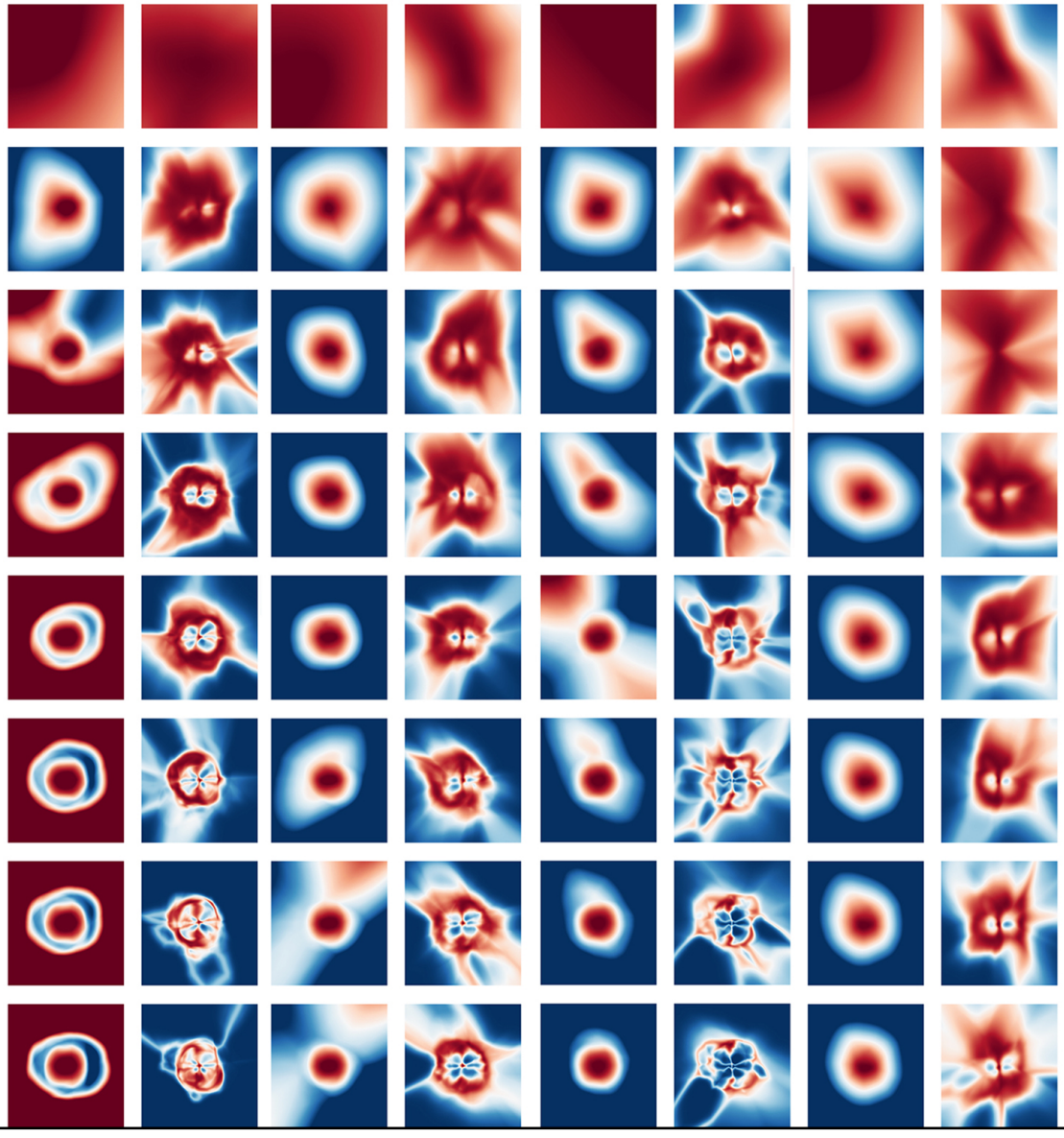
Generalization and locality in the AlphaZero algorithm

A study in single-player, fully observable, deterministic environments

A. Deichler

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Generalization and locality in the AlphaZero algorithm

A study in single-player, fully observable, deterministic environments

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

A. Deichler

March 7, 2019

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

Recently, the AlphaGo algorithm has managed to defeat the top level human player in the game of Go. Achieving professional level performance in the game of Go has long been considered as an AI milestone. The challenging properties of high state-space complexity, long reward horizon and high action branching factor in the game of Go are also shared by many other complex planning problems, such as robotics applications. This makes the algorithmic solutions of AlphaGo particularly interesting for further research. One of the key innovations in the algorithm is the combination of Monte Carlo tree search (MCTS) with deep learning. The main hypothesis of the thesis is that the success of the algorithm can be attributed to the combination of the generalization capacity of deep neural networks and the local information of tree search. This hypothesis is evaluated through the application of the AlphaZero algorithm (extension of Alphago) in single-player, deterministic and fully-observable reinforcement learning environments. The thesis presents answers to two research questions. First, what changes need to be made to transfer the AlphaZero algorithm to these environments. The changes in the reward support in these new environments can cause failure of learning, since assumption in the MCTS algorithm are violated. The thesis offers solutions that deal with this problem, including adaptive return normalization. The second research question examines what is the relative importance of the locality and generalization in the performance of the AlphaZero algorithm. This research question is answered by comparing the performance of search trees of varying sizes in several RL environments under fixed time budgets. While building small trees support generalization, through allowing more frequent training of the neural network, building larger trees provide more accurate estimates. This creates a trade-off between improved generalization capacity and more accurate local information under a fixed time budget. The experiment results show that mid-size trees achieve the best performance, which suggests that balancing local information and generalization is key to the success of the algorithm. Based on this results, possible extensions to the algorithm are proposed. At last, the thesis also highlights the relevance of the two-component system from a broader perspective and discusses the possible future impact of the algorithm.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Thomas Moerland for being able to work on this project and for all his support throughout the thesis. His extensive knowledge about reinforcement learning was very inspiring to me and I found the thesis dicussions very valuable.

I am also thankful to my DCSC supervisor, dr. Simone Baldi for his attention and useful comments about the thesis.

I would also like to thank my mother, who has always encouraged me to pursue my interest.

Chapter 1

Introduction

Achieving top human level performance in the game of Go has been considered milestone in artificial intelligence (AI) research for a long time. In the game of Go, two players place stones on a 19×19 board in turn and the goal is to control more territory than the enemy (figure 1-1). What makes the game particularly challenging for computer programs to solve is its state-space complexity, large action branching factor and long reward horizon. The game has more than 10^{170} states, 250 possible moves on average and feedback for a move might be received after 50-100 steps [1]. These properties make traditional search methods ineffective for Go and necessitate a more focused search algorithm. It has been suggested that in order to be able achieve good performance in the game of Go, a computer program would have to possess a pattern knowledge similar to human experts or compensate for the lack of such knowledge through search [1]. Monte Carlo tree search (MCTS) [3] was the first algorithm to achieve

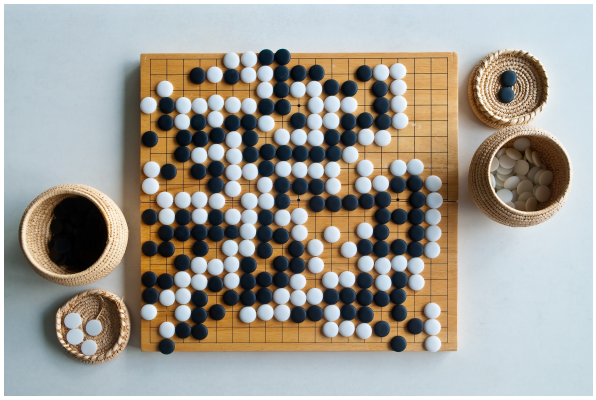


Figure 1-1: View of board in game of Go [2]

significant performance improvements in Go. In contrast to traditional search methods, where static evaluation functions store knowledge about all positions, MCTS uses local and dynamic position evaluation [4]. It has been suggested that the locality of information is the main strength of the MCTS algorithm. As each edge stores its own statistics, it is easier to locally

separate the effect of actions [5]. MCTS brought significant performance improvements, but did not play at the level of human professionals [4]. Human expert players rely on their experience to rule out the majority of legal moves and only consider a small number of them in selecting the next move. The lack of intuition have been considered a major hurdle in the advancement of computer programs to compete at human level [1]. Intuition in case of the game of Go can be understood as recognizing patterns of game positions from previous experience and using them to facilitate decision-making in the current position. In recent years, deep neural networks have achieved significant performance improvements in pattern recognition. The main strength of deep neural network is their generalization capacity, which allows them to transfer information from previous experience to new situations.

The AlphaGo [6] algorithm has professional human performance on the game of Go by relying on both pattern knowledge through the use of a deep neural network and search using Monte Carlo tree search. Defeating a professional Go player was a significant achievement in AI research, since the challenging properties of large state and action spaces and long reward horizons in the game of Go are also shared by other complex planning and decision making problems (e.g robotics) [4]. For these reasons, it is worth to examine the structure of the algorithm more in depth.

The AlphaGo algorithm was later generalized to chess and shogi in the AlphaZero algorithm [7]. This thesis will look at the question of transferring the AlphaZero algorithm to single-player, deterministic and perfect information RL environments. Secondly, the thesis also examines what is the relative of importance of local search and generalization from deep learning in the algorithm. The next section formulates these research questions.

1-1 Research questions

The main aim of investigations in this thesis can be summarized in two research questions:

(R1) ***The applicability of the AlphaZero algorithm to single-player, deterministic environments*** The original AlphaZero algorithm was created for games [7]. This thesis looks at how the algorithm performance changes for problems of different scales (action and state spaces) and reward functions and what modifications can be made on the original algorithm for improved performance in the different environment settings.

(R2) ***The investigation of the relative importance of locality and generalization in the performance of the AlphaZero agent*** The algorithm requires the number of node expansions in the MCTS search tree to be set. This hyperparameter controls how much effort is given for selecting each action. Assuming a fixed computational budget, this parameter also balances locality and generalization. With a higher number of expansions, more time is spent on selecting actions at each time step (locality) and less budget remains for training the neural network on real environment experience (generalization). Performing MCTS search against the ground-truth model of the environment can have the benefit of a more stable learning process. However, with higher number of node expansions in the search tree, less time remains for training the neural network and therefore the generalization capability of the neural network will be worse, which affects the learning performance.

1-2 Thesis outline

The thesis is organized in six chapters. This first chapter introduced the research questions of the thesis. The second chapter provides background information, which is necessary for examining the research questions. It introduces the reinforcement learning framework, in which the sequential decision making problems can be studied and the components of the AlphaZero algorithm. The third chapter examines the first research question (R1), the transferability of the AlphaZero algorithm. The fourth chapter examines the second research question (R2), the trade-off between generalization and locality in the AlphaZero algorithm. The fifth chapter is the discussion, which focuses on results from the second research question, puts the algorithm structure in a broader perspective and looks at related and possible future research directions. Finally, the sixth chapter concludes the thesis.

Chapter 2

Background

2-1 The reinforcement learning framework

Reinforcement learnig (RL) is a machine learning approach to sequential decision making problems. Machine learning is a data-driven approach to AI, where the rules for decision making tasks are derived from data. Based on the provided data, ML approaches can be further classified to supervised and unsupervised learning. In supervised learning the task is to learn a predictor based on examples of input-output data ("good behaviour"), that can generalize and make predictions for unseen input data. In unsupervised learning no labels are provided, the task is to discover regularities in the input data. In reinforcement learning, the learner is responsible for data collection and the task is to predict a sequence of actions. The learning is an interactive process between the two main component of the RL framework, the learning agent and the environment an it is guided by a reward signal (reinforcement), that the agent receives for each of its predicted actions from the environment. The agent takes actions to maximize the expected cumulative reward from the environment.

2-1-1 Markov Decision Process

The reinforcement learning problem can be modeled as a Markov decision process (MDP) if the environment configuration is fully known to the agent (fully observable). An MDP is a mathematical framework for decision making problems and can be described by a tuple of five elements: $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot,\cdot), \rho(\cdot,\cdot), \gamma \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}$ is the state transition probability function, which maps each state-action pair to a distribution over successor states, usually : $P : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \rightarrow [0, \infty)$, ρ is the reward function, usually $\rho : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, representing rewards given for an action in a given state, $\gamma \in [0, 1]$ is the discount factor for future rewards. State transitions of an MDP have to satisfy the Markov property, which states that the next state only depends on the current state and action. The environment is said to be deterministic, if the state transition function and the reward function are deterministic. An environment is fully-observable (perfect-information) if the agent can directly observe the underlying state, otherwise it is partially observable [8].

2-1-2 Value function and policy

In the RL system the agent follows a reward-seeking behaviour, which is defined through the policy and the value functions. The expected future rewards depend on the actions taken by the agent, which is defined by the policy function. The policy function maps states to probabilities of selecting each possible action, defining the agent's behaviour $\pi : S \to A$.
For a given policy the return of a state-action (s, a) pair can be defined as the (discounted) sum of rewards, starting in (s, a), following π

$$R^{\pi}(s,a) = \sum_{t=0}^{\infty} \gamma^t r_t | (s, \pi(\cdot|s)). \tag{2-1}$$

The expected value of $R(s, a)$ is called the value function of policy π

$$V^{\pi}(s) = \mathbb{E}[R^{\pi}(s,a)] = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t | (s, \pi(\cdot|s))\Big]. \tag{2-2}$$

The action-value function defined the expected rewards, when the agent starts in state s, takes action a and then follows π

$$Q^{\pi}(s,a) = \mathbb{E}[R^{\pi}(s,a)] = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t | (s, a)\Big]. \tag{2-3}$$

The value function can be written in terms of the immediate reward and the values of the successors states, following π as a recursive equation for Q^{π} (Bellman equation).

$$Q^{\pi}(s,a) = r(s,a) + \gamma \mathbb{E}_{a' \ \pi(\cdot|s')}\big[Q^{\pi}(s', a')\big]. \tag{2-4}$$

The value function maps states to a scalar value. It provides a long-term view for the agent in choosing actions by quantifying how much reward the agent is expected to accumulate in the future starting from the given state. The reward-seeking behaviour now can be formulated in terms of the value function. The agent's goal is to find the optimal policy π^*, which maximizes the expected accumulated reward for every initial state $s \in S$. $Q^{\pi^*}(s,a) = Q^*(s,a)$, where $Q^*(s,a) = \sup_{\pi} Q^{\pi}(s,a)$. The policy is said to be greedy with respect to the value function, if for all $s \in S, a \in A$

$$\pi(s) = \underset{a \in A}{\operatorname{argmax}} Q(s,a).$$

Most RL algorithms can be described as an interaction between two processes: estimating Q^{π} (policy evaluation) and following the policy (policy improvement). This is expressed by the generalized policy iteration schema (GPI).

2-1-3 Reinforcement learning algorithms

Value function estimation Distinction between algorithms can be made based on how the value function is estimated in the policy evaluation step of the algorithm. The first distinction can be made whether an environment model is used in the value updates. Apart from updating a policy and value function, the agent can also maintain an environment model. These

algorithms are called model-based. Dynamic programming (DP) uses one step expected updates and requires knowledge of the exact environment dynamics P. There are also model-based RL algorithms, where an approximate environment model is learned from experience. If the agent has no knowledge about the environment model (model-free), the value function has to be estimated from experience that the agent collects through interacting with the environment following a fixed policy (sample-based).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[\zeta - Q(s_t, a_t)\Big] \tag{2-5}$$

Based on how the new value target ζ is set in the estimation, Monte-Carlo and Temporal Difference methods can be distinguished. In Monte-Carlo (MC) algorithms complete roll-outs are performed from the starting state until a terminal state is reached and the returns of these roll-outs are averaged to estimate the value function for the given state.

$$\zeta = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} .. + \gamma^{T-t-1} r_T \tag{2-6}$$

In contrast, one-step temporal difference (TD) algorithms, use incremental updates and bootstrap. Bootstrapping means the reuse of previous value estimates in the new value estimate.

$$\zeta = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \tag{2-7}$$

While the TD (2-7) update suffers from high bias, due to initialization of the $Q(s_{t+1}, a_{t+1})$ bootstrap value, the Monte Carlo (MC) (2-6) update has low bias, but high variance due to the target covering multiple time steps. N-step algorithms and eligibility traces form a bridge between the one step TD and the MC algorithms. Eligibility traces turn the n-step return to a compound return (λ-return), by averaging n-step returns ($n = 1, ..T$). Intermediate bootstrapping leads to better performance, in contrast to one step updates [8].

Data structures maintained Model-free algorithm can be further grouped based on the data structure they maintain and update: value-function based algorithms, policy-based algorithms and actor-critic algorithms **Value-based algorithms** are also called critic-only algorithms. These algorithms first estimate the value function, then derive the policy based on the estimated value function. **Policy-based algorithms** maintain a parametrized policy, the actor $\pi_{\boldsymbol{\theta}}$ and search for the optimal policy in the neighbourhood of the current policy. The mapping from observations to actions is learned directly, without estimating a value function. **Actor-critic algorithms** incorporate the advantages of both value function-based and policy-based methods by maintaining and updating both the value function and the policy. The policy is explicitly maintained in addition to the value function for the current policy. The critic estimates the action-value function using a policy evaluation algorithm (e.g. temporal-difference learning). The actor adjusts the policy parameters by gradient ascent [9].

2-2 Challenges in reinforcement learning

There have been some major advances in recent years in RL, but there still remains some open questions. These challenges are connected to the RL framework, and shared by all the

RL algorithms distinguished above. This section discusses three general RL challenges and recent advances on these: the high-dimensional perception, long-term credit assignment and dealing with sparse reward through balancing exploitation and exploration.

2-2-1 Exploration in reinforcement learning

A key-problem in sequential decision making problems is the exploration-exploitation trade-off. In RL, the goal of the agent is to maximize its cumulative reward. This can be achieved by using a greedy policy, which always chooses the action which maximizes the learned value-function estimate (exploitation). In most cases this will lead to a suboptimal solution, since it ignores the uncertainties present in the RL system. Uncertainties can arise from limited amount of experience and environment stochasticity. A greedy algorithm can miss the discovery of higher reward spaces and can end-up in local minima. In order to prevent this, exploration should be introduced in the RL algorithm. The two main approaches to exploration are directed and undirected exploration [10].

2-2-1-1 Undirected exploration approaches

Undirected exploration relies only on randomness. The two main undirected algorithms are ϵ-greedy and the softmax algorithms.

ϵ-greedy exploration is used frequently (e.g. DQN [11]) to ensure exploration. The ϵ-greedy approach takes random actions with a pre-defined probability ϵ, while behaving according to the greedy policy with probability $(1-\epsilon)$. One disadvantage of this approach is that it samples actions with a uniform distribution and wastes resources on exploring low values.

Softmax action selection Softmax exploration strategies drive exploration towards promising actions based on the action-value function. Contrary to ϵ-greedy, the non-optimal actions are weighted based on their values. The most common softmax exploration approach uses the Boltzmann distribution,

$$\pi(a|s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{e^{\sum_{a' \in A} \frac{Q(s,a)}{\tau}}}, \tag{2-8}$$

where $Q(s,a)$ is the estimated mean of the rewards for action a, τ is the temperature variable, which controls the amount of randomness in the action selection, $\tau = 0$ means no exploration, $\tau \to \infty$ means taking actions randomly.

2-2-1-2 Directed approaches

Several directed exploration approaches aim at reducing uncertainty within the RL system. The main idea behind these exploration strategies is that the more uncertain the agent is about an action-value, the more it should focus on those regions of the action-state space, because they have the potential to turn out to be highly rewarding. This follows the optimism in the face of uncertainty heuristic in decision making. Uncertainty about a parameter can be described with confidence bounds in the frequentist approach or inferred from the posterior over the parameter in the Bayesian approach [12].

Frequentist methods for exploration Count-based methods directly use the state-action visit count to enforce action-selection that reduces uncertainty.

Upper-confidence bound based action selection is a count-based approach to uncertainty based exploration. The UCB1 [13] algorithm is a general purpose multi-armed bandit (MAB) algorithm, which balances exploration and exploitation and can also be used in the RL context. It applies a count-based approach to measure the uncertainty of the value estimate. Each-time the value of a state-action pair is updated the counter ($n_t(a)$) increases, and the uncertainty of that state-action value decreases. The action selection follows as:

$$a_t = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln N}{n_t(a)}} \right] \tag{2-9}$$

The first term is the empirical mean action-value estimate, which supports exploitation. The second term defines a confidence-interval and represents uncertainty about the estimate and supports exploration by increasing the weight of selecting less frequently selected actions. N is the total number of simulations, $n_t(a)$ is the number of simulations for specific action. The formula's connection to confidence interval's stems from the exploration term's connection to the one sided confidence interval of the mean reward, within which the true expected reward falls with high probability [14]. The UCB1 algorithm assumes the reward distribution to have a support of [0,1].

Bayesian approach for exploration In Model-free RL two approaches relevant for exploration are distinguished, parametric uncertainty and return uncertainty [15]. When considering parametric uncertainty, the $Q(s,a)$ is assumed to be random variable, since it is approximated from a finite number of observations. A posterior over the mean action-value function is maintained. The learned distribution is $p_\phi(Q|s,a,\mathcal{D}) = Q_\phi(s,a)$, where $\phi \sim p(\phi|D)$ are the action-value function approximate parameters. This is contrary to standard RL approaches, where $Q(s,a)$ is computed as a point estimate. In case of uncertainty of the return, the uncertainty is not only assumed on the mean of the action-value function, but the full return distribution is considered. No stochasticity in the environment is assumed, the uncertainty is induced by the followed policy. For this reason actions are picked optimistically with respect to the learned distributions of the return $p_\phi(Z|s,a,D)$.

Thomspon sampling Contrary to the UCB1 algorithm, Thompson sampling [16] is a probabilistic heuristic for balancing exploration and exploitation in multi-arm bandit problems. Thomspon sampling uses the Bayesian paradigm for selecting the action with the highest expected reward. This means that a belief distribution (e.g. Gaussian distribution) is maintained over the mean action-value for each possible action, which is iteratively refined with new observations.

$$\mathcal{N}(\bar{Q}_t(s,a_i),\ \frac{1}{n_t(s,a_i)}) \tag{2-10}$$

2-2-2 The perception problem

Tabular solutions for value function representation are only applicable to problems with small action and state spaces. In case of the high dimensional and continuous state spaces of

real world control problems these solutions no longer work. As the dimensionality of the RL problem increases, tabular representations become computationally intractable, therefore approximate solutions are needed. In large scale RL problems the agent needs to be able to generalize from experience of small part of the action-state space. Following the breakthrough results of deep neural networks for computer vision problems [17], in recent years neural networks have become the primary function approximators in high-dimensional RL problems.

Neural networks An artificial neural network is a programming paradigm that is inspired by information processing in the biological nervous system, applied for learning from data. The basic computational unit of neural networks are artificial neurons (figure 2-1). An neuron takes several weighted inputs, and computes a function (g) of the weighted sum of the received inputs. Neural networks are modeled as a collection of neurons, organized layer-wise [18]. The mapping $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$ from the network inputs ($\mathbf{x} \in \mathbb{R}^d$) to the network outputs ($\mathbf{y} \in \mathbb{R}^m$) is governed by a set of tunable parameters $\mathbf{w}$, which are trained based on a loss function $\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f(\mathbf{x}^{(i)}, \mathbf{w}))$. Where $\mathbf{x}_i$ and y_i are pairs from the training dataset $D = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$.

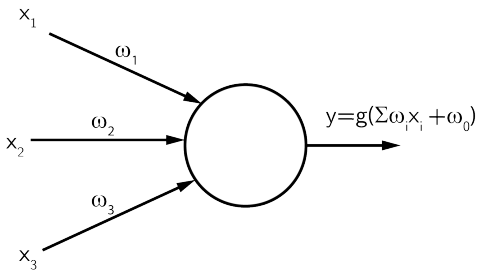


Figure 2-1: The artificial neuron model (from [18])

A deep neural network (DNN) is a feed-forward artificial neural network that has more than one layer of hidden units between its inputs and its outputs. Increasing depth in neural networks was inspired by how humans perceive the world, representing the world in terms of hierarchical structure: concepts at one level of abstraction as composition of concepts at lower levels [19].

2-2-2-1 Deep Reinforcement Learning

Deep reinforcement learning (DRL) combines deep neural networks with reinforcement learning. Contrary to previous RL approaches, in DRL the state features are learned from high-dimensional input data (e.g. images). DNNs can be for representing the state-action value functions [11], the policy [20] or for the dynamics model [21].

The DQN network The first DRL algorithm was the DQN network for the Atari 2600 games [11]. This was the first algorithm to demonstrate that a single model can be used to automatically learn policies for different environments and action sets. Four subsequent images from the game are used as a state representation. Using the images, and not a smaller set of game-specific state environments, results in a general framework for learning. On the

other hand this makes the RL problem very high-dimensional. To tackle the dimensionality problem a deep CNN based network $Q(s, a; \theta)$ is used for approximating the optimal action-value function $Q^*(s, a)$. Figure 2-2 shows the schematic view of the algorithm.

The RL problem is framed as a minimization problem with respect to the network parameters.

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta^-) - Q(s, a, \theta)\right)^2\right] \tag{2-11}$$

Previous attempts of using neural networks in RL had been unsuccessful due to convergence problems. The DQN algorithm was the first algorithm to provide a stable solution. The key ideas to the success of the DQN algorithm are the use of experience replay and using a a target network. θ^- are the parameters of the target network, which are updated periodically to stabilize the learning. The training data is drawn from a replay buffer of previously observed transitions (experience replay). Drawing random batches of samples from the replay memory was crucial, since training the neural network requires independent and identically distributed (i.i.d) training data.

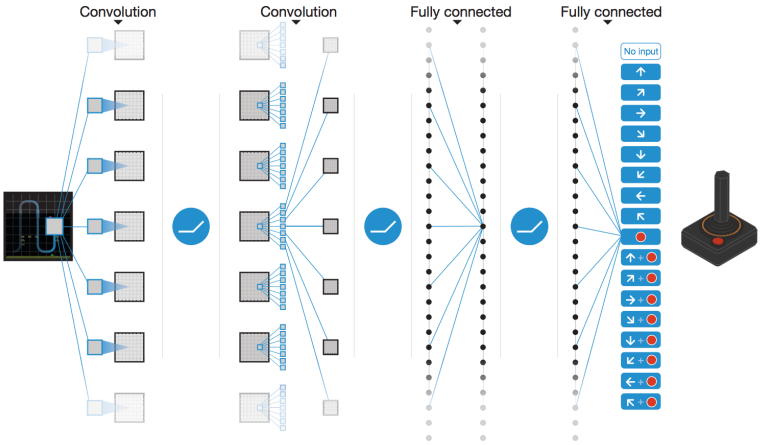


Figure 2-2: The DQN model from [11]

2-2-3 The long-term credit assignment problem

One of the key challenges of the RL framework is giving credit to actions, when the result of a given action is not immediate. This problem is also known as the long-term credit assignment problem in RL. Using eligibility traces can help assign credit to past states and actions [8]. Eligibility traces use a combination over n-step returns. It has been shown that using multi-step returns can significantly improve performance of the DQN algorithm [22].

2-3 Planning and reinforcement learning

Planning algorithms refer to a wide range of fields, which are considered with finding strategies for a sequential task for one or more decision makers [23]. In the reinforcement learning

context, planning involves looking-ahead from the current state and deciding for an optimal action based on imaginary trajectories. In contrast to learning, in planning the policy is improved without interacting with the real environment. Planning requires a generative model of the MDP, which makes it possible to acquire imaginary experience without taking actions. This model can be a learned environment model (e.g. DYNA-Q algorithm [24]) or in simulation-based planning a simulator of the environment [8]. The effectiveness of simulation-based search depends heavily on the accuracy of the model. In simulated-based planning the optimal action for the current state is computed through the estimated value function based on simulated experience. Search-based planning algorithms are decision-time planning methods, which compute the optimal action by looking-ahead, focusing on the current state. Search methods can be distinguished based on how the simulated experience is acquired [25].

2-3-1 Monte Carlo tree search

Monte Carlo search in planning Monte Carlo (MC) methods are computational algorithms, which rely on random sampling to acquire an estimate of a value based on stored statistics, e.g. means and variances. Monte Carlo search algorithms simulate complete episodes to estimate the state or state-action values. MC search is based on the assumption that sampling can be used to approximate the value of an action, and the the mean outcome of random simulations can be used to estimate action-values, $Q(s,a) = \frac{1}{n(s,a)} \sum_{i=1}^{n(s)} \mathbb{1}_i(s,a) v_i$, where $n(a)$ is the complete number of simulated games, $n(s,a)$ is the number of games, where the specific action was chosen, and $\mathbb{1}$ indicates if the action was selected in the ith simulation [26]. Monte Carlo control methods are decision-time planning algorithms, where random simulations are used to evaluate the value of actions and approximate the optimal policy in sequential decision making problems. MC control has been applied successfully in a variety of games, including non-deterministic and imperfect information games (e.g. Poker [27], backgammon [28]) and has also proven to be very effective in large state-space MDPs [3].

The Monte Carlo tree search algorithm Monte Carlo tree search (MCTS) denotes a family a MC control algorithms, where MC simulation is used to evaluate nodes of a search tree and subsequently use these values to select the best action. The optimal action is selected based the highest average return. MCTS decomposes the sequential decision making problems into a sequence of elementary decision making problems, which are modeled as multi-armed bandit problems. In MC search multiple trajectories from the current state to terminal states are sampled from the environment. In simple MC control these simulated roll-outs are governed by a fixed policy. In MCTS [3] the tree policy is improved via keeping roll-out statistics in a tree for guiding action selection. Every iteration the search tree is updated. In MCTS each iteration consists of four steps:

1. **Selection**: traversing the tree is guided by the tree policy π_{tree}, which is based on a variant of the UCB rule (2-9)
2. **Expansion**: expand a child node (action option) in leaf node
3. **Roll-out**: follow a random/informed policy π_{rand} until terminal state s_T
4. **Backup**: backup value to all parent nodes, update and store return (R_i) and visit counts (n_i) at each node

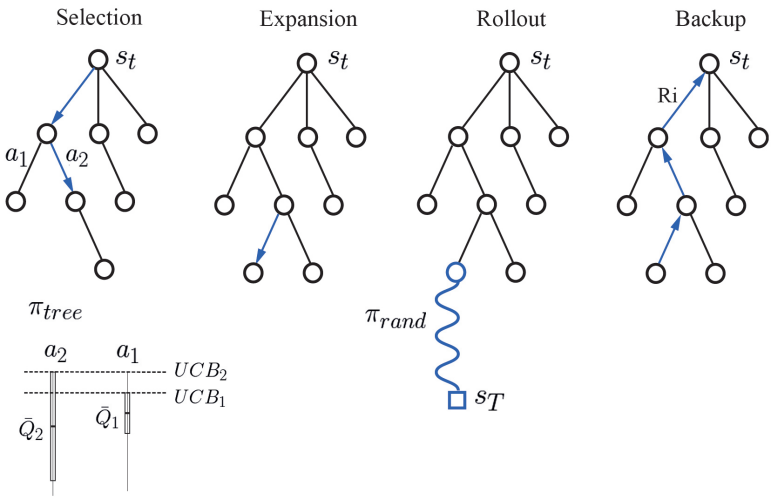


Figure 2-3: UCT algorithm, Monte-Carlo tree search with UCB tree policy. One iterations steps.

As the search tree grows with the MCTS search iterations, the estimated values become more accurate. The convergence rate depends on the RL domain state and action space complexity. The iterations are usually halted before converging to the optimal values and the iteration is run until a pre-defined computational limit or iteration number. The action which has the highest count is selected for the current state (root node).

Tree policy Key to the success of the MCTS algorithm is the tree policy π_{tree}. The most commonly used tree policy is acting greedily with respect to the upper confidence bound for trees algorithm (UCT) [29], which is based on the UCB1 algorithm [13]. As in the UCB1 algorithm, the UCT algorithm includes exploitation through the the estimated value of the state-action pair, and exploration through the visit counts term. The UCT algorithm takes three parameters, the number of iterations (trajectories), the maximum depth on one roll-out and a exploration-exploitation trade-off parameter. The UCT algorithm also includes an exploration-exploitation balancing constant c_{UCT}, which is usually chosen in the range of [1,2].

$$UCT(s,a) = \frac{\sum_{i=1}^{n(s,a)} R_i(s,a)}{n(s,a)} + c_{UCT}\sqrt{\frac{\log n(s)}{n(s,a)}} = Q(s,a) + c_{UCT}\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{2-12}$$

Applications of the MCTS algorithm MCTS is applicable for problems with large branching factors (large action space), where exhaustive search is computationally inefficient, by reducing the search breadth through the UTC formula. The main shortcomes of the MCTS algorithm is poor performance, when the number of trajectories and maximum search depth are low relative to the application domain and when rewards are sparse.

2-3-2 Combining UCT-based planning with deep learning

After the success of deep learning in learning feature representations from data for passive computer vision tasks many approaches has been taken to incorporate the generalization

capability of deep neural networks in sequential decision making problems. In the first approaches, deep learning was combined with model-free RL algorithms (e.g DQN) for the Atari games. One of the main drawbacks of the DQN algorithm is it's need for a large amount of samples. Subsequently, [30] examined the possible performance gains from combining UCT-based planning with deep learning also in the Atari games building on the DQN algorithm. In contrast to DQN, UCT-based agents achieve sample efficiency by relying on simulation based policy improvement. They identified computational speed as a key drawback of the original UCT-based agent compared to the DQN algorithm. The UCT-based agent does not build an explicit policy, thus requires more computational time for each action selection. They showed that by slowing down the game, thus giving more computational for the algorithm, the UCT-based algorithm performs significantly better than the DQN algorithm. This motivated combining deep learning with UCT-based planning. They found that using planning combined with deep learning achieves significantly better performances than the DQN algorithm over all the examined games. Based on the DQN algorithm, they examined several approaches of combining an UCT-agent with deep neural networks. A common aspect of these algorithms is the one-way connection from the tree search to the network. The trained network was not utilized in the MCTS search and there is no policy iteration in the MCTS component of the algorithms. Most recently, the AlphaGo [6] and AlphaZero [7] algorithms have achieved unforeseen success in the field of AI for games. The AlphaGo has achieved superhuman performance for the game of Go for the first time. In parallel with the AlphaZero algorithm, the ExIt algorithm also uses a similar structure of combining MCTS with deep learning for a sequential decision making problem. In both of these algorithms, there is two-way interaction between the generalization, deep learning component and the planning, MCTS component. In the following section the AlphaZero algorithm will be described. However, MCTS has a major drawback of evaluating each state independently, without generalization between states. and high variance estimates [25].

2-3-3 The AlphaZero algorithm

The game of Go had been a challenge in AI research, because due to the large branching factor of the game, brute-force search methods did not manage to achieve good results.
AlphaGo [6] achieved a break-through results by using a policy iteration framework that combines MCTS and deep neural networks.

The first AlphaGo [6] algorithm used supervised learning, it relied on example moves from expert human players to learn the optimal policy. The AlphaZero [7] algorithm was a follow-up for the original algorithm. It retained combined deep learning and MCTS, but the human expert moves were removed from the learning. The agent learns to play by self-learning, starting from completely random play. By using neural networks, knowledge from the MCTS tree can be transferred to the next iteration, the algorithm receives a gradually improving training signal. One of the key differences of the AlphaZero algorithm compared to previous algorithms, combining UCT-based planning and deep learning is the incorporation of policy improvement in the algorithm. There is a two-way connection between the deep neural network and the MCTS components of the algorithm. While [30] uses fixed MCTS, in the AlphaZero algorithm, policy recommendations from the neural networks are used in the MCTS search.

The neural network structure of the AlphaZero agent In model-free DRL a neural network is usually used to approximate the action-value function (equation 2-3), mapping a state-action pair to the estimated cumulative reward from the current state-action pair (e.g. the DQN algorithm [31]). In the case of AlphaZero, a two-headed deep convolutional neural network is used to map the current state s_t to action probabilities $\pi_\theta(s_t)$ and the estimated cumulative reward $V_\theta(s_t)$ from the current state.

The two deep convolutional neural networks outputs are used by the algorithm to reduce the search space and make the search tractable.

1. **Policy network** $\pi_\theta(a|s)$ reduces the search breadth. It is responsible for action selection recommendations. The input is the state (position representation of game layout, $\mathbb{R}^{19\times19\times48}$) and the output is a softmax layer of policy distributions of available actions for both players ($\pi_\theta(a|s) \in [0,1]^{19\times19}$ with $\sum_a \pi(a|s) = 1$). By only considering moves recommended by the policy network the search space of MCTS is reduced.

2. **Value network** $V_\theta(s)$ reduces the search depth. The value network aggregates knowledge together to a scalar output, the prediction for winning the game from a given state. The standard MCTS algorithm uses rollouts in unexplored parts of the state space to acquire a value estimate (rollout on figure 2-3). In case of large search spaces random rollouts do no provide useful information due to high variance. By replacing sub-trees with a single number from the value network, the search depth is reduced.

The state represents a position on the game board and additional features about each board position $s \in \mathbb{R}^{19\times19\times48}$. The MCTS tree provides the dataset for the neural networks to train on $D = \{s_t, \hat{\pi}_t, \hat{V}_t\}$. Where s_t is the current state, $\hat{\pi}_t$ is the policy estimate for s_t and $\hat{V}_t$ is the final outcome of the game, $\hat{V}_t \in \{-1, 1\}$, for winning or losing the game. The policy estimate is calculated as the normalized action counts of the root state in the search tree:

$$\hat{\pi}_t = \frac{n(s,a)^\tau}{\sum n(s,a)^\tau}, \tag{2-13}$$

where τ is the temperature parameter, which controls the randomness of action selection and therefore the exploration-exploitation trade-off.

The loss is defined as the sum of cross-entropy loss for the policy network and mean squared error of the value network.

$$L(\theta) = \sum_t (V_\theta(s_t) - \hat{V}_t)^2 - \hat{\pi}_t log(\pi_\theta(s_t)) \tag{2-14}$$

MCTS search in the AlphaZero algorithm MCTS is used within the training loop to look ahead before each real environment step, and using the environment simulator, return the estimated value and the probability of possible actions from the current states. The action is then selected based on the root action counts.

The AlphaZero algorithm uses a modified version of the UCT tree policy (equation 2-12) in the MCTS tree search:

$$UCT\text{-}\alpha 0(s,a) = \bar{Q}(s,a) + c_{UCB} \cdot \pi_\theta(a|s) \frac{\sqrt{n(s)}}{n(s,a)+1} \tag{2-15}$$

The MAB action selection within the tree in the AlphaZero algorithm becomes:

$$\underset{a}{\operatorname{argmax}} \Big[\bar{Q} + c \cdot \pi_\theta(a|s) \cdot \frac{\sqrt{n(s)}}{1+n(s,a)} \Big] \tag{2-16}$$

There are two main differences compared to the baseline UCT algorithm. First, compared to the baseline UCT algorithm, policy recommendations from the neural network are used in the node selection during the tree search. The relative action selection count for the given node in the tree is scaled by the estimated action probabilities output $\pi_\theta(a|s)$ from the neural network of the AlphaZero agent. Second, there is an additional exploration bias in the exploration term. Expansion of all child nodes is prevented by adding a plus one the counts of all possible actions from the state.

At the end of the MCTS search, the child action with the highest number of simulations is selected. In the AlphaZero algorithm, the search tree is kept between subsequent time steps. The subtree of the selected action child node is retained along with all its statistics, while the rest of the tree is discarded.

Chapter 3

The transferability of the AlphaZero algorithm

The AlphaZero algorithm has achieved unforeseen success for the game of Go. The question arises, how well-suited is the algorithm for other RL tasks and what factors could hinder transferring the algorithm to other RL environments. The game of Go can be described as a fully-observable episodic MDP, with a discrete action space, deterministic transition function and [0,1] bounded returns. This chapter will focus on how changes in the reward function affects the algorithm's performance and examines the applicability of the AlphaZero algorithm for fully observable, single player, deterministic reinforcement learning environments with reward distribution, whose support is not in [0,1]. First, the problems stemming from the changed reward distributions is described by examining the assumptions of the AlphaZero algorithm. Following, modifications are suggested, which could face these challenges and render the algorithm more suitable for the new environments. This chapter answers the first research question (R1).

3-1 Incompatibilities between the AlphaZero algorithm and environments with unbounded returns

The game of Go has a discrete, bounded reward function, where the agent is rewarded from {-1,1} for either losing or winning the game and 0 for every non-terminal step. This reward definition has a key importance in the UCT-$\alpha 0$ tree policy of the AlphaZero algorithm, which builds on the UCT algorithm. The UCT formula has its roots in games, where in general, rewards are sparse and the value estimate is treated as a random variable with a bounded support included in [0,1]. In contrast, RL environments often have returns outside [0,1] and problems can arise when applying naively the UCT formula. Many RL environments have step-wise, unbounded reward functions and applying the AlphaZero algorithm needs some further considerations. This section identifies possible problems, which might occur when applying the AlphaZero to these environments and examines the performance of the original AlphaZero algorithm in these environments.

3-1-1 Motivations for the UCT-$\alpha 0$ tree policy in the game of Go

The main challenge of solving the game of Go for AI algorithms is the high action branching factor (in general more than 200 moves per state). The MCTS algorithm's strength comes from the focused search. Solving Go required further increasing the sparsity in action selection during MCTS. This motivated modifying the tree policy of AlphaZero, compared to the the original UCT algorithm (equation 3-1). Two modifications were made in order to achieve sparsity in action selection.

$$UCT(s,a) = \bar{Q}(s,a) + c_{UCB} \cdot \frac{\sqrt{n(s)}}{n(s,a)} \tag{3-1}$$

The first modification was to include the policy network output in the UCT formula. The role of the policy network $\pi_\theta(a|s)$ is giving action selection recommendations for the given state-action node. In the AlphaZero algorithm, the policy network output is included in the exploration term. The policy network gives suggestions through the exploration term, which action to choose. As the learning progresses, the policy network's output becomes increasingly closer to one-hot encoding, therefore reducing the search space in MCTS and introducing sparsity in action selection (equation 3-2).

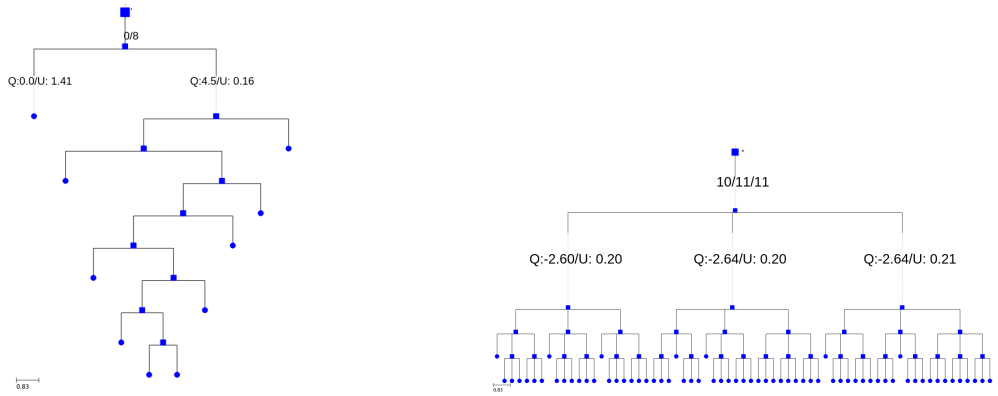
$$UCT\text{-}\alpha 0(s,a) = \bar{Q}(s,a) + c_{UCB} \cdot \pi_\theta(a|s) \frac{\sqrt{n(s)}}{n(s,a)+1} \tag{3-2}$$

The second modification for action sparsity was introducing an exploration bias term in the exploration term of the UCT algorithm. In contrary to the original UCT formula, this +1, added in the denominator, prevents expanding all child nodes, therefore also leading to action sparsity. By tailoring the UCT formula, the AlphaZero algorithm managed to have sparsity in the action, which facilitated solving the game of Go. These modification made sense in the case of Go, where the action space dimension was very large, and expanding all options would have been (even more) computationally demanding. For small scale RL problems with unbounded returns this reasoning does not hold, and in fact can even be harmful for the algorithm's performance on these domains.

3-1-2 Incompatibility of UCT-$\alpha 0$ tree policy with general reward functions

The reward definition plays a significant role in the performance of the RL algorithm for a given task. RL environments often have step-wise reward definitions, where the agent receives a positive or negative reward for every action taken. For example step-wise negative rewards often occur, where there is some kind of resource limitation for solving the task. Examples for this kind of reward definition often arise in robotics problems, e.g. navigation or manipulation tasks. This step-wise reward can be negative unit reward or distance based reward, where the reward is defined as the negative distance from the goal. Both cases result in negative unbounded returns. This is in contrast to the usual game domain reward definition, where in general unit rewards are given at the end of the game, and therefore the unknown reward distribution has a bounded support in [0,1]. The absolute episodic returns from these environments can get very high, compared with the general [0,1] episodic return of games. These kind of reward definitions might render the UCT-$\alpha 0$ formula inefficient, as explained in the next paragraph.

Initialization problem In the beginning of MCTS search, the AlphaZero algorithm initializes all child action values to zeros in the root of the current state node. This is a sensible initialization, when returns are within [0,1] range. In case of RL environments with returns outside the [0,1] range, this initialization does not reflect the real action values. After initialization, the first root action child is expanded randomly and the value estimate of this child action is updated based on the simulation. Zero initialization becomes a problem in the next steps, as there is now a discrepancy between the expanded child action's updated value estimate and the rest of the action's zero initialized values. This initialization bias influences the progressing of the MCTS algorithm. Figure 3-1 illustrates the problems arising from inaccurate value initialization in the AlphaZero algorithm.



(a) tree build in cart-pole problem **(b)** tree built in mountain car problem

Figure 3-1: Illustration of problems arising from zero value initialization of child actions in MCTS search tree when the original AlphaZero algorithm is applied to environments with returns out of the [0,1] range. 3-1a shows an exampled of early greedy behaviour in an environment with step-wise positive rewards. 3-1b shows a non-sparse behaviour in an environment with step-wise negative rewards.

Figure 3-1a shows the tree built in the first step's MCTS search, when the AlphaZero algorithm is applied for the cart-pole problem (A-1), which has a step-wise positive reward. In this case the zero initialization of the action values is too low, compared to the returns of the environment. After the first node is expanded randomly at the beginning, the algorithm follows this trace greedily according to the UCT-α formula, without expanding other action(s) at the root. Figure 3-1b shows the tree built in the first step, when the AlphaZero algorithm is applied for the mountain-car problem (A-1), which has a step-wise negative reward. In this case the zero initialization of child action values is too high, and results in expanding all the child nodes of the root. This does not necessarily result in failure of learning, but action selection sparsity is lost, and computation budget is not allocated efficiently.

Scaling problem In case of returns with a support outside of [0,1], a scaling problem can occur between the exploration and exploitation terms in the UCT-$\alpha 0$ (equation 3-2) tree policy. In the $UCB1$ algorithm and its variants, the value estimate is a random variable and the exploration term represents a measure of uncertainty of the value estimate in the form of the upper confidence bound. In order to have exploration properly function, the exploration and exploitation terms should be on the same scale. When the action value estimates no longer

have a bounded support included in [0,1], there is a scaling problem, and exploration might be hindered by having non-overlapping confidence bounds from the beginning of learning.

3-1-3 Results for applying the original AlphaZero algorithm for environments with unbounded returns

Figure 3-2 shows the results for the cart-pole problem, which has a step-wise positive reward (A-1). The episodic rewards plot shows that the agent fails to learn in the the environment. This is not surprising, since as it could be seen on, the agent exhibits a quasi-random behaviour as a result of inaccurate value initialization (3-1-2). Certainty in a chosen action an be described by the entropy A-4 of the normalized root action counts. The entropy plot on figure 3-2 illustrates the greedy behaviour stemming from the too low value initialization. The entropy is low from the beginning of learning, since the too low action value initialization prevents expanding the rest of the child nodes. The entropy plot is in accordance with the tree illustrates for the first step of learning in the cart-pole environment on figure 3-1a. The loss plot on figure 3-2 indicates that compared to supervised learning, in case of deep reinforcement learning the neural network loss is not a good indicator of the algorithm's performance. As the agent explores, the loss function remains high. In general, interpreting the neural network loss in DRL is not as straight-forward as in supervised learning.

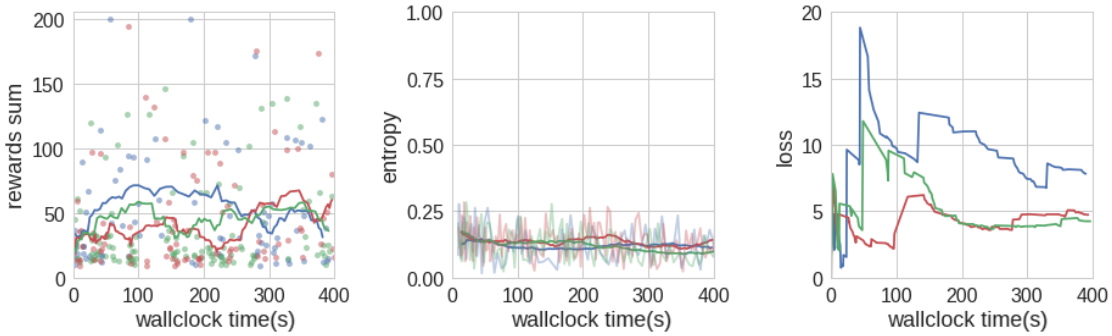


Figure 3-2: Episodic total reward, episodic mean decision entropy, neural network loss results for original AlphaZero algorithm applied cart-pole problem for 3 iterations

3-2 Modifications to original AlphaZero algorithm

In this section some possible modifications to the AlphaZero algorithms are recommended, which confront the incompatibilities between the AlphaZero algorithm and environments with returns outside the range of [0,1]. As stated before, problems can stem from differences in the reward functions in RL environments compared to the game of Go. The AlphaZero tree search formula (equation 2-16) assumptions might be incompatible non-game reward functions, where the reward distribution does not necessarily has a bounded in support in [0,1]. In the previous section two main problems were identified with the UCT-α formula: the initialization problem and the scaling problem. In this section, first algorithmic modifications are suggested for the initialization problem, then possible modifications, which resolve the scaling problem between the exploration and exploitation terms in the tree policy.

3-2-1 Solving the initialization problem

These modifications aim at resolving the initialization problem of the UCT-α algorithm. As seen before, initializing the roots child action values to zero can be very inaccurate in RL problems and hinders the functioning of the MCTS algorithm. The goal is to initialize the action values to sensible values for the RL environment, without using prior knowledge about the value ranges in the algorithm.

Removing the exploration bias As seen before, the AlphaZero algorithm has an additional exploration bias term in the exploration term of the UCT-$\alpha 0$ formula in contrast to the original UCT algorithm. While in the UCT algorithm, all child nodes were expanded, since the exploration term is treated as infinity in case of $n(s,a) = 0$, the exploration bias in the AlphaZero algorithm prevents expanding all child nodes. This makes sense in case of Go, where for the large action space, expanding all child nodes would be computationally inefficient. However in smaller scale problems, this behaviour is not necessarily wanted, and removing the exploration bias might prove beneficial and could prevent biasing towards a suboptimal policy early on. By removing the exploration bias, the modified action selection rule in the MCTS tree becomes:

$$UCT\text{-}\alpha 0_{mod,I} = \bar{Q}(s,a) + c_{UCB} \cdot \pi_\theta(a|s) \frac{\sqrt{n(s)}}{n(s,a)} \tag{3-3}$$

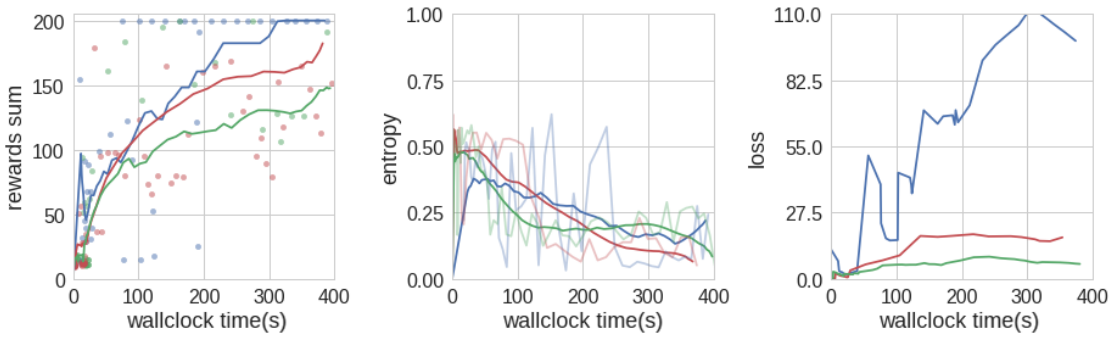


Figure 3-3: Episodic total reward, episodic mean decision entropy, neural network loss results for AlphaZero algorithm applied cart-pole problem, with exploration bias removed

Figure 3-3 show that removing the exploration bias does bring performance improvements for the cart-pole problem by forcing all action nodes to be expanded. The entropy plot shows that with the exploration bias removed, the entropy starts relatively high and decreases during the learning process, as the Q estimates become more accurate.

Action-value initialization in search tree Instead of zero, child action values could also be initialized by the parent's state mean value estimate. This can prevent ignoring actions, because their state-action value is too low, compared to the already expanded action's.

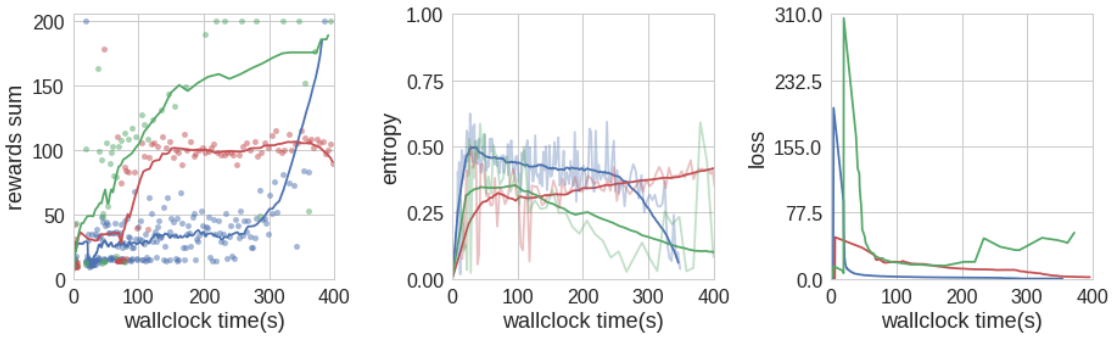


Figure 3-4: Episodic total reward, episodic mean decision entropy, neural network loss results for AlphaZero algorithm applied cart-pole problem, action values initialized with parent's value, results for 3 iterations

Figure 3-4 shows that initializing the child action-values with the parent's value does bring some improvement compared to the original algorithm on the cart-pole problem. On the other hand, it also makes the algorithm more prone to be stuck in local minima and this actually de-accelerates/halts the learning process.

3-2-2 Solving the scaling problem

With the initialization methods suggested in the previous section, extreme cases of greedy and non-sparse tree policy behaviour can be avoided, but a scaling problems still remains a scaling problem between the exploration and exploitation terms in the UCT-$\alpha 0$ formula. Several approaches can be taken to confront the scaling problem. Normalization can be achieved by leveraging prior knowledge about environment and it's reward function. Rewards can be divided by a fixed value from the return range, e.g. the mean or maximum episodic return in each step or the exploration-exploitation balancing hyperparameter, c_{UCB} parameter of MCTS can be tuned to the environment's return range. If no prior knowledge is assumed, there is the option to use adaptive normalization on the returns or adaptively tuning the c_{UCB} parameter. In the rest of this section reward function modifications are examined, first assuming prior knowledge, then adaptive normalization without prior knowledge.

Modifying the reward function Figure 3-5 shows the results for the AlphaZero algorithm's performance on the cart-pole problem with a modified reward function (A-1), where each step's reward was divided by the maximum episodic return. In this case the agent receives normalized rewards from the environment emulator to bring the Q and U values of equation 3-2 to the same scale. The episodic reward plot shows that the algorithm's performs better on the normalized environment, compared to the unnormalized (figure 3-2) and similarly to the removed bias (figure 3-3), the decision entropy decays gradually, as the agent learns. However, the plots also show that the agent's performance is not stable. Normalizing the reward function with a the maximum episodic return is not ideal. It causes low value estimates in the MCTS search of the first environment steps, therefore we might achieve the opposite of the original scaling problem, now the exploration term dominating with too low value estimates.

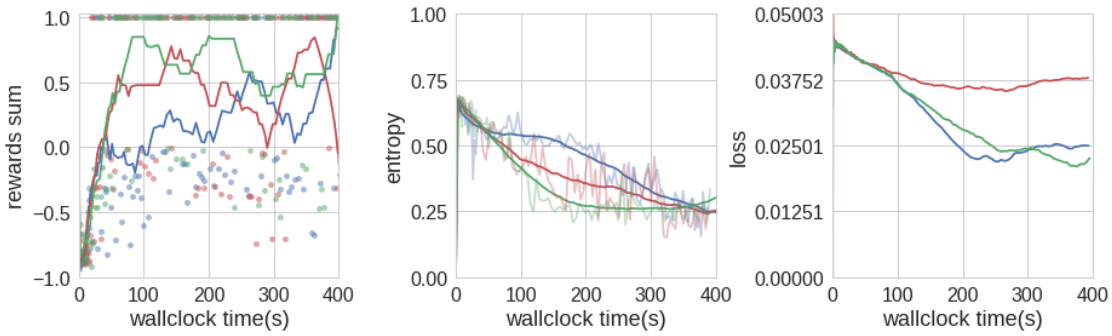
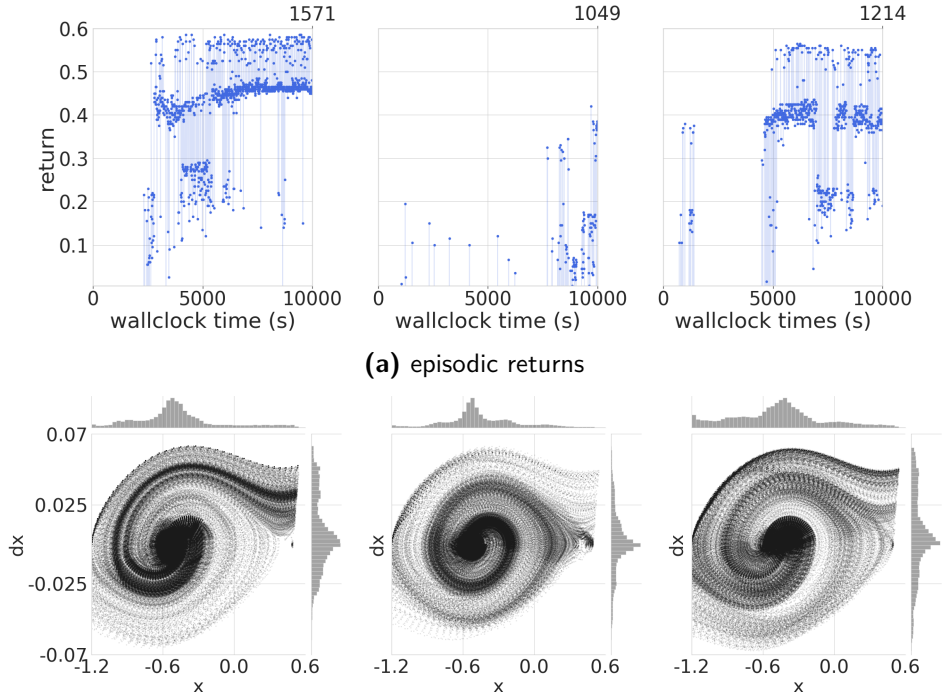


Figure 3-5: Episodic total reward, episodic mean decision entropy, neural network loss results for AlphaZero algorithm applied cart-pole problem with normalized reward function, with c_{UCB} =0.5, results for 3 iterations

Modifying the reward function in order to bring the exploration and exploitation terms to the same scale has also proven efficient in case of the mountain-car problem. Escaping the local minima in the mountain-car problem is a a relatively challenging task exploration wise. Having a non-sparse action selection behaviour (figure 3-1b) can prevent learning under a fixed time constraint, by not utilizing the given resources efficiently. Figures 3-6 illustrates the effects of improved exploration on the algorithm's performance in the mountain car problem.



(a) episodic returns

(b) Visualization of visited state space points during learning, x axis is position, y axis is velocity of car, traces show agent learns policy to escape local minima of valley

Figure 3-6: Illustration of performance of the AlphaZero algorithm for the mountain-car problem after dividing the step reward with the maximum episodic return. Plots show episodic total rewards and states space exploration for the normalized mountain-car problem for for 3 different seeds. The number of executed episodes is indicated on the reward plots.

Adaptive normalization In RL action values can span a wide range, which is often unknown prior to to learning process. This non-stationary nature of the values makes it difficult to use the standard normalization techniques of machine learning. Adaptive return scaling has not been studied extensively in DRL literature. Pop-Art [32] (Preserving Outputs Precisely, while Adaptively Rescaling Targets), was introduced as a scale-invariant algorithm for value-based RL. the Pop-Art algorithm aims at maintaining a normalized reward distribution with zero mean and unit variance throughout learning by treating normalization as a separate learning process. The main motivation was to remove domain knowledge from the algorithm. Removing reliance on prior domain knowledge is not only useful in single domains, but also in multi-task learning, where different tasks can have different reward ranges, which would be difficult to solve with an algorithm, which is not invariant to reward scales [33]. A Pop-Art-like approach would be also beneficial when applying the AlphaZero algorithm to environments with returns outside of [0,1]. However, learning to predict normalized values in the AlphaZero algorithm can cause discrepancies in the value scales in the backup step of MCTS. In the AlphaZero algorithms value estimates in the UCT-α action selection formula can be scaled adaptively, by utilizing the first and second moments of the value estimates.

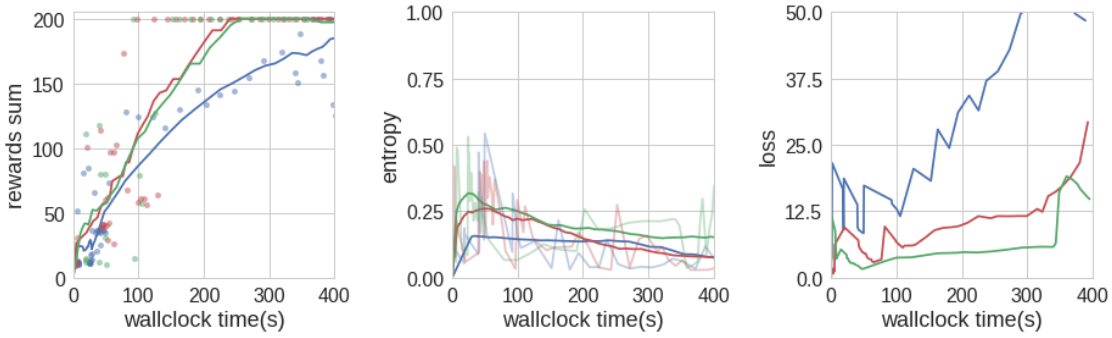


Figure 3-7: Episodic total reward, episodic mean decision entropy, neural network loss results for AlphaZero algorithm applied cart-pole problem with normalized adaptive value scaling, with c_{UCB} =0.5, results for 3 iterations

$$R_t = \frac{1}{n_{MCTS}} \sum_{i}^{n_{MCTS}} R_i, \tag{3-4}$$

$$\mu_t = (1-\beta)\mu_{t-1} + \beta R_t \tag{3-5}$$

$$\nu_t = (1-\beta)\nu_{t-1} + \beta R_t^2 \tag{3-6}$$

$$\sigma_t = \sqrt{\nu_t - \mu_t^2}, \tag{3-7}$$

where R_i is the i^{th} root return during tree search,μ_t is the first moment, σ_t is the second moment and $\beta \in [0,1]$ is the step size parameter. Keeping track of the first and second moments of the value estimates, the action values can be normalized as following in the UCT-$\alpha 0$ action selection formula.

$$UCT\text{-}\alpha 0_{mod,II} = \frac{\bar{Q}_t(s,a) - \mu_t}{\sigma_t} + c_{UCB} \cdot \pi_\theta(a|s) \frac{\sqrt{n(s)}}{n(s,a)+1} \tag{3-8}$$

Figure 3-7 shows that adaptive normalization has an improved performance and stability compared to normalizing with a fixed value. The modification 3-8 was only applied in the

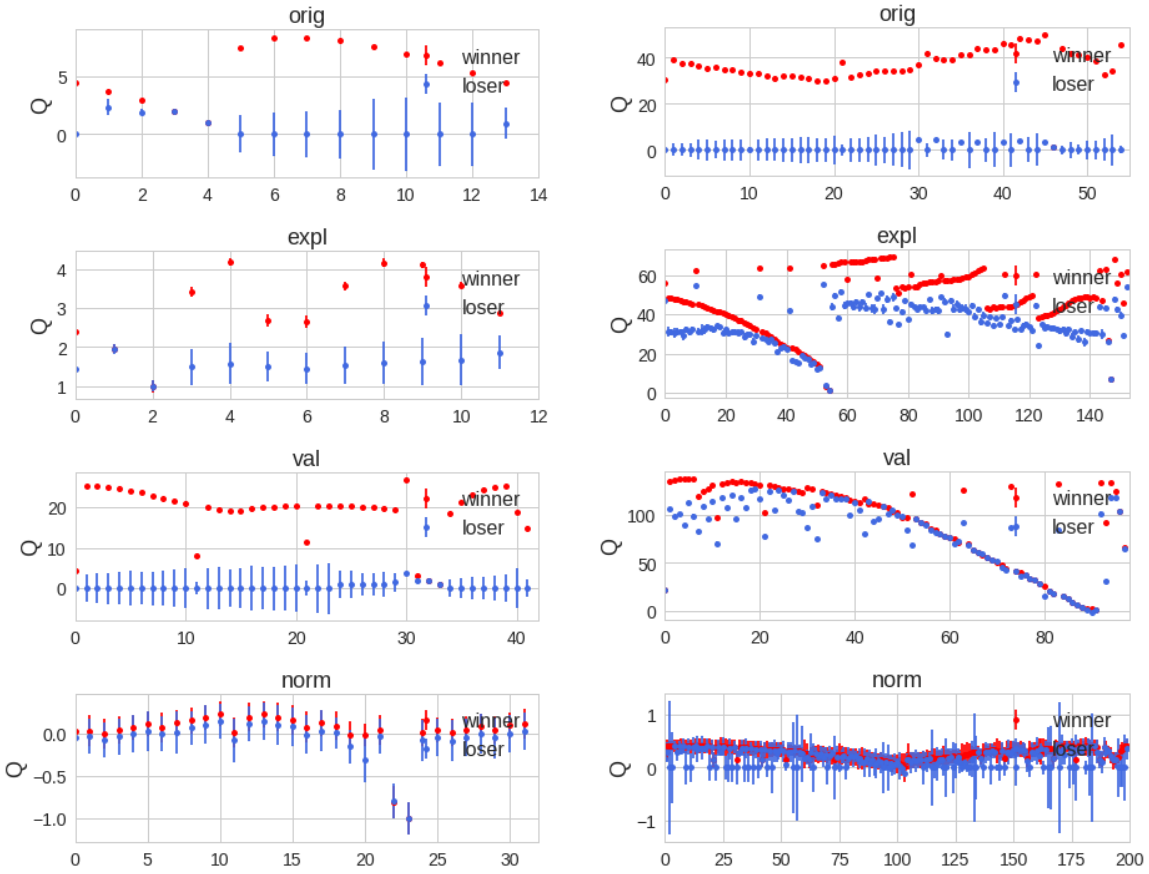
action selection within MCTS (figure 2-3). It is also an option to learn the normalized values with the neural network. This would have multiple advantages. First, it has been reported that learning normalized values is beneficial for backpropagation during neural network training [19]. Second, value ranges can be crucial from a learning dynamics perspective. Large return ranges can also be a problem in the learning dynamics for the AlphaZero algorithm. The AlphaZero has a two-head neural network with value and policy estimate outputs and a additive loss function. In multi-task learning, large differences in value ranges can cause unintended emphasis on tasks through difference in the value updates [33]. A Pop-Art-like approach could be taken to normalize the value network predictions. However this requires more considerations in deep learning combined with UCT-based planning, since backups in MCTS search uses outputs from the value network during node evaluation along with unnormalized external rewards from the environment emulator.

3-3 Effects of algorithm modifications on the action value means and UCB bounds

The main problem of unbounded rewards functions is that the difference of the mean value estimates for the current state's child action can get large early on in the learning process, when the state-action value estimates are probably still inaccurate. The following plot illustrates the problems with this. Figure 3-8b shows the mean value estimates for the MCTS root's child actions in different learning phases. It can be seen that for the original algorithm, the mean values estimates are far from each other and the confidence bounds for the mean value estimates do not overlap. Removing the exploration bias and initializing the action value with the parent's state value solves the mean problem, but the confidence intervals are still very small relative to the mean. Normalizing the reward function offers a solution for the confidence interval problem. The last row of figure 3-8b shows the intended behaviour of the UCT tree search formula, with the confidence bounds overlapping in the beginning of learning.

3-4 Discussion for the problem of unbounded reward functions

This chapter examined the applicability of the AlphaZero algorithm for RL environments, where the returns do not have a bounded support in [0,1] due to the external reward function of the MDP. It was shown that disregarding value ranges can cause failure in learning. The roots of this problem were identified as the initialization and scaling problems in the UCT-$\alpha 0$ tree search formula and modifications were suggested to circumvent these problems. It was shown that after applying these modifications, the the AlphaZero algorithms has an improved performance. The scaling between the exploitation and exploration terms is crucial to have balanced exploration. Normalizing the reward function with a fixed value is an option to solve this problem. However this is not ideal for two main reasons. First, it is difficult to decide what this value should be, since the returns can take a wide range of values. Second, the reward function is considered to be a fixed part of the environment MDP, external to the agent. According to the RL framework, the agent should not have knowledge about the rewards, prior to learning. An adaptive normalization technique was suggested to confront



(a) episode at beginning **(b)** episode from mid-learning

Figure 3-8: Importance of value initialization and scaling in the AlphaZero algorithm shown through the comparison of resulting Q value estimates in the original and modified algorithms. Step-wise value estimates are shown in the cart-pole environment from episode in the beginning and middle of training. The values and their confidence bounds (equation 3-2) are shown for both actions in the cart-pole problem. Winner denotes chosen action, loser denotes other action. Abbreviations denote 'orig' for original algorithm, 'expl' for exploration bias removed, 'val' for action value initialization and 'norm' for normalized reward function. The value intialization problem can be observed in the original algorithm, as zero initialization causes random preference for actions in case of the step-wise positive rewards. The scaling problem be observed by comparing the normalized results with the other results. Overlapping confidence bounds, which support exploration are only present in the normalized case.

this problem. This was shown to have an improved performance for the cart-pole problem and will be used for other environments in the next chapter.

The scaling problem between the exploration and exploitation terms in the UCT-$\alpha 0$ formula points to a general problem of unknown return ranges in value-based RL. Return ranges become especially important in multi-task RL, where multiple reward functions are present in the RL system and different return ranges can affect the learning dynamics.

Transferring the success of the AlphaZero to other domains raises several questions. One of this questions is given an environment with a reward distribution, which no longer has a

support in [0,1], what problems arise in the algorithm, and if the algorithm can still function properly. This chapter showed that the main problems from this altered reward distribution come forward in the MCTS component of the algorithm, more specifically in the $UCT\text{-}\alpha 0_{mod}$ tree policy. A reward distribution out of the range [0,1] causes problem in the initialization of the MCTS search and in the exploration-exploitation balancing. Possible solutions have been offered to these problems and it has been shown that the performance of the AlphaZero algorithm improves after applying these. However, some of these solutions introduce unwanted bias and prior knowledge to the algorithm. One of the key aspects of the AlphaZero algorithm is self-learning. The AlphaZero agent managed to solve the game of Go, with limited prior knowledge, only having access to the rules of the game. Adaptive normalization of the returns and value network predictions could be a promising approach to limit the need for prior knowledge, and remove the introduces additional bias.

Chapter 4

Generalization versus locality in the AlphaZero algorithm

The AlphaGo algorithm was first to combine the strength of local planning of the UCT algorithm and the generalization power of deep learning with the aim of achieving top level professional performance in the game of Go. The strength of UCT algorithm is attributed to focused local search, where computational capacities are only spent on the most promising regions of the state space. However in case of large state and/or action space and limited search time, keeping only local information is not efficient and it becomes necessary to share knowledge, to generalize across states. Sharing values estimates in MCTS has been previously achieved by grouping states according to expert knowledge or heuristics [34]. The AlphaGo algorithm employs a deep neural network to achieve generalization in MCTS.

The AlphaGo algorithm was created specifically for the game of Go, but the use of deep neural networks in planning algorithms have further implications and could have a great potential in other planning problems. One interesting aspect of the combined use of planning algorithms and deep learning is the allocation of resources on improving generalization versus concentrating on evaluating the current step. Under strict time budgets this question becomes increasingly relevant. The hypotheses of the second research question of this thesis is that under a fixed budget, there exist an optimum balance of UCT and deep learning. Moving away towards either building a larger tree, or focusing on building a larger tree hurts the performance. The question of balancing local search and generalization will be investigated through examining the performance while increasing the MCTS tree size for different RL environments under a fixed time budget. AlphaZero is based on the AlphaGo algorithm, which generalized the success of AlphaGo to chess and shogi [7]. The chapter has the following layout: first, background information is given on the two-component structure of the AlphaZero algorithm while showing references of AlphaZero's combined deep-learning UCT-based architecture to more general frameworks and properties of global and local information is briefly discussed. Second, experimental results are presented and results are discussed for the second research question of examining the globalization-locality trade-off in the AlphaZero algorithm. This chapter answers the second research question (R2).

4-1 The AlphaZero algorithm as an interplay of two systems

The AlphaZero algorithm has two main components: tree search and deep learning. The motivation for incorporating a deep neural network in tree search in the algorithm was to make the computationally challenging search problem of the game of Go more tractable. The success of achieving professional human level performance on the game of Go can be attributed to the reduced search depth and breadth through the use of the value and policy networks. The search breadth is reduced by utilizing the policy network's output in the tree policy and the search depth is reduced by substituting the rollout step of the MCTS algorithm with the value estimate from the value network. This two-component system can also be examined as a general framework, outside of the realm of the game of Go. The ExIt (Expert Iteration) algorithm [35] has a similar combined tree-search neural network architecture to the AlphaZero algorithm, but different motivation. The ExIt algorithm was inspired by the dual-process theory [36], which models human reasoning as an interplay between a "slow system" for planning and a "fast system" for intuitive behaviour. Simulation-based tree search is an analogue of the slow system, while neural network is the analogue for the fast system. The motivation for using neural networks in tree search is similar to AlphaZero's motivation: gaining "intuition" about the problem can fasten up the search. In both algorithms learning is an iterative process. The deep network, trained on examples from tree search, provides increasingly good guidance for the tree search. Figure 4-1 shows the schematic view of the AlphaZero algorithm.

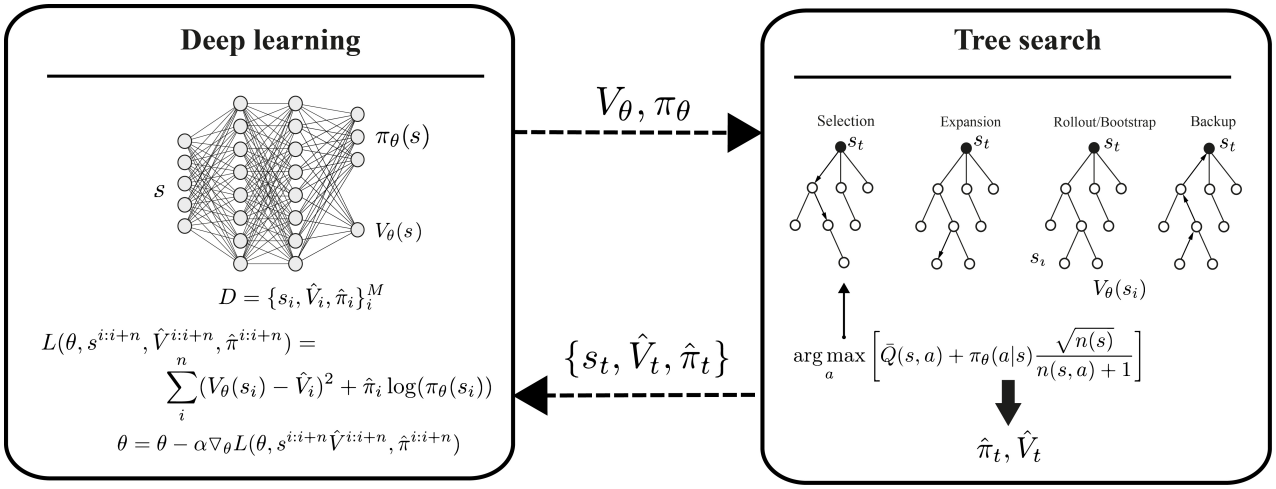


Figure 4-1: Schematic view of the AlphaZero algorithm as an interplay of two systems. The AlphaZero algorithm uses (Monte Carlo) tree search and deep learning to solve sequential decision making problems. The algorithm can be thought of as an interaction between a learning and a planning system. The plot depicts the two-way connection between the two systems. At each decision step, tree search is used to decide for the current action. The two-headed deep neural network provides guidance during tree search process (π_θ, V_θ). At the end of tree search, root statistics of the root's value estimate and root's child action probabilities are fed into the replay memory of the deep neural network ($\hat{V}_t, \hat{\pi}_t$) and the network is trained when an episode ends. Image is based on [5].

4-2 Generalization and locality in the AlphaZero algorithm

The two-system architecture of the AlphaZero algorithm can be also approached from a global versus local approach. In general there are two main approaches to store information during learning: (1) the global approach describes the whole state space by approximation (2) the local approach only keeps the currently relevant part of the state space [34]. Most value-based DRL approaches use the global approach, where a deep neural network is used to approximate the value function. In contrast, search-based solutions store local information about the currently relevant part of the state space. In the AlphaZero algorithm the deep neural network provide global estimates, while MCTS provides a local estimates. Both approaches have their advantages and disadvantages, which is discussed briefly next.

4-2-1 Global and local information in learning systems

Global approach The main advantage of using global function approximators for value estimation is its **generalization** power. In the context of machine learning, generalization refers to the model's ability to apply previously acquired knowledge in previously unseen situations. Deep neural networks have showcased great generalization power and have been shown to outperform other ML approaches in supervised learning settings. In value-based DRL deep neural networks have taken the role of tabular solutions for storing information about value functions, which made it possible to tackle high dimensional problems through exploiting generalization [11]. Generalization allows transferring knowledge between similar states, which becomes crucial in high dimensional or continuous state spaces. State-value pairs are no longer stored in tables, value estimates can be obtained through a forward pass of the input state through the network. The knowledge about the value estimates is encoded in the network's weights, which is much more memory efficient. The DLR the neural network is kept between the episodes and the weights of the deep neural network are updated regularly (e.g. every episode) during the learning process based on training data acquired from interacting with the environment. In RL context, neural networks can be thought as a global approach for storing information about value estimates.

The major disadvantage of using global estimators is that updating the network weights can affect output for the whole state space. This puts an increased importance on the training data. Selecting training data has been shown to be of crucial importance in DRL (experience replay [11]. Due to the non-stationary nature of RL, it has also been argued that global approximators are less well-suited for RL, than local representations. In value-based RL, the value estimate is computed iteratively from samples which are obtained from interacting with the environment. Value-based RL methods use bootstrapping, meaning that targets are based on current estimations. Additionally, the actions are based on the current value estimates. This means that both input distribution and target function are non-stationary, which makes training the neural network more challenging than in supervised learning [37].

Motivation for local approach **Local updates** are an advantage of local approaches. In case of global approximators, such as deep neural networks, all parameters are updated together. Local approximators have the advantage of quick updates, they allow quicker adjustment to the changes in the reward distribution and updates are specific to the recent states [38]. In

MCTS learning data is acquired during the simulation phase, where imaginary rollouts are performed. Based on learning data acquired from the rollouts, value estimates are updated during the backup phase using locally relevant training data.

Access to uncertainty estimates is also and advantage in contrast to deep neural networks. In case of global approximators, such as deep neural networks, uncertainty estimates are difficult to obtain [39]. Local knowledge representation has the advantage of having an uncertainty estimate, which can facilitate exploration.

The lack of generalization is the major drawback of local estimators. In the standard MCTS algorithm, the search trees are discarded and there is no knowledge sharing between the nodes (states) in the tree. The standard MCTS algorithm does not generalize across states. Standard MCTS treats the state space of the game as a tree, where multiple nodes can represent the same state, which do not share information. Although standard MCTS does not generalize across states, generalization has been studies in MCTS in the form of transpositions. Transpositions occur, when multiple paths leading to the same state. Information sharing amongst nodes can be taken into account by treating the search tree as a graph, which brings improvement compared to standard MCTS [40].

4-3 Results for examining trade-off between generalization and locality in the AlphaZero algorithm

4-3-1 Experiment setup

One possible approach to examine the question of the trade-off between generalization and locality in the AlphaZero algorithm is through varying the number of MCTS iteration steps n_{MCTS}, while keeping other hyperparameters of the algorithm fixed. The n_{MCTS} parameter corresponds to the number of simulated trajectories performed using the environment emulator before each action selection step in the real environment. Under a fixed time budget the number of MCTS iterations defines how much effort is spent on acquiring more accurate values through building large search trees at each decision step versus improving generalization by updating the network more frequently. In order to compare the performance of different settings and address the question of balancing local search and generalization (R2), the experiments are carried out under a fixed time budget, which was pre-fixed for each environment separately. The range of n_{MCTS} values was chosen according to action and state space based complexity of the environment. The choice of using relatively low numbers of iterations was motivated by the fact that decision time is often constrained in planning situations.

For each environment, first a hyperparameter search was carried out, then the algorithm was trained on the selected n_{MCTS} grid. Three independent experiments were carried out for each setting. Each of the experiments was repeated 3 times with different random seeds. Since all environments have returns outside [0,1] interval, modifications from the third chapter were applied for each environment. The examined RL environments all have a low state dimensionality, therefore fully connected neural networks were used, in contrast to the original algorithm's residual convolutional network, but the original two-head structure of the neural network is used. Details about hyperparameter choices can be found in the appendix (A-2-3).

4-3-2 Experiment results

Return The standard approach for assessing the RL algorithm's performance is to look at how the episodic total reward changes through training. The main findings of the experiment are presented on figures 4-2 and 4-3. Figure 4-2 shows that the AlphaZero algorithm achieves best performance with middle sized trees in all the examined environments. Figure 4-3 highlights the performance ranking among the different tree sizes by visualizing the averaged total episodic reward of the last 15 % of training time for the examined tree sizes. The graph shows clearly that peak performance is achieved for mid-sized trees, outperforming the extremes of the chosen grid.

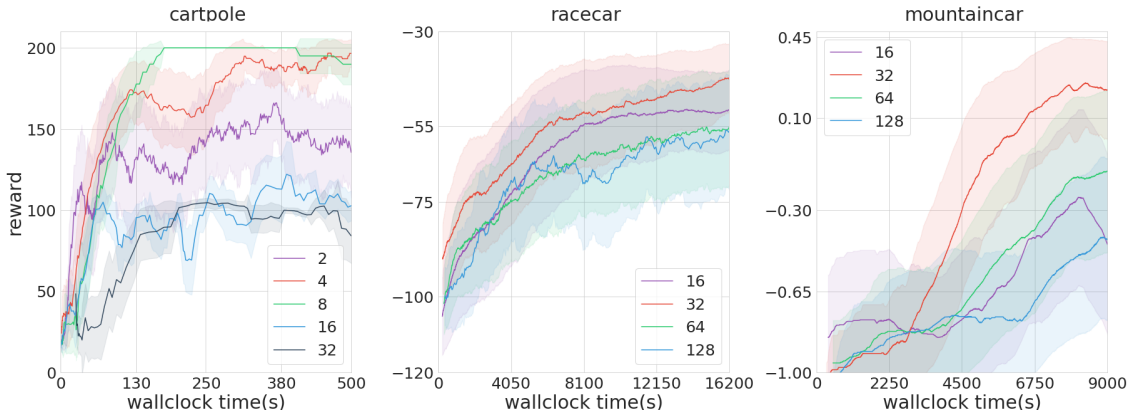


Figure 4-2: Comparison of total episodic reward based performance of varied number of MCTS iteration steps per decision shows that balanced focus on locality (high n_{MCTS}) and generalization (low n_{MCTS}) is best performing in all the examined RL environments (each result curve is averaged from three different seeds).

The results indicate that the strength of the AlphaZero 's algorithm lies in balancing local information and generalization. Choosing a high number of MCTS iterations puts emphasis on local information. Value estimates are likely to be more accurate with increasing the number of iterations, as values for the encountered nodes in the search tree are estimated as the average return of the simulated trajectories passing through the given node. On the other hand, executing a high number of n_{MCTS} is time costly and therefore less time remains for training the neural network and improving it's generalization capacity. Under strict time constraint, this will hurt the overall performance of the algorithm as it can be observed on figures 4-2, 4-3. In each environment the highest number of n_{MCTS} setting performs the worst. Choosing a lower n_{MCTS} allows more frequent training, which could improve generalization, but it carries the risk of obtaining incaccurate value estimates. Value estimation is likely to have high variance and therefore the accuracy cannot be guaranteed. As a result although the network is trained more frequently, the possibly highly inaccurate samples used in the training set could prevent the network from convergence or make the network converge to a suboptimal local minima. Overall, the reward plots show a clear performance ranking among the evaluated tree sizes for the examined environments with mid-range tree sizes overperforming the extremes. The following sections contain further evaluations of ther results.

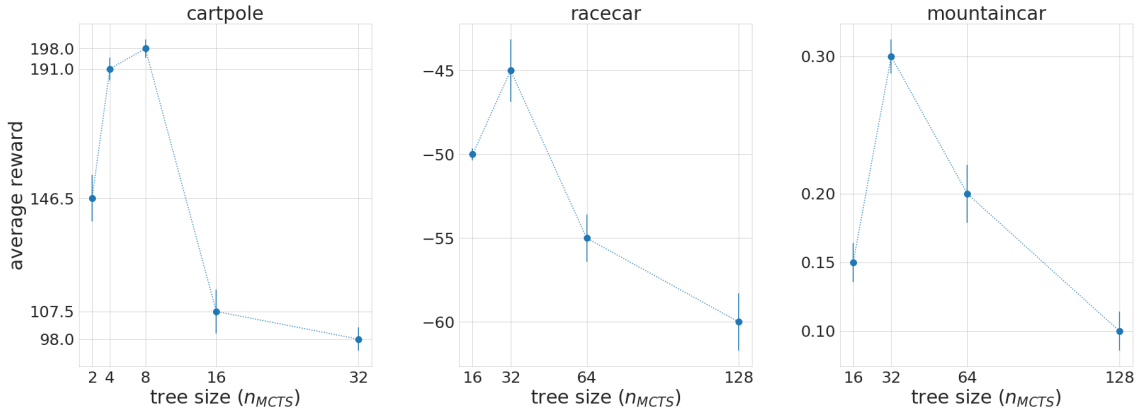


Figure 4-3: Performance peaks for mid-range n_{MCTS} iterations highlights importance of balancing tradeoff between locality (high n_{MCTS}) and generalization (low n_{MCTS}) for different RL environments. Curve points are obtained by averaging episodic reward of last 15% of wallclock training time from result curves of figure 4-2.

Entropy and loss Further insights can be gained about the trainig process and the effects of using different n_{MCTS} iterations by looking at how the entropy and loss change during training time. As in standard MCTS, at the end of the search, the AlphaZero algorithm select the action for the current state s_t based on the normalized action counts of search tree's root node $\hat{\pi}_t$ (equation 2-13). The child node with the highest number of simulations gets selected. Initially, the normalized count vector will be close to uniform, as without any prior information, all child nodes are equally likely to be selected in the search tree. As more simlulations are carried out, the node statistics are updated, which will guide the tree building (equation 2-15), and promising actions will get selected more. Information entropy (A-4) could be used to visualize the algorithm's certainty about it's decision $H(\hat{\pi}_t) = -\sum_i \frac{n(s_t,a_i)}{n(s_t)} log(\frac{n(s_t,a_i)}{n(s_t)})$. .

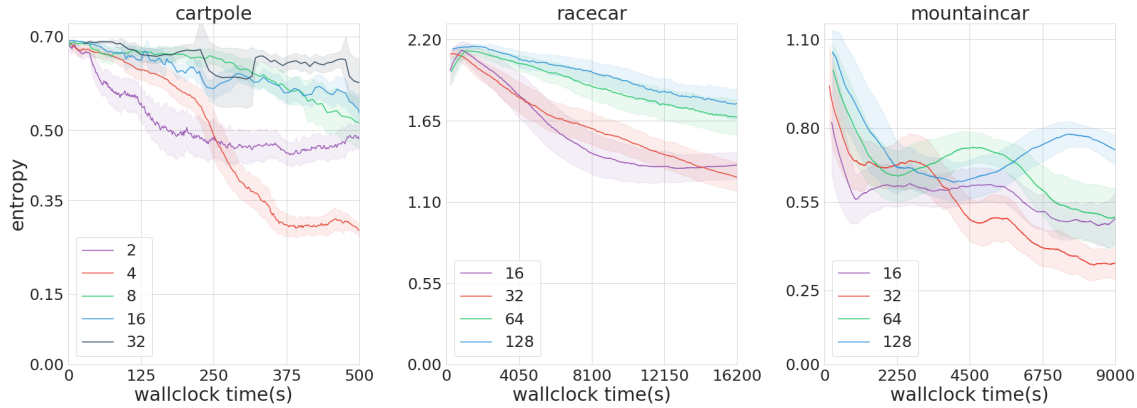


Figure 4-4: Comparison of mean decision entropy of episodes over time of varied number of MCTS iterations shows decrease in entropy (decison uncertainty) is largest for n_{MCTS} iterations which perform best in case of mountaincar and racecar problems (figure 4-2). The entropy is upper bounded by the cardinality of the discrete action set ($log(|A|)$)

.

Figure 4-4 shows how the action decision entropy changes for different number of n_{MCTS}

iterations. It is important to point out that in contrast to later entropy plots in the report, these plots were obtained by taking the entropy of the normalized counts at the end of the MCTS search $\hat{\pi}_t$, and not the entropy of the policy network's output $\pi_\theta(s_t)$. First, looking at the racecar and mountaincar problems it can be seen that the entropy decreases the most in case of the best performing n_{MCTS} iterations setting (n_{MCTS}=32 in both environments) as seen on figure 4-2. It can be also observed from the graphs that in case of low MCTS iterations ($n_{MCTS} = 16$), the entropy drops quickly (due to more episodes completed), but the decrease halts at some point. This could be caused by false convergence of the network due to the inaccurate estimates.

A similar behaviour can be observed in the network loss curves (figure 4-5). In case of the racecar and mountaincar problems it can be clearly seen that the loss stops decreasing at some point in case of $n_{MCTS} = 16$, and remains higher than in the best performing $n_{MCTS} = 32$ case. It should be mentioned that although the network loss could give an indication about the convergence of the network, but as the neural network's training data is updated constantly during the learning process as the agent explores, it is less straightforward to interpret loss plots in DRL than in supervised learning problems with a fixed training set. In case of the cartpole environment, the loss over time does not reflect the algorithm's performance.

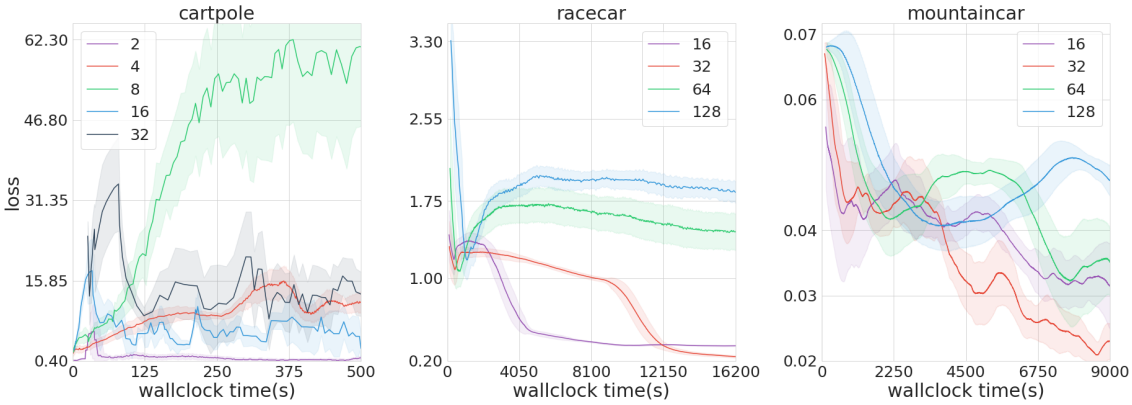


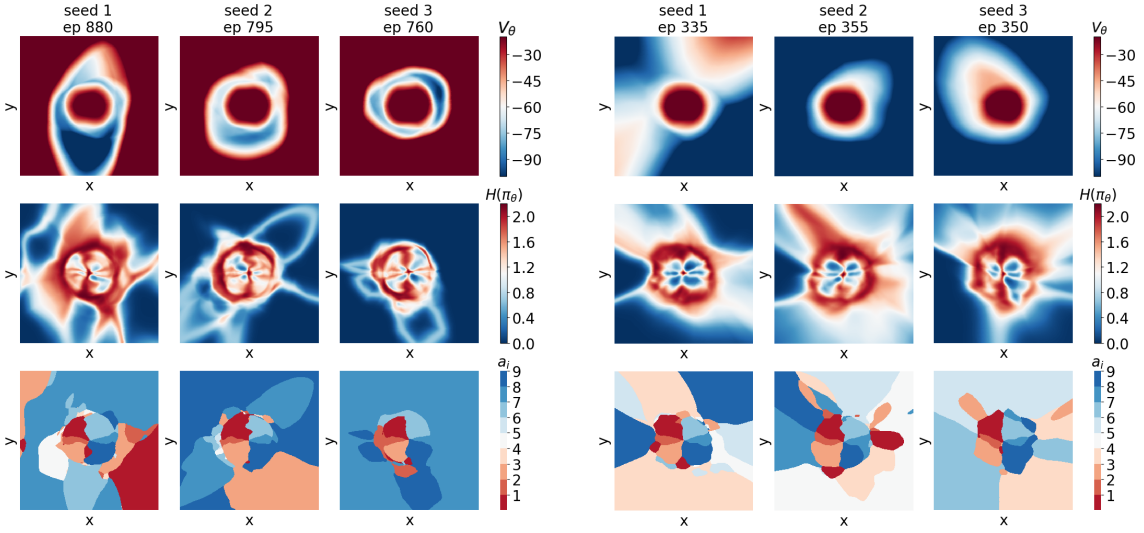
Figure 4-5: Comparison of the AlphaZero algorithm's loss (equation 2-14) over time of varied number of MCTS iterations shows decrease in network's training loss is largest for n_{MCTS} iterations which perform best in case of mountaincar and racecar problems 4-2)

4-3-3 Further evaluation of learning process

In the following, evaluations are provided for the results presented in the previous section, which provide additional insights about the learning process in case of varied number of MCTS iterations.

Network output As the AlphaZero algorithm's generalization capacity comes from the neural network, the trade-off between generalization and locality can also be evaluated through examining the neural network's behaviour for different n_{MCTS}. The neural network in the AlphaZero algorithm has two outputs for a given input state: the scalar value prediction $V_\theta(s)$ and the policy prediction vector $\pi_\theta(s)$, which are utilized during MCTS to guide the search

process. The value network output is used to estimate values of newly added state nodes in the search tree, while the policy network output is used in the action selection formula (equation 2-15) during tree traversal. Since the neural network outputs play an important role in building the search tree and consequently in action selection, comparing the network behaviour for different n_{MCTS} can give further insights into the locality-generalization trade-off.



(a) $n_{MCTS} = 16$ **(b)** $n_{MCTS} = 32$

(c) $n_{MCTS} = 64$ **(d)** $n_{MCTS} = 128$

Figure 4-6: Visualization of AlphaZero neural network predictions evaluated at a fixed state space grid for different number of MCTS iterations (n_{MCTS}) gives insights into generalization-locality trade-off in the AlphaZero algorithm. Network predictions were taken at the end of pre-fixed training time (4.5h) in the racecar environment for 3 different seed in each case. Row-wise results are value network (V_θ) prediction, entropy of policy network prediction ($H(\pi_\theta)$) and predicted action (a_i) from policy network. Axes of subplots correspond to state variables (x,y distance from goal in center).

The relatively low complexity of the examined problems allows gaining insights about the algorithm's behaviour. Figures 4-6 - 4-8 visualize the value and policy network predictions

evaluated on a fixed set of state space points, which is acquired by creating a grid over the 2D state space. As in previous experiments, the comparison is made between different number of node expansion steps in the MCTS search in order to study the locality-generalization trade-off. The mountain-car and racecar problems (A-1) are well-suited for this visualization of the learning process, since they have a 2D state-spaces.

Figure 4-6 compares the network outputs at the end of the pre-fixed training time for different number of MCTS simulations in case of the racecar problem. Network outputs are shown for 3 different seeds in each n_{MCTS} case, whose averaged performance is visualized on the reward plot of Figure 4-2. In each n_{MCTS} case the first row shows the value predictions, V_θ the second row the entropy of the policy network prediction $H(\pi_\theta)$ and the third row the predicted action a_i for the different seeds. The entropy map of the policy predictions can give an indication about the convergence of the policy network, since as the learning progresses, the policy recommendations π_θ will move from the initial uniform distribution, which causes it's entropy to decrease.

The most relevant observation with regards to the generalization-locality trade-off is the difference between the value prediction maps for different n_{MCTS} cases. The racecar problem's state is defined as the racecar's position (x,y) and the reward function is defined as negative current distance from the goal (x=0,y=0). Value plots for $n_{MCTS} = 32, 64, 128$ show the expected circular pattern, with values being higher closer to the goal in the centre, but for $n_{MCTS} = 16$ this pattern cannot be observed. Reason for this discrepancy could be the inaccuracy of the value estimates from insufficient number of MCTS simulations per decision. As value estimates have a great influence on building the search tree through the UCT-$\alpha 0$ formula (equation 2-15), this could be a cause for under-performance compared $n_{MCTS} = 32$.

On the other hand, as shown by the reward plot of Figure 4-2 the algorithm performs surprisingly well with the falsely converged value network, compared to cases, where the value network output shows the expected circular pattern. An explanation for this could be the relatively slower convergence of the policy network compared to the value network, which is indicated by the entropy plots ($H(\pi_\theta)$). The lower n_{MCTS} is, the lower the overall entropy of policy prediction is for the state space grid at the end of the pre-fixed training time. Low values of $H(\pi_\theta)$ indicate strong preference for one action, the policy recommendations used in UCT-$\alpha 0$ are nearly one-hot vectors.

In case of the best performing case $n_{MCTS} = 32$ in racecar, both the circular pattern in the value estimate map and relatively low policy prediction entropy can be observed.
Figure 4-6 also displays the action prediction for the selected state space grid points out of the 9 possible discrete actions of the racecar problem (A-1). It is not as straight-forward to assess the validity of these predictions as the value network prediction. Looking at both $H(\pi_\theta)$ and a_i, it can be observed that the action entropy remains relatively high at the decision boundaries, compared to other part of the state space.

Network convergence Figures 4-7 and 4-8 provide additional insights about the learning process for the racecar and mountaincar problems respectively by visualizing the progression of V_θ and $H(\pi_\theta)$ predictions over trainig episodes. The motivation of the following plots is that visualizing patterns from the prediction map can give an indication about the network's convergence.

For both figures 4-7 and 4-8 results for different n_{MCTS} cases are presented rowwise. In each case, the first plot shows the progression of the value network prediction V_θ, the second plot shows the progression of the policy network prediction entropy $H(\pi_\theta)$ and the third plot shows the corresponding learning curve for a single seed. Episodes, for which the predictions were evaluated were taken from an equally spaced grid of the training episodes. As previously, the training was exectuted under afixed training time (t=4.5h for racecar and t=2.5h for mountaincar). Similarly to 4-6 the predictions are evaluated on the same state space grid in each case. Axes of the V_θ, $H(\pi_\theta)$ subplots represent the state space grid of the environment. The episode, from which the network weights were taken is indicated in each subplot's title.

In the racecar problem, the agent has to reach the goal in a center, information provided in the state vector is the distance in camera frame from the goal in the center given in x and y coordinates. In the mountaincar problem, the agent has to escape the initial local minima of the valley and reach the top of the hill through learning to gain momemtum, as the car is underpowered. The 2D state vector consists of the car's position and velocity and the available actions are not moving, moving left or moving right. The episode terminates after a fixed number of episode steps or when it reaches the top of the hill, where it gets positive reward.

At first glance, both figures (4-7, 4-8) show a slower convergence of the networks for higher n_{MCTS}, which is expected as fewer episodes and therefore fewer network trainings are completed in the given maximum training time. In both cases the the value networks appear to converge faster than the policy network. This is can be seen clearly in case of the racecar problem, as the circular pattern in the value network prediction appears early in training in all n_{MCTS} cases. The policy entropy remains relatively high and no patterns appear, which indicates no clear action boundaries appearing as in Figure 4-6. In case of the mountaincar problem, the difference in convergence rate is more apparent in the cases of $n_{MCTS} = 64, 128$.

A second observation is that for both the mountaincar and racecar problems, the high variance and inaccurate value estimates from low MCTS iterations ($n_{MCTS} = 16$) causes instable and false value predictions. In case of the mountaincar problem, the value predictions are quite instable, while in case of the racecar problem it converges to false values. The instability of the value predictions is also reflected in the performance, the rewards plots show a noisy behaviour for both environments in case of $n_{MCTS} = 16$.

Furthermore, the plots also show that a higher number of n_{MCTS} iterations does not help significantly the policy network to converge faster under a limited time budget. The policy network's training samples are the normalized child action counts from the MCTS search. It could be expected that executing more MCTS iterations would result in a stronger preference for one of the actions and more reliable policy samples, and therefore lead to a faster convergence of the policy network (This effect can be observed by comparing the entropy of the policy predictions for $\tilde{1}$00th episode in the racecar problem). But as there is a time limit, these possibly more accurate samples do not affect the final performance, since there are not enough updates executed when too much time is spent on acquiring the training samples.

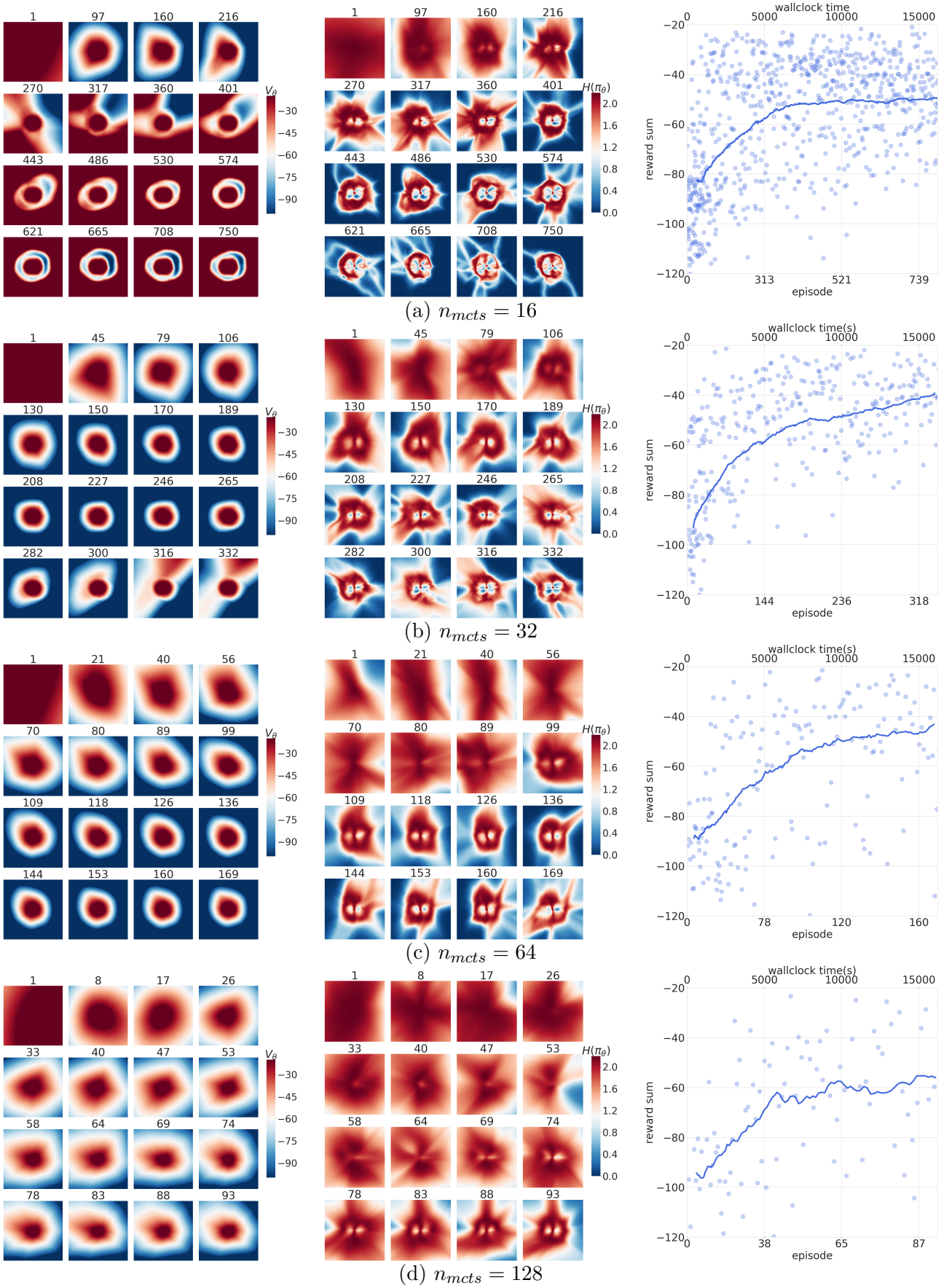
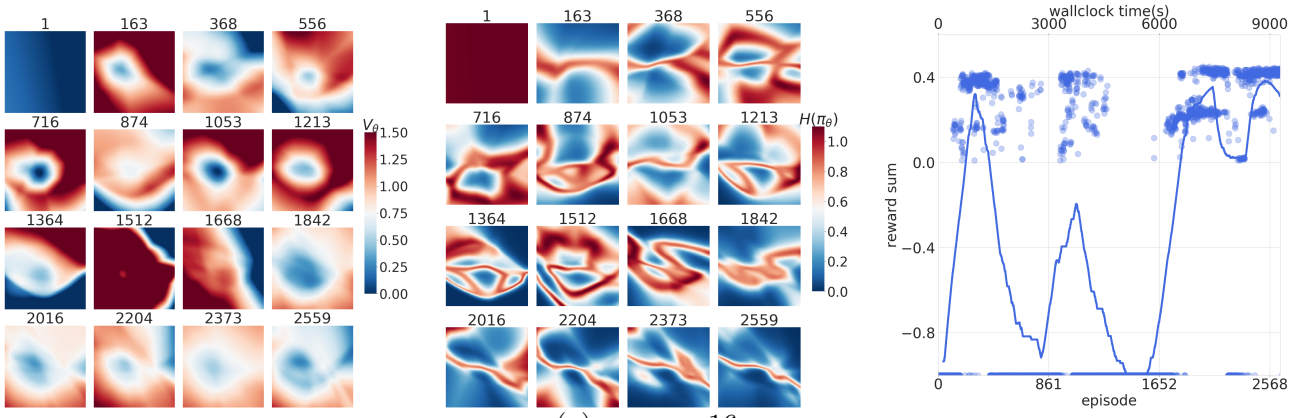
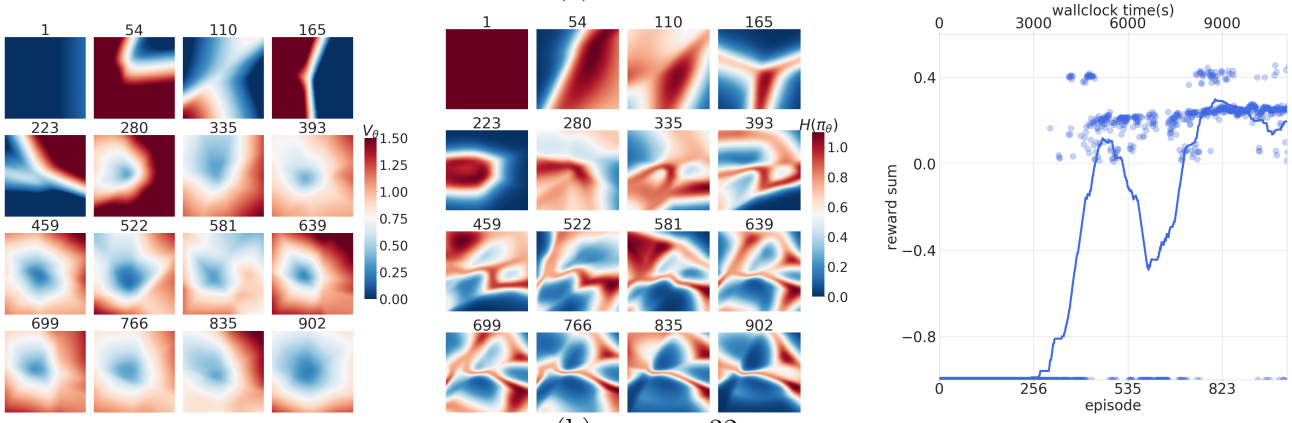


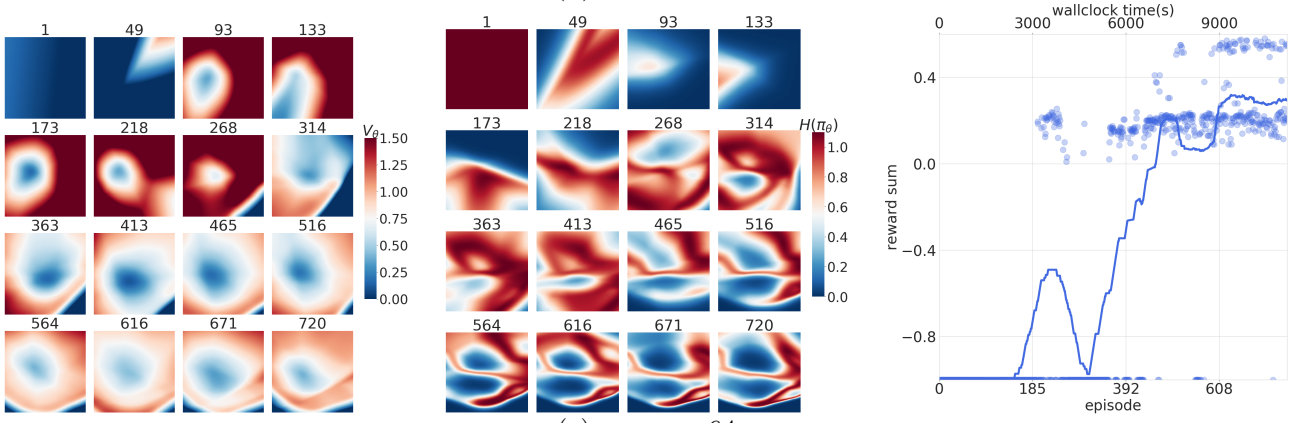
Figure 4-7: Visualization of changes in value network prediction, entropy of policy network prediction and total episodic reward over episodes during the training process for different number of n_{MCTS} in case of the racecar problem. Network prediction plots display predictions over state space (x axis is car position, y axis is car velocity) for different episodes through training. Network predictions are evaluated for the same fixed grid over state space, individual subplots title represent episode, from which network weights were taken. In the racecar problem, the agent is rewarded based on it's distance from the goal in the center, hence the circutal patterns in value maps. High entropy of policy predictions suggests policy network's slower convergence. All experiments were ran for 4.5 hours.



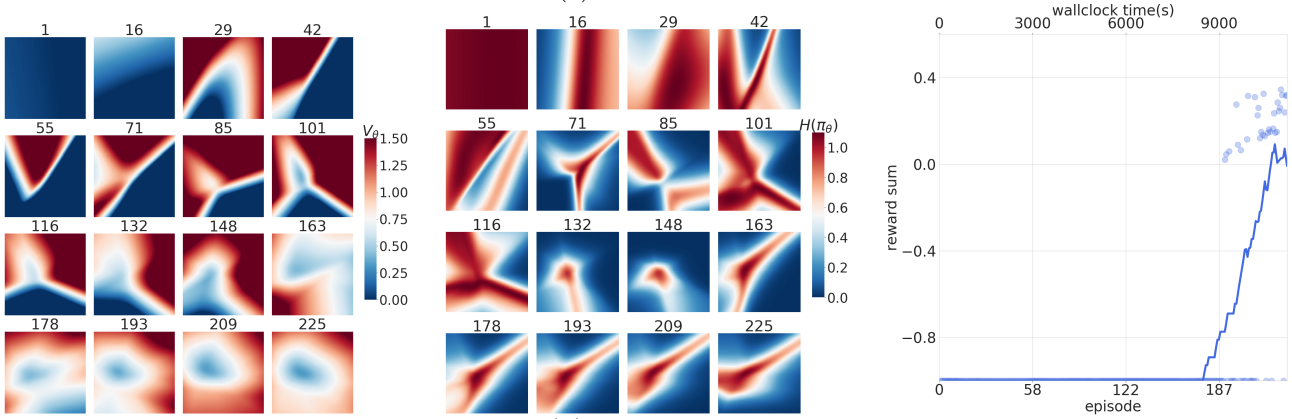
(a) $n_{mcts} = 16$



(b) $n_{mcts} = 32$



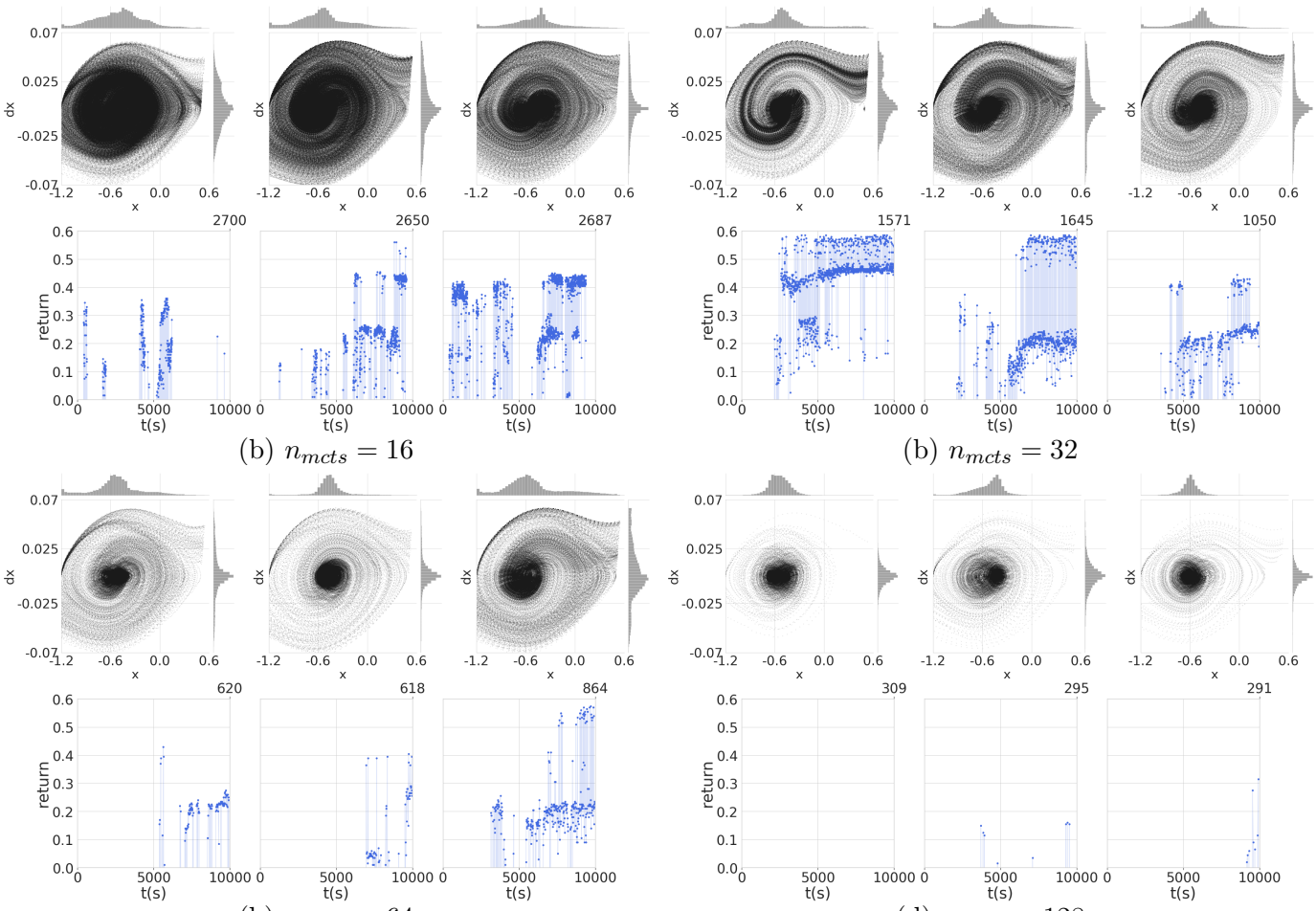
(c) $n_{mcts} = 64$



(d) $n_{mcts} = 128$

Figure 4-8: Visualization of changes in value network prediction, entropy of policy network prediction and total episodic reward over episodes during the training process for different number of n_{MCTS} in case of the racecar problem. Network prediction plots display predictions over state space (x axis is car position, y axis is car velocity) for different episodes through training. Network predictions are evaluated for the same fixed grid over state space, individual subplots title represent episode, from which network weights were taken. All experiments were ran for 2.5 hours

State space exploration It is also interesting to look at how the state space coverage changes while varying the number of MCTS iterations. Figure 4-9 allows insight into the learning process through visualizing the visited state space points encountered during training in the mountaincar environment. In the mountaincar environment (A-1) the underpowered car has to escape initial local minima (bottom of valley) through learning a policy of gaining momentum. As expected, for higher n_{MCTS} there is less state space coverage, as less episodes are completed. More interesting are the comparison between the $n_{MCTS} = 16, 32$ plots. In both cases the state space coverage is extensive and the agent manages to escape the local minima, but the return plots show that in case of $n_{MCTS} = 32$ the agent manages to learn a stable policy, which is also reflected on the state space plot.



(b) $n_{mcts} = 16$ (b) $n_{mcts} = 32$

(b) $n_{mcts} = 64$ (d) $n_{mcts} = 128$

Figure 4-9: Visualization of state space exploration and obtained rewards for different n_{MCTS} iterations in the AlphaZero algorithm in the mountain-car environment. Experiments were ran for a fixed time and three different seeds in each case. Plots show that using a low number of traces ($n_{MCTS} = 16$) results in extensive state space exploration, but failure to learn a stable policy, while executing more traces per decisions allows executing only few traces in the given training time ($n_{MCTS} = 128$). Axis x represents position, axis y represents velocity on state space plots.

4-4 Proposed additional connections in the two systems

As seen in the previous section, under a strict time constraint the generalization-locality trade-off plays a decisive role in the AlphaZero algorithm's performance. The previous results have shown that using too low MCTS iteration number can cause instable and inaccurate predictions and consequently have a detrimental effect on the performance. This could be observed clearly in the examined problems, where the inaccurate training samples from low iteration MCTS search ($n_{MCTS} = 16$) caused instable performance (figures 4-8, 4-7). Executing a higher number of MCTS iterations per decision results in more accurate and lower variance training samples for the neural network, and consequently the performance improves. On the other hand, under resource limitations, the iterations number should not be set too high, because the network cannot learn if not enough trainings are executed. In the examined problems, the effects of executing too few trainings in the given training time were more apparent in the policy network predictions, as the entropy of the action recommendations from the policy network did not decrease significantly through training if n_{MCTS} was set relatively high. Overall, the experiments have shown that the AlphaZero algorithm performs best, when the algorithm's two components play balanced roles. In case this optimal n_{MCTS} setting, the number of MCTS iterations is high enough to acquire accurate training samples for the neural network and also sufficient time remains to collect training samples and exeucute training epochs for the neural network. The next sections look over some possible improvements of the previous results.

4-4-1 Variance based adaptation

As results have shown in the previous section, setting the n_{MCTS} parameter right has a crucial role in the AlphaZero algorithm's performance under tight time contraints. A correctly set n_{MCTS} parameter is high enough to acquire accurate estimates, but not too high, so sufficient time remains for training the neural network and improving generalization. The previous results, as the standard MCTS algorithm used a fixed number of MCTS iterations throughout tranining. It would also be more resource efficient to adjust n_{MCTS} according to the learning process. As the learning progresses, the algorithm will become increasingly confident in the chosen decision for frequently visited states. Therefore it could be beneficial to carry out less MCTS iterations in these states, and concentrate the search efforts on states, where the algorithm is still uncertain about the estimates.

It is not straight-forward to assess, which value estimates could be assumed more accurate, relative to other states. One possible approach to consider is the variance of the root returns during the MCTS search. Return variances have been previously used in the tree policy in the UCBV [41] algorithm. In that approach variance estimates $\tilde{R}_{k,n(s,a)}$ for each child action are used in the tree policy in order to detect suboptimal actions with low values estimates and low variance after a few $n(s,a)$ of the given action. This has been shown to improve performance through better exploration. While each action's return variance gives an indication about certainty in the child action's action-value estimate, the variance of returns of the root states also includes the indeciveness between the child actions. Comparing the variance of the current root's backed up returns $\tilde{R}_t$ to a baseline value could imply if the MCTS iteration can be stopped, because the algorithm's fairly certain about the decision already or should be continued, because it is still quite uncertain about the action selection.

One of the simplest approach to get a baseline to compare the current return variance is to keep a rolling estimate of the mean root return variances, which is updated with the current root's return variance at each decision step. At each decision, a small number MCTS iterations are carried out (n_{base}) and the backed up returns are kept for the root state (figure 2-3). After these iterations, the number of additional MCTS iterations n_{adt} is calculated based on the comparison of current variance to the baseline value. Additional iterations are only carried out, if n_{adt} overpasses a minimum number of iterations n_{min} and has an upper limit of n_{max}.

$$\hat{R}_t = \frac{1}{n_{base}} \sum_{i}^{n_{base}} R_i, \qquad \tilde{R}_t = \frac{1}{n_{base}} \sum_{i}^{n_{base}} (R_i - \hat{R}_{n_t})^2,$$
$$\tilde{R}_{roll} = \frac{t-1}{t}\tilde{R}_{roll} + \frac{1}{t}\tilde{R}_t, \qquad n_{adt} = \max(n_{min}, \min(n_{max}, n_{min}\frac{\tilde{R}_t}{\tilde{R}_{roll}})) \tag{4-1}$$

, where R_i is the backed up root return from iteration i in the MCTS search (figure 2-3), $\hat{R}_t$ is the mean return estimate from n_{base} iterations and $\tilde{R}_t$ is the variance of the root returns after n_{base} iterations at state s_t.

This approach was tested for the racecar problem with the same hyperparameter settings and training time. The iteration settings were $n_{base} = 16, n_{min} = 8, n_{max} = 64$. The modified algorithm produced a learning curve close to the best performing case of the fixed tree sizes of the previous experiments ($n_{MCTS} = 32$). This shows that controlling search efforths through adaptively changing the number of MCTS traces based on the return variance can help the learning process.

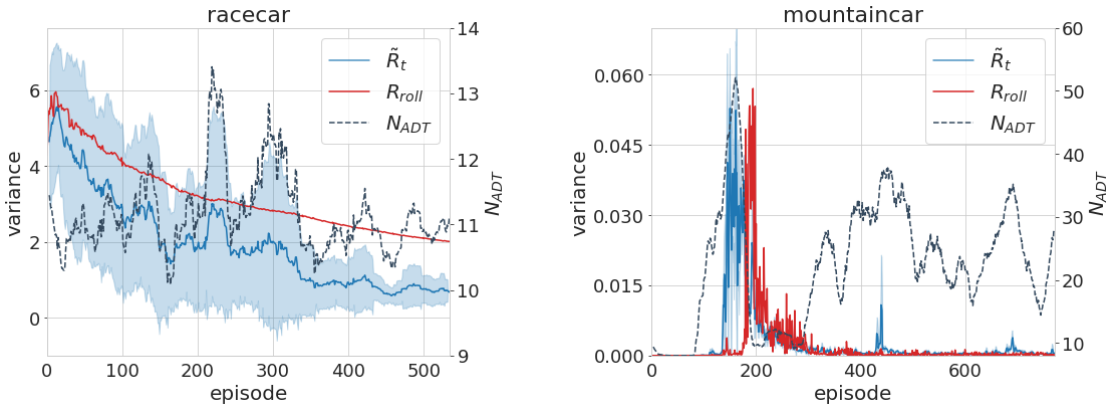


Figure 4-10: Changes in MCTS search root return variance $\tilde{R}_t$, tracked mean return variance R_{roll} and additional MCTS iteration count n_{adt} over the training process for the racecar and mountaincar problems for a single seed. Variances and iteration count are averaged over training episodes, graphs for n_{adt} and $\tilde{R}_t$ are smoothed over 25 episodes. Plots show that taking into account all previous samples (eq. 4-1) results in mean estimate lag behind real root return variance. However the additional iteration count n_{ADT} does follow trends in variance change.

For further performance improvements it is interesting to compare the convergence of the value network and the rate of policy entropy decrease in the previous results on figures 4-8 and 4-7. The previous results have shown that the decrease of entropy of the policy network output is relatively slow. This might be improved by tuning the temperature parameter in the action counts normalization formula (equation 2-13), which prepares the training examples

for the policy network at the end of MCTS search. In all experiments from the previous section, the temperature parameter of the action count normalization at the end of the MCTS search (equation 2-13) was kept at $\tau = 1$. The temperature parameter allows controlling the exploration-exploitation trade-off. A higher temperature value moves action selection towards exploitation, as the policy recommendations become closer to one-hot-encoding. Time-based decaying of $1/\tau$ is often used to move the emphasis from exploration towards exploitation during training. This approach does not take into account that decisions in more often visited parts of the state space space might be more certain. An alternative to this could be to set the temperature based on the return variance ratio (eq. 4-2). This means that if the return variance is relatively low, then the MCTS would return a policy network training sample $\hat{\pi}$, which is close to one-hot encoding.

$$r = \frac{\tilde{R}_t}{\tilde{R}_{roll}} \qquad \tau(r) = \begin{cases} exp(1-r) & \text{if } r<1.0 \\ 1 & \text{if } r>1.0 \end{cases} \tag{4-2}$$

The following plots present results for return variance modifications of adaptively changing the number of MCTS traces and the temperature for the policy training samples. Figure 4-10 illustrates that the modified algorithm follows the changes in variance relatively good and additional MCTS iterations are carried out when there is a peak in the root return variance. It can be also seen on the mountaincar plot that the MCTS iterations number is kept at the n_{base}, when the variance is low in the beginning of learning. The plots also show that the rolling estimate of the root return variance lags behind the root return variance quite significantly, which is caused by taking into account all previous data 4-1. Using windowed calculation might be a better approach. Figure 4-11 shows that adaptively changing n_{MCTS} and the temperature does bring improvements compared to using a fixed number of n_{MCTS} iterations and a temperature of $\tau = 1$. in both the racecar and mountaincar environments. As in previous results, the plots show the averaged results from three different seed for both environments. Similarly to previous experiment results, figure 4-12 shows the network output changes throughout the training for a single seed for both environments. It is clearly visible that the entropy of the policy network outputs decreases faster than in previous results, due to changing the temperature parameter.

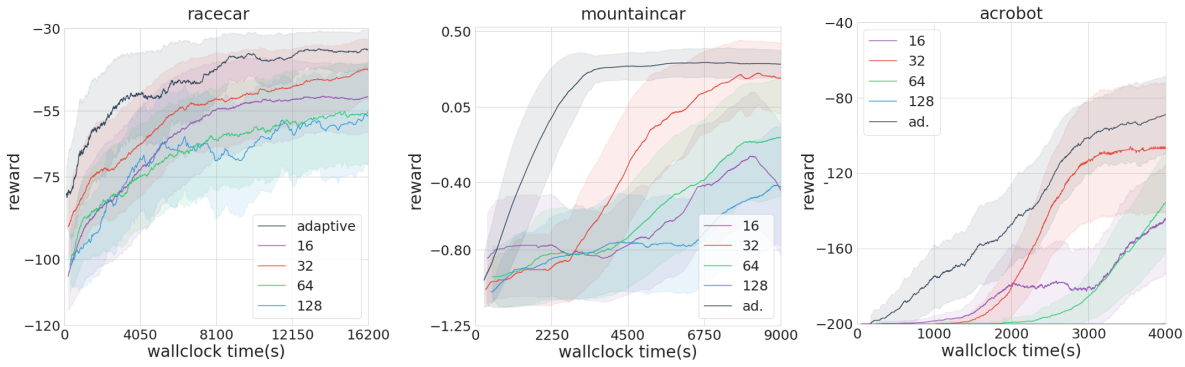


Figure 4-11: Episodic reward based performance comparison of fixed MCTS iteration counts and variance based MCTS adaptive iteration counts in racecar and mountaincar problems. Each curve is averaged from 3 different seeds. Plots show that adaptively changing iteration count does help to improve performance in both cases, as the adaptive version outperforms fixed iteration counts in both environments under a fixed budget.

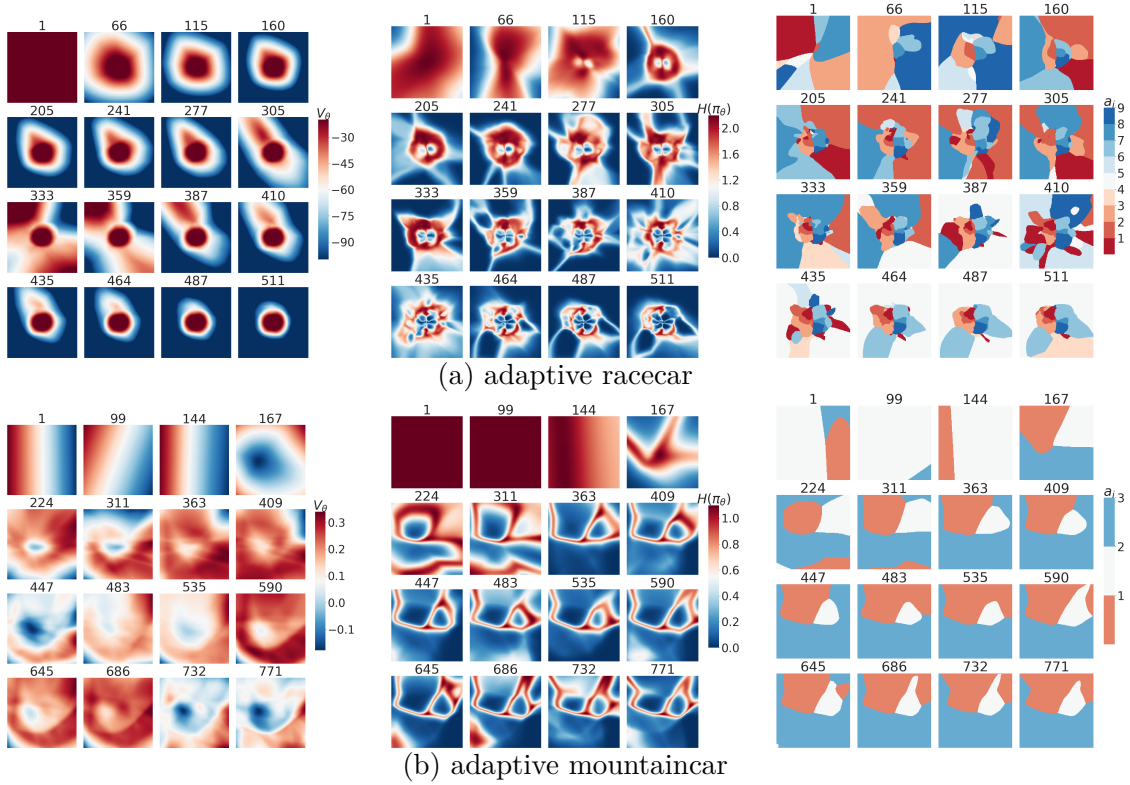


Figure 4-12: Visualization of changes in value network prediction, entropy of policy network prediction and maximum probability action during the training process in case of the return variance based adaptive n_{MCTS} modification. Results are shown for the racecar and mountaincar problems A-1. The network prediction plots display predictions over state space points for different episodes through training. Network predictions are evaluated for the same fixed grid over state space, individual subplots title represent episode, from which network weights were taken. The plots visualize faster convergence of the network, compared to the fixed n_{MCTS} case.

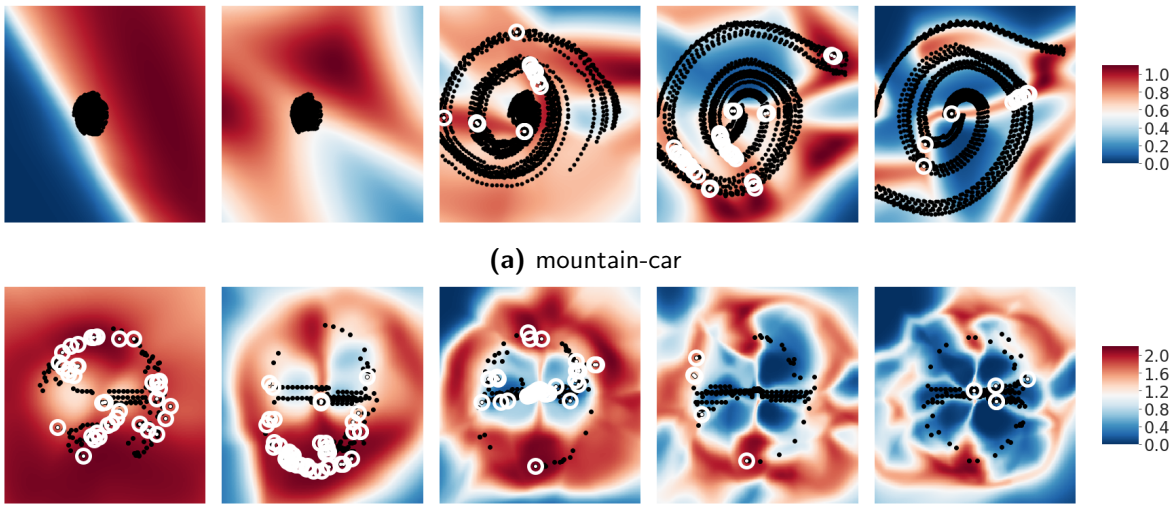
Focusing search efforts on more uncertain parts of the state space has great importance under time constraints. The previous results have shown that heuristic approaches of exploiting return variance based uncertainty information can bring performance improvements. However, it introduces new hyperparameters to tune and has several pitfalls. The range of additional MCTS iterations was selected based on previous results of fixed MCTS iterations. This helped to avoid incorrect value predictions from too few traces and delayed learning from too many traces. Selecting the range of MCTS iterations without relying on additional experiments would be more efficient. Furthermore, the lag in the rolling variance estimation can cause the algorithm to be insensitive to small changes in the return variance. Actions leading to negative rewards are likely to be selected in the beginning of learning. This results in a high return variance in $\tilde{R}_{roll}$, which can prevent increasing n_{MCTS}, when $\tilde{R}_t$ grows due to uncertainties in action selection or encountering positive rewards in the search tree.

4-4-2 Other possible extensions

Collecting its own training data is an important aspect of RL learning algorithms, which distinguishes them from supervised learning. The AlphaZero algorithm is an iterative learning

algorithm, where learning is realized from the two-way interaction of tree search and deep learning. Tree search provides training data for the neural network, whose outputs are used to guide the tree search process. Results of the previous section have demonstrated that it is worth looking into methods to control search efforts. The previous, return variance based results have shown that learning can be greatly accelerated by focusing search efforts on high variance (more uncertain) parts of the state space. More accurate estimates can be obtained by increasing the number of MCTS iterations in unexplored areas. This improves the training data for the neural network, which can improve the learning performance.

An additional option could be to use the difference between the policy network's output and the MCTS search's output in a given state. As the learning progresses and the neural network converges, the selected action from MCTS (action with the highest visitation count) should match the highest probability action from the policy network in a given state. Figure 4-13 shows the policy network's entropy change and visited states (trajectories) throughout episodes during learning for the mountain-car and racecar problems. The policy network output's entropy gradually decreases through learning. After training the network several times, high policy entropy regions indicate action boundaries. Visited states, where the maximum probability action from the policy network did not match the visit count based action from MCTS ($\underset{a}{\text{argmax}}\pi_\theta \neq \underset{a}{\text{argmax}}\hat{\pi}_{MCTS}$) are highlighted with white. In both the environments, at the end of training, the two predictions differ only at the action boundaries. The difference between the policy network prediction and the MCTS action prediction could give information about the task structure and potentially be used to improve the learning performance.



(a) mountain-car

(b) racecar

Figure 4-13: Visualization of the policy network entropy and trajectories from 5 equally spaced episodes during learning in case of the mountain-car and racecar environments. The trajectory scatter plots (black marks) of visited states are overlayed on the entropy heat maps on each plot. The entropy heat maps are evaluated on a fixed state space grid for each environment. The white marks highlight visited states, where the MCTS based selected action did not match the policy network recommendation for the given state ($\underset{a}{\text{argmax}}\pi_\theta \neq \underset{a}{\text{argmax}}\hat{\pi}$).

Chapter 5

Discussion

The previous chapters have demonstrated that the AlphaZero algorithm's combined local search and generalization architecture can perform well on simple single-player deterministic RL environments after minor adjustments. Results of the second research question have shown that balancing the trade-off between the two components has a crucial role in the algorithm's performance under strict time constraints. It has also been showed that focusing search efforts on more uncertain states, through varying the number of traces in MCTS search can bring significant performance improvements.

It is important to point out that results from the previous chapters have also shown that the AlphaZero algorithm's performance on the examined small scale problems is not prominent. Training takes relatively long and performance was often in stable. The results of the second research question indicate that executing the right number of MCTS iterations is crucial for the algorithm's performance and stability. First, it effects the accuracy of the training samples for the neural network, whose output is then used in the subsequent search processes. Secondly, it is resource demanding, therefore the number of MCTS iterations should be limited. The improved performance results from the adaptive MCTS modification suggest that in future research it would be worth looking at methods to control the number of MCTS iterations in order to have a more stable and better performance.

Furthermore, this thesis was mainly concerned with examining the AlphaZero algorithm from the locality versus generalization aspect, and did not consider some factors, which were important to the algorithm's success in Go. An example for this is the convolutional-residual neural network structure. This neural network architecture was important to prevent overfitting and improve generalization. The low state dimensionality of the examined environments allowed the use of a simple, feedforward deep neural network. In future research it could be interesting to examine the algorithm, and the relevance of the combination of local and global knowledge in environments with high state-dimensionality.

The rest of this chapter will take a broader look on the algorithm's structure and how it relates to the experiment results of this thesis. It will also consider further possible research directions and possible applications areas of combining the generalization capacity of deep neural networks and the local focus of planning algorithms.

5-1 Relevance of the two system architecture

Incorporating a deep neural network in the UCT algorithm was a novel approach to achieve information sharing among nodes in the MCTS algorithm, which brought a great algorithmic performance increase for the game of Go. It is interesting to look at motivations for the combined deep learning and MCTS architecture, beyond the game of Go.

5-1-1 Generalization and temporality

The success of deep reinforcement learning algorithms can be greatly attributed to their generalization capabilities, compared to traditional RL algorithms. Similar states are grouped together by compressing the high dimensional state vector into a lower dimensional representation. Apart from generalization, it has been argued that focusing on local information is also crucial for problems with very high dimensional state spaces, such as the game of Go. Temporality is the concept of focusing the agent's representation on the current region of the state space [42]. The motivation to focus on local regions of the state space comes from the ignorance of the temporal structure of RL tasks in temporal-difference algorithms. These algorithms optimize a single value function based on randomized batches of all previous experience for the whole state space. An example of this is the use of experience replay in the DQN algorithm. This can be disadvantageous for two reasons. First, in case of larges state spaces optimizing the value function over the whole state space can be very resource inefficient. This could be observed in the long training times of recent successful deep reinforcement learning algorithms, which optimize a single value function for the whole state space [43],[11]. These algorithms have very high sample complexities, which means that the a large number of training samples is needed to learn the policy. Secondly, for non-stationary environments, model resource limitations and the slow incremental updates of the current deep neural network solutions can prevent necessary adjustments to changes in the environment. The AlphaZero has proven that the best performance can be achieved if the algorithm uses the advantages of both generalization from a value function approximation over the whole state space and temporality through focusing on local regions in the decision making. The experiment results from the second research question of this thesis have also showed that the strength of the algorithm lies in balancing focus on local information and generalization.

5-1-2 Computational efficiency in model-free and model-based RL

A differentiation between RL algorithms can be made based on whether the algorithm learns an environment model and uses it for finding the optimal policy (model-based) or derives the optimal policy purely from samples from environment interactions (model-free). Model-free and model-based RL have been used to model behaviours during decision making in humans [44]. Model-free RL has been linked to habitual or reactive control while model-based RL to reflexive or deliberate control. The main property of habitual control is the use of a 'caching', which means that a scalar summary of the state-action pair is stored. This is computationally light, and allows fast inference at decision time, but it is also inflexible, it does not allow fast updates to the value estimate. On the other hand, the model-based systems is more flexible, since value predictions are estimated on the fly. However estimating values at decision time

require more computational effort. Therefore the two models represents the two ends in the trade-off of computational efficiency and flexibility [45]. It has been suggested that humans arbitrate between the two systems based on uncertainty estimates of each system. This trades-off the expected benefits from acquiring more accurate value estimates against the computational/time cost of building a larger tree [44].

This two system structure is also reflected in the AlphaZero algorithm. The deep neural network functions similarly as the model-free system. It 'stores' pairs of state and long-term value associations and it is capable of fast inference at decision time. The MCTS is a model-based system, which has a higher computational cost, but allows quicker updates. However in the AlphaZero algorithm the deep neural network plays a supporting role for the MCTS module, and at each decision step, MCTS is used for deciding on the current action. Furthermore, model-based RL algorithms are in general only concerned with learning models for predicting immediate, one-step transitions. In contrast, the AlphaZero algorithm assumes access to a perfect environment model, which enables multi-step or even complete roll-outs. Still, the basic ideas of combining caches of long-term value estimates with a computationally more demanding planning algorithm is present in the AlphaZero algorithm.

Interactions between the two components are present in the AlphaZero algorithm, but possible further benefits are not fully exploited in the algorithm. Under time constraints it would be more computationally efficient to concentrate search efforts by building larger trees on uncertain regions. As a result of building larger trees, more accurate training data could be gathered from these regions, which could fasten up learning. This reflects the computational efficiency-flexibility trade-off of model-based and model-free RL. The previous chapter showed that using heuristics to control the search effort in the learning process based on root return variance based uncertainty estimate can improve the performance (4-4). The variance of the returns indicates whether it is necessary to build a larger search tree to acquire more accurate estimates or the algorithm is fairly certain in the current state's estimates and therefore computational efforts can be saved with a smaller search tree.

5-2 Connected research directions and future work

The combination of deep learning and search-based planning holds a great potential for multiple fields, e.g. motion panning [46] and robotic manipulation. In order to exploit the benefits of local search, the algorithm needs a forward model of the environment dynamics. Knowledge of the environment dynamics allows building local search trees, which enables effectively searching the state space. In the original AlphaZero algorithm, this constituted as knowledge of the game rules. In more complex environments, access to a forward model of the environment dynamics is more problematic. Two directions can be distinguished, which leverage the advantages of model-based planning, but are different in their treatment of the environment model. The first directions is to use physics engines or environment emulators. The experiments in this thesis used this approach. The second approach is to learn the environment model from interaction data. This is the traditional model-based RL approach.

5-2-1 Physics engines and sim2real

One possible approach to exploit the benefits of combined search and deep learning in complex environments is to use physics engines. Relying on an environment simulator to learn the policy offers an alternative to model-learning. Simulation based learning provides several advantages, including resettable conditions, experiment parallelization and enabling multi-step planning. One of the drawback of physics engines is that they are often computationally heavy from rendering and calculating physics that are not immediately relevant for the current task. Physics engines can be used to learn a policy, which is then transferred to the real world problem. Sim2real is the collective name of algorithms, which transfer policies learned in simulators to the real world [47]. The sim2real approach has a great potential in robotics applications, where real world learning can be costly. Combining the strengths of multi-step planning and deep learning can be beneficial for these approaches.

5-2-2 Model learning and the successive representation

Interest in model-based approaches have been motivated by their sample-efficiently compared to model-free algorithms. Model-based RL algorithms learn a model of the MDP, which can be used as an internal simulator for planning and policy learning. It has been shown previously that model-based algorithms can outperform model-free algorithms in sample-efficiency in tabular settings. [48]. The sample-efficiency of model-based algorithms is especially relevant in problems, where limiting the number of real environment interactions is important due to safety and cost considerations (e.g. robotics) [49].

Recent model-based RL approaches include recurrent neural network based models [50] and graph neural networks. Graph neural networks could be a promising approach for handling intuitive physics by treating intuitive physics as inference over a learned physics engine [51]. This model could also allow imagination-based planning, similarly to MCTS in the AlphaZero algorithm. Many model-based RL algorithms have focused on learning one-step transition dynamics. These suffer from compounding model errors in multi-step predictions. Small inaccuracies in the learned transition model can result in large deviations from the true state after multiple predictions, which makes predicting future rewards difficult [52]. Using successor features has been proposed as an alternative approach to confront the compounding errors of one-step transition models [53]. Successor features are a continuous extension of the successor representation. The successor representation (SR) [54] is based on reformulating the value function into a linear combination of state transitions and learned reward representation. As a result learning the value function is decoupled into learning the expected state occupancy (successor representation) and goal-specific features of the reward. The successor representation is also interesting, since it forms a bridge between model-based and model-free RL. Compared to the model-based RL, planning is simplified with SRs, since expectations of future state occupancies are stored in the successor representation and quickly available, similarly to values in model-free RL. The main relevance of SR to the AlphaZero algorithm is that SR also introduces temporality to model-free learning through storing multi-step relations between states. Using SR could exploit advantages of both model-free and model-based RL, similarly to the AlphaZero algorithm. Another advantage of the value function decomposition in the successor representation approach is that it separate dynamics from the reward definition, therefore allows faster transfer learning for changed reward definitions [55].

Chapter 6

Conclusions

One key innovation in the AlphaZero algorithm was the combination of the deep neural networks with Monte Carlo tree search based planning. The algorithm's success in the game of Go has proved its capabilities in a very complex environment. The main hypothesis of this thesis was that the combination of local information from tree search and the generalization capacity of deep neural networks was of key importance to the algorithm's success. The thesis has examined the importance of generalization and locality in the AlphaZero algorithm in single-player, deterministic reinforcement learning environments through two research questions.

The first research question examined what changes need to be made to transfer the AlphaZero algorithm to single-player RL environments, where the return is no longer bounded in [0,1] as in the game of Go. First, possible problems arising from the differences in the reward function definition of board games and single player environments were identified. It was shown that if assumptions in the tree traversal policy of Monte Carlo tree search are violated, the algorithm can fail to learn in the new environment. Several modifications were suggested, which could render the AlphaZero algorithm more suitable for environments with reward distributions that do not have a support in [0,1]. Experiment results have shown that the suggested modifications improve the algorithm's performance in the new environments. These included adaptively normalizing the returns during MCTS. Adaptive return normalization is a promising approach in RL, since return values can span a wide range. The main results of the first research question was highlighting the importance of return ranges and offering an adaptive solution for return normalization, which does not rely on prior knowledge of the return range.

The second research question's hypothesis was that under time constraints, balancing local information and generalization is of crucial importance for the algorithm's success. This question was investigated through changing the number of MCTS iterations and comparing the algorithm's performance for the different settings under a fixed training time in several different environments. These environments were all single-player, deterministic, perfect-information environments and experiments built on results from the first research question. Executing a small number of MCTS iterations puts more emphasis on generalization, since

the network is trained more often. On the other hand, executing a higher number of MCTS iterations puts emphasis on local search and less time remains for training the network and generalization. The main finding of these experiments was intermediate sized trees achieve the best episodic return based performance. Visualizing the learning process through evaluating the policy and value network predictions over a fixed state space grid allowed further insights into the trade-off between generalization and locality. As expected, the network prediction plots have shown that using a low number of MCTS iterations results in inaccurate value estimates, and the value network fails to converge or converges to false values. On the other hand, a high number of MCTS iterations slows down the learning process significantly and the network fails to converge under the time constraints. For intermediate size trees, MCTS provides accurate training samples and the network is updated for a sufficient number of times to ensure convergence under the time budget. The results from the locality-generalization trade-off experiments has motivated adapting the number of MCTS iterations to the learning process. Under strict time constraints, focusing search efforts on more uncertain regions of the state space by executing a higher number of MCTS iterations, while saving resources in regions, where the algorithm is already fairly certain in the values can bring performance improvements. A return variance based approach has been proposed to achieve this behaviour. The experiment results of the return variance based adaptive MCTS approach have shown improved performance and faster convergence of the network.

Based on the results of this thesis, it can be concluded that balancing local information from tree search and generalization of deep learning was of key importance to the success of the AlphaZero algorithm in achieving professional level performance in the game of Go.

The AlphaZero algorithm has undoubtedly a significant impact on AI research. By combining many novel AI techniques it has managed to defeat the top level human player in the game of Go, which has been considered a major challenge in AI research. The results in the game of Go has shown that the algorithm is capable of learning tabula rasa, discovering knowledge about the task independently in a self-learning mechanism. Overall, the success of the AlphaZero on the game of Go is not only a milestone in AI research, but also an important step forward in understanding how humans learn to excel at tasks, which will have an important impact on further AI research.

Appendix A

Appendix

A-1 Environment emulators

A key assumption of the AlphaZero algorithm is that the agent has access to the ground truth model of the environment. Having access to the environment model makes it possible for the agent to perform Monte Carlo tree search before each action selection in the real environment. In case of the games of Go and chess, the knowledge of the environment equals knowledge of the game rules. In the experiments of this thesis, the agent has access to replicates of the environment emulators. Several software packages exist, which offer environment emulators for RL algorithms. In this thesis, the OpenAI toolkit [56] and PyBullet [57]. are used. OpenAI is a popular toolkit for developing reinforcement learning algorithms. By providing the environment part of the RL framework, it allows the comparison and benchmarking of RL algorithms. Another option for environment emulators is pybullet, which has a python interface to the physics engine Bullet [57].

The Cart-pole system

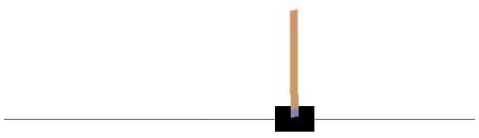


Figure A-1: Visualization of the the cart-pole problem from OpenAI gym

The cart-pole system is a classical control problem, where the goal is to balance a pole attached to a moving cart by a joint, through exerting force on the cart. It is a relatively simple problem, which is often used for control benchmarks. The initial state of the pole is in the upright position. When using a reinforcement learning approach to solve the cart-pole problem, the agent has to learn a policy, that gives the correct action for a given state for keeping the pole in an upright position.

The OpenAI [56] RL environment has a discrete action space with two actions $|A| = 2$ and a continuous state

space $S \in \mathbb{R}^4$. The possible action are moving the cart left or right. The states vector is defined as $s = [x, \dot{x}, \theta, \dot{\theta}]$, where x is the cart position, $\dot{x}$ is the cart velocity, θ is the pole angle and $\dot{\theta}$ is the angular velocity of the pole. The learning is in an episodic setting, where an episode terminates if the cart-pole system violates states bounds or the maximum number of episode steps is reached of balancing the pole ($t_{max} = 200$). The agent receives a positive unit reward for every non-terminal step of keeping the pole in an upright position without violating state bounds. The reward function is defined as r_t in A-1. The reward function can be normalized by dividing the one step reward by the maximum episode reward and giving -1 for the pole falling over as $r_{t,norm}$ (A-1).

$$r_t(s,a) = \begin{cases} 1 & \text{if } s \text{ not terminal} \\ 0 & \text{else} \end{cases} \qquad r_{t,norm}(s,a) = \begin{cases} 0.005 & \text{if } s \text{ not terminal} \\ -1 & \text{if terminal} \end{cases} \tag{A.1}$$

The Mountain-car system

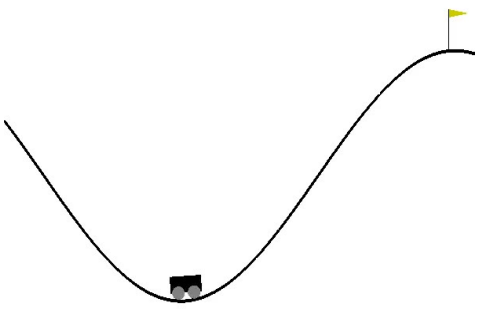


Figure A-2: Visualization of the the mountain-car problem from OpenAI gym

The mountain-car problem is a RL algorithm benchmark problem, where an under-powered car has to reach the top of a hill from a valley. Reaching the top of the hill from the initial state is not possible in one-go, since the motor of the car is too weak to defeat gravity, the agent has to learn to gain momentum. The mountain car problem is a more challenging problem exploration-wise. The agent has to learn to escape the local minima of the valley, which is only possible if the state space is sufficiently explored. This is challenging, since the agent only received a positive reward, when reaching the top of the hill. Although the state space and action spaces are discrete as in the case for the cart-pole problem, exploration makes the mountain-car problem challenging for RL algorithms.

The OpenAI [56] RL environment has a discrete action space $|A| = 3$ and a continuous state space $S \in \mathbb{R}^2$.The possible actions are moving left, moving right, or doing nothing. The state vector is defined as $s = [x, \dot{x}]$, where x is the car position, $\dot{x}$ is the car velocity. The learning is in an episodic setting, where an episode terminates if the car reaches the top of the mountain or taking 200 environment steps. The agent receives a unit negative reward for every time step, until reaching the goal position.

The reward function is defined as

$$r_t(s,a) = \begin{cases} -1 & \text{if } s \text{ not terminal} \\ 0 & \text{else} \end{cases} \qquad r_{t,norm}(s,a) = \begin{cases} -0.005 & \text{if } s \text{ not terminal} \\ 1 & \text{if terminal} \end{cases} \tag{A.2}$$

In case of the mountain-car problem the reward function can be normalized by dividing the minimum episodic reward, taking the episodic maximum number of real environment steps, without reaching the goal (-200). In this case a plus one reward was also given for reaching the top of the hill ($r_{t,norm}$ in A.2) .

The Racecar problem

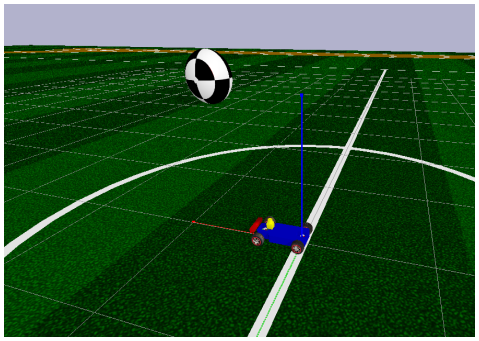


Figure A-3: Visualization of the the racecar problem from pybullet gym

The racecar problem is an environment emulator available from pybullet [57]. In this environment, the simulated the MIT RC racecar has to be controlled to reach a static goal (a football in this caseA-4). This is a simple navigation task, with no obstacles present in the environment. Since a physics engine runs in the background, the simulation is slower than in the simpler openAI environments.

The RL environment has a discrete action space with nine possible actions $|A| = 9$ and a continuous state space $S \in \mathbb{R}^2$. The possible actions are defined as a combination of ({not moving,moving back,moving forward } and { not steering,steering right, steering left }). The state vector is defined as the ball position in camera frame, $s = [x, y]$, where x and y are camera frame coordinates. The learning is in an episodic setting, where an episode terminates after taking 20 actions. The action repetition can be controlled. In these simulations it was left at default 50 repetitions. Therefore each episode consists of 1000 steps in the real environment. The reward function is defined as the distance between the car and the football.

$$r_t(s, a) = \sqrt{x^2 + y^2}$$

The Acrobot problem

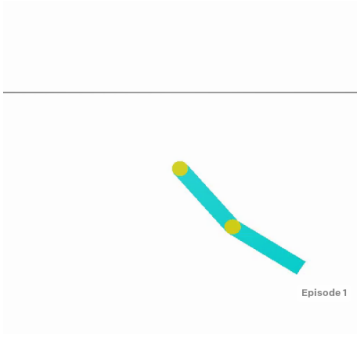


Figure A-4: Visualization of the the acrobot problem from OpenAI gym

The acrobot environmen is available from OpenAI gym [56]. The acrobot is a two-joint pendulum swing-up problem, where only the second joint is actuated. The goal is to swing up the end-effector higher than the baseline plus one link height. The problem has a discrete action space with three possible actions $|A| = 3$, and a four dimensional state space $S \in \mathbb{R}^4$. The possible actions are either applying +1, 0 or -1 torque on the joint between the two links. The state consists of the sine and cosine of the two rotational joint angles and the joint angular velocities. The learning is in an episodic setting, where an episode terminates after 200 environment steps or if the acrobot's link reaches above a certain height.

The reward function is defined as

$$r_t(s, a) = \begin{cases} -1 & \text{if } s \text{ not terminal} \\ 0 & \text{else} \end{cases}$$

A-2 Experiment settings

This section describes considerations and specifics about the experiments settings in the thesis.

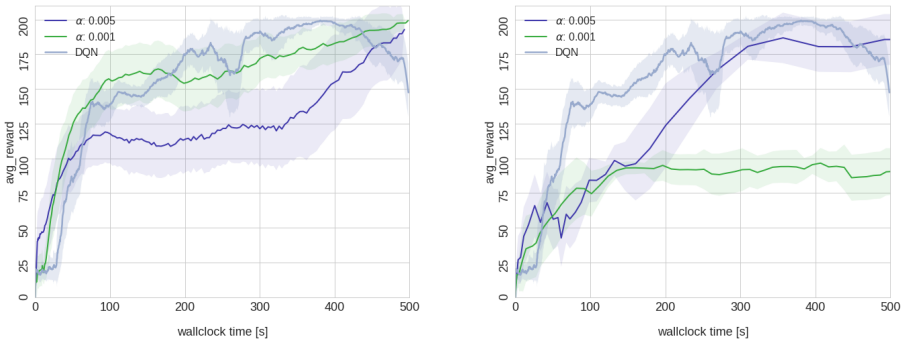
A-2-1 Hyperparameter tuning

Hyperparameter optimization is the process of finding the optimal set of parameter for a parametric machine learning algorithm, which minimizes the generalization error of the learning algorithm. In RL, hyperparameter search means finding a set of hyperparameters, which maximizes the long-time returns from the environment specific reward function. Contrary to model parameters, hyperparameters control the algorithm's behaviour and are not estimated from data. Initializing the learning process with a set of hyperparameters plays a crucial role in the performance of the learning algorithm. However, finding the optimal set of hyperparameter over the whole hyperparemer space is usually not feasible, due to the very high dimensional search space and limited time/computational budget. The simplest hyperparameter search method is grid search, which tries combinations of selected hyperparameters. This makes the search computationally more feasible, but the grid design requires intuition about the problem. In this thesis grid search was used for hyperparameter tuning. An alternative to the grid search is random search, which builds on the low effective dimensionality of the hyperparameter optimization problems [58]. The low effective dimensonality means that only a small subset of the hyperparameters has a measurable effect on the learning algorithm's performance and therefore it is a better approach to use randomly chosen trials than trial on a grid.

Hyperparameter tuning in the AlphaZero algorithm The set of hyperparameters to be tuned is algorithm specific. In deep reinforcement learning approaches hyperparameters include optimization hyperparameters (learning rate and other optimization algorithm-specific hyperparameters), hyperparameters of the deep neural network architecture (number of hidden layers, units, activation functions), exploration controlling hyperparameters, etc. In the AlphaZero algorithm, hyperparameters are also have to be set for the Monte-Carlo tree search (number of node expansions per iteration, upper-confidence bound parameter in PUCT, etc). This makes the hyperparameter search space very complex.

Hyperparameter tuning for the experiments On of the major questions of this thesis was balancing locality and generalization. This question can be approached by varying the number of MCTS node expansions per iteration (n_{MCTS}) parameter and keeping the rest of the hyperparameters fixed to a pre-tuned value. The hyperparameter search was carried out for only one n_{MCTS} value, which was chosen based on the number of possible actions in the environment. This might introduce bias towards the n_{MCTS} value (figure A-5), for which the rest of the hyperparameters were chosen, but due to time limitations the research question was only evaluated for one set of tuned hyperparameters.

Other indicators of hyperparameter setting performance The difficulty of the learning problem also defines the hyperparameter tuning process. In simpler environments (e.g. cart-pole problem), hyperparameter search can be carried out in the chosen maximum time budget. In more complex environment, the hyperparameter search were carried out for a shorter time



(a) $n_{MCTS} = 8$ **(b)** $n_{MCTS} = 64$

Figure A-5: Cart-pole hyperparmeter tuning for the AlphaZero algorithm. Figures illustrate change of optimal learning rate for different number of node expansion steps n_{MCTS} in MCTS search under a limited time constraints in case of the cart-pole problem. DQN performance illustrated only for algorithm performance reference.

than the environment-specific maximum time budget due to computational limitations. In case of a sparse reward function, the agent might not encounter positive rewards during the training process. In this case indicators othern than the episodic reward sum are helpful for the tuning process. In 2D state spaces (such as the mountaincar and racecar problems), visualizing the state space can give an indication of the agent's performance with the hyperparameter setting. Figure A-6 shows the tuning process for the mountain-car problem.

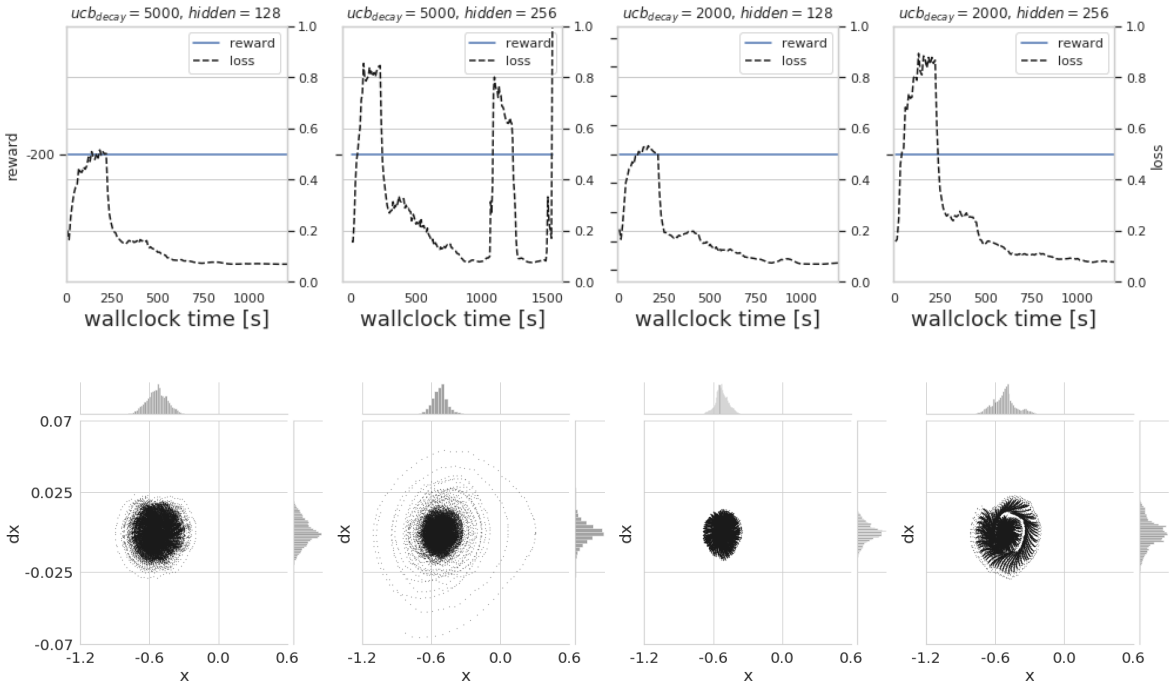


Figure A-6: Subset of reward and state space exploration plots for mountain-car hyperparameter tuning for the AlphaZero algorithm. The plots demonstrate that indicators othern than the episodic reward sum might be useful during hyperparameter search.

A-2-2 Architectural choices and optimization choices in the neural network

In the original AlphaZero algorithm a convolutional neural network is used in order to achieve generalization power without over-fitting due to the large state space. In these experiments simple environment are used, where the game states are low dimensional (e.g. position values), therefore two fully-connected layers are used in each environment with the same number of hidden units $h_1 = h2$ (Table A-1). The ADAM optimizer (A-5) is used in the network updates. The network is trained after every episode in each experiment for one epoch over the replay memory.

A-2-3 Hyperparameter setting details of the experiments

The following table shows environment specific details about the hpyerparameter settings used in the experiments for the cart-pole (CP), mountain-car (MC), acrobot (AC) and racecar (RC) environments. Along with details about the neural network and MCTS hyperparameter, the table also shows hyperparameter for the extensions: return normalization (equation 3-7) and adaptive n_{MCTS} (equation 4-1). The mountain-car experiment was not ran with return normalization and the cart-pole problem was not run with adaptive n_{MCTS}, therefore these are left empty in the table. α denotes learning rate , ϵ is stability parameter in the ADAM optimizer (A-5), h_1 and h_2 are the number of hidden nodes in the first and second fully-connected layers and *db* refers to the size of the replay memory. The exploration-exploitation parameter c_{UCB} in UCT (equation 2-15) is decayed from c_{MAX} to c_{MIN} over the specified decay steps. In the main experiments, the temperature parameter τ (equation 2-13) was left at 1.0.

	neural network hyperparameters					MCTS hyperparameters				return norm. (3-7)		adaptive n_{MCTS} (4-1)		
	α	batch size	ϵ	$h_1 = h_2$	db size	τ	c_{MAX}	c_{MIN}	decay steps	β	η	n_{base}	n_{min}	n_{max}
CP	1e-3	16	1e-4	256	5e3	1.0	0.8	0.05	500	1-3	1e3	-	-	-
MC	1e-3	16	1e-6	128	5e3	1.0	5	0.5	5000	-	-	16	8	64
AC	5e-4	16	1e-4	256	5e3	1.0	1.0	0.05	5000	1e-3	1e3	32	8	32
RC	1e-3	16	1e-5	256	5e3	1.0	1.0	0.05	1500	1e-2	1e3	16	9	64

Table A-1: Hyperparameter settings of neural network, Monte Carlo tree search and algorithm extensions for cart-pole (CP), mountain-car(MC), acrobot(AC) and racecar (RC) environments.

A-3 Other result visualizations

A-3-1 Newick tree visualization

The Newick tree representation summarizes search trees in a string descriptor. Trees defined by a parent-child node class structure are converted to a Newick tree representation, which can be visualized using existing Python packages (ete2 bioinformatics tool [59]).

Example of Newick string and corresponding tree visualization:

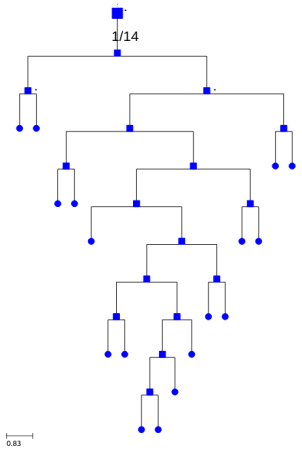


Figure A-7: Newick tree example

((((.,.),(((.,.),(((.,.),((.,(.,(.,.))),(.,.))),.)),(.,.))),(.,.)));

A-4 Entropy

The (Shannon) entropy [60] is an information theoretic measure and represents a degree of randomness of a random variable. The the entropy of a discrete random variable, with probability mass function p can be calculated as:

$$H(X) = -\sum_{x} p(x)\log(p(x)) \tag{A.3}$$

If the discrete random variable can take K values, and p_i indicated the probability of the variable taking the i_th value, the entropy can be calculated as following:

$$H(p_1, p_2, ..p_K) = -\sum_{i=1}^{K} p_i \log(p_i) \tag{A.4}$$

The entropy is non-negative and bounded,

$$0 \leq H \leq \log(K)$$

0 entropy corresponds to a deterministic variable and maximum entropy occurs in case of a uniform distribution, when the variable can take the k values with equal probability $\sum^{K} \frac{1}{K}\log(K) = \log(K)$.

Given a vector of observed counts, entropy can be also used to give a measure of uniformity of the vector values through the normalized counts. Entropy can also be taught as a measure of the expected uncertainty of the random variable. In case of the AlphaZero algorithm, the entropy of the child action counts from the Monte Carlo tree search can give an indication about the policy network's certainty of the action predictions in the current state. Initially, the policy network predicts equal probability of all actions, therefore the entropy takes it's maximum value $\log |A|$, which gradually decreases, as the policy network converges. Therefore, the entropy could give and indication the convergence of the policy network [61].

A-5 The ADAM optimizer

The Adaptive Moment Estimation (ADAM) [62] optimization algorithm computes adaptive learning rates for each parameter in deep neural network optimization. The ADAM utilizes the means (first moment) and variances (second moment) of the gradients, which are estimated as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \end{aligned} \tag{A.5}$$

where m_t is the first moment estimate and v_t is the second moment estimate at update step t and β_1 and β_2 are the exponential decay rates for the first and second moment estimates. As the moments are initialized as zero vectors, they are biased towards zero. These biases are confronted vis bias correction:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{A.6}$$

The ADAM parameter update rule for the network weights θ:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t, \tag{A.7}$$

where α is the learning rate or step size and ϵ is for numeric stability.

Bibliography

[1] L. V. Allis, "Searching for solutions in games and artificial intelligence," 1994.

[2] Photograph by Saran Poroong, "Go board view from above." https://www.dreamstime.com/stock-photo-go-board-view-above-ended-game-image62665392, 2015. [Online; accessed February 10, 2018].

[3] L. Kocsis and C. SzepesvÃąri, "Bandit based monte-carlo planning," in *In: ECML-06. Number 4212 in LNCS*, pp. 282–293, Springer, 2006.

[4] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artif. Intell.*, vol. 175, pp. 1856–1875, July 2011.

[5] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, "A0C: alpha zero in continuous action space," *CoRR*, vol. abs/1805.09613, 2018.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[9] J. Kober, J. A. D. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *International Journal of Robotics Research*, July 2013.

[10] S. B. Thrun, "Efficient exploration in reinforcement learning," tech. rep., Pittsburgh, PA, USA, 1992.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 02 2015.

[12] A. Gelman, *Bayesian Data Analysis*. Texts in statistical science, Chapman & Hall/CRC, 2004.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, 2002.

[14] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, S. Tavener, D. Perez, S. Samothrakis, S. Colton, and et al., "A survey of monte carlo tree search methods," *IEEE Transactions On Computational Intelligence and AI*, 2012.

[15] T. M. Moerland, J. Broekens, and C. M. Jonker, "Double uncertain exploration," 2017.

[16] W. R. Thompson, "On the Likelihood that one Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika*, vol. 25, pp. 285–294, 1933.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[18] M. A. Nielsen, "Neural networks and deep learning," *Determination Press*, 2015.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[21] C. Finn and S. Levine, "Deep visual foresight for planning robot motion," *arXiv preprint arXiv:1610.00696*, 2016.

[22] M. Hessel, J. Modayil, H. P. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *AAAI*, 2018.

[23] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[24] R. S. Sutton, "Integrated architecture for learning, planning, and reacting based on approximating dynamic programming," in *Proceedings of the Seventh International Conference (1990) on Machine Learning*, (San Francisco, CA, USA), pp. 216–224, Morgan Kaufmann Publishers Inc., 1990.

[25] D. Silver, R. S. Sutton, and M. Müller, "Temporal-difference search in computer go," *Machine Learning*, vol. 87, pp. 183–219, May 2012.

[26] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, pp. 1856–1875, 2011.

[27] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron, "The challenge of poker," *Artificial Intelligence*, vol. 134, no. 1, pp. 201 – 240, 2002.

[28] G. Tesauro and G. R. Galperin, "On-line policy improvement using monte-carlo search," in *Advances in Neural Information Processing Systems 9* (M. C. Mozer, M. I. Jordan, and T. Petsche, eds.), pp. 1068–1074, MIT Press, 1997.

[29] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, (Berlin, Heidelberg), pp. 282–293, Springer-Verlag, 2006.

[30] X. Guo, S. P. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 3338–3346, 2014.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[32] H. P. van Hasselt, A. Guez, A. Guez, M. Hessel, V. Mnih, and D. Silver, "Learning values across many orders of magnitude," in *Advances in Neural Information Processing Systems 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 4287–4295, Curran Associates, Inc., 2016.

[33] M. Hessel, H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. van Hasselt, "Multi-task deep reinforcement learning with popart," *CoRR*, vol. abs/1809.04474, 2018.

[34] T. Vodopivec, S. Samothrakis, and B. Šter, "On monte carlo tree search and reinforcement learning," *J. Artif. Int. Res.*, vol. 60, pp. 881–936, Sept. 2017.

[35] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *NIPS*, 2017.

[36] N. D. Daw, "Are we of two minds?," *Nature Neuroscience*, vol. 21, 10 2018.

[37] B. Ratitch and D. Precup, "Sparse distributed memories for on-line value-based reinforcement learning," in *Machine Learning: ECML 2004* (J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, eds.), pp. 347–358, Springer Berlin Heidelberg.

[38] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning for control," *Artif. Intell. Rev.*, vol. 11, pp. 75–113, Feb. 1997.

[39] D. Hafner, D. Tran, T. Lillicrap, A. Irpan, and J. Davidson, "Reliable Uncertainty Estimates in Deep Neural Networks using Noise Contrastive Priors," *ArXiv e-prints*, July 2018.

[40] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in monte carlo tree search," in *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 389–395, Dec 2008.

[41] J.-Y. Audibert, R. Munos, and C. Szepesvári, "Exploration-exploitation tradeoff using variance estimates in multi-armed bandits," *Theoretical Computer Science*, vol. 410, no. 19, pp. 1876–1902, 2009.

[42] D. Silver, *Reinforcement Learning and Simulation-based Search in Computer Go*. PhD thesis, Edmonton, Alta., Canada, 2009. AAINR54083.

[43] S. Levine, P. P. Sampedro, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," 2017.

[44] N. D. Daw, Y. Niv, and P. Dayan, "Uncertainty-based competition between prefrontal and dorsolateral striatal systems for behavioral control.," *Nat Neurosci*, vol. 8, pp. 1704–1711, Dec. 2005.

[45] S. J. Gershman, E. J. Horvitz, and J. B. Tenenbaum, "Computational rationality: A converging paradigm for intelligence in brains, minds and machines," *Science*, vol. 349, pp. 273–278, 2015.

[46] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov, "Combining neural networks and tree search for task and motion planning in challenging environments," *CoRR*, vol. abs/1703.07887, 2017.

[47] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," *CoRR*, vol. abs/1804.10332, 2018.

[48] I. Szita and C. Szepesvári, "Model-based reinforcement learning with nearly tight exploration complexity bounds," in *ICML*, pp. 1031–1038, Omnipress, 2010.

[49] M. Deisenroth and C. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, pp. 465–472, Omnipress, 2011.

[50] S. Racaniere, T. Weber, D. Reichert, L. Buesing, A. Guez, D. Jimenez Rezende, A. Puigdomenech Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. Silver, and D. Wierstra, "Imagination-augmented agents for deep reinforcement learning," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 5690–5701, Curran Associates, Inc., 2017.

[51] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, "Graph networks as learnable physics engines for inference and control," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80 of *Proceedings of Machine Learning Research*, (StockholmsmÃďssan, Stockholm Sweden), pp. 4470–4479, PMLR, 10–15 Jul 2018.

[52] E. Talvitie, "Self-correcting models for model-based reinforcement learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.* (S. P. Singh and S. Markovitch, eds.), pp. 2597–2603, AAAI Press, 2017.

[53] L. Lehnert and M. L. Littman, "Successor features support model-based and model-free reinforcement learning," 2019.

[54] P. Dayan, "Improving generalization for temporal difference learning: The successor representation," *Neural Computation*, vol. 5, p. 613, 1993.

[55] A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. P. van Hasselt, and D. Silver, "Successor features for transfer in reinforcement learning," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 4055–4065, Curran Associates, Inc., 2017.

[56] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[57] E. Coumans, "Bullet physics simulation," in *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, (New York, NY, USA), ACM, 2015.

[58] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[59] J. Huerta-Cepas, J. Dopazo, and T. Gabaldón, "ETE: a python environment for tree exploration," *BMC Bioinformatics*, vol. 11, p. 24, 2010.

[60] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 7 1948.

[61] L. Devroye, L. Gyorfi, and G. Lugosi, *A Probabilistic Theory of Pattern Recognition*. Springer, 2014.

[62] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization.," *CoRR*, vol. abs/1412.6980, 2014.

Glossary

List of Acronyms

AI	artificial intelligence
RL	Reinforcement learnig
MDP	Markov decision process
TD	temporal difference
DP	Dynamic programming
MC	Monte Carlo
DRL	Deep reinforcement learning
MCTS	Monte Carlo tree search
MAB	multi-armed bandit