The impact of model learning losses on the sample efficiency of MuZero in Atari

Daniel Popovici

Supervisor(s): Frans Oliehoek, Jinke He

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Daniel Popovici
Final project course: CSE3000 Research Project
Thesis committee: Frans Oliehoek, Jinke He, Michael Weinmann

Abstract

Recent advances in reinforcement learning (RL) have achieved superhuman performance in various domains but often rely on vast numbers of environment interactions, limiting their practicality in real-world scenarios. MuZero is a RL algorithm that uses Monte Carlo Tree Search with a learned dynamics model, which is trained only to predict rewards, values, and policies, without any explicit objective to match real environment transitions. This work investigates how constraining the learned model of MuZero to follow the real environment dynamics with either a temporal-consistency loss over latent states or a pixel-level observation-reconstruction loss impacts the sample efficiency of MuZero, tested under the Atari100k benchmark. We evaluate performance on Pong, Breakout, and MsPacman analyzing the impact of each loss and its sensitivity to loss weight. Our results show how the temporal-consistency loss can improve performance in certain environments while the observation-reconstruction loss fails to do so, and that both losses are highly sensitive to their weight coefficient, indicating that they might require task-based fine tuning.

1 Introduction

Reinforcement learning (RL) has recently surpassed human performance in domains ranging from Atari games to board games, demonstrating the power of deep neural networks to learn policies directly from high-dimensional inputs. For example, DQN [1] showed that it is possible to learn policies directly from pixel inputs with deep neural networks, paving the way for algorithms such as Rainbow [2], MuZero [3], and DreamerV2 [4] to achieve superhuman play. However, these breakthroughs often rely on hundreds of millions of environment interactions, equivalent to thousands of hours of real-world gameplay [5; 6; 7]. This makes them impractical for domains where each interaction is costly or risky.

To address this sample inefficiency, recent work has explored several complementary approaches: model-based methods (e.g., SimPLe [5], SPR [6]), data augmentation (DrQ [8]), and self-supervised objectives (EfficientZero [7], DreamerV3 [9], BBF [10]).

MuZero [3] is a model-based RL algorithm that builds on top of AlphaZero [11]. Compared to its predecessor, MuZero learns an internal model of environment dynamics, rather than relying on a known simulator. It uses Monte Carlo Tree Search (MCTS) to plan over this learned model, but this model is only meant to be value-equivalent with the real environment[12]; it is only trained to predict the correct value and policy without ever being constrained to learn the environment dynamics faithfully. As a result, the latent model can drift away from real dynamics in low-data regimes, limiting MCTS planning accuracy in low-data scenarios [7].

EfficientZero [7] improves the sample-efficiency of MuZero by adding a few modifications to the base algorithm, including a temporal-consistency loss for the learned model, specifically meant to address the issue of learned model inconsistencies. However, EfficientZero does not test the impact of the temporal-consistency loss in isolation, compare it against alternative model-learning losses, or investigate the sensitivity of performance to the weight of the consistency term. Moreover, other studies have proposed SPR-style consistency, contrastive losses, and autoencoder-based reconstruction objectives for the learned model of MuZero [13; 14; 15], but these have been evaluated mainly for generalization or model interpretability, rather than exploring their performance impact in strict low-data benchmarks.

This leaves an open question: *How do different model-learning losses impact the sample efficiency of MuZero, measured through the scores obtained in Atari games after 100,000 environmental steps?*

We systematically evaluate two approaches: SimSiam-based temporal-consistency loss (a latent state loss) and autoencoder-based observation-reconstruction loss (a pixel-level reconstruction loss) under the Atari100k [5] benchmark. We focus on temporal-consistency because EfficientZero attributes its sample-efficiency gains mainly to it [7], and reconstruction due to its wide use in other model-based RL algorithms [4; 16; 17].

We measure how these losses influence MuZero's performance in Pong, Breakout, and MsPacman when training data is limited, while also examining the sensitivity to loss weights. Our contributions show that temporal-consistency loss improves MuZero's sample efficiency in simple environments, while reconstruction loss can degrade performance, and that performance is sensitive to loss weighting and does not always generalize between tasks.

The rest of the paper is organized as follows: Section 2 reviews background on MuZero and the losses studied. Section 3 surveys related work. Section 4 outlines our methodology, the implemented losses and setup. Section 5 presents experimental results. Section 6 discusses their implications and limitations. Section 7 reflects on responsible research considerations, and Section 8 concludes with future directions.

2 Background

In this section, we provide the necessary technical background and formal definitions that underlie our research. First, we introduce the general concepts of model-based reinforcement learning. Next we describe MuZero, the algorithm studied in the paper. We then review two key concepts: autoencoder-based reconstruction objectives and SimSiam representation learning.

2.1 Model-Based Reinforcement Learning

Reinforcement learning (RL) addresses the problem of how an agent should make decisions while interacting with an environment. The goal is to maximize cumulative reward determined by the agent's actions [18]. The environment is typically modeled as a Markov Decision Process (MDP), defined by a transition function that specifies state changes given actions, and a reward function that assigns scalar values to each transition [19].

RL algorithms are often classified as either *model-free* or *model-based*. In contrast to model-free methods, *model-based* RL involves learning or using an explicit model of the environment's dynamics, which can be queried for arbitrary state–action pairs [20]. Such models can be either provided in advance or learned from interaction data. By incorporating planning into the decision process, model-based methods have demonstrated superhuman performance in complex games such as Go, Chess, Shogi, and Atari, most notably through algorithms like MuZero [3].

2.2 MuZero

MuZero [3] is a model-based RL algorithm that learns its own latent dynamics and uses Monte-Carlo Tree Search (MCTS) for planning. It consists of three learned functions:

1. *Representation* $s_t^0 = h(o_t)$ encodes the current observation o_t into a hidden state s_t^0. [1]
2. *Dynamics* $(r_t^k, s_t^k) = g(s_t^{k-1}, a_t^k)$ predicts the next hidden state s_t^k and immediate r_t^k reward given a hidden state s_t^{k-1} and an action a_t^k.
3. *Prediction* $(p_t^k, v_t^k) = f(s_t^k)$ outputs a policy prior p_t^k and value estimate v_t^k for a hidden state s_t^k.

At each decision step t, MuZero runs Monte Carlo Tree Search (MCTS) on hidden states: it encodes the current observation at the root of the tree using the *representation function*, then it expands nodes using the *dynamics function* and employs the *prediction function* to approximate the value of states in the leaf nodes (Fig. 1a). The next action is picked based on the values obtained through MCTS (Fig. 1b). Compared to traditional MCTS, MuZero replaces random rollouts at leaf nodes with the value estimations, and uses the predicted policy as a prior to guide the search in the tree .

It is important to note that MuZero works on the idea of building a value-equivalent model of the environment [12]. Its internal model is optimized only to make accurate reward, value, and policy predictions, not to learn a representation and dynamics function that are consistent with the true environment dynamics.

Training is end-to-end, optimizing all functions (representation, dynamics, and prediction) together. For each real observation o_t, the latent model is unrolled K steps and compared to MCTS targets (Fig. 1c). The loss aggregates the policy, value and reward errors across all timesteps.

MuZero enhances training with prioritized experience replay and the *reanalyze* procedure, which refreshes MCTS targets (i.e. policy and value targets) by re-planning on past trajectories using updated network parameters.

In the original study by Schrittwieser et al.[3], the baseline MuZero trained for 20 billion steps achieved a mean human-normalized score of 4999.2% on Atari 2600, whereas the reanalyze-augmented variant, reached a score of 2168.9% after being trained for only 200 million steps.

[1] Subscripts like t denote environment timesteps, while superscripts like k denote the learned model rollout steps (if the superscript is omitted or 0, it denotes a state encoded through the represenation function, while a value of 1 or higher denote states unrolled through the dynamics function). We maintain this distinction throughout the paper.

2.3 Autoencoders

Autoencoders (AEs) are a class of unsupervised learning methods for dimensionality reduction and feature extraction [21]. Autoencoders operate by learning to encode an input into a latent space and then reconstruct it. The objective used in training is the similarity between the input and its reconstruction.

Autoencoders typically consist of an encoder and a decoder network. Many specialized types of autoencoders exist depending on their network architecture (i.e. CAEs, VAEs). Given an input, for example an image, it is first processed through the encoder, and then the result is passed through the decoder, with the training objective being to make the output of the decoder similar to the original input. The joint training of the two components ensures the encoder learns to work as a dimensionality reduction operation, learning to encode the most relevant features of the input.

Formally, if x is an input, enc the encoder and dec the decoder, then autoencoders minimize a loss of the form:

$$\mathcal{L} = l_{\text{rec}}(dec(enc(x)), x)$$

where l_{rec} is a loss function such as mean square error (MSE).

Autoencoders have been extensively used in anomaly detection and machine vision to learn visual representations for many tasks, including classification, clustering or image generation [21]. In reinforcement learning autoencoder-like methods have been previously used to learn environment models [4; 16; 17].

2.4 SimSiam

SimSiam is a self-supervised learning method originally developed for learning visual representations without labels [22]. It operates by maximizing the similarity between different augmented views of the same input, encouraging the learned representations to be invariant to data augmentations.

The SimSiam framework consists of an encoder network and a predictor network. The encoder is made of a backbone network (e.g. ResNet) and a MLP projection network. The predictor is also an MLP. Given two differently augmented versions of the same input image, one is processed through the encoder followed by the predictor, while the other is passed through only the encoder and then a stop-gradient is applied (gradients are not back-propagated through the second branch). The objective is to make the output of the predictor match the representation produced by the stop-gradient branch. The use of the predictor and stop-gradient prevents representational collapse, allowing the model to learn meaningful features without contrastive sampling or negative pairs.

Formally, if x_1 and x_2 are two augmentations of the same image, enc the the encoder and $pred$ the predictor. Then, SimSiam minimizes a loss of the form:

$$\mathcal{L} = l_{\text{sim}}\left(pred(enc(x_1)), \text{stopgrad}(enc(x_2))\right)$$

where l_{sim} is a loss such as negative cosine similarity, and `stopgrad` indicates that gradients are not back-propagated through that branch.

SimSiam stands out because it avoids model collapse and achieves strong performance without contrastive sampling,

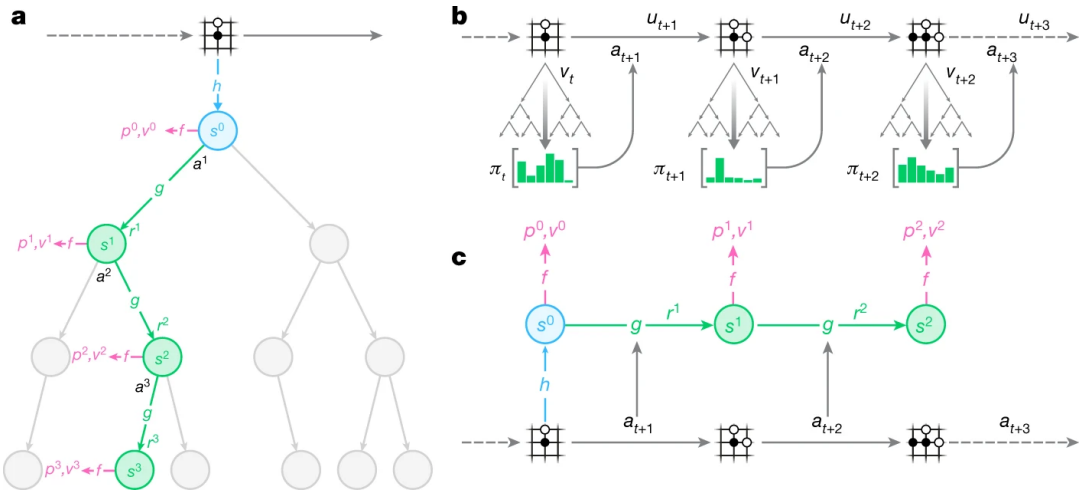


Figure 1: Overview of MuZero's architecture and training process, adapted from [3]. **(a)** MuZero plans with MCTS, using a learned model composed of a representation function h, a dynamics function g, and a prediction function f. At each step, the model receives a hidden state and action to output the next hidden state, reward, value, and policy. **(b)** At each environment step, MuZero performs MCTS as shown in **a** to select an action, which is executed in the environment and a reward u is received. **(c)** During training, past trajectories are unrolled for K steps, the model is trained end-to-end to predict rewards, values, and policies from these rollouts.

negative pairs, momentum encoders, memory banks, or large batch sizes [22]. This has made it attractive for adaptation in other domains beyond image classification, including reinforcement learning [7].

3 Related Work

In this section, we review prior research on data-efficient reinforcement learning, focusing on extensions to MuZero, particularly EfficientZero, that introduce self-supervised consistency losses and other architectural modifications to accelerate learning under strict data budgets.

3.1 Data Efficient Reinforcement Learning

Most state-of-the-art deep RL algorithms require vast amounts of experience, often hundreds of millions of environment steps, to reach strong performance. For example, MuZero Reanalyze is typically trained with 200M+ frames [3] to achieve its top performance. This means thousands of hours of real gameplay.

In contrast, recent work has focused on dramatically reducing data requirements. For instance, Kaiser et al.'s SimPLe [5] algorithm popularized the idea of training agents on around 100,000 environment interactions in Atari games (around 2 hours of play), then many other papers started evaluating their agents on the same benchmark [6; 8; 9; 7; 10]. Furthermore, algorithms such as DreamerV3, BBF and EfficientZero have surpassed human performance on the benchmark [9; 10; 7].

3.2 Model-learning losses for MuZero

Some works have previously augmented MuZero with model-learning losses in order to explore various aspects of the model. Anand et al. have previously explored if augmenting MuZero with SPR-style consistency, reconstruction,, and contrastive losses improves the generalizability of MuZero in various tasks and procedurally generated environments [13]. Others have tested the impact of reconstruction-based model-learning losses on planning [15] or for interpreting the hidden states of MuZero [14].

However, the impact of model-learning losses on the sample efficiency of MuZero remains largely unexplored. The only relevant work on this topic we have found is EfficientZero [7].

3.3 EfficientZero

EfficientZero was proposed specifically to address the poor sample efficiency of MuZero. Ye et al. report how on the full Atari100k benchmark their agent improves the mean human-normalized score of MuZero from 56.2% to 194.3% [7].

EfficientZero largely retains MuZero's architecture but introduces multiple key modifications aimed at improving sample efficiency.

Most importantly, EfficientZero adds a self-supervised temporal-consistency loss. This loss enforces that the hidden states predicted by the dynamics model remain consistent with real environment observations. Ye et al. argue that this internal consistency is critical for sample-efficient learning, as MCTS relies heavily on accurate internal models. To implement this, EfficientZero applies a SimSiam [22] based approach to jointly train the representation and dynamics functions: the stop-gradient branch encodes the actual observation and the other unrolls the learned dynamics model. This encourages the model to learn a representation and dynamics functions that more closely align the hidden states with their real counterparts. Data augmentations are also applied to improve the robustness of learned representations.

Ye et al. report that the temporal-consistency loss contributes the most to EfficientZero's performance gains. Their ablation studies explore the impact of removing one modification at a time. These experiments showed the largest perfor-

mance drop when the consistency loss was omitted. However, since it was not evaluated in isolation and only one weight coefficient was reported, its direct impact is hard to clearly quantify.

4 Methodology

In this paper we aim to answer what is the impact of different model-learning losses on MuZero's sample efficiency. We address two questions: (1) How augmenting MuZero with model-learning losses affects sample efficiency relative to the baseline? and (2) How varying the weight of these losses influences final agent performance. This section presents the model-learning losses explored, the environments used in experiments and the evaluation metrics.

4.1 Model-learning losses

We focus on two model-learning losses: (a) temporal-consistency, an objective that penalizes discrepancies between predicted and actual hidden states, as introduced by EfficientZero [7], and (b) observation-reconstruction, a loss that penalizes pixel-level differences between observations and reconstructions obtained from hidden states. The model-learning losses are only applied during training, the agent's planning remaining unmodified.

The baseline agent implements the MuZero Reanalyze algorithm [3], which implements a value-equivalent loss trained only on value, reward and policy targets. Each model-learning loss is individually applied on top of the baseline, being weighed and added to the full loss. These losses contribute to training the representation and dynamics networks and should lead to state representations and predictions better aligned with the true environment's model. We theorize that this should help the agent plan more effectively over its latent model. This claim is supported by prior work showing how the efficiency and generalizability of planning in latent space benefits from enforcing alignment between latent transitions and true environment dynamics [23].

(a) Temporal-consistency loss. Following EfficientZero [7], we introduce a SimSiam-style loss between the hidden states predicted by the dynamics network and the encodings of the real next observations. To apply this loss we augment the baseline model with two networks: a projector m and a predictor n. The projector maps hidden states to a lower-dimensional latent space where the consistency loss is calculated, while the predictor prevents model collapse [22].

Starting from time t, at each step $1, \dots, k-1, k$ we pass the hidden state s_t^i predicted by unrolling the dynamics network through the projector m and the predictor n to obtain $q_t^i = n(m(s_t^i))$. The target for the stop gradient branch $q_{t+1} = \text{stopgrad}(m(h(o_{t+1}))$ is obtained by passing the real future observations o_{t+1} through the representation network and the projector, with a stop-gradient operation applied. Both branches share the same network parameters for the projector and predictor, but the stop-gradient branch does not propagate gradients during updates. The temporal-consistency loss is calculated as the summation of the negative cosine similarities of q_t^i and q_{t+i} for each timestep:

$$l_{\text{cons}} = -\frac{1}{k}\sum_{i=1}^{k} \frac{q_t^i}{\|q_t^i\|} \cdot \frac{q_{t+i}}{\|q_{t+i}\|}.$$

(b) Observation-reconstruction loss. This loss aims to make the learned dynamics predictive of the real environment by adding a pixel-level reconstruction loss between real observations and reconstructions obtained from the unrolled hidden states. To implement this loss, we augment the baseline agent with a decoder network d, used to reconstruct observation-like tensors from hidden states. The decoder mirrors the representation network h but with deconvolution operations.

Starting from time t, at each step $1, \dots, k-1, k$ we pass the hidden state predicted by unrolling the dynamics network s_t^i through the decoder d to obtain the observation reconstruction $o_t^i = d(s_t^i)$. We calculate this loss as the means-squared error (MSE) of the reconstruction and the real observation o_{t+i} obtained from the environment

$$l_{\text{rec}} = \frac{1}{k}\sum_{i=1}^{k} \| o_{t+i} - o_t^i \|^2.$$

No image augmentations are applied to the observations for any of the model-learning losses. Both losses are computed for five unrolled steps ($k = 5$). Figure 2 illustrates the losses for one unrolled step.

For all variants of the MuZero agent we maintain the same architecture for the representation, dynamics and prediction network. A detailed description of all networks can be found in appendix C. We also keep the model hyper-parameters the same for all models, except the ones related to the additional losses. The hyper-parameters used can be found in appendix B.

4.2 Environments

We follow the Atari100k benchmark, which evaluates agents on Atari 2600 games with a limit of around 100,000 environment interactions, or roughly 2 hours of real gameplay [5].

In our setup, we use: 2,000 interactions for initial data collection with a random policy and 100,000 interactions when training, totaling 102,000 interactions. We use a frame-skip of 4 and a stack of 4 frames per observation.

We focus our experiments on Pong, Breakout, and MsPacman, representative Atari100k games, to compare the effects of different model-learning losses on MuZero's performance.

It is important to note that in Breakout the agent needs to start the level with the "fire" command. If the agent does not learn this it might get stuck in some runs. To avoid this issue the environment is modified to automatically start the level.

4.3 Evaluation Metrics

During training each agent configuration goes through ten evaluation episodes. An evaluation episode has the agent play 50 game rounds and then averages the resulting scores. This averaging over multiple games provides a more stable estimate of each run's performance. The final evaluation metrics

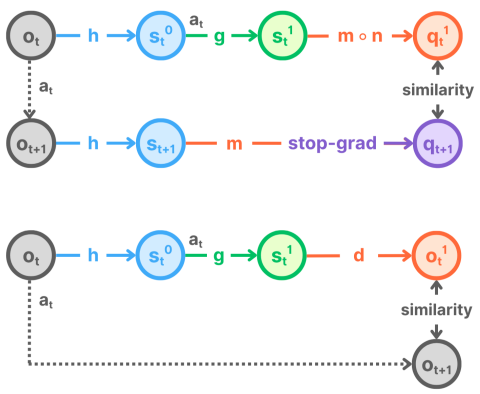


Figure 2: Illustration of the model learning losses for one step. (Top) Temporal-consistency loss: the latent state s_t^1 predicted by the dynamics function is passed through a projection network m and predictor n to produce q_t^1, which is matched via cosine similarity to a stop-gradient target q_{t+1}, derived from the real observation o_{t+1}. (Bottom) Observation-reconstruction loss: the predicted latent s_t^1 is decoded by network d to reconstruct o_t^1, which is compared to the true observation o_{t+1} using mean squared error.

are computed across the scores of all runs testing the same environment-loss configuration.

For easy interpretability and consistency with previous work we report performance by mean evaluation scores with 95% confidence intervals in the main text. Additional metrics, including the median and interquartile mean (IQM), which offers greater robustness in low-data or high-variance settings [24], are presented in Appendix A.1. These other metrics however show the same trends with differences only in absolute values, making the mean illustrative enough for our purposes. We choose not to base our comparisons on statistical significance tests like p-values, as they can be misleading in small-sample experiments and do not indicate whether differences are practically meaningful [24; 25].

5 Results

To evaluate the impact of model learning losses on the sample efficiency of MuZero, we conduct two sets of experiments. First, we isolate the effect of model-learning losses and compare them with the baseline MuZero Reanalyze agent; then, we compare the effect of different weights on each loss.

In all experiments, we use the same training hyperparameters and neural network architecture, except for the differences introduced by the additional loss terms.

We focus most of our resources on Pong in order to enable a thorough investigation of the loss effects, using ten runs for every agent configuration. For the other environments, we use five runs per configuration.

In our evaluation we focus on mean evaluation scores and present plots to aid in the visualization of performance differences. More metrics are presented in Appendix A.1 and tabular summaries of the results in Appendix A.2.

5.1 Effect of model learning loss type

In the first experiment, we aim to isolate the contribution of each model learning loss to sample efficiency. We compare the performance of the baseline agent with the two model-learning loss augmented variants in multiple environments. For this comparison, we fix the loss coefficient for the model learning objectives to a common value of 2 and keep it constant throughout training. This value was chosen to match the setting used in EfficientZero [7], and serves as a reasonable starting point. The second experiment further investigates the effect of this coefficient and, to some extent, validates this choice, showing that a weight of 2 leads to peak or near-peak performance for both model losses in Pong. This setup allows us to directly observe the performance difference attributed to the additional loss signal introduced by each method.

The final evaluation scores are presented in Figure 3. In Pong, the temporal-consistency agent consistently outperforms both the baseline MuZero Reanalyze and the observation-reconstruction variant across all aggregated metrics. Training curves in Figure 4a show the temporal-consistency agent improving more rapidly in early training and maintaining higher scores throughout, while the other two exhibit similar progress and final performance.

In Breakout, temporal-consistency again outperforms both the baseline and observation-reconstruction. While the baseline's mean score is high, its estimates have large confidence intervals due to one outlier run (with a score of 134.6). Observation-reconstruction shows the most consistent results between runs with tight confidence intervals. Training curves in Figure 4b indicate similar progress for the baseline and the temporal-consistency agent for most of the training, before diverging in the last quarter, while the observation-reconstruction stays consistently behind the two.

In MsPacman, the baseline clearly outperforms both temporal-consistency and observation-reconstruction in final evaluation. Training curves in Figure 4c show all agents progressing similarly until late training, where the baseline continues to improve while the other two drop in performance.

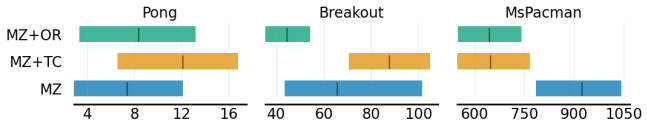
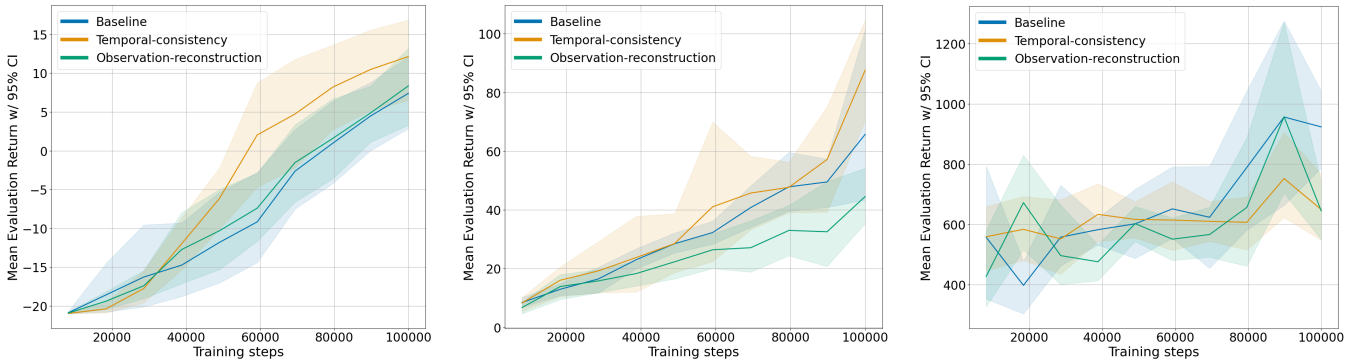


Figure 3: Means of the final evaluation scores for all agent variants across environments. All agent variants are evaluated with a loss coefficient of 2. Mean shown as black lines, with 95% confidence intervals represented as shaded boxes. Pong results are aggregated over 10 runs; Breakout and MsPacman over 5 runs. Notation: MZ = baseline MuZero Reanalyze, TC = temporal-consistency loss, OR = observation-reconstruction loss

These results suggest that augmenting MuZero with a temporal-consistency loss term provides sample-efficiency benefits, but only in a subset of environments, while observation-reconstruction does not lead to improvements and most likely will degrade performance. This indicates that the usefulness of model-learning objectives is highly environment-dependent and might not generalize to all tasks.



(a) Pong (10 runs) (b) Breakout (5 runs) (c) MsPacman (5 runs)

Figure 4: Training curves of agent variants across environments. All agent variants are evaluated with a loss coefficient of 2. The mean is plotted with 95% confidence intervals shown as shaded regions. Pong results are averaged over 10 runs; Breakout and MsPacman over 5 runs.

Furthermore, the effectiveness of the losses is most likely related to the complexity of the environment's visuals and dynamics, since in MsPacman, the more complex environment tested, the model-loss agents underperformed significantly.

5.2 Effect of model learning loss weight

In the second experiment, we investigate how sensitive the performance of the modified agents is to the choice of the model learning loss coefficient. We train the consistency-loss and reconstruction-loss variants of MuZero using a range of coefficient values and record their achieved scores in Pong. We selected the loss coefficient values on a logarithmic scale, doubling at each step from 0.5 up to 16, in order to explore a wide range. Then we test if the performance of certain weight coefficients in one environment generalizes to others.

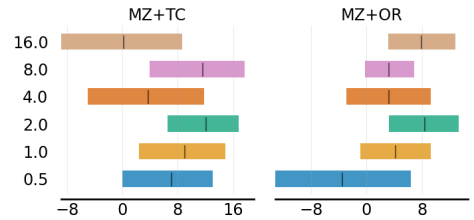
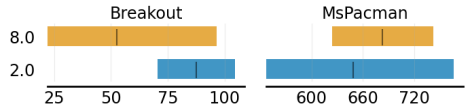


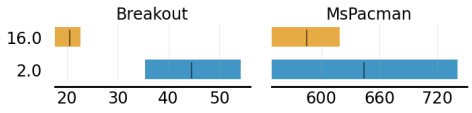
Figure 5: Means of the final evaluation scores per model-learning loss weight coefficient in **Pong** for the model-loss augmented agents. Means are shown as black lines, with 95% confidence intervals represented as shaded boxes. Results obtained from 10 runs per coefficient. Notation: MZ = MuZero Reanalyze, TC = temporal-consistency loss, OR = observation-reconstruction loss

As shown in Figure 5, varying the weight of the model-learning loss produces irregular changes in both performance and confidence. The mean returns show two local performance peaks at coefficients 2.0 for both loss variants and at 8.0 or 16.0 for the temporal-consistency and observation-reconstruction loss respectively. Confidence intervals widen substantially with larger coefficients for temporal-consistency, indicating less consistent outcomes, except at 8.0, where the confidence is comparable to the lower-weight settings. However, the reconstruction-based agent generally produces tighter confidence intervals across most coefficients, suggesting more stable estimates, with the exception of the coefficient 0.5 where the agent shows high uncertainty of the estimated performance.

The multiple peaks in performance seen for both loss variants indicate a non-monotonic relationship with their weights. For both variants, we observe performance peaks at intermediate and higher weights (e.g., 2.0 and 8.0/16.0), indicating that moderate to strong weightings can be beneficial, though not reliably so. The performance dips between peaks suggest that the model-learning losses interact in complex ways with other components of the algorithm. These findings underscore the need for careful tuning, as even small adjustments in weight can produce significantly different results. Furthermore, the sensitivity and instability observed in Pong motivate the need to validate whether these weight settings generalize to other environments.



(a) Temporal-consistency loss



(b) Observation-reconstruction loss

Figure 6: Means of the final evaluation scores per model-learning loss weight coefficient in **Breakout** and **MsPacman** for the two augmented MuZero variants. Means are shown as black lines with 95% confidence intervals as shaded boxes. Results obtained from 5 runs per coefficient.

Building on the findings from Pong, we investigate whether the most effective model-loss weightings transfer to other environments. We evaluate the augmented agents in Breakout and MsPacman using the best-performing coefficients from the Pong experiments (2.0 and 8.0 or 16.0). As shown in

Figure 6, increasing the weight coefficient in Breakout leads to diminishing evaluation scores for both augmented agents. For the observation-reconstruction variant, higher weights tighten the confidence intervals, suggesting more consistent, but worse, performance, while the temporal-consistency variant shows inflated intervals, indicating instability. In MsPacman, the temporal-consistency agent performs better with the higher weight, while the observation-reconstruction does not. Both losses show tighter confidence intervals with higher weights, indicating less variability between runs. Nonetheless, even while the temporal-consistency shows improvements, it still falls behind the baseline in MsPacman. These results indicate that the impact of loss weighting is environment-dependent and suggest the need for task-specific tuning.

6 Discussion

Our results show that the impact of augmenting MuZero with model-learning losses is highly dependent on the environment used. We saw the best performance compared to the baseline in Pong, arguably the simplest environment used in regards to game dynamics and visual complexity, the agent having to predict only the dynamics of the ball and the opponent. The worst performance was seen in MsPacman, the most complex one where the agent needs to predict the results of actions, accounting both for the maze walls and multiple opponents. We theorize that this relates to model-loss efficiency in low-data settings. Reconstruction likely underperformed temporal-consistency because it is a more difficult objective, optimizing a more restrictive loss on high-dimensional data. Even in Pong, which has minimal visual complexity, the final reconstructions still show visible artifacts on dynamic elements like the ball and paddle, indicating that the model struggles to learn accurate pixel-level predictions even under favorable conditions (Figure 7) In contrast, consistency losses seem more robust with limited data, being able to bring performance improvements on two of the three tested environments. This observation aligns with prior work suggesting that training the model of MBRL algorithms with latent-state level losses improves planning more efficiently than pixel-level reconstruction objectives [23].

Furthemore we have seen how both model-learning losses perform in unpredictable ways when their weighting is changed, and how the optimal weights found for a game might not generalize to other Atari games. Going beyond Atari, the coefficients found, or even the model-learning losses tried, might not be suitable for other environments, board games or real-world applications. Furthermore, we cannot conclude that the better coefficients found generalize for bigger data-budgets (e.g. 200 million steps).

During our experiments we have observed big discrepancies between the performance of our baseline MuZero Reanalyze agent the results reported in other works. For example EfficientZero, reports a mean return of -6.7 for their baseline MuZero [7] while ours achieves 7.4, a difference approximately equal to a third of the total range of possible scores. Both their agents and ours use mostly the same network architecture but a few implementation differences, such as the normalization and optimizer used. This significant difference in the performance of the baseline MuZero agents suggests that a lot of factors that influence the performance of MuZero in low-data scenarios still remain underexplored.

Furthermore, EfficientZero does not report scores for the agent augmented only with temporal-consistency, reporting only ablations where they remove one modification at a time [7]. This makes it hard to accurately compare the evaluation scores achieved in our experiments with theirs directly. However, we observed behaviors that stay consistent with their reports: temporal-consistency improves the evaluated scores in Pong and Breakout but performs worse in MsPacman, where their agent also failed to meaningfully improve compared to the baseline.

It's also worth noting that many recent papers using the Atari100k benchmark report results using only three runs per experiment [7; 26; 27; 28]. Agarwal et al. highlight issues with drawing conclusions based on point estimates from a limited number of runs, and that around 20–30 runs are needed for reliable CIs on single tasks in the Atari100k benchmark [24]. Furthermore, Mathieu et al. propose methods for comparing RL algorithm performance that automatically adapt the number of runs needed [29], but due to the setting in which we ran the experiments (i.e. on the available HPC cluster every run needed a clear estimate of runtime) we could not use their methods. We also noticed high variance in performance between training runs with all agent variants in our tests. Due to computational constraints, we used 5–10 runs, aiming for the best trade-off between statistical rigor and resource limitations.

Limitations: The main limitations of this research are due to the high computational cost of training the models and the limited resources available. Local development was constrained, and access to high-performance computing clusters involved scheduling delays, especially during peak usage. This technical limitation narrowed the scope of the research.

We only conducted tests on three environments: Pong, Breakout and MsPacman. Moreover, only two model-loss augmentations were explored despite others existing (e.g. contrastive losses), a limited range of weight coefficients was tested, the number of runs was limited, image augmentations and different values for the number of future steps on which the model-losses are calculated were not explored. Broader testing across more model configurations and environments is necessary to provide stronger conclusions and validate the generalizability of the results.

7 Responsible Research

This section addresses the ethical and practical aspects of our research, focusing on the reproducibility of methods, ethical considerations, environmental impact, and the acknowledgment of AI tools usage.

This research builds on a JAX[30]-based implementation of MuZero maintained by the supervisor team. The code can be accessed at https://gitlab.tudelft.nl/jinkehe/bachelor-research-project/-/tree/daniel_rp_final?ref_type=heads. All experiments are conducted in a controlled software environment, with consistent

Figure 7: Reconstructed observations from the observation-reconstruction loss MuZero agent trained on Pong with a weight coefficient of 2. Predictions of observations one step in the future are sampled from early (left), mid (center), and late (right) stages of training. Early in training, the model primarily reconstructs static scene elements such as the score and boundary lines. As training progresses, it increasingly captures dynamic game elements, with clearer representations of the paddles and ball emerging toward the end.

training configurations and different random seeds between runs. Nonetheless, full reproducibility remains a challenge, particularly because of limited access to consistent hardware configuration.

It was not possible to ensure the same hardware configuration for all experiments. Experiments were run on either Nvidia A40, A100 or V100 gpus, depending on availability. The same code (even using set random seeds) has no reproducibility guarantees if run on different generations of Nvidia gpus [31]. As a result fully reproducing the same exact values might not be possible. To mitigate this, the results reported are aggregated over multiple independent training and evaluation runs and 95% confidence intervals are presented. Detailed logging of training parameters, environment configurations, and network architectures is also performed to support transparency and replicability (see Appendix B and C).

In this research,, we investigate modifications to the loss function of MuZero, evaluated exclusively on Atari games within controlled simulations. The research does not involve human participants, personal data, private or public datasets or deployment in real-world applications. However, it's important to acknowledge that reinforcement learning algorithms have potential applications in domains with ethical implications, such as autonomous systems and decision-making processes [32; 33].

Considering the extensive use of GPUs for training, we also note environmental concerns. In recent years, both the energy consumption and the environmental impact of hardware production related to ML training have increased, despite mitigation strategies being applied [34]. Our models require between 3 to 16 hours for one training and evaluation, depending on the loss configuration and GPU model used. This means that even limited scale research such as ours requires hundreds of GPU runtime hours. As models become more complex and resource-intensive, researching better strategies for mitigating their environmental cost should become a priority.

The use of Large Language Models (LLMs) should be acknowledged as part of responsible research. In writing this paper, LLMs were used to assist with spell checking, assessing the quality of writing, and improving LaTeXformatting (e.g., creating tables, organizing figures). Use for code was limited to debugging and creating plots for figures. We thoroughly verified all AI-generated content and avoided AI use in areas critical to the research.

8 Conclusions and Future Work

In this work, we investigated how augmenting MuZero with different model-learning losses affects its sample efficiency under the Atari100k benchmark. Our evaluation on Pong, Breakout and MsPacman demonstrates that incorporating a temporal-consistency loss can improve the low-data performance of MuZero in certain environments. In contrast, using the observation-reconstruction loss does not provide any improvements and can even degrade performance. We also observed that the effectiveness of model-learning losses is highly sensitive to their weighting, showing a complex and unpredictable behavior under different coefficients and between environments. Finally, we noted considerable variance in the baseline results and discrepancies when compared to previously reported scores, underscoring the substantial influence of implementation details, model architecture, optimization settings and random initialization on performance.

Looking ahead, there are several promising directions to build on these insights. A direct continuation of this research is to explore additional self-supervised objectives such as contrastive losses. It will also be valuable to experiment with hybrid loss formulations or variable losses during training. Furthermore, the impact of image augmentations on model performance remains unexplored, but we theorize they might have a significant role on the performance of model-losses for MuZero. Moreover, the high differences between the performance of our baseline agent and results reported in related literature highlight the need for a systematic study on the impact of model architecture, training details, and hyper-parameters on MuZero's efficiency in low data scenarios. Finally, extending this evaluation beyond Pong and Breakout to a broader suite of Atari titles and other environments will clarify the generality of the results presented in this research.

In conclusion, our work identifies the strengths and limitations of model-learning losses for MuZero in low-data settings, but further research is still needed to generalize these findings and explore ideas that were outside the scope of this paper.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," Dec. 2013. arXiv:1312.5602 [cs].

[2] M. Hessel, J. Modayil, H. v. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," Oct. 2017. arXiv:1710.02298 [cs].

[3] 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," *Nature*, vol. 588, pp. 604–609, Dec. 2020. arXiv:1911.08265 [cs].

[4] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering Atari with Discrete World Models," Feb. 2022. arXiv:2010.02193 [cs].

[5] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, A. Mohiuddin, R. Sepassi, G. Tucker, and H. Michalewski, "Model-Based Reinforcement Learning for Atari," Apr. 2024. arXiv:1903.00374 [cs].

[6] M. Schwarzer, A. Anand, R. Goel, R. D. Hjelm, A. Courville, and P. Bachman, "Data-Efficient Reinforcement Learning with Self-Predictive Representations," May 2021. arXiv:2007.05929 [cs].

[7] W. Ye, S. Liu, T. Kurutach, P. Abbeel, and Y. Gao, "Mastering Atari Games with Limited Data," Dec. 2021. arXiv:2111.00210 [cs].

[8] I. Kostrikov, D. Yarats, and R. Fergus, "Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels," Mar. 2021. arXiv:2004.13649 [cs].

[9] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, "Mastering Diverse Domains through World Models," Apr. 2024. arXiv:2301.04104 [cs].

[10] M. Schwarzer, J. Obando-Ceron, A. Courville, M. Bellemare, R. Agarwal, and P. S. Castro, "Bigger, Better, Faster: Human-level Atari with human-level efficiency," Nov. 2023. arXiv:2305.19452 [cs] version: 3.

[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," Dec. 2017. arXiv:1712.01815 [cs].

[12] C. Grimm, A. Barreto, S. Singh, and D. Silver, "The Value Equivalence Principle for Model-Based Reinforcement Learning," Nov. 2020. arXiv:2011.03506 [cs].

[13] A. Anand, J. Walker, Y. Li, E. Vértes, J. Schrittwieser, S. Ozair, T. Weber, and J. B. Hamrick, "Procedural Generalization by Planning with Self-Supervised World Models," Nov. 2021. arXiv:2111.01587 [cs].

[14] H. Guei, Y.-R. Ju, W.-Y. Chen, and T.-R. Wu, "Interpreting the Learned Model in MuZero Planning," Nov. 2024. arXiv:2411.04580 [cs].

[15] J. B. Hamrick, A. L. Friesen, F. Behbahani, A. Guez, F. Viola, S. Witherspoon, T. Anthony, L. Buesing, P. Veličković, and T. Weber, "On the role of planning in model-based deep reinforcement learning," Mar. 2021. arXiv:2011.04021 [cs].

[16] D. Ha and J. Schmidhuber, "World Models," Mar. 2018. arXiv:1803.10122 [cs].

[17] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to Control: Learning Behaviors by Latent Imagination," Mar. 2020. arXiv:1912.01603 [cs].

[18] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction,"

[19] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition ed., 1994.

[20] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, "Model-based Reinforcement Learning: A Survey," Mar. 2022. arXiv:2006.16712 [cs].

[21] K. Berahmand, F. Daneshfar, E. S. Salehi, Y. Li, and Y. Xu, "Autoencoders and their applications in machine learning: a survey," *Artificial Intelligence Review*, vol. 57, p. 28, Feb. 2024.

[22] X. Chen and K. He, "Exploring Simple Siamese Representation Learning," Nov. 2020. arXiv:2011.10566 [cs].

[23] E. van der Pol, T. Kipf, F. A. Oliehoek, and M. Welling, "Plannable approximations to mdp homomorphisms: Equivariance under actions," in *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020)* (B. An, A. E. F. Seghrouchni, and G. Sukthankar, eds.), vol. 2020-May of *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, pp. 1431–1439, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2020.

[24] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, and M. G. Bellemare, "Deep Reinforcement Learning at the Edge of the Statistical Precipice," Jan. 2022. arXiv:2108.13264 [cs].

[25] S. Greenland, S. J. Senn, K. J. Rothman, J. B. Carlin, C. Poole, S. N. Goodman, and D. G. Altman, "Statistical tests, P values, confidence intervals, and power: a guide to misinterpretations," *European Journal of Epidemiology*, vol. 31, pp. 337–350, Apr. 2016. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 4 Publisher: Springer Netherlands.

[26] S. Hansen, W. Dabney, A. Barreto, T. V. d. Wiele, D. Warde-Farley, and V. Mnih, "Fast Task Inference with Variational Intrinsic Successor Features," Jan. 2020. arXiv:1906.05030 [cs].

[27] R. Saphal, B. Ravindran, D. Mudigere, S. Avancha, and B. Kaul, "SEERL: Sample Efficient Ensemble Reinforcement Learning," May 2021. arXiv:2001.05209 [cs].

[28] T. Kulkarni, A. Gupta, C. Ionescu, S. Borgeaud, M. Reynolds, A. Zisserman, and V. Mnih, "Unsupervised Learning of Object Keypoints for Perception and Control," Nov. 2019. arXiv:1906.11883 [cs].

[29] T. Mathieu, R. D. Vecchia, A. Shilova, M. M. Centa, H. Kohler, O.-A. Maillard, and P. Preux, "AdaStop: adaptive statistical testing for sound comparisons of Deep RL agents," Dec. 2024. arXiv:2306.10882 [cs].

[30] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[31] "Odds and Ends — NVIDIA cuDNN Backend." https://docs.nvidia.com/deeplearning/cudnn/backend/latest/developer/misc.html.

[32] S. Govinda, B. Brik, and S. Harous, "A Survey on Deep Reinforcement Learning Applications in Autonomous Systems: Applications, Open Challenges, and Future Directions," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–26, 2025.

[33] V. Singh, S.-S. Chen, M. Singhania, B. Nanavati, A. K. Kar, and A. Gupta, "How are reinforcement learning and deep learning algorithms used for big data based decision making in financial industries–A review and research agenda," *International Journal of Information Management Data Insights*, vol. 2, p. 100094, Nov. 2022.

[34] C. Morand, A.-L. Ligozat, and A. Névéol, "How Green Can AI Be? A Study of Trends in Machine Learning Environmental Impacts," Dec. 2024. arXiv:2412.17376 [cs].

[35] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, pp. 50–60, Mar. 1947. Publisher: Institute of Mathematical Statistics.

[36] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," Feb. 2016. arXiv:1511.05952 [cs].

[37] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee, "Flax: A neural network library and ecosystem for JAX," 2024.

[38] Delft AI Cluster (DAIC), "The delft ai cluster (daic)," 2024.

[39] Delft High Performance Computing Centre (DHPC), "DelftBlue Supercomputer (Phase 2)." https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

Appendix

A Extended Results

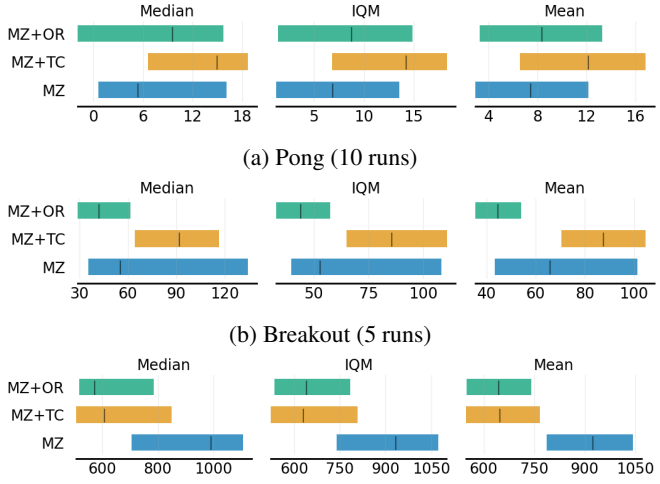
This appendix presents additional evaluation metrics and analysis of the results and detailed summaries of the results in tabular forms as an extension to section 5.

A.1 Further Result Analysis

In addition to the mean scores and 95% confidence intervals shown in the main text, we report median and interquartile mean (IQM) evaluation scores to provide a more robust assessment of agent performance. All aggregate scores are presented with 95% confidence intervals using stratified bootstrap sampling with 50,000 samples. IQM, the mean of the values inside the interquartile range, is used as a more stable metric robust to outliers in low-data settings while 95% confidence intervals are reported to account for variability and provide more reliable comparisons, as advised in [24]. Mean and median are included for completeness and consistency with prior work.

Figure 8 presents the extended set of evaluation metrics for the final results of the first experiment, comparing the effectiveness of the model-learning losses when all hyperparameters are constat. Figure 9 presents the model-learning loss weight impact on the augmented agents in Pong, as discussed in the second experiment, while Figure 10 presents it for the other two environments. We present the same results in tabular form in appendix A.2.

Overall we observe the same trends discussed in section 5 between all metrics, with the only exception being the median of the observation-reconstruction agent in MsPacman increasing for the higher weight coefficient while the other metrics decrease (Figure 10). Furthermore, the IQMs in Figure 8 show overall higher differences in performance than the means in Pong and Breakout. This can be attributed to the presence of extreme outliers in some runs (e.g. in Breakout 4 of the 5 runs of the baseline present evaluated scores with values in [35.44, 55.36] and one run a score of 134.60, see Table 4 in Appendix A.2).

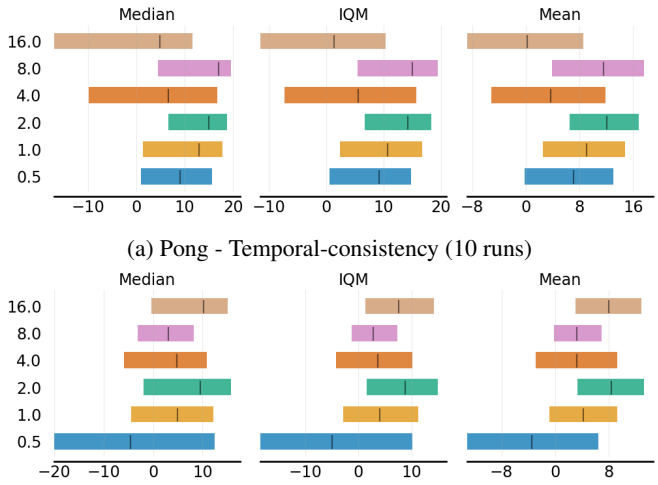


(a) Pong (10 runs)

(b) Breakout (5 runs)

(c) MsPacman (5 runs)

Figure 8: Aggregate metrics for agent variants across environments. All agent variants are evaluated with a loss coefficient of 2. Median, IQM, and Mean are shown as black lines, with 95% confidence intervals represented as shaded boxes. Pong results are aggregated over 10 runs; Breakout and MsPacman over 5 runs.



(a) Pong - Temporal-consistency (10 runs)

(b) Pong - Observation-reconstruction (10 runs)

Figure 9: Aggregate metrics per model-learning loss weight coefficient for augmented agents in Pong. Median, IQM, and Mean are shown as black lines, with 95% confidence intervals represented as shaded boxes. Results obtained from 10 runs per coefficient.

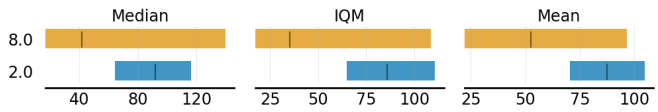
Furthermore, to obtain a more complete analysis of the differences in agent variant performances we compare them using the probability of improvement metric as recommended by [24], based on the Mann-Whitney U-statistic [35]:

$$P(X_m > Y_m) = \frac{1}{NK} \sum_{i=1}^{N} \sum_{j=1}^{K} S(x_{m,i}, y_{m,j})$$

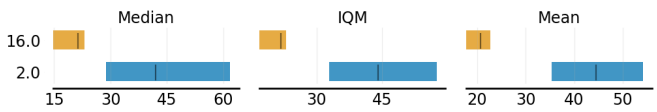
where

$$S(x, y) = \begin{cases} 1, & \text{if } y < x, \\ 0.5, & \text{if } y = x, \\ 0, & \text{if } y > x. \end{cases}$$

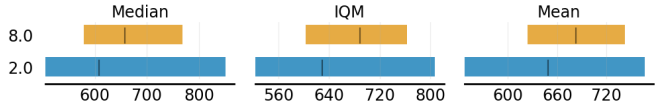
This is used to estimate how likely agent X is to outperform agent Y in task m, when the agents are evaluated for N and K runs respectively . It is important to note that this metric does not account for the magnitude of the improvements one variant brings



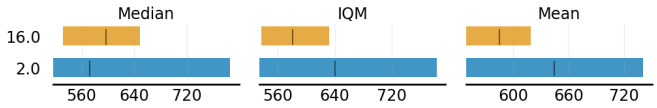
(a) Breakout – Temporal-consistency (5 runs)



(b) Breakout – Observation-reconstruction (5 runs)



(c) MsPacman – Temporal-consistency (5 runs)



(d) MsPacman – Observation-reconstruction (5 runs)

Figure 10: Aggregate metrics per model-learning loss weight coefficient in **Breakout** and **MsPacman** for the two augmented MuZero variants. Median, IQM, and Mean are shown as black lines with 95% confidence intervals as shaded boxes. Results obtained from 5 runs per coefficient.

over another, just how often it outperforms it. A probability of improvement around 0 shows that algorithm X underperforms algorithm Y almost always, a probability around 1 means X always performs better almost all the time, and a probability around 0.5 means the two algorithms performs similarly. We provide a probability of improvement per environment and then the overall probability of improvement across all environments. For the latter the scores are normalized as:

$$\frac{score_{agent,env} - random_{env}}{mean_{baseline,env} - random_{env}}$$

where $random_{env}$ is the score obtained in an environment by a random policy, $score_{agent,env}$ is the score achieved by the agent in the environment in one run and $mean_{baseline,env}$ is the mean of the scores achieved by the baseline agent in an environment. We use this normalization because raw scores vary drastically across environments, making direct comparisons impossible. Some environments, like MsPacman and Breakout, have very high upper bounds on achievable scores, while others like Pong have a much lower maximum score which is actually reached in practice on some runs. Unlike a min-max normalization that scales between the absolute minimum and maximum possible scores, we set the baseline agent's mean performance as the upper bound (normalized to 1) and a random policy as the lower bound (normalized to 0).

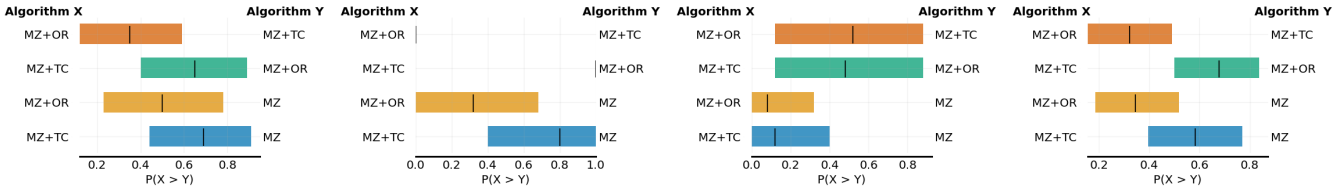
The probability for improvements of the agent variants are shown in Table 1 and Figure 11. We see that across all tasks the temporal-consistency agent is likely to outperform the baseline on 58% of the runs, while the observation-reconstruction in 34% of them. Again these result do not account for the size of the improvement. This shows however that even when taking into account the underwhelming performance in MsPacman the temporal-consistency agent is likely to outperform the baseline and observation-reconstruction augmented agents across all tested environments, while the observation-reconstruction has significantly lower chances to outperform it, the probability of improvement coming mostly from its decent performance in Pong.

Table 1: Probability of Improvement between algorithm pairs for Pong, Breakout, MsPacman, and across all environments, with 95% confidence intervals. For each algorithm pair (X, Y), each entry represents $P(X > Y)$ with 95% confidence intervals. Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent.

Variant Pair	Pong	Breakout	MsPacman	All Envs
MZ+TC, MZ	0.69 [0.44, 0.91]	0.80 [0.40, 1.00]	0.12 [0.00, 0.40]	0.58 [0.40, 0.77]
MZ+OR, MZ	0.50 [0.23, 0.78]	0.32 [0.00, 0.68]	0.08 [0.00, 0.32]	0.34 [0.18, 0.52]
MZ+TC, MZ+OR	0.65 [0.40, 0.89]	1.00 [1.00, 1.00]	0.48 [0.12, 0.88]	0.68 [0.50, 0.84]
MZ+OR, MZ+TC	0.35 [0.12, 0.59]	0.00 [0.00, 0.00]	0.52 [0.12, 0.88]	0.32 [0.15, 0.49]

A.2 Tabular Results

This appendix presents additional tables with experiment results and summaries. Tables 2 and 3 present the by run results of each agent configuration (model-learning loss and weight coefficient) and the summaries of the results respectively for the environment of Pong. In a similar fashion the results for Breakout are presented in Tables 4 and 5, and the results for MsPacman are presented in tables 6 and 7.



(a) Pong (b) Breakout (c) MsPacman (d) All Envs

Figure 11: Probability of improvement for each algorithm pair in different environments. Each bar represents $P(X > Y)$ with 95% confidence intervals. Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent.

Table 2: Pong Final Return Values per Run. Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent, W = weight parameter. Numbers rounded to 2 decimal places.

Model	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**	**Run 9**	**Run 10**
MZ	-1.90	-1.54	1.28	2.74	2.76	8.00	10.32	16.10	16.84	19.38
MZ+TC (W=0.5)	-18.18	-2.62	3.40	4.40	6.74	11.26	15.00	15.02	16.30	19.66
MZ+TC (W=1.0)	-13.04	0.82	1.36	3.74	11.36	14.50	15.44	17.84	17.90	20.52
MZ+TC (W=2.0)	-6.88	2.12	8.54	11.08	13.56	16.36	17.22	18.68	20.24	20.44
MZ+TC (W=4.0)	-20.44	-12.22	-9.94	-1.32	-0.28	13.36	14.80	16.76	17.52	18.70
MZ+TC (W=8.0)	-17.14	3.20	4.46	12.84	14.84	19.16	19.34	19.40	19.84	20.54
MZ+TC (W=16.0)	-21.00	-20.98	-17.08	0.42	3.74	5.90	7.50	7.7000	15.66	19.9800
MZ+OR (W=0.5)	-21.00	-20.98	-20.06	-17.34	-14.60	-0.02	5.52	8.80	16.70	20.28
MZ+OR (W=1.0)	-6.86	-5.40	-4.40	-2.40	1.68	8.28	9.34	11.86	14.90	14.92
MZ+OR (W=2.0)	-3.72	-2.68	-1.96	7.42	8.10	11.10	12.38	15.76	18.12	19.00
MZ+OR (W=4.0)	-13.26	-8.94	-5.90	0.16	4.64	4.84	9.02	9.06	12.74	19.98
MZ+OR (W=8.0)	-4.48	-3.22	-3.10	1.24	2.22	3.94	4.40	8.28	8.68	14.50
MZ+OR (W=16)	-3.10	-1.36	0.12	0.74	9.34	11.18	11.44	12.76	18.76	19.70

Table 3: Summary Statistics for Pong (10 runs per entry). Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent, W = weight parameter. Numbers rounded to 2 decimal places; Mean, IQM, and Median include 95% confidence intervals in brackets.

Model	Min	Max	Mean	IQM	Median	Std Dev
MZ	-1.90	19.38	7.40 [2.86, 12.11]	6.87 [1.08, 13.58]	5.38 [0.60, 16.10]	7.92
MZ+TC (W=0.5)	-18.18	19.66	7.10 [-0.04, 13.13]	9.30 [0.70, 14.81]	9.00 [0.89, 15.65]	11.25
MZ+TC (W=1.0)	-13.04	20.52	9.04 [2.41, 14.76]	10.71 [2.37, 16.74]	12.93 [1.36, 17.84]	10.57
MZ+TC (W=2.0)	-6.88	20.44	12.14 [6.53, 16.80]	14.24 [6.82, 18.25]	14.96 [6.60, 18.73]	8.81
MZ+TC (W=4.0)	-20.44	18.70	3.69 [-4.93, 11.87]	5.56 [-7.07, 15.75]	6.54 [-9.94, 16.76]	14.40
MZ+TC (W=8.0)	-17.14	20.54	11.65 [3.83, 17.64]	15.01 [5.45, 19.44]	17.00 [4.46, 19.59]	11.95
MZ+TC (W=16.0)	-21.00	19.98	0.18 [-8.82, 8.57]	1.36 [-11.63, 10.43]	4.82 [-17.08, 11.58]	14.84
MZ+OR (W=0.5)	-21.00	19.84	-3.48 [-13.16, 6.33]	-4.95 [-18.40, 9.89]	-4.54 [-20.06, 12.50]	16.77
MZ+OR (W=1.0)	-6.86	14.92	4.19 [-0.88, 9.26]	4.06 [-2.95, 11.27]	4.98 [-4.40, 12.12]	8.62
MZ+OR (W=2.0)	-3.72	19.00	8.35 [3.27, 13.28]	8.80 [1.48, 14.89]	9.60 [-1.96, 15.76]	8.58
MZ+OR (W=4.0)	-13.26	19.98	3.23 [-2.87, 9.19]	3.64 [-4.12, 10.16]	4.74 [-5.90, 10.88]	10.31
MZ+OR (W=8.0)	-4.48	14.50	3.25 [-0.22, 6.90]	2.83 [-1.33, 7.22]	3.08 [-3.10, 8.28]	6.06
MZ+OR (W=16.0)	-3.10	19.70	7.96 [3.15, 12.87]	7.60 [1.37, 14.28]	10.26 [-0.31, 15.10]	8.33

Table 4: Breakout Final Return Values per Run. Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent, W = weight parameter. Numbers rounded to 2 decimal places.

Model	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**
MZ	35.44	48.04	55.32	55.36	134.60
MZ+TC (W=2.0)	64.50	65.88	91.88	99.40	116.44
MZ+TC (W=8.0)	17.24	17.24	41.94	46.70	139.60
MZ+OR (W=2.0)	28.80	41.04	42.02	49.16	61.78
MZ+OR (W=16.0)	14.68	21.22	21.38	22.82	23.0400

Table 5: Summary Statistics for Breakout (5 runs per entry). Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent, W = weight parameter. Numbers rounded to 2 decimal places; Mean, IQM, and Median include 95% confidence intervals in brackets.

Model	Min	Max	Mean	IQM	Median	Std Dev
MZ	35.44	134.60	65.75 [43.39, 101.43]	52.91 [39.64, 108.19]	55.32 [35.44, 134.60]	39.34
MZ+TC (W=2.0)	64.50	116.44	87.62 [70.53, 104.71]	85.72 [64.96, 110.76]	91.88 [64.50, 116.44]	22.33
MZ+TC (W=8.0)	17.24	139.60	52.54 [22.18, 96.55]	35.29 [17.24, 108.63]	41.94 [17.24, 139.60]	50.54
MZ+OR (W=2.0)	28.80	61.78	44.56 [35.40, 53.88]	44.07 [32.88, 57.57]	42.02 [28.80, 61.78]	12.09
MZ+OR (W=16.0)	14.68	23.04	20.63 [17.62, 22.63]	21.81 [16.86, 22.97]	21.38 [14.68, 23.04]	3.43

Table 6: MsPacman Final Return Values per Run. Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent, W = weight parameter. Numbers rounded to 2 decimal places.

Model	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**
MZ	706.20	807.80	992.20	1003.20	1107.20
MZ+TC (W=2.0)	505.60	558.80	607.80	722.60	850.20
MZ+TC (W=8.0)	578.80	652.20	658.00	755.80	767.60
MZ+OR (W=2.0)	516.60	569.40	572.20	778.20	785.20
MZ+OR (W=16.0)	531.80	547.80	596.60	599.20	648.80

Table 7: Summary Statistics for MsPacman (5 runs per entry). Models: MZ = Baseline MuZero Reanalyze agent, MZ+TC = Temporal-Consistency augmented MuZero Reanalyze agent, MZ+OR = Observation-Reconstruction augmented MuZero Reanalyze agent, W = weight parameter. Numbers rounded to 2 decimal places; Mean, IQM, and Median include 95% confidence intervals.

Model	Min	Max	Mean	IQM	Median	Std Dev
MZ	706.20	1107.20	923.32 [785.92, 1042.60]	934.40 [740.07, 1072.53]	992.20 [706.20, 1107.20]	162.34
MZ+TC (W=2.0)	505.60	850.20	649.00 [547.32, 766.40]	629.73 [523.33, 807.67]	607.80 [505.60, 850.20]	138.11
MZ+TC (W=8.0)	578.80	767.60	682.48 [624.00, 742.16]	688.67 [603.27, 763.67]	658.00 [578.80, 767.60]	78.88
MZ+OR (W=2.0)	516.60	785.20	644.32 [548.84, 740.64]	639.93 [534.20, 782.87]	572.20 [516.60, 785.20]	127.38
MZ+OR (W=16.0)	531.80	648.80	584.84 [548.48, 618.68]	581.20 [537.13, 632.27]	596.60 [531.80, 648.80]	46.42

B Hyper-parameters and Environment Settings

This appendix provides a comprehensive overview of the hyperparameters used in our MuZero implementation for Atari game environments. The configuration is structured into several key components that control different aspects of the training process, from environment setup to learning dynamics and Monte Carlo Tree Search parameters.

B.1 Training Configuration

The training configuration defines the overall experimental setup for learning and evaluation. The system is configured for a 100K environment step regime as per the Atari100K [5] benchmark.

Table 8: Training Configuration Parameters

Parameter	Value	Description
Environment Steps	102,000	Total number of environment interactions, 2,000 of which with a random policy to reach the minimum size of the replay buffer
Offline Update Steps	0	Number of offline learning steps (disabled)
Evaluation Episodes	10	Number of evaluations
Learning Steps	12750	Number of learning steps
Reanalyze Steps	510	Number of reanalyze steps
Gradient Steps	8	Gradient updates per learning step

Table 9: Environment-Specific Parameters

Parameter	Training	Evaluation	Description
Number of Environments	8	50	Parallel environment instances for parallel data gathering or evaluation
Max Episode Steps	3,000	10,000	Episode length limit
NoOp Max	30	30	Random no-op actions
Reward Clipping	True	False	Clip rewards to [-1, 1]
Terminal on Life Loss	True	False	Episode ends on life loss

Learning occurs at every step, ensuring continuous parameter updates throughout training. Training uses shorter episodes with reward clipping for stability, while evaluation uses full-length episodes without clipping for accurate performance assessment. Multiple environments are used in parallel for data gathering to speed up training. The significant number of evaluation environments ensures robust performance estimates. Each learning steps 8 new observations are gathered from each of the parallel training environments, then 8 gradient updates are performed, leading to 100,000 network updates for training.

B.2 Environment Configuration

The environment setup defines the Atari game configuration and preprocessing pipeline. We use 4 stacked frames for each observation, leading to observation sizes of $96 \times 96 \times 12$ when accounting for the RGB channels of the frames.

Table 10: Common Environment Parameters

Parameter	Value	Description
Frame Skip	4	Action repetition frames, only save every 4th frame in the replay buffer
Frame Stack	4	Consecutive frames stacked to make an observation with a frame skip of 4 this leads to one observation representing 16 real world frames
Screen Size	96×96	Resized frame dimensions
Frame channels	3	Three channels for RGB frames

B.3 Monte Carlo Tree Search Configuration

MCTS parameters control the planning process during action selection.

Table 11: MCTS Configuration Parameters

Parameter	Training	Eval	Reanalysis	Description
Simulations	50	50	50	MCTS simulation count
PB-C Init	1.25	1.25	1.25	UCB exploration constant
PB-C Base	19,652	19,652	19,652	UCB base parameter
Dirichlet Alpha	0.3	0.3	0.3	Dirichlet noise parameter
Dirichlet Fraction	0.25	0.0	0.25	Noise mixing ratio
Q-Transform	Min-Max	Min-Max	Min-Max	Value normalization method

The temperature schedule for training starts with high exploration, temperature = 1.0 for the first 50% of training, reducing to 0.5 at 75%, and finally to 0.25 for the remainder. Evaluation uses zero temperature for deterministic action selection, while reanalysis maintains the same exploratory schedule as training. We use min-max normalization to scale Q-values in the MCTS backup to avoid overconfidence in UCT calculations, as described by [7].

B.4 Replay Buffer and Prioritized Experience Replay

The replay buffer configuration incorporates Prioritized Experience Replay (PER) [36] to improve sample efficiency by focusing learning on more informative transitions.

Table 12: Replay Buffer and PER Configuration

Parameter	Value	Description
Minimum Size	2,000	Buffer size before learning starts
Maximum Size	102,000	Maximum buffer capacity
PER Alpha	0.6	Prioritization exponent
PER Beta Start	0.4	Initial importance sampling correction
PER Beta End	1.0	Final importance sampling correction
Priority Method	Value prediction error	Priority calculation method
Correction Frequency	25	Steps between sum-tree corrections

The buffer size matches the total training steps, allowing the system to store the entire training trajectory. The PER beta parameter is annealed from 0.4 to 1.0 throughout training to gradually increase the importance sampling correction.

B.5 Optimizer Configuration

The optimization setup uses AdamW with gradient clipping to ensure stable learning dynamics throughout the training process.

Table 13: Optimizer Configuration

Parameter	Value	Description
Optimizer Type	AdamW	Optimization algorithm
Batch Size	256	Mini-batch size
Learning Rate	0.001	Step size parameter
Weight Decay	1×10^{-4}	L2 regularization coefficient
Gradient Clipping	5.0	Maximum gradient norm
Warmup Ratio	0.0	Learning rate warmup (disabled)

B.6 Loss Function Configuration

The loss function balances different learning objectives in MuZero, with specific coefficients for each component and temporal difference learning parameters.

The model-learning loss weight used in the main comparison is equal to 2. Other values for the model-loss are also used in experiments, see section 5. The other coefficients are kept the same in all experiments.

B.7 Value Transformation

MuZero employs a categorical value representation approach instead of direct scalar regression for value prediction. This technique discretizes the continuous value space into categorical bins. The signed hyperbolic transformation is applied to handle extreme values.

Table 14: Loss Function Configuration

Parameter	Value	Description
Unroll Steps	5	Model unroll depth
Reward Loss Coeff.	1.0	Reward prediction weight
Policy Loss Coeff.	1.0	Policy prediction weight
Value Loss Coeff.	0.25	Value prediction weight
TD Steps	5	Temporal difference horizon
Entropy Coefficient	0.0	Policy regularization (disabled)
Model Loss	2	Model-learning loss weight

Table 15: Value Transformation Parameters

Parameter	Value	Description
Number of Bins	601	Discretization resolution
Support Range	$[-300, 300]$	Value function range
Transformation	Enabled	Use signed hyperbolic transform
Discount Factor	0.997	Reward discount factor

C Network architecture

The MuZero network architecture is composed of several interconnected sub-networks. The implementation is done in JAX [30], using FLAX [37] to define the networks.

The primary activation function used throughout the networks is Leaky ReLU (unless specified otherwise for optional components), and the primary normalization technique is Layer Normalization (LN). Specific configurations, such as the number of output planes or units, are detailed below. Weight initialization is generally done using a variance scaling initializer.

C.1 Base networks

The base MuZero networks are the (a) Representation, (b) Dynamics, (c) Reward, (d) Value and (e) Policy networks. These network are implemented by all MuZero variants (baseline, temporal-consistency augmented, observation-reconstruction augmented), their architecture being kept the same.

(a) Representation Network This network is responsible for encoding the input observation o_t (i.e., a stack of game frames, $96 \times 96 \times C_{in}$ where $C_{in} = 12$, corresponding to 4 stacked RGB images) into a lower-dimensional hidden state representation s_t^0. Input observations are first normalized by dividing pixel values by 255.0. The layers are as follows:

1. 3×3 Convolution (output planes: 32, stride: 2, padding: SAME, no bias) + LN + Leaky ReLU
2. Residual Block (output planes: 32, stride: 1)
3. Residual Block (output planes: 64, stride: 2, with downsampling shortcut)
4. Residual Block (output planes: 64, stride: 1)
5. Average Pooling (3×3 window, stride: 2, padding: SAME)
6. Residual Block (output planes: 64, stride: 1)
7. Average Pooling (3×3 window, stride: 2, padding: SAME)
8. Residual Block (output planes: 64, stride: 1)

The output of this network is the hidden state s_t^0 (of shape $6 \times 6 \times 64$). This output hidden state is subsequently normalized using min-max scaling, applied per-channel across the spatial dimensions.

(b) Dynamics Network The Dynamics Network models the environment's transitions. Given a current hidden state s_t^k and an action a_t^{k+1}, it predicts the next hidden state s_t^{k+1}. The input action a_t^{k+1} is one-hot encoded and broadcasted to form a channel plane, which is then concatenated with the current hidden state s_t^k (64 planes), resulting in an input tensor with 65 planes. The gradient flowing into this network from subsequent computations is scaled by a factor of 0.5. The layers for next state prediction are:

1. 3×3 Convolution (output planes: 64, stride: 1, padding: SAME, no bias), applied to the state-action input + LN
2. Element-wise sum with the input hidden state s_t^k (residual connection) + Leaky ReLU
3. Residual Block (output planes: 64, stride: 1)

The output is the predicted next hidden state s_t^{k+1}. This state is also normalized using min-max scaling per-channel.

(c) Reward Prediction Network This head predicts the distribution of immediate rewards from a given hidden state s_t^k. The input is the hidden state s_t^k (64 planes). The layers are as follows:

1. Residual Block (output planes: 64, stride: 1)
2. 1×1 Convolution (output planes: 128, stride: 1, padding: SAME, no bias) + LN + Leaky ReLU
3. Flatten spatial dimensions
4. Dense layer (output units: 64, no bias) + LN + Leaky ReLU
5. Dense layer (output units: 601, corresponding to reward bins, with bias, weights zero-initialized)

The output is a tensor of reward logits (a distribution over 601 bins).

(d) Value Prediction Head This head predicts the distribution of the state value from a processed hidden state s'_k. The input s'_k is obtained by passing the original hidden state s_k (64 planes) through one Residual Block (output planes: 64, stride: 1, Leaky ReLU activation, Layer Normalization). The layers are as follows:

1. Residual Block (output planes: 64, stride: 1)
2. 1×1 Convolution (output planes: 128, stride: 1, padding: SAME, no bias) + LN + Leaky ReLU
3. Flatten spatial dimensions
4. Dense layer (output units: 64, no bias) + LN + Leaky ReLU
5. Dense layer (output units: 601, corresponding to value bins, with bias, weights zero-initialized)

The output is a tensor of value logits (a distribution over 601 bins).

(e) Policy Prediction Head This head predicts the policy (distribution over actions) from a processed hidden state s'_k. Similar to the Value Head, the input s'_k is obtained by passing the original hidden state s_k (64 planes) through one Residual Block (output planes: 64, stride: 1, Leaky ReLU activation, Layer Normalization). The layers are as follows:

1. 1×1 Convolution (output planes: 128, stride: 1, padding: SAME, no bias) + LN + Leaky ReLU
2. Leaky ReLU activation
3. Flatten spatial dimensions
4. Dense layer (output units: 64, no bias) + LN + Leaky ReLU
5. Dense layer (output units: N_A, where N_A is the number of actions, with bias, weights zero-initialized)

The output is a tensor of policy logits (a distribution over N_A actions).

Residual Block Used throughout the base networks (Representation, Dynamics, and prediction heads), this block adds a skip connection over two convolutional layers.

Core path:

1. 3×3 Convolution (output planes: `F`, stride: `S`, no bias) + LN + Leaky ReLU
2. 3×3 Convolution (output planes: `F`, stride: 1, no bias) + LN

Shortcut path:

- Identity if **no downsample is required**
- Otherwise: 3×3 Convolution (output planes: `F`, stride: 2, no bias)

Output: Element-wise sum of core and shortcut + Leaky ReLU

C.2 Temporal-consistency loss specific networks

If the SimSiam-style temporal-consistency loss is employed the baseline MuZero agent is augmented with two aditional networks: Projector and Predictor networks. The Projector network is used on both branches of the SimSiam loss (see section 2) sharing the same parameters, but only the Predictor branch propagates gradients during updates.

Projector Network It projects the flattened hidden state from the Representation and Dynamics Networks into an embedding space. The activation function is ReLU and normalization is LayerNorm. The input hidden state is first flattened into a one dimensional vector. The layers are as follows:

1. Dense layer (output units: 1024) + LN + Leaky ReLU
2. Dense layer (output units: 1024) + LN + Leaky ReLU
3. Dense layer (output units: 1024)

The output is a projected representation (1024 units).

Predictor Network It further processes the output of the Projection Network for one branch of the SimSiam architecture. The activation function is ReLU and normalization is LayerNorm. The input is the output from Projection Network (1024 units). The layers are as follows:

1. Dense layer (output units: 512) + LN + Leaky ReLU
2. Dense layer (output units: 1024)

The output is a prediction for SimSiam loss (1024 units).

C.3 Observation-reconstruction loss specific networks

If the autoencoder-style observation-reconstruction loss is used the baseline MuZero agent is augmented with a hidden state to observation decoder.

Representation Decoder It decodes a hidden state back into an observation-like space, architecturally mirroring the Representation Network in reverse, replacing all convolutions with deconvolutions (transpose convolution layers). The internal activation is Leaky ReLU and normalization is LayerNorm by default. The final activation is Sigmoid by default. The input is a hidden state s_t^k (64 planes). The layers are as follows:

1. Decoder Residual Block (input/output planes: 64, stride: 1)
2. 3×3 Transposed Convolution (output planes: 64, stride: 2, padding: SAME, no bias) + LN + Leaky ReLU
3. Decoder Residual Block (input/output planes: 64, stride: 1).
4. 3×3 Transposed Convolution (output planes: 64, stride: 2, padding: SAME, no bias) + LN + Leaky ReLU
5. Decoder Residual Block (input/output planes: 64, stride: 1)
6. Decoder Residual Block (input planes: 64, output planes: 32, stride: 2, with upsampling shortcut)
7. Decoder Residual Block (input/output planes: 32, stride: 1)
8. 3×3 Transposed Convolution (output planes: 12, stride: 2, padding: SAME, with bias)
9. Sigmoid activation

The output is a reconstructed observation-like tensor, with all channel values in the range $[0, 1]$.

Decoder Residual Block Used in the Representation Decoder, this block mirrors the Residual Block but uses transposed convolutions to upsample hidden states.

Core path:

1. 3×3 Transposed Convolution (output planes: `F`, stride: `S`, no bias) + LN + Leaky ReLU
2. 3×3 Transposed Convolution (output planes `F`, stride: 1, no bias) + LN

Shortcut path:

- Identity if **no upsampling is required**
- Otherwise: 3×3 Transposed Convolution (output planes: `F`, stride: 2, no bias)

Output: Element-wise sum of core and shortcut + Leaky ReLU

D Acknowledgements

Research reported in this work was partially or completely facilitated by computational resources and support of the Delft AI Cluster (DAIC) at TU Delft (RRID: SCR_025091), but remains the sole responsibility of the authors, not the DAIC team [38]. The authors also acknowledge the use of computational resources of the DelftBlue supercomputer, provided by Delft High Performance Computing Centre (https://www.tudelft.nl/dhpc) [39].