Autonomous Learning of Physical Environment through Neural Tree Search

Bowen Fang
IEOR Department
Columbia University
UNI: bf2504

***Abstract*—The problem of active simultaneous localization and mapping (SLAM) is an important and challenging problem in the field of autonomous operations. The task involves actively guiding the agent to explore an unknown environment and build a map of the environment while localize the agent within that environment. Although several recent works have showcased the capacities of reinforcement learning(RL) method and the success of active slam based on RGB sensor, there is a lack of model-based method that has planning ability. Monte carlo tree search(MCTS)-based reinforcement learning has been shown to be highly effective on tasks where planning is required. However, due to its root in board games, most open source system follows the design of board game AI, which presents non-trivial difficulties to extend to wider range of reinforcement learning tasks. To enrich the toolkit of RL researchers and practitioners, in this project I present a new open source library Muax, a flexible and modular toolkit for mcts-based RL. I also proposed a Neural Tree Search(NTS) method for acitve SLAM, where a novel search process and a new loss function are used. Comparison experiments on Gibson exploration tasks are conducted through Habitat platform. The proposed method realized competitive performance against the state-of-the-art method Neural-SLAM with significant less training times, demonstrating the efficiency of the model-based method.**

***Index Terms*—intelligent robots, learning control systems, predictive control**

I. Introduction

Image you were a new student in Columbia and were asked to go to Mudd. How would you find the way? Probably as most people do, you will need a map. And if you have a map with your location within it, it's even better. It is kind of similar for the robotic tasks, where autonomous operations requires the agent to have access to a consistent model of the surrounding environment. To have a map with one's location is about localization and mapping, which are correlated and dependent of one another. Previous works focus on incrementally building the map of the environment while at the same time locating the robot within it, which is referred to as SLAM. Moreover, different strategies can be applied to guide the robot to explore the unknown environment, which are gaining increasing attention [1]. The rapid advances in deep learning and more RL friendly simulators have stimulated the deep reinforcement learning (DRL) research interest in active SLAM. Several works [2], [3] also demonstrate the success of introducing DRL into embodied AI tasks.

However, the current DRL for active SLAM has their focus on model-free algorithms, which is lack of planning abilities and is less efficient compared to model-based ones. And the recent progress of MuZero [4] has tackled the difficulties that other model-based algorithms have when solving Atari Games, to outperform the state-of-the-art result from model-free algorithm and establish a new one. Despite the growing interest in the mcts-based algorithms, implementing these algorithms and adapting them to specific tasks remains a challenging endeavor. For instance, there are plenty of reinforcement learning libraries such as stable-baselines [5], rllib [6], ElegantRL [7] and etc. However, none of these libraries support mcts-based algorithms, though MuZero could have already been similar as model-free algorithms in the pipeline. Meanwhile, the current most widely used MuZero implementation EfficientZero [8] suffer from extending the algorithm to customized loss design, customized neural networks design and extending the experiments to different gym environments with ease.

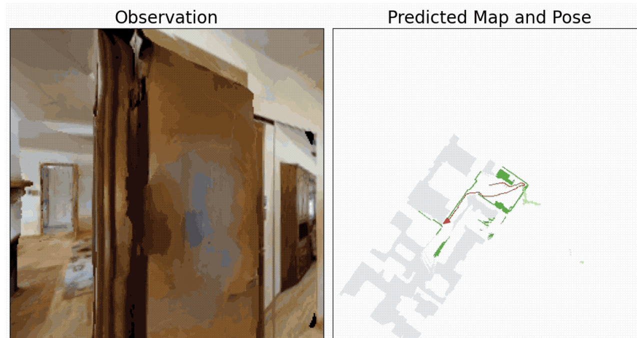

Fig. 1. Neural Tree Search on Gibson exploration task

To address these challenges, in this report, I present a modular and extendable library for mcts-based deep reinforcement learning, named muax. Muax aims to provide a user-friendly toolkit for deployment and testing of mcts-based algorithms. By focusing on modularity and extensibility, the library allows users to easily adapt various components of the reinforcement learning pipeline to suit their specific needs. To enable users to train and evaluate their models on a wide range of tasks with minimal effort, Muax is designed to be compatible with Gym environments, which is the mostly used RL environments. Further, a model-based method Neural Tree Search(NTS) has been proposed to solve active SLAM. The

model uses ResNet [9] as backbone and follows the search process as MuZero, while introduces a SLAM module, which decodes the predicted map from the hidden state for path planning. A corresponding loss function is proposed to train the NTS model.

II. Related Work

In recent years, the active SLAM has caught increasing attention [1]. This is due to the rapid advances in deep learning (DL), which creates new opportunities for using neural networks to solve navigation and active SLAM tasks [10] [2] [11]. Second, more reinforcement learning (RL) friendly simulators [12] [13] are published, which provide powerful tools for researchers to test their idea on different tasks with different baseline models. Further, the availability of open source benchmark datasets [14] [15] [16] from cameras (RGB images) and odometry sensors have further fueled the interest in this area.

The survey [1] has outlined the current state of the art in active SLAM and has it focus on recent advances in deep learning and reinforcement learning methods. Placed [1] also identified some of the key challenges and potentials corresponding to the deep learning methods. They have also discussed the limitations of existing approaches and identified areas for further research and development.

Chen et al. 2019 [2] and Chaplot et al. 2019 [11] both developed Deep Reinforcement Learning(DRL) methods for active SLAM based on visual inputs. Chen's approach uses imitation learning and is trained end-to-end, while Chaplot's approach uses hierarchical learning to achieve better performance. They both use PPO to train the model. Their experiments were conducted on the Habitat platform [13]. Furthermore, Chaplot's method was the winning entry of the CVPR 2019 Habitat PointGoal Navigation Challenge. Their experiments demonstrate the potential of using RL for active SLAM and show the advantages of using visual inputs in combination with deep learning algorithms.

The work of Chen and Chaplot show the possibility to develop a deep learning approach for active SLAM that is based on visual inputs. And more efficient learning or search method could strengthen the performance.

The model-based RL algorithm MuZero [4] has gained considerable attention due to its ability to learn strong policies without access to an explicit environment model, unlike its ancestors AlphaGo [17] and AlphaZero [18]. MuZero has also outperformed the state-of-the-art result from model-free algorithm and establish a new one. Further, MuZero has been evolving to tackle more complex action space and environment. Sampled MuZero [19] extends the original discrete action space to more complex action space, such as continuous action space. Stochastic MuZero [20] , on the other hand, improved the performance of MuZero on environment that is inherently stochastic. In summary, MuZero has shown its strong planning ability in not only board games, but also more general tasks.

III. Data

Habitat platform [13] provides support to do the simulation with various datasets. Habitat platform also provided several baseline tasks for the robot to solve. The data that are used throughout the project are Gibson [14]. The Gibson dataset provides large-scale, photorealistic environments for testing autonomous agents. It includes a variety of indoor and outdoor environments, making it a suitable dataset for evaluating models in a wide range of scenarios. The exploration task is performed, where the agent is required to navigate in an unfamiliar environment and the goal is to explore as much space as possible with the least time spent. The task datasets are available on the Habitat's repository, while the Gibson scene data require the permission from the official website.

The size of the datasets are summarized

To process the data and set up the environment, Furthermore, it takes around half an hour to set up the necessary environment each time on Colab, so the platform for developing the program is changed to an VM with 1 Nvidia T4 GPU. Another difficulty to handle is the licence for using the scene datasets. Different from the task datasets which can be downloaded directly from the Habitat simulator, the scene datasets Gibson and MatterPort3D all have to be downloaded from official website after the user signing agreement to get the licenses. Therefore, it takes some time to access the data. Once the datasets have been downloaded, it is required to follow the instruction to format the dataset as table I shows.

The licenses for referenced dataset:

Gibson: svl.stanford.edu/gibson2/assets/GDS_agreement.pdf

TABLE I
The size of the datasets

Dataset name	Size	Require permission
Gibson Scene	11GB	Yes
Gibson Point Goal Navigation task	385MB	No

IV. Method

The proposed Neural Tree Search is based on MuZero [4] , which uses Monte Carlo Tree Search(MCTS) as its search algorithm and directly learns a model for environment dynamic from interaction with the unknown environment during simulation. Further, it is a scalable model which was first used on board games and reached the same performance level as AlphaZero. At the same time, it is also capable to play Atari games, which is considered to be hard for AlphaZero and model-based algorithms, at the state-of-the-art level.

The model uses a loss function to minimize the error between predicted and actual bootstrapping value of a state, the action logits with searched action logits and the estimated reward and the actual reward. The canonical model does not explicitly includes the error from the estimated state and the actual state. Thus, the model is focus on planning, while learns a model that provides only information related to the planning. However, the procedure of MuZero fits well with the active SLAM problem settings. So I proposed a modified

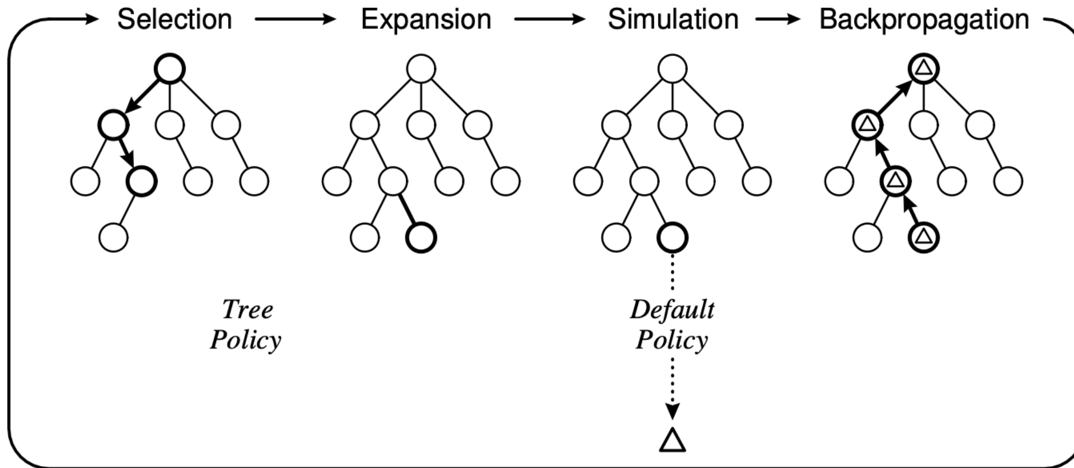

Fig. 2. One iteration of the general MCTS approach. [21]

MuZero algorithm, named Neural Tree Search to explicitly add "SLAM error" to solve the localization, mapping and planning problems.

A. Modeling: MCTS

Algorithm 1 Algorithm for MCTS

> **procedure** MCTSSEARCH(s_0)
>   create root node v_0 with state s_0
>   **while** within computational budget **do**
> $v_l \leftarrow TREEPOLICY(v_0)$
> $\Delta \leftarrow DEFAULTPOLICY(s(v_l))$
> $BACKUP(v_l, \Delta)$
>   **end while**
>   **return** $a(BESTCHILD(v_0))$
> **end procedure**

Monte Carlo Tree Search (MCTS) is a search algorithm used in decision-making problems, particularly in game-playing scenarios. It is based on the principle of Monte Carlo simulations, where possible actions are randomly sampled and played out multiple times to determine the best one.

MCTS operates by building a tree of possible actions and their outcomes. It starts with a single root node representing the current game state and expands the tree by adding child nodes that represent possible actions from the current state. The algorithm then simulates a number of playouts from each child node by randomly selecting actions until the end of the game is reached, resulting in a score or reward for each playout. Once the playouts are completed, the algorithm updates the scores of the nodes in the tree, backpropagating the results from the leaves to the root node. The update rule is typically based on the average or maximum score of the playouts, and may include exploration bonuses to encourage the algorithm to try new actions.

The process of selection, expansion, simulation, and back-propagation is repeated for a fixed number of iterations or until a time limit is reached. After the search is complete, the algorithm chooses the action that leads to the most promising child node, based on the scores and visit counts stored in the tree.

MCTS has been used successfully in a variety of game-playing scenarios, including Go, Chess, and Poker, where it has achieved state-of-the-art performance. It is also used in other decision-making problems, such as robotics and scheduling. [21]

B. Modeling: MuZero Search[1]

The search algorithm used by MuZero is based upon Monte-Carlo tree search with upper confidence bounds(UCB), an approach to planning that converges asymptotically to the optimal policy in single agent domains [4].

Every node of the search tree is associated with an internal state s. For each action a from s there is an edge (s, a) that stores a set of statistics $\{N(s,a), Q(s,a), P(s,a), R(s,a), S(s,a)\}$, respectively representing visit counts N, mean value Q, policy P, reward R, and state transition S. The search is divided into three stages, repeated for a number of simulations.

Selection: Each simulation starts from the internal root state s^0, and finishes when the simulation reaches a leaf node s^l. For each hypothetical timestep $k = 1...l$ of the simulation, an action a^k is selected according to the stored statistics for internal state s^{k-1}, by maximizing over an upper confidence bound,

$$a^k = \arg\max_a [Q(s,a) + P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} (c_1 + \log(\frac{\sum_b N(s,b) + c_2 + 1}{c_2})]$$

[1]This section is based on MuZero paper [4]'s Appendix B

Where c_1 and c_2 are used to control the influence of the prior $P(s,a)$ relative to the value $Q(s,a)$ as nodes are visited more often. In the experiments, $c_1 = 1.25$ and $c_2 = 19652$. For $k < l$, the next state and reward are looked up in the state transition and reward table $s^k = S(s^{k-1}, a^k)$, $r^k = R(s^{k-1}, a^k)$.

Expansion: At the final timestep l of the simulation, the reward and state are computed by the dynamics function, $r^l, s^l = g_\Theta(s^{l-1}, a^l)$, and stored in the corresponding tables, $R(s^{l-1}, a^l) = r^l$, $S(s^{l-1}, a^l) = s^l$. The policy and value are computed by the prediction function, $\mathbf{p}^l, v^l = f_\Theta(s^l)$. A new node, corresponding to state s^l is added to the search tree. Each edge (s^l, a) from the newly expanded node is initialized to $\{N(s^l, a) = 0, Q(s^l, a) = 0, P(s^l, a) = \mathbf{p}^l\}$. The search algorithm makes at most one call to the dynamics function and prediction function respectively per simulation.

Backup: At the end of the simulation, the statistics along the trajectory are updated. The backup is generalized to the case where the environment can emit intermediate rewards, have a discount γ. For $k = l...0$, the backup forms an $l-k$-step estimate of the cumulative discounted reward, bootstrapping from the value function v^l,

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

Where for $k = l...1$, updates the statistics for each edge (s^{k-1}, a^k) in the simulation path:

$$Q(s^{k-1}, a^k) := \frac{N(s^{k-1}, a^k)Q(s^{k-1}, a^k) + G^k}{N(s^{k-1}, a^k) + 1}$$

$$N(s^{k-1}, a^k) := N(s^{k-1}, a^k) + 1$$

C. Proposed model: Neural Tree Search

The proposed model is named Neural Tree Search(NTS), whose planning and training process is shown as Fig.5. NTS follows the procedure of MuZero, while includes a SLAM module for learning the environment. At each timestep t, NTS runs MCTS simulation and predicts the map m_t^k, the location l_t^k, the policy p_t^k, the value v_t^k for $k = 1, .., K$. NTS consists of four parts, representation, prediction, dynamics and SLAM. The representation module is called at the beginning to encode the raw observation into the hidden state, which are then used for MCTS. Inside the MCTS search tree, the dynamics of the environment is represented by the dynamics module, which takes the hidden state and the candidate action recurrently as $r^k, s^k = g(s^{k-1}, a^k)$, to simulate one hypothetical environment step given the action a^k, and estimated the reward r^k for the move. It mirrors the structure of an MDP model that computes the expected reward and state transition for a given state and action [23]. Yet in the MuZero, hidden state s^k has no semantics of the actual environment. The SLAM module decodes the hidden state to generate the predicted map and location as $m^k, l^k = q(s^k)$, which realizes the mapping from sensor output to the map of physical environment. The policy and value are computed from the internal state s^k by the prediction module, $p^k, v^k = f(s^k)$, which are then used for MCTS simulation. The MCTS outputs a recommended policy π_t and estimated value v_t. An action a_{t+1} π_t is then selected. All parameters of the four modules are trained jointly to match the predicted quantities from every hypothetical step k, to the observations for k actual timesteps.

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) + l^{map}(d_{t+k}, m_t^k) + l^{pose}(e_{t+k}, l_t^k) + c||\theta||^2$$

where $l^r, l^v, l^p, l^{map}, l^{pose}$ are loss functions for reward, value, policy, map, location, respectively.

1) Neural Network for NTS: Fig.3 and fig.4 display the neural networks used by the four modules of NTS. The representation module takes the RGB images as input, which have a resolution of 256x256, with 3 channels. The three channels are rescaled to the range [0,1]. Since the RGB observations have large spatial resolution, the representation module uses a sequence of convolutions with stride 2 to reduce the spatial resolution. The output is the hidden state of resolution 16x16, 64 channels.

For the prediction module, the policy head and value head preserved the spatial resolution while changes the number of channels, followed by linear layers to map the size of the output to the number of actions and support size, respectively.

The SLAM module takes the hidden state as the input and performs deconvolution to generate a decoded state that has the same resolution of the ground truth map, while at the same reduces the number of channels to 2, one represents the predicted map, the other one represents the explored area. The decoded state is then fed into pose estimator, which outputs the location by convolutions and fully connected layers.

The dynamics module first encodes the action into a single panel of the same resolution as the hidden state(16x16) and rescaled to the range [0,1]. The panel is then appended to the hidden state, and passes through 8 Residual Blocks to generate the next hidden state of the same shape.

To improve the stability of the process, the hidden state is scaled to the range [0, 1] at each step. The loss is multiplied by $\frac{1}{K}$, to ensure that the unroll step size has no effect on the magnitude of total gradient. And the gradient scaled down by $\frac{1}{2}$ at the start of the dynamic function to ensure that the total gradient applied to dynamic module remain constant.

The kernel size is 3x3 for all convolution operations.

V. System

To support the research, a comprehensive system is proposed that includes the following components, Data Source, Simulation, ML Pipeline and Visualization. An overview of the Neural Tree Search system is visualized by Fig.6.

The datasets I will use is the Gibson scene and task dataset, which provide a rich and diverse range of room environments for evaluating the approach.

The simulation environment is based on the Habitat platform, which provides a realistic and controlled environment

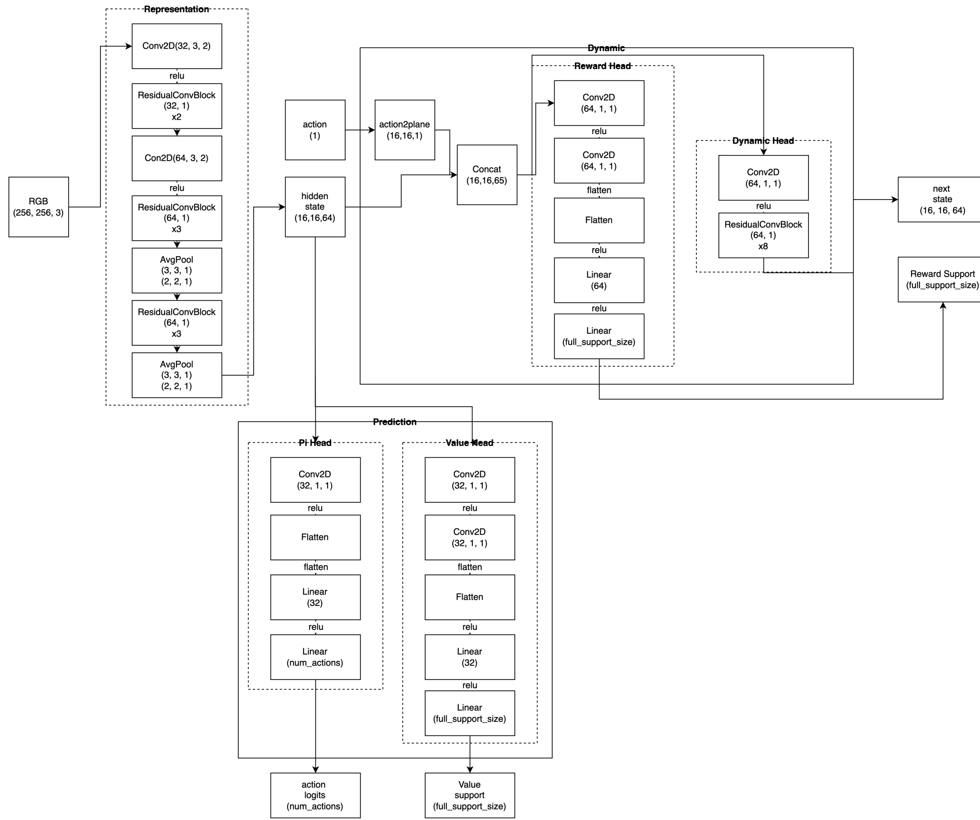

Fig. 3. The neural networks employed in NTS for planning.

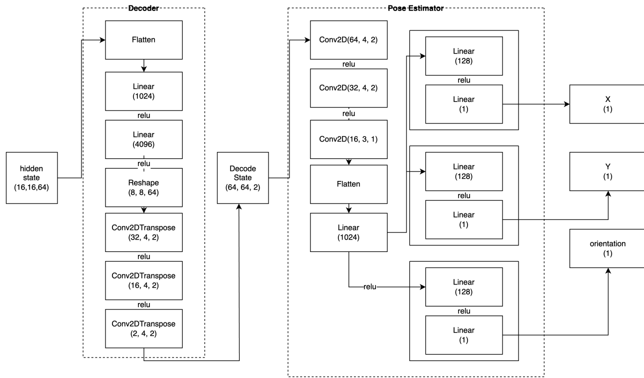

Fig. 4. The neural networks employed in NTS for SLAM.

for testing and evaluating agents and allows researchers to conduct experiments in a repeatable and reliable manner.

As for modeling, I will use Jax framework and the proposed Muax library.

Front-end tools will be used to analyze the results and to present the main findings in a clear and accessible manner.

In summary, the system provides a comprehensive and end-to-end solution for training and evaluating DRL approaches for active SLAM.

A. *Muax*[2]

Muax is implemented as separate to Neural Tree Search for the reason that the lack of flexible tools and the inherent complexity of mcts-based algorithms often impede researchers' and practitioners' ability to customize, extend, and deploy these methods in new domains. Therefore, muax is presented to provide a user-friendly toolkit for deployment and testing of mcts-based algorithms. In the following subsection, I will describe the design of Muax, highlight its key features and show how users can use and extend Muax for neural tree search research. I present several extensible examples that showcase the versatility of the framework in addressing different reinforcement learning tasks, from classic control problems to more complex, high-dimensional environments. I also created several example notebooks and tutorials in this open source project to make Muax functionalities clear for beginners and showcase how Muax can be extended and customized.

Muax is built around the principles of modularity, extensibility, and usability. To facilitate these principles, Muax decomposes the components in the MuZero algorithm and implements entire reinforcement learning pipeline. Fig.7 is a high-level overview of Muax system. There are 10 compo-

[2]Muax is available at **https://github.com/bwfbowen/muax**

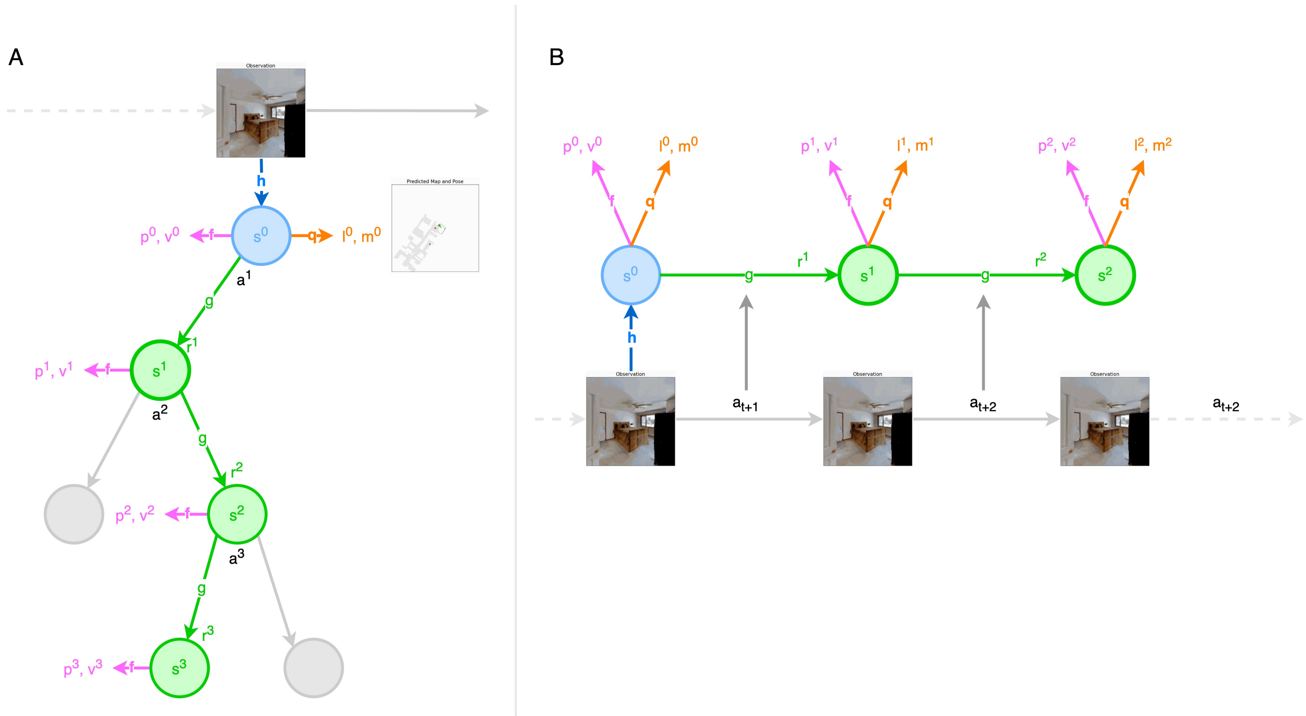

Fig. 5. An overview of NTS planning and training process. (A) shows how NTS uses its model to plan. The model consists of four connected components for representation, dynamics, prediction and SLAM. The initial hidden state s^0 is the output of representation module h, whose input is the raw observations, for instance, an RGB image of the room to be explored. For each hidden state s^k, the policy p^k and value v^k are predicted by the prediction module f and the predicted map m^k and predicted location l^k(in the predicted map) are gathered from SLAM module. Then, Given a hidden state s^{k-1} and a candidate action a^k, the dynamics module g produces an immediate reward r^k and a new hidden state s^k. (B) shows how NTS is trained. For one sub trajectory sampled from replay buffer, the representation module h generates the initial hidden state s^0 from the past observations o_t from the first timestep of the sampled trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, five pairs of quantities are predicted to calculate the loss, namely the predicted policy p^k with the action probability from the root node π_{t+k}, the preidcted value v^k with the n-step bootstrapping value z_{t+k}, the predicted reward r^k with the actual reward received u_{t+k}, the predicted map m^k with the ground truth map d_{t+k}, the predicted location l^k with the actual location e_{t+k}. Then the dynamics module g receives as input the hidden state s^{k-1} from the previous step and the real action a_{t+k}, to generate the next hidden state s^k. Four modules are jointly trained.

nents, which can be divided into 3 groups, namely model, training/testing and environment.

Model: Muax decomposes the MuZero algorithm into tree search policy, neural network and optimizer.

a. **Model class**: This class serves as the primary interface as reinforcement learning agent does, which interacts with the environment. It is responsible for managing the model's representation, prediction, and dynamics functions and their parameters, as well as the optimizer and the optimization process. Model class also controls which tree search policy to use. Users can customize the model by providing their own implementation of each functions or using the default ones provided by the framework.

b. **Tree search policy**: Muax uses mctx [22] as tree search policy. Through model class interface, the user specifies which policy to use. The tree search by mctx is fully compiled just-in-time and runs in parallel, making the process very efficient.

c. **Neural network**: Muax uses haiku to build neural networks. The networks are used for Representation, Prediction, and Dynamic classes. These classes are used to build the major functions for tree search policies. As the name suggested, Representation is for encoding the raw observation into hidden state, Prediction is for evaluating the state and generates the prior action logits and Dynamic is for transferring the hidden state into the next state and calculates the reward associated with the transfer. By separating these functions, Muax promotes modularity and simplifies the process of customizing the model.

d. **Optimizer**: Muax uses optax to build optimizer. The loss function can be provided through Model class interface. Therefore, Muax makes it easy to design and experiment with customized loss.

Training/Testing: Muax employs a flexible training and testing loop. The main training loop is implemented in the fit function, which takes care of environment interaction, model training, and performance monitoring. Users can customize the training loop by providing their own implementation. Meanwhile, the testing loop is a simple interaction with the environment with trained model. The training and testing loop are both close to the typical loop for model-free algorithms, which makes it easy for RL practitioners to get started.

a. **Episoder Tracer**: This module is responsible for tem-

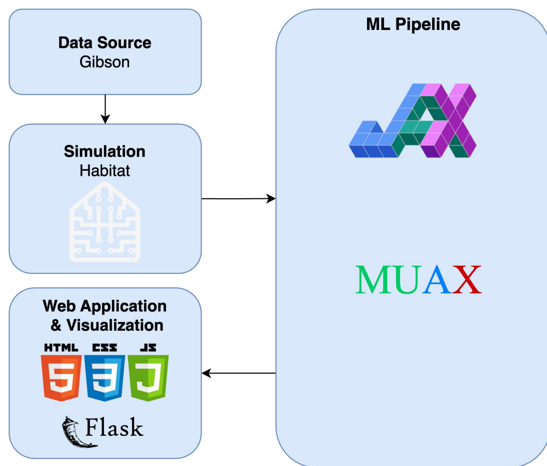

Fig. 6. An overview of Neural Tree Search system structure

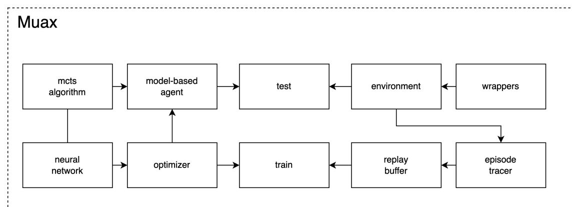

Fig. 7. Overview of Muax architecture

porarily handling step-wise data collected through interaction with the environment. The NStep and PNStep classes calculate the n-step bootstrapping and prioritized weight, respectively. Users can customize the behavior of these components to suit their specific requirements.

b. **Trajectory Replay Buffer**: This module is responsible for handling trajectories' data. The ReplayBuffer class stores the collected data and provides the interface for sampling data.

Environment: Muax is designed to be compatible with Gym environments, allowing users to train and evaluate their models on a wide range of tasks. The framework provides utility functions for easily wrapping Gym environments, enabling seamless integration with the training loop. Additionally, Muax will support the use of vectorized environments for more efficient training and evaluation.

Any implementations that are compliant with the interfaces defined by Muax can be seamlessly integrated.

1) Extending Muax: Muax is designed to be flexible and extensible, allowing users to adapt the framework to various reinforcement learning tasks and requirements easily. With Muax it is possible to explore more effective neural networks, test novel loss functions, and experiment with customized environments. The customized modules can be integrated into Muax seamlessly if the interfaces are properly implemented.

Customize Model Components: Users can implement customized components, for instance, representation, prediction, and dynamic functions by extending the corresponding base classes from the muax.nn module and providing their own implementation for the `__call__` method. This enables users to create custom models with different architectures.

```
from muax.nn import (Representation,
                     Prediction,
                     Dynamic)

class CustomRepr(Representation):
    def __call__(self, x):
        # Implement custom method
        return out

class CustomPred(Prediction):
    def __call__(self, x):
        # Implement custom method
        return out

class CustomDynamic(Dynamic):
    def __call__(self, x, a):
        # Implement custom method
        return out
```

Customize Loss Function: The loss function plays an important role in algorithm training. Some work [8] obtains better performance through loss function designs. Suppose we want to use a custom loss function for training the model. We can achieve this by defining a new loss function and passing it to the MuZero class as an argument.

```
def custom_loss_fn(model,
                   batch,
                   params):
    # Compute the custom loss
    # based on the model, batch,
    # and parameters
    # ...
    return custom_loss

model = muax.MuZero(
                ...,
                loss_fn=custom_loss_fn
                )
```

Customize Environments and Wrappers: To use custom environments or apply additional functionality to existing environments, users can create their own environment classes by extending the `gym.Env` class and implementing the required methods `step()` and `reset()`. Additionally, users can also create custom wrappers to modify the behavior of existing environments.

```
import gymnasium as gym

class CustomEnvironment(gym.Env):
    def __init__(self):
        # Initialize the custom
```

```
        # environment
        pass

    def step(self, action):
        # Implement the custom
        # step logic
        return obs, reward, done, info

    def reset(self):
        # Implement the custom
        # reset logic
        return init_obs, info
```

These extension mechanisms empower users to easily adapt Muax to address various reinforcement learning tasks and requirements, facilitating the development of customized algorithms and techniques.

VI. Experiment

In this section, I performed comparison experiments on Gibson exploration task against Neural SLAM [3]. Therefore, I used the same task setup. The reward is defined as the exploration ratio of the entire room. The observation is the RGB images from the visual sensor, and the pose from the motion sensor. However, Neural Tree Search is an end-to-end model and is different from Neural SLAM. So there are differences in particular settings.

A. Experiment Settings

The action for NTS is defined as the location on the predicted map. By the use of analytical path planner(Fast Marching Method [24]), which takes pose, map and goal as input to calculate the robot action sequence, it is able to guide the model in the actual environment. Therefore, the robot maintains a predicted map, and by continuously specifying a position in this map, with the analytical path planner, the model guides the robot to explore the environment. The key to successfully explore the unknown environment is whether the predicted map and location matches the ground truth. Otherwise, the robot action sequence calculated from FMM would lead the robot to undesired place. The parameters for the Gibson scene and robot is the same as Neural SLAM, while two different action settings are tested. The 4x4 action represents equally dividing the map into 16 grids, and each action is mapped to the center of each grid. 10x10 results in 100 grids, which enable the robot to have more directions to choose. More parameters are listed in table.II

B. Results

The result is presented as in III and 8. NTS 10 realizes competitive performance to Neural SLAM with 30 episodes, the latter requires over 70 episodes and vectorized environment, which demonstrates the efficiency of NTS and the gain from planning. However, NTS 4 is not comparable to NTS 10 and Neural SLAM, and has higher variance. One possible reason is that the robot is blocked by some obstacle and unable to make small adjustment. For example, entering a door.

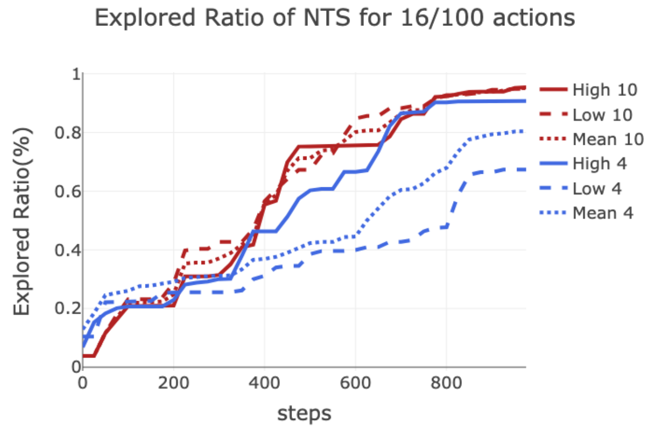

Fig. 8. Comparison of NTS with different action size.

TABLE II
THE PARAMETERS FOR EXPERIMENTS

Parameters	**Value**
Max Episode Length	1000
Number of episodes	30
Environment Frame Width	256
Environment Frame Height	256
Camera Height	1.25
Frame Height	128
Frame Width	128
Vision Range	64
Number of Local Steps	25
Number of Global Steps	2
Number of Simulations	30
Number of Trajectories	4
Sample per Trajectory	16
K steps	5
N bootstrapping	10
Action Width	4, 10
Action Height	4, 10

Further, pose estimation error accumulates in the long run, when the robot is happened to be blocked for a long time, the predicted map could be far from correct. Therefore, localization needs to be improved.

TABLE III
THE PERFORMANCE COMPARISON

Method	**Gibson Val Cov.**
Neural Tree Search 4	0.804
Active Neural SLAM	0.948
Neural Tree Search 10	**0.952**

VII. Conclusion

In this project, I proposed an open source mcts-based reinforcement learning library and proposed a model-based algorithm for solving active SLAM tasks. I have also conducted experiments on Habitat simulation platform with Gibson dataset.

As for the mcts-based library muax, I have implemented ten connected modules, which enable the general pipeline for RL. Three different backbones, from MLP, ResNet to EfficientZero architecture are also implemented, with two examples for demonstrating the usage of muax being included. Moreover, the key functions are JIT-able, which further improves the efficiency of the tree search process.

As for the active SLAM, I have proposed the model Nerual Tree Search for exploring the unknown environment with planning. By introducing SLAM module, the robot is able to learn not only the dynamics of the environment, but also builds the map from exploration. Competitive performance is obtained compared to previous works with less training time.

REFERENCES

[1] Placed, Julio A., et al. "A survey on active simultaneous localization and mapping: State of the art and new frontiers." IEEE Transactions on Robotics (2023).

[2] Chen, Tao, Saurabh Gupta, and Abhinav Gupta. "Learning exploration policies for navigation." arXiv preprint arXiv:1903.01959 (2019).

[3] Chaplot, Devendra Singh, et al. "Learning to explore using active neural slam." arXiv preprint arXiv:2004.05155 (2020).

[4] Schrittwieser, Julian, et al. "Mastering atari, go, chess and shogi by planning with a learned model." Nature 588.7839 (2020): 604-609.

[5] Raffin, Antonin, et al. "Stable-baselines3: Reliable reinforcement learning implementations." The Journal of Machine Learning Research 22.1 (2021): 12348-12355.

[6] Liang, Eric, et al. "RLlib: Abstractions for distributed reinforcement learning." International Conference on Machine Learning. PMLR, 2018.

[7] Liu, Xiao-Yang, et al. "ElegantRL-Podracer: Scalable and elastic library for cloud-native deep reinforcement learning." arXiv preprint arXiv:2112.05923 (2021).

[8] Ye, Weirui, et al. "Mastering atari games with limited data." Advances in Neural Information Processing Systems 34 (2021): 25476-25488.

[9] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[10] Mirowski, Piotr, et al. "Learning to navigate in complex environments." arXiv preprint arXiv:1611.03673 (2016).

[11] Chaplot, Devendra Singh, et al. "Neural topological slam for visual navigation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020.

[12] Beattie, Charles, et al. "Deepmind lab." arXiv preprint arXiv:1612.03801 (2016).

[13] Savva, Manolis, et al. "Habitat: A platform for embodied ai research." Proceedings of the IEEE/CVF international conference on computer vision. 2019.

[14] Xia, Fei, et al. "Gibson env: Real-world perception for embodied agents." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

[15] Chang, Angel, et al. "Matterport3d: Learning from rgb-d data in indoor environments." arXiv preprint arXiv:1709.06158 (2017).

[16] Ramakrishnan, Santhosh K., et al. "Habitat-matterport 3d dataset (hm3d): 1000 large-scale 3d environments for embodied ai." arXiv preprint arXiv:2109.08238 (2021).

[17] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." nature 529.7587 (2016): 484-489.

[18] Silver, David, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." arXiv preprint arXiv:1712.01815 (2017).

[19] Hubert, Thomas, et al. "Learning and planning in complex action spaces." International Conference on Machine Learning. PMLR, 2021.

[20] Antonoglou, Ioannis, et al. "Planning in stochastic environments with a learned model." International Conference on Learning Representations. 2021.

[21] Browne, Cameron B., et al. "A survey of monte carlo tree search methods." IEEE Transactions on Computational Intelligence and AI in games 4.1 (2012): 1-43.

[22] Danihelka, Ivo, et al. "Policy improvement by planning with Gumbel." International Conference on Learning Representations. 2022.

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

[24] Sethian, James A. "A fast marching level set method for monotonically advancing fronts." Proceedings of the National Academy of Sciences 93.4 (1996): 1591-1595.