Moving towards Generalized A.I. [MuZero]

A Practical Paper Project

Che-Hsien Lin
205429284
ECE, UCLA
iced2000@ucla.edu

Neng Wang
505427801
ECE, UCLA
nengwang19@ucla.edu

Pratyush Srivastava
504946196
MAE, UCLA
pratyushucla@ucla.edu

Rahul Shenoy
405028879
MAE, UCLA
rahulshenoy77@gmail.com

Abstract

Tree-based planning methods have shown great success wherever a perfect simulation of the environment can be performed. However, in practical world problems, environment behavior can be quite complex or unknown. Reinforcement learning methods solve this issue by either constructing a model of the environment in Model-based approach, or by estimating the policy / value function by interacting with the environment in Model-free approach. Model-free RL have shown better performance than model-based in visually rich domains like Atari games, but still haven't reached the state-of-the-art in domains which require precise and sophisticated lookahead, like board games. In the current work, we present our implementation of the MuZero algorithm which recently achieved a new state of the art among all planning approaches when implemented on both board and Atari games. By learning a model and iteratively evaluating the rewards, action-selection policy and value, it has shown to perform better than Model-free RL methods. In order to analyze the generalization performance of the algorithm, we have evaluated it on "non-game" environments such as classical controls and robotics.

1 Introduction

MuZero [10], the latest model-based reinforcement learning algorithm released on 19th November 2019 by DeepMind, is a very crucial milestone towards General Artificial Intelligence. Similar to AlphaZero [11], MuZero denies itself human expert strategy to learn from, however, it is not aware of rules of the game. It is therefore able to mimic human intelligence: when you learnt to swim, you weren't given the rulebook to fluid dynamics first, or when you learnt to build towers with blocks, you weren't prepped with Newton's laws of gravity. Previously, model-based AlphaZero achieved super human performance in board games like Chess, Shogi and Go, where it had access to a perfect simulator of the environment (or rulebook of the game) and played against itself using Monte Carlo Tree Search (MCTS). State of the art performance in Atari games was achieved by model-free RL algorithms like Impala [2], R2D2 [7], and Ape-X [5], by estimating the optimal policy and/or value function based on their interaction with the environment. But these algorithms are far from state of the art in domains where a precise and sophisticated look-ahead is required like in chess or shogi. MuZero bridges this gap between model-based and model-free reinforcement learning, by preserving AlphaZero's powerful search and search-based policy iteration algorithm (MCTS), and using a learned model trained on its interaction with the environment instead of a perfect simulator. Thus, MuZero not only achieved state of the art performance in visually rich domains like Atari 2600, but maintained super human performance in precision planning tasks like chess and shogi.

The main idea of the algorithm is to use MCTS to expand every observation state (such as current snapshot of the game image) into a tree of hidden states which don't have to represent the real environment, then implement three deep neural networks to predict quantities directly relevant for planning: value, policy and reward, at each of those hidden states. Therefore the observation becomes an input, which is transformed in a hidden state using one of the networks called "representation function". This first hidden state is called a "root node". Second network called "prediction function" predicts the value and policy at the current hidden state, whereas the third

network "dynamics function" facilitates in tree expansion by predicting the next hidden state along with the reward for taking some action. These networks are trained end-to-end on the "targets" generated using TD learning over the actual rewards obtained upon interaction with the environment. In this way, the agent can invent, internally, the rules or dynamics that lead to most accurate planning.

In this practical project, we explain in detail about the MuZero algorithm, how it works, and then implement it on non-game environments on Open AI Gym, such as classical controls, Gridworld, and Box2D. We built the entire working code from scratch using the pseudocode provided by the Deepmind team. The report is organized as follows. In section 2 we discuss the previous reinforcement learning approaches which achieved superhuman performance in visually rich domains, such as Atari. In section 3, we explain the pseudocode of MuZero and discuss the key features which make this algorithm successful in achieving state-of-the-art performance. We then present our implementation of the algorithm on different environments and the results in section 4. Concluding remarks are finally given in section 5.

2 Prior Work

To use reinforcement learning successfully in real world situations, agents face a difficult task: derive efficient representation of the environment leading to optimal planning and also generalize past experience to new situations. This task becomes even more difficult if the domain is not fully observable due to lack of sensory feedback. In such cases, Model-free RL approaches enjoy great success as they don't care about environment representations. Atari games present almost the same amount of complexity in environments and numerous algorithms have been developed to achieve superhuman performance in them. Mnih et. al. [9] introduced a deep Q-network (DQN) that can learn successful policies directly from high-dimensional sensory inputs. The DQN agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. It leveraged experience replay for data efficiency and stacked a fixed number of consecutive frames to overcome the partial observability in Atari.

While being widely successful, DQN progress has been primarily in single task environment and one of the main challenges in training a single agent on many tasks at once is scalability. Another key challenge is to handle the increased amount of data and extended training time. Espeholt et. al. [2] developed a new distributed agent IMPALA (Importance Weighted Actor-Learner Architecture) that not only uses resources more efficiently in single machine training but also scales to thousands of machines without sacrificing data efficiency or resource utilisation. IMPALA uses a first-in-first-out queue with a novel off-policy correction algorithm called V-trace, to learn sequentially from the stream of experience generated by a large number of independent actors. Espeholt et. al. showed that IMPALA could achieve strong performance in the Atari-57 and DMLab-30 benchmark suites, and furthermore was able to use a single large network to learn all tasks in a benchmark simultaneously while maintaining human-level performance.

Human players can learn to play visually rich games like Atari in minutes [12]. However, above mentioned model-free reinforcement learning algorithms require tens or hundreds of millions of time steps - equivalent of several weeks of training in real time ([2], [5], [7]). How is it that humans learn so fast? The key element that allows humans to learn so fast is that they have an intuitive understanding of the physical processes that are represented in the game: we know that planes can fly, balls can roll and bullets can destroy aliens. We can therefore predict the outcome of our actions. In other words, this intuitive understanding translates to a prelearned dynamics model of the environment based on previous interactions with similar environments (other games or real life interactions) which helps us learn a new game in minutes. This is why, lot of recent research effort has been focused towards model-based reinforcement learning algorithms that first learns a dynamic model of the environment. These algorithms are more promising to efficiently extract valuable information from available data, making them more data-efficient compared to model-free methods.

Model-based reinforcement algorithms have typically focused on reconstructing the true environment state([1], [4], [8]). A major problem model-based methods suffer from is model bias i.e., they inherently assume that the learned dynamics model sufficiently accurately resembles the real environment. PILCO [1] copes with this problem by learning a probabilistic dynamic model to incorporate the model uncertainty into planning and policy evaluation. In another work, the model bias is dealt with by computing value gradients along real system trajectories instead of planned ones and use model to only compute policy gradient thus combining the benefits of both model-free and model-based methods [4]. Even though these algorithms were able to cope up with the problem of model bias, they still focused on reconstructing the true environment state. The learning tasks in the above works were relatively low dimensional control tasks where the states were position, velocity which have

much more information relevant to the task encoded in them compared to pixels which is the input in visually rich tasks like Atari games. It has been hypothesized that deep, stochastic models may mitigate the problems of compounding error [3], [6]. However, planning at pixel-level granularity is not computationally tractable in large scale problems. None of these prior methods has constructed a model that facilitates effective planning in visually complex domains such as Atari; results lag behind well-tuned, model-free methods, even in terms of data efficiency.

The key to the success of MuZero is that it doesn't actually try to predict the next true state but instead transforms the observations into a hidden state. The main idea is to construct an abstract MDP model such that planning in the abstract MDP is equivalent to planning in the real environment. This equivalence is achieved by ensuring value equivalence, i.e. that, starting from the same real state, the cumulative reward of a trajectory through the abstract MDP matches the cumulative reward of a trajectory in the real environment. In this way, instead of predicting the next entire environment state, only the aspects relevant to planning are predicted via the hidden state.

3 MuZero Algorithm

At a higher level, Muzero uses three deep neural networks to learn a model of the environment, and uses this model to plan and act. The overall architecture of the algorithm can be summarised in three major steps, also shown in figure 1:

1. **Planning**: The model consists of three connected components for representation, dynamics and prediction. Given a previous hidden state s^{k-1} and a candidate action a^k, the dynamics function g produces an immediate reward and a new hidden state s^k. The policy p^k and value function v^k are computed from the hidden state s^k by a prediction function f. The initial hidden state s^0 which becomes the root of the Monte-Carlo tree, is obtained by passing the past observations into a representation function h.
2. **Acting**: A Monte-Carlo Tree Search is performed at each timestep t. An action a_{t+1} is sampled from the search policy π_t, which is proportional to the visit count for each action from the root node. The environment receives the action and generates a new observation and reward u_{t+1}. At the end of the episode the trajectory data is stored into a replay buffer.
3. **Training**: A trajectory is sampled from the replay buffer. For the initial step, the representation function h receives as input the past observations $\{O_1, ..., O_t\}$ from the selected trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, the dynamics function g receives as input the hidden state s^{k-1} from the previous step and the real action a_{t+k}. The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by backpropagation-through-time, to predict three quantities: the policy $p_t^k \approx \pi_{t+k}$, value function $v_t^k \approx z_{t+k}$, and reward $r_t^k \approx u_{t+k}$, where z_{t+k} is a sample return. The loss function thus becomes,

$$l_t(\theta) = \sum_{k=0}^{K} l^r\left(u_{t+k}, r_t^k\right) + l^v\left(z_{t+k}, v_t^k\right) + l^p\left(\pi_{t+k}, p_t^k\right) + c||\theta||^2 \tag{1}$$

3.1 Detailed analysis of the algorithm

A summarized process flow is shown in algorithm 1. As shown in the process flow, MuZero algorithm can be broadly divided into two components: *self-play* (generating game data) and *training* (producing improved versions of the learned environment/game model and value functions). The algorithm then creates two important objects: `ReplayBuffer` and `SharedStorage`. `ReplayBuffer` stores data from previous games generated during *self-play*, whereas, `SharedStorage` object saves a version of the neural network and helps in retrieving the latest neural network. In subsequent subsections we will be discussing in detail the pseudo-code for these two important components.

3.1.1 Self-Play

First, a new `Game` object is created and the main game loop is started. The game ends when it either reaches a `terminal_state` or has played `max_moves`. At each time step, we ask the game to return current observation (o_t). Next, the `initial_inference` ($h + f$) function is called to give us the root node (s^0) for the MCTS tree. The root node is expanded using the known legal actions provided by the game. An exploration noise is also

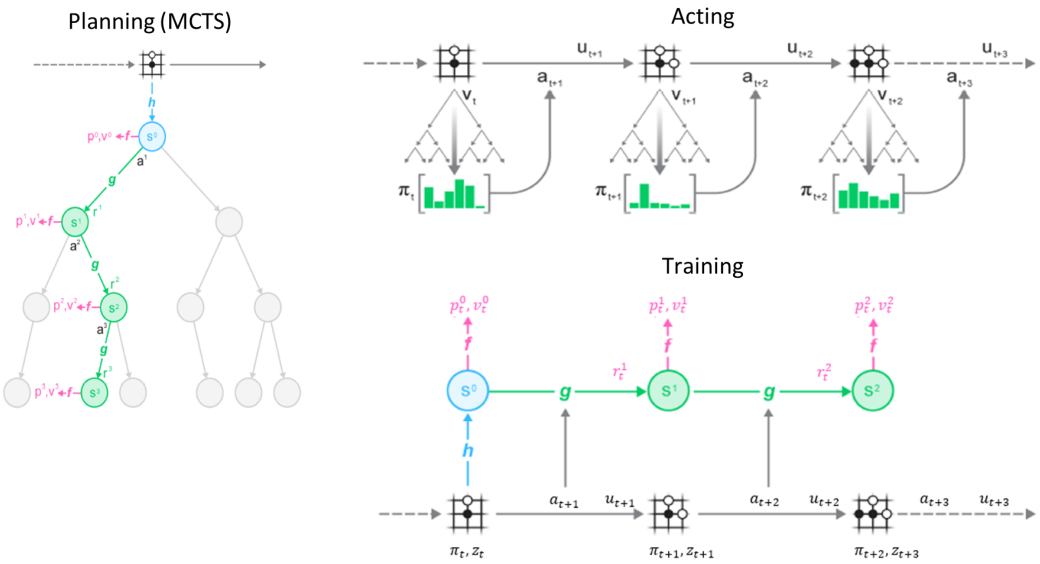

Figure 1: **How Muzero plans, acts and trains its networks.** It creates numerous parallel MC trees using the 3 deep neural networks (h, g, f) and uses it to select an action based on its visit count. Then the actual rewards collected based on its interaction with the environment are converted into "targets" (u, z, π) using TD learning, which are then used to train the networks. (Taken from [10] with some modifications).

added to the root node actions to ensure that the MCTS explores a range of possible actions rather than only exploring the action which it currently believes to be optimal. We now hit the main MCTS process.

As MuZero has no knowledge of the environment rules, it also has no knowledge of the bounds on the rewards that it may receive throughout the learning process. The `MinMaxStats` object is created to store information on the current minimum and maximum rewards encountered so that MuZero can normalise its value output accordingly. Alternatively, this can also be initialised with known bounds for a game such as chess (-1, 1). The main MCTS loop iterates over `num_simulations`, where one simulation is a pass through the MCTS tree until a leaf node (i.e. unexplored node) is reached and subsequent backpropagation. Let's walk through one simulation now.

First, the history is initialised with the of list of actions taken so far from the start of the game. The current node is the root node and the `search_path` contains only the current node.The simulation then proceeds as follows:

1. Traverse from root using actions with maximum UCB scores until the leaf node is reached.
2. Call `recurrent_inference` $(g + f)$ function on the parent of the leaf node with the chosen action.
3. Leaf node is expanded - the policy priors are written to the newly created children nodes.
4. The value of the leaf node and collected rewards are backpropagated to the root node.

The UCB score is a measure that balances the estimated value of the action $Q(s, a)$ with a exploration bonus based on the prior probability of selecting the action $P(s, a)$ and the number of times the action has already been selected $N(s, a)$. This completes one simulation of the MCTS process. After `num_simulations` passes through the tree, the process stops and an action is chosen based on the number of times each child node of the root has been visited. Though the number of visits may feel a strange metric on which to select the final action, it isn't really, as the UCB selection criteria within the MCTS process is designed to eventually spend more time exploring actions that it feels are truly high value opportunities, once it has sufficiently explored the alternatives early on in the process.

The chosen action is then applied to the true environment and relevant values are appended in the `Game` object. This process continues, creating a new MCTS tree from scratch each turn and using it to choose an action, until the end of the game. All of the game data (`rewards`, `history`, `child_visits`, `root_values`) is saved to the `ReplayBuffer` and the actor is then free to start a new game.

Algorithm 1: MuZero, Process Flow Summary

Parameter : Initialize MuZero(num_actors, num_sim, td_steps, unroll_steps, max_moves, action_space), Shared_storage(), Replay_buffer(), Network(), Node(visits, prior, value, children, reward, hidden_state);

Parameter : Initialize Networks: h, g, f with random parameters;

for ***Parallel*** *Game play and Network training* **do**
  while *play game* **do**
    Extract latest Networks (h, g, f) from Shared_storage ;
    for ***Parallel*** *self play games in num_actors* **do**
      for *play game until terminal or max_moves* **do**
        O_t = current game image;
        for *num_sims* **do**
          root = $s_0 \leftarrow h(O_t)$, policy, value = $p_0, v_0 \leftarrow f(s_0)$;
          Create a monte-carlo tree search starting at s_0 up to K (initially 1) nodes;
          for $k = 0 \rightarrow K - 1$ **do**
            $a^k \leftarrow$ UCB(s^{k-1}), store path (s^k) and actions (a^k);
          end
          $s^K, r^K \leftarrow g(s^{K-1}, a^K)$, and $p^K, v^K \leftarrow f(s^K)$ expand tree (increase K) by adding node s^K to the path ;
          Backpropagate values on the path all the way to the root node ;
        end
        Select action a_{t+1} according to $\pi(a_{t+1} = a) \leftarrow \frac{N(a_{t+1}=a)}{\sum(N(a_{t+1}))}$, where N is child node visits ;
        Play game using a_{t+1} and store O_t statistics (value, action, policy, reward);
      end
      save Game() in Replay_buffer();
    end
  end
  while *Train network and update weights* **do**
    Make targets (value, policy, actual reward) for the network up to unroll_steps using TD learning (up to td_steps) and store in Replay_buffer();
    Create training batch from Replay_buffer (batch_size, targets) using prioritized replay;
    for *state, actions, targets in batch* **do**
      Value, reward, policy = initial_inference ($h()$ and $f()$);
      for *action in actions* **do**
        Value, reward, policy = recurrent_inference ($g()$ and $f()$) # runs for unroll_steps ;
      end
      Loss += scalar_loss(value, target_value) + scalar_loss(reward, target_reward) + ... Cross_Entropy(policy, target_policy);
    end
    Loss += L2_regularization(**for** weights **in** Network.weights);
    Optimizer.minimize(Loss);
    save Network() in Shared_storage();
  end
end

3.1.2 Training

Network *training* happens in parallel with *self-play*, where the function `train_network` continually samples the data from the `ReplayBuffer`. It first creates a new `Network` object (which stores randomly initialized networks of the algorithm) along with the decay in learning rate based on training steps completed. For updating weights of the network, stochastic gradient descent with momentum optimizer is used. Finally this function iterates over `training_steps` (=1,000,000 in the paper, for chess). At each step, it samples a batch of positions from the replay buffer and uses them to update the networks. The updated network parameters are stored every `checkpoint_interval` in `Shared_storage`. Next we explain how the algorithm creates a batch of training data and updates the weights of the neural networks.

Creating a training batch
The `ReplayBuffer` class contains a `sample_batch` method to sample a batch of observations from the buffer. The default `batch_size` of MuZero for chess is 2048. This number of games are selected from the buffer and one position is chosen from each. A single batch is a list of tuples, where each tuple consists of three elements:

- `g.make_image(i)` — the observation at the chosen position, `i`
- `g.history[i:i + num_unroll_steps]` — a list of the next `num_unroll_steps` actions taken after the chosen position (if they exist)
- `g.make_target(i, num_unroll_steps, td_steps, g.to_play())` — a list of the targets that will be used to train the neural networks. Specifically, this is a list of tuples: `target_value`, `target_reward` and `target_policy`.

Each observation in the batch is unrolled `num_unroll_steps` into the future using the actions provided. The `make_target` function then uses TD learning to create estimates of actual value, reward and policy, which basically become "targets" for the neural networks to train on. For the first position, `initial_inference` ($h + f$) function is used to predict the value, reward and policy and compared with the target value, target reward and target policy. For the unrolled positions, the `recurrent_inference` ($g + f$) function is used to perform the same task. This way, all three networks are trained and updated end-to-end. It is noteworthy that TD learning approach to create targets leads to bootstrapping.

After targets are created, next step is to compute the loss function and optimize the networks.

Updating the three MuZero networks
The `update_weights` function builds the loss piece by piece, for each of the positions in the batch. First the initial observation is passed through the `initial_inference` network to predict the value, reward and policy from the current position. These are used to create the predictions list, alongside a given weighting of 1.0. Then, each action is looped over in turn and the `recurrent_inference` function is asked to predict the next value, reward and policy from the current `hidden_state`. These are appended to the predictions list with a weighting of `1/num_rollout_steps` (so that the overall weighting of the `recurrent_inference` function is equal to that of the `initial_inference` function).

We then calculate the loss that compares the predictions to their corresponding target values — this is a combination of `scalar_loss` for the reward and value and `softmax_crossentropy_loss_with_logits` for the policy. The optimiser then uses this loss function to train all three of the MuZero networks simultaneously.

4 Implementation and Results

Based on the pseudo code provided by the authors of the original paper [10], we implemented MuZero algorithm with Pytorch and Open AI Gym. Instead of classic board games and Atari games, we applied our implementation on "non-game" environments to demonstrate the generalization performance. First, we chose CartPole, which is the simplest task of classic control, to verify the correctness of our code. Then, we trained our model on LunarLander and FrozenLake, which belong to Box2D and planning problem respectively.

Table 1 summarizes the hyperparameters of the three tasks. For the purpose of efficiency and considering the shape of observations, we utilized fully connected network instead of convolutional network to implement the representation, dynamics and prediction model. As for activation function, we chose LeakyReLU with slope 0.01.

Figure 2 shows the performance of three tasks. As shown in figure, MuZero algorithm successfully solved all of them. For CartPole, the maximum reward is 500 limited by the number of maximum moves, and our model got mean reward of 400 across 10 independent experiments. For LunarLander, generally the problem is considered solved when the reward is over 200, we obtained the mean reward of over 200 after 80000 steps. For FrozenLake, since its reward is either 0 or 1, we focus on its density instead of value. The result shows that after 15000 steps there is a high probability for the model to find the goal. However, it does not always obtain reward 1 comparing to other simple reinforcement learning algorithms such as value iteration.

To have a better understanding of MuZero, we ran two experiments to test the performance of different simulation times of MCTS. Due to time constraint and the binary reward of FrozenLake, we ran the first experiments on CartPole and LunarLander while the second experiment on Cartpole only. For the first experiment, we investigated the scalability of planning by testing the pretrained model on different thinking time, to wit, on

		CartPole	Lunarlander	Frozenlake
(A)	hidden state size	8	10	10
	representation hidden layers	0	0	0
	dynamics hidden layers	1	1	1
	reward hidden layers	1	1	1
	value hidden layers	0	1	1
	policy hidden layers	0	0	0
	hidden layers length	16	64	16
(B)	simulation times	50	50	50
	maximum moves	500	500	20
(C)	replay buffer size	100	1000	1000
	TD steps	50	50	50
(D)	batch size	128	32	32
	optimizer	SGD	SGD	SGD
	learning rate	0.05	0.1	0.1
	weight decay	1e-4	1e-4	1e-4
	momentum	0.9	0.9	0.9

Table 1: **Summary of hyperparameters for each task.** (**A**) Hidden state size is the shape of the encoded observation. Hidden layers denotes the number of hidden layers of each model, and 0 means the input directly connects the output. The dimension of each hidden layer is the same in a task. (**B**) Simulation times is the number of simulation times in MCTS, and maximum moves is the maximum steps in each game. (**C**) Replay buffer size denotes the number of self-play games to store in the buffer, and TD steps is the number of steps in the future that we need to consider when calculating the target value. (**D**) Following the original paper, we used SGD optimizer across all the tasks. Adam is another common training option, but we found little difference in the results.

different number of simulation of MCTS. The result is shown in figure 3. As we can see in the figure, the mean reward significantly increases as the number of simulations increase, which is similar to the result of Go and contrast with that of Atari in the original paper.

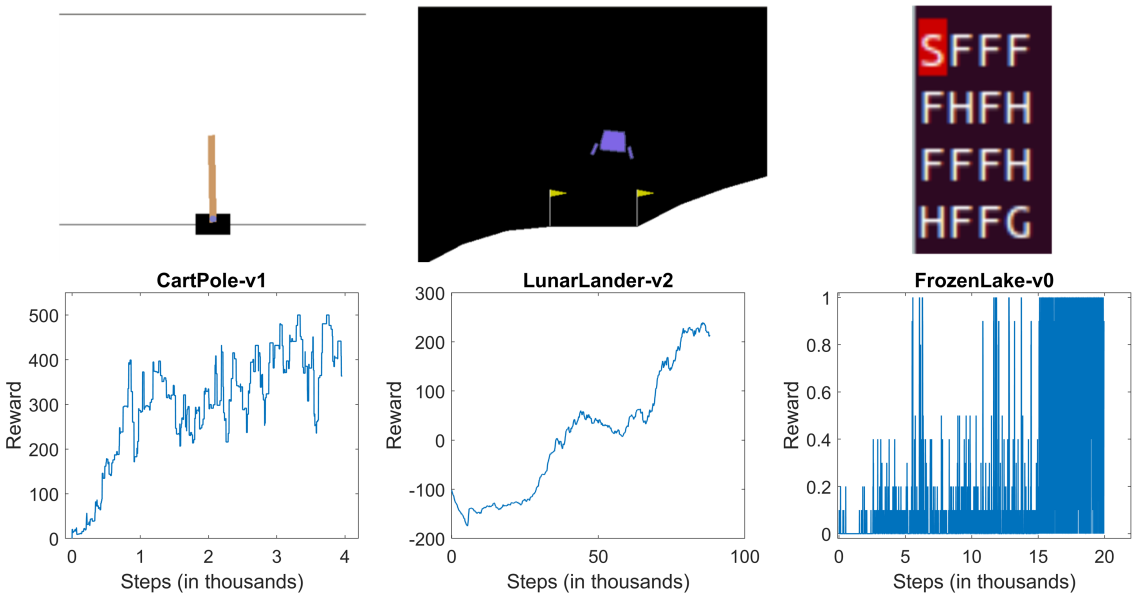

Figure 2: **Training results of CartPole-v1, LunarLander-v2 and FrozenLake-v0.** The figures below is drawn by averaging 10 independent training results. The x-axis is the training steps, and y-axis is mean reward of an episode in each training steps. All of the tasks are evaluated using 50 simulation per step.

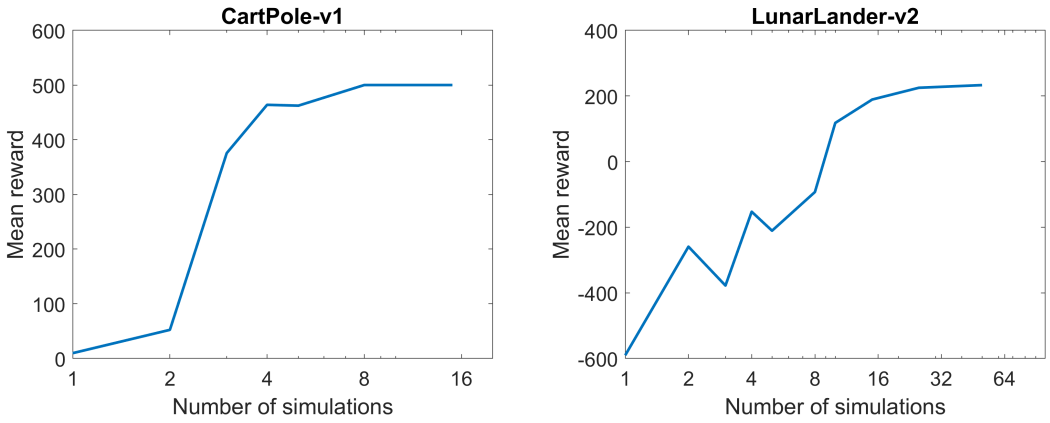

Figure 3: **Evaluation of scalability of planning** We use the same pretrained model which is trained on the hyperparameters listed in table 1 for testing. The x-axis is the number of simulations during test and the y-axis is the mean reward across 10 independent tests.

For the second experiment, we would like to know how the amount of search of MCTS during training affect the result. Since training takes a large amount of time, we only select Cartpole as the environment. Nevertheless, as shown in fig 4, the result shows that the model improves faster if we simulate more times in each step, and this is consistent with what we expect.

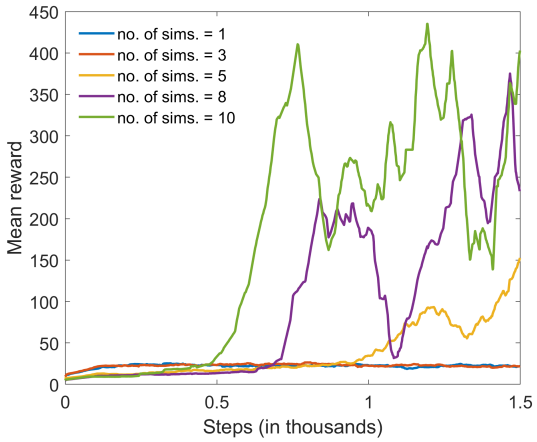

Figure 4: **Evaluation of different searching time during training.** We train different model on different number of simulation per move, and 50 simulations are applied during testing. The rest hyperpameters are the same as in table 1. The x-axis is the training steps, and y-axis is mean reward of 10 independent experiments.

4.1 Limitations

Although in this project we showed that MuZero has great generalization ability and its performance is great across all the tasks that we explored, it also requires a large amount of time for training due to MCTS. At the beginning of the project, we planned to test our code on Atari, but we found that it would take approximately one month to train a single model on our computer which is equipped with AMD ThreadRipper 1920X and Nvidia RTX 2080ti. For the three tasks we use in this project, CartPole takes 15 minutes, LunarLander takes 12 hours and FrozenLake takes 2 hours to train. Unless we have the hardware resources such as Google TPU, the application of MuZero is restricted to some simple tasks.

On the other hand, we tried our model on MuJoCo tasks such as Ant and Reacher, but failed to obtain any result. The reason we fail is that MuZero can only be used for discrete action space due to MCTS and visit count

approach for action selection. To solve this problem, discretizing the action space may be a possible solution, and this is left for future research.

5 Conclusion

In this project, we carried out an in-depth study on a recent state-of-the-art model-based reinforcement learning algorithm, MuZero.

- We surveyed various model-based and model-free RL methods for achieving super-human performance in games. Before MuZero, no single algorithm could achieve state-of-the-art in both board (planning based) games and visually rich games (like Atari).
- We understood how MuZero resembles human intelligence, where by learning a model it creates hidden states which only concern about parameters relevant for planning (i.e. value, policy and reward).
- We successfully implemented MuZero algorithm based on the pseudo code provided in the original paper and carried out experiments with different types of environments : Gridworld (FrozenLake-v0), Classic Control (CartPole-v1), and 2D Robotics (LunarLander-v2). The good performance we got from the trained model validates our implementation and shows MuZero's generalization potential to non-game environments.
- Although its performance is good and general, it takes too much computational power and time to achieve such a good performance. Also, in its current form, it can not be used in environments having continuous action space.

Code

Our code is at the GitHub link: `https://github.com/Noir97/ECE239AS-RL-Project_MuZero`

Author Contribution

All four authors contributed equally to this project.

References

[1] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[2] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[3] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

[4] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

[5] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[6] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[7] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. 2018.

[8] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[10] 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. *arXiv preprint arXiv:1911.08265*, 2019.

[11] 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.

[12] Pedro A Tsividis, Thomas Pouncy, Jaqueline L Xu, Joshua B Tenenbaum, and Samuel J Gershman. Human learning in atari. In *2017 AAAI Spring Symposium Series*, 2017.