In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN 2021)*, Shenzhen, China, July 18-22, 2021.

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

Julien Scholz, Cornelius Weber, Muhammad Burhan Hafez and Stefan Wermter
Knowledge Technology, Department of Informatics, University of Hamburg, Germany
julienscholzo@hotmail.de, {weber, hafez, wermter}@informatik.uni-hamburg.de

***Abstract*—Using a model of the environment, reinforcement learning agents can plan their future moves and achieve superhuman 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 derived from real environment states for its predictions. In this paper, we bind the model's predicted internal state representation to the environment state via two additional terms: a reconstruction model loss and a simpler consistency loss, both of which work independently and unsupervised, acting as constraints to stabilize the learning process. Our experiments show that this new integration of reconstruction model loss and simpler consistency loss provide a significant performance increase in OpenAI Gym environments. Our modifications also enable self-supervised pretraining for MuZero, so the algorithm can learn about environment dynamics before a goal is made available.**

I. Introduction

Reinforcement learning algorithms are classified into two groups. Model-free algorithms can be viewed as behaving instinctively, while model-based algorithms can plan their next moves ahead of time. The latter are arguably more human-like and have the added advantage of being comparatively sample-efficient.

The *MuZero Algorithm* [1] outperforms its competitors in a variety of tasks while simultaneously being very sample-efficient. Unlike other recent model-based reinforcement learning approaches that perform gradient-based planning [2]–[6], MuZero does not assume a continuous and differentiable action space. However, we realized that MuZero performed relatively poorly on our experiments in which we trained a robot to grasp objects using a low-dimensional input space and was comparatively slow to operate. Contrary to our expectations, older, model-free algorithms such as *A3C* [7] 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* [8], produced unsatisfactory outcomes.

Since these results did not match those that MuZero achieved in complex board games [1], we are led to believe that MuZero may require extensive training time 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 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.

II. Background

A. MuZero Algorithm

The MuZero algorithm [1] is a model-based reinforcement learning algorithm that builds upon the success of its predecessor, *AlphaZero* [9], which itself is a generalization of the successful *AlphaGo* [10] and *AlphaGo Zero* [11] algorithms to a broader range of challenging board and Atari games. 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 traditional model-based algorithms that need to be able to predict future observations (e.g. noisy visual input).

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 parameters 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.

- The representation function h_θ maps real observations to internal state representations. For a sequence of recorded observations $o_1, \dots, o_t$ at timestep t, an embedded representation $s^0 = h(o_1, \dots, o_t)$ may be produced. As previously mentioned, s^0 has no semantic meaning, and typically contains significantly less information than

© 2021 IEEE. Personal use of this material is permitted.

$o_1, \ldots, o_t$. 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, 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 policy of the prediction function 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 in Fig. 1. By 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 action sequence and subsequently following π:

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

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 the dynamics function 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

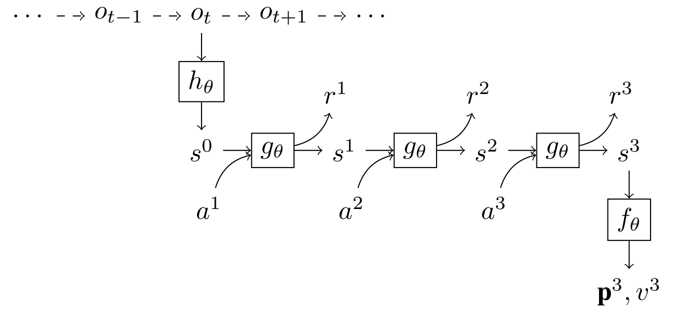

Fig. 1. Testing a single action sequence made up of three actions a^1, a^2, and a^3 to plan based on observation o_t using all three MuZero functions.

$\mathbf{p}$ 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 $c||\theta||^2$ [12] form the complete loss function:

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

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 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*) [13] (see Section II-B).

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.

B. Monte-Carlo Tree Search

We will now explain the tree search algorithm used in MuZero's planning processes, a variant of the Monte-Carlo tree search, in detail [1], [9]. 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.

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 $\sum_{a \in \mathscr{A}(s)} P(s,a) = 1$. 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 new leaf state. At each step, we follow the edge (or action) that maximizes an upper confidence bound called pUCT[1]:

$$a^k = \underset{a}{\operatorname{argmax}} \left[Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \cdot \left(c_1 + \log \left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right] \quad (3)$$

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 $N\left(s^l, a\right) = 0$, $Q\left(s^l, a\right) = 0$, and $P\left(s^l, a\right) = \mathbf{p}^l(a)$. 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 $k = l \ldots 0$, we create bootstrapped return estimates

$$G^k = \sum_{\tau=0}^{l-1-k} \gamma^\tau r_{k+1+\tau} + \gamma^{l-k} v^l \quad (4)$$

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

$$Q\left(s^{k-1}, a^k\right) \leftarrow \frac{N\left(s^{k-1}, a^k\right) \cdot Q\left(s^{k-1}, a^k\right) + G^k}{N\left(s^{k-1}, a^k\right) + 1}, \quad (5)$$

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:

$$N\left(s^{k-1}, a^k\right) \leftarrow N\left(s^{k-1}, a^k\right) + 1 \quad (6)$$

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 T:

$$\overline{Q}\left(s^{k-1}, a^k\right) = \frac{Q\left(s^{k-1}, a^k\right) - \min_{s,a \in T} Q(s,a)}{\max s, a \in T Q(s,a) - \min_{s,a \in T} Q(s,a)} \quad (7)$$

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

$$p_a = \frac{N(a)^{1/T}}{\sum_b N(b)^{1/T}}, \quad (8)$$

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.

[1]polynomical Upper Confidence Trees

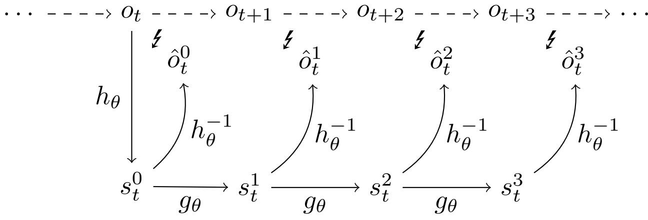

Fig. 2. The new reconstruction function h_θ^{-1} being used to predict future observations o_{t+k} from internal states s^k with the help of the representation function h_θ as well as the dynamics function g_θ.

III. 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 weighable loss terms, meaning they can be viewed as a generalization of the regular MuZero algorithm and may be used either independently or combined.

A. 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. While optimizing reconstruction as an auxiliary task to learn state representations in reinforcement learning has been investigated [14]–[18], reconstructing observations from states sampled from a learned dynamics model has not been considered. 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 Fig. 2), since the dynamics function internally reflects the progression of the environment.

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

$$l_t(\theta) = \sum_{k=0}^{K} \big(l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, \mathbf{p}_t^k) + l^g(o_{t+k}, \hat{o}_t^k) + c||\theta||^2 \big). \quad (9)$$

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. 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. However, it will skip out on gaining any insights unless it is actually given rewards, 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.

B. 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. MuZero is completely unconstrained in choosing a different internal state representation 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 embedded state representations $s_t^0 = h_\theta(o_1, ..., o_t)$ and $s_{t+1}^0 = h_\theta(o_1, ..., o_t, o_{t+1})$. 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

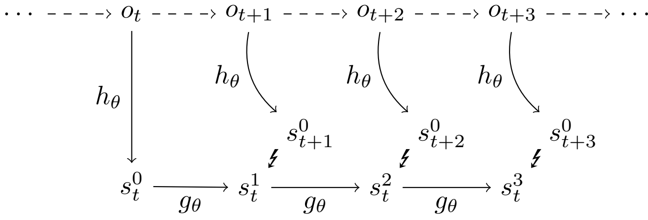

Fig. 3. Visualization of the discrepancy between state outputs of the dynamics function g_θ and representation function h_θ.

iteration of g_θ, as is apparent in Fig. 3. To clarify further, imagine a constructed but theoretically legitimate example in which state vector s^0_{t+1} uses only the first half of its dimensions, whereas s^1_t 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 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^k_t of the dynamics function is close to output s^0_{t+k} of the representation function. For 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

$$l_t(\theta) = \sum_{k=0}^{K} \big(l^r(u_{t+k}, r^k_t) + l^v(z_{t+k}, v^k_t) + l^p(\pi_{t+k}, \mathbf{p}^k_t) + l^c(s^0_{t+k}, s^k_t) + c||\theta||^2 \big). \quad (10)$$

Note that we treat s^0_{t+k}, the first parameter of l^c, 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.

IV. Experiments

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

A. Environments

For our experiments, we choose environments provided by *OpenAI Gym* [19], 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 packaged with OpenAI Gym, namely *CartPole-v1* and *LunarLander-v2*.

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, 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.

B. Setup

We adopt muzero-general [8], an open-source implementation of MuZero that is written in the Python programming language and uses PyTorch for automatic differentiation, as our baseline agent. It is heavily parallelized by employing a worker architecture, with each worker being a separate process that communicates with its peers through message passing. This allows for flexible scaling, even across multiple machines.

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.

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 I shows the hyperparameters used for 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 computational limitations.

TABLE I

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.

	CartPole	LunarLander
Training steps	10000*	30000*
Discount factor (γ)	0.997	0.999
TD steps (n)	50	30
Unroll steps (K)	10	10
State dimensions	8	10
MuZero Reanalyze	Enabled	Enabled
Loss optimizer's parameters (Adam [20])		
Learning rate (β)	$0.02 \times 0.9^{t \times 0.001}$	0.005
Value loss weight	1.0	1.0
L2 reg. weight	10^{-4}	10^{-4}
Replay parameters		
Replay buffer size	500	2000
Prioritization exp.	0.5	0.5
Batch size	128	64
MCTS parameters		
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
Temperature (T)	1.0 if $t < 5000$, 0.5 if $5000 \leq t < 7500$, 0.25 if $t \geq 7500$	0.35

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

C. Results

We show a comparison of the performance of agents with different weights applied to each loss term proposed in this paper. 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 $\frac{1}{2}l^c$ 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 Fig. 4 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 improvement 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 agent with the consistency loss term matched or only very slightly exceeded the performance of MuZero in the CartPole-v1 environment, as can be seen in Fig. 5 (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 $\frac{1}{2}$.

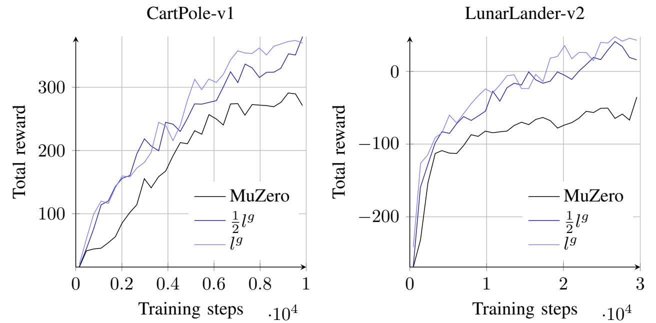

Fig. 4. Total episode reward comparison of agents using the reconstruction loss term (l^g) and the default MuZero agent in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

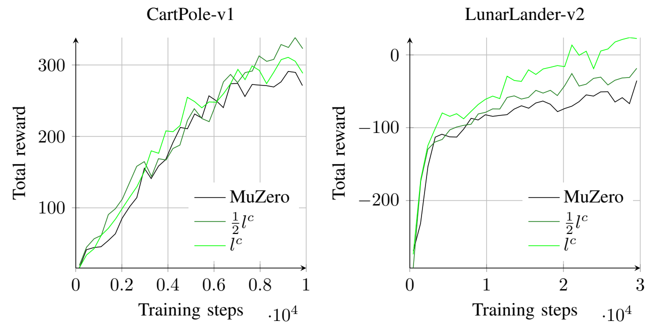

Fig. 5. Total episode reward comparison of agents using the consistency loss (l^c) and the default MuZero agent in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

An agent using both loss terms simultaneously outperforms MuZero (cf. Fig. 6), and even scores marginally better than the reconstruction loss agent, in all environments tested.

When using self-supervised pretraining (see Fig. 7), 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 trained agents are compared in Table II. 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.

D. Discussion

The results show that all modified agents we tested exceed the performance of the unmodified 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 one 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

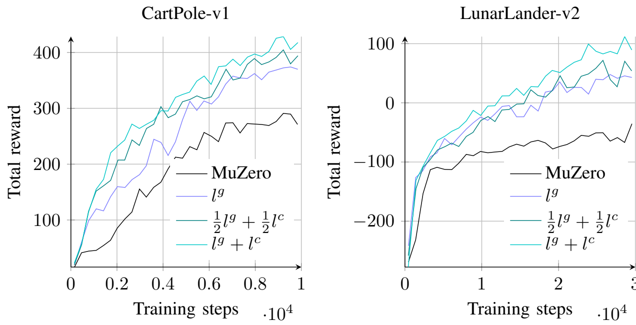

Fig. 6. Total episode reward comparison of agents using both the reconstruction function loss (l^g) as well as the consistency loss term (l^c) simultaneously in the CartPole-v1 and LunarLander-v2 environments, averaged across 32 and 25 runs, respectively.

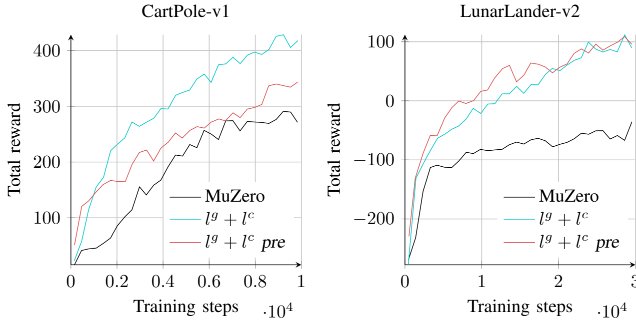

Fig. 7. Total episode reward comparison of agents using both the reconstruction function loss (l^g) as well as the consistency loss term (l^c) simultaneously as a non-pretrained and pretrained (denoted *pre*) variant in the CartPole-v1 and LunarLander-v2 environments averaged across 32 and 25 runs, respectively.

function, potentially made up of complex neural networks requiring domain expertise for their parameterization, to be added.

Agents that were pretrained in a self-supervised fashion ahead of time boosted initial training but were unable to outmatch their non-pretrained 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 even performed 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. Use of L2 regularization during pretraining may prevent this unwanted convergence.

TABLE II

COMPARISON OF THE DEFAULT MUZERO ALGORITHM AND THE MODIFICATIONS DESCRIBED IN THIS PAPER 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, *pre* FOR PRETRAINED. THE RESULTS SHOW THE MEAN AND STANDARD DEVIATION OF THE TOTAL REWARD FOR THE FINAL 500 TRAINING STEPS ACROSS ALL TEST RUNS.

	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
$l^g + l^c$ *pre*	335.72 ± 162.49	$\mathbf{104.99 \pm 116.67}$

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.

Due to computational limitations we did not simulate computationally complex environments such as Go, Atari games, or vision-driven robotics scenarios. 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. We leave these topics to future work.

V. CONCLUSION

We have proposed changes to the MuZero Algorithm consisting of two new loss terms to the overall loss function. One of these requires an auxiliary neural network to be introduced and allows unsupervised pretraining of MuZero agents. Experiments on simple OpenAI Gym environments have shown that they significantly increase the model's performance, especially when used in combination.

The full source code used for the experiments is available on GitHub at https://github.com/pikaju/muzero-g.

ACKNOWLEDGMENT

The authors gratefully acknowledge support from the German Research Foundation DFG under project CML (TRR 169).

REFERENCES

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

[2] A. Srinivas, A. Jabri, P. Abbeel, S. Levine, and C. Finn, "Universal planning networks," in *International Conference on Machine Learning (ICML)*, 2018.

[3] F. Ebert, C. Finn, S. Dasari, A. Xie, A. Lee, and S. Levine, "Visual foresight: Model-based deep reinforcement learning for vision-based robotic control," *arXiv preprint arXiv:1812.00568*, 2018.

[4] N. Wagener, C. an Cheng, J. Sacks, and B. Boots, "An online learning approach to model predictive control," in *Proceedings of Robotics: Science and Systems*, Freiburg im Breisgau, Germany, June 2019.

[5] M. B. Hafez, C. Weber, M. Kerzel, and S. Wermter, "Curious meta-controller: Adaptive alternation between model-based and model-free control in deep reinforcement learning," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[6] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson, "Learning latent dynamics for planning from pixels," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2555–2565.

[7] V. Mnih, A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *arXiv e-prints*, p. arXiv:1602.01783, Feb. 2016.

[8] W. Duvaud and A. Hainaut, "MuZero General: Open Reimplementation of MuZero," https://github.com/werner-duvaud/muzero-general, 2019.

[9] 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. [Online]. Available: http://arxiv.org/abs/1712.01815

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

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

[12] A. Y. Ng, "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance," in *Proceedings of the Twenty-First International Conference on Machine Learning*, ser. ICML '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 78. [Online]. Available: https://doi.org/10.1145/1015330.1015435

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

[14] S. Lange and M. Riedmiller, "Deep auto-encoder neural networks in reinforcement learning," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2010, pp. 1–8.

[15] S. Lange, M. Riedmiller, and A. Voigtländer, "Autonomous reinforcement learning on raw visual input data in a real world application," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2012, pp. 1–8.

[16] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman, "Deep successor reinforcement learning," *arXiv preprint arXiv:1606.02396*, 2016.

[17] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "Integrating state representation learning into deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1394–1401, 2018.

[18] A. Nair, S. Bahl, A. Khazatsky, V. Pong, G. Berseth, and S. Levine, "Contextual imagined goals for self-supervised robotic learning," in *Conference on Robot Learning*. PMLR, 2020, pp. 530–539.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv e-prints*, p. arXiv:1606.01540, Jun. 2016.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.