaboration with Kiarash Aghakasiri. UA-Expansion, UA-Simulation, and DQ-Simulation methods are developed by myself. Kiarash Aghakasiri developed UA-Simulation, UA-Backpropagation, and DQ-Expansion. We both discussed the details and results of these methods together. All other ideas and implementations are equal collaborated work. The methods mentioned here are described in Chapters 3 and 4.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning (RL) [20] is learning from interactions with an environment to maximize a numerical signal, called reward. An agent must choose actions in the environment, and observe the immediate reward and the changed situation.

2.1.1 Markov Decision Process

$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \cdots$.

An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$ where:

- $\mathcal{S}$ is the set of states. A state is a representation of the environment.
- $\mathcal{A}$ is the set of actions.
- $\mathcal{R}$ is the set of rewards. Each reward is a numerical signal.
- p is the *dynamics* of the MDP: $p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$. It defines the probability of both the reward R_t and the next state S_t based only on the current state S_{t-1} and action A_{t-1}.
- γ is the discount rate and $0 \leq \gamma \leq 1$. It determines how much the future rewards are important to the current state.

In a finite MDP, $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are finite sets.

In *deterministic* MDP, for all state-action pairs (s, a) there is only one next state s' and reward r, so $p(s', r|s, a) = 1$.

2.1.2 Episodes, Returns, and Policies

In general, there are two types of tasks in RL: *episodic* and *continuing* tasks [20]. In episodic tasks, the agent's interactions with the environment break into subsequences called *episodes*. Each episode ends in a *terminal* state; a special state that resets the agent's state to one of the starting states with a zero reward. In continuing tasks there is no terminal state and the agent interacts with the environment forever.

The goal of an RL agent is to maximize the expected *return* after time step t which is defined as below:

$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

For episodic tasks, $R_T, R_{T+1}, R_{T+2}, \cdots$ are all zero where T is the time step when the agent reached a terminal step.

A *policy* maps each state s in the state space to a probability $\pi(s, a)$ for each action a in the action space.

2.1.3 Value Functions and Bellman Equation

The value of a state s under policy π is denoted as $v_\pi(s)$ and defined as:

$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$

v_π is called the *state-value* function [20]. The value of terminal states is always zero. The *Bellman equation* for a state-value function v_π indicates the relationship between the value of each state and its successor states and is defined as follows [20]:

$$\forall s \in S, v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.1}$$

The value of taking action a in state s and then following the policy π is denoted as $q(s, a)$ and defined as:

$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$

This is called the *action-value* function [20]. The Bellman equation for $q_\pi(s, a)$ indicates the relationship between the value of each state-action and

its successor state-actions. It is defined as follows [20].

$$\forall s \in S, \forall a \in A, q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')] \quad (2.2)$$

Every MDP has an *optimal* policy [20]. The optimal state-value function v_* corresponds to an optimal policy. For an optimal policy π_*, for all states s and all other policies π, $v_*(s) \geq v_\pi(s)$. Similarly, q_* is the optimal action-value function corresponding to the optimal policy. The *Bellman optimality equations* for v_* and q_* are defined as follows:

$$\forall s \in S, v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$$
$$\forall s \in S, \forall a \in A, q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \max_{a'} q_*(s',a')]$$

2.1.4 Reinforcement Learning Methods

Solving the reinforcement learning task means finding an optimal policy, where the agent chooses actions that result in maximum cumulative rewards. Reinforcement learning methods can be categorized into two general cases: *Model-based* and *model-free* [20]. In model-based methods such as *dynamic programming* (DP), it is assumed the agent has access to the model of the environment. The model is a prediction of the transition dynamics. In model-free reinforcement learning methods, such as *Monte Carlo* methods and *temporal difference* (TD) learning, the agent assumes that it does not have access to the model of the environment and it can only learn from interaction with the environment.

2.1.5 Model-Based Reinforcement Learning

Model-based reinforcement learning methods mostly use *planning*: the computation process that uses transition dynamics to evaluate and improve policy [20]. Model-free reinforcement *learning* methods rely on learning: the computation process that does not use the transition dynamics to evaluate and improve policy [20]. In this thesis, we denote the transition dynamics for any state-action (s,a) as $M(s,a)$ which outputs one possible next state-reward pair (s',r) based on p. In the deterministic MDP case, $M(s,a)$ outputs exactly one next state-reward pair.

Dynamic programming is a model-based reinforcement learning method. In DP, the agent finds the optimal policy by doing iterations of *policy evaluation* and *policy improvement* steps. This process is called *policy iteration*.

In policy evaluation, the agent uses equation 2.2 to find the value function of policy π. The agent starts with a random estimate of the action-value function denoted as $Q(s, a)$. Then, it updates $Q(s, a)$ for all the state-actions using the following formula until $Q(s, a)$ does not change anymore for any state-action.

$Q_\pi(s, a) \leftarrow \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s')Q_\pi(s', a')]$

When $Q(s, a)$ does not change anymore, the value of the policy π has been found because it satisfies the Bellman equation. The above update is called *full backup*. It requires full access to the transition dynamics p.

In policy improvement, a new policy π' is achieved by choosing the greedy policy with respect to the current estimated value function Q:

$\pi'(s) = \underset{a}{\text{argmax}} Q_\pi(s, a)$

Starting with an arbitrary policy π_0, using policy evaluation the agent finds the estimated value function Q_{π_0} for π_0. Then using policy improvement, the agent achieves π_1. Again, by using policy evaluation, the agent finds Q_{π_1}, and so on until the policy cannot be improved. The resulting policy is optimal. Below the sequence of obtained policy and action-value functions is demonstrated.

$\pi_0 \xrightarrow{\text{E}} Q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} Q_{\pi_1} \xrightarrow{\text{I}} \cdots \xrightarrow{\text{I}} Q_{\pi_*} \xrightarrow{\text{E}} \pi_*$

$\xrightarrow{\text{E}}$ and $\xrightarrow{\text{I}}$ denote the policy evaluation and policy improvement steps respectively.

2.1.6 Model-Free Reinforcement Learning

Temporal difference (TD) methods are model-free methods for evaluating the value function. TD learning does not need access to the transition dynamics. It updates the value function from the samples it obtained by interacting with the environment. Below is the update formula for TD learning:

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$

Generalized policy iteration (GPI) is a more general approach to value iteration [20]. In GPI, the value function does not need to be evaluated completely in each iteration; it just needs to get updated. Moreover, a complete policy improvement is not needed in each iteration; a partial improvement is enough. The combination of TD methods for learning the value function and using an ϵ-*greedy* policy is a kind of GPI. An ϵ-greedy policy chooses the optimal action based on the current estimation of value function with probability $1 - \epsilon$, and chooses a random action with probability ϵ.

Both DP and TD methods use a technique called *bootstrapping*; estimating the value of a state or state-action based on the value of their successors. $Q - learning$ [24] is a TD learning method combined with the ϵ-greedy policy that tries to update the action-value function by the formula:

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$

Here S_t, A_t, R_{t+1} and S_{t+1} are obtained from interactions with the real environment and α is a step size parameter.

2.1.7 Function Approximation and Deep Q-Networks

When the size of the state space S or the state-action space $S \times A$ is too large, there is not enough memory to store all the state or state-action values. In this case, RL methods use function approximation to estimate a value or action-value function.

A Deep Q-Network (DQN) [13] is a variant of Q-learning with function approximation. It uses a convolutional neural network to approximate the action-value function.

DQN training uses two techniques for stability: *experience replay* and *target network*. In experience replay, the agent uses a replay buffer B to store transitions. For each update, the agent gets a sample batch from the replay buffer B. Using the replay buffer eliminates the correlations between the transitions since the samples are random. Moreover, using a batch of samples, results in a smooth gradient update. The target network is used to compute the estimated value of the successors. It is an older version of the action-value function network and is not updated after each step for stability. It updates

periodically with a frequency F to the current learned action-value function.

After each interaction with the environment, the agent gets a batch of transitions from the replay buffer B and updates the neural network with stochastic gradient descent using the following formula:

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \alpha[R_{t+1} + \gamma \max_a Q^-(S_t, a, \mathbf{w}^-) - Q(S_t, A_t, \mathbf{w_{t+1}})]\nabla Q(S_t, A_t, \mathbf{w_t}) \tag{2.3}$$

Here, Q is the current action-value function network, Q^- is the target action-value function network, $\mathbf{w}$ is the weight vector of Q, and $\mathbf{w}^-$ is the weight vector of Q^-.

Algorithm 1 shows pseudo code for the DQN algorithm.

2.1.8 Model Learning and Uncertainty

An RL agent may or may not have access to the true model of the environment M. The model that the agent is using is denoted as $\hat{M}$. $\hat{M}$ can be perfect or it can have some errors and be imperfect. The agent can learn a model by itself from interacting with the environment. Learning the model of the environment can be viewed as a supervised learning task. The inputs are the current state and action and the outputs are the next state and reward.

There are generally three types of models of the environment: 1) Distribution models, 2) Sample models, and 3) Expectation models [23]. Given a state and action, each of these types of models predicts a different type of output. Distribution models predict the distribution of the possible next states and rewards. Sample models predict one of the possible next states and the associated reward. Expectation models predict an expectation over the possible next states and rewards. Learning an expectation model is a much easier choice for an RL agent especially when the size of the state or state-action space is large. Neural networks are a common choice for learning such models.

A learned or given model $\hat{M}$ may be imperfect and have some errors. We define the function *true uncertainty* $U(s, a)$ for each state-action pair (s, a) as the squared difference between vectors representing the predictions made by

$\hat{M}$ and M. These differences are caused by three main sources of predictive uncertainty in the model [1]:

- Aleatoric uncertainty: This uncertainty is caused by the stochasticity in the transition dynamics. If there are multiple next states and rewards for a state-action, then the expectation model will learn an expectation of these next states and rewards and has aleatoric uncertainty. Aleatoric uncertainty is irreducible because it cannot be reduced by increasing the capacity of the model or by gathering more data.

- Parameter uncertainty: Given a dataset and a hypothesis class for the model, there might be multiple choices for the best parameter set that describe the data. The uncertainty about which of these parameter sets generated the data is parameter uncertainty. Parameter uncertainty can be reduced by gathering more data.

- Structure uncertainty: This type of uncertainty occurs when the chosen model class does not have the capability to learn the observed data. This uncertainty can be reduced by increasing the capacity of the model.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) approaches sequential decision-making problems by selective lookahead search. [6]. MCTS has shown great performance in stochastic and deterministic games such as Hex [4, 17], Shogi [18], and Poker [15], and in real-world planning [3, 7, 14] and optimization [8, 11, 12] problems. MCTS first became popular for its application in computer Go [19]. Before MCTS, Go was one of the few classical board games where computers could not achieve strong human-level performance.

Monte Carlo methods are a class of methods that approximate a value by taking random samples in the action space. Monte Carlo methods can be used to approximate the action-value function $q(s, a)$ by getting the mean of returns of random samples from that state-action:

$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i$

Here, $Q(s,a)$ is the estimated action-value function, $N(s,a)$ is the number of times that action a is selected from state s, $N(s)$ is the number of simulations from state s, z_i is the result of the ith simulation, and $\mathbb{I}_i(s,a)$ is 1 if action a was selected from state s in the ith simulation and 0 otherwise.

MCTS uses the Monte Carlo method to approximate the value of state-action pairs in a MDP. This algorithm builds a tree incrementally until it reaches the limits of the computational budget such as time or memory. With more computation, more accurate values are obtained. Each node in this search tree represents a state in S, and each edge represents an action from A from a state. MCTS predicts the value of each edge by using Monte Carlo methods.

One iteration of MCTS consists of of 4 steps: Selection, Expansion, Simulation, and Backpropagation.

1. Selection: The method chooses a child recursively starting from the root of the tree based on the *tree policy*, until it reaches a terminal or expandable node. The tree policy chooses among the children of a node within the tree. An expandable node is a node which has unvisited children.

2. Expansion: If the selected node is expandable and nonterminal, all of its children are added to the tree.

3. Simulation: One or more simulations are run from the newly added node based on the *default policy*. We denote a single simulation as a rollout. This is the Monte Carlo sampling part of MCTS. A rollout will stop when it reaches a terminal node, or after a predefined maximum depth. The default policy produces the actions in the rollout. The rollout is used to find the first estimated value for a newly added node. An average of all rollouts outcomes is returned as the outcome of the simulation step.

4. Backpropagation: The results of the simulation are backed up through the path of selected nodes towards the root.

One of the main questions about MCTS is whether it can find the optimal action. The UCT method [10] extended the UCB algorithm in bandit problems to MCTS. UCT introduces a tree policy which balances exploitation and exploration. Exploitation means searching in parts of the tree with higher estimated values to achieve a more accurate value. Exploration means searching in parts of the tree that have not been visited enough to reduce uncertainty and make sure that any promising path has not been missed. The UCT value for node i defined as:

$UCT(v) = \overline{X(v)} + 2c\sqrt{\frac{2\ln N(Par(v))}{N(v)}}$

where c is the exploration constant, $Par(i)$ is the parent of node i in the tree, and $N(i)$ is the number of times node i has been visited. $\overline{X(i)}$ is the approximated value of node i and is equal to $Q(i)/N(i)$. $Q(i)$ is the sum of the rewards for all simulations containing node i.

The probability that UCT chooses the optimal action converges to 1 in the limit [10].

One of the main advantages of MCTS is it does not need any domain-specific knowledge except the value of terminal states which are known in RL environments. MCTS can be used by an RL agent for planning at decision time. When MCTS is used in an RL domain, it needs full access to the transition dynamics.

MCTS with UCT is described in Algorithm 2. In this thesis, we use the following notation for node v in the search tree T:

- Each node in T represents a state. $S(v)$ is the feature vector of the state of the node v.
- Each node in T (except the *root* node) has a parent. $Par(v)$ is the parent of v in T and $Par(root) = NULL$.
- $R(v)$ is the reward observed when adding node v to the tree.
- Each node v has a set $Ch(v)$ of children in T.
- $N(v)$ is the number of times v has been visited throughout the search.

- $Q(v)$ is the sum of rewards observed at v.
- N_I is the total number of iterations of the MCTS main loop.
- N_S is the number total of simulations used to evaluate a leaf node.
- D_S is the maximum depth of each rollout.

Algorithm 1 DQN Algorithm.

Initialize a replay buffer B of size N
Initialize an action-value function Q with random weights $\mathbf{w_0}$
Initialize Q^- as a copy of Q with weights $\mathbf{w}^- = \mathbf{w_0}$
counter $\leftarrow 0$
for *episode* $\leftarrow 1$ to E **do**
  $s_1 \leftarrow$ initial state of the environment
  for t $\leftarrow 1$ to T **do**
    rand $\leftarrow$ a random number between 0 and 1
    if *rand* $< \epsilon$ **then**
      $a_t \leftarrow$ a random action
    else
      $a_t \leftarrow \max_a Q^-(s_t, a, w^-)$
    end if
    $s_{t+1}, r_{t+1} \leftarrow$ execute action a_t
    store transition $(s_t, a_t, r_{t+1}s_{t+1})$ to buffer B
    $(s_i, a_i, r_{i+1}, s_{i+1}) \leftarrow$ a random sample mini batch from B
    if s_{i+1} is terminal **then**
      $y_i \leftarrow r_{i+1}$
    else
      $y_i \leftarrow r_{i+1} + \max_{a'} Q^-(s_{i+1}, a', \mathbf{w}^-)$
    end if
    perform a gradient descent update on $(y_i - Q(s_i, a_i, \mathbf{w_t}))^2$ and update $\mathbf{w_t}$ to $\mathbf{w_{t+1}}$ based on update formula 2.3
    counter $\leftarrow$ *counter* $+ 1$
    if *counter* $= F$ **then**
      $\mathbf{w}^- \leftarrow \mathbf{w_{t+1}}$
      counter $\leftarrow 0$
    end if
  end for
  $\mathbf{w_0} \leftarrow \mathbf{w_T}$

Algorithm 2 MCTS Algorithm with UCT.

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)
    end if
    $value \leftarrow$ SIMULATE($S(v_s)$)
    BACKPROPAGATE(v_s, $value$)
  end for
  $v_{best} \leftarrow$ choose the most visited child of v_0
  return $action(v_{best})$
end function=0

Algorithm 3 MCTS Selection Algorithm.

function SELECT(v)
  while v is expanded **do**
    $v \leftarrow \underset{v_i \in Ch(v)}{\text{argmax}} \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\ln N(v)}{N(v_i)}}$
  end while
  return v
end function

Algorithm 4 MCTS 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
    $N(v_i) \leftarrow 0$
    $Q(v_i) \leftarrow 0$
  end for
  return a random child of v
end function

Algorithm 5 MCTS Simulation Algorithm.

function SIMULATE($s, depth$)
  for $i \leftarrow 1$ to N_S **do**
    $g_i \leftarrow$ ROLLOUT(s)
    $\alpha_i \leftarrow 1/\mathrm{N}_S$
  end for
  return $\sum_{i=1}^{N_S} \alpha_i \cdot g_i$
end function

function ROLLOUT(s)
  $count \leftarrow 0$
  $rewards \leftarrow 0$
  $discount \leftarrow 1$
  while s is not terminal & $count < D_S$ **do**
    choose a random action a from $\mathcal{A}$
    $s, r \leftarrow \hat{M}(s, a)$
    $count \leftarrow count + 1$
    $rewards \leftarrow rewards + discount \cdot r$
    $discount \leftarrow discount \cdot \gamma$
  end while
  return $rewards$
end function

Algorithm 6 MCTS Backpropagation Algorithm.

function BACKPROPAGATE(v, $value$)
  while v is not NULL **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \cdot value$
    $value \leftarrow value \cdot \gamma + R(v)$
    $v \leftarrow Par(v)$
  end while
end function

Chapter 3

MCTS and Model Uncertainty

MCTS can be used by AI agents in sequential decision-making problems. However, in its original form, it needs a perfect simulation model. If the agent's model is imperfect, then MCTS does not perform well at all. We show this in Section 3.2. In this chapter, we propose a new framework that adapts the MCTS framework to the case where the agent's model is imperfect. If an agent knows which parts of the model have a higher uncertainty, then one strategy is to behave more conservatively in those parts. In our approach, we discourage searching through the inaccurate parts of the model, where the uncertainty is high, and encourage searching through the more certain parts.

In section 3.1, we describe our approach to adapt the MCTS framework to model uncertainty. In Section 3.2, we describe our modified version of MinAtar environments and our method for learning the model uncertainty, explain our experimental design to investigate UA-MCTS's efficiency, and present the results of the experiments.

3.1 UA-MCTS

How can search in an imperfect model $\hat{M}$ improve decision-making in M? Of course, this task is hopeless if there is no exploitable relation between $\hat{M}$ and M. Vemula et al. study robust robot path planning [21, 22]. In the case where the robot behaves differently from the model while interacting with the real environment, their approach completely disables these parts of the model and replans. For example, a motor may be malfunctioning, or an arm may be

overloaded with too much weight. The algorithms CMAX and CMAX++ plan and learn in a real-time A* search framework.

In our UA-MCTS approach, we adapt MCTS to behave more conservatively in states where the uncertainty U is large. We discourage, but do not completely give up on, searching through the more uncertain parts of the model. One design goal is to prevent inaccuracies from compounding as the search tree grows.

Of course, the true uncertainty U is not available to the agent. However, after each search, one action a is executed from some state s in the real world. The difference between the real next state and the predicted next state by $\hat{M}$ is a sample of U. From these samples, the agent can build and improve an estimate $\hat{U}(s, a)$. In Section 3.2.2 we provide a specific method for learning $\hat{U}$. However, UA-MCTS is a general framework: $\hat{U}$ and its implementation can be chosen freely and is not restricted to our specific learning method.

MCTS iterates 4 steps: selection, expansion, simulation, and backpropagation. We modify each of these 4 steps in UA-MCTS to adapt MCTS to uncertainty. In the following section, we describe these modifications. The MCTS framework remains the same as described in Algorithm 2. We use the same notation we used in section 2.2. Each step uses $\hat{U}$, the uncertainty estimation.

3.1.1 UA-Selection

We changed the selection step of MCTS in a way that children with a higher uncertainty have a lower chance to be selected. We adapt the UCT formula to uncertainty by introducing a new multiplicative term $1-\alpha_v$ in the exploration term. The new UCT formula (UAS-UCT) is shown in equation 3.1. This new term is proportional to the certainty of the children. Therefore children with higher uncertainty will be selected in the selection step with a lower probability. α_v is a softmax of $\hat{U}$ with a temperature parameter τ and is defined in equation 3.2. Algorithm 7 provides a pseudo-code for UA-Selection, with the modified

parts in red. More details are explained in Kiarash [2].

$$UAS - UCT(v) = \frac{Q(v)}{N(v)} + 2c\sqrt{\frac{2\ln N(Par(v))}{N(v)}} \times (1 - \alpha_v) \tag{3.1}$$

$$\alpha_v \doteq \frac{e^{\hat{U}(v)/\tau}}{\sum_{v_i \in Ch(Par(v))} e^{\hat{U}(v_i)/\tau}} \tag{3.2}$$

Algorithm 7 UA-Selection Algorithm

Parameter: temperature τ for softmax

function UA-SELECT(v)

  while v is expanded **do**

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

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

    end for

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

  end while

  return v

end function

3.1.2 UA-Expansion

We changed the MCTS expansion step so that it discourages adding children with high uncertainty. When expanding node v, we choose one of its children for possible elimination based on a probability function defined for each of the children. The chosen child is eliminated from the tree with a fixed probability τ, a hyperparameter of the search. With probability $1 - \tau$, no child is eliminated. Assume v_i is the ith child of node v. α_i is the corresponding probability of eliminating of ith child node, v_i. It is defined as in equation 3.3. α_i is higher if the child node v_i has a higher uncertainty.

$$\alpha_i \doteq \frac{\hat{U}(v_i)}{\sum_{v_j \in Ch(v)} \hat{U}(v_j)} \tag{3.3}$$

Pruning children with high uncertainty provides a search tree with more certain edges. Therefore, the outcome of the search is more reliable. This method also reduces the branching factor of the tree, which results in a deeper search. Algorithm 8 provides a pseudo-code for UA-Expansion, with the modified parts in red.

Algorithm 8 UA-Expansion Algorithm

Parameter: probability τ for deleting a node

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

    $\hat{U}(v_i) \leftarrow \hat{U}(S(v), a_i)$

    $N(v_i) \leftarrow 0$

    $Q(v_i) \leftarrow 0$

  end for

  $x \leftarrow$ random number $\in (0, 1)$

  if $\sum_{v_j \in Ch(v)} \hat{U}(v_j) > 0$ and $x < \tau$ **then**

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

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

    end for

    choose node v_i with probability α_i

    delete node v_i

  end if

  return a random child of v

end function

3.1.3 UA-Simulation

In MCTS, the final return of the simulation step is the (unweighted) average of the returns of the N_S rollouts. In UA-Simulation we weigh the return of each rollout based on the total uncertainty of its trajectory, and then use a weighted average of all rollouts for the return of the simulation. A rollout

trajectory consisting of many nodes with high uncertainty should have a lower weight. Here, we explain how we calculate an uncertainty estimation and a weight for each rollout.

Assume $\mathcal{T} = \{T_1, T_2, \cdots, T_{N_S}\}$ is the set of N_S rollout trajectories for a simulation and T_i is the ith rollout trajectory. Let σ_i be the uncertainty of a rollout trajectory T_i, denoted as σ_i, with length h and trajectory $(s_1, a_1, s_2, v_2, \cdots, s_h, a_h, s_{h+1})$. σ_i is computed as in equation 3.4. The uncertainty measure σ_i is the sum of discounted uncertainties in the trajectory. The weight α_i for the ith rollout is calculated as a softmax function as in equation 3.5. This weight is higher if the uncertainty of the trajectory T_i is lower.

$$\sigma_i \doteq \sum_{k=1}^{h} \gamma^{k-1} \hat{U}(s_k, a_k) \tag{3.4}$$

$$\alpha_i \doteq e^{-\sigma_i/\tau} \Big/ \sum_{i=1}^{N_S} e^{-\sigma_j/\tau} \tag{3.5}$$

τ in the above equation is the softmax temperature parameter. Equation 3.6 presents how the return of the simulation, G, is computed based on the α_i and g_i, the return of the ith rollout.

$$G \doteq \sum_{i=1}^{N_S} \alpha_i \cdot g_i \tag{3.6}$$

Algorithm 9 provides a pseudo-code for UA-Simulation, with the modified parts in red.

3.1.4 UA-Backpropagation

We changed the backpropagation step in a way that children with high uncertainty have less impact on their parents while backpropagating values. The value of a parent v is updated towards the backpropagated value from child node v_i based on $\hat{U}(v_i)$. If $\hat{U}(v_i)$ is high, then the update has a lower effect on the value of v. Equation 3.7 shows the updated UCT formula (UAB-UCT). α_i in this equation is higher if the uncertainty of child v_i is lower. α_i is defined in

Algorithm 9 UA-Simulation Algorithm

Parameter: temperature τ for softmax

function SIMULATE($s, depth$)

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

    g_i, $\sigma_i \leftarrow$ ROLLOUT(s)

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

  end for

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

    $\alpha_i \leftarrow e^{-\sigma_i/\tau} \Big/ \sum_{i=1}^{N_S} e^{-\sigma_j/\tau}$

  end for

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

end function

function ROLLOUT(s)

  $count \leftarrow 0$

  $rewards \leftarrow 0$

  $\sigma \leftarrow 0$

  $discount \leftarrow 1$

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

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

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

    $count \leftarrow count + 1$

    $rewards \leftarrow rewards + discount \cdot r$

    $\sigma \leftarrow \sigma + discount \cdot \hat{U}(s, a)$

    $discount \leftarrow discount \cdot \gamma$

  end while

  return $rewards$, σ

end function

equation 3.8. Algorithm 10 provides a pseudo-code for UA-Backpropagation, with the modified parts in red. More details are explained in Kiarash [2].

$$UAB - UCT(v) = \frac{\sum_{v_i \in Ch(v)} Q(v_i) \times N(v_i) \times \alpha_i}{\sum_{v_i \in Ch(v)} N(v_i) \times \alpha_i} + 2c\sqrt{\frac{2 \ln N(Par(v))}{N(v)}} \quad (3.7)$$

$$\alpha_i = \frac{e^{-\hat{U}(v_i)}}{\sum_{v_j \in Ch(Par(v_i))} e^{-\hat{U}(v_j)}} \quad (3.8)$$

3.2 Experiments: Design and Results

In this section, we present the details of the experiments we have done to investigate the performance of UA-MCTS. [1] We also investigate the effect of UA-Expansion and UA-Simulation steps independently. We investigate two scenarios:

- Offline scenario: the agent has access to the true uncertainty U of the model, but not to the perfect model M itself. This scenario evaluates the performance of UA-MCTS with a "perfect uncertainty" model. The reason for this experiment is to separate out the difficulty of not knowing M from the extra difficulty due to the training errors in the estimate $\hat{U}$.

- Online scenario: the agent does not have access to U, and therefore has to learn an approximation $\hat{U}$ online from real experience. The agent starts with baseline MCTS without any of the UA adaptations, and collects the buffer B of transitions that is used to train $\hat{U}$. Once this buffer is full, the agent performs a fixed number e of training steps to create $\hat{U}$, and switches over to using UA-MCTS with $\hat{U}$.

The experiments are done on three modified environments in the MinAtar framework [26]: Space Invaders, Freeway and Breakout.

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

	Space Invaders	Freeway	Breakout
N_I	10	100	100
D_S	20	50	50
N_S	10	10	10

Table 3.1: Search hyperparameters for the modified MinAtar environments.

We test a total number of five algorithms. The two baselines use standard MCTS without any UA modifications:

1) "True Model" MCTS is allowed to use M in its search.

2) "Corrupted Model" MCTS uses $\hat{M}$ for all its planning.

3-5) All versions of UA-MCTS work as follows: they use $\hat{M}$ for all their planning. The UA-MCTS versions labeled "UA-Expansion" and "UA-Simulation" use the UA modifications only for this one component of MCTS, and use unmodified MCTS for the other three parts. The "UA-MCTS" version uses all four enhancements.

For each combination of the environment, scenario, and algorithm, we performed a parameter sweep over τ and c from the sets $\tau \in \{0.1, 0.5, 0.9\}$, $c \in \{0.5, 1, \sqrt{2}, 2\}$ respectively for 30 runs. The mean and standard deviation of the best-performing configuration for each experiment is reported as the final result. Table 3.1 presents the hyperparameters used for each environment. For the online scenario and each of these combinations, we used 3 different buffer sizes to investigate the effect of the learned uncertainty $\hat{U}$ on the performance of UA-MCTS and its components. We used 1000, 3000, and 7000 for the buffer size.

In Section 3.2.1 we describe the details of the environments. In Section 3.2.2 we describe our method for learning the uncertainty estimate $\hat{U}$. In Sections 3.2.3 - 3.2.5 we describe the experimental setup and results for each environment. Finally, in Section 3.3 we provide a summary of the results of the experiments.

3.2.1 Environments

We test the UA-MCTS framework on the three deterministic modified games Space Invaders, Freeway and Breakout in the MinAtar framework [26]. We modified the transition dynamics in each of these games, but the agent does not have any information about these modifications. In each game the search agent "believes" it is playing the real game. However, the rules of the game itself have changed, and the agent only learns about this change slowly when it acts in the real environment. More formally, M includes the new modifications, but $\hat{M}$ has no information about them. In the following, we explain the details of each of the games and the way we modified them.

Space Invaders

In Space Invaders, the agent controls a player that tries to eliminate 24 aliens by shooting at them [26]. The player is at the bottom of the screen in any of the positions from 0 to 9. The player can move to the left (action "Left") and right (action "Right"), or it can shoot a bullet upward (action "Shoot"), or do nothing (action "None"). At the start of the game, there are 24 aliens at the top of the screen as a cluster and they move together in the same direction. At each time step, all the aliens move one step to the left (or right), all in the same direction. If one of them hits the wall, then they all switch direction and move one step down. At each time step the bullet shot by the agent moves one step up. If a bullet hits an alien, the alien is removed and the agent gets a reward of +1. The aliens also shoot bullets downward. At each time step, the bullets shot by the aliens move one step downward. If one of these bullets or an alien hits the player, the agent dies and the game terminates. If all the aliens are killed, the game also terminates.

We modified this environment so that executing the action "Shoot" does nothing in five out of ten positions, at index 2, 3, 4, 5, and 6. But, the agent is unaware of its limitations on shooting in these five positions. To have a better performance, the agent should avoid staying in these positions to be able to shoot and kill aliens.

Figure 3.1 presents a snapshot of the Space Invaders environment. [2]

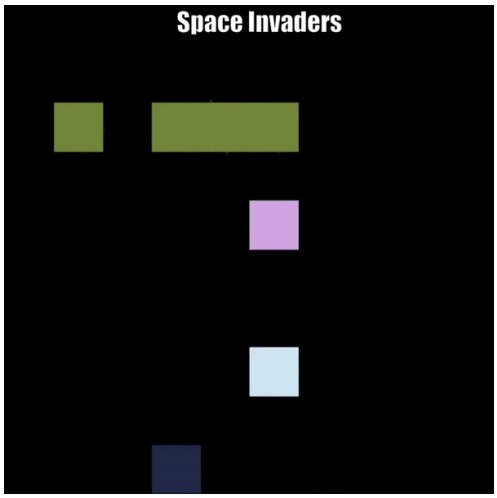


Figure 3.1: A snapshot of the Space Invaders environment. The navy square is the player, the grey square is the aliens' bullet, the pink square is the player bullet, and the green cluster are the aliens.

Freeway

In Freeway, the player's goal is to reach the top of the screen without hitting cars on the way [26]. The player begins at the bottom of the screen. At each time step it can go one step up (action "Up"), down (action "Down"), or do nothing (action "None"). If the player reaches the top of the screen, it gets a reward of +1 and the game terminates. Cars move horizontally on the screen with different speeds. If a car reaches an edge of a screen, it teleports to the other side. If the player is hit by a car, the game terminates with a reward of 0.

We modified this environment so that executing the action "None" moves the agent one step up in six out of ten positions, at index 1, 2, 3, 5, 6, and 7. Therefore, to reach the top of the screen, the agent must plan ahead so that it does not need to stay idle in these positions to not hit by a car. Figure 3.2

[2]The image is from `https://github.com/kenjyoung/MinAtar`.

presents a snapshot of the Freeway environment. [3]

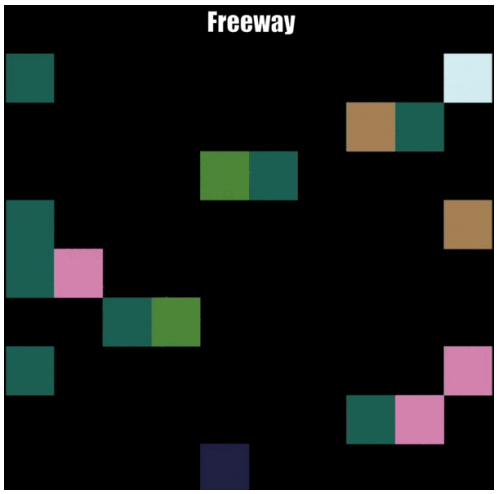


Figure 3.2: A snapshot of the Freeway environment. The navy square is the player, and the other rectangular shapes are the cars.

Breakout

In Breakout, the agent controls a paddle at the bottom of the screen to bounce a ball to break 30 bricks at the top of the screen [26]. The agent can move the paddle to the left (action "Left") and right (action "Right"), or do nothing (action "None"). The ball moves diagonally and based on which side of the paddle it hits, it bounces to the left or right. The agent gets a reward of +1 for each brick broken by the ball. If the ball hits the bottom of the screen or all the bricks are broken, the game terminates.

We modified the game so that the paddle fails to bounce the ball in two out of ten positions, at index 2 and 4, and the game terminates. To overcome this, the agent must plan ahead to keep the ball in play without using these corrupted positions. Figure 3.3 presents a snapshot of the Breakout environment. [4]

[3]The image is from `https://github.com/kenjyoung/MinAtar`.
[4]The image is from `https://github.com/kenjyoung/MinAtar`.

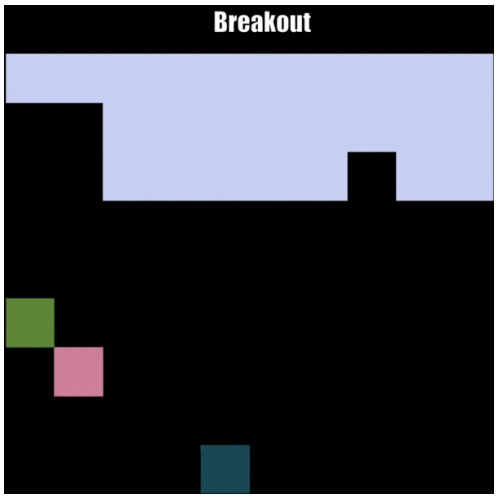


Figure 3.3: A snapshot of the Breakout environment. The navy square is the paddle, the pink-green shape is the ball, and the grey cluster are the bricks.

3.2.2 Learning the Model Uncertainty

In this section, we present our method to estimate the true uncertainty U. In this work, we assume $\hat{M}$, the model that is given to the agent, is fixed. Moreover, we assume there is no error in prediction of the next reward. We try to estimate the uncertainty of the predicted next state by the model of the environment. In the UA-MCTS framework the implementation for estimated uncertainty, $\hat{U}$ does not have any restrictions. Also $\hat{U}$ does not need to be fixed and it can change in each step.

We use an artificial neural network to learn an estimation of U. We collect a buffer B with size β. For each interaction (s, a, r, s') with the environment, we add a sample $\langle s, a, U(s, a)\rangle$ where s, a are the current state and action, and $U(s, a)$ is the squared difference between the vector representing s', the true next state the next state predicted by $\hat{M}$.

$$U(s, a) = \big(\hat{M}(s, a) - M(s, a)\big)^2.$$

When the buffer B is full, we train an artificial neural network with the inputs of (s, a) and the target U for a fixed number e of steps by performing

a gradient descent update with a random batch from buffer B and the loss function L (equation 3.9). After this one-time training, we do not change the uncertainty model $\hat{U}$ anymore for stability reasons.

$$L = \sum_{s,a,U(s,a)\in B'} \left(\hat{U}(s,a) - U(s,a)\right)^2 \tag{3.9}$$

In our experiments, the neural network representing $\hat{U}$ is a fully connected neural network with two hidden layers with 32 units in each. We use a batch size of 128, a number of training steps $e = 5000$, and the Adam optimizer [9] with a step size of 10^{-3}. In the UA-MCTS framework, we define the uncertainty of node v, $\hat{U}(v)$, as the trained uncertainty $\hat{U}(s,a)$ where (s,a) is the transition in the tree leading to node v.

3.2.3 Space Invaders

Offline Scenario

In this section, we present the performance of UA-Expansion, UA-Simulation, and UA-MCTS for the offline scenario in the Space Invaders environment. Figure 3.4 presents the performance of 5 algorithms for this case. The best exploration constant c and temperature parameter τ are provided in Table 3.2.

When MCTS uses an imperfect model (MCTS Corrupted Model in the figures and tables), its performance drops in comparison to the case where it has access to the true model (MCTS True Model). UA-Expansion, UA-Simulation, and UA-MCTS restored the performance MCTS and even performed better than MCTS with the true model. The reason is that the states in which the player cannot shoot have a low value. These states have high uncertainty U and UA-MCTS tends to avoid them. Therefore, with a limited budget for search, Uncertainty Adapted methods performed better than MCTS with the true model.

UA-Simulation perfomed worse than UA-Expansion and UA-MCTS. UA-Expansion and UA-MCTS prevent building a tree within uncertain parts directly. UA-Expansion removes a child with a high uncertainty. However, UA-Simulation only adjusts the outcome of a simulation based on the uncertainty

of the rollouts. It needs to be combined with other components to improve the performance.

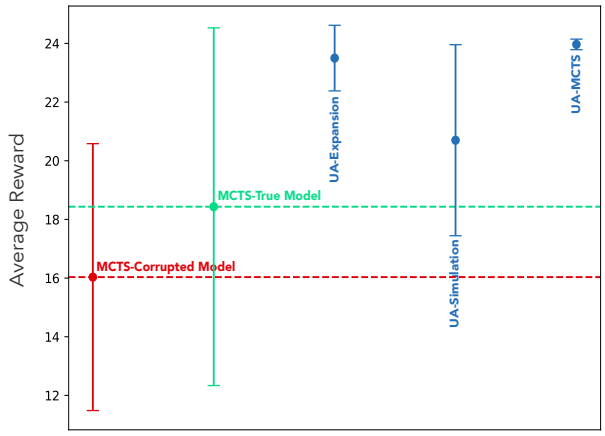


Figure 3.4: Performance of UA-Expansion, UA-Simulation, UA-MCTS, and two MCTS baselines in the Space Invaders environment in the offline scenario. Each error bar shows ± standard deviation.

Online Scenario

Figure 3.5 presents the performance of UA-Expansion, UA-Simulation, and UA-MCTS for different buffer sizes β, as well as MCTS baselines. The best exploration constant c and temperature parameter τ for each combination are provided in Table 3.2.

We observe that UA-Expansion, UA-Simulation, and UA-MCTS performed better than MCTS Corrupted Model and MCTS True Model. The performance of these algorithms is better with the perfect uncertainty (offline scenario). UA-MCTS performed better with a larger buffer size β.

3.2.4 Freeway

Offline Scenario

Figure 3.6 presents the performance of the 5 algorithms in the Freeway environment for the offline scenario. The best exploration constant c and temperature parameter τ are provided in Table 3.3.

MCTS performance degrades when it does not have access to the true model M. UA-MCTS almost achieves the MCTS True Model performance.

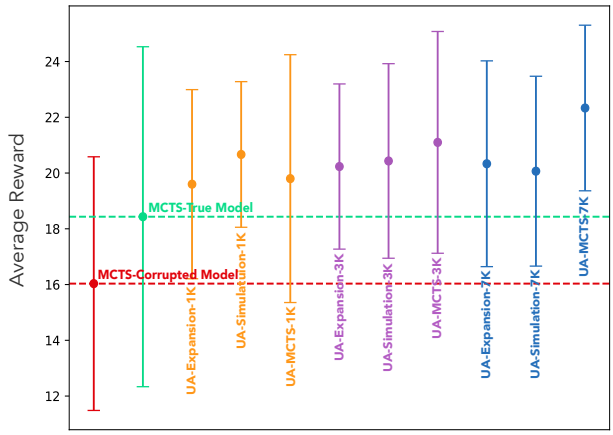


Figure 3.5: Performance of UA-Expansion, UA-Simulation, UA-MCTS, and two MCTS baselines for different βs for the Space Invaders environment in the online scenario. Each error bar shows $\pm$ standard deviation.

	Offline		β =1K		β =3K		β =7K	
	c	τ	c	τ	c	τ	c	τ
UA-Expansion	$\sqrt{2}$	1	1	0.5	0.5	0.5	1	0.1
UA-Simulation	2	0.1	$\sqrt{2}$	0.1	2	0.1	2	0.1
UA-MCTS	1	0.1	1	0.1	1	0.1	0.5	0.1
MCTS Corrupted Model	2	NA	2	NA	2	NA	2	NA
MCTS True Model	0.5	NA	0.5	NA	0.5	NA	0.5	NA

Table 3.2: Best exploration constant c and temperature parameter τ in the Space Invaders environment. The first column presents the best hyperparameters for the offline scenario and the other 3 columns present best hyperparameters for the online scenario.

The reason is that choosing action "None" in the five corrupted positions have a high uncertainty and UA-MCTS tends to this. Moreover, UA-Expansion and UA-Simulation outperform MCTS Corrupted Model.

Online Scenario

Figure 3.7 present the performance of UA-Expansion, UA-Simulation, and UA-MCTS for different βs in the Freeway environment for the online scenario. The best exploration constant c and temperature parameter τ are provided in Table 3.3.

UA-Expansion and UA-MCTS performed better than MCTS Corrupted Model, but they could not achieve MCTS True Model performance. Also,

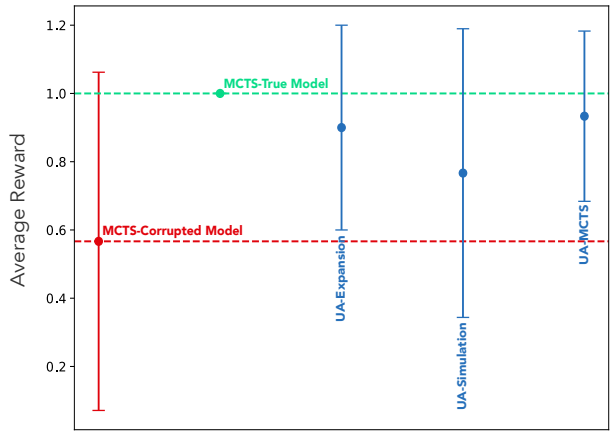


Figure 3.6: Performance of UA-Expansion, UA-Simulation, UA-MCTS, and two MCTS baselines in the Freeway environment in the offline scenario. Each error bar shows ± standard deviation.

their performance did not improve with a higher β. UA-Simulation performed better than MCTS Corrupted Model with $\beta = 1000$ and 7000, but not did it improve the performance with $\beta = 3000$.

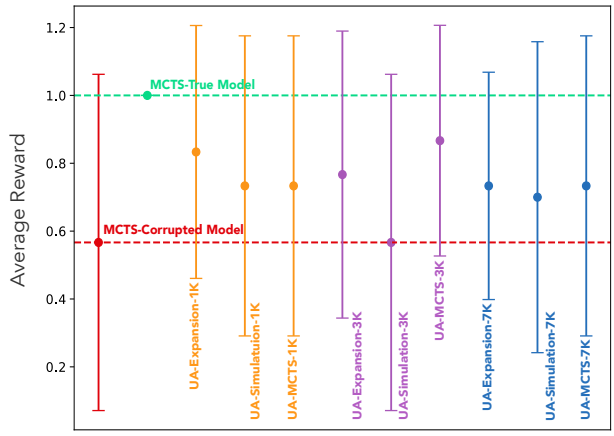


Figure 3.7: Performance of UA-Expansion, UA-Simulation, UA-MCTS, and two MCTS baselines for different βs in the Freeway environment in the online scenario. Each error bar shows ± standard deviation.

3.2.5 Breakout

Offline Scenario

Figure 3.8 presents the performance of MCTS and its components UA-Expansion, and UA-Simulation. The best exploration constant c and temperature param-

	Offline		β =1K		β =3K		β =7K	
	c	τ	c	τ	c	τ	c	τ
UA-Expansion	1	0.1	1	0.5	2	0.1	2	0.1
UA-Simulation	$\sqrt{2}$	0.5	$\sqrt{2}$	0.5	$\sqrt{2}$	0.1	1	0.1
UA-MCTS	2	0.1	2	0.1	$\sqrt{2}$	0.1	1	0.1
MCTS Corrupted Model	$\sqrt{2}$	NA	$\sqrt{2}$	NA	$\sqrt{2}$	NA	$\sqrt{2}$	NA
MCTS True Model	$\sqrt{2}$	NA	$\sqrt{2}$	NA	$\sqrt{2}$	NA	$\sqrt{2}$	NA

Table 3.3: Best exploration constant c and temperature parameter τ in the Freeway environment. The first column presents the best hyperparameters for the offline scenario and the other 3 columns present best hyperparameters for the online scenario.

eter τ are provided in Table 3.4.

MCTS Corrupted Model performance degrades significantly in comparison to MCTS True Model. UA-Expansion and UA-MCTS performed much better than the MCTS Corrupted Model. Moreover, UA-MCTS achieved the performance of MCTS True Model. However, UA-Simulation could not improve the performance of MCTS Corrupted Model.

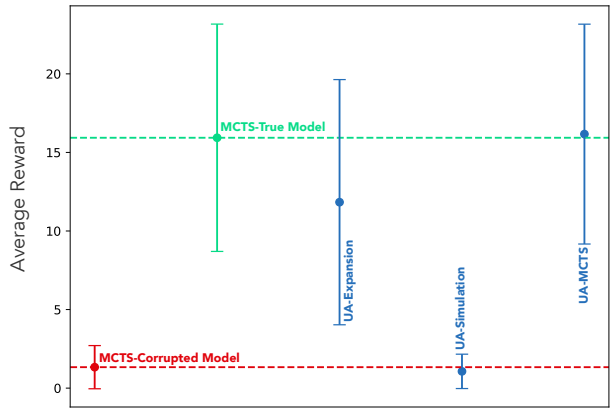


Figure 3.8: Performance of UA-Expansion, UA-Simulation, UA-MCTS, and two MCTS baselines for different βs in the Breakout environment for the online scenario. Each error bar shows $\pm$ standard deviation.

Online Scenario

Figure 3.7 present the performance of our approach for different βs in the Breakout environment for the online scenario. The best exploration constant c and temperature parameter τ are provided in Table 3.4.

UA-Simulation did not improve the performance of MCTS Corrupted Model. UA-Expansion and UA-MCTS performed better than MCTS Corrupted Model, but there is a huge gap between their performance and performance of the MCTS True Model. Since these algorithms performed much better with the perfect uncertainty in the offline scenario, we hypothesise that the gathered buffer does not have sufficient samples. We compared the number of unique samples in the Space Invaders, Freeway, and Breakout environments. The buffer gathered in the Space Invaders and Freeway environment has more than 90% unique samples, whereas it consists of less than 20% unique samples in the Breakout environment. Therefore, the uncertainty learned poorly.

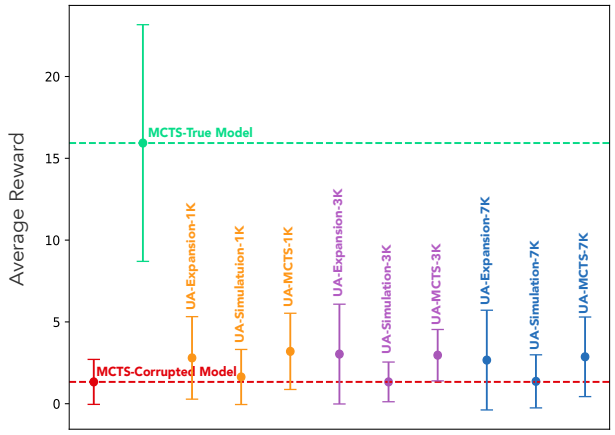


Figure 3.9: Performance of UA-Expansion, UA-Simulation, UA-MCTS, and two MCTS baselines for different βs in the Breakout environment in the online scenario. Each error bar shows $\pm$ standard deviation.

	Offline		β =1K		β =3K		β =7K	
	c	τ	c	τ	c	τ	c	τ
UA-Expansion	$\sqrt{2}$	0.1	1	0.1	0.5	0.1	2	0.5
UA-Simulation	2	0.5	1	0.1	1	0.1	0.5	0.5
UA-MCTS	$\sqrt{2}$	0.1	2	0.9	$\sqrt{2}$	0.1	1	0.1
MCTS Corrupted Model	0.5	NA	0.5	NA	0.5	NA	0.5	NA
MCTS True Model	2	NA	2	NA	2	NA	2	NA

Table 3.4: Best exploration constant c and temperature parameter τ in the Breakout environment. The first column presents the best hyperparameters for the offline scenario and the other 3 columns present best hyperparameters for the online scenario.

3.2.6 Scaling Experiments

In this section, we scale the experiments we described in Sections 3.2.3 - 3.2.5 by scaling N_I.

Figures 3.10-3.12 present performance of MCTS baselines with different N_I in the three environments. We observe that performance MCTS True Model improves with a higher N_I in all the three environments and is always better than MCTS Corrupted Model. Performance of MCTS Corrupted Model does not improve after certain value of N_I which suggests that using a higher N_I in MCTS when the model is imperfect does not improve the performance.

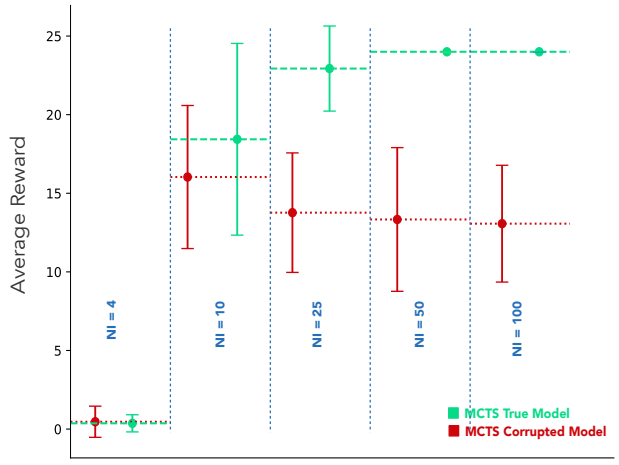


Figure 3.10: Performance of MCTS True Model and MCTS Corrupted Model for different N_Is in the Space Invaders environment. Each error bar shows $\pm$ standard deviation.

Figures 3.13-3.15 present performance of UA-MCTS for the offline and online scenarios with different N_Is in the three environments. We observe that in the offline scenario the performance of UA-MCTS improves with a higher N_I in all the environments. In the online scenario, the performance of UA-MCTS improves with a higher N_I in the Space Invaders and Freeway environments, but not in the Breakout environment. The reason is that the uncertainty learned poorly in the Breakout environment as we explained in Section 3.2.5. Therefore, the performance of UA-MCTS does not improve with a higher N_I in the Breakout environment.

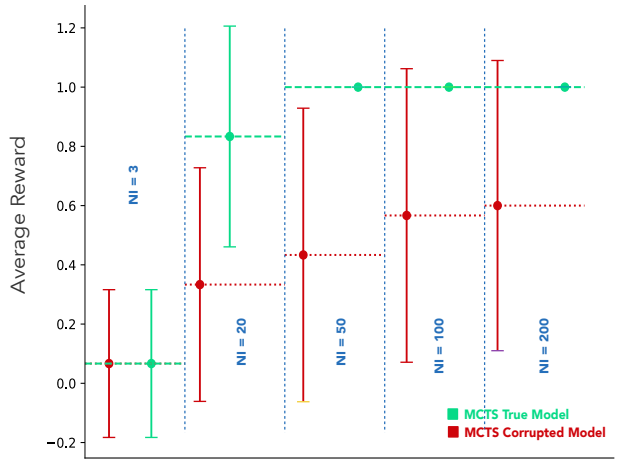


Figure 3.11: Performance of MCTS True Model and MCTS Corrupted Model for different N_Is in the Freeway environment. Each error bar shows ± standard deviation.

3.3 Chapter Summary

Our results show that baseline MCTS suffers from severe performance degradation in the face of model uncertainty. Also, UA-MCTS outperforms MCTS for imperfect models. It can recover from performance degradation, or at least lessen its effects when used in conjunction with a learned uncertainty estimate. The precision of the learned uncertainty model used by UA-MCTS has a very strong effect on agent performance. Moreover, we observed that UA-Expansion can restore MCTS performance, but UA-Simulation failed to do in some of the cases. The reason is that UA-Simulation does not prevent building a tree within the uncertain parts. It adjusts the sampled values. UA-Simulation needs to combine with other components to improve MCTS performance.

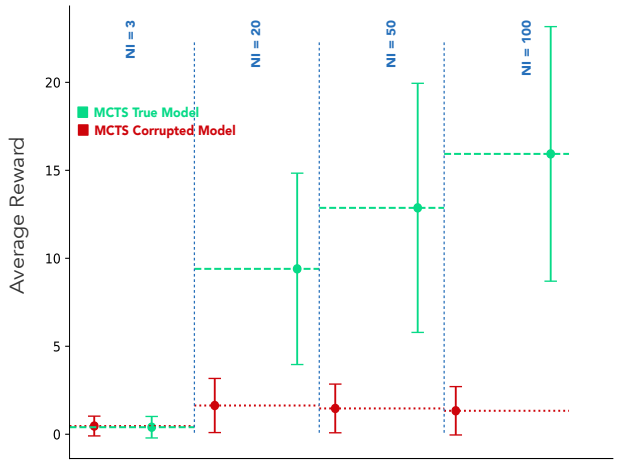


Figure 3.12: Performance of MCTS True Model and MCTS Corrupted Model for different N_Is in the Breakout environment. Each error bar shows ± standard deviation.

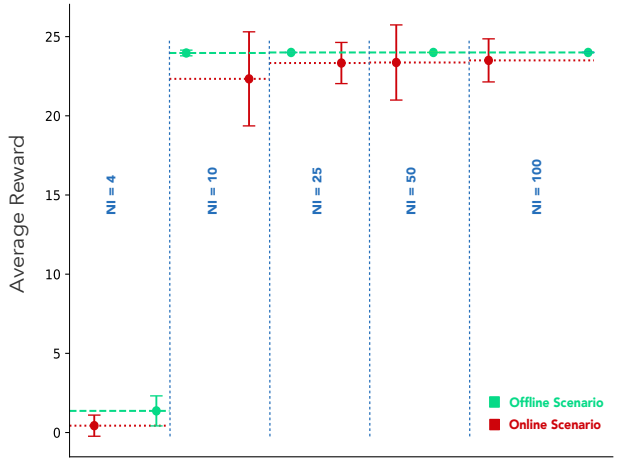


Figure 3.13: Performance of UA-MCTS in the offline and online scenarios for different N_Is in the Space Invaders environment. Each error bar shows ± standard deviation.

Algorithm 10 UA-Backpropagation Algorithm.

Parameter: temperature τ for softmax

function UA-BACKPROPAGATE(v, $value$)

  while v is not NULL **do**

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

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

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

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

    $v \leftarrow Par(v)$

  end while

end function

function UA-SELECT(v)

  while v is expanded **do**

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

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

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

      end for

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

    end for

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

  end while

  return v

end function

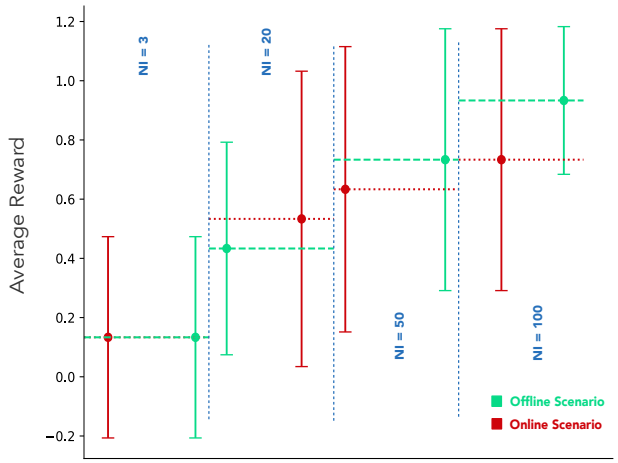


Figure 3.14: Performance of UA-MCTS in the offline and online scenarios for different N_Is in the Freeway environment. Each error bar shows ± standard deviation.

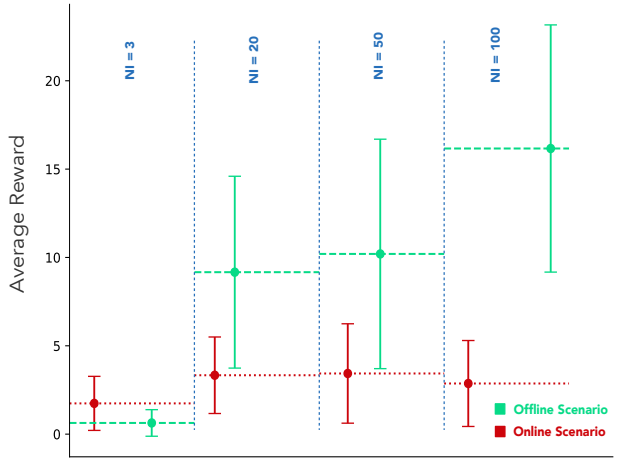


Figure 3.15: Performance of UA-MCTS in the offline and online scenarios for different N_Is in the Breakout environment. Each error bar shows ± standard deviation.

Chapter 4

MCTS and DQN

Heuristics can improve the performance MCTS when the model of the environment is misleading. In domains where RL methods can be applied, the learned values from RL methods can be a good heuristic. When the model of the environment is imperfect, the learned values from model-free RL methods are preferable because the errors in the model do not affect their learning. In this chapter, we study how to improve MCTS performance in the imperfect model domain by using DQN. We call this method DQ-MCTS.

In Section 4.1, we describe our method to use DQN within MCTS. In Section 4.2, we describe our experimental design to investigate the performance of DQ-MCTS, and presents the results of the experiments.

4.1 DQ-MCTS

One strategy to make MCTS work with an imperfect model is to use the learned values from model-free RL methods as a heuristic. We pretrain a DQN as such a heuristic for MCTS. Since DQN learns the value of state-action pairs without using the model of the environment, the errors in the model do not affect the learned values. We present two methods, DQ-Expansion and DQ-Simulation, which use learned DQN values in MCTS in Sections 4.1.1 and 4.1.2. The DQ-MCTS method uses both these techniques.

4.1.1 DQ-Expansion

Usually, MCTS uses the default value of 0 as the initial value for newly added nodes. Using a heuristic to evaluate the newly added nodes can guide the search when the random rollouts can result in erroneous values due to model errors. To do that, we estimate a node's initial value using a DQN. Algorithm 11 provides the pseudo-code of this method. More details are explained in Kiarash [2].

Algorithm 11 DQ-Expansion Algorithm.

function DQ-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
    $N(v_i) \leftarrow 0$
    $Q(v_i) \leftarrow$ DQNVALUEFUNCTION(s_i)
  end for
  return a random child of v
end function

4.1.2 DQ-Simulation

When a rollout of MCTS is performed, inaccuracies in the model compound as the search goes deeper. To prevent this, one idea is to stop a rollout early by choosing a lower maximum depth D_S of each rollout, and use a heuristic to evaluate the endpoint s. This heuristic should give an approximation of the return value from the endpoint s if the rollout was not stopped early. In DQ-Simulation, we use a DQN for such an evaluation. Algorithm 12 provides the pseudo-code of this method.

4.2 Experiments: Design and Results

In this section, we present the details and results of the experiments for testing the performance of DQ-MCTS when the model of the environment is imper-

Algorithm 12 DQ-Simulation Algorithm.

function SIMULATE($s, depth$)
  for $i \leftarrow 1$ to N_S **do**
    $g_i \leftarrow$ ROLLOUT(s)
    $\alpha_i \leftarrow 1/\mathrm{N}_S$
  end for
  return $\sum_{i=1}^{N_S} \alpha_i \cdot g_i$
end function

function ROLLOUT(s)
  $count \leftarrow 0$
  $rewards \leftarrow 0$
  $discount \leftarrow 1$
  while s is not terminal & $count < D_S$ **do**
    choose a random action a from $\mathcal{A}$
    $s, r \leftarrow \hat{M}(s, a)$
    $count \leftarrow count + 1$
    $rewards \leftarrow rewards + discount \cdot r$
    $discount \leftarrow discount \cdot \gamma$
  end while
  if s is not terminal **then**
    $bootstrap \leftarrow$ DQNVALUEFUNCTION(s)
    $rewards \leftarrow rewards + discount \cdot bootstrap$
  end if
  return $rewards$
end function

fect. [1] We also investigate the performance of the DQ-Simulation component independently. The experiments are done on the three modified environment MinAtar environments [26] described in Section 3.2.1. To reduce randomness, we pretrain 30 independent DQN for each of the environments for 30000 episodes. We then choose 3 time points for each environment to sample DQNs' value function. In Section 4.2.1, we describe more details of training DQNs and sampling from them. We want to investigate the effect of D_S on DQ-MCTS. We use different D_S to observe its effect on DQ-MCTS. We hypothesize that a lower D_S causes better results, because it uses the model less and prevents compounding the model's error.

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

We test total number of eleven algorithms.

1-3) "DQN-t" A greedy policy w.r.t. the learned value function by DQN at time step t.

4) "True Model" MCTS is allowed to use M in its search.

5) "Corrupted Model" MCTS uses $\hat{M}$ for all its planning.

6-11) All versions of DQ-MCTS work as follows: they use $\hat{M}$ for all their planning and a sampled DQN value function. The time step that the value function is sampled from is attached to the label of the algorithm. The DQ-MCTS versions labeled "DQ-Simulation" use the DQ modifications only for this one component of MCTS, and use unmodified MCTS for the other three parts. The "DQ-MCTS" version uses both DQ-Expansion and DQ-Simulation enhancements.

For each combination of the environment, algorithms 4-11, and D_S, we perform a parameter sweep over the exploration constant c from the set $c \in \{0.5, 1, \sqrt{2}, 2\}$ respectively for 5 runs for algorithms 6-11 and 30 runs for algorithms 4-5. Since we have 30 independent DQN runs, the total number of runs for algorithms 6-11 is be 150. The mean and standard deviation of the best-performing configuration for each experiment is reported as the final result.

Table 4.1 presents the hyperparameters we used in each environment. In Sections 4.2.2 - 4.2.4, we show the experiment setup and results for each environment.

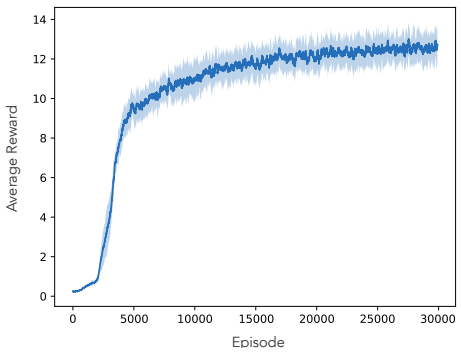
	Space Invaders	Freeway	Breakout
N_I	10	100	100
D_S	$[0, 5, 10, 20]$	$[0, 5, 10, 25, 50]$	$[0, 5, 10, 25, 50]$
N_S	10	10	10

Table 4.1: Search hyperparameters for the modified MinAtar environments.

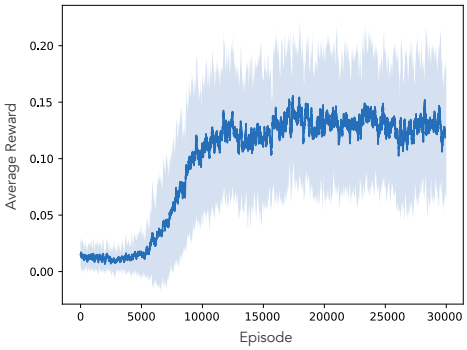
4.2.1 Pretraining DQN

For each environment, we pretrain 30 DQN for 30000 episodes independently. Each neural network representing the state-action value function in DQN is a

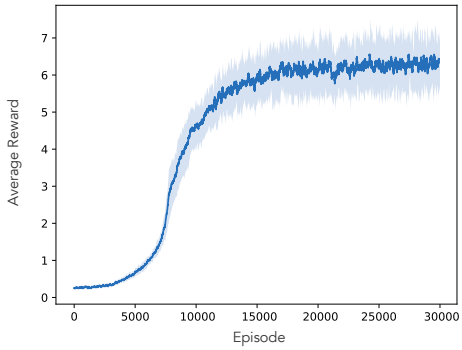
fully connected neural network with two hidden layers with 64 units in each. The target network update frequency is 1000, and the replay buffer size is 10000. The ϵ starts with 1.0 and decays with time linearly. We use a batch size of 32 and the RMSProp optimizer with a step size of 0.00025. Figure 4.1 presents the learning curve of the DQNs for each of the three environments.



(a) Space Invaders



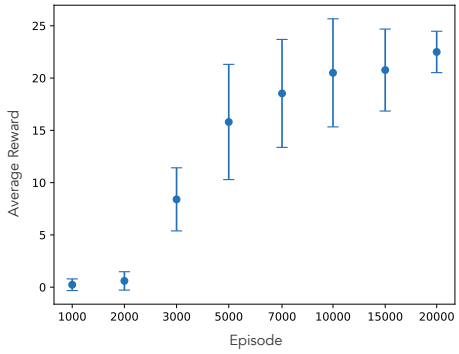
(b) Freeway



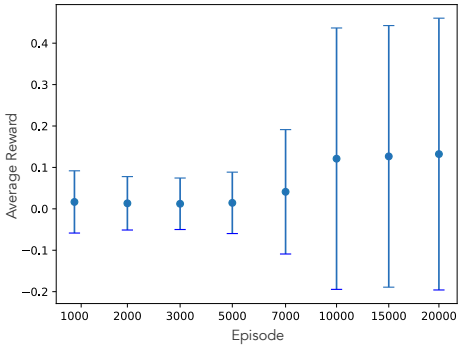
(c) Breakout

Figure 4.1: Learning curve of DNQ for the Space Inavder, Freeway, and Breakout environments. The shaded region shows ± standard deviation.

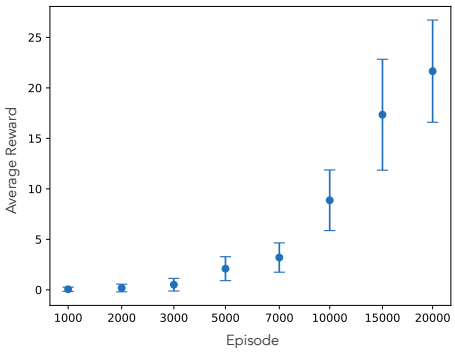
For each of the environments and each of the 30 runs, we sampled the learned state-action value function for each DQN in episodes 1000, 2000, 3000, 5000, 7000, 10000, 15000, and 20000. Then we measured the performance of each learned value function by running experiments using the greedy policy with respect to that learned value function for 30 episodes. Figure 4.2 presents the performance of the sampled value functions for each of the environments.



(a) Space Invaders



(b) Freeway



(c) Breakout

Figure 4.2: The average performance of sampled DQN value function at episodes 1000, 2000, 3000, 5000, 7000, 10000, 15000, and 20000 over 30 episodes for (a) Space Invaders, (b) Freeway, and (c) Breakout environment. Each error bar shows ± standard deviation.

4.2.2 Space Invaders

For the Space Invaders environment, we sampled the learned value function by each 30 DQNs at episodes 3000, 7000, and 20000. For each of these samples and each $D_S \in [0, 5, 10, 20]$, we evaluate the performance of DQ-Simulation and DQ-MCTS.

Figure 4.3 presents the performance of the greedy policy with respect to the sampled value functions (DQN-3000, DQN-7000, DQN-20000), MCTS Corrupted Model, MCTS True Model, DQ-Simulation, and DQ-MCTS. We observe that with a lower D_S, DQ-Simulation and DQ-MCTS perform better. When D_S gets higher, there is a higher probability that the rollouts reach

incorrect states. By evaluating that incorrect state using the learned value function, even when the value for the incorrect states has no error, a wrong value will be calculated for the rollout. Therefore, using a lower depth is preferred when we have a heuristic to estimate the value of the states, because it prevents accumulating errors in the model. Also, a better learned value function (from DQN at episode 20000), resulted in better performance of DQ-Simulation and DQ-MCTS when D_S is 0. But when D_S is higher, the errors in the model accumulate and the performance of DQ-Simulation and DQ-MCTS is not good at all. In general, DQ-MCTS and DQ-Simulation with $D_S = 0$ performed better than all the MCTS with a corrupted model. MCTS with the corrupted model performed better with a higher D_S. Although there are errors in the model, MCTS could benefit by performing deeper rollouts.

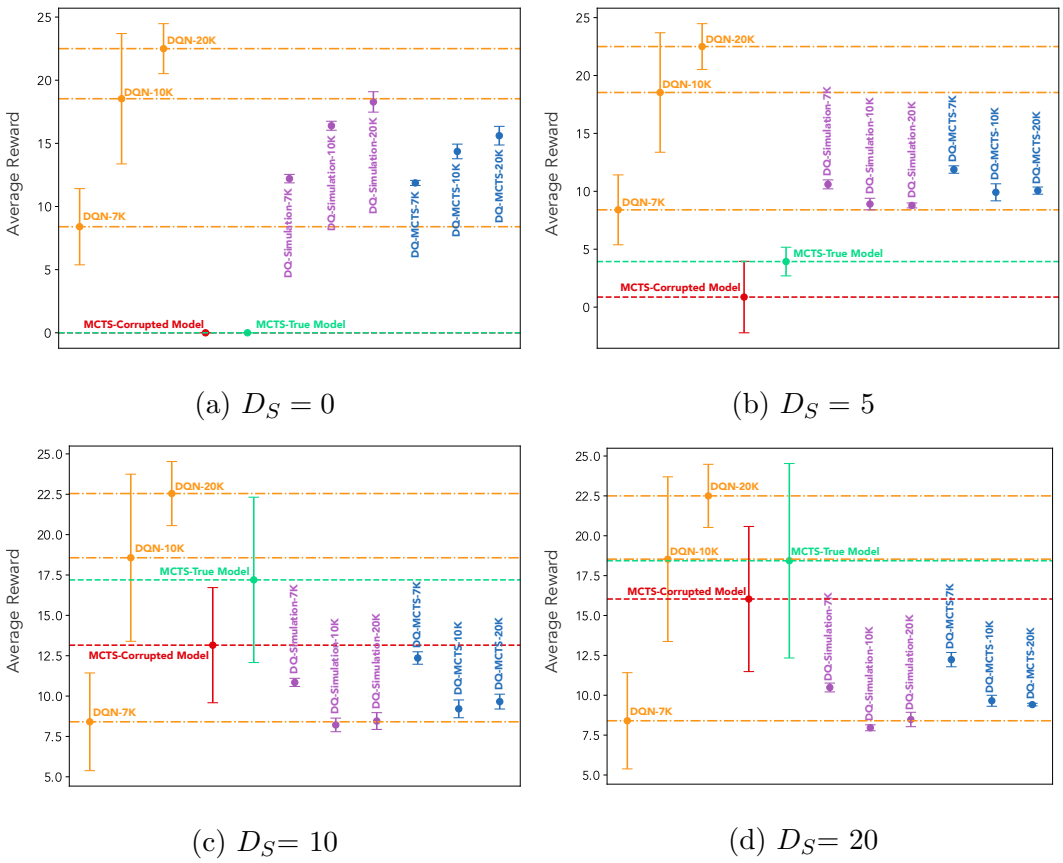
The best exploration constant c is provided in Table 4.2.

	$D_S = 0$	$D_S = 5$	$D_S = 10$	$D_S = 20$
MCTS Corrupted Model	2	1	0.5	0.5
MCTS True Model	2	1	2	2
DQ-Simulation 3K	$\sqrt{2}$	1	$\sqrt{2}$	2
DQ-Simulation 7K	0.5	2	2	0.5
DQ-Simulation 20K	1	2	1	1
DQ-MCTS 3K	$\sqrt{2}$	2	2	$\sqrt{2}$
DQ-MCTS 7K	0.5	2	0.5	2
DQ-MCTS 20K	$\sqrt{2}$	$\sqrt{2}$	2	1

Table 4.2: Best exploration constant c for each method for the Space Invaders environment

4.2.3 Freeway

For the Freeway environment, we sample the value function learned by 30 DQNs at episodes 7000, 10000, and 20000. We evaluate the performance of DQ-Simulation and DQ-MCTS using the sampled value functions for 5 runs for each $D_S \in [0, 5, 10, 25, 50]$. Figure 4.4 presents the performance of the greedy policy with respect to the sampled value functions, MCTS with corrupted and true model, DQ-Simulation, and DQ-MCTS. We can observe that Freeway is a hard task for DQN, but even with a poorly learned value



(a) $D_S = 0$ (b) $D_S = 5$

(c) $D_S = 10$ (d) $D_S = 20$

Figure 4.3: The performance of DQN-3000, DQN-7000, DQN-20000, MCTS-Corrupted Model, MCTS-True Model, DQ-Simulation and DQ-MCTS in the Space Invaders environment for (a) $D_S = 0$, (b) $D_S = 5$, (c) $D_S = 10$, and (d) $D_S = 20$. Each error bar shows ± standard deviation.

function, DQ-MCTS and DQ-Simulation with $D_S = 0$ performed better than all the MCTS with a corrupted model. Also, we can observe that using a higher D_S did not improve the performance of MCTS with a corrupted model, DQ-Simulation, and DQ-MCTS due to the compounded error from the model. Moreover, we can observe that a better learned value function results in higher performance in DQ-Simulation and DQ-MCTS when D_S is low (0 and 5). The best exploration constant c is provided in Table 4.3.

4.2.4 Breakout

For Space Invaders, we sample the value function learned by 30 DQNs at episodes 7000, 10000, and 20000. We evaluate the performance of DQ-Simulation

	$D_S = 0$	$D_S = 5$	$D_S = 10$	$D_S = 25$	$D_S = 50$
MCTS True Model	2	2	2	2	$\sqrt{2}$
MCTS Corrupted Model	2	2	$\sqrt{2}$	1	$\sqrt{2}$
DQ-Simulation 7K	2	2	2	2	1
DQ-Simulation 10K	1	$\sqrt{2}$	$\sqrt{2}$	2	1
DQ-Simulation 20K	1	2	$\sqrt{2}$	0.5	1
DQ-MCTS 7K	2	$\sqrt{2}$	1	2	0.5
DQ-MCTS 10K	2	$\sqrt{2}$	0.5	$\sqrt{2}$	2
DQ-MCTS 20K	$\sqrt{2}$	1	2	2	2

Table 4.3: Best exploration constant c for each method for the Freeway environment

and DQ-MCTS using the sampled value functions for 5 runs for each $D_S \in [0, 5, 10, 25, 50]$. Figure 4.5 presents the performance of the greedy policy with respect to the sampled value functions, MCTS with corrupted and true model, DQ-Simulation, and DQ-MCTS. We can observe the same results as in the Freeway environment. We can observe that DQ-MCTS and DQ-Simulation with $D_S = 0$ performed better than MCTS with a corrupted model. Again, with $D_S = 0$ better learned value function (the learned value function at episode 20000) resulted in higher performance in DQ-Simulation and DQ-MCTS. Also, we can observe that using a higher D_S did not improve the performance of MCTS with corrupted model significantly, and decreased the performance of DQ-Simulation, and DQ-MCTS due to the compounded error from the model.

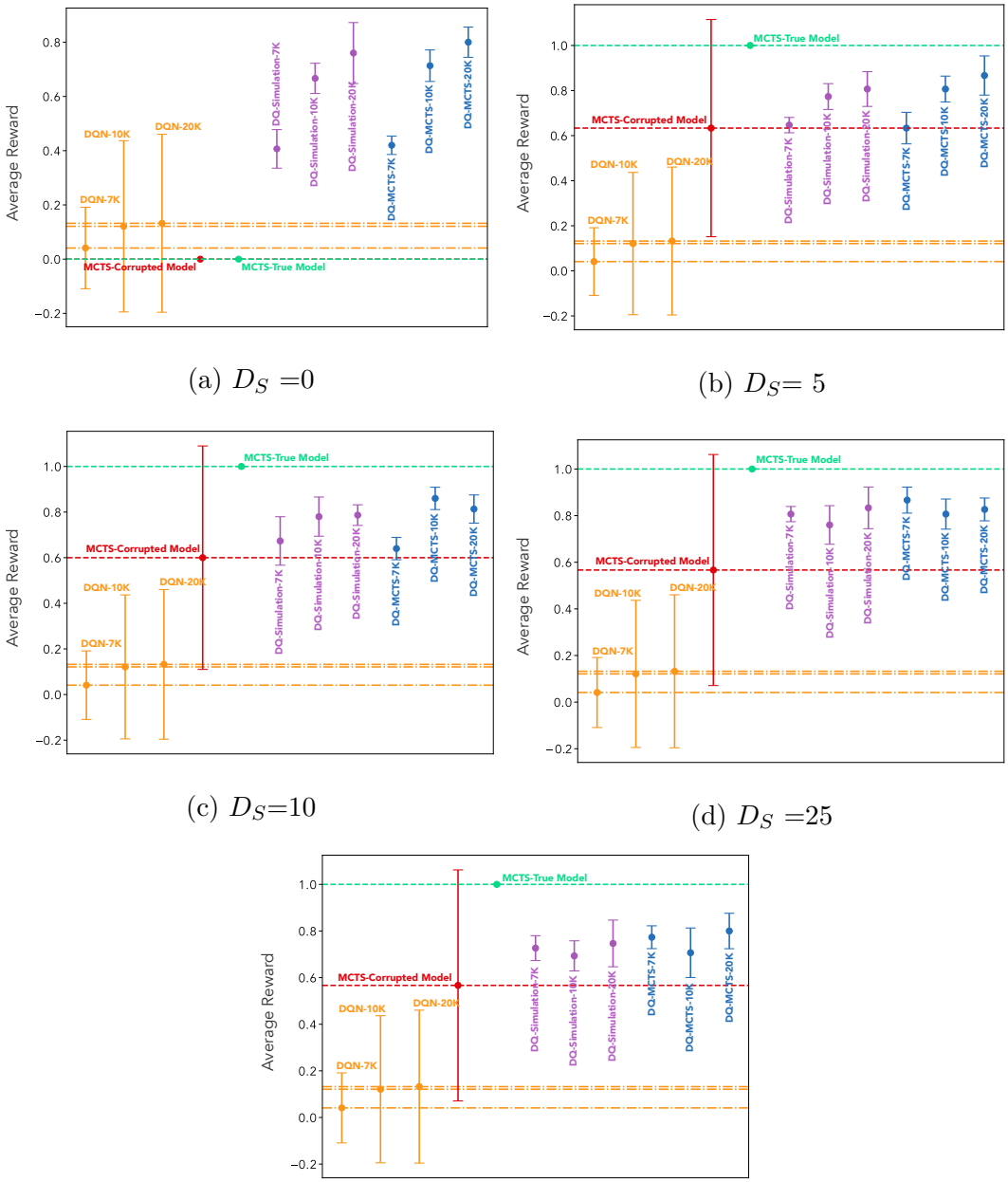
The best exploration constant c is provided in Table 4.4.

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

Table 4.4: Best exploration constant c for each method for the Breakout environment

4.3 Chapter Summary

We observed that DQ-Simulation and DQ-MCTS with a low D_S perform better than MCTS with an imperfect model. When D_S is low, a better DQN results in higher performance in DQ-Simulation and DQ-MCTS. Also, we observed that when D_S is high, due to the compounded error from the model, more unrealistic states will be added to the tree. Evaluating these unrealistic states does not help the performance of DQ-Simulation and DQ-MCTS. One major drawback of the methods presented in this chapter is that none of them achieved the performance of the best DQN.



(a) D_S =0

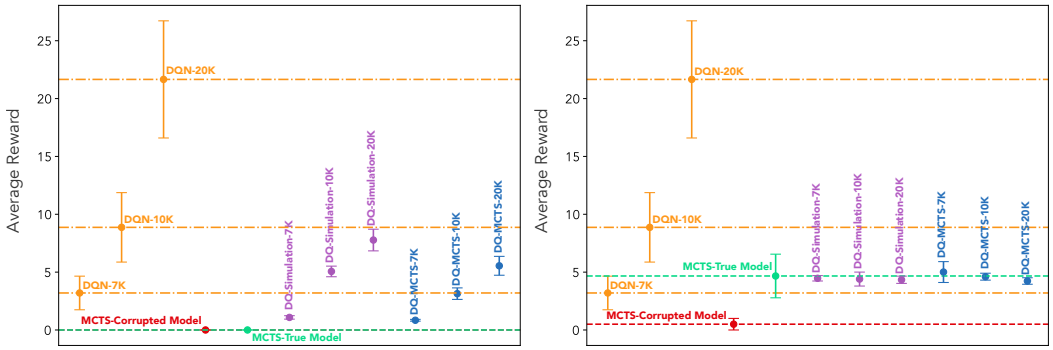
(b) D_S= 5

(c) D_S=10

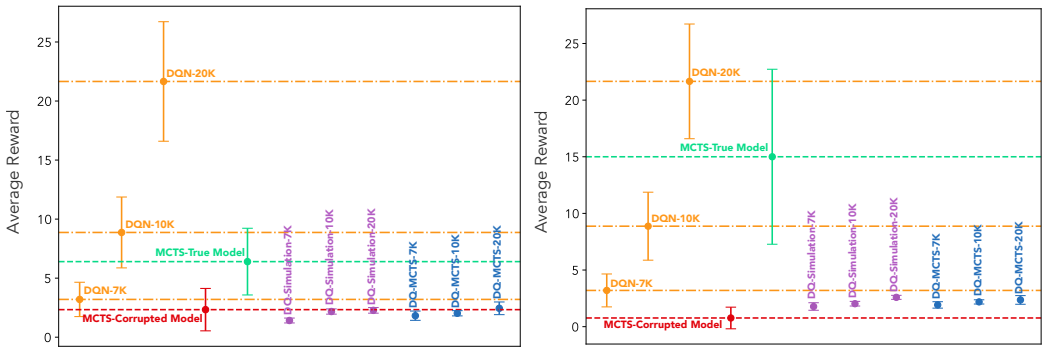
(d) D_S =25

(e) D_S= 50

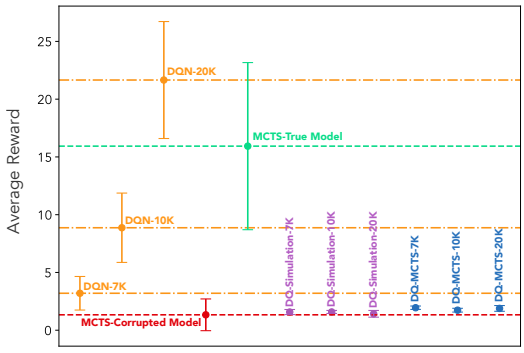
Figure 4.4: The performance of DQN-3000, DQN-7000, DQN-20000, MCTS-Corrupted Model, MCTS-True Model, DQ-Simulation and DQ-MCTS in the Freeway environment for (a) D_S= 0, (b) D_S= 5, (c) D_S= 10, (d) D_S= 25, and (e) D_S = 50. Each error bar shows ± standard deviation.



(a) $D_S = 0$ (b) $D_S = 5$



(c) $D_S = 10$ (d) $D_S = 25$



(e) $D_S = 50$

Figure 4.5: The performance of DQN-3000, DQN-7000, DQN-20000, MCTS-Corrupted Model, MCTS-True Model, DQ-Simulation and DQ-MCTS in the Breakout environment for (a) D_S= 0, (b) D_S= 5, (c) D_S= 10, (d) D_S= 25, and (e) $D_S = 50$. Each error bar shows ± standard deviation.

Chapter 5

Conclusion and Future Work

In this thesis, we proposed UA-MCTS, an adaptation of the MCTS framework to model uncertainty. In UA-MCTS, the selection, expansion, simulation, and backpropagation steps of MCTS are modified so that they discourage searching through parts of the model with high uncertainty. Moreover, we proposed a method to learn the uncertainty of the model. We tested the performance of the UA-MCTS framework on three modified MinAtar environments. The results show that UA-MCTS can outperform MCTS, or at least lessen the negative effects of model errors when used in conjunction with a learned uncertainty estimate. The precision of the learned uncertainty model used by UA-MCTS has a very strong effect on agent performance.

The UA-MCTS framework is not restricted to our method of learning the model uncertainty. Learning a more accurate model is one direction for future work. Moreover, an uncertainty model can be continually learned in the real environment and used in UA-MCTS. We also leave such online learning as future work.

We proposed the DQ-MCTS framework, a combination of MCTS framework with DQN, a state of art model-free RL method. Since model-free RL methods do not use the model of the environment, the errors in the model do not affect their learned values. In DQ-MCTS, we used the learned value function of a pretrained DQN as a heuristic in the MCTS framework. The value function learned by the DQN is used to initialize the value of the newly added nodes in the tree and also bootstrap from the last state in the rollouts.

The results showed that DQ-MCTS can perform better than MCTS when the model is imperfect, but it could not achieve the performance of the pretrained DQN.

DQ-MCTS can use other value functions instead of the DQN learned value function. Comparing different model-free RL methods for pretraining a value function is another interesting future work. These value functions do not need to be fixed and they can change while the agent is interacting with the environment. Learning such an online value function also remains as future work.

References

[1] Zaheer Abbas et al. "Selective dyna-style planning under limited model capacity." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 1–10.

[2] K. Aghakasiri. "Monte Carlo Tree Search in the Presence of Model Uncertainty." MA thesis. University of Alberta, 2022. URL: `in%20preparation`.

[3] Mohamed R Amer et al. "Monte Carlo Tree Search for scheduling activity recognition." In: *Proceedings of the IEEE international conference on computer vision*. 2013, pp. 1353–1360.

[4] Debangshu Banerjee. "Hex and Neurodynamic Programming." In: *arXiv preprint arXiv:2008.06359* (2020).

[5] Zahy Bnaya et al. "Repeated-task Canadian traveler problem." In: *AI Communications* 28.3 (2015), pp. 453–477.

[6] Cameron B Browne et al. "A survey of Monte Carlo Tree Search methods." In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[7] Patrick Clary et al. "Monte Carlo planning for agile legged locomotion." In: *Twenty-Eighth International Conference on Automated Planning and Scheduling*. 2018, pp. 446–450.

[8] Jie Jia, Jian Chen, and Xingwei Wang. "Ultra-high reliable optimization based on Monte Carlo Tree Search over Nakagami-m Fading." In: *Applied Soft Computing* 91 (2020), p. 106244.

[9] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *3rd International Conference on Learning Representations, ICLR*. 2015.

[10] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. "Improved Monte Carlo Search." In: *Univ. Tartu, Estonia, Tech. Rep* 1 (2006).

[11] Jan Kuipers et al. "Improving multivariate Horner schemes with Monte Carlo Tree Search." In: *Computer Physics Communications* 184.11 (2013), pp. 2391–2395.

[12] Chris Mansley, Ari Weinstein, and Michael Littman. "Sample-based planning for continuous action Markov Decision Processes." In: *Twenty-First International Conference on Automated Planning and Scheduling*. 2011, pp. 335–338.

[13] Volodymyr Mnih et al. "Playing Atari With Deep Reinforcement Learning." In: *NIPS Deep Learning Workshop*. 2013.

[14] Teresa Neto et al. "A multi-objective Monte Carlo Tree Search for forest harvest scheduling." In: *European Journal of Operational Research* 282.3 (2020), pp. 1115–1126.

[15] Marc Ponsen, Geert Gerritsen, and Guillaume Chaslot. "Integrating opponent models with Monte Carlo Tree Search in poker." In: *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010.

[16] Arpad Rimmel. "Improvements and Evaluation of the Monte Carlo Tree Search Algorithm." PhD thesis. Paris 11, 2009.

[17] Abdallah Saffidine. "Utilisation dUCT au Hex." In: *Ecole Normale Super. Lyon, France, Tech. Rep* (2008).

[18] Yoshikuni Sato, Daisuke Takahashi, and Reijer Grimbergen. "A shogi program based on Monte Carlo tree search." In: *Icga Journal* 33.2 (2010), pp. 80–92.

[19] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *nature* 529.7587 (2016), pp. 484–489.

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

[21] Anirudh Vemula, J Andrew Bagnell, and Maxim Likhachev. "CMAX++: Leveraging Experience in Planning and Execution using Inaccurate Models." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 7. 2021, pp. 6147–6155.

[22] Anirudh Vemula et al. "Planning and Execution using Inaccurate Models with Provable Guarantees." In: *CoRR* abs/2003.04394 (2020). arXiv: 2003.04394. URL: https://arxiv.org/abs/2003.04394.

[23] Yi Wan et al. "Planning with expectation models." In: *arXiv preprint arXiv:1904.01191* (2019).

[24] Christopher JCH Watkins and Peter Dayan. "Q-learning." In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[25] Chenjun Xiao et al. "Learning to combat compounding-error in model-based reinforcement learning." In: *arXiv preprint arXiv:1912.11206* (2019).

[26] Kenny Young and Tian Tian. "Minatar: An Atari-inspired testbed for thorough and reproducible Reinforcement Learning experiments." In: *arXiv preprint arXiv:1903.03176* (2019).