Universität Hamburg
MIN Faculty
Department of Informatics
BSc Informatics
at Research Group Knowledge Technology, WTM

Bachelor's Thesis

Improving Model-Based Reinforcement Learning with Internal State Representations through Self-Supervision

by

Julien Scholz
7065721

Thesis advisors:
Dr. Cornelius Weber,
Dr. Muhammad Burhan Hafez

Abstract

Using a model of the environment, reinforcement learning agents can plan their future moves and achieve super-human performance in board games like Chess, Shogi, and Go, while remaining relatively sample efficient. As demonstrated by the MuZero Algorithm, the environment model can even be learned dynamically, generalizing the agent to many more tasks while at the same time achieving state-of-the-art performance. Notably, MuZero uses internal state representations instead of real environment states for its predictions. In this thesis, we introduce two additional, independent loss terms to MuZero's overall loss function, which work entirely unsupervised and act as constraints to stabilize the learning process. Experiments show that they provide a significant performance increase in simple OpenAI Gym environments. Our modifications also enable self-supervised pretraining for MuZero, meaning the algorithm can learn about environment dynamics before a goal is made available.

Zusammenfassung

Mithilfe eines Modells der Umgebung können Reinforcement Learning Agenten vorausplanen, und so beispielsweise Menschen in Brettspielen wie Schach, Shōgi und Go schlagen, ohne dafür viele Trainingsbeispiele zu benötigen. Insbesondere hat der MuZero Algorithmus bestätigt, dass das Modell durch Interaktion mit der Umgebung erlernt werden kann (und somit nicht bereitgestellt werden muss) ohne dabei die herausragenden Ergebnisse zu beeinträchtigen. MuZero verwendet hierfür interne Repräsentationen der Umgebungszustände. In dieser Bachelorthesis beschreiben wir zwei zusätzliche Loss-Terme, welche vollständig unüberwacht funktionieren, und MuZeros Loss-Funktion beigefügt werden können um den Lernprozess zu stabilisieren. Experimentell können wir so eine Verbesserung der Leistung in einfachen OpenAI Gym Umgebungen zeigen. Zusätzlich erlauben die Änderungen, dass der MuZero Agent die Mechaniken seiner Umgebung bereits vor Einführung eines Ziels erlernen kann.

Note: Figures that do not include a citation were drawn by the author.

Contents

1 Introduction

The field of artificial intelligence (AI) has been extremely popular over recent years, due to the wide range of applications the technology has developed with the help of modern computer hardware. Machine learning algorithms are capable of outperforming humans at a variety of different tasks while being significantly faster and cheaper to operate. Possible applications include self-driving vehicles, computer-aided interpretation of medical imagery, stock market analysis, and robotics. However, despite their sometimes impeccable performance, current artificially-intelligent programs have a common drawback, which is their inability to generalize to other tasks. Each algorithm is developed by human experts specifically to solve a single problem and is hence referred to as *artificial narrow intelligence.* A program capable of learning many different tasks, similar to a human, is called *artificial general intelligence* (*AGI*), and developing such a program is the holy grail of AI research.

Learning systems can be subdivided into three categories, namely *supervised learning*, *reinforcement learning*, and *unsupervised learning.* In supervised learning, the learner is presented with examples of input-output combinations, for which it must adopt a mapping that generalizes well to previously unseen inputs. For some problems, however, no sufficient number of examples that include their respective solutions (or outputs) exist, making supervised learning intractable. Consider the movement of a bipedal robot. Gathering a large set of training examples on how to walk for a variety of different scenarios for a learning system would require an already existing solution to the problem we are trying to solve. It is, however, relatively trivial to judge the robot's performance. For instance, the act of falling over is easy to detect and clearly undesirable, whereas a steady forwards movement should be encouraged. Learning, that is only based on this feedback, is called reinforcement learning. The final category, unsupervised learning, is used for data in which neither a perfect solution nor a rating (as in the former categories) is available. These algorithms try to structure data by finding hidden patterns or similarities, which may, for example, be used in data visualizations.

While supervised learning is widely applicable and even considered somewhat solved by some people, reinforcement learning is still in its infancy. This is not due to an absence of use cases, however. A grasping motion of a robotic arm works well with a pre-written script in a factory setting but requires a learning system to deal with the disorderliness of the real world. Despite remarkable strides being made in recent times, with reinforcement learning algorithms playing complex board and video games at a superhuman level, *sample efficiency*, the effectiveness with which the algorithm learns from few training examples, continues to be a big issue. Sample efficiency is especially critical in the field of robotics, as real-world training is costly and throttled by physical limitations, whereas simulations are computationally expensive and inaccurate.

Reinforcement learning algorithms are further classified into two groups. There are *model-free* algorithms, which can be viewed as behaving instinctively, and *model-based* algorithms, which can plan their next moves ahead of time. The latter is arguably more human-like and has the added advantage of being comparatively sample efficient. Furthermore, it is clear why a robot interacting with an environment, for instance, by catching a ball, would benefit from anticipating the behavior of other objects within its vicinity, especially when we consider the delay between sensory information arriving and its motors being actuated.

The *MuZero Algorithm* [Schrittwieser et al., 2019] has made recent breakthroughs by becoming the new state-of-the-art model-based reinforcement learning algorithm. It outperforms its competitors in a variety of tasks, while simultaneously being very sample efficient. This makes it appear to be a fairly general-purpose algorithm and an excellent candidate for our experiments involving robot grasping (see Figure 1).

Unfortunately, upon further investigation, we realized that MuZero performed relatively poorly on our robot experiments and was slow to operate. Contrary to our expectations, older, model-free algorithms such as *A3C* [Mnih et al., 2016] delivered significantly better results while being more lightweight and comparatively easy to implement. Even in very basic tasks, our custom implementation as well as one provided by other researchers, *muzero-general* [Werner Duvaud, 2019], produced unsatisfactory outcomes.

Since this contradicts the results provided in the MuZero publication, we are led to believe that MuZero may require a team of experts and expensive hardware to reach its full potential. We question some of the design decisions made for MuZero, namely, how unconstrained its learning process is. After explaining all

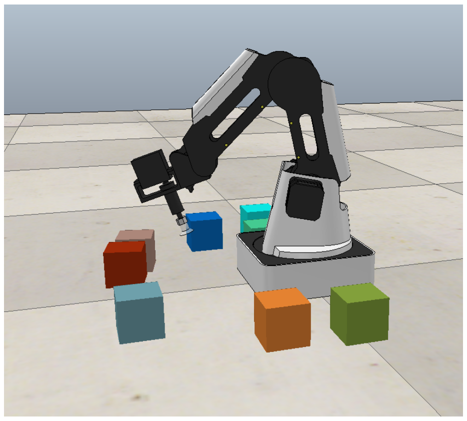


Figure 1: A *Dobot* robotic arm being tasked with picking up and stacking simulated cubes in *CoppeliaSim* (`https://www.coppeliarobotics.com/`).

necessary fundamentals, we propose two individual augmentations to the algorithm that work fully unsupervised in an attempt to improve its performance and make it more reliable. Afterward, we evaluate our proposals by benchmarking the regular MuZero algorithm against the newly augmented creation.

2 Background

In this chapter, we introduce some of the fundamentals necessary to understand the changes we set out to make further on. In particular, we look at reinforcement learning from a more mathematical standpoint and give a brief overview of its history. We also describe model-based reinforcement learning formally before eventually explaining the MuZero Algorithm.

2.1 Reinforcement Learning

When talking about reinforcement learning (*RL*), we refer to a learning instance interacting with its surroundings and improving its behavior over time to reach a given goal. As an example, a learner could play the board game Chess by interacting with the pieces on the board and aiming to reach a winning state.

Formally, reinforcement learning is the act of learning a mapping from *observations* to *actions* so as to maximize a real-valued *reward* [Sutton and Barto, 2018]. In contrast to one of the other learning paradigms, supervised learning, the learner is not told which actions result in the largest reward for different observations. Instead, the learner must try out various actions in different situations to discover useful behavior.

The principles behind reinforcement learning are inspired by the way animals learn. Behavioral psychology has found that actions can be encouraged or suppressed when they are followed up with positive or negative stimuli [Thorndike, 1898].

2.1.1 Markov Decision Process

The aforementioned interaction of a learner with its surroundings to achieve a goal can be modeled mathematically through a *Markov Decision Process* (or *MDP*) [Sutton and Barto, 2018]. This enables us to prove statements concerning learning algorithms theoretically, such as the convergence of an agent's behavior to its optimum.

We start by defining that the learning and acting instance is called the *agent*, whereas its surroundings, which deliver situations, respond to the agent's actions, and control the reward signal, are called the *environment*. The agent, therefore, strives to maximize environment rewards by performing actions based on previously perceived situations in the environment.

In Markov Decision Processes, the agent-environment interaction only occurs at discrete timesteps $t \in \{0, 1, 2, ...\}$. At each timestep t, the environment provides a *state* $S_t \in \mathcal{S}$ to the agent, where $\mathcal{S}$ is the set of all possible states. Based on this state, the agent must select an *action* $A_t \in \mathcal{A}(S_t)$, with $\mathcal{A}(S_t)$ being the set of legal actions in state S_t. As a result of this action, the agent receives a *reward* $R_t \in \mathcal{R} \subset \mathbb{R}$ and progresses onwards to the next state S_{t+1}. Note that this means there is no reward associated with the very first timestep. A sequence following this pattern of states, actions, and rewards is called a *trajectory.* This process is visualized in Figure 2.

In a *finite Markov Decision Process* [Sutton and Barto, 2018], the sets of states $\mathcal{S}$, actions $\mathcal{A}(s) : \forall s \in \mathcal{S}$, and rewards $\mathcal{R}$ are all finite. With this constraint, we can create a function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ to denote probabilities of state transitions occurring. For example, $p\left(s', r \,\middle|\, s, a\right)$ refers to the probability of seeing state s' and receiving reward r after taking action a in state s. This implies that transition probabilities are only dependent on the current state and action, but not on historical information.

Usually, an agent is not supposed to focus solely on choosing the action A_t which results in the highest *immediate reward* R_{t+1}. Instead, future rewards should also be taken into consideration. Formally, we define the *return* G_t for timestep t, which is the sum of all rewards at subsequent timesteps until the final timestep T.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$$

Accordingly, an agent shall aim to maximize the expected return at each step. The concept of a final timestep only applies to tasks that have a clearly defined beginning and end, making them repetitive in nature. We call a single cycle of these repetitive tasks *episode.*

For tasks that can potentially continue indefinitely, i.e. $T = \infty$, G_t may also become infinite and can, therefore, no longer be subject to maximization. To avoid this issue, we introduce a *discount rate* (or *discount factor*) $\gamma \in [0, 1]$, and

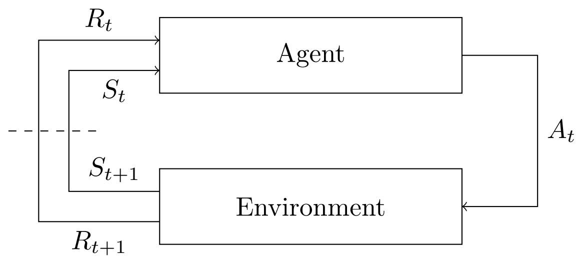


Figure 2: The agent-environment feedback loop in a Markov Decision Process.

define the *discounted return* as follows:

$$G_t^{(\gamma)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

The parameter γ determines to what extend the agent prefers rewards in the near over those in the far future. As an example, an agent with $\gamma = 0$ is only concerned with the immediate reward, whereas the same agent with $\gamma = 0.99$ will optimize for a large time window of rewards. Choosing $\gamma < 1$ ensures that G_t will never become infinite.

In some cases, it is not realistic that the agent receives the full state of the environment. For instance, a robot with a camera cannot possibly measure all physical properties of its surroundings at all times. While a well defined *latent state* may exist, the agent is only provided with an *observation* produced by this state instead. These types of environments are modeled as *Partially Observable Markov Decision Processes*, or *POMDPs* [Sutton and Barto, 2018].

2.1.2 Policy and Value Functions

An agent's behavior can be described stochastically. We define the *policy* of an agent to be the probability distributions for taking each available action in the respective states. For a state $s \in \mathcal{S}$, a legal action $a \in \mathcal{A}(s)$, and a policy π, $\pi\left(a \mid s\right)$ is the probability that the agent will perform a in s. The learning process consists of adapting π over time to maximize the expected return [Sutton and Barto, 2018].

The *value* is another useful measure for a reinforcement learning agent. It describes how favorable a specific state is for the agent when following a given policy. In other words, the value is the expected return for a policy. We define the *state-value function for policy* π [Sutton and Barto, 2018] as

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t | S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \forall s \in \mathcal{S}$$

for Markov Decision Processes.

State-value functions do not apply to cases in which the agent tries to diverge from its current policy π. Because of this, we additionally define the *action-value function for policy* π [Sutton and Barto, 2018], q_π, as the expected return when taking any action $a \in \mathcal{A}(s)$ in state $s \in \mathcal{S}$, and subsequently following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right], \forall s \in \mathcal{S}$$

2.1.3 Policy-Based Methods

We now explain one possible approach to optimize an agent's behavior by directly manipulating its parameterized policy π_θ with regards to the expected returns $\mathbb{E}\left[G_t\right]$. Typically, this is done using gradient descent. The *REINFORCE* algorithm [Williams, 1992] defines an estimate for $\nabla_\theta \mathbb{E}\left[G_t \mid S_t\right]$ as

$$\nabla_\theta \mathbb{E}\left[G_t \mid S_t\right] \approx \nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right) G_t,$$

coining the term *policy gradient*. Intuitively, this may be thought of as increasing or decreasing the probability of actions on a given trajectory depending on the trajectory's return.

Because the returns G_t cannot be determined before a terminal state has been reached, REINFORCE may, by itself, only be used in episodic tasks where training occurs at the end of an episode.

The regular policy gradient is prone to high variance, which can slow down learning. A baseline $b(S_t)$ can be introduced, which is shown to remain unbiased and often improves performance:

$$\nabla_\theta \mathbb{E}\left[G_t \mid S_t\right] \approx \nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right) \big(G_t - b(S_t)\big)$$

2.1.4 Value-Based Methods

In value-based methods, the agent maintains estimates of the state-value or action-value functions to improve its behavior. The key concept of value-based methods is the recursive definition for value functions, known as the *Bellman equation* [Sutton and Barto, 2018]:

$$\begin{aligned}
v_\pi &= \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_t + G_{t+1} \mid S_t = s\right] \\
&= \sum_a \pi\left(a \mid s\right) \sum_{s'} \sum_r p\left(s', r \mid s, a\right) \Big[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \,\big|\, S_{t+1} = s'\right]\Big] \\
&= \sum_a \pi\left(a \mid s\right) \sum_{s'} \sum_r p\left(s', r \mid s, a\right) \Big[r + \gamma v_\pi\left(s'\right)\Big]
\end{aligned}$$

Put into words, the value of a state is the sum of all possible immediate rewards and discounted values of subsequent states, weighted by their occurrence probabilities. A similar definition can be created for action-value functions:

$$q_\pi\left(s, a\right) = \sum_{s'} \sum_r p\left(s', r \mid s, a\right) \left[r + \gamma q_\pi(s', a')\right]$$

The optimal policy π^*, given the correct action-value function q_{π^*}, would always perform the action with the highest q-value, i.e. the largest expected return.

In turn, the action-values created by this policy are always the highest possible expected returns of any action. We find ourselves in a stable condition, known as the *Bellman optimality equation* [Sutton and Barto, 2018], where q_* denotes the action-value function for the optimal policy:

$$q_*(s,a) = \sum_{s'} \sum_{r} p\left(s', r \mid s, a\right) \left[r + \gamma \max_{a'} q_*(s', a')\right]$$

In practice, the transition probabilities p are usually not known. Q-learning [Watkins and Dayan, 1992] approaches this issue by iteratively updating q-value estimates (which we will refer to as Q) in a tabular form, known as the Q-table. The iterative formula

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \beta \left(R_{t+1} + \gamma \max_{a} Q\left(S_{t+1}, A_t\right) - Q\left(S_t, A_t\right)\right),$$

where β is the learning rate hyperparameter, closely resembles the Bellman optimality equation. We are updating the current value estimate from the Q-table entry using the acquired information R_{t+1} as well as a future, *bootstrapped* value estimate $\max_a Q\left(S_{t+1}, A_t\right)$. This form of learning from future estimates is referred to as *temporal-difference learning* or *TD learning* [Sutton, 1988]. The difference between the current estimate and the bootstrapped estimate combined with new experience is called the *TD error*.

Note that Q-learning, similarly to other value-based methods, requires discrete, finite actions, whereas the aforementioned REINFORCE algorithm is capable of handling continuous action spaces. To define a Q-table, the number of possible environment states must also be finite and relatively small, which makes visual tasks virtually impossible. However, unlike REINFORCE, Q-learning is an *online* algorithm [Sutton and Barto, 2018], meaning it can learn from data as soon as it is available (while interacting with the environment), and does not need to wait for the end of an episode. Thus, it can also be used for non-terminating tasks.

2.1.5 Actor-Critic Methods

Actor-critic methods [Sutton and Barto, 2018] try to combine policy- and value-based methods to exploit strengths and mitigate the weaknesses of the two parts. The term is inspired by the way an agent, following these methods, performs actions in the environment using the policy (actor) component and then judges itself using the value (critic) component.

A possible approach is to use the state-value function estimate v as a baseline for the policy gradient:

$$\nabla_\theta \mathbb{E}\left[G_t \mid S_t\right] \approx \nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right) \left(G_t - v_\pi(S_t)\right)$$

We can replace the actual return G_t with the expected return outputted by the action-value function $q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right]$:

$$\nabla_\theta \mathbb{E}\left[G_t \mid S_t\right] \approx \nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right)\big(q(S_t, A_t) - v_\pi(S_t)\big)$$

Note that the term $q(S_t, A_t) - v_\pi(S_t)$ is merely the difference in the value of state S_t when choosing action A_t instead of following policy π_θ. Using only a value estimate V of v_π, this can also be described as $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ [Mnih et al., 2016]. We call this difference the *advantage* of taking action A_t in state S_t. Less formally, the advantage policy gradient can be thought of as increasing the probability of an action whenever the critic is positively surprised by the resulting state, and vice versa.

Updates can be performed online as in Q-learning, and similarly to the default REINFORCE policy gradient, actions must not be discrete. We can see how actor-critics leverage the strengths of both approaches.

2.2 Artificial Neural Networks

Artificial neural networks (*ANNs*), sometimes referred to as *neural networks* (*NNs*), are a category of bio-inspired algorithms that are structured like and can learn similarly to a brain. They are the foundation of many modern machine learning systems. Early concepts have been developed in 1943 [Mcculloch and Pitts, 1943], but were not widely applicable due to the lack of sufficient computational resources. However, today, even consumer-grade graphics cards intended for video games are powerful enough to run complex neural network models and are starting to be engineered specifically to accelerate machine learning applications [Markidis et al., 2018].

Commonly treated as a black box, artificial neural networks can be used as function approximators. Say, for example, we want to predict the output of a function g without knowledge of its inner workings. Instead, we are only given several input-output samples produced by g. In this case, we may use a neural network, which, given a set of parameters θ, also produces an input-output mapping f_θ. This mapping, by default, is unlikely to accurately represent g. We now iteratively adjust θ using the input-output samples created by g so that f_θ also produces a similar output for each given input. The network is then expected to behave much like g, even for previously unknown inputs.

The fundamental unit of a neural network is often the McCulloch-Pitts neuron [Mcculloch and Pitts, 1943]. Inspired by biological neurons, McCulloch-Pitts mathematical neurons receive a vector of real-valued inputs $x = (x_1, ..., x_n)$, which are weighted with parameters $w = (w_1, ..., w_n)$ and then summed to produce a single, real-valued output. This output is then further transformed by an *activation function* h, enabling the neuron to produce non-linear mappings.

A *threshold* term (or *bias*) b is usually introduced as well, forming the equation:

$$y = h\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

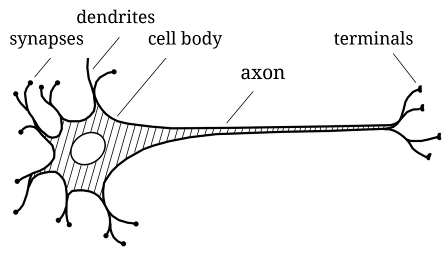
Common activation functions include *sigmoid*, *tanh*, or *ReLU*[1]. The full neuron is visualized in Figure 3. Individual neuron units were later put together to form the *perceptron* [Rosenblatt, 1958]. Multiple layers of neurons are called a *multi-layer perceptron*. Having many of such layers popularized the terms *deep neural network* and *deep learning*. These types of perceptrons can be expressed using a series of matrix and vector multiplications and additions

$$f(x) = h_n\Big(W_n \dots h_2\left(W_2 h_1(W_1 x + b_1) + b_2\right) \dots + b_n\Big),$$

where $W_1, \dots, W_n$ are weight matrices, $b_1, \dots, b_n$ are bias vectors and $h_1, \dots, h_n$ are activation functions for each layer. Note that the input vector x is mapped to another vector $f(x)$, which may have a different number of dimensions.

Perceptrons can be trained using gradient descent of an *error* (or *loss*) term with respect to its parameters θ, which, in the context of neural networks, is referred to as *back-propagation* [Rumelhart et al., 1988]. Calculating gradients for each neuron in a complex model is difficult and must be highly optimized. Programming libraries such as *TensorFlow* [Abadi et al., 2015] or *PyTorch* [Paszke et al., 2019] solve this issue by performing automatic differentiation and optionally parallelizing mathematical operations like matrix multiplications on the GPU.

[1]Rectified Linear Unit



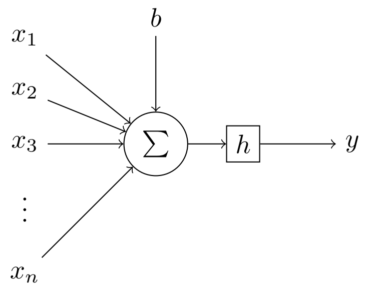


Figure 3: Comparison of a biological neuron [Mehlig, 2019] (left) and a mathematical McCulloch-Pitts neuron (right).

2.3 Deep Reinforcement Learning

The advancements in artificial neural networks and deep learning have also spawned a new era of reinforcement learning algorithms that use neural networks as function approximators.

The first published *Deep Reinforcement Learning* algorithm is known as *Deep Q-learning* (*DQN*) [Mnih et al., 2013], a variation of the Q-learning algorithm that employs an artificial neural network referred to as *Q-network* instead of a Q-table. It also introduces *Experience Replay*, a system in which the past experience of an agent is stored in a *replay buffer* so as to make the training process more sample efficient. DQN achieved state-of-the-art, super-human performance in several video games running on the *Atari 2600 console* while receiving only the very high dimensional visual information from the game screen as observations.

Over the years, several improvements have been made to the original DQN. For instance, *Double DQN* [van Hasselt et al., 2015] uses two neural networks to mitigate DQN's tendency to overestimate action-values. *Prioritized Experience Replay* (*PER*) [Schaul et al., 2016] adjusts the probabilities of experience samples being chosen from the replay buffer based on how good the model already is at correctly estimating the respective sample's values. Thus, an experience sample that is difficult for the neural network to learn is chosen more often for training. The *Rainbow* algorithm [Hessel et al., 2017] combines a variety of individual improvements to DQN and manages to significantly outperform each individual improvement.

Actor-critic architectures have also been enhanced with deep neural networks. Notably, the *asynchronous advantage actor-critic* (*A3C*) [Mnih et al., 2016] achieved state-of-the-art results even compared to DQN and can handle continuous state and action spaces. It works by letting several copies of the agent interact with separate environment instances asynchronously and combining the gathered experience for training. Similarly to DQN, A3C has received several modifications over the years. For example, the *actor-critic with experience replay* (*ACER*) [Wang et al., 2016] improves the sample efficiency of A3C by introducing Experience Replay to the otherwise *on-policy* [Sutton and Barto, 2018] algorithm.

Recently, *OpenAI Five* [OpenAI, 2018] has beaten the world champions at the complex, team-based strategy game *Dota 2*, demonstrating that deep reinforcement learning is even applicable to environments with huge action spaces and very long time horizons. *AlphaStar* [Vinyals et al., 2019] has reached grandmaster level performance in the real-time strategy game *StarCraft 2*, which was said to be a new challenge for reinforcement learning [Vinyals et al., 2017]. This was done by training many different versions of the agent in a league system to make each version robust to different strategies. In the field of robotics, a deep reinforcement learning system has managed to solve a Rubik's Cube[2] using a robotic hand by training in simulations with varying physical attributes to improve robustness of the real world agent, a process called *Automatic Domain Randomization* [OpenAI et al., 2019].

2.4 Model-Based Reinforcement Learning

An *environment model* refers to any means with which an agent is able to predict, to any extent, the behavior of the real environment [Sutton and Barto, 2018]. As opposed to their counterpart, model-free agents, agents belonging to the *model-based reinforcement learning* class can use their knowledge to plan ahead and, for example, avoid catastrophic, irreversible actions.

A simple example of an environment model is a function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{R}$ which maps any state and action to a predicted subsequent state as well as a reward estimate for the transition [Thrun et al., 1991]. Note that, in a stochastic environment, this model cannot always be accurate. Stochastic models are, however, more difficult to create. Given f, an agent could take the current environment state or observation and recursively apply f using different action sequences to find favorable trajectories (see Figure 4).

For some environments, it is possible to provide the agent with a perfect model. As an example, a board game like chess has clearly defined rules which can easily be used for planning. Usually, we are not given such rules, meaning the model must also be learned. Neural networks can be trained to predict environment behavior by providing examples of previously observed transitions. Note that an environment model should be able to predict transitions regardless of which action is chosen, and can, therefore, even be trained using recordings

[2]3D combination puzzle invented by Ernő Rubik

$$\begin{tikzcd}
 & \hat{A}_t \arrow[d] & & \hat{A}_{t+1} \arrow[d] & & & \hat{A}_{t+n-1} \arrow[d] & \\
S_t \arrow[r] & \boxed{f} \arrow[r] \arrow[d] & \hat{S}_{t+1} \arrow[r] & \boxed{f} \arrow[r] \arrow[d] & \hat{S}_{t+2} \arrow[r] & \cdots \arrow[r] & \boxed{f} \arrow[r] \arrow[d] & \hat{S}_{t+n} \\
 & \hat{R}_{t+1} & & \hat{R}_{t+2} & & & \hat{R}_{t+n} &
\end{tikzcd}$$

Figure 4: An environment model $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{R}$ being applied recursively to predict the outcome of a trajectory. S_t is the current environment state, $\hat{A}_t, ..., \hat{A}_{t+n-1}$ is a sequence of actions, and $\hat{S}_{t+1}, ..., \hat{S}_{t+n}$ as well as $\hat{R}_{t+1}, ..., \hat{R}_{t+n}$ are the predicted states and rewards, respectively.

by a random agent. Because every experience sample is valuable to some extent, model-based reinforcement learning algorithms are relatively sample efficient.

A particularly good example of an environment that benefits heavily from planning is the puzzle video game *Sokoban* (see Figure 5), in which the player must push boxes into predefined target positions to solve a level. Since boxes cannot be pulled, some actions, such as pushing a box into a corner, are irreversible and may leave the player in a losing state. Thus, the solution to a Sokoban level should ideally be planned out before making a single move.

2.5 MuZero Algorithm

The MuZero algorithm [Schrittwieser et al., 2019] is a model-based reinforcement learning algorithm that builds upon the success of its predecessor, *AlphaZero* [Silver et al., 2017]. Similar to other model-based algorithms, MuZero can predict the behavior of its environment to plan and choose the most promising action at each timestep to achieve its goal. In contrast to AlphaZero, it does this using a learned model. As such, it can be applied to environments for which the rules are not known ahead of time.

MuZero uses an *internal* (or *embedded*) state representation that is deduced from the environment observation but is not required to have any semantic meaning beyond containing sufficient information to predict rewards and values. Accordingly, it may be infeasible to reconstruct observations from internal states. This gives MuZero an advantage over algorithms akin to what is described in Figure 4. We explain this advantage using an example. Consider a robot agent receiving visual information through a camera as its observations. Predicting the future color of each of the potentially millions of pixels would be unnecessarily difficult and slow, given that we likely only need some key information, such as the position of an object in the environment. Now consider an alternative approach in which we first extract only key information from the observation that

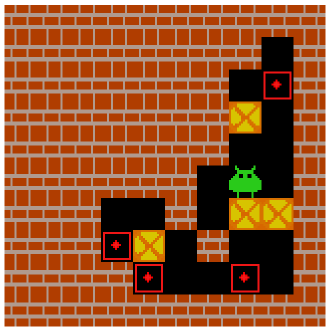


Figure 5: Screenshot of the puzzle video game Sokoban. [Schrader, 2018]

is relevant to our task, and then advance this derived knowledge into the future to make decisions. We can see that the latter approach may be advantageous.

There are three distinct functions (e.g. neural networks) that are used harmoniously for planning. Namely, there is a *representation* function h_θ, a *prediction* function f_θ, as well as a *dynamics* function g_θ, each being parameterized using θ to allow for adjustment through a training process. The complete model is called μ_θ. We will now explain each of the three functions in more detail. Note that we will diverge from our naming conventions to stay consistent with those employed by the MuZero paper.

- The representation function h_θ is a mapping from real observations to internal state representations. For a sequence of recorded observations $o_1, \ldots, o_t$ at timestep t, an embedded representation $s^0 = h(o_1, \ldots, o_t)$ may be produced. As previously mentioned, s^0 has no semantic meaning, and typically contains significantly less information than $o_1, \ldots, o_t$. Thus, the function h_θ is tasked with eliminating unnecessary details from observations, for example by extracting object coordinates and other attributes from images.
- The dynamics function g_θ tries to mimic the environment by advancing an internal state s^{k-1} at a hypothetical timestep $k-1$ based on a chosen action a^k to predict $r^k, s^k = g_\theta\left(s^{k-1}, a^k\right)$, where r^k is an estimate of the real reward u_{t+k} and s^k is the internal state at timestep k. This function can be applied recursively, similarly to what is shown in Figure 4, and acts as the simulating part of the algorithm, estimating what may happen when taking a sequence of actions $a^1, ..., a^k$ in a state s^0.
- The prediction function f_θ can be compared to an actor-critic architecture, having both a policy and a value output. For any internal state s^k, there shall be a mapping $\mathbf{p}^k, v^k = f_\theta(s^k)$, where policy $\mathbf{p}^k$ represents the probabilities with which the agent would perform actions and value v^k is the expected return in state s^k. Whereas the value is very useful to bootstrap future rewards after the final step of planning by the dynamics function, the prediction function's policy may be counterintuitive, keeping in mind that the agent should derive its policy from the considered plans. We will explore the uses of $\mathbf{p}^k$ further on.

Given parameters θ, we can now decide on an action policy π for each observation o_t, which we will call the *search policy*, that uses h_θ, g_θ and f_θ to search through different action sequences and find an optimal plan. As an example, in a small action space, we can iterate through all action sequences $a_1, ..., a_n$ of a fixed length n, and apply each function in the order visualized through Figure 6. By appropriately discounting the reward and value outputs with a discount factor $\gamma \in [0, 1]$, we receive an estimate for the return when first performing the

$$\begin{tikzcd}
\cdots \arrow[r, dashed] & o_{t-1} \arrow[r, dashed] & o_t \arrow[r, dashed] \arrow[d] & o_{t+1} \arrow[r, dashed] & \cdots & & & & \\
 & & \boxed{h_\theta} \arrow[d] & & r^1 & & r^2 & & r^3 \\
 & & s^0 \arrow[r] & \boxed{g_\theta} \arrow[r] \arrow[ur, bend right] & s^1 \arrow[r] & \boxed{g_\theta} \arrow[r] \arrow[ur, bend right] & s^2 \arrow[r] & \boxed{g_\theta} \arrow[r] \arrow[ur, bend right] & s^3 \arrow[d] \\
 & & a^1 \arrow[ur, bend left] & & a^2 \arrow[ur, bend left] & & a^3 \arrow[ur, bend left] & & \boxed{f_\theta} \arrow[d] \\
 & & & & & & & & \mathbf{p}^3, v^3
\end{tikzcd}$$

Figure 6: Testing a single action sequence made up of three actions a^1, a^2, and a^3 to plan based on observation o_t using all three MuZero functions.

action sequence and subsequently following π:

$$\mathbb{E}_\pi \left[u_{t+1} + \gamma u_{t+2} + \ldots \,|\, o_t, a_1, \ldots, a_n \right] = \sum_{k=1}^{n} \gamma^{k-1} r^k + \gamma^n v^n$$

Given the goal of an agent to maximize the return, we may now simply choose the first action of the action sequence with the highest return estimate. Alternatively, a less promising action can be selected to encourage the exploration of new behavior. At timestep t, we call the return estimate for the chosen action produced by our search ν_t.

Training occurs on trajectories previously observed by the MuZero agent. For each timestep t in the trajectory, we unroll our functions K steps through time and adjust θ such that the predictions further match what was already observed in the trajectory. Each reward output r_t^k of g_θ is trained on the real reward u_{t+k} through a loss term l^r. We also apply the prediction function f_θ to each of the internal states $s_t^0, ..., s_t^K$, giving us policy predictions $\mathbf{p}_t^0, ..., \mathbf{p}_t^K$ and value predictions $v_t^0, ..., v_t^K$. Each policy prediction $\mathbf{p}_t^k$ is trained on the stored search policy π_{t+k} with a policy loss l^p. This makes $\mathbf{p}$ an estimate (that is faster to compute) of what our search policy π might be, meaning it can be used as a heuristic. For our value estimates ν, we first calculate n-step bootstrapped returns $z_t = u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n \nu_{t+n}$. With another loss l^v, the target z_{t+k} is set for each value output v_t^k. The three previously mentioned output targets as well as an additional *L2 regularization* term $c||\theta||^2$ [Ng, 2004] form the complete loss function:

$$l_t(\theta) = \sum_{k=0}^{K} \left(l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, \mathbf{p}_t^k) + c||\theta||^2 \right)$$

The aforementioned brute-force search policy is highly unoptimized. For one, we have not described a method of caching and reusing internal states and prediction function outputs so as to reduce the computational footprint. Furthermore, iterating through all possible actions is very slow for a high number of actions or longer action sequences and impossible for continuous action spaces. Ideally, we would want to focus our planning efforts on only those actions that are promising from the beginning and spend less time incorporating clearly unfavorable behavior. AlphaZero and MuZero employ a variation of *Monte-Carlo tree search* (*MCTS*) [Coulom, 2006], which we will elaborate in the following section.

It is advisable to use a replay buffer for MuZero to reach its potential with regards to sample efficiency. That is, we store trajectories gathered through interaction with the environment in a buffer for some time and reuse the same experience samples multiple times during training. Because the losses are calculated based on the respective search policy π and value ν, π and ν must be readily available. Recalculating search results for each training iteration would take a tremendous amount of time and processing power, which is why we instead store them alongside the actual experience after they are determined during environment interaction. This, in turn, means that the stored policies and values, produced by the parameterized model μ_θ, become increasingly outdated as the training progresses and θ is updated. Nevertheless, empirical evidence shows that the algorithm works despite this inaccuracy. However, the problem can be mitigated through an advanced method called *MuZero Reanalyze* [Schrittwieser et al., 2019], in which replay samples are regularly updated by performing the search algorithm with the latest available model parameters. This has been shown to further improve MuZero's performance but comes at the cost of significantly larger computational requirements.

MuZero matches the state-of-the-art results of AlphaZero in the board games Chess and *Shogi*, despite not having access to a perfect environment model. It even exceeded AlphaZero's unprecedented rating in the board game *Go*, while using less computation per node, suggesting that it caches useful information in its internal states to gain a deeper understanding of the environment with each application of the dynamics function. Furthermore, MuZero outperforms humans and previous state-of-the-art agents at the Atari game benchmark, demonstrating its ability to solve tasks with long time horizons that are not turn-based.

2.6 Monte-Carlo Tree Search

We will now explain the tree search algorithm used in MuZero's planning processes, a variant of Monte-Carlo tree search, in detail [Silver et al., 2017, Schrittwieser et al., 2019]. The tree, which is constructed over the course of the algorithm, consists of states, that make up the nodes, and actions, represented by edges. Each path in the tree can be viewed as a trajectory. Our goal is to find the most promising action, that is, the action which yields the highest expected return, starting from the root state. A simple tree is visualized in Figure 7.

Let $S(s,a)$ denote the state we reach when following action a in state s. For each edge, we keep additional data:

- $N(s,a)$ shall store the number of times we have visited action a in state s during the search.
- $Q(s,a)$, similar to the Q-table in Q-learning, represents the action-value of action a in state s.
- $P(s,a) \in [0,1]$ is an action probability, with $\sum_{a \in \mathcal{A}(s)} P(s,a) = 1$. In other words, P defines a probability distribution across all available actions for each state s, i.e. a policy. We will see that these policies are taken from the policy output of the prediction function.
- $R(s,a)$ is the expected reward when taking action a in state s. Again, these values will be taken directly from the model's outputs.

At the start of the algorithm, the tree is created with an initial state s^0, which, in the case of MuZero, can be derived through the use of the representation function on an environment state. The search is then divided into three stages that are repeated for a number of *simulations*.

1. In the *selection* stage, we want to find the part of the tree that is most useful to be expanded next. We want to balance between further advancing already promising trajectories, and those, that have not been explored sufficiently, as they seem unfavorable.

   We traverse the tree, starting from the root node s^0, for $k = 1 \ldots l$ steps, until we reach the currently uninitialized state s^l, which shall become our

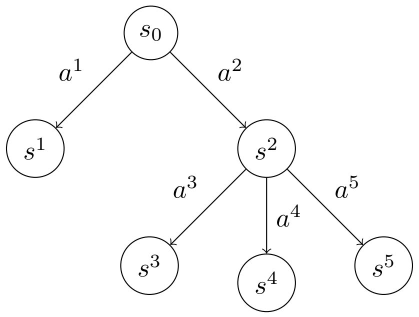


Figure 7: A simplified view of a Monte-Carlo search tree. Note that the number of available actions in state s^0 differs from those in state s^2.

new leaf state. At each step, we follow the edge (or action) that maximizes an upper confidence bound, called pUCT[3]:

$$a^k = \underset{a}{\operatorname{argmax}} \left[Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \cdot \left(c_1 + \log\left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right]$$

The $Q(s,a)$ term prioritizes actions leading to states with a higher value, whereas $P(s,a)$ can be thought of as a heuristic for promising states provided by the model. To encourage more exploration of the state space, these terms are balanced through the constants c_1 and c_2 with the visit counts of the respective edge. An edge that has been visited less frequently therefore receives a higher pUCT value.

For each step $k < l$ of the traversal, we take note of a^k, $s^k = S\left(s^{k-1}, a^k\right)$, and $r^k = R\left(s^{k-1}, a^k\right)$. The entries for $S\left(s^{l-1}, a^l\right)$ and $R\left(s^{l-1}, a^l\right)$ are yet to be initialized.

2. In the *expansion* stage, we attach a new state s^l to the tree. To determine this state, we use the MuZero algorithm's dynamics function $r^l, s^l = g_\theta\left(s^{l-1}, a^l\right)$, advancing the trajectory by a single step, and then store $S\left(s^{l-1}, a^l\right) = s^l$ and $R\left(s^{l-1}, a^l\right) = r^l$. Similarly, we compute $\mathbf{p}^l, v^l = f_\theta\left(s^l\right)$ with the help of the prediction function. For each available subsequent action a in state s^l, we store

$$N\left(s^l, a\right) = 0, \quad Q\left(s^l, a\right) = 0, \quad \text{and} \quad P\left(s^l, a\right) = \mathbf{p}^l(a).$$

This completes the second stage of the simulation.

3. For the final *backup* stage, we update the Q and N values for all edges along our trajectory in reverse order. First, for $k = l \dots 0$, we create bootstrapped return estimates

$$G^k = \sum_{\tau=0}^{l-1-k} \gamma^\tau r_{k+1+\tau} + \gamma^{l-k} v^l$$

for each state in our trajectory. We update the action-values associated with each edge on our trajectory with

$$Q\left(s^{k-1}, a^k\right) \leftarrow \frac{N\left(s^{k-1}, a^k\right) \cdot Q\left(s^{k-1}, a^k\right) + G^k}{N\left(s^{k-1}, a^k\right) + 1},$$

which simply creates a cumulative moving average of the expected returns across simulations. Finally, we update the visit counts of all edges in our path:

$$N\left(s^{k-1}, a^k\right) \leftarrow N\left(s^{k-1}, a^k\right) + 1$$

[3]polynomical Upper Confidence Trees

This completes the three stages of the Monte-Carlo tree search algorithm. However, there is an issue with our pUCT formula. The $P(s,a)$ term should never leave the interval $[0,1]$, whereas $Q(s,a)$ is theoretically unbounded, and depends on the magnitude of environment rewards. This makes the two terms difficult to balance. Intuitively, we are adding up unrelated units of measurement. A simple solution is to divide $Q(s,a)$ by the maximum reward that can be observed in the environment, as a means of normalizing it. Unfortunately, the maximum reward may not be known, and adding prior knowledge for each environment would make MuZero less of a general-purpose algorithm. Instead, we normalize our Q-values dynamically through min-max normalization with other Q-values in the current search tree:

$$\overline{Q}\left(s^{k-1}, a^k\right) = \frac{Q\left(s^{k-1}, a^k\right) - \min_{s,a \in Tree} Q(s,a)}{\max s, a \in Tree Q(s,a) - \min_{s,a \in Tree} Q(s,a)}$$

In our pUCT formula, we may simply replace $Q(s,a)$ with $\overline{Q}(s,a)$.

After the tree has been fully constructed, we may define a policy for the root state as

$$p_a = \frac{N(a)^{1/T}}{\sum_b N(b)^{1/T}},$$

where p_a is the probability of taking action a in state s^0, and T is a temperature parameter further balancing between exploration and exploitation. The search value shall be computed from all action-values $Q(s^0, a)$ based on this policy.

By itself, the Monte-Carlo tree search is only designed for discrete, finite action spaces. This makes MuZero infeasible for, for example, robotics, as joint actuations are commonly treated as continuous actions. However, prior work has shown that MuZero can be made continuous through a strategy named *progressive widening*, in which discrete actions are sampled from the continuous action space and added to the tree over the course of the algorithm [Yang et al., 2020].

3 Proposed Methods

In this section, we propose two changes to the MuZero algorithm that may improve its performance. These changes are based on the idea that MuZero is very unconstrained in its embedded state representations, which, while making the algorithm very flexible, may harm the learning process.

Our changes extend MuZero by individually weightable loss terms, meaning they can be viewed as a generalization of the regular MuZero algorithm and may be used either independently or combined.

3.1 Reconstruction Function

For our first proposed change, we introduce an additional θ-parameterized function, which we call the *reconstruction* function h_θ^{-1}. Whereas the representation function h_θ maps observations to internal states, the reconstruction function shall perform the inverse operation, that is, mapping internal states to real observations, thereby performing a generative task. Notably, since we are using function approximation, reconstructed observations are unlikely to be perfectly accurate. Moreover, in the probable case that the embedded states store less information than the real observations, it becomes theoretically impossible for h_θ^{-1} to restore what has been discarded by the representation function. We call $\hat{o}_t^k = h_\theta^{-1}(s_t^k)$ the reconstructed observation for the embedded state s_t^k, meaning it is an estimate of the real observation o_{t+k} (see Figure 8), given that the action sequences given to the environment and the dynamics function are the same.

The reconstruction function is trained with the help of another loss term l^g added to the default MuZero loss equation

$$l_t(\theta) = \sum_{k=0}^{K} \left(l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, \mathbf{p}_t^k) + l^g(o_{t+k}, \hat{o}_t^k) + c||\theta||^2 \right).$$

This loss term shall be smaller the better the model becomes at estimating observations, for example by determining the mean squared error between the real and the reconstructed observation. Notably, the error gradients must be propagated further through the dynamics function g_θ, and eventually the representation function h_θ. This means h_θ and g_θ are incentivized to maintain information that is useful for observation reconstruction.

Note that, so far, we have not specified any use cases for these reconstructed observations. We do not incorporate observation reconstruction in MuZero's planning process, and, in fact, the reconstruction function h_θ^{-1} may be discarded once the training process is complete. We are only concerned with the effects of the gradients for l^g on the representation function h_θ and dynamics function g_θ.

$$\begin{tikzcd}
\cdots \arrow[r, dashed] & o_t \arrow[r, dashed] \arrow[dd, "h_\theta"'] & o_{t+1} \arrow[r, dashed] & o_{t+2} \arrow[r, dashed] & o_{t+3} \arrow[r, dashed] & \cdots \\
& \hat{o}_t^0 \arrow[u, phantom, "\lightning"] & \hat{o}_t^1 \arrow[u, phantom, "\lightning"] & \hat{o}_t^2 \arrow[u, phantom, "\lightning"] & \hat{o}_t^3 \arrow[u, phantom, "\lightning"] & \\
& s_t^0 \arrow[r, "g_\theta"'] \arrow[u, bend right, "h_\theta^{-1}"'] & s_t^1 \arrow[r, "g_\theta"'] \arrow[u, bend right, "h_\theta^{-1}"'] & s_t^2 \arrow[r, "g_\theta"'] \arrow[u, bend right, "h_\theta^{-1}"'] & s_t^3 \arrow[u, bend right, "h_\theta^{-1}"'] &
\end{tikzcd}$$

Figure 8: The reconstruction function h_θ^{-1} being used to predict future observations o_{t+k} from internal states s^k with the help of the representation function h_θ as well as the dynamics function g_θ.

To understand why, consider a MuZero agent navigating an environment with sparse rewards. It will observe many state transitions based on its actions that could reveal the potentially complex inner workings of the environment. Unless it is actually given rewards, however, it will skip out on gaining any insights, as the model's only goal is reward and value prediction. Even worse, the agent may be subject to *catastrophic forgetting* of what has already been learned, as it is only being trained with a reward target of 0 and small value targets. The reconstruction loss term l^g shall counteract these issues by propagating its error gradients such that the representation function and dynamics function are forced, so to speak, to comprehend the environment beyond the rewards it supplies. Reconstructed observations are not meant to be accurate and their error gradients must not overpower the reward and value prediction. Instead, they should act merely as a guide to stabilize and accelerate learning.

An additional benefit is the ability to pretrain an agent in a self-supervised fashion. That is, the agent can explore an environment without being given any rewards (or any goal) in order to learn about its mechanics and develop a world model. This model can then be specialized to different goals within the same environment. The process is comparable to a child discovering a new task in an environment it is already familiar with, giving it an advantage by not having to learn from scratch.

3.2 Consistency Loss Term

We additionally propose a simple loss term for MuZero's loss equation, which we call the *consistency* loss, and that does not require another function to be introduced into the system. The name originates from the possible inconsistencies in embedded state representations after each application of the dynamics function. The algorithm is completely unconstrained in choosing a different data format, so to speak, for each simulation step k.

Say, as an example, we have two subsequent observations o_t and o_{t+1}, between which action a_t was performed. We can create their corresponding internal state representations with the help of the representation function h_θ:

$$s_t^0 = h_\theta(o_1, ..., o_t), \quad s_{t+1}^0 = h_\theta(o_1, ..., o_t, o_{t+1})$$

By applying the dynamics function g_θ to s_t^0 as well as action a_t, we receive another state representation s_t^1 that is intuitively supposed to reflect the environment at timestep $t+1$, much like s_{t+1}^0. However, so long as both state representations allow for reward and value predictions, MuZero does not require them to match, or even be similar. This pattern persists with every iteration of g_θ, as is apparent in Figure 9. To clarify further, imagine a constructed but theoretically legitimate example in which state vector s_{t+1}^0 uses only the first half of its dimensions, whereas s_t^1 uses only the second half.

While the existence of multiple state formats may not be inherently destructive, and has been suggested to provide some benefits for the agent's understanding of the environment, we believe it can cause several problems. The most obvious of which is the need for the dynamics and prediction functions to learn how to use each of the different representations for the same estimates. If instead, h_θ and g_θ agree on a unified state format, this problem can be avoided. A more subtle but potentially more significant issue becomes apparent when we inspect MuZero's learning process. The functions are unrolled for K timesteps across a given trajectory, and losses are computed for each timestep $k \in \{0, ..., K\}$. This means that g_θ and f_θ are trained on any state format that is produced by g_θ after up to K iterations. For any further iterations, accuracy may degenerate. Depending on the search policy, it is not unlikely for the planning depth to become larger than K. In fact, the hyperparameters used for the original MuZero experiments have a small $K = 5$ unroll depth for training, while at the same time using 50 or even 800 simulations for tree construction during planning, which can easily result in branches that are longer than K. By enforcing a consistent state representation, we may be able to mitigate the degeneration of performance for long planning branches.

We implement our change by continuously adjusting θ so that output s_t^k of the dynamics function is close to output s_{t+k}^0 of the representation function. For

$$\begin{tikzcd}
\cdots \arrow[r, dashed] & o_t \arrow[r, dashed] \arrow[dd, "h_\theta"'] & o_{t+1} \arrow[r, dashed] \arrow[d, bend right, "h_\theta"'] & o_{t+2} \arrow[r, dashed] \arrow[d, bend right, "h_\theta"'] & o_{t+3} \arrow[r, dashed] \arrow[d, bend right, "h_\theta"'] & \cdots \\
 & & s_{t+1}^0 \arrow[d, phantom, "\lightning"] & s_{t+2}^0 \arrow[d, phantom, "\lightning"] & s_{t+3}^0 \arrow[d, phantom, "\lightning"] & \\
 & s_t^0 \arrow[r, "g_\theta"'] & s_t^1 \arrow[r, "g_\theta"'] & s_t^2 \arrow[r, "g_\theta"'] & s_t^3 &
\end{tikzcd}$$

Figure 9: Visualization of the discrepancy between state outputs of the dynamics function g_θ and representation function h_θ.

example, we can perform gradient descent on the mean squared error between the state vectors. Mathematically, we express the consistency loss as $l^c(s^0_{t+k}, s^k_t)$, leading to an overall loss of

$$l_t(\theta) = \sum_{k=0}^{K} \left(l^r(u_{t+k}, r^k_t) + l^v(z_{t+k}, v^k_t) + l^p(\pi_{t+k}, \mathbf{p}^k_t) + l^c(s^0_{t+k}, s^k_t) + c||\theta||^2 \right).$$

Note that we treat the first parameter of l^c, s^0_{t+k}, as a constant, meaning the loss is not propagated directly to the representation function. Doing so would encourage h_θ and g_θ to always produce the same state outputs, regardless of the input. As a bonus, by only propagating the error through s^k_t instead, the model is forced to maintain relevant information to predict subsequent states even in environments with sparse rewards, similar to the reconstruction loss from our previous proposal.

4 Experiments

We will now perform several experiments to test and ideally validate our hypothesis that the proposed methods improve the performance of the MuZero algorithm.

Note that, in artificial intelligence research, training processes are stochastic, meaning we need to repeat experiments many times to get statistically significant results. Furthermore, in reinforcement learning in particular, the algorithms are often fragile and highly sensitive to hyperparameter changes. Errors in the implementation are sometimes difficult to find or even notice, given that their impact on the algorithm's performance may be minor.

4.1 Environments

For our experiments, we choose environments provided by *OpenAI Gym* [Brockman et al., 2016], which is a toolkit for developing and comparing reinforcement learning algorithms. It provides us with a simple and easy to use environment interface and a wide range of environments to develop general-purpose agents.

More specifically, we choose two of the environments shipped with OpenAI Gym, namely *CartPole-v1* and *LunarLander-v2*, shown in Figure 10. These environments are relatively iconic in the reinforcement learning community and, due to their simplicity, the learning progress and pitfalls can be easily understood by a human. A more meaningful benchmark would include significantly more complex environments, such as Chess, Go, and Atari games, as was done for the original MuZero agent. Furthermore, experiments with robot agents, like a robotic arm grasping for objects, could demonstrate real-world viability for the algorithms. Unfortunately, due to MuZero's high computational requirements,

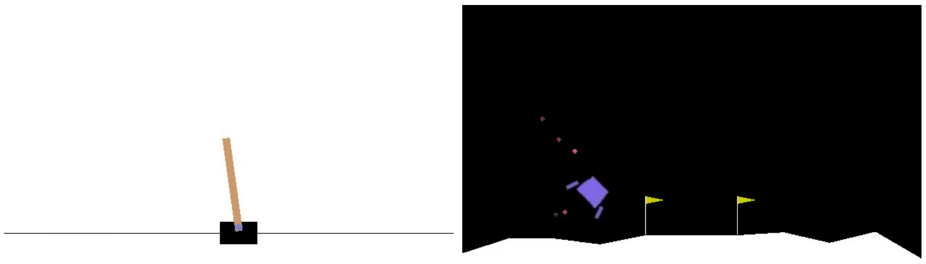


Figure 10: Screenshots of the CartPole-v1 (left) and LunarLander-v2 (right) OpenAI Gym environments [Brockman et al., 2016].

we are unable to properly test these environments with various hyperparameters and achieve sufficient statistical significance. The exploration of more demanding environments is therefore left to future work.

In CartPole-v1, the agent controls a cart that can only move horizontally on one axis. Attached to the cart is a pole that starts off relatively upright and can spin freely around the horizontal axis perpendicular to the cart's movement. The goal is to balance the pole as long as possible so that it never exceeds 15 degrees from being perfectly vertical. Moving the cart too far to the left or right (i.e. out of bounds) also results in episode termination. Only two actions are available, namely the acceleration in either direction, and a reward of 1 is given for each timestep foregoing termination, up to a maximum of 500 steps.

The objective in the LunarLander-v2 environment is, as the name suggests, to safely land a lander on the moon's surface. This is done by controlling three thrusters: The main engine, and two orientation rockets. Various rewards are given to incentivize the agent to pursue this behavior, such as a bonus for approaching the landing pad, and a small negative penalty for using the main engine, encouraging fuel efficiency. Perhaps most notably, however, a large negative reward of -100 is given for crashing the lander.

4.2 Setup

We adopt muzero-general [Werner Duvaud, 2019], an open-source implementation of MuZero that is written in the Python programming language and uses PyTorch for automatic differentiation, as our baselines agent. It is heavily parallelized by employing a worker architecture (see Figure 11), with each worker being a seperate process that communicates with its peers through message passing. This allows for flexible scaling, even across multiple machines. Worker types include

- a *Replay Buffer* worker, responsible for storing game histories as experience for the learning process,
- a *Shared Storage* worker, which holds logging information and distributes neural network parameters,
- at least one, but potentially many *Self Play* processes, each interacting with its own instance of the environment, collecting experience and filling the replay buffer,
- a *Trainer*, which uses stored experience to perform gradient descent on the loss function, thereby improving the model,
- the *Reanalyze* worker, responsible for continuously updating outdated search policies and values within the replay buffer and

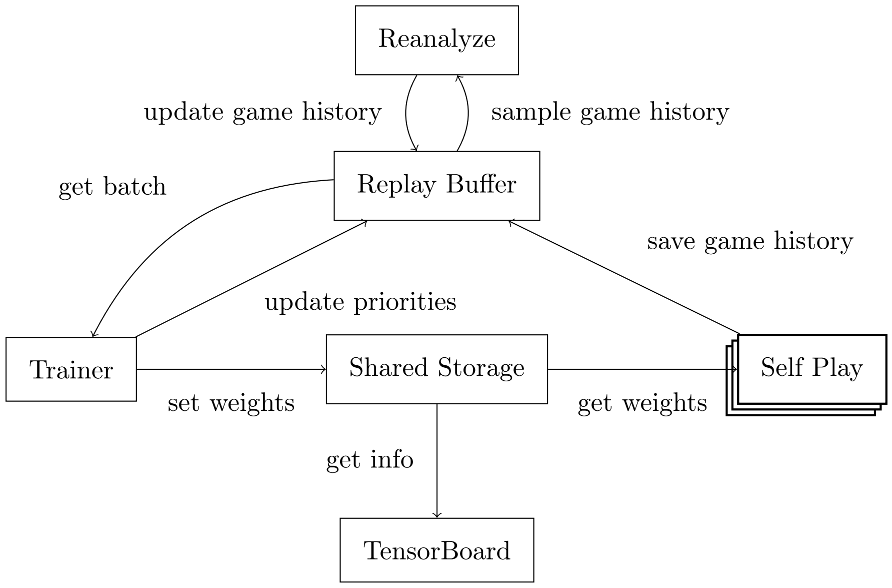


Figure 11: The parallelized worker structure of muzero-general.

- the main process, which takes logging information from the Shared Storage and outputs it to *TensorBoard*, a tool that allows for live visualization of the training process.

Note that the term *self-play* originates from AlphaZero or MuZero agents playing a board game like Chess against themselves to learn strategies without any human influence. Regardless, we use the term even for environments operated by only a single player, such as LunarLander-v2.

We modify the source code of muzero-general to include a reconstruction function and the additional loss terms. The readout of the replay buffer must also be tweaked to include not only the observation at timestep t, but the K subsequent observations $o_{t+1}, ..., o_{t+K}$ as well, as they are critical for calculating our new loss values.

The neural networks making up each of the three MuZero functions and our new reconstruction function follow a very simple structure in all experiments. They consist of simple two-layered fully connected perceptrons. The first (*hidden*) layer contains 16 neurons in the case of CartPole-v1 and 64 for LunarLander-v2. The number of neurons in the second layer is determined by the desired output dimensionality. For example, the representation function must output an internal state, which, for CartPole-v1, is an eight-dimensional vector. Thus, the second layer is made up of eight neurons.

A variety of different weights are tested for each of the two loss terms in order to gauge their capability of improving performance, both individually and in a union. Furthermore, as a means of showcasing self-supervised learning for MuZero, we pretrain a hybrid agent, that is, an agent using both modifications at the same time, for 5000 training steps using only the newly added loss terms instead of the full loss formula.

Table 1 shows the hyperparameters used by all tested model configurations. Based on the muzero-general default parameters, only the number of training steps was reduced to be able to perform additional test runs with the newly freed resources. Note that we are unable to perform a comprehensive hyperparameter search due to technical limitations.

		CartPole-v1	LunarLander-v2
	Training steps	10000*	30000*
	Discount factor (γ)	0.997	0.999
	TD steps (n)	50	30
	Unroll steps (K)	10	10
	Internal state dimensions	8	10
	MuZero Reanalyze	Enabled	Enabled
Losses	Optimizer	Adam	Adam
	Learning rate (β)	$0.02 \times 0.9^{t \times 0.001}$	0.005
	Value loss weight	1.0	1.0
	L2 regularization weight	10^{-4}	10^{-4}
Replay	Replay buffer size	500	2000
	Prioritization exponent	0.5	0.5
	Batch size	128	64
MCTS	Simulations	50	50
	Dirichlet α	0.25	0.25
	Exploration factor	0.25	0.25
	pUCT c_1	1.25	1.25
	pUCT c_2	19652	19652
	Softmax temperature (T)	1.0 if $t < 5000$, 0.5 if $5000 \leq t < 7500$, 0.25 if $t \geq 7500$	0.35

Table 1: Hyperparameter selection for all tested agents. Parameters based on the current training step use the variable t. Only parameters marked with a * are different from the muzero-general defaults.

Performance is measured by training an agent for a specific amount of training steps and, at various time steps, sampling the total episode reward the agent can achieve.

4.3 Results

We show a comparison of the performance of agents with different weights applied to each loss term proposed in this thesis. For our notation, we use l^g and l^c for the reconstruction function and consistency loss modification, respectively. With $\frac{1}{2}l^g$ and $\frac{1}{2}l^g$ we denote that the default loss weight of 1 for each term has been changed to $\frac{1}{2}$. Finally, we write a plus sign to indicate the combination of both modifications.

The results in Figure 12 show an increase in performance when adding the reconstruction function together with its associated loss term to the MuZero algorithm on all testing environments. Weighting the reconstruction loss term with $\frac{1}{2}$ only has a minor negative impact on the learning process. Note that, in the LunarLander-v2 environment, a penalty reward of -100 is given to an agent for crashing the lander. The default MuZero agent was barely able to exceed this threshold, whereas the reconstruction agent achieved positive total rewards.

The consistency loss term agent matched or only very slightly exceeded the performance of MuZero in the CartPole-v1 environment, as can be seen in Figure 13 (left). However, in the LunarLander-v2 task, the modified agent significantly outperformed MuZero, being almost at the same level as the reconstruction agent. A loss weight of 1 is also notably better than a loss weight of $\frac{1}{2}$.

An agent using both loss terms simultaneously outperforms MuZero (visible in Figure 14), and even scores marginally better than the reconstruction loss agent, in all environments tested.

When using self-supervised pretraining (see Figure 15), training progresses very rapidly as soon as the goal is introduced. In the LunarLander-v2 environment, a mean total reward of 0 is reached in roughly half the amount of training steps that are required by the non-pretrained agent. However, at later stages of training, the advantage fades, and, in the case of CartPole-v1, agents using self-supervised pretraining perform significantly worse than agents starting with randomly initialized networks.

The fully trained agents are compared in Table 2. Training took place on NVIDIA GeForce GTX 1080 and NVIDIA GeForce RTX 2080 Ti GPUs. Each experiment required roughly two to three days to complete.

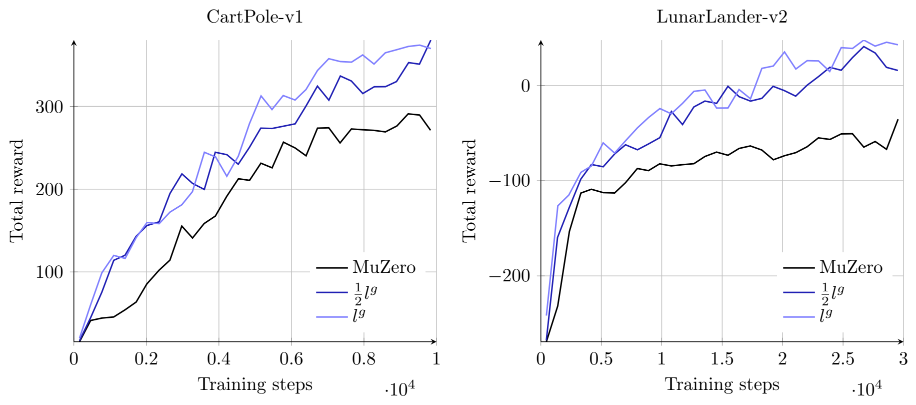


Figure 12: Total episode reward comparison of agents using the reconstruction loss term (l^g) and the default MuZero agent in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

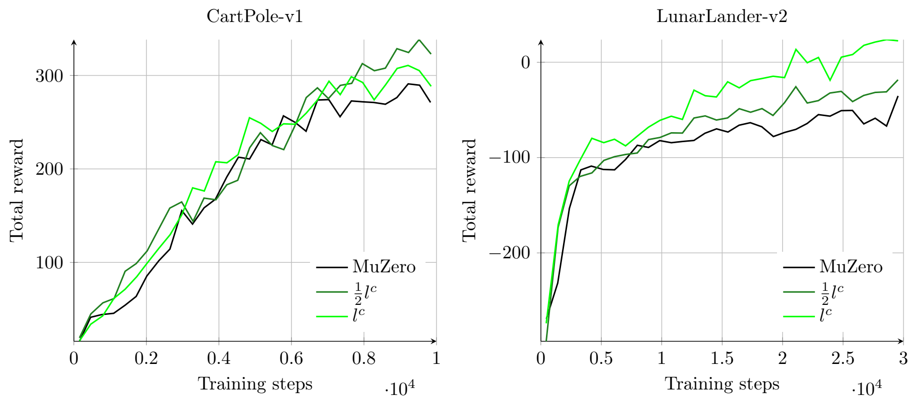


Figure 13: Total episode reward comparison of agents the consistency loss (l^c) and the default MuZero agent in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

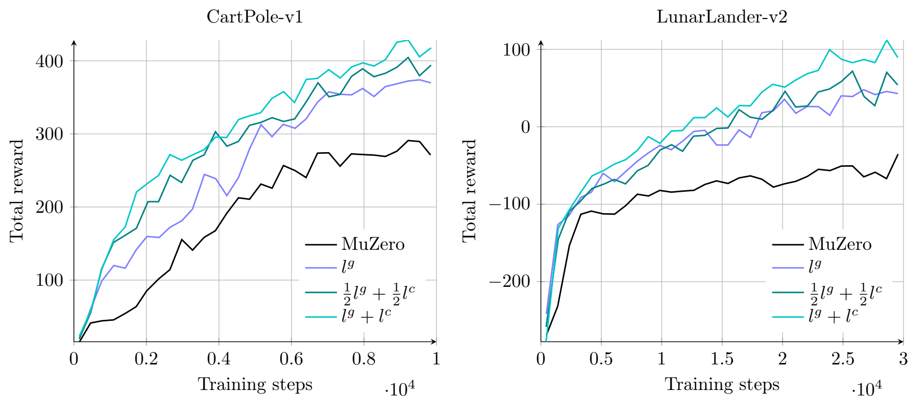


Figure 14: Total episode reward comparison of agents using both the reconstruction function loss (l^g) as well as the consistency loss term (l^c) simultaneously in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

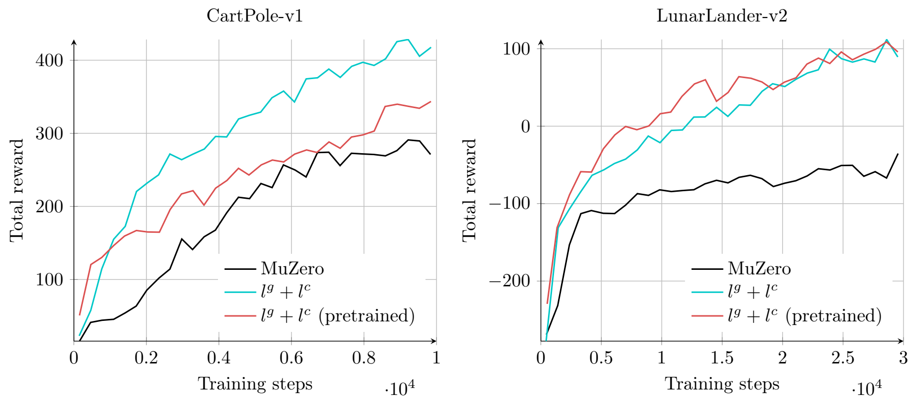


Figure 15: Total episode reward comparison of agents using both the reconstruction function loss (l^g) as well as the consistency loss term (l^c) simultaneously as a pretrained and non-pretrained variant in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

	CartPole-v1	LunarLander-v2
MuZero	281.42 ± 162.48	-34.86 ± 92.87
l^g	375.85 ± 149.38	42.67 ± 132.59
l^c	296.45 ± 174.55	32.17 ± 145.90
$l^g + l^c$	$\mathbf{410.59 \pm 130.71}$	100.46 ± 123.82
$l^g + l^c$ (pretrained)	335.72 ± 162.49	$\mathbf{104.99 \pm 116.67}$

Table 2: Comparison of the default MuZero algorithm and the modifications described in this thesis on the CartPole-v1 and LunarLander-v2 environments. The terms l^g and l^c stand for the addition of a reconstruction or consistency loss, respectively. The results show the mean and standard deviation of the total reward for the final 500 training steps across all test runs.

4.4 Discussion

The results show that all modified agents we tested exceed the performance of the unchanged MuZero agent on all environments, some, especially the agents including both proposed changes simultaneously, by a large amount. The reconstruction function seems to be the more influential of the two changes, given that it achieved a bigger performance increase than the consistency loss, and comes very close to even the hybrid agent. Even so, the consistency loss has the advantage of being simpler in its implementation by not requiring a fourth function, potentially made up of complex neural networks requiring maintenance by experts, to be added.

Agents that were pretrained in a self-supervised fashion ahead of time performed very well in the early stages of training but were unable to outmatch their fully trained counterparts. This is to be expected. While self-supervised pretraining is meant to accelerate learning, there is no argument as to why training with partial information should yield higher maximum scores in the long run. In CartPole-v1, the pretrained agent was even significantly worse than the non-pretrained version after roughly 1000 training steps. This is likely due to the parameters and internal state representations of the model prematurely converging, to some extent, to a local minimum, which made reward and value estimations more difficult after the goal was introduced. The use of L2 regularization during pretraining may prevent this unwanted convergence.

It is still unclear whether the consistency loss provides additional value to the algorithm that is not reproducible by the reconstruction loss. While the excellent performance of the hybrid agent does suggest a benefit, we may see similar results when further increasing the weight of the reconstruction loss, considering that the agent using the highest tested loss weight performed the best. This would render the hybrid architecture unnecessary. Thus, further investigation regarding hyperparameter choices is required.

While the consistency loss resulted in a performance increase, the experiments were not designed to confirm that state representations did indeed become more consistent. Future work should also validate our hypothesis stating that the consistency loss may lessen the accuracy falloff when planning further than K timesteps into the future, with K being the number of unrolled steps during training.

5 Conclusion

We proposed changes to the MuZero Algorithm consisting of two new loss terms to the overall loss function, one of which requires an auxiliary neural network to be introduced into the system. Experiments on simple OpenAI Gym environments have shown that they significantly increase the model's performance, especially when used in combination. We also demonstrated that the changes may be used for unsupervised pretraining of MuZero agents, although caution is advised, as premature convergence may implicate a learning performance reduction.

Due to technical limitations, we were unable to properly investigate the usefulness of our changes in more complex environments, such as Go, the Atari game benchmark, or our initially attempted cube stacking scenario. Furthermore, an extensive hyperparameter search regarding the optimal combination of loss weights is yet to be performed, particularly because our best performing agents were the ones with the highest weight settings. This would also give insight into whether or not one of the changes deprecates the other. We leave the aforementioned topics up to future work.

The full source code used for the experiments will be made publicly available on GitHub at `https://github.com/pikaju/bachelors-thesis`. Note that some environments are still missing their respective hyperparameter configuration.

6 References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *arXiv e-prints*, page arXiv:1606.01540.

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag.

[Hessel et al., 2017] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning. *CoRR*, abs/1710.02298.

[Markidis et al., 2018] Markidis, S., Chien, S. W. D., Laure, E., Peng, I. B., and Vetter, J. S. (2018). NVIDIA Tensor Core Programmability, Performance & Precision. *CoRR*, abs/1803.04014.

[Mcculloch and Pitts, 1943] Mcculloch, W. and Pitts, W. (1943). A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5:127–147.

[Mehlig, 2019] Mehlig, B. (2019). Artificial Neural Networks. *CoRR*, abs/1901.05639.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602.

[Mnih et al., 2016] Mnih, V., Puigdomènech Badia, A., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1602.01783.

[Ng, 2004] Ng, A. Y. (2004). Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 78, New York, NY, USA. Association for Computing Machinery.

[OpenAI, 2018] OpenAI (2018). OpenAI Five. `https://blog.openai.com/openai-five/`.

[OpenAI et al., 2019] OpenAI, Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., Schneider, J., Tezak, N., Tworek, J., Welinder, P., Weng, L., Yuan, Q., Zaremba, W., and Zhang, L. (2019). Solving Rubik's Cube with a Robot Hand. *ArXiv*, abs/1910.07113.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv e-prints*, page arXiv:1912.01703.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

[Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA.

[Schaul et al., 2016] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized Experience Replay. *CoRR*, abs/1511.05952.

[Schrader, 2018] Schrader, M.-P. B. (2018). gym-sokoban. `https://github.com/mpSchrader/gym-sokoban`.

[Schrittwieser et al., 2019] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. (2019). Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv e-prints*, page arXiv:1911.08265.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, abs/1712.01815.

[Sutton, 1988] Sutton, R. (1988). Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3:9–44.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

[Thorndike, 1898] Thorndike, E. (1898). Some experiments on animal intelligence. *Science*, 7(181):818–824.

[Thrun et al., 1991] Thrun, S., Möller, K., and Linden, A. (1991). Planning with an Adaptive World Model. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 450–456. Morgan-Kaufmann.

[van Hasselt et al., 2015] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. *CoRR*, abs/1509.06461.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., and et al. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

[Vinyals et al., 2017] Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J. P., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T. P., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. (2017). Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782.

[Wang et al., 2016] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016). Sample Efficient Actor-Critic with Experience Replay. *CoRR*, abs/1611.01224.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

[Werner Duvaud, 2019] Werner Duvaud, A. H. (2019). MuZero General: Open Reimplementation of MuZero. `https://github.com/werner-duvaud/muzero-general`.

[Williams, 1992] Williams, R. J. (1992). *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning.* Kluwer Academic Publishers.

[Yang et al., 2020] Yang, X., Duvaud, W., and Wei, P. (2020). Continuous Control for Searching and Planning with a Learned Model.

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorthesis im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum   Unterschrift

Erklärung zur Veröffentlichung

Ich stimme der Einstellung der Bachelorthesis in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum	Unterschrift