Model-Based Reinforcement Learning: MuZero

Sana Moin
sana.moin@informatik.uni-hamburg.de

Navneet Singh Arora
navneet.arora@informatik.uni-hamburg.de

Knowledge Processing in Intelligent Systems: Practical Seminar
Knowledge Technology, WTM
Department of Informatics, University of Hamburg

***Abstract*—MuZero is currently one of the best performing deep Reinforcement Learning (RL) algorithms with discrete actions. A model-based RL algorithm, foreseeing an internal representation and future rewards for planning via Monte Carlo Tree Search (MCTS) despite not predicting the state itself. It uses the environmental response to alter its behavior compared to model-free models. This paper deals with the experimentation and analysis on six different grid and non-grid based scenarios involving various games to explore the strengths and weaknesses of MuZero in different situations.**

I. Introduction

Model-free RL has had great success recently, where the planning is done based on the knowledge of the environment's dynamics. However, this exposure to knowledge prevents direct implementation of these planning algorithms to real-world domains like robotics or intelligent systems. Conversely, model-based RL addresses this issue by first learning the environment dynamics and then planning/predicting based on the learning environment [3].

MuZero builds on top of the model-based RL approach by incorporating *AlphaZero's* [5] search and search based policy iteration. The algorithm works recurrently by considering the current observations, forming the hidden states through these observations, and considering the previous ones to predict the action, value function and, the immediate reward [3].

A. Model-based Reinforcement Learning

The term model-based or model-free strictly alludes to the use of predictions of the environmental response by the agent during the training. If the model applies these responses to alter its subsequent steps, it is model-based learning. In other words, the agent collects data from past experiences through interaction, creates tuples of (state, action, and next state) and it uses it to learn a predictive model given the current state and actions.

Maximum likelihood-based training is done for the future states using both, the current action state as well as the current action sequence. The expected value of the reward are maximized for planning. In other words, it aims to learn a model of the environment's dynamics and then plan concerning the learned model [3].

B. Monte Carlo Tree Search

MCTS algorithm is used to look out for the set of nodes in a tree that can be the optimal solution of a game policy. It first selects a node with the highest winning probability, then expands this node which corresponds to the next moves in the game and then selects one of those expanded nodes by exploration through reinforcement learning. The game playout keeps on expanding and accumulating the best reward through exploration. The resultant scores, or the accumulated rewards, are backpropagated to the nodes of the tree that aid in the future selection process. Essentially, it tries to find the high rewarding trajectories in the search tree through Monte Carlo Simulation instead of using expensive brute force.

In this paper, we first describe the prior work done around the MuZero algorithm and how the algorithm progressed over time from AlphaGo to MuZero. In the next section, we will describe the MuZero algorithm in detail and also how the latent space of the model can be visualized. We then describe the experiments done on the model for grid and non-grid based games, present the corresponding results and finally conclude with the discussion and conclusion section.

II. Prior Work

MuZero is the latest addition in the series of RL algorithms developed by DeepMind. However, there is not only progression in the improvement of these algorithms, starting from *AlphaGo* [4] but also a pattern. This progressive pattern is vital in visualizing the model-based nature of this approach and similar super-human performance. All of them are built on the MCTS algorithm, each one combining the capabilities of MCTS in its way to enhance the overall performance.

A. AlphaGo

It is the first among Deepmind's series of RL algorithms and lays the foundation for MuZero. It was the first program ever to defeat a professional Go player [4]. This model comprises of both the Supervised Learning (SL) module and RL module. Here, SL takes the human/expert moves providing immediate feedback. It bases on the position-feature vector mapping computed using a convolutional neural network

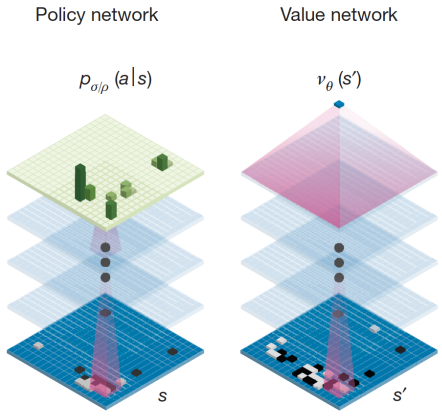

Figure 1. Schematic representation of AlphaGo's neural network architecture [4] with separate policy and value networks. The representation of the board positions is taken as an input for the policy network, passes it through many convolutional layers with parameters (σ) (SL policy network) or (ρ) (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, shown by a probability map over the board. The value network similarly uses man convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that then predicts the expected outcome for position s' [4].

(CNN) for every possible board position. On top of it, RL value-policy network training (Figure-1) further improves the outcome. The policy network aims at improving the end-goal of winning the game and, the value network provides the probability distribution for action over a sampled state (s, a). It happens during the self-play procedure (current and previous policy gradient where the network plays with itself).

B. *AlphaGo Zero*

The second stepping stone is the *AlphaGo Zero* algorithm, which works entirely on RL, without the SL used in *AlphaGo*. Significant differences include three different aspects. Firstly, the network runs solely on RL self-play (Figure-2) without the SL, as mentioned above. Secondly, policy and value networks, which were separated modules in the *AlphaGo* are combined to form a single network (Figure-3). And finally, it replaces Monte-Carlo rollouts with lookup search for stable internal loops and learning [6].

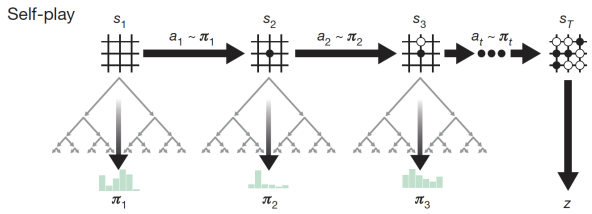

Figure 2. The program plays a game with itself. s_1 to s_t are multiple self-play games which the model plays using the MCTS and updated neural network learnings. Thus, computing and selecting the best possible probabilities for winning the game. The terminal position s_t is scored according to the rules of the game to compute the game winner z [6].

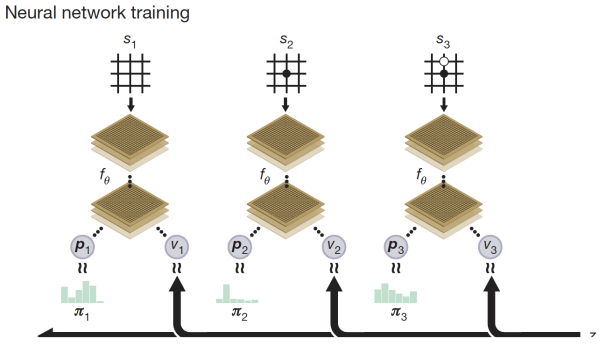

Figure 3. The figure depicts the single network architecture computing both the policy and value vector. Different board positions are the input, similar to that of self-play games. Hence, updating the network. It is further applied to execute the next self-play game [6].

C. *AlphaZero*

The third stepping stone is the *AlphaZero*. It uses the same *AlphaGo Zero* network but instead only provides general rules for multiple games, compared to the domain-specific rule information provided in *AlphaGo Zero*. In other words, it is a generic implementation of *AlphaGo Zero*.

Other minor differences include the non-assumption of symmetry, no training data augmentation, and exclusion of binary outcome assumption. Symmetrical non-assumption considers that both shogi and chess are asymmetric but go is not. No Augmentation means that training data contains different corner positions of the game board with no SL. Games ending up in a draw are accounted for in *AlphaZero* compared to *AlphaGo Zero*. Hence, the output is non-binary.

Despite these subtle differences, the same CNN architecture as *AlphaGo Zero* was used for all three games achieving better performance and also demonstrating the general-purpose learning of the RL algorithm.

III. MuZero: Generalized Approach

Finally, the milestone is the generalized approach of *MuZero*. This approach removes the basic game rules supplied in all previous methods and still achieves state-of-the-art performance. It involves the iterative updating of hidden states created from the input observation received by the model. The model tries to predict only the relevant aspects for future planning compared to the entire probability distribution predicted in previous versions of *AlphaGo*, *AlphaGo Zero* and *AlphaZero*.

The model works on hidden states and is represented by the representation function, dynamics function and prediction function [3]. The model takes the initial board position as the input. This input is passed into a representation function h which is taken up by the function, along with any past observations, and converts it into a hidden state s_0. Using this

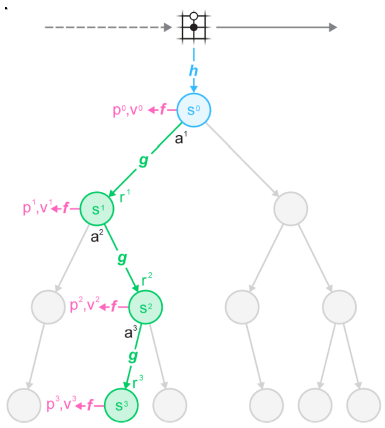

Figure 4. Planning state of the model. Uses the input to form a hidden state representation in order to compute the policy and value vector for each such state.

hidden state and a candidate action a_k, the dynamics function g (Equation-1) produces an immediate reward function r_k and next hidden state s_k. The prediction function f produces the subsequent value v_k and policy p_k from the hidden state.

$$r_k, s_k = g_\theta(s^{k-1}, a^k) \tag{1}$$

The model works on hidden states. Given a previous hidden state (initially, board position as an input is converted to a hidden state s_0), the model dynamics function g as presented in (Figure-4) produces an immediate reward function $r1$ using a candidate action a_1. Henceforth, creating a new hidden state s_1. Every hidden state created is used to compute the policy vector p_1 and value vector v_1.

MCTS is used to search for the optimum policy network, after all the hidden states have finished their computation starting from the root node, taking into consideration the visit count of each node in Figure-4. It is applied to search for the action proportional to the optimum policy (Figure-5). This action generates a new observation and a reward U_{t+1}. All these observations and rewards are stored in the replay buffer of the model in the form of trajectories. Finally, this replay buffer helps train the model by taking all the past observations of the sampled trajectory. Hence, the model is back-propagated jointly for the above-mentioned steps to get the final action and reward.

IV. Experiment Setup

To test MuZero and its performance, we choose a set of scenarios for training and playing, against itself or with a human. We aim to train different games to compare how the model learns the rules by looking at their rewards and hyper-parameters. The games use OpenAI Gym environment for the configuration [1]. All the configuration files are prepared and set up separately for each game/scenario. It includes hyper-parameters and environment integration with the model.

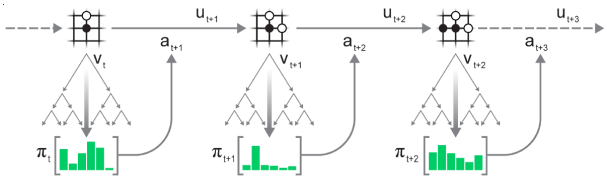

Figure 5. Acting state of the model. Using the searched optimum policy through MCTS, the model computes the proportional action and acts upon the environment to get the immediate actual reward.

A. Games/Scenarios Trained

The following games are selected to observe the rewards and loss values. It also includes the number of training steps required and self-play analysis. The existing configuration remains unchanged for all of these games. Ubuntu with 32 GB RAM, Intel i9 and, RTX 2070 Mobile testing works as a test environment.

- Grid Based Games:
  - Simple Grid: 3x3 grid game.
  - Gridworld: 4x4 grid game.
  - Tic-tac-toe: 3x3 grid game.
- Non-Grid Based Games:
  - Cartpole: Inverted pendulum, where the goal is to prevent it from falling over.
  - Lunalander: Land a vessel with the correct speed and angle.
  - Twenty-One: Blackjack, a card game.

B. MuZero Latent Space Representation

We now know that the model generates the representation of the embedding that helps in learning the value function efficiently. Moreover, we can better understand how these were structured in the latent space if we could visualize them. We, therefore, analyze a model that helps us do that [2]. We use the tool provided by them to visualize how the embeddings look after Principal Component Analysis(PCA) is applied in 4D space for the game Mountain car. We observe that the agent eventually covers the entire latent space after complete training and the group states are dynamically close, and hence it can retain the dynamics of the game. The visualizations and their further description can be seen in the Appendix section C.

V. Results

A. Rewards and Loss

The total rewards per training step for the games are shown in graphs, which show a comparison among all grid games and non-grid games in Figure-6. Figure-7 shows a total weighted loss for all grid games and all the non-grid games per training steps.

It can be seen clearly from Figure-6 that the model receives consistent rewards throughout the training time steps for the grid-based environment. However, for the non-grid games, the rewards received are much more dynamic. Combining these observations with the re-analysis hyper-parameter (Table-I), it can be seen that the model requires much more analysis for the non-grid games to achieve similar performance. Another aspect that indicates towards similar statement is the number of self-played games required per training step, which was way more than the ones observed for non-grid games.

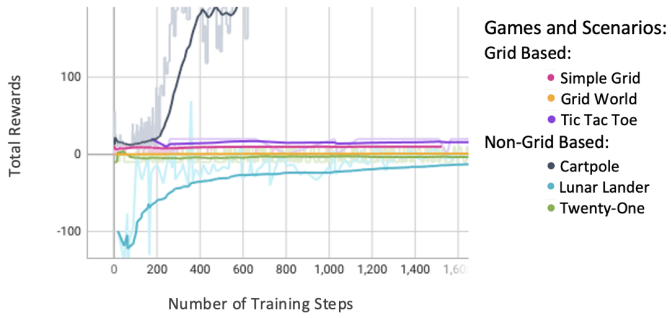

Figure 6. Total Reward Comparison for the games including both grid and non-grid based scenarios. Here the line trends are smoothed with the shadows showing the actual trend of the reward values.

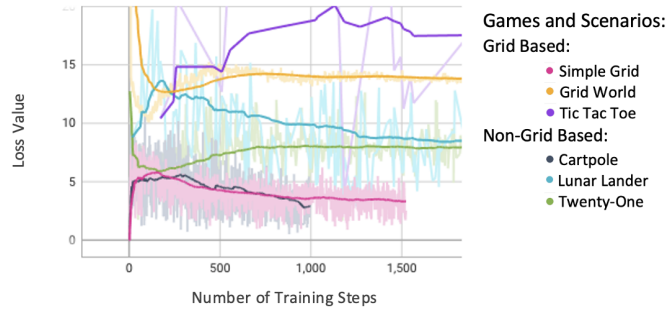

Figure 7. Total Weighted Loss comparison for all the games including both grid and non-grid based scenarios. Here the line trends are smoothed with the shadows showing the actual trend of the reward values.

VI. Discussion

In the results, we show the comparative charts for the games that are grid-based and that are non-grid based. For the grid-based, the rewards (Figure-6) for the games *Simple Grid* and *Grid World* eventually come to a consistent value after some training steps whereas for the *Tic Tac Toe* game, it fluctuates throughout the training steps. It is because there is a chance of a draw and hence the variations in the rewards.

For the non-grid games, we see a sharp increase in *Cartpole* game as the time steps increase, but similar behavior is not noticed in the other two games, namely *Lunalander* and *Twenty-One*. For *Twenty-One*, there is a consistent reward given throughout the training steps but there is a small incline seen for *Lunalander*.

For the total weighted loss, the loss slowly diminishes for *Simple Grid* and *Grid World* but it goes back and forth for *Tic Tac Toe*. It could be for the same reason as it was for the rewards. We can infer that different games and their complexity influences the way training works and how the network learns the policies.

There are a lot of technicalities present in the *MuZero* algorithm. With merged policy and value networks, absence of game rules, hidden state computation, and the involvement of self-play games. Therefore, it requires a lot of understanding in-order to completely understand the working of this model.

A. Strengths

The MuZero algorithm achieves a state-of-the-art performance surpassing its previous set of algorithms even with less information. Not only is the model able to achieve more but it is also able to compute a lot of other intermediary information through the hidden states. Apart from interacting with the environment and fetching the rewards, the model also utilizes the past stored memory (past observations) to further improve its hidden state, policy, and value computation before making the subsequent actions. This much consumption of information and computation makes it one of a kind. These key components also make it possible for the model to generalize to any given model which its predecessor algorithms were unable to do.

B. Weaknesses

Even though the model is state-of-the-art in terms of its capability and performance, it still has its limitations. Because of real-time updating of the weights to adapt the policy, this model is not only computationally expensive but even restricts much further research to optimally learn an environment. Computation of hidden states is also one of the key components of this model but it also makes it hard to understand as the complete working of these hidden states is still not known and works as a black-box.

VII. Conclusion

In this paper, we outlined the concepts around the MuZero algorithm and also shed a light on how the algorithm came to the current state following the previous algorithms that aimed to solve similar tasks, namely AlphaGo, AlphaGo Zero, and AlphaZero. We experimented with the MuZero algorithm on a set of games, distinguished by the property of them being grid or non-grid based. We concluded that multiple factors determine the performance of the algorithm, as well as analyzed the strengths and weaknesses of this model. We also briefly looked at the internal latent state representation of the model and its implications. We further concluded that there are many unexplored factors in the algorithm that can be investigated.

REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Joery A. de Vries, Ken S. Voskuil, Thomas M. Moerland, and Aske Plaat. Visualizing muzero models. *CoRR*, abs/2102.12924, 2021.

[3] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[6] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

APPENDIX

A. Hyperparameters

As shown in Table-I, these are the set of different hyperparameters which are used for a specific environment. It can be seen that the grid-based games have very specific hyperparameters compared to non-grid based games. For example, the amount of batch sizes needed for grid-based games is very specific to which scenario is being executed whereas it is much more generalized with the non-grid based scenarios. The same can also be noticed for the re-analysis.

Grid Game	Batch Size	Training Steps	Re-analyzed
Simple Grid	32	3e+4	1.24e+5
Gridworld	128	3e+4	0
Tic-tac-toe	64	5.2e+5	3.9e+6
Non-Grid Game	**Batch Size**	**Training Steps**	**Re-analyzed**
Cartpole	128	1e+4	5.46e+4
LunaLander	64	2e+5	2.95e+5
Twenty-One	64	7013	4.11e+4

Table I
MAJOR HYPER-PARAMETERS AND THEIR SUBTLE DIFFERENCES FOR BOTH GRID-BASED AND NON-GRID BASED GAMES/SCENARIOS. FOR GRID-BASED GAMES, HIGHER NUMBER OF TRAINING STEPS ARE NEEDED AS COMPARED TO THE NON-GRID BASED GAMES. ON THE OTHER HAND, NON-GRID BASED GAMES REQUIRE MORE RE-ANALYSIS.

B. Learning Rate

Figure-8 shows the type of learning rates used for training. It is fascinating to note that apart from *Cartpole* which uses decaying learning rate, all other scenarios rely more or less on the constant learning rate.

C. Latent Space Visualization

The latent space is visualized in two different ways. In Figure 9, we see a 4D latent space of representation after PCA is applied. The colors show the learned values of each state. We can also see a green line that goes through the plot, which shows the trajectory followed after training is complete. Figure 10 shows the relation between true state and latent space by their value estimate near convergence. We can see that similar values are grouped. The line through the plot represents the trajectory of the agent.

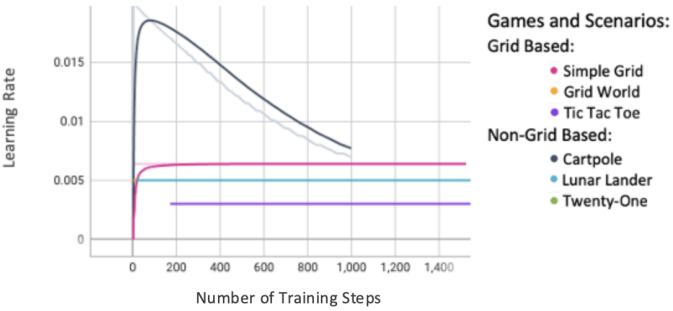

Figure 8. Learning Rate comparison for all six game scenarios. Cartpole is the only scenario where decaying learning rate is used.

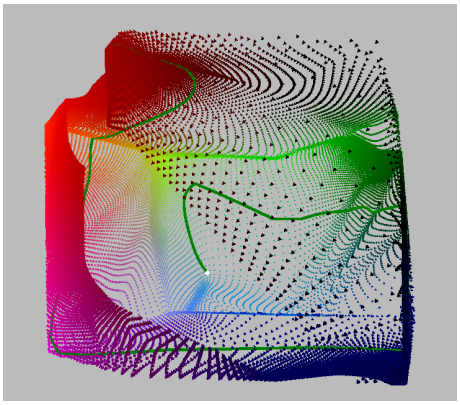

Figure 9. 4D latent space representation for Mountain Care game after PCA

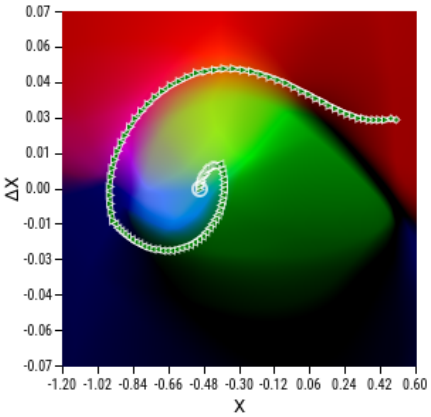

Figure 10. Cart position vs velocity plot for value estimates near convergence