# TransZero: Parallel Tree Expansion in MuZero using Transformer Networks

Emil Malmsten

# TransZero: Parallel Tree Expansion in MuZero using Transformer Networks

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Emil Malmsten

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology

Delft, Netherlands
www.ewi.tudelft.nl

# TransZero: Parallel Tree Expansion in MuZero using Transformer Networks

Author: Emil Malmsten
Student id: 5256941

**Abstract**

Over the past decade, model-based reinforcement learning (MBRL) has become a leading approach for solving complex decision-making problems. A prominent algorithm in this domain is MuZero, which integrates Monte Carlo Tree Search (MCTS) with deep neural networks and a latent world model to predict future states and outcomes. Despite its effectiveness, MuZero is inherently limited by the sequential nature of its search-tree construction during planning. In this work, we address this limitation by introducing TransZero-Parallel, the first model capable of constructing MCTS without any sequential constraints. This method replaces MuZero's recurrent dynamics model with a transformer-based network, enabling the computation of a sequence of latent future states in parallel. We combine this with the MVC evaluator, which allows the search tree to be built without depending on the inherently sequential *visitation counts*. Together with small modifications to the MCTS algorithm, this enables the parallel expansion of entire subtrees within the search tree. Experiments in MiniGrid and LunarLander environments demonstrate that this combined approach yields up to an eleven-fold reduction in wall clock time while maintaining sample efficiency. These results highlight the potential of TransZero-Parallel to improve planning performance and reduce training time in model-based RL—bringing the field closer to real-time, real-world applications. The code is available through GitHub [1].

Thesis Committee:

| | |
|---|---|
| Supervisor: | |
| Chair: | Dr. J.W. (Wendelin) Böhmer, Faculty EEMCS, TU Delft |
| Committee Member: | Dr. Tom Viering, Faculty EEMCS, TU Delft |

[1] `https://github.com/emalmsten/TransZero`

# Preface

This work would not have been possible without the insight and encouragement of Wendelin Böhmer. His introduction to the topic and direction gave this project its foundation. I also thank Tom Viering for his role on the thesis committee. Finally, I'm grateful to Albin Jaldevik for taking the time to answer my questions and share insights from his work.

Emil Malmsten
Delft, Netherlands
June 28, 2025

# Contents

# List of Figures

# Chapter 1

# Introduction

Deep Reinforcement Learning has shown consistent success in various closed-domain environments, particularly in games. A notable milestone was *AlphaGo* [1], the first system to reach superhuman performance in the game of Go, which is known for its vast state space. It combined deep learning with *Monte Carlo Tree Search* (MCTS) [2] to guide and evaluate moves. Its successor, *AlphaZero* [3], extended this achievement to multiple board games, relying solely on self-play without incorporating human expert knowledge. In 2019, *MuZero* [4] further advanced this line of research by attaining superhuman performance not only in the same board games as AlphaZero but also in a wide range of Atari games. This was accomplished by learning an implicit model of the environment without requiring explicit access to the environment's transition dynamics.

While MuZero has achieved notable results, it still has limitations. One of the main challenges lies in its sequential planning strategy. Specifically, MuZero unrolls action sequences recurrently, as each predicted latent state depends on the previous one and must be computed step by step using the model's dynamics network. In addition, it can expand only one promising node at a time in its search tree. This expansion is guided by visitation statistics. Each step requires updating these statistics before continuing to the next expansion. This inherently sequential process limits the efficiency and scalability of planning.

This research aims to overcome MuZero's sequential bottleneck by enabling it to expand multiple nodes in parallel during planning. Parallelizing the planning process improves both speed and scalability, significantly reducing training and inference time. The goal is to make MuZero more suitable for real-time decision-making in complex, dynamic environments such as robotics and autonomous systems.

To our knowledge, this is the first work to construct MCTS without any sequential constraints. Our approach consists of three main parts. First, to eliminate MuZero's reliance on recurrent rollouts, we introduce *TransZero*, which is a MuZero architecture using a transformer-based dynamics network. This model can perform an entire rollout in parallel by leveraging the self-attention mechanism to model temporal dependencies. Second, we adopt the *Mean-Variance Constrained (MVC)* evaluator from Jaldevik [5] to expand promising actions independently of how the search tree was built, i.e., without relying on visitation counts. Finally, we combine these two approaches, along with modifications to the MCTS, into what we call *TransZero-Parallel*. This model can evaluate the nodes of entire subtrees in parallel, allowing significantly faster planning than MuZero.

This research is guided by the following questions:

- How do the sample efficiency and computational speed of TransZero compare to those of MuZero?
- Can the latent search tree in MuZero be expanded using MVC while preserving sample efficiency?
- To what extent can planning speed be increased using TransZero-Parallel without sacrificing sample efficiency?

## 1.1 Contributions

The main contributions of this thesis are as follows:

- We propose a novel MuZero architecture in which the dynamics network is replaced by a transformer.
- This is the first work to integrate the MVC evaluator, originally created for AlphaZero, into the MuZero framework.
- We introduce a novel method for expanding entire subtrees in parallel by combining a transformer-based dynamics model with the MVC evaluator along with alterations to the MCTS.
- We empirically evaluate all proposed architectures in the MiniGrid and LunarLander environments.

## 1.2 Outline

The remainder of this thesis is organized as follows:

- **Chapter 2** gives the relevant background. It includes explanations of relevant reinforcement learning concepts, introduces how MuZero works, explains the general-purpose tree evaluation framework, and provides an overview of transformer architectures relevant to this work.
- **Chapter 3** presents our proposed method. It begins by detailing how we replace MuZero's dynamics network with a transformer-based architecture, followed by the introduction of the MVC evaluator and the parallel expansion technique.

- **Chapter 4** reviews related work. It covers the use of transformers in model-based reinforcement learning, their application in AlphaZero and MuZero, and prior efforts to parallelize Monte Carlo Tree Search.
- **Chapter 5** describes our experimental setup. We introduce the agents used, the environments tested, and the different types of experiments conducted.
- **Chapter 6** presents the results. It evaluates the proposed transformer-based architecture, the integration of the MVC evaluator, and the performance of the parallelized version.
- **Chapter 7** discusses the experimental findings and outlines future directions.
- **Chapter 8** summarizes the contributions of this thesis and explains their significance.

# Chapter 2

# Background

This chapter begins in section 2.1 with a brief overview of reinforcement learning. Section 2.2 details the MuZero algorithm, highlighting the components most relevant to this work. In section 2.3, we describe tree evaluation techniques from Jaldevik [5] that decouple value estimation from tree construction. Section 2.4 concludes with a summary of transformer networks.

## 2.1 Reinforcement Learning

Reinforcement Learning (RL) is a framework for training agents to make decisions through interaction with an environment. At its core is the concept of a *Markov Decision Process* (MDP) [6]. An MDP models the environment as a set of states. At each time step $t$, the agent is in a state $s_t$, selects an action $a_t$, receives a reward $r_t$, and transitions to a new state $s_{t+1}$. The key principle is that the next state depends only on the current state and action, not on the full history; this is known as the Markov property.

The agent often does not observe the full state directly. Instead, it receives an observation $o_t$, which provides partial information about the underlying state. In some environments, the observation fully reveals the current state; this corresponds to a standard MDP, where $o_t = s_t$. In other environments, the observation only provides partial information about the state. These cases are modeled as *Partially Observable Markov Decision Processes (POMDPs)*.

The main objective in both MDPs and POMDPs is to learn a policy $\pi(s,a)$ — a rule for choosing an action $a$ given a state $s$ — to maximize the agent's *expected return* over time. Formally, starting from state $s$, this return is captured by the *value function*:

$$V_\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, s_0 = s \right].$$

Here, $V_\pi(s)$ denotes the expected total reward the agent receives when it starts in state $s$ and follows policy $\pi$. The discount factor $\gamma \in [0,1]$ weights the immediate rewards more heavily than the future ones.

A common extension of RL is *model-based reinforcement learning* (MBRL), which incorporates a model of the environment's dynamics. This allows the agent to simulate future trajectories in a latent space and plan actions accordingly. The latent space is a lower-dimensional learned representation of the environment that captures the most essential features to make accurate predictions. This added structure typically improves sample efficiency.

## 2.2 MuZero: Monte Carlo Tree Search in Latent Space

MuZero [4] is a *Model-Based Reinforcement Learning* (MBRL) algorithm that extends AlphaZero [3] by learning an implicit model of the environment's dynamics from experience, rather than relying on known dynamics. It combines this learned model with Monte Carlo Tree Search (MCTS) [2] in latent space to plan future actions. MCTS is a popular heuristic search algorithm for sequential decision-making problems. It combines principles from search-tree algorithms — such as those used in Minimax [7] — with elements of Monte Carlo sampling. In this thesis, we define a search tree as a structure in which each node $x$ represents a unique action sequence that starts from an initial state or observation.

In subsection 2.2.1, we describe how MCTS builds a latent search tree by rolling out latent trajectories from an observation received from the environment. In subsection 2.2.2, we explain how the algorithm selects an action to take in the real environment based on the statistics generated by the latent search tree. Finally, subsection 2.2.3 discusses how MuZero learns from the trajectories generated during this process.

### 2.2.1 Search-Tree Construction in Latent Space

In MuZero, the initial state is the observation $o_t$, which is transformed into a latent space using a representation network $h$, parameterized by $\theta$:

$$\tilde{s}_{\text{root}} = h_\theta(o_t). \tag{2.1}$$

The latent root state $\tilde{s}_{\text{root}}$ serves as the root node of the search tree. Throughout this work, we use the terms latent state $\tilde{s}$ and tree node $x$ interchangeably, as they represent the same underlying concept in the search process. We denote $\mathcal{T}_x$ as the subtree rooted at node $x$.

The tree is built iteratively from the root node $x_{\text{root}}$ by performing three main steps: *selection*, *expansion*, and *backup*. These steps are detailed below and illustrated in Figure 2.1. Since the process is iterative, we assume that part of the search tree has already been constructed using this procedure. The three-step process is executed for a fixed number of *simulations* before selecting an action in the real environment, as described in subsection 2.2.2.

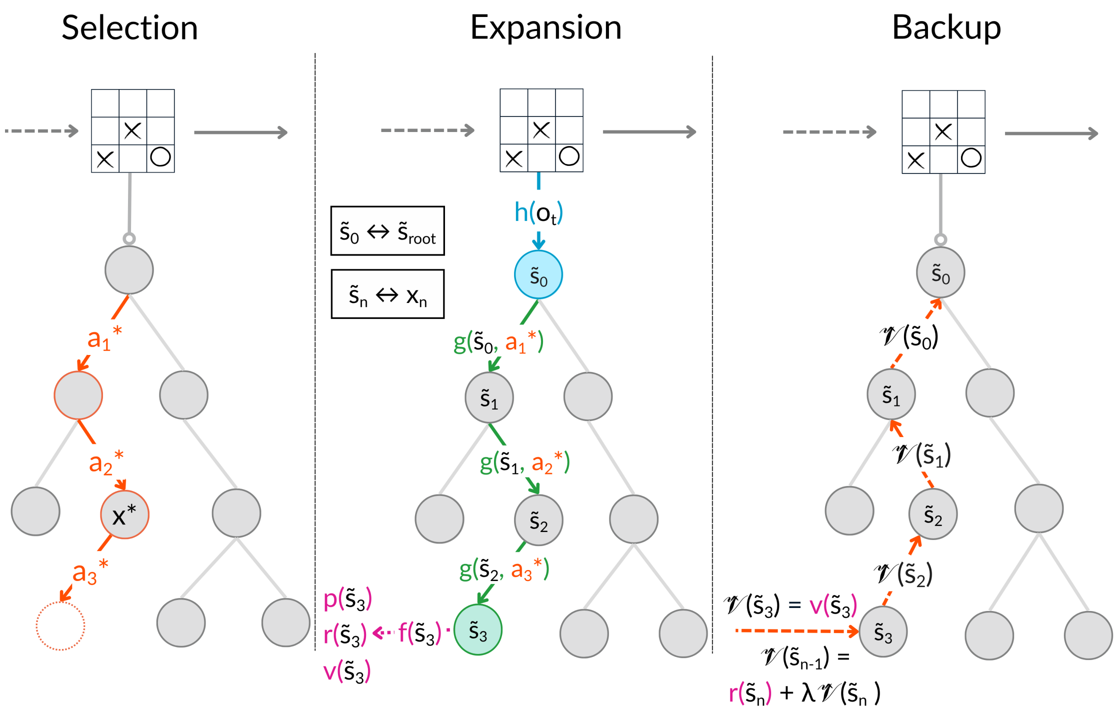

Selection
Expansion
Backup
$a_1^*$
$a_2^*$
$x^*$
$a_3^*$
$\tilde{s}_0 \leftrightarrow \tilde{s}_{root}$
$\tilde{s}_n \leftrightarrow x_n$
$h(o_t)$
$\tilde{s}_0$
$g(\tilde{s}_0, a_1^*)$
$\tilde{s}_1$
$g(\tilde{s}_1, a_2^*)$
$\tilde{s}_2$
$g(\tilde{s}_2, a_3^*)$
$\tilde{s}_3$
$p(\tilde{s}_3)$
$r(\tilde{s}_3)$
$f(\tilde{s}_3)$
$v(\tilde{s}_3)$
$\mathcal{V}(\tilde{s}_0)$
$\mathcal{V}(\tilde{s}_1)$
$\mathcal{V}(\tilde{s}_2)$
$\mathcal{V}(\tilde{s}_3) = v(\tilde{s}_3)$
$\mathcal{V}(\tilde{s}_{n-1}) =$
$r(\tilde{s}_n) + \lambda \mathcal{V}(\tilde{s}_n)$

Figure 2.1: Overview of how MCTS works in MuZero. The selection stage (orange arrows) shows which nodes and actions are chosen. In the expansion stage, an observation $o_t$ is encoded into a latent state $\tilde{s}_{\text{root}}$ using the representation network $h_\theta$ (blue arrow). The dynamics network $g_\theta$ (green arrows) updates latent states from selected actions and previous latent states. The prediction network $f_\theta$ (pink arrow) outputs the policy, value, and reward from the newly created latent state. During the backup stage, the value estimate $\hat{\mathcal{V}}$ is propagated back through its ancestors (dashed orange arrows).

### Selection

From the root node $x_{\text{root}}$, MuZero recursively selects actions in the search tree until it reaches a leaf node, i.e. a node that has not yet been expanded (expansion is explained in section 2.2.1). This selection process uses the PUCT formula, which balances exploration and exploitation:

$$a^* = \operatorname*{ArgMax}_{a \in \mathcal{A}} [Q(x \uplus a) + U(x \uplus a)] \tag{2.2}$$

where:

- $a^*$ is the action selected *within the search tree* (as opposed to the real environment),
- $\mathcal{A}$ is the action space of the environment—that is, all actions available to the agent,
- $x \uplus a$ denotes the node reached by taking action $a$ from node $x$,
- $Q(x)$ is the estimated Q-value for node $x$,
- $U(x)$ is an exploration bonus based on visit counts and prior probabilities.

The Q-value estimate $Q(x \uplus a)$ is computed as:

$$Q(x \uplus a) = r(x \uplus a) + \gamma \cdot \hat{V}(x \uplus a) \tag{2.3}$$

where:

- $r(x)$ is the reward predicted by the prediction network for node $x$. Note that it is different from the actual reward received in a time step $t$, denoted $r_t$.
- $\hat{V}(x)$ is a value estimate of $x$, computed as a discounted average of value estimates from the nodes in the subtree $\mathcal{T}_x$. While $V_\pi(s)$ is the true expected return under policy $\pi$ from state $s$, $\hat{V}(x)$ is MuZero's internal estimate based on simulations in the latent space. The computation of this value estimate will be detailed in section 2.2.1.

The exploration bonus $U(x \uplus a)$ is calculated as follows:

$$U(x \uplus a) = C_{\text{puct}} \cdot p(x,a) \cdot \frac{\sqrt{N(x)}}{1 + N(x \uplus a)}$$

where:

- $C_{\text{puct}}$ is a constant that controls the degree of exploration,
- $p(x,a)$ is the prior probability of taking action $a$ from node $x$. It is predicted by the policy network to guide the search toward the most promising actions.
- $N(x) = |\mathcal{T}_x|$ is the number of nodes in the subtree rooted at $x$, often referred to as the visitation count. It includes $x$ if $x$ has been *expanded* (see section 2.2.1).

### Expansion

When a leaf node $x_n$ is reached through the selection process, the algorithm moves onto the expansion stage. A leaf node is defined as one that has not yet been expanded, which means that predictions for value, reward, and policy priors have not been computed. The parent of the leaf node to be expanded is $x_{n-1}$ and we will denote it as $x^*$. To obtain these predictions, we first compute the latent state representation by recursively applying the learned dynamics function $g$, parameterized by $\theta$, as follows:

$$\tilde{s}_n = g_\theta(\tilde{s}_{n-1}, a^*).$$

Here, $\tilde{s}_{n-1}$ is the latent state representation of the parent node of $x_n$, and $a^*$ is the action selected by the selection process that leads from the parent node to $x_n$.

The application of the dynamics function must start from the latent root state $\tilde{s}_{\text{root}}$ and proceed recurrently. However, the intermediate latent states are cached at each node to avoid redundant computation.

From the latent state $\tilde{s}_n$, MuZero then uses its *prediction network* $f_\theta$, parameterized by $\theta$, to predict three outputs:

$$p(x_n),\ v(x_n),\ r(x_n) = f_\theta(\tilde{s}_n)$$

- $p(x_n) \in \mathbb{R}^{|\mathcal{A}|}$ is the *prior policy*, a probability distribution over the action space $\mathcal{A}$ when being in state $s_n$ i.e the state represented by $\tilde{s}_n$. Note that $\sum_{a\in\mathcal{A}} p(x_n,a) = 1$ and $p(x_n,a) \geq 0$.
- $v(x_n) \in \mathbb{R}$ is the *value*: the expected total return from state $s_n$,
- $r(x_n) \in \mathbb{R}$ is the *reward*: the immediate reward received for reaching state $s_n$.

**Backup**

After expansion and prediction, the value estimates at the leaf nodes are backed up to update the estimate values $\hat{V}$ at each parent node along the search path. At the leaf node the initial value is set such that:

$$\hat{\mathcal{V}}(x) = v(x) \quad \text{if leaf}(x).$$

Here, $\hat{\mathcal{V}}(x)$ denotes the value estimate at node $x$ for this specific simulation, note that this is different from $\hat{V}(x)$ which is the running average over all simulations. The operator leaf($x$) returns true if $x$ is a leaf node. This value estimate is then propagated upward through the tree using:

$$\hat{\mathcal{V}}(x) = r(x \uplus a) + \gamma \cdot \hat{\mathcal{V}}(x \uplus a) \quad \text{if not leaf}(x).$$

Finally, $\hat{\mathcal{V}}(x)$ is used to update the running average estimate $\hat{V}(x)$ at node $x$:

$$\hat{V}(x) \leftarrow \frac{N(x)\cdot \hat{V}(x) + \hat{\mathcal{V}}(x)}{N(x)+1}.$$

### 2.2.2 Action Selection

After running a fixed number of simulations, MuZero selects an action to execute in the real environment based on the search statistics generated by MCTS. Specifically, it defines a policy $\pi(s_t,a)$ as follows:

$$\pi(s_t,a) = \frac{N(s_t,a)^{1/\tau}}{\sum_{b\in\mathcal{A}} N(s_t,b)^{1/\tau}}.$$

This policy is derived from the subtree visit counts $N(x_{\text{root}} \uplus a)$ for all $a \in \mathcal{A}$, which, outside the context of MCTS, are denoted $N(s_t,a)$. Similarly, we refer to the value estimate $\hat{V}(x_{\text{root}})$ as $\hat{V}(s_t)$ outside the search. The parameter $\tau \in (0,\infty)$ is a temperature that controls the level of exploration. When $\tau > 0$, the policy assigns non-zero probabilities to multiple actions, encouraging exploration. This setting is typically used during training. In contrast, during inference or evaluation, we let $\tau \to 0$, which results in selecting the action corresponding to the child node with the largest subtree:

$$\lim_{\tau\to 0}\pi(s_t,a)=\begin{cases}1 & \text{if } a=\text{PolicyMax}_{a'\in\mathcal{A}}N(s_t\uplus a')\\ 0 & \text{otherwise}\end{cases}.$$

We define the *PolicyMax* function to return a uniform distribution over all actions that share the highest visit count. If only one action has the highest count, it is selected deterministically.

Once MuZero executes an action in the environment, it receives the next observation $o_{t+1}$ and constructs a new MCTS using the process described in subsection 2.2.1.

### 2.2.3 Training the Model

MuZero improves its model by training on environment trajectories obtained by *self-play*. Self-play refers to the agent generating trajectories through its own interaction with the environment. Each trajectory is *unrolled* over $K$ steps. Starting from the state $s_t$, it collects environment data and search statistics up to $s_{t+K}$. MuZero uses its learned dynamics model to reconstruct the trajectory in latent space. These latent states are generated recurrently, just as during MCTS:

$$\tilde{s}_0=h_\theta(o_t),\quad \tilde{s}_{i+1}=g_\theta(\tilde{s}_i,a_{t+i})\quad \text{for } i=0,\ldots,K-1.$$

At each latent state $\tilde{s}_i$, the prediction function $f_\theta$ outputs the policy priors, reward, and value:

$$p(\tilde{s}_i),r(\tilde{s}_i),v(\tilde{s}_i)=f_\theta(\tilde{s}_i)\quad \text{for } i=0,\ldots,K-1.$$

For the reward targets, we use the true environment rewards $r_i$ at each unrolled time step:

$$r_i^{target}=r_i.$$

The policy targets are the normalized visitation counts produced by MCTS at each step:

$$p_i^{\text{target}}=\frac{N(s_i,a)}{\sum_{b\in\mathcal{A}}N(s_i,b)}\ \forall a\in\mathcal{A}.$$

The value target is computed using a temporal-difference (TD) estimate:

$$v_i^{\text{target}}=\sum_{j=0}^{n_{TD}-1}\gamma^j r_{i+j}+\gamma^{n_{TD}}\cdot\hat{V}(s_{i+n_{TD}})$$

where $n_{TD}$ is the number of TD steps.

The overall training loss at time step $t$ is:

$$\mathcal{L}_t(\theta)=\sum_{i=0}^{K}\left[\ell^v(v(\tilde{s}_i),\,v_i^{\text{target}})+\ell^r(r(\tilde{s}_i),\,r_i^{\text{target}})+\ell^p(p(\tilde{s}_i),\,p_i^{\text{target}})\right]+\lambda\cdot||\theta||^2$$

- $\ell^v, \ell^r, \ell^p$: loss terms for value, reward, and policy. In our implementation, these are cross-entropy losses.
- $\lambda$: regularization coefficient.

## 2.3 General Tree Evaluation

In Jaldevik [5], the author proposes a method for evaluating the value of nodes in the MCTS of AlphaZero that is decoupled from how the tree was constructed. In both AlphaZero and MuZero, visitation counts are used to guide exploration and value estimation. However, this reliance on visitation counts poses challenges if one wishes to modify the tree construction strategy. For example, switching to breadth-first search would invalidate the use of visitation counts, since they would no longer reflect a meaningful measure of node value or exploration.

In subsection 2.3.1, we define the general framework for evaluating nodes in a search tree independent of visitation counts. Then, in subsection 2.3.2 we introduce the Q-evaluator, which focuses on maximizing estimated value. Next, in subsection 2.3.3 we present the minimal variance evaluator, which minimizes uncertainty in value estimates. Finally, subsection 2.3.4 combines both goals through the MVC evaluator, which balances high-value estimation with low variance.

### 2.3.1 Defining a Tree Evaluation Policy

Let $\tilde{\pi}$ denote a *tree evaluation policy* that operates over an *extended* action set:

$$\mathcal{A}_v = \mathcal{A} \cup \{a_v\}.$$

where $a_v$ is a special *simulation action* that corresponds to directly evaluating a node, in the case of MuZero, it is a neural-network value estimate. Since $a_v$ cannot be taken in the real environment, we can translate the tree evaluation policy $\tilde{\pi}$ into a usable environment policy $\pi$ by normalizing over the real action space:

$$\pi(x,a) = \frac{\tilde{\pi}(x,a)}{1-\tilde{\pi}(x,a_v)} \propto \tilde{\pi}(x,a).$$

Note that in this context, $\pi(x,a)$ is a policy determined over a tree node $x$, not a state $s$.

We define the estimated value of node $x$ under the tree evaluation policy $\tilde{\pi}$ as:

$$\hat{V}_{\tilde{\pi}}(x) = \sum_{a \in \mathcal{A}_v} \tilde{\pi}(x,a) Q_{\tilde{\pi}}(x \uplus a),$$

where $Q_{\tilde{\pi}}(x)$ is given by

$$Q_{\tilde{\pi}}(x) = r(x) + \gamma \cdot \hat{V}_{\tilde{\pi}}(x).$$

In this formulation, $\tilde{\pi}(x,a)$ is a probability distribution over all actions in $\mathcal{A}_v$. For the special *simulation action* $a_v$, we define its associated value directly using the network prediction:

$$Q_{\tilde{\pi}}(x \uplus a_v) = v(x).$$

### 2.3.2 The Q-evaluator

When designing a statistical estimator for node values, two primary considerations are *bias* and *variance*. In the context of search, bias occurs when the estimator systematically deviates from the true optimal value. One way to eliminate (negative) bias is by using the *Q-evaluator*, defined as:

$$\tilde{\pi}_Q(x) = \operatorname*{ArgMax}_{\tilde{\pi}} \hat{V}_{\tilde{\pi}}(x).$$

This policy selects the action that yields the highest Q-value estimate:

$$\tilde{\pi}_Q(x,a) = \operatorname*{PolicyMax}_{\mathcal{A}} Q_{\tilde{\pi}}(x \uplus a).$$

However, a limitation of this evaluator is that the Q-values are empirically estimated and may introduce significant variance. Because the Q-evaluator does not account for this variance, its value estimates may fluctuate considerably.

### 2.3.3 The Minimal Variance Evaluator

If the goal is to reduce the variance in the estimated value at each node, one can use the *minimal variance evaluator*, defined as:

$$\tilde{\pi}_{\text{var}} = \operatorname*{ArgMin}_{\tilde{\pi}} \mathbb{V}[\hat{V}_{\tilde{\pi}}].$$

To compute $\mathbb{V}[\hat{V}_{\tilde{\pi}}(x)]$, we make several assumptions: the environment is deterministic; rewards and simulation values are independent, and rewards across steps are uncorrelated. Under these assumptions, the variance simplifies to:

$$\mathbb{V}[\hat{V}_{\tilde{\pi}}(x)] = \gamma^{-2}\mathbb{V}[Q_{\tilde{\pi}}(x)].$$

If we further assume that the variance of each simulation value is constant and equal to $\sigma^2$, we can express the variance of the Q-value estimate as:

$$\mathbb{V}[Q_{\tilde{\pi}}(x)] = \sigma^2 \sum_{y \in \mathcal{T}_x} \frac{\Lambda_{\tilde{\pi}}(x,y,\gamma)^2}{\tilde{\pi}(y,a_y)^2},$$

where $\Lambda_{\tilde{\pi}}(x,y,\gamma)$ is the discounted probability of reaching node $y$ from node $x$ under policy $\tilde{\pi}$.

Based on this formulation, the minimal variance evaluator selects actions using the inverse of the Q-value variance:

$$\tilde{\pi}_{var}(x,a) = \frac{\mathbb{V}\left[Q_{\tilde{\pi}}(x \uplus a)\right]^{-1}}{\sum_b \mathbb{V}\left[Q_{\tilde{\pi}}(x \uplus b)\right]^{-1}} \propto \mathbb{V}\left[Q_{\tilde{\pi}}(x \uplus a)\right]^{-1}.$$

While this approach effectively reduces the variance in value estimates, it can lead to underestimation of the optimal value.

### 2.3.4 Balancing Mean and Variance: The MVC Evaluator [5]

In practice, neither a purely variance-minimizing nor a purely Q-maximizing approach is ideal. To interpolate between these extremes, one can use the *Mean-Variance Constrained (MVC) evaluator*, which balances assigning probability to high-value actions while keeping variance under control. This is achieved using a hyperparameter $\beta > 0$. The evaluator is defined as:

$$\tilde{\pi}_{\mathrm{MVC}} = \underset{\tilde{\pi}}{\mathrm{ArgMax}} \left( \hat{V}_{\tilde{\pi}} - \tfrac{1}{\beta} KL\big(\tilde{\pi}\,,\, \tilde{\pi}_{\mathrm{Var}}\big) \right)$$

where $KL(\cdot\,,\cdot)$ denotes the Kullback–Leibler divergence. This optimization has a closed-form solution:

$$\tilde{\pi}_{\mathrm{MVC}}(x,a) \propto \tilde{\pi}_{\mathrm{Var}}(x,a) \exp\big(\beta Q_{\tilde{\pi}}(x \uplus a)\big). \tag{2.4}$$

As $\beta \to 0^+$, the MVC policy converges to the minimal-variance policy $\tilde{\pi}_{\mathrm{Var}}$; as $\beta \to \infty$, it converges to the Q-evaluator $\tilde{\pi}_Q$.

## 2.4 Transformer Networks

Transformers are a deep learning architecture designed to process sequential data. Unlike recurrent networks, which handle sequences step by step, transformers use a self-attention mechanism that allows them to consider all positions in the sequence simultaneously. In subsection 2.4.1, we describe how the input sequence is constructed for the transformer encoder. In subsection 2.4.3 masking is introduced. In subsection 2.4.2 we present the self-attention mechanism, the core component of the transformer. In subsection 2.4.4, we extend this mechanism to multi-head attention. Then, in subsection 2.4.5, we explain how these components are integrated into the full transformer encoder. Finally, subsection 2.4.6 introduces the Vision Transformer, which adapts the transformer architecture for visual tasks.

### 2.4.1 Creating the Input Sequence

The input sequence we wish to transform into a latent representation is denoted as $X$. It consists of $n$ elements, referred to as tokens. In our setting, the first token represents the initial latent state, and each subsequent token corresponds to one action in an action sequence. Consequently, each token $x_i \in X$ can be interpreted as representing a node in the underlying search tree of the MCTS. Each token is mapped to a vector of dimension $d_t$ through an embedding process. Typically, a learned embedding layer is used for this purpose. We denote the token embedding operator as Token_Embed$(\cdot)$, which maps each token to a vector in $\mathbb{R}^d_t$.

Since transformers do not process inputs sequentially and do not use convolution, they require an additional mechanism to encode the position of each token in the sequence. This is achieved by adding *positional encodings* to the input embeddings. A common approach uses fixed sinusoidal functions:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{F^{2i/d_t}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{F^{2i/d_t}}\right).$$

Here, $pos \in \{1,\dots,n\}$ denotes the position index in the sequence, $i \in \{0,\dots,d_t/2-1\}$ is the dimension index, and $F$ is a constant controlling the frequency scale, typically set to 10,000. Sine functions are used for even indices $2i$ and cosine functions for odd indices $2i+1$. These encodings assign a unique, continuous pattern to each position and enable the model to infer both absolute and relative positions due to the linear properties of sinusoids.

If we add both the token embeddings and their corresponding positional encodings, we obtain the *input embeddings*. Formally, for a sequence of $n$ tokens $x_1,\dots,x_n$, the input embedding for the $i$-th token is given by:

$$X^{\text{emb}} = \text{Input\_Embed}(X) = [\text{Token\_Embed}(x_i) + PE(\mathcal{D}_i)]_{i=1}^{n} \in \mathbb{R}^{n \times d_t}.$$

Here, $\mathcal{D}_i$ denotes the position of token $x_i$ in the sequence, which in our setting corresponds to the depth of node $x_i$ in the search tree. The resulting sequence of embedded tokens $X^{\text{emb}}$ will serve as the input to the first layer of the transformer encoder.

### 2.4.2 Self-Attention Mechanism

Given the embedded tokens $X^{\text{emb}}$, the self-attention mechanism computes projections for the *query* ($Q'$), *key* ($K'$), and *value* ($V'$) matrices as follows:

$$Q' = X^{\text{emb}} W_{Q'}, \quad K' = X^{\text{emb}} W_{K'}, \quad V' = X^{\text{emb}} W_{V'}$$

where $W_{Q'}, W_{K'}, W_{V'} \in \mathbb{R}^{d_t \times d_k}$ are learned projection matrices, and $d_k$ is the dimensionality of the query and key vectors. Note that the term ”value” used in the context of transformers refers to the input to the attention mechanism and is unrelated to the value used in MuZero.

The scaled dot-product attention is computed as:

$$\text{Attention}(X^{\text{emb}}) = \text{softmax}\left(\frac{Q' K'^{T}}{\sqrt{d_k}}\right) V'.$$

The product $Q'K'^{\top} \in \mathbb{R}^{n \times n}$ is the *attention matrix*. Each entry $(i,j)$ is a raw score that tells how much token $j$ influences token $i$—this influence is what we mean by *attention*. These scores are used to weight the value vectors $V'$, which contain the actual information from each token. In effect, token $i$ builds its final representation by combining information from all other tokens, with each one contributing according to its attention score. The division by $\sqrt{d_k}$ is a scaling factor that keeps gradients stable.

### 2.4.3 Masking

In transformer architectures, *masking* is used to control which tokens can *attend*, i.e. influence, others during self-attention. A common form is the *causal attention mask*, denoted $M_{\text{causal}} \in \{0,1\}^{n \times n}$. This mask restricts the flow of information so that a token at position $i$ cannot attend any future positions $j > i$. Such masking is essential in autoregressive settings, where predictions must not depend on future inputs.

Formally, the causal mask is defined elementwise as:

$$M_{ij} = \begin{cases} 1 & \text{if } j \leq i \\ 0 & \text{otherwise.} \end{cases}$$

This binary matrix is applied during the computation of scaled dot-product attention. Given the projected queries $Q'$, keys $K'$, and values $V'$, the masked attention is computed as:

$$\text{Attention}(X^{\text{emb}}, M) = \text{softmax}\left(\frac{Q'K'^{\top}}{\sqrt{d_k}} + \log M\right)V'.$$

In practice, $\log M$ is not computed directly, since $\log(0)$ is undefined. Instead, positions where $M_{ij} = 0$ are replaced by a large negative constant (e.g., $-10^9$) to approximate the effect of $\log(0) \to -\infty$. This ensures that the softmax assigns near-zero probability to those positions, so each token only attends to itself and to previous tokens, as enforced by the mask.

### 2.4.4 Multi-Head Attention

Multi-head attention runs several independent self-attention computations—called *heads*—in parallel. Each head has its own learned projection matrices and can capture different types of relationships between embedded tokens.

For each head $i \in \{1, \ldots, h\}$, we define separate projection weights:

$$W_{Q',i},\ W_{K',i},\ W_{V',i} \in \mathbb{R}^{d_t \times d_k}.$$

Each head then computes:

$$Q'_i = X^{\text{emb}} W_{Q',i}, \quad K'_i = X^{\text{emb}} W_{K',i}, \quad V'_i = X^{\text{emb}} W_{V',i}$$

$$\text{head}_i = \text{softmax}\left(\frac{Q'_i K_i'^{\top}}{\sqrt{d_k}} + \log M\right) V'_i.$$

The outputs from all heads are concatenated and projected back to the original embedding dimension using a final projection matrix $W_O \in \mathbb{R}^{h \cdot d_k \times d_t}$:

$$\text{MultiHead}(X^{\text{emb}}, M) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W_O.$$

This mechanism allows the model to learn attention patterns in multiple representation subspaces and integrate them into a single output.

### 2.4.5 Transformer Encoder

The *transformer encoder* takes the embedded tokens $X^{\text{emb}}$ and outputs an updated sequence $\tilde{S} \in \mathbb{R}^{n \times d_t}$, where each token's representation has been refined based on information from the entire sequence. This is done by passing $X^{\text{emb}}$ through one or more stacked encoder layers.

Each encoder layer consists of two main components: masked multi-head self-attention and a feedforward network (FFN). Both are followed by residual connections and layer normalization. The FFN is a two-layer MLP applied independently to each token. It allows the model to transform features at each position without mixing information across tokens. Each residual connection adds the input of a sub-layer (e.g., attention or feedforward) to its output before applying layer normalization. This helps preserve information and improves training stability.

Given input $X^{\text{emb}}$, a single encoder layer performs the following steps:

$$\begin{aligned}
MHA &= \text{MultiHead}(X^{\text{emb}}, M_{\text{causal}}) && \text{(masked multi-head self-attention)}\\
LN &= \text{LayerNorm}(X^{\text{emb}} + MHA) && \text{(residual connection + normalization)}\\
FFN &= \text{ReLU}(LNW_1 + b_1)W_2 + b_2 && \text{(feedforward network)}\\
\tilde{S}^{(l)} &= \text{LayerNorm}(LN + FFN) && \text{(second residual connection + normalization)}
\end{aligned}$$

Here, $W_1 \in \mathbb{R}^{d_t \times d_{\text{ff}}}$ and $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_t}$ are weight matrices of the feedforward network, and $b_1 \in \mathbb{R}^{d_{\text{ff}}}$ and $b_2 \in \mathbb{R}^{d_t}$ are the corresponding bias terms. The intermediate dimension $d_{\text{ff}}$ is typically larger than $d_t$ and controls the capacity of the feedforward layer. The output of a single encoder layer is denoted $\tilde{S}^{(l)} \in \mathbb{R}^{n \times d_t}$, where $l$ refers to the layer index. Stacking $L_e$ such layers produces the final encoder output $\tilde{S} \in \mathbb{R}^{n \times d_t}$, computed as:

$$\tilde{S} = \text{TransformerEncoder}(X^{\text{emb}}, M_{\text{causal}}) = \text{EncoderLayer}_{L_e}(\dots(\text{EncoderLayer}_1(X^{\text{emb}}, M_{\text{causal}}))\dots).$$

### 2.4.6 Vision Transformer

The *Vision Transformer* [8] adapts the transformer architecture—originally designed for sequential data—to image classification and related computer vision tasks. The core idea is to treat an image as a sequence of fixed-size patches, analogous to tokens in a language model.

Given an input image, it is first divided into a grid of $n$ non-overlapping patches. These patches are then flattened into vectors, forming a patch sequence. Each patch vector is linearly embedded. As in the standard transformer, positional information is added to the patch embeddings. In our case, we use a simplified variant of the Vision Transformer [9], which employs a fixed 2D sinusoidal positional encoding that respects the original 2D spatial layout of the image patches.

The resulting embedded sequence is passed through a standard transformer encoder, identical in structure to that described in subsection 2.4.5, but typically without causal masking, as all tokens can attend each other. The simplified Vision Transformer then performs global average pooling over the final layer's patch embeddings to produce the final image representation.

# Chapter 3

# The TransZero Algorithm

In section 3.1, we describe how the *TransZero* agent is created by replacing the dynamics network with a transformer encoder. Then, in section 3.2, we introduce *TransZero with MVC*, which uses the Mean Variance Constrained (MVC) evaluator to decouple tree construction from node evaluation. Finally, in section 3.3, we present *TransZero-Parallel*, which expands multiple nodes in parallel during MCTS, substantially accelerating the planning process.

## 3.1 TransZero

We create *TransZero* by replacing the dynamics network $g_\theta$ in MuZero with a transformer-based network, denoted $g_\theta^{\text{trans}}$. This modification enables the model to generate a new latent state $\tilde{s}_n$ from an initial latent state $\tilde{s}_{\text{root}}$ and the sequence of actions that leads to the node $x_n$ associated with $\tilde{s}_n$. A figure illustrating the difference in the *expansion* stage when using this new dynamics network is shown in Figure 3.1. In describing the process, we omit the batch dimension for simplicity, i.e., the shapes are shown as if $B = 1$.

In subsection 3.1.1, we describe how the action sequence and root latent state are tokenized and embedded to form the input to the transformer dynamics network. subsection 3.1.2 explains how this network generates latent states using self-attention. Finally, in subsection 3.1.3, we show how this setup enables parallel computation of the loss.

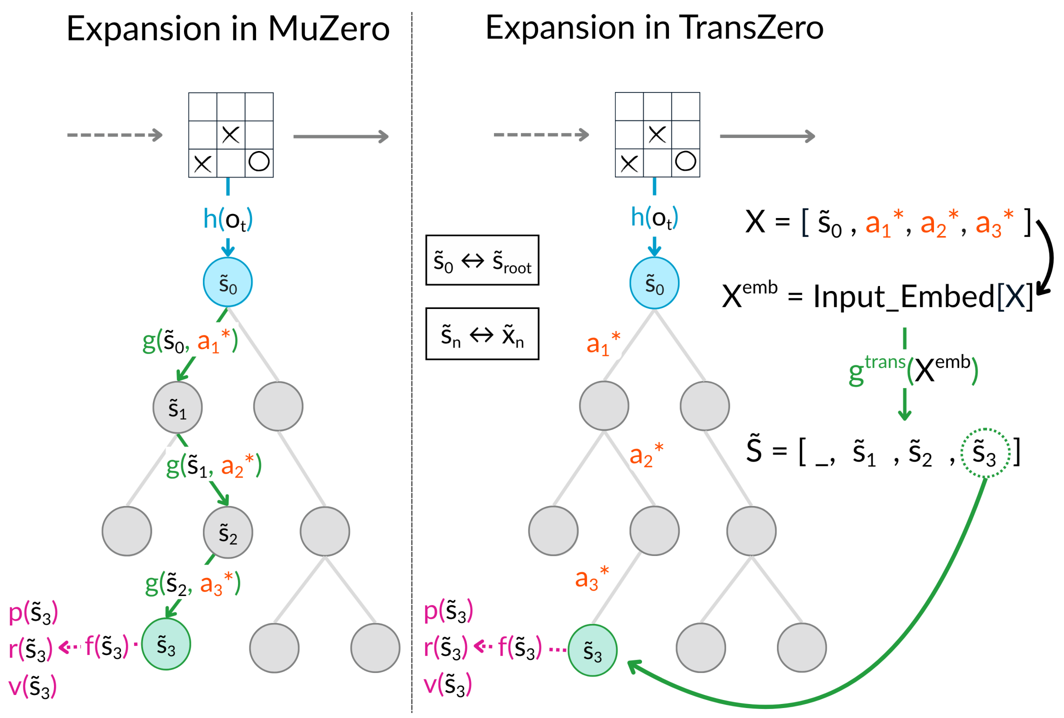


Figure 3.1: Comparison of MCTS in TransZero and MuZero. The expansion step in MuZero is the same as shown in Figure 2.1, and the backup and selection steps remain unchanged in TransZero. In TransZero, the new dynamics network $g_\theta^{\text{trans}}$ (green arrows) generates latent states $\tilde{s}_1$, $\tilde{s}_2$, and $\tilde{s}_3$ using the initial latent state $\tilde{s}_{\text{root}}$ (produced by the representation network) and the sequence of selected actions (orange) that leads to the new node $x_3$. The prediction network $f_\theta$ (pink arrow) operates as in the original MuZero.

### 3.1.1 Tokenization of Observations and Actions

We begin by defining our sequence of tokens $X$ as:

$$X = [\tilde{s}_{\text{root}} \parallel \mathbf{a}_{\tilde{s}_{\text{root}} \to x_n}] \in \mathbb{R}^{n \times d_t}.$$

Here, $\mathbf{a}_{\tilde{s}_{\text{root}} \to x_n}$ denotes the ordered sequence of actions leading from the initial latent root state $\tilde{s}_{\text{root}}$ to the target node $x_n$ that is to be expanded. The operator $\parallel$ represents the concatenation of one element or a list with another list.

The latent root state $\tilde{s}_{\text{root}}$ is obtained using the representation network $h_\theta$, as previously described by Equation 2.1. Since $\tilde{s}_{\text{root}}$ is already a vector in $\mathbb{R}^{d_t}$, it does not require further embedding. Each action in the sequence is embedded using a learnable action embedding layer and augmented with sinusoidal positional encoding to form the embedded token sequence $X^{\text{emb}}$:

$$X^{\text{emb}} = [\ \tilde{s}_{\text{root}} \parallel \text{Input\_Embed}(X_{1:n})\ ].$$

We use Python-style slice notation $X_{1:n}$, which selects the subsequence from index 1 up to (but not including) index $n$.

### 3.1.2 The Transformer Dynamics Network

We define the new dynamics network $g_\theta^{\text{trans}}$ to operate as TransformerEncoder$(\cdot)$ and apply it to the embedded input sequence $X^{\text{emb}}$ as follows:

$$\tilde{S} = g_\theta^{\text{trans}}(X^{\text{emb}}, M_{\text{causal}}) \in \mathbb{R}^{n \times d_t},$$

where $\tilde{S} = (\tilde{s}_0, \tilde{s}_1, \ldots, \tilde{s}_n)$, i.e., a sequence of latent states. The causal mask $M_{\text{causal}}$ is applied to ensure that future actions do not influence past ones.

To obtain predictions for $\tilde{s}_n$—the latent state corresponding to the node being expanded—we apply the prediction network $f_\theta$ to $\tilde{s}_n$. This follows the standard expansion process described in section 2.2.1. The backup process also remains unchanged from the description in section 2.2.1.

An implementation detail is that we set the first latent state in $\tilde{S}$ to $\tilde{s}_{\text{root}}$, which means that the predictions are made using the output of the representation network rather than the first latent state produced by the dynamics network.

### 3.1.3 Parallelizing the Loss Function

In standard MuZero, each node in the MCTS stores a cached latent state, which means that the approach described in subsection 3.1.2 does not provide direct computational benefits during planning. However, it does enable parallelization of the loss computation across the unroll steps $K$. In the conventional setup, each new latent state is computed sequentially, with each step depending on the previous one. This requires $K$ sequential forward passes to produce the full sequence of latent states. In contrast, TransZero processes the entire action sequence in a single forward pass, producing all latent states $\tilde{S}_{0:K}$ simultaneously for the full unroll.

## 3.2 TransZero with MVC

In this section, we describe the concrete steps taken to eliminate reliance on visitation counts when constructing the MCTS. We refer to the resulting agent as *TransZero with MVC* since we only evaluate it in combination with TransZero. However, it can also be used with standard MuZero without requiring any additional modifications.

We begin by explaining how to compute the Q-value estimate and its variance in subsection 3.2.1, which allows us to use $\tilde{\pi}_{\text{MVC}}$ as the action selection policy. In subsection 3.2.2, we discuss how the UCB score can be computed using the new Q-value estimate and its variance. Next, subsection 3.2.3 introduces updated value and policy targets. Finally, we describe a new caching strategy that accelerates the process in subsection 3.2.4.

### 3.2.1 Calculation of the Q-Value Estimate and its Variance

Although Jaldevik [5] developed this method for AlphaZero, we can largely retain their approach to calculating both $\mathbb{V}[Q_{\tilde{\pi}}(x)]$ and $Q_{\tilde{\pi}}(x)$. In these computations, $\tilde{\pi}$ refers to $\tilde{\pi}_{\text{MVC}}$, and we assume the following constraint:

$$\tilde{\pi}(x,a) = 0 \quad \forall\, a \in \mathcal{A} \quad \text{if leaf}(x).$$

If $x$ is not a leaf, $\tilde{\pi}(x,a)$ is calculated as described in Equation 2.4.

The Q-value estimate $Q_{\tilde{\pi}}$ is recursively computed as:

$$Q_{\tilde{\pi}}(x) = r(x) + \gamma \sum_{a \in \mathcal{A}_v} \tilde{\pi}(x,a) Q_{\tilde{\pi}}(x \uplus a),$$

and the variance as:

$$\mathbb{V}[Q_{\tilde{\pi}}(x)] = \mathbb{V}[r(x)] + \gamma^2 \cdot \left(\tilde{\pi}(x,\mathcal{A}_v)^\top \tilde{\pi}(x,\mathcal{A}_v)\right) \mathbb{V}[Q_{\tilde{\pi}}(x \uplus \mathcal{A}_v)].$$

The variance of the leaf is defined as:

$$\mathbb{V}[Q_{\tilde{\pi}}(x)] = \mathbb{V}[r(x)] + \gamma^2 \cdot \mathbb{V}[v(x)] \quad \text{if leaf}(x).$$

In a deterministic environment, the reward variance is zero. As for the variance of the value prediction $\mathbb{V}[v(x)]$, the original formulation sets this to $\frac{1}{N(x)}$ if the node $x$ is terminal, and to 1 otherwise. Since MuZero does not explicitly represent terminal nodes, we default to using a value variance of 1 for all nodes. This results in a variance of $\gamma^2$ at every leaf node.

### 3.2.2 PUCT Calculations using MVC

As described by Equation 2.2, PUCT is the sum of $Q(x \uplus a)$ and $U(x \uplus a)$. MVC already defines a new Q-value estimate in the form of $Q_{\tilde{\pi}}(x \uplus a)$, so we replace $Q(x \uplus a)$ with that. Since the calculation of $U(x \uplus a)$ is dependent on visitation counts, it also needs to be altered for our implementation. The proposal from Jaldevik [5] was to replace $N(x)$ with $\mathbb{V}[Q_{\tilde{\pi}}(x)]$ since it was proved that the variance of the average return is inversely proportional to the visitation count. This leads to the formulation

$$U_{\tilde{\pi}}(x \uplus a) = C_{puct} \cdot p(x,a) \cdot \frac{\sqrt{\mathbb{V}[Q_{\tilde{\pi}}(x)]^{-1}}}{1 + \mathbb{V}[Q_{\tilde{\pi}}(x \uplus a)]^{-1}}.$$

### 3.2.3 Target Computation using MVC

The policy and value networks are trained using data generated by the visitation count policy. However, since the agent's final policy is modified, the corresponding learning targets must also be updated. Specifically, the target for the policy network should be the policy derived from the tree evaluator, $\tilde{\pi}_{\text{MVC}}$, rather than the one based on visitation counts. In the original formulation by Jaldevik [5], the value target $\hat{V}(x)$ is replaced by the predicted value $v(x)$. However, we argue that a more accurate target would be $\hat{V}_{\tilde{\pi}}(x)$, as it better reflects the estimated value under the same policy used during planning.

### 3.2.4 Extended Caching

In Jaldevik [5], the authors computed and cached $Q_{\tilde{\pi}}(x)$ and $\mathbb{V}[Q_{\tilde{\pi}}(x)]$ during action selection. These values were then discarded during the backup phase for every node encountered, since value estimates propagate up the search tree. The affected nodes are the parents of the new leaf node, those with dashed orange arrows that pass through them in Figure 2.1 (backup).

This caching mechanism effectively reduced computation. However, the initial implementation of TransZero with MVC still exhibited approximately a 6.5x slowdown in wall-clock time compared to TransZero. To address this, we introduced two key improvements that resulted in a 3.6x speedup relative to the original implementation. This reduced the overall slowdown to 1.8x compared to the visitation count-based approach.

Our first improvement was to recalculate $Q_{\tilde{\pi}}(x)$ and $\mathbb{V}[Q_{\tilde{\pi}}(x)]$ *during* the backup phase instead of discarding them. This ensured that the depth of the search tree was traversed only once during each simulation. Previously, the Q-value estimates and variances were discarded during backup, and recalculating them later during action-selection required a separate pass down to the newly introduced leaf node. Note that this recalculation applies *only* to nodes where the Q-values and variances were discarded, not to the entire search tree.

Our second improvement was to cache each child's Q-value estimate and variance directly at its parent node. When a node is initialized, the estimated Q-values of its children are set to 0, and the variances to $\gamma^2$. During backpropagation, each child updates its corresponding entry at the parent with its newly calculated Q-value and variance. This caching avoids the need to initialize and populate an array by looping over the children to retrieve their values when computing $\tilde{\pi}(x,a)$.

## 3.3 TransZero-Parallel

By using a transformer, we can evaluate an entire action sequence in a single forward pass, rather than unrolling it one action at a time. Additionally, the use of the MVC evaluator decouples tree construction from node evaluation. Together, these two features—combined with an alternative strategy for selection, generating action sequences, and masking, enable the parallel evaluation of entire subtrees during planning. Although we expand a full subtree during the expansion phase of MCTS, we still consider the combination of selection, expansion, and backup a single simulation.

We refer to this agent as *TransZero-Parallel*. A visualization of the selection and backup processes in this setting is shown in Figure 3.2, while expansion is shown in Figure 3.3.

In subsection 3.3.1, we describe how to construct action sequences that represent subtrees to be expanded in parallel. Next, in subsection 3.3.2 we introduce a causal tree mask necessary for parallel expansion. In subsection 3.3.3, we explain how the dynamics and prediction networks process these sequences in parallel to generate value, reward, and policy estimates for entire subtrees. Finally, subsection 3.3.4 discusses how subtrees are stored and processed efficiently during expansion and backup.

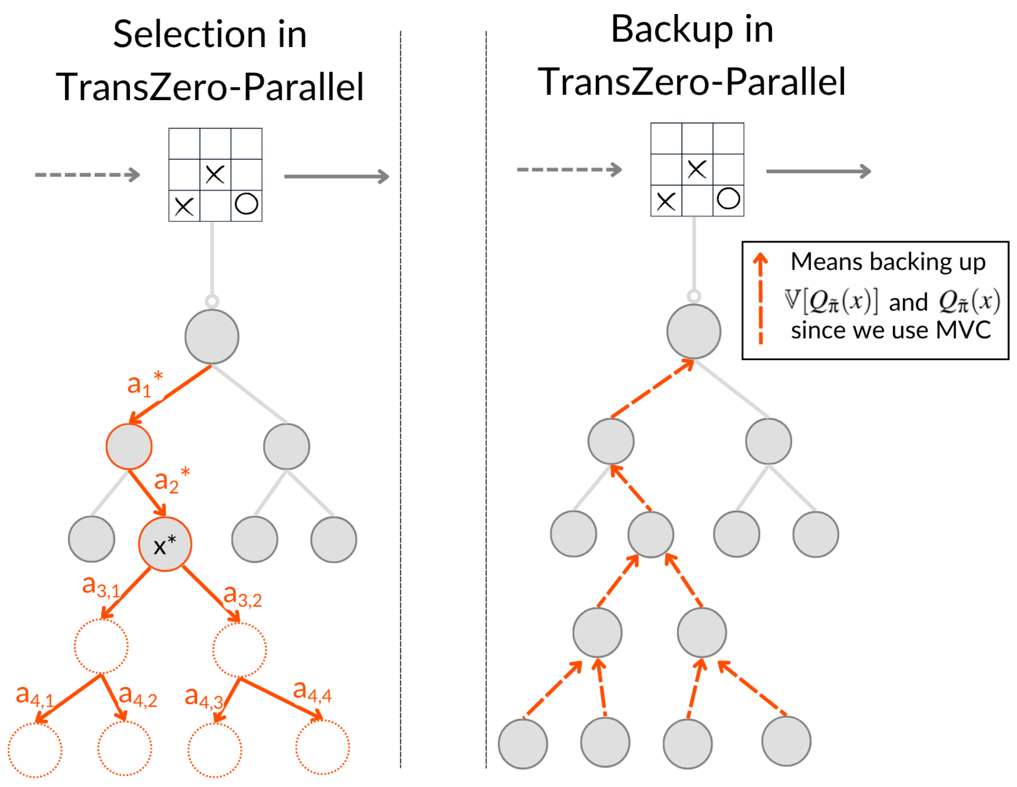

Selection in TransZero-Parallel
$a_1{}^*$
$a_2{}^*$
$x^*$
$a_{3,1}$
$a_{3,2}$
$a_{4,1}$
$a_{4,2}$
$a_{4,3}$
$a_{4,4}$
Backup in TransZero-Parallel
Means backing up $\mathbb{V}[Q_{\tilde{\pi}}(x)]$ and $Q_{\tilde{\pi}}(x)$ since we use MVC

Figure 3.2: Selection and backup for MCTS in TransZero-Parallel. Expansion is shown in Figure 3.3. The selection step in MuZero is roughly the same as shown in Figure 2.1, except that we now select the entire subtree under $x^*$ instead of the most promising child node. The backup is also the same except we now backup values relevant to the MVC-calculations.

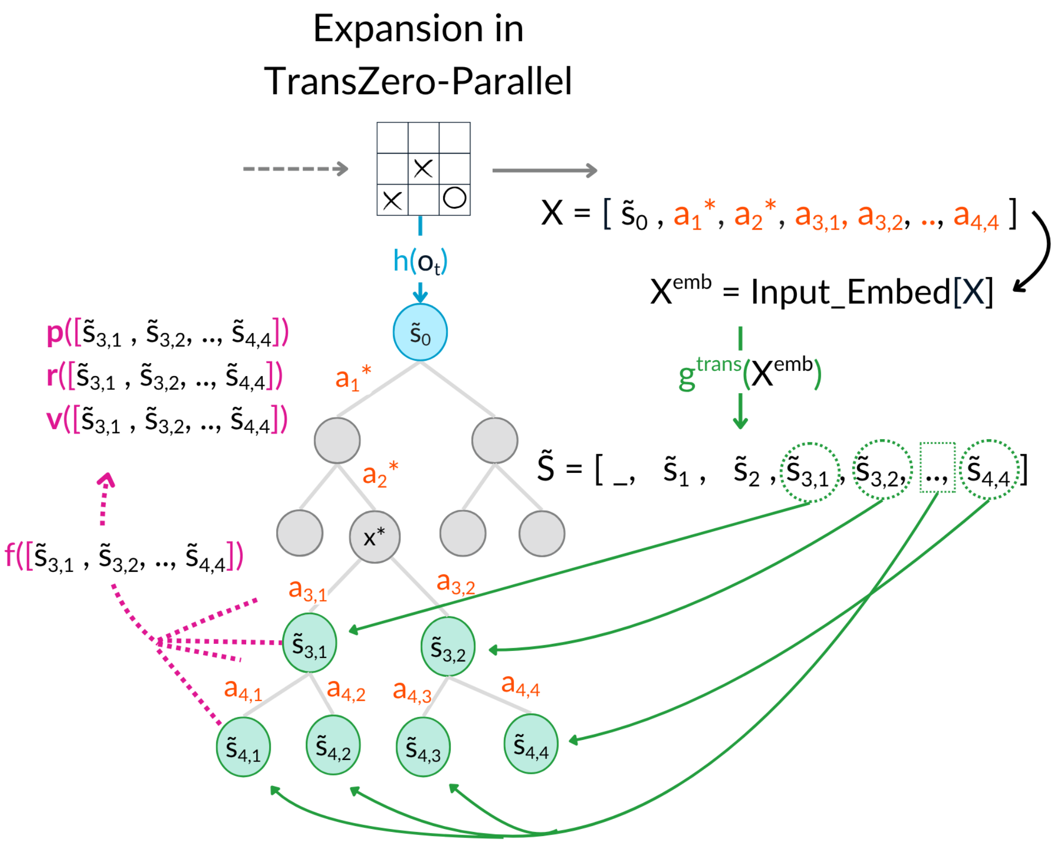

Expansion in TransZero-Parallel
h(o_t)
s̃0
X = [ s̃0 , a1*, a2*, a3,1, a3,2, .., a4,4 ]
X^emb = Input_Embed[X]
g^trans(X^emb)
S̃ = [ _, s̃1 , s̃2 , s̃3,1 , s̃3,2, .., s̃4,4 ]
p([s̃3,1 , s̃3,2, .., s̃4,4])
r([s̃3,1 , s̃3,2, .., s̃4,4])
v([s̃3,1 , s̃3,2, .., s̃4,4])
f([s̃3,1 , s̃3,2, .., s̃4,4])
a1*
a2*
x*
a3,1
a3,2
s̃3,1
s̃3,2
a4,1
a4,2
a4,3
a4,4
s̃4,1
s̃4,2
s̃4,3
s̃4,4

Figure 3.3: One expansion for MCTS in TransZero-Parallel. Backup and selection are shown in Figure 3.2. The expansion step in MuZero is the same as shown in Figure 2.1, except that we now expand the entire subtree under $x^*$ as well. We use the same dynamics network $g_\theta^{\text{trans}}$ as in TransZero (green arrows) to generate all latent states $\tilde{s}_1$ until $\tilde{s}_{4,4}$ using the initial latent state $\tilde{s}_{\text{root}}$ and the sequence of selected actions (orange) leading to each node. The prediction network $f_\theta$ (pink arrows) operates as in the original MuZero, but now processes all latent states in a single batch, outputting a corresponding batch of values, rewards, and policy priors.

### 3.3.1 Action Sequence Construction for Trees

As in standard MCTS, we traverse the search tree by repeatedly selecting the child node with the highest PUCT score. However, instead of expanding only the most promising child of $x^*$ (i.e., $x_{n-1}$), we expand an entire subtree rooted at $x^*$. Specifically, we aim to construct and evaluate the latent states of all nodes in $\mathcal{T}_{x^*}$. The number of tree-layers $N_l$ in $\mathcal{T}_{x^*}$ is a tunable hyperparameter, where a *tree-layer* refers to all nodes at the same depth relative to $x^*$.

To construct the latent states for the entire subtree, we first build the token sequence $X$. This sequence begins with the latent root state $\tilde{s}_{\text{root}}$, as in previous sections. Next, we append

$\mathbf{a}_{\tilde{s}_{\text{root}} \to x^*}$, which denotes the ordered sequence of actions leading from the root state $\tilde{s}_{\text{root}}$ to the selected node $x^*$. Finally, we collect the sequences of actions required to reach every node within $\mathcal{T}_{x^*}$ and concatenate them into a list, denoted $\mathbf{a}_{\mathcal{T}}$.

To obtain $\mathbf{a}_{\mathcal{T}}$, we traverse the subtree in a breadth-first manner and add every encountered action to a list. These actions correspond to the actions $a$ without the $^*$ marker in Figure 3.1. In practice, this results in a sequence containing elements from the action space $\mathcal{A}$ repeated $R_l = \sum_{i=1}^{N_l} |\mathcal{A}|^i$ times. This occurs because to reach every node in the first sublayer, we must take all actions in $\mathcal{A}$ from the root of the subtree, and this pattern continues recursively for each layer.

We now formally define the full token sequence $X$ as:

$$X = [\tilde{s}_{\text{root}} \parallel \mathbf{a}_{\tilde{s}_0 \to x^*} \parallel \mathbf{a}_{\mathcal{T}}] \,.$$

The embedded token sequence is then, just like in subsection 3.1.1, defined as:

$$X^{\text{emb}} = [\ \tilde{s}_{\text{root}} \parallel \text{Input\_Embed}(X_{1:n})\ ].$$

### 3.3.2 Causal Tree Masking

Since $X$ is constructed via breadth-first traversal, its ordering naturally ensures that actions associated with nodes in deeper tree-layers appear later in the sequence than those from shallower layers. However, we must also enforce that an action can only attend its ancestors, not to other unrelated nodes at the same or shallower depth. For example, in Figure 3.3, if we consider the action $a_{4,2}$, it should only be able to attend to $a_{3,1}$, $a_2^*$, $a_1^*$, and $\tilde{s}_{\text{root}}$. It must *not* attend to sibling actions such as $a_{4,1}$, even if they appear earlier in the sequence $X$.

To enforce this constraint, we define a *causal tree mask* $M_{\text{tree\_causal}} \in \{0,1\}^{L_X \times L_X}$, where $L_X = |X|$ is the length of the token sequence. The mask is defined as:

$$M_{\text{tree\_causal}\, ij} = \begin{cases} 1 & \text{if } x_j \in \mathbf{a}_{\tilde{s}_{\text{root}} \to x_i}, \\ 0 & \text{otherwise.} \end{cases}$$

### 3.3.3 Parallel Expansion

To obtain predictions for multiple nodes in parallel, we first feed the embedded token sequence $X^{\text{emb}}$ through the TransZero dynamics network $g_\theta^{\text{trans}}$:

$$\tilde{S} = g_\theta^{\text{trans}}(X^{\text{emb}}, M_{\text{tree\_causal}}) \in \mathbb{R}^{L_X \times d_t}.$$

We then extract the last $N(x^*)$ latent states from $\tilde{S}$, which correspond to the nodes in the subtree $\mathcal{T}_{x^*}$. Technically, this number should be $N(x^*) - 1$ since we should not include the subtree-root $x^*$, but for simplicity, we use $N(x^*)$. Formally, the subsequence of latent state is the slice:

$$\tilde{S}_{\mathcal{T}} = \tilde{S}_{L_X - N(x^*) : L_X} \,.$$

We pass this batch of latent states through the prediction network $f_\theta$ to obtain rewards, values, and policy priors for all nodes in the subtree:

$$(\mathbf{v}_{\mathcal{T}}, \mathbf{r}_{\mathcal{T}}, \mathbf{p}_{\mathcal{T}}) = f_\theta(\tilde{S}_{\mathcal{T}}) \in \mathbb{R}^{N(x^*) \times d_t},$$

where $\mathbf{v}_{\mathcal{T}}$, $\mathbf{r}_{\mathcal{T}}$, and $\mathbf{p}_{\mathcal{T}}$ denote the predicted values, rewards, and policy priors for each node in the subtree $\mathcal{T}$. Although we expand an entire subtree in parallel, we will still call this process a single simulation.

### 3.3.4 Storing Subtrees as Nodes

Implementation-wise, we introduce a few additional changes to how nodes are represented within the MCTS. In the original setup, each node was stored individually in a tree structure. Here, we instead store each subtree as a flat list. Because each subtree is fully expanded, we can index nodes directly and compute parent or child indices trivially. This change significantly accelerates both the expansion and the backup phases.

In the classical setup, expanding a subtree of size $N(x^*)$ would require $N(x^*)$ separate node lookups and assignments. With our new representation, we can assign the vectors $\mathbf{v}_{\mathcal{T}}$, $\mathbf{r}_{\mathcal{T}}$, and $\mathbf{p}_{\mathcal{T}}$ to the entire list in a single operation, eliminating the need for individual lookups and repeated assignments.

For the backup phase, we process the subtree in bottom-up order, since each node's estimated Q-value and variance depend only on its children. Because nodes at the same depth are independent of one another, we can calculate these values of all nodes within a tree-layer in parallel on the GPU. As a result, the full backup procedure runs in

$$O(N_l),$$

instead of

$$N(x^*) = \sum_{i=0}^{N_l} |\mathcal{A}|^i = \frac{1 - |\mathcal{A}|^{N_l+1}}{1 - |\mathcal{A}|} \in O(|\mathcal{A}|^{N_l}).$$

This is under the assumption that parallel computation over a list is as fast as processing a single node sequentially. In subsection 6.3.3 this is shown to hold true for the environments we tested.

# Chapter 4

# Related Work

We begin this chapter by discussing how transformers have been used in model-based reinforcement learning (MBRL) in section 4.1. Then, in section 4.2, we examine how transformers have been applied in both MuZero [4] and AlphaZero [3]. Finally, in section 4.3, we review other efforts to parallelize Monte Carlo Tree Search (MCTS).

## 4.1 Model-based Reinforcement Learning with Transformers

There have been several efforts to incorporate transformers into model-based reinforcement learning (MBRL). One notable example is TransDreamer [10], which we discuss in subsection 4.1.1. It replaces the recurrent component in the original Dreamer architecture [11] with a transformer-based design. Another example is IRIS (Imagination-based Reinforcement Learning with transformers) [12], which trains a world model using an autoencoder and a transformer. This work is covered in more detail in subsection 4.1.2.

### 4.1.1 Enhancing Dreamer with Transformer-Based World Models

Dreamer [11] follows three main steps that repeat until convergence: learning a world model, training a policy, and interacting with the environment. First, it learns a dynamics model of the environment and a reward function using data stored in an experience replay buffer. Then, it trains an actor-critic policy using latent trajectories—simulated rollouts generated with the learned world model. Finally, the agent runs the trained policy in the real environment to collect new data and update the replay buffer.

TransDreamer [10] replaces the recurrent component in the original Dreamer architecture with a transformer-based model, referred to as a Transformer State-Space Model (TSSM). Instead of relying on recurrence to maintain memory over time, the transformer processes entire sequences of embedded past states and actions to capture temporal dependencies and predict future states.

TransDreamer performs comparably to Dreamer on Atari tasks that do not require long-term memory, and it outperforms Dreamer on 2D and 3D navigation tasks that demand long-term, complex memory interactions. This demonstrates that, in this context, transformers are better suited for modeling long-range dependencies and complex temporal patterns—especially in tasks where recurrent networks struggle to retain sufficient context.

### 4.1.2 Transformers are Sample-Efficient World Models

IRIS (Imagination-based Reinforcement Learning with Transformers) [12] is a model-based reinforcement learning method that combines a discrete latent representation with an autoregressive transformer to model environment dynamics. While similar in motivation to Dreamer, it employs a different architectural approach.

The model uses a discrete variational autoencoder to encode observations into sequences of discrete tokens. A transformer is then trained to autoregressively model transitions in this latent space, predicting rewards, terminal signals, and tokens of the next observation, conditioned on past observations and actions. It also uses a decoder to decode tokens representing a future observation, from which the policy then predicts the next action. This setup enables the agent to simulate long sequences in the latent space and optimize its policy through latent rollouts.

IRIS was evaluated on the Atari 100k benchmark, where it achieved a mean human-normalized score of 1.046 and exceeded human-level performance in 10 of 26 games. Notably, these results were obtained without explicit planning or lookahead search, highlighting the ability of transformer-based world models to capture useful dynamics directly through sequence modeling.

## 4.2 Transformers in AlphaZero and MuZero

Some works have explored the use of transformers in AlphaZero and MuZero. Chessformer [13], which we describe in subsection 4.2.1, replaces the convolutional neural networks (CNNs) in the representation network with a transformer, using a specialized attention mechanism to handle 2D inputs. In subsection 4.2.2, we describe the model proposed by Pu et al. [14], called UniZero, which uses a transformer as the backbone in MuZero to provide contextual support for the dynamics and prediction networks.

### 4.2.1 Chessformer

Chessformer [13] replaces the CNN in the representation network with a transformer, as traditional CNNs often struggle to capture the long-range dependencies inherent in chess due to their limited receptive fields. Transformers, with their global self-attention mechanisms, address this limitation by allowing the model to consider all parts of the input simultaneously.

The architecture uses an encoder-only transformer model with a context length of 64 tokens, corresponding to the 64 squares on a chessboard. The model employs the relative position encoding scheme proposed by Shaw et al. [15], which incorporates information about the spatial relationships between pieces. This enhances the model's ability to capture the dynamics of the board. The transformer's output is passed through separate policy and value heads to produce the corresponding predictions. Unlike MuZero, AlphaZero does not predict rewards; instead, it receives these signals directly from the environment.

Results show that Chessformer outperforms previous models. Notably, it surpasses AlphaZero in both playing strength and puzzle-solving ability while requiring significantly fewer computational resources.

### 4.2.2 UniZero: Generalized and Efficient Planning with Scalable Latent World Models

UniZero [14] puts all previous observations and actions through a *transformer backbone* to create the latent root state of MCTS. It is similar to TransZero in that it creates tokens from observations and actions, although it does not use the transformer to create future latent states. It is feasible to combine aspects of this approach with TransZero, largely in the manner described in subsection 7.3.4.

The motivation of UniZero is that the authors identify two core limitations in MuZero-style architectures that hinder scalability and performance. First, the recurrent structure entangles latent representations with historical information. Second, MuZero under-utilizes available trajectory data during training by relying heavily on initial observations and recursive latent predictions. Together, these issues reduce data efficiency and impair the model's ability to scale to complex, long-horizon, or multitask environments.

To address these limitations, UniZero introduces a transformer-based latent world model that decouples memory from state representation and fully leverages available trajectory data. At each timestep $t$, UniZero encodes the current observation $o_t$ into a latent state $\tilde{s}_t$, and the action $a_t$ into an action embedding. These, along with all previous latent states and actions $(\tilde{s}_{0:t}, a_{0:t})$, are inputted into a transformer encoder, which constructs an implicit latent history $h_t$. This history captures long-term temporal context and enables UniZero to condition its predictions on the full sequence of past experiences.

The architecture consists of four main components:

- **Encoders** $h_\theta$: Encode observations and actions into latent states and embeddings. The observation encoder is what we call a representation network and denote $h_\theta$, the action encoder we have denoted as $\text{Input\_Embed}(\cdot)$.
- **Transformer Backbone**: Aggregates all past latent-action pairs to form the context-rich history $h_t$.
- **Dynamics Head** $g_\theta$: Predicts the next latent state $\tilde{s}_{t+1}$ and reward $r(\tilde{s}_t)$, conditioned on the full latent-action history.

- **Prediction Head** $f_\theta$: Outputs the policy $p(\tilde{s}_t)$ and value $v(\tilde{s}_t)$ based on the same history (excluding the current action).

This setup explicitly *decouples* the current latent state $\tilde{s}_t$ from memory, which is captured in the transformer-based history $h_t$. Unlike MuZero, which encodes only the initial observation of a trajectory (even if that observation is a stack of frames), UniZero utilizes *all* observations and actions up to time $t$ during training. This allows it to model long-term dependencies more effectively.

The experimental results show that UniZero consistently outperforms the baseline methods in various environments. It shows strong performance in both discrete and continuous control tasks, handles single-task and multitask learning effectively, and excels in environments that require long-term memory.

## 4.3 Parallelizing MCTS

In Chaslot et al. [16], several strategies were proposed to parallelize MCTS: *leaf*, *root*, and *tree* parallelization. Their approach differs from ours in that it creates multiple independent instances of MCTS and later combines their results, rather than constructing a single MCTS tree in parallel. As such, their method is not mutually exclusive with TransZero-Parallel; in principle, many of the same techniques could be incorporated into our framework.

**Leaf parallelization** is the simplest approach. A single thread expands the tree to a leaf node, after which multiple threads perform simulations from that point. Once all simulations are complete, one thread updates the tree. This method is easy to implement and works well on distributed systems, but since simulations do not share information and may vary in length, the performance gains are often limited.

A simulation in pure MCTS is not quite the same as a network evaluation as is used in MCTS for MuZero. Instead, a simulation involves selecting actions according to a policy, traversing the search tree, and estimating the value of the resulting state, often using a rollout to a terminal state. Unlike MuZero, this process interacts directly with the environment instead of relying on a learned model.

**Root parallelization** creates a separate MCTS tree for each thread. Once the search time is exhausted, the results from all trees are merged at the root. Since threads do not need to share memory, this method avoids coordination overhead. It also helps prevent convergence to local optima by exploring multiple independent search paths and often performs better than running a single, longer search.

**Tree parallelization** is more complex. Here, several simultaneous games are played from a shared tree. This differs from leaf parallelization, where all simulated games start from the same leaf node. Since all threads operate on a shared tree, they must coordinate access to avoid data corruption. This coordination is managed using *mutexes* (mutual exclusions), which are locks that control access to parts of the tree.

The tree parallelization approach comes closest to fully constructing the tree in parallel. However, it cannot roll out an entire sequence of the search tree in parallel. We also theorize that significantly increasing the number of threads could undermine the usefulness of visitation counts and lead to coordination bottlenecks. Therefore, unlike our method, it still imposes some sequential constraints.

Experiments show that root parallelization yields the best results, achieving a 14.9x speedup with 16 threads. Tree parallelization, using local mutexes and virtual loss, reaches 8.5x, while leaf parallelization performs worst at 2.4x.

# Chapter 5

# Experimental Setup

In this chapter, we begin by discussing the environments used in section 5.1. Then, in section 5.2, we describe the different agents used in the experiments. Lastly, in section 5.3, we outline the configuration of the experiments conducted.

## 5.1 Environments

The experiments are conducted in two distinct environments: MiniGrid[1] that we will describe in subsection 5.1.1 and LunarLander[2] which we will describe in subsection 5.1.2. Each environment was selected to evaluate different aspects of the agents' behavior and performance. In subsection 5.1.3, we outline the requirements to solve each environment.

### 5.1.1 MiniGrid

MiniGrid was selected for its simplicity and flexibility. It allows for easy scaling of complexity through modifications such as grid size, number of obstacles, and randomness. Our implementation generates a new **random** map at each episode, which will be further detailed below.

Our specific MiniGrid implementation includes the following dynamics:

- Lava tiles reset the episode with a reward of 0.
- The goal tile rewards the agent between 5 and 10 points, depending on the optimality of the path taken.
- Episodes end with no reward if the agent times out, i.e., takes more than the maximum allowed number of steps.

[1]`https://github.com/Farama-Foundation/Minigrid`
[2]`https://github.com/Farama-Foundation/Gymnasium`

We implemented three distinct MiniGrid maps. A random example of each is shown in Figure 5.1.

| **Environment** | **Lava Tiles** | **Timeout (steps)** | **Figure** |
|---|---|---|---|
| $3 \times 3$ map | 2 | 15 | 5.1 (a) |
| $4 \times 4$ map | 3 | 18 | 5.1 (b) |
| $5 \times 5$ map | 4 | 25 | 5.1 (c) |

Table 5.1: MiniGrid environment details

Each map initializes the agent's position and orientation randomly, while the goal is always located in the bottom-right corner. Lava tiles are placed randomly, but the environment is guaranteed to include at least one viable path from the agent's starting position to the goal. The agent's perspective is top-down, and it can take one of three actions: turn left, turn right, or move forward. The environment is represented using one-hot encoding. Specifically, there is one channel for lava, one for the goal, and four separate channels for the agent's possible orientations. At any given time, the agent occupies coordinates $(x,y)$ in exactly one of the orientation channels, while the remaining three contain no data at that position.

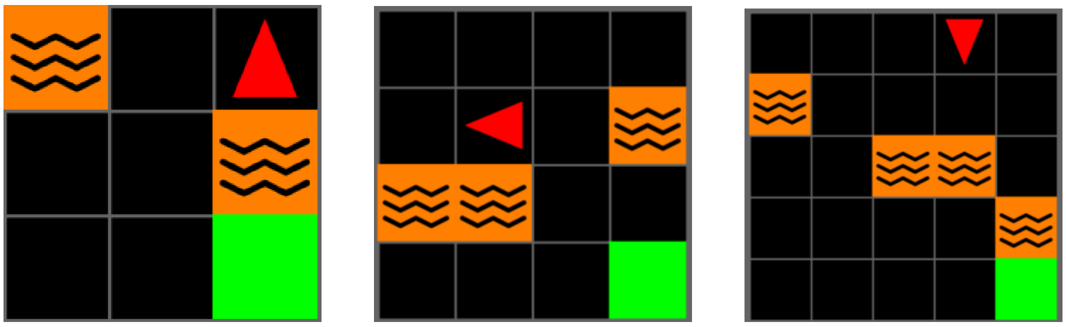


Figure 5.1: A random instance of each MiniGrid environment used to evaluate the different agents: (a) $3 \times 3$ grid, (b) $4 \times 4$ grid, (c) $5 \times 5$ grid.

### 5.1.2 LunarLander

LunarLander provides an interesting testing environment due to its dynamics, which are comparable to those found in Atari games. However, it also differs significantly from MiniGrid in several ways: it has a one-dimensional observation space, a continuous state space, and greater complexity overall.

The objective is to safely land the vehicle between two designated flags while conserving fuel. The available actions are: fire the left orientation engine, fire the right orientation engine, fire the main engine, or do nothing. Key dynamic factors include gravity, inertia, and fuel constraints, all of which increase the difficulty of landing maneuvers.

### 5.1.3 Solving the Environments

LunarLander is considered solved when an average score of 200 is reached, although scores exceeding this are achievable. In contrast, there is no official threshold for solving the MiniGrid environments, as these were custom-designed for this study. We therefore define an arbitrary target score of 7.5, corresponding to 75% of the maximum possible score of 10. This target strikes a balance between a perfect score of 10 and the minimum score required for success, which is 5. For both environments, we require the agent to achieve a rolling average above the target score over 50 consecutive episodes. While this threshold is also arbitrary, it ensures that performance is consistent and not the result of a few lucky episodes.

## 5.2 Agent Architectures

In subsection 5.2.1 and subsection 5.2.2, we describe two baseline agents based on the standard MuZero design. Then, in subsection 5.2.3, subsection 5.2.4, and subsection 5.2.5 we present the setup for TransZero, TransZero with MVC, and TransZero-Parallel, respectively.

### 5.2.1 MuZero Agent

The standard agent, referred to as *MuZero*, follows the architecture described in the original work by Schrittwieser et al. [4]. Our implementation is based on the *muzero-general* repository on GitHub[3].

The agent encodes the environment state using a convolutional network: a $3 \times 3$ convolution is used for grid-based environments, while a $1 \times 1$ convolution is used for the LunarLander environment due to its non-spatial (non-2D) input. This initial encoding is followed by a series of normalization layers and a sequence of residual blocks. Each residual block consists of two convolutional layers, each followed by batch normalization and activation functions between and after the layers. The resulting root latent state is normalized to lie within the range $[0, 1]$.

For the dynamics network, the one-hot encoded action is concatenated with the previous latent state. This combined input is passed through an architecture similar to the representation network, producing the next latent state, which is then normalized.

Predictions for reward, value, and policy priors are obtained by applying a $1 \times 1$ convolution to the latent state. The output is then flattened and processed by a multilayer perceptron (MLP) to generate the final predictions. The reward predictions are based on the unnormalized latent state, whereas value and policy prior predictions use the normalized version.

### 5.2.2 MuZero-FC Agent

The *MuZero-FC* agent uses a smaller and simpler architecture compared to the standard MuZero agent. Both the representation and dynamics networks are implemented as MLPs

[3]`https://github.com/werner-duvaud/muzero-general`

that operate on a flattened version of the input observation. Similarly, the prediction heads for reward, value, and policy priors are also MLPs. This design maintains the core structure of MuZero while significantly reducing architectural complexity.

### 5.2.3 TransZero Agent

The *TransZero* agent corresponds to the architecture described in section 3.1. We use a Vision Transformer, as described in subsection 2.4.6, for the MiniGrid environments since they are two-dimensional. However, for LunarLander, we use an MLP as the representation network, as we believe that a Vision Transformer provides negligible benefit for a small, flat observation like the one received from LunarLander.

### 5.2.4 TransZero with MVC Agent

This agent is referred to as *TransZero with MVC*, as it combines TransZero with the modifications required to support the MVC evaluator, detailed in section 3.2. We chose not to test the MVC evaluator with MuZero, as we did not expect it to result in significantly different behavior compared to its use with TransZero.

### 5.2.5 TransZero-Parallel Agent

Here we will describe the TransZero-Parallel agent. The workings of this agent are described in section 3.3. For MiniGrid, we used $N_l = 2$, which means that in each simulation two subtree layers of nodes were expanded. Using three layers failed to converge, while a single layer did converge but led to slower training. We also used 4 simulations per MCTS, resulting in a total of $1 + 12 \times 4 = 49$ nodes being expanded per search. This is nearly twice the number used by the other agents (25 nodes), but it required approximately six times fewer simulations.

For LunarLander, we used $N_l = 3$ for similar reasons: deeper trees failed to converge, and shallower ones were slower. In this case, we used 2 simulations per MCTS, which resulted in $1 + 84 \times 2 = 169$ nodes expanded per search. This is more than three times the number expanded by the other agents, but required only a twenty-fifth as many simulations.

## 5.3 Run Configurations

In the experiments, we let each agent run for an equal number of environment steps. To ensure fairness in wall-clock comparisons, all agents were run with identical configurations, including batch size, number of self-play workers, and other speed-related parameters. Each agent was run in the LunarLander environment for five different random seeds and on each MiniGrid map for ten seeds. Henceforth, when we refer to runs with different random seeds, we will simply say seeds.

All experiments were performed on an NVIDIA Tesla P100 GPU paired with an Intel Xeon CPU (4 vCPUs). For measuring the speed of a single action selection, we used an NVIDIA

GeForce RTX 4090 paired with an AMD EPYC 7542 CPU (64 vCPUs). We chose the RTX 4090 setup for this specific test because it better reflects the high-performance hardware commonly used in industry. The Tesla P100 system was selected for the main experiments because using the RTX 4090 system would have been prohibitively expensive.

See the associated GitHub repository (linked in the abstract) for all hyperparameter settings.

# Chapter 6

# Results

This chapter presents the results of our study. In section 6.1, we present the results for TransZero; in section 6.2, the results for TransZero with MVC; and in section 6.3, the results for TransZero-Parallel.

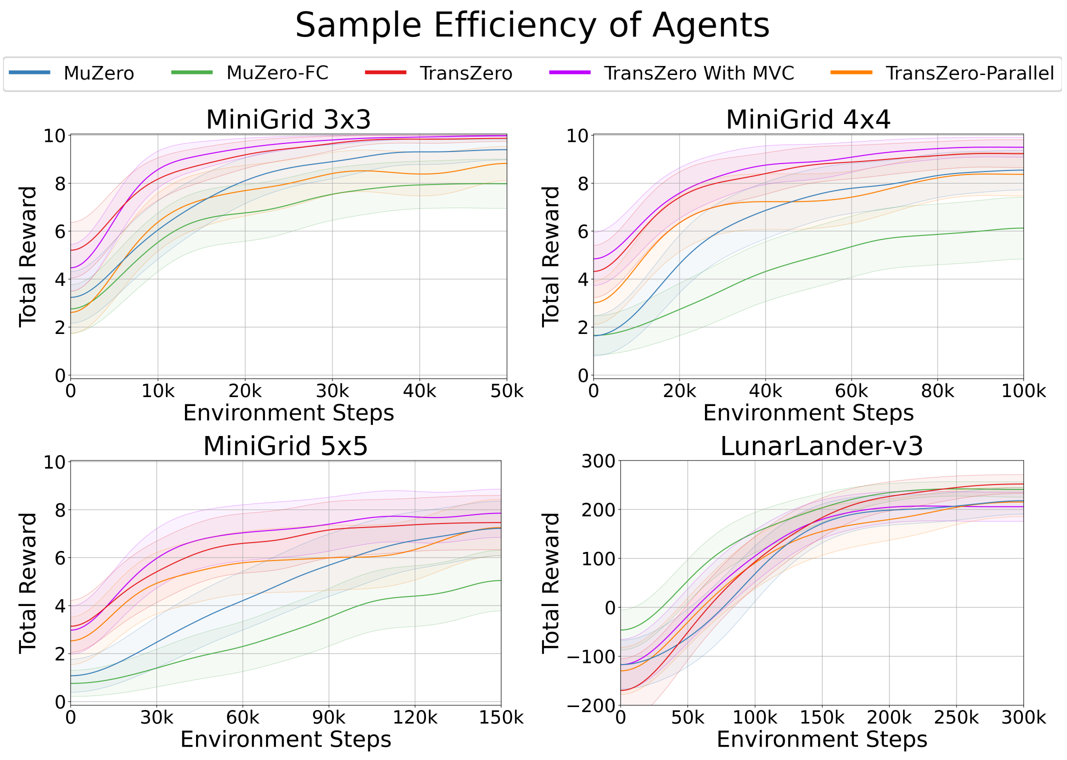
## 6.1 TransZero Results

In subsection 6.1.1 we demonstrate that TransZero is at least as sample-efficient as MuZero, and often performs better. In subsection 6.1.2, we show that TransZero is faster than MuZero in terms of wall-clock time, but generally slower than MuZero-FC.

### 6.1.1 TransZero Sample Efficiency

In Figure 6.2 (top), we show the reward obtained by each agent as a function of the number of environment steps taken. We observe that the sample efficiency of TransZero is not worse than that of MuZero in any of the evaluated environments. TransZero even performs slightly better during the initial training phase in all environments. Toward the end, it shows better average performance, although the differences fall within the standard error. To establish statistical significance, additional seeds would be required.

TransZero outperforms MuZero-FC across all MiniGrid environments. In the LunarLander environment, MuZero-FC performs slightly better than TransZero during the first half of training but reaches a similar performance toward the end.



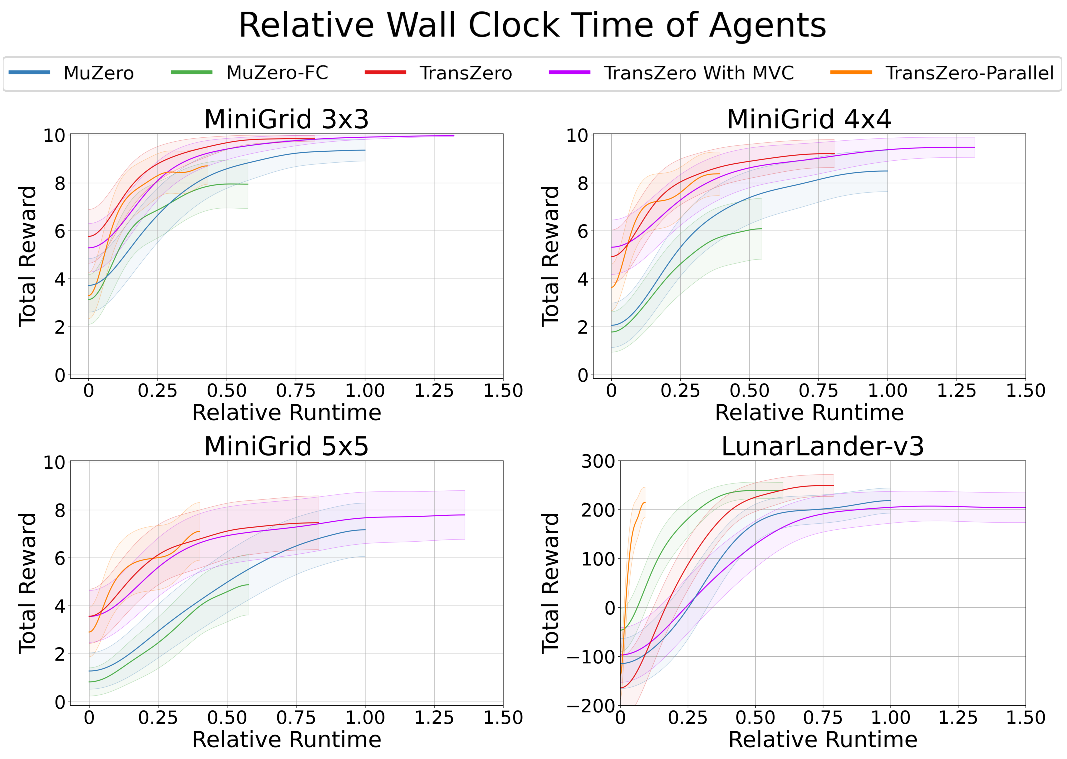


Figure 6.2: (Top) Average reward of the agents on MiniGrid $3 \times 3$, $4 \times 4$, $5 \times 5$, and LunarLander, as a function of environment steps. (Bottom) Same results plotted over relative wall-clock time (compared to MuZero). The shaded area shows standard error. See section 5.1 for environment details.

Table 6.1 summarizes key metrics related to solving the environments. The upper rows show the proportion of environment steps used to reach a solution, expressed as a percentage of the total available. The middle rows report the relative wall-clock time to solve it, compared to MuZero. The bottom rows indicate how many of the seeds successfully solved the environment. For LunarLander, an environment is considered solved when an average score of 200 is achieved over 25 episodes. For MiniGrid, the threshold is an average score of 7.5 over 50 episodes. More details about the evaluation criteria can be found in subsection 5.1.3. For seeds that did not solve the environment, we report the last timestep as the point of solution. If 50% or more seeds failed to solve it - which only occurred for MuZero-FC in the $4 \times 4$ and $5 \times 5$ MiniGrid environments - we omit the score, as it would not accurately reflect the agent's performance.

**Percentage of Environment Steps and Relative Times to Solve the Environment**

| **Model** | **3×3** | **4×4** | **5×5** | **Lunar** |
|---|---|---|---|---|
| *Environment Steps as a Percentage of Total* | | | | |
| MuZero | 33% (±3%) | 46% (±4%) | 80% (±4%) | 61% (±10%) |
| MuZero-FC | 45% (±5%) | – | – | **45%** (±4%) |
| TransZero | **15%** (±1%) | **21%** (±2%) | **59%** (±10%) | **54%** (±10%) |
| TransZero with MVC | **17%** (±1%) | **24%** (±3%) | **49%** (±8%) | **51%** (±3%) |
| TransZero-Parallel | 28% (±3%) | 31% (±4%) | 73% (±8%) | **56%** (±8%) |
| *Relative Time Compared to MuZero* | | | | |
| MuZero | 1.00 (±0.10) | 1.00 (±0.10) | 1.00 (±0.05) | 1.00 (±0.14) |
| MuZero-FC | 0.81 (±0.08) | – | – | 0.39 (±0.02) |
| TransZero | **0.37** (±0.02) | 0.38 (±0.04) | 0.61 (±0.11) | 0.59 (±0.09) |
| TransZero with MVC | 0.67 (±0.04) | 0.67 (±0.09) | 0.83 (±0.14) | 1.25 (±0.07) |
| TransZero-Parallel | **0.36** (±0.04) | **0.26** (±0.03) | **0.36** (±0.04) | **0.070** (±0.01) |
| *Percentage of Seeds Ran that Solved the Environment* | | | | |
| MuZero | 100% | 100% | 80% | 100% |
| MuZero-FC | 100% | 50% | 0% | 100% |
| TransZero | 100% | 100% | 80% | 100% |
| TransZero with MVC | 100% | 100% | 100% | 100% |
| TransZero-Parallel | 100% | 100% | 80% | 100% |

Table 6.1: Percentage of total environment steps to solve the environements(top), percent of seeds that solved (middle), and wall-clock time relative to MuZero to solve (bottom). Agents with $< 80$ % solved runs are omitted. Solving thresholds are defined in subsection 5.1.3.

From Table 6.1 (top), we see that TransZero reached the solving threshold in significantly fewer steps than MuZero across the MiniGrid environments. In the $3 \times 3$ and $4 \times 4$ maps, it required approximately half as many steps. For LunarLander, the number of steps was similar between the two, with differences falling within the standard error.

Compared to MuZero-FC, TransZero used approximately one-third as many steps to solve the $3 \times 3$ MiniGrid. In the $4 \times 4$ MiniGrid, MuZero-FC solved the environment in only half of the runs, and in the $5 \times 5$, it failed to solve it entirely. This can be seen in Table 6.1 (middle). MuZero-FC required on average fewer steps to solve LunarLander than TransZero, although the difference was not statistically significant.

### 6.1.2 TransZero Wall-Clock Time Performance

In Figure 6.2 (bottom), we show the reward obtained by each agent as a function of the relative wall-clock time compared to MuZero. As seen in the plot, TransZero converges faster than MuZero in all MiniGrid environments. This is further confirmed in Table 6.1 (middle), where TransZero solves each environment significantly faster. Compared to MuZero-FC, TransZero is faster in the $3 \times 3$ MiniGrid environment, but slower in LunarLander.

The differences between wall-clock time and sample efficiency arise from variation in the time it takes each agent to process the same number of environment steps. In Table 6.2, we report the time required to complete training (left) and the time required for planning, that is, selecting an action, relative to MuZero (right). As shown, TransZero is approximately 20 percentage points faster than MuZero, but about 20 percentage points slower than MuZero-FC in both training and planning speed.

**Relative Total Time to Train the Model and Relative Time for One Action Selection**

| **Model** | **Rel. Total Time** | | **Rel. Planning Time** | |
|---|---|---|---|---|
| | MiniGrid | Lunar | MiniGrid | LunarLander |
| MuZero | 1.0 (±0.01) | 1.0 (±0.01) | 1.0 | 1.0 |
| MuZero-FC | 0.57 (±0.01) | 0.60 (±0.01) | 0.60 | 0.59 |
| TransZero | 0.82 (±0.01) | 0.76 (±0.01) | 0.79 | 0.81 |
| TransZero with MVC | 1.3 (±0.02) | 1.6 (±0.08) | 1.1 | 1.3 |
| TransZero-Parallel | **0.41** (±0.0) | **0.092** (±0.0) | **0.27** | **0.089** |

Table 6.2: Comparison of the time to complete training (left) and the time it takes to do planning, i.e. a single action selection (right) in MiniGrid and LunarLander. The times are relative to the times it took for MuZero.

## 6.2 TransZero with MVC Results

When it comes to sample efficiency, Figure 6.2 (top) shows that TransZero with MVC performs very similarly to TransZero in all environments. As seen in Table 6.1 (top), it solves the MiniGrid environments with slightly fewer steps than MuZero and requires slightly more steps in LunarLander. However, none of these differences are statistically significant. We therefore conclude that incorporating MVC does not degrade sample efficiency compared to the baseline.

According to Table 6.2 (left), the use of the MVC architecture increases the training time by an average of 80% compared to TransZero in the two environments. This overhead comes from the additional computations required during each MCTS backup.

## 6.3 TransZero-Parallel Results

In subsection 6.3.1, we show that TransZero-Parallel is as sample-efficient as MuZero, though not generally as efficient as TransZero with MVC. In subsection 6.3.2, we present the substantial improvements in computational speed achieved with TransZero-Parallel. Finally, in subsection 6.3.3, we show that TransZero-Parallel could, in theory, be orders of magnitude faster than MuZero in certain environments.

### 6.3.1 TransZero-Parallel Sample Efficiency

As shown in Figure 6.2 (top), TransZero-Parallel performs similarly to MuZero in all environments in terms of sample efficiency. This is further confirmed by the time required to solve each environment, where the two agents perform within the standard error of each other in all environments except $4 \times 4$ MiniGrid, where TransZero-Parallel is slightly better. These results are visible in Table 6.1 (top).

Compared to MuZero-FC, TransZero-Parallel performs better on average across all MiniGrid environments, particularly during the early stages of training. In LunarLander, MuZero-FC shows an initial advantage, although the difference is not statistically significant at the end of training. When comparing time to solve each environment, TransZero-Parallel completes MiniGrid tasks significantly faster, while MuZero-FC solves LunarLander faster on average, though the difference is not significant.

Compared to TransZero with MVC—the architecture most similar to TransZero-Parallel—we observe slightly worse sample efficiency from TransZero-Parallel on average in MiniGrid. However, the difference is only statistically significant in the $3 \times 3$ environment. TransZero-Parallel also requires approximately 13% more steps on average to solve the MiniGrid environments. In LunarLander, the two agents perform similarly, both in terms of time to solve and overall sample efficiency across training.

### 6.3.2 TransZero-Parallel Wall-Clock Time Performance

The wall clock time is where TransZero-Parallel excels. As shown in Figure 6.2 (bottom), it is significantly faster than MuZero and TransZero with MVC in all environments, with the most notable speed-up observed in LunarLander. Compared to TransZero and MuZero-FC, the advantage is slightly smaller but still substantial in the case of LunarLander.

Looking at the total training time in Table 6.2 (left), TransZero-Parallel completes LunarLander training approximately 11x faster than MuZero, 6.5x faster than MuZero-FC, 8.2x faster than TransZero, and 17x faster than TransZero with MVC. For MiniGrid, the speedup is smaller but still notable—around 2.5x faster than MuZero and 1.4x faster than MuZero-FC.

This increased speed is also reflected in the time required to solve each environment, as shown in Table 6.1 (middle). TransZero-Parallel solves LunarLander approximately 14x faster than MuZero and around 5.5x faster than MuZero-FC. It is also the fastest agent across all MiniGrid environments, with the exception of the $3 \times 3$ case, where it shares the top performance with TransZero.

The reason why TransZero-Parallel achieves greater relative speedups in LunarLander than in MiniGrid is due to the difference in the number of simulations performed by each agent. TransZero-Parallel runs 4 simulations per action selection in MiniGrid and 2 in LunarLander, whereas the other agents run 25 and 50 simulations respectively. This means that, relatively, TransZero-Parallel performs 6.25 times fewer simulations in MiniGrid and 25 times fewer in LunarLander. Although each simulation in LunarLander expands 84 nodes (compared to 12 in MiniGrid), this does not significantly affect the per-simulation speed since expansions are executed in parallel. We expect this performance trend to continue in larger environments—the more simulations required, the greater the relative advantage of TransZero-Parallel.

### 6.3.3 Limits of Parallel Node Expansion

From Table 6.3, we observe how many more nodes TransZero-Parallel can expand in the time it takes MuZero to expand a single node, across different configurations of tree-layers and simulations. For example, consider the case with an action space of 18 and two subtree-layers. This results in a subtree with 342 nodes: 18 for the first layer and $18^2 = 324$ for the second. If 4 simulations are performed, a total of $4 \cdot 342 = 1368$ nodes are expanded during a single action selection. As shown in the table, TransZero-Parallel performs this action selection approximately 160x faster than MuZero would have performed an action selection with 1368 simulations, even though an equal number of nodes were expanded. These measurements were obtained by running action selection in isolation for the specified configurations and do not correspond to any particular environment. For simplicity, we omitted the root node of the MCTS in the column that shows the number of nodes in a subtree.

**Comparative Node Expansions Between TransZero-Parallel and MuZero**

| Subtree Layers | Num. Nodes in Subtree | Node Expansions 1 Sim. | 4 Sim. | 16 Sim. |
|---|---|---|---|---|
| **Action Space = 4** | | | | |
| 1 | 4 | 3.0 | 2.4 | 1.8 |
| 2 | 20 | 11 | 9.2 | 7.6 |
| 3 | 84 | 34 | 31 | 28 |
| 4 | 340 | 150 | 120 | 100 |
| 5 | 1364 | 460 | 350 | 340 |
| 6 | 5460 | 300 | 280 | 280 |
| **Action Space = 18** | | | | |
| 1 | 18 | 11 | 8.8 | 7.9 |
| 2 | 342 | 180 | 160 | 150 |
| 3 | 6174 | 270 | 270 | 250 |
| **Action Space = 40** | | | | |
| 1 | 40 | 23 | 19 | 17 |
| 2 | 1640 | 560 | 530 | 520 |

Table 6.3: Number of nodes TransZero-Parallel can expand in the time MuZero expands one, across different configurations. This is a controlled experiment measuring the time for a single action selection

Up to approximately 1600 node expansions in a single subtree, we observe that increasing the number of nodes expanded in parallel leads to greater speed gains. The maximum speed-up is achieved when expanding two subtree-layers with a single simulation in an action space of 75, resulting in up to a 560-fold speed increase compared to MuZero expanding the same number of nodes. However, beyond 1600 node expansions, the relative speed improvement begins to diminish. This slowdown is due to the quadratic complexity of the transformer architecture.

As shown in Table 6.4, the time spent on MVC calculations increases roughly linearly, about 0.4 ms per additional subtree-layer. In contrast, the time required for the network's forward pass rises sharply beyond approximately 1364 nodes. We also observe that GPU memory usage remains low until this point, after which it increases roughly 15 fold. In subsection 7.3.1, we propose a modified attention mechanism that may partially address this computational bottleneck.

**System Profiling for TransZero-Parallel**

| Subtree Layers | Num. Nodes in Subtree | Network Pass (ms) | MVC-Calculations (ms) | Peak GPU-Memory (MB) |
|---|---|---|---|---|
| 1 | 4 | 2.5 | 1.0 | 18 |
| 2 | 20 | 2.5 | 1.4 | 18 |
| 3 | 84 | 2.5 | 1.8 | 18 |
| 4 | 340 | 2.5 | 2.2 | 26 |
| 5 | 1364 | 4.3 | 2.7 | 270 |
| 6 | 5460 | 50 | 3.0 | 4000 |

Table 6.4: Comparison of the time to complete training (left) and the time it takes to do a single action selection (right) in MiniGrid and LunarLander. The times are relative to the times it took for MuZero.

From Table 6.3 we also see that TransZero-Parallel is not dependent on the size of the action space but on the number of nodes expanded. For example, expanding 20 nodes when the action space is 4 is equally fast as expanding 18 nodes when the action space is 18. This also applies to the 340 vs 342 nodes for the same action spaces.

We also observe that increasing the number of simulations makes the method relatively slower, although the gap narrows as the total number of expanded nodes increases. This is because each additional simulation requires recalculating more nodes. Moreover, since we always expand as deep as possible—by repeatedly selecting the same action—this setup represents a worst-case scenario. Therefore, performance under multiple simulations can be seen as a conservative lower bound.

## 6.4 Additional Findings

In this section, we present findings that do not directly relate to the main comparisons of the architectures in terms of sample efficiency and wall-clock time, but are still relevant to understanding overall performance. In subsection 6.4.1, we compare the relative time spent planning in relation to the total training time for each agent. In subsection 6.4.2 we discuss the training step difference between the agents.

### 6.4.1 Action Selection Compared to Training Time

From Table 6.2, we observe a strong correlation between the relative total training time (left) and the relative time required for a single action selection (right). This supports our hypothesis that reducing action selection time is the key factor in decreasing overall training time.

There are, however, some notable exceptions. For instance, we would expect the relative action selection time for TransZero-Parallel in MiniGrid to more closely match its total training time, especially since this is the case for LunarLander. One possible explanation is that, because TransZero-Parallel is faster in absolute terms, operations unrelated to action selection account for a larger proportion of the total training time. This could include generating the random MiniGrid environment and verifying that a path exists from the agent to the goal. Also, these operations would have been faster when doing a single action selection since we used better hardware during those experiments compared to when we trained the agents.

We further observe that TransZero with MVC shows relatively faster action selection times compared to its total training time. We also believe this is due to the stronger hardware used for the single action selection measurement, which likely accelerated the MVC computations. The fact that TransZero with MVC is relatively slower in LunarLander compared to MiniGrid is likely explained by the MVC computations as well. This is because deeper search trees are created when using 50 simulations instead of 25, leading to more MVC calculations per simulation during the backup phase.

### 6.4.2 Training Steps per Agent

In Table 6.5 (left), we show the relative total number of training steps each agent performs during training compared to MuZero. On the right, we show the ratio of training steps to environment steps. A training step corresponds to a single gradient update in the training phase of the algorithm.

Looking at the ratio between TransZero and MuZero, we see that TransZero performs nearly twice as many training steps per environment step. This is because training is asynchronous from self-play, and TransZero's training phase is substantially faster than MuZero's. The key reason is that TransZero can perform gradient updates in parallel over the entire rollout, rather than recurrently as in MuZero—see subsection 3.1.3 for details. Self-play time, on the other hand, is roughly similar between the two agents, since MuZero can cache previously unrolled nodes.

The differences in training-to-environment step ratios for TransZero with MVC and TransZero-Parallel can be attributed to their self-play speeds: TransZero with MVC is slower during self-play, while TransZero-Parallel is faster. Their training procedures are otherwise identical to that of TransZero.

The difference in ratios between models is also reflected in the total number of training steps they perform during training, which is conducted over a fixed number of environment steps. On average across the two environments, TransZero with MVC performs approximately 3 times more training steps than MuZero and nearly twice as many as TransZero. Conversely, TransZero-Parallel performs significantly fewer training steps than both TransZero and MuZero, particularly in LunarLander. These differences in the number of training steps per fixed environment budget correlate somewhat with the observed sample efficiency in MiniGrid, although not in LunarLander.

| **Model** | **Rel. Total Training Steps** | | **Tr. to Env. Step Ratio** | |
|---|---|---|---|---|
| | MiniGrid | LunarLander | MiniGrid | LunarLander |
| MuZero | 1.0 (±0.00) | 1.0 (±0.01) | 0.47 (±0.00) | 1.3 (±0.01) |
| MuZero-FC | 1.2 (±0.00) | 1.1 (±0.02) | 0.55 (±0.00) | 1.7 (±0.02) |
| TransZero | 1.7 (±0.10) | 1.4 (±0.01) | 0.87 (±0.01) | 1.8 (±0.02) |
| TransZero with MVC | 3.2 (±0.02) | 2.5 (±0.06) | 1.6 (±0.01) | 4.51 (±0.08) |
| TransZero-Parallel | 0.81 (±0.01) | 0.23 (±0.00) | 0.39 (±0.00) | 0.41 (±0.01) |

Table 6.5: (Left) Relative total training steps across training compared to MuZero. (Right) Training-to-environment step ratio.

# Chapter 7

# Discussion

We begin this chapter in section 7.1 by interpreting the results obtained. In section 7.2, we discuss the limitations of the study. Finally, in section 7.3, we outline potential future improvements for both TransZero and TransZero-Parallel.

## 7.1 Interpretation of Results

In subsection 7.1.1, we examine the potential and limitations of parallel expansion. In subsection 7.1.2, we discuss the discrepancies observed in some agents across the two environments.

### 7.1.1 Parallel Expansion Analysis

**Potential of Parallel Expansion**

In domains such as board games, MuZero employs up to 800 simulations [4]. A rough estimate based on the LunarLander and MiniGrid environments suggests that TransZero-Parallel requires expanding approximately two to three times as many nodes as MuZero. In the case of chess, using a conservative estimate of four times as many node expansions, this would correspond to $800 \times 4 = 3200$ nodes. Referring to Table 6.3, for an action space of 40, expanding roughly 3200 nodes could be achieved by using two simulations where each expands two subtree-layers ($1640 \times 2 = 3280$).

According to the table, expanding 3280 nodes using TransZero-Parallel is approximately 530 to 560x faster than MuZero performing the same number of expansions; we conservatively estimate this speedup as 540x. Thus, if MuZero uses 800 expansions while TransZero-Parallel performs 3280, the relative speedup is approximately $540 \div 4 = 135$-fold. Such a performance gain not only enables significantly faster decision-making during inference, but also drastically reduces overall training time. For example, models that currently take 24 hours to train could be trained in slightly more than 10 minutes.

However, this analysis is highly theoretical and may not generalize well across environments. For example, in an environment with an action space of 40, expanding two layers of a subtree may lead to a broader expansion than is actually useful. Consider chess: its action space is 4672, yet MuZero uses only 800 simulations. This means it cannot fully expand even a single layer of the search tree. Instead, MuZero relies more heavily on its prior policy prediction to guide deeper search. This suggests that expanding all actions at every tree-layer may offer limited benefit in practice.

#### Limitations of Parallel Expansion

We observed that expanding more tree-layers led to a greater relative speedup compared to using fewer layers with more simulations. However, expanding beyond three layers for LunarLander and two layers for MiniGrid resulted in a decrease in sample efficiency. One possible explanation comes from He et al. [17], who found that MuZero learns accurate value predictions only for policies that remain close to its training distribution. In MuZero, the prior policy helps guide action selection toward regions where the model is more accurate, thus compensating for this limitation.

In contrast, TransZero-Parallel makes only minimal use of the prior policy. Although the prior is applied at each selection step, each simulation still expands many more nodes than are actually selected. For example, in the case of LunarLander, 84 additional nodes are expanded beyond those chosen by the policy in each simulation. This significantly reduces the influence of the prior in guiding the search. TransZero-Parallel appears to manage this discrepancy when the tree is relatively shallow, but as the depth increases, the divergence between the expanded nodes and the learned policy may become too large, potentially reducing value accuracy and overall performance.

### 7.1.2 Discrepancies Between Environments

First, we observe that MuZero-FC performs significantly better in LunarLander than in MiniGrid, despite LunarLander being the more complex environment. We hypothesize that this is due to the lack of convolutional layers in MuZero-FC. In this architecture, the two-dimensional observations are flattened and passed directly into the representation network, removing the spatial structure present. Without convolutional layers to exploit spatial locality, the model lacks an inductive bias for spatial reasoning and must learn spatial dependencies from scratch. This generally leads to slower generalization and reduced sample efficiency, especially in environments like MiniGrid where spatial relationships—such as agent position, obstacles, and goals—are critical for decision making.

We also observe in Figure 6.2 (top) that TransZero with MVC consistently achieves one of the highest average performances in MiniGrid, yet one of the lowest in LunarLander. We believe this is primarily due to hyperparameter settings rather than the action selection mechanism itself. This is supported by the fact that there is nothing inherent to the LunarLander environment that should make it particularly ill-suited for the MVC evaluator. Additionally, due to the high computational cost of running TransZero with MVC on

LunarLander, we were unable to invest as much effort into hyperparameter tuning for this agent compared to others.

One might assume that the hyperparameters used for TransZero would generalize to TransZero with MVC. Although this is partially true for architecture-related parameters, it does not hold for those governing action selection. Specifically, parameters involved in the UCB score calculation—such as the exploration constant $C_{puct}$ and the temperature used for sampling actions—require separate tuning due to the differences between using visitation counts and using the MVC evaluator. Furthermore, MVC introduces an additional parameter, $\beta$, which also needs careful adjustment. These factors interact and influence the exploration–exploitation trade-off, making the overall tuning process substantially different.

## 7.2 Limitations

This study has two main limitations. First, potential unfairness in the hyperparameter tuning, which we cover in subsection 7.2.1. Second, the large standard error observed in the MiniGrid plots, which we cover in subsection 7.2.2.

### 7.2.1 Hyperparameter Tuning

Model-based Reinforcement Learning (MBRL) is challenging to tune due to its reliance on several interacting sub-components, each with design parameters that can significantly affect overall performance [18]. This was evident in our study, where even small changes in the hyperparameters had a substantial impact on the performance of all models. As such, it is possible that some of the observed performance differences are due to suboptimal hyperparameter choices.

Furthermore, the amount of time spent tuning each agent varied. For example, in the case of LunarLander, we used a set of hyperparameters from a public GitHub repository for MuZero-FC[1]. However, we do not know how much tuning effort was invested in those parameters compared to the time we dedicated to tuning the TransZero architectures.

For MiniGrid, we used a custom training setup, which required us to tune the hyperparameters for all agents ourselves. Reusing parameters from other environments was not feasible as we found that each environment required its own distinct configuration. As a result, some models received more tuning effort than others, either because they were more sensitive to hyperparameter changes or because we happened to identify a suitable parameter set earlier in the tuning process. Although further tuning could potentially affect the results to some extent, we believe that significant performance differences are unlikely, given the level of attention and tuning effort dedicated to each agent.

[1] `https://github.com/alexZajac/muzero_experiments/tree/master`

### 7.2.2 Large Standard Error

As shown in the MiniGrid results, the standard error for all agents is relatively large. This makes it more difficult to determine whether the observed average performance differences between agents are statistically significant. The main reason for this is that we only ran each agent for 10 random seeds. The limited number of seeds was due to constraints in GPU availability and the time required to run each experiment. For example, a single full run across all MiniGrid environments and agents took approximately 48 hours.

However, we believe that this limitation does not affect the general conclusions of the thesis. It remains evident that TransZero and TransZero-Parallel achieve a level of sample efficiency comparable to that of MuZero and MuZero-FC. In fact, their average performances are extremely close—if not better in most cases. Additionally, it is unambiguous that TransZero-Parallel offers a significant speed advantage, being many times faster than MuZero.

## 7.3 Future Work

This section outlines five potential improvements. The first, described in subsection 7.3.1, is a more efficient attention computation that could lead to exponential gains in the matrix operations of TransZero-Parallel. The second, discussed in subsection 7.3.2, involves adopting techniques proposed for EfficientZero [19], with the goal of increasing sample efficiency. Next, in subsection 7.3.3, we explore ways to redesign the subtree expansion process, particularly for environments with large state spaces. In subsection 7.3.4, we discuss how stacked observations could be implemented for TransZero. Finally, in subsection 7.3.5, we outline additional environments that should be tested with TransZero and its variants.

### 7.3.1 New Attention Matrix

In TransZero-Parallel, some of the computations in the key-query attention matrix during subtree expansion are redundant, due to the repeated presence of tokens with identical embeddings. Specifically, tokens that represent the same action and are located at the same depth in the search tree receive the same learned embedding and identical positional encoding. These tokens are therefore functionally identical in the attention calculation. A visualization of these nodes can be seen in Figure 7.1 (top); they are the nodes with the same color in the same layer of the subtree.

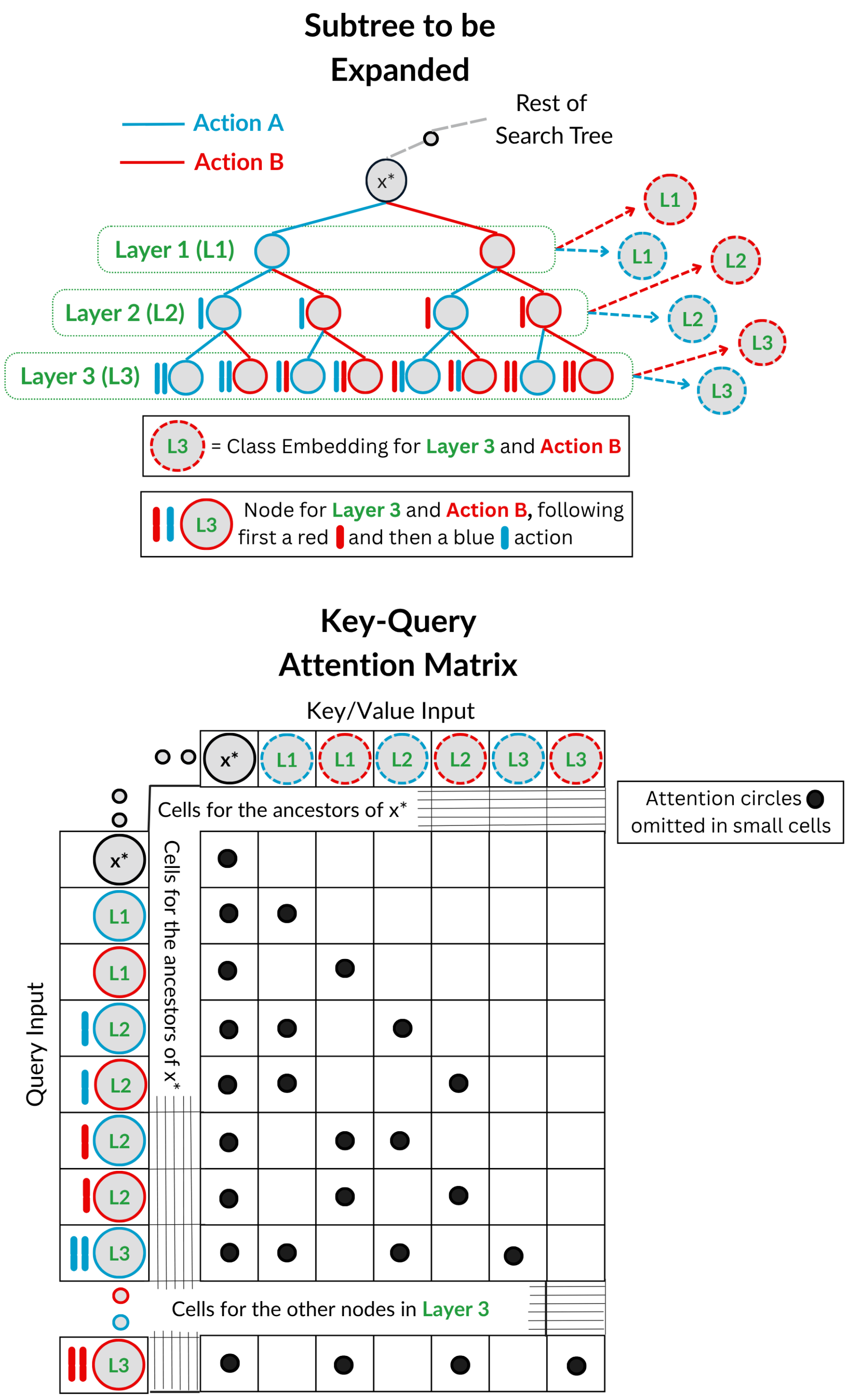

Subtree to be Expanded
Action A
Action B
Rest of Search Tree
x*
Layer 1 (L1)
Layer 2 (L2)
Layer 3 (L3)
L1
L2
L3
L3 = Class Embedding for Layer 3 and Action B
L3 Node for Layer 3 and Action B, following first a red and then a blue action
Key-Query Attention Matrix
Key/Value Input
Query Input
Cells for the ancestors of x*
Attention circles ● omitted in small cells
Cells for the other nodes in Layer 3

Figure 7.1: (Top) The subtree to be expanded in parallel. (Bottom) The resulting attention matrix using our proposal in subsection 7.3.1.

We define all nodes that share an identical embedding as belonging to an *embedding class* $u$. Although this is not a literal form of containment, as being in $u$ simply indicates that the nodes share the same embedding, we adopt this terminology for ease of reference. The nodes in question share an embedding because they represent the same action and occur at the same depth in the tree.

In Figure 7.1 (top), we illustrate how the nodes at each layer are grouped into their respective embedding classes: for example, the red nodes in a layer are grouped into the red class-embedding for that layer, and similarly for the blue nodes. Red and blue correspond to two different actions in an action space of size two. The full ordered list of these embedding classes, taken from top to bottom in the search tree, is referred to as the *embedding class sequence* $\mathcal{U} = (u_1, u_2, \dots, u_n)$.

For each layer $i$ (out of $N_l$ layers) in the subtree we wish to expand, there are $|\mathcal{A}|^i$ nodes. However, only $|\mathcal{A}|$ distinct embedding classes exist per layer, since each node at a given depth corresponds to one of the actions in $\mathcal{A}$ and shares its embedding with all other nodes representing the same action at that depth. This implies that at layer 1, each embedding class contains one node; at layer 2, each class contains $|\mathcal{A}|$ nodes; at layer 3, $|\mathcal{A}|^2$, and so on.

The total number of nodes in the full subtree is given by

$$\sum_{i=0}^{N_l} |\mathcal{A}|^i = \frac{1 - |\mathcal{A}|^{N_l+1}}{1 - |\mathcal{A}|} \in O(|\mathcal{A}|^{N_l}),$$

while the total number of embedding classes are

$$\sum_{i=0}^{N_l} |\mathcal{A}| = N_l \cdot |\mathcal{A}| \in O(N_l \cdot |\mathcal{A}|).$$

This means that if we could restructure the attention computation to operate on the layer-wise unique embeddings, at least in part, we could reduce the computational cost exponentially.

### Cross-Attention

For our first approach, we require the use of *cross-attention*. Up to this point, we have been using self-attention, where the queries $Q'$, keys $K'$, and values $V'$ are all derived from the same embedded input sequence $X^{emb}$. In contrast, cross-attention allows the queries to come from one sequence while the keys and values come from another. This means that instead of using $X^{emb}$ for all three components, we compute:

$$Q' = X_{\mathrm{q}}^{emb} W_{Q'}, \quad K' = X_{\mathrm{kv}}^{emb} W_{K'}, \quad V' = X_{\mathrm{kv}}^{emb} W_{V'}.$$

Here, $X_{\mathrm{q}}^{emb}$ and $X_{\mathrm{kv}}^{emb}$ are two distinct embedded input sequences.

### Using the Embedding Class Sequence for Keys and Values

Our first proposal is to set $X_{\mathrm{q}}^{emb} = X^{emb}$ and $X_{\mathrm{kv}}^{emb} = \mathcal{U}^{emb}$. A similar approach has been proposed by Jaegle et al. [20] and is reminiscent of a recurrent neural network structure. A visualization of the resulting attention matrix is shown in Figure 7.1. The nodes before $x^*$ are omitted from both the visualization and our descriptions for simplicity, the computations of these will work as before and do not add significant computational complexity.

With this setup, the length of the keys and values becomes $|K'| = |V'| = O(N_l \cdot |\mathcal{A}|)$, while the length of the queries remains $|Q'| = O(|\mathcal{A}|^{N_l})$. The resulting attention matrix has the size of:

$$O(|\mathcal{A}|^{N_l}) \times O(N_l \cdot |\mathcal{A}|) = O(N_l \cdot |\mathcal{A}|^{N_l+1}).$$

In comparison, standard self-attention applied to the same subtree has $|V'| = |K'| = |Q'| = O(|\mathcal{A}|^{N_l})$, resulting in an attention matrix of size:

$$O(|\mathcal{A}|^{N_l}) \times O(|\mathcal{A}|^{N_l}) = O(|\mathcal{A}|^{2N_l}).$$

This represents an exponential reduction in computational complexity. Furthermore, the multiplication of the attention matrix with the value matrix $V'$ yields an equal exponential gain. We omit the hidden dimension $d_t$ from these calculations, as it remains constant for both approaches.

From the first transformer layer, we obtain the output $\tilde{S}^{(1)}$, which can be used to compute the query matrix $Q'$ for the next transformer layer. However, the question remains how to compute the corresponding $K'$ and $V'$ matrices. The most straightforward approach is to cache $K'$ and $V'$ and reuse them in all layers. This has the advantage of requiring the projection matrices $W_{K'}$ and $W_{V'}$ to be computed only once, reducing the total parameter count. This approach was also suggested in Jaegle et al. [20].

The drawback of this method is that a fixed set of keys and values cannot adapt to information processed in earlier layers. Since each layer attends to the exact same $K'$ and $V'$, later layers receive no additional contextual input, only the layer-specific query transformations vary. This could potentially limit the model's expressiveness, although an empirical study would be needed to confirm this. An alternative is to learn new key and value projections at each layer. Although this approach increases the parameter cost compared to caching, the added expense is relatively modest due to the small size of these matrices. For both of these methods, we would need to adapt training to use cross-attention in the same manner.

### Masking the New Attention Matrix

We also need to construct a new attention mask, which we call $M_{\mathrm{set\_causal}}$, to ensure that future actions do not attend previous ones. This mask is conceptually similar to $M_{\mathrm{tree\_causal}}$, but adapted to operate at the level of embedding classes rather than individual nodes. Specifically, a node should attend to an embedding class $u$ if, in the original tree-based setting, it would have attended any node $x$ contained in $u$ when all tokens were nodes. Formally, we define:

$$M_{\text{set\_causal}\,ij} = \begin{cases} 1 & \text{if } \exists x \in \mathbf{a}_{\tilde{s}_{\text{root}} \to x_i} \text{ such that } \mathcal{E}(x) = u_j, \\ 0 & \text{otherwise.} \end{cases}$$

Here, $\mathcal{E}(x)$ returns the embedding class $u$ of node $x$. In other words, a node $x_i$ attends the embedding class $u_j$ if there exists a node along the path from the root to $x_i$ that belongs to that embedding class. The effect of the mask is visualized in Figure 7.1 where the black dots show which tokens should attend which.

**Growing the Embedding Class Sequence as Query**

The reason we cannot use the embedding-class sequence $\mathcal{U}$ as input to both the queries and the keys/values is that the network must be aware of which specific sequence it generates outputs for. When using only the embedding-class sequence, the model lacks what we refer to as *ancestral information*—it cannot infer the order in which actions occur. Furthermore, the output would have dimensionality $N_l \cdot |\mathcal{A}|$, making it unclear how to map the results back to the full set of nodes that we intend to expand.

However, an alternative approach could involve progressively expanding $\mathcal{U}^{emb}$ to the size of $X^{emb}$ over multiple transformer layers. This would produce an output with the correct dimensionality, while gradually introducing more ancestral information across layers. In such a setup, the early layers would operate on a compact representation, resulting in linear computational complexity in tree depth, while the later layers would match the complexity of the approach described in section 7.3.1.

Technically, this strategy could also be applied with $X^{emb}$ used as input to the keys and values (instead of $\mathcal{U}^{emb}$). This would provide full ancestral information throughout the transformer, but the computational benefits would be reduced. The model would start with the same complexity as the approach in section 7.3.1 and eventually grow to match the full complexity of TransZero-Parallel toward the final layers.

For the growth of $\mathcal{U}^{emb}$, we believe that a smooth progression across transformer layers is important. If growth occurs too quickly in the early layers, the amount of computation saved would be minimal. Conversely, if the expansion happens too late, the network may not have sufficient depth to learn enough ancestral information. Our proposal of how this expansion could proceed over two iterations with $|\mathcal{A}| = 2$ is visualized in Figure 7.2.

In the first transformer layer, we use $\mathcal{U}^{emb}$ directly as the input to the queries. At this stage, the embedding classes corresponding to the first tree-layer map 1:1 to their associated nodes. For the second tree-layer, each embedding class represents two nodes, resulting in a $1 : |\mathcal{A}|$ ratio. For the third layer, the ratio becomes $1 : |\mathcal{A}|^2$, and so on. At this stage, the actions completely lack ancestral information—they cannot distinguish which actions come earlier in the sequence due to the fact that no masking is possible.

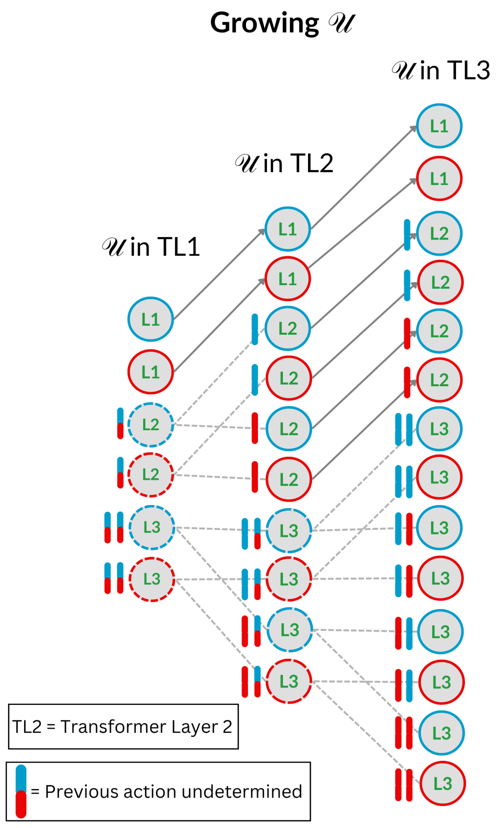


Figure 7.2: Suggestion of how $\mathcal{U}$ can be grown between transformer-layers.

Before the second transformer layer, we project the embedding classes that do not yet have a 1:1 correspondence with nodes by a factor of $|\mathcal{A}|$. In the first transformer layer, the projection is thus applied to all embedding classes except those corresponding to the first tree-layer. In the second transformer layer, we further project all embedding classes except those associated with the first two tree-layers, and so on. These projections are illustrated in Figure 7.2 as striped lines and can be implemented, for example, using an MLP or copying.

What this approach essentially does is to progressively add ancestral information to the embedding classes at each layer. After the first projection, the embedding classes associated with the first tree-layer remain unchanged, as they already have a 1:1 correspondence with the nodes in that layer. The embedding classes corresponding to the remaining tree-layers are projected from size $|\mathcal{A}|$ to $|\mathcal{A}|^2$. As a result, all embedding classes now "know" which first-level action preceded them, since irrelevant actions can now be masked out in the attention matrix. However, they still lack information about the rest of the actions. This additional context is gradually revealed across the transformer layers, as we continue projecting any embedding classes that do not yet have a 1:1 correspondence with nodes. Through this process, one additional action becomes known at each transformer layer.

One benefit of this method is that each projection increases the size of the embedding classes by a factor of $|\mathcal{A}|$, ensuring a smooth and incremental increase in representation capacity—rather than a sudden expansion at the end. The growth of $\mathcal{U}^{emb}$ between transformer layers is given by:

$$(N_l - i) \cdot |\mathcal{A}|^i \cdot (|\mathcal{A}| - 1)$$

where $i$ denotes a specific transformer layer, with $1 < i \leq N_l$. This leads to exponential growth when $|\mathcal{A}| > 2$, which is computationally beneficial as the matrices will remain relatively small until the last transformer layers.

We do not elaborate on the exact masking mechanism here, but it could be constructed as a hybrid between $M_{\text{set\_causal}}$ and $M_{\text{tree\_causal}}$. This is feasible since, at each transformer layer, the available ancestral information is explicitly known. We also do not go into detail on how the training procedure would change. In essence, it must mirror the progressive construction described above, with additional ancestral information being revealed at each transformer layer. This allows the model to learn the correct dependencies step by step.

The proposed progressive growth scheme assumes that the number of transformer layers is exactly one less than the number of tree-layers. However, this constraint can be relaxed. If the number of transformer layers is greater than or equal to the number of tree-layers, we can simply omit projections between some layers. Conversely, if there are more tree-layers than transformer layers, we can apply multiple projections between certain transformer layers—for example, expanding all affected embedding classes by a factor of $|\mathcal{A}|^2$. Intermediate strategies are also possible, such as projecting only a subset of the embedding classes.

### 7.3.2 EfficientZero Techniques for TransZero

EfficientZero [19] introduces three key modifications that improve sample efficiency compared to MuZero. It achieved superhuman performance in 14 out of 26 Atari games, using only as many environment interactions as a human would generate by playing for approximately two hours. This section discusses how two of EfficientZero's techniques—*self-supervised consistency loss* and *end-to-end prediction of the value prefix*—could be adapted for TransZero. The third technique, *off-policy correction*, is unaffected by the changes introduced in TransZero, and can therefore be implemented in the same way as for MuZero.

#### End-To-End Prediction of the Value Prefix

In MuZero, the network attempts to predict the reward at each timestep, which presents challenges due to the difficulty of identifying the exact frame or state that yields the reward. This can result in compounding prediction errors during the recurrent rollout—a problem referred to as *state aliasing*. EfficientZero addresses this issue by predicting the cumulative reward over a window of timesteps in an end-to-end manner.

More specifically, the predicted reward is used in the Q-value estimation:

$$Q(x_t \uplus a) = \sum_{i=0}^{T-1} \gamma^i \cdot r(x_{t+i}) + \gamma^T \cdot v(x_{t+T}),$$

where $T$ is the number of recurrent steps taken during the rollout in latent space. This formulation is equivalent to the one given in Equation 2.3. The prefix sum of rewards $\sum_{i=0}^{T-1} \gamma^i r(x_{t+i})$ is referred to as the *value prefix* $\mathcal{P}$, as it serves as the prefix to the full Q-value computation. EfficientZero proposes a new method for predicting this prefix, formulated as $\mathcal{P} = pf(s_t, \tilde{s}_{t+1}, \ldots, \tilde{s}_{t+k-1})$, where $pf$ is a neural network architecture—specifically, an LSTM in their case.

TransZero does not roll the dynamics model step-by-step, which reduces the drift associated with repeatedly predicting what the next frame looks like. However, even with one-shot rollouts, the model still needs to infer which latent state in the returned sequence encodes the reward signal. This ambiguity is particularly problematic when rewards are delayed, sparse, or noisy. To address this, we propose changing the reward prediction target to the value prefix. That is, the reward network would predict $\mathcal{P}(\tilde{s})$ instead of $r(\tilde{s})$. Then, during search, the reward can be reconstructed as:

$$r(\tilde{s}_n) = \mathcal{P}(\tilde{s}_n) - \gamma \mathcal{P}(\tilde{s}_{n-1}).$$

If $\tilde{s}_n$ is the initial latent state, we define $\mathcal{P}(\tilde{s}_{n-1}) = 0$.

**Self-Supervised Consistency Loss**

The self-supervised consistency loss is designed to ensure that the model predicts state transitions accurately over time. It operates by comparing the next latent state $\tilde{s}_{t+1}$, produced by the dynamics network $g_\theta$, with the latent state $\tilde{s}'_{t+1}$, obtained by passing the true observation $o_{t+1}$ through the representation network $h_\theta$. To minimize the discrepancy between $\tilde{s}_{t+1}$ and $\tilde{s}'_{t+1}$, EfficientZero uses an architecture inspired by the SimSiam feature learning framework [21].

For TransZero, we would extend this approach by taking a sequence of observations $o_t, \ldots, o_{t+K}$ and processing them as a batch through the representation network to obtain the latent targets $\tilde{s}'_t, \ldots, \tilde{s}'_{t+K}$. These can then be compared to the corresponding predictions $\tilde{s}_t, \ldots, \tilde{s}_{t+K}$ generated by the TransZero dynamics model $g_\theta^{\text{trans}}$.

### 7.3.3 Alternate Expansion

Since the tree grows exponentially with respect to $|\mathcal{A}|$, subtree expansion in TransZero-Parallel becomes increasingly challenging as the action space increases. In chess, for example, AlphaZero [3] used an action space of size 4672. Expanding just two layers of the subtree would result in approximately 22 million nodes. As shown in Table 6.4, expanding around 5500 nodes already consumes 4 GB of GPU memory, making the expansion of 22 million nodes computationally infeasible.

To address this, one could consider expanding only the nodes for which all ancestors along the path exceed a certain PUCT score threshold. If, on average, we wish to expand $\bar{N}_n$ nodes per layer, then a subtree of depth $N_l$ would result in at most around $\bar{N}_n^{N_l}$ nodes being expanded in parallel. To apply this method efficiently, without recalculating the UCB score for all nodes at each step, a caching mechanism similar to that used for Q-values and variances in MVC (see subsection 3.2.4) can be used. Specifically, PUCT scores can be cached and updated along the path of the backup.

An issue with this approach is that the $U$-term of the PUCT score for a node is heavily influenced by its sibling nodes. First, the prior policy is normalized to sum to one across all child actions, meaning its scale depends on the quality of the sibling nodes. To mitigate this, we could weigh the prior policy by the Q-value of the parent. This adjustment would favor the exploration of promising children of high-value nodes over that of relatively strong children of low-value nodes.

Additionally, the last term of $U_{\tilde{\pi}}(x \uplus a)$ is

$$\frac{\sqrt{\mathbb{V}[Q_{\tilde{\pi}}(x)]^{-1}}}{1 + \mathbb{V}[Q_{\tilde{\pi}}(x \uplus a)]^{-1}}.$$

This term also gives the node a score that is relative to its siblings. As an alternative, we could combine the inverse variances of the parent and child—e.g., by multiplying them—to produce a score that is more comparable across the search tree.

Another limitation arises in states where all available actions yield low rewards. In such cases, all PUCT scores may fall below the fixed threshold, preventing any node from being expanded. To address this, we propose a dynamic threshold based on the average PUCT score. Specifically, we define the threshold as $\alpha$ times the average PUCT score, where $\alpha > 0$ is a tunable hyperparameter. This threshold could also be annealed over the course of training: a higher $\alpha$ early on encourages broader exploration, while a lower $\alpha$ later allows for more selective expansion as the network becomes more confident in its predictions.

This approach could also help address what we suspect was the reason deeper tree expansions did not yield additional benefits in subsection 7.1.1. Specifically, MuZero relies on the prior to guide action selection toward regions where the model is more accurate. In the above method, the prior is used at every selection step, rather than only at the root of the subtree being expanded.

### 7.3.4 Stacked Observations

In partially observable environments (POMDPs), the agent does not have full access to the underlying state of the environment. To mitigate this, MuZero uses a stack of recent observations to provide temporal context. Instead of feeding a single observation $o_t$ into the representation network $h_\theta$, MuZero concatenates the past $k$ observations into a single input:

$$o_t^{\text{stacked}} = \{o_{t-k+1}, \ldots, o_t\}.$$

This stacked observation $o_t^{\text{stacked}}$ is then encoded by the representation network:

$$\tilde{s}_{\text{root}} = h_\theta(o_t^{\text{stacked}}).$$

To adapt this approach for TransZero, several strategies can be considered. One straightforward method is to create a token for each observation in the stack using the representation network $h_\theta$, and prepend these tokens to the embedded action sequence. The positional encoding for these tokens could simply follow an increasing sequence, starting with the earliest observation in the stack. A drawback of this method is that the model must implicitly infer the type of each token—whether it represents an observation or an action. To address this, we can add an explicit type embedding to each token. The final token representation would then consist of the sum of the action (or observation) embedding, the positional encoding, and the type embedding.

Another approach is to extend the previous method by interleaving each stacked observation with the action that led to it. In some scenarios, it may be beneficial for the model to know which actions were taken to arrive at specific observations. To implement this, we follow a similar process as before: each stacked observation is passed through the representation network to produce a latent state. The interleaved actions are embedded in the same way as other actions. In this setup, it is crucial that positional encoding distinguishes between observations and actions. Otherwise, the model may misinterpret the interleaved action as part of the latent state sequence, rather than as the transition leading to the next observation. One solution is to use separate positional encoding schemes for observations and actions. In addition, a type embedding can be included to make the token type explicit, further helping the model to differentiate between observation and action tokens.

This approach could also be used beyond POMDPs, UniZero [14] uses a similar approach to improve sample efficiency in MDPs. Although UniZero uses some additional techniques in their transformer backbone, the principle of using a root state informed by previous observations remains the same.

### 7.3.5 More Environments

So far, we have only evaluated the agents in toy environments due to GPU and time constraints. It remains to be seen whether they would perform equally well in more complex settings, such as Atari games. Although all TransZero agents scaled effectively to LunarLander, which is more complex than MiniGrid, it is not guaranteed that this trend will continue. This is particularly uncertain for TransZero-Parallel, as it is the agent that differs most significantly from MuZero.

It would also be valuable to evaluate TransZero-Parallel in environments with even larger state spaces to assess whether it maintains scalability in such settings. This would help identify the practical limits of the proposed parallel subtree expansion and determine how much speed-up can realistically be achieved.

# Chapter 8

# Conclusion

This thesis presented *TransZero-Parallel*, a modified MuZero variant that enables parallel planning in Monte Carlo Tree Search (MCTS). The approach builds on two earlier agents. The first, *TransZero*, replaces MuZero's recurrent dynamics model with a transformer, allowing full rollouts to be computed in a single forward pass. The second, *TransZero with MVC* uses the Mean-Variance-Constrained (MVC) evaluator [5] to decouple how the search tree is built from how its nodes are evaluated. TransZero-Parallel combines both ideas with minor additional modifications to the MCTS, allowing for parallel expansion of entire subtrees.

Experiments showed that TransZero achieved sample efficiency and wall-clock speed comparable to MuZero. The addition of the MVC evaluator maintained this performance, although at the cost of increasing the planning time by 60%. TransZero-Parallel preserved sample efficiency while decreasing the planning speed by up to 11x, with similar reductions in the total training time. Theoretical analysis suggests that much larger planning speedups—over 100x—may be possible in certain cases, although how often this applies in practice is still unclear.

Future work could focus on scaling this method to more complex environments. Reducing attention computation time—possibly exponentially—and exploring alternative subtree expansion strategies are also promising directions. Lastly, incorporating techniques from EfficientZero [19] might further improve sample efficiency.

The primary objective was to reduce the inference time, thus enhancing the applicability of model-based reinforcement learning in settings where fast decision making is essential. This research represents a strong step towards this goal.

# Bibliography

[1] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

[2] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, 2006. URL `https://api.semanticscholar.org/CorpusID:16724115`.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL `https://arxiv.org/abs/1712.01815`.

[4] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265, 2019. URL `http://arxiv.org/abs/1911.08265`.

[5] R.A. Jaldevik. General tree evaluation for alphazero. Master's thesis, Delft Univeristy of Technology, 2024. URL `https://repository.tudelft.nl/record/uuid:5d5fd035-eed6-4176-85d3-f31deecb6133`.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. URL `https://arxiv.org/abs/2010.11929`.

[9] Lucas Beyer, Xiaohua Zhai, and Alexander Kolesnikov. Better plain vit baselines for imagenet-1k, 2022. URL `https://arxiv.org/abs/2205.01580`.

[10] Chang Chen, Yi-Fu Wu, Jaesik Yoon, and Sungjin Ahn. Transdreamer: Reinforcement learning with transformer world models, 2022. URL `https://arxiv.org/abs/2202.09481`.

[11] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination, 2020. URL `https://arxiv.org/abs/1912.01603`.

[12] Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are sample-efficient world models, 2023. URL `https://arxiv.org/abs/2209.00588`.

[13] Daniel Monroe and Philip A. Chalmers. Mastering chess with a transformer model, 2024. URL `https://arxiv.org/abs/2409.12272`.

[14] Yuan Pu, Yazhe Niu, Zhenjie Yang, Jiyuan Ren, Hongsheng Li, and Yu Liu. Unizero: Generalized and efficient planning with scalable latent world models, 2025. URL `https://arxiv.org/abs/2406.10667`.

[15] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations, 2018. URL `https://arxiv.org/abs/1803.02155`.

[16] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87608-3.

[17] Jinke He, Thomas M. Moerland, Joery A. de Vries, and Frans A. Oliehoek. What model does muzero learn?, 2024. URL `https://arxiv.org/abs/2306.00840`.

[18] Baohe Zhang, Raghu Rajan, Luis Pineda, Nathan Lambert, André Biedenkapp, Kurtland Chua, Frank Hutter, and Roberto Calandra. On the importance of hyperparameter optimization for model-based reinforcement learning, 2021. URL `https://arxiv.org/abs/2102.13651`.

[19] Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data, 2021. URL `https://arxiv.org/abs/2111.00210`.

[20] Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. Perceiver: General perception with iterative attention, 2021. URL `https://arxiv.org/abs/2103.03206`.

[21] Xinlei Chen and Kaiming He. Exploring simple siamese representation learning. *CoRR*, abs/2011.10566, 2020. URL `https://arxiv.org/abs/2011.10566`.