Should You Live in Fear of the 9? A Monte-Carlo Tree Search Reinforcement Learning Approach to Playing NMBR 9

Emmett Springer and Luis Brena and Rohit Shankar
Johns Hopkins University
{espring6, lbrenap1, rshanka6}@jhu.edu

Abstract

In this paper, we apply reinforcement learning methods to the popular board game NMBR 9. Specifically, we implemented two Monte-Carlo Tree Search(MCTS) variations with the hope of achieving human level performance. The first implementation of MCTS exploits local node/action similarities to decrease the state space of the tree. The second implementation uses an actor critic model and a MCTS to substitute for the lack of expert move data. In order to implement and evaluate these methods, we developed a simulation environment in PyGame. Some features of the simulation environment include human playability as well as computer simulated games.

1 Introduction

The real-world applications of Reinforcement Learning (RL) extend to various fields, including energy management and resource planning. These areas contain similar challenges to those found in board games. One of the most common issues found in both board games and real-world application is the management of a massive state and action spaces. More broadly, the challenge of balancing exploration and exploitation is also encountered in both domains (Dulac-Arnold et al., 2019).

The goal of our paper is to present two Monte-Carlo Tree Search based algorithms to effectively play the game NMBR 9. NMBR 9 is a strategy board game in which players place number tiles of varying shapes to create a stacked board. Points in NMBR 9 are scored such that placing numbers on higher levels multiplies their point value. This scoring mechanism well-embodies the challenge of trade-off between immediate rewards and delayed rewards. Furthermore, while the (immediate) points rely deterministically on the player's actions, there is immense complexity in the game by there mere size of the possible states. The deck, which consists of two copies each of ten cards (the numbers 0 through 9) can be shuffled in $\frac{20!}{10!10!} = 184,756,720$ unique ways. From these unique sequences of numbers there are numerous ways to arrange the tiles according to the game's rules such that enumerating them becomes impractical.

2 Related Work

The famous MCTS algorithms used in AlphaGo and AlphaZero (Silver et al., 2017) are the primary inspiration for our project. AlphaGo uses deep learning architectures and Monte Carlo Tree Search to achieve greater than human abilities at playing GO. A policy network structured as a convolutional neural network is trained on 30 million expert moves. A value network is also trained as a convolutional neural network to conduct board evaluation(+1 or -1) for win or loss. Similarly, Alpha Zero uses a combination of deep learning and Monte Carlo Tree search to achieve super-human abilities at chess. The policy and value networks are trained as a double head convolutional neural network. One head outputs the probability of each move and the value head indicates a evaluation number indicating outcome(+1 to -1) for winning or losing. The primary strategy in both AlphaGo and AlphaZero is the use of convolutional neural networks to estimate the massive state space. Due to computational restriction and lack of expert move data, we had to modify the method presented in AlphaGo and AlphaZero. Ultimately, we came up with two MCTS variation which used node similarity and an actor critic model to effectively play NMBR 9.

3 Problem Formulation

We formulate the game as a finite-horizon, stationary MDP based on the rules of the game. The most notable rules are that a tile must always touch another tile and tiles on higher levels must be supported by two tiles. In order to make the game more computationally friendly, we made two main assumptions. Firstly, we assume that the game is played "single player" with the goal of maximizing points scored. Secondly, we assume the board size to be 24 by 24 with a maximum of 4 layers.

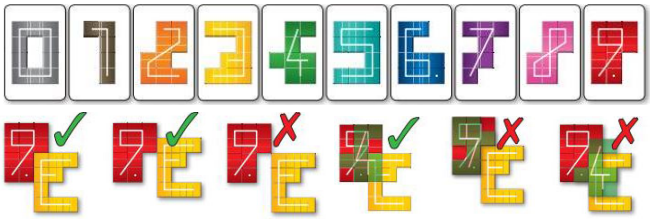

Figure 1: Tile shapes and placement rules for the NMBR 9 game.

State Space. The state space for this problem is all possible board configurations across all time points in the game. We represent a given state with the following tuple

below. Where B_t and T_t are representations of the board where each element is the value and the unique identifier, respectively. -1 indicates invalid move. The element n_t is the number to be placed on turn t, and d_t is a vector of the number of each type of card remaining in the deck at t.

$$
\begin{aligned}
&s_t = (B_t, T_t, n_t, d_t),\\
&B_t \in \{-1, 0, 1, \ldots, 9\}^{25\times25\times5},\\
&T_t \in \{0, 1, \ldots, 20\}^{25\times25\times5},\\
&n_t \in \{0, 1, \ldots, 9\},\\
&d_t \in \{0, 1, 2\}^{10}
\end{aligned}
$$

Action Space. The total action space includes all possible placements of any tile on the board, with two-dimensional position (x, y), level z, and orientation θ where 0,1,2,3 represent $0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}$.

$$
\begin{aligned}
\mathbf{A} = \{(x, y, z, \theta) \mid\ & x, y \in \{0, 1, \ldots, 24\},\\
& z \in \{0, 1, \ldots, 4\},\\
& \theta \in \{0, 1, 2, 3\}\}
\end{aligned}
$$

Rewards. Rewards are deterministic and incurred immediately after tile placement such that $r(s, a) = s_n a_z$(number multiplied by layer). The goal is to maximize the number of points scored in 20 moves.

State Transitions. The only stochasticity in the game is the order in which the number tiles are pulled. Therefore, the state transitions are deterministic in the component B_t, and stochastically independent from the action taken in n_t and d_t:

$$
\mathbb{E}[n_{t+1}] = \frac{d_{tn_t}}{\sum_{i=0}^{9} d_{ti}}
$$

$$
d_{t+1} = \begin{cases} d_{ti} - 1 & \text{for } i = n_{t+1} \\ d_{ti} & \text{otherwise} \end{cases}
$$

4 Game implementation

We used the Pygame library to create a graphical implementation of the board, which aims to visualize machine moves and record trajectories that will later be utilized in our algorithms. The user interface features a trajectory tracker that saves states S, actions A, and an estimated value $v(s)$. These trajectories are stored and used by our proposed methods. Additionally, the user interface can simulate random actions and load a trajectory to identify patterns that could aid in visualizing and empirically refining procedures for better results. In Figure 3, we illustrate one of the trajectories generated using this module.

5 Method

Games such as such as chess, Go, and NMBR 9 present problems with very large state spaces that are difficult to optimize manually. These problems are effectively learned by algorithms consisting of MCTS and deep learning. We first present the usage of MCTS with a node similarity mechanism. After which, we present safety actor-critic MCTS(SAM).

Monte-Carlo Tree Search (MCTS) Algorithm. MCTS is a tree based sampling algorithm where nodes represent states. The MCTS algorithm has four major steps: 1. Selection 2.Expansion. 3. Simulation and 4. Backpropagation. MCTS commonly employs a neural network f to guide its simulations (Sutton and Barto, 2018). It stores $P(s, a)$, $N(s, a)$ and $Q(s, a)$ for each edge (s, a) in a tree. Starting from the root, the algorithm selects moves that maximize the expression $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$. This process continues until a leaf node is reached. Once at the leaf node, the position s is expanded and evaluated using $f(s)$. The output of MCTS consists of moving probabilities that are proportional to $N(sa)^{\frac{1}{\tau}}$, where τ is a temperature parameter. MCTS utilizes self-play to train the neural network f through gradient descent.

Heuristic Local Node/Action Similarity. The first MCTS method we explore exploits the structure of NMBR 9 to iteratively estimate action values starting with actions at the end of the game. This approach is motivated by two key observations. Firstly, the action values become more certain as turns progress and the number of unique tiles left in the deck decreases. The accumulation of rewards throughout the game means that it is always best to be greedy on the final turn, as this is the last opportunity to get rewards. The transition from the second-to-last (19th) turn to the final turn is deterministic (since there is only one card left in the deck), so the action value can be calculated exactly. The transition from the 18th turn to the 19th turn is stochastic, but at most has only two possible next tiles to draw, and the degree of this stochasticity progressively increases for each layer up the tree. The second key observation is that objectively more strategic placement in early turns (building up a good base) are predicated on taking good actions later in the game (scoring high tiles on that base). Thus, until the player learns to take good actions later in the game, the empirical action values of early actions are not very meaningful.

Together, these two observations suggest an advantage to training the NMBR 9 Monte-Carlo tree search extensively on nodes at the bottom of the tree and progressively propagating this learning up the tree. To accomplish this, we developed a custom heuristic for estimating any given valid action value from a local abstraction of the board space around the tile to be placed, per Figure 2. A local $x \times y \times 5$ section of the board matrix B_t is translated into a simplified $x \times y$ matrix, where $(x, y) = (7, 8)$ for $\theta = 0$ or 2 and $(x, y) = (8, 7)$ for $\theta = 1$ or 3. In the L

matrix, only the current highest level of a spot on the grid is recorded. The goal of this simplification is to greatly reduces the number of possible iterations of the *local level space* L, increasing the sample size of the number of times an action matching L is taken, and thus improving the trustworthiness of the empirical action value associated with L.

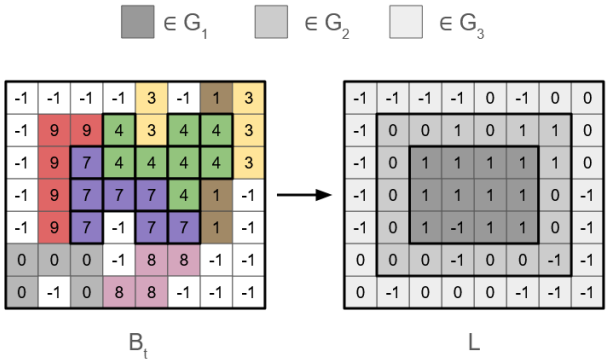

Figure 2: Example of local abstracted state space L for the placement of a 7 tile with $z = 1$ and $\theta = 1$.

Let L^* indicate the local level space of an action placing tile n with level z and rotation θ. Then, for any pre-trained local level space L obtained from placing the same tile with the same z and θ, we can calculate a similarity score as follows:

$$\textit{similiarity score} =$$

$$0.5 \sum_{\{i,j\} \in G_1} \frac{\gamma_{ij}}{12} + 0.4 \sum_{\{i,j\} \in G_2} \frac{\gamma_{ij}}{18} + 0.1 \sum_{\{i,j\} \in G_3} \frac{\gamma_{ij}}{26}$$

$$\gamma_{ij} = \begin{cases} 1 & L_{ij} = L^*_{ij} \\ 0.5 & L_{ij} \neq L^*_{ij}, \quad L_{ij}, L^*_{ij} > -1 \\ 0 & \text{otherwise} \end{cases}$$

The similarity score ranges from 0-1 and equals 1 if L and L^* are identical. Grid spaces closest to the placed piece are weighed most heavily, and matching the specific level of placement is rewarded more than having the space occupied, but at a different level. If a grid space is occupied (at any level) in one L but is not occupied at all in the other, nothing is added to the similarity score.

Using the similarity score heuristic, we develop an iterative MCTS training process in which simulation/rollout policies differ between game turns/tree layers. We utilize three different policies for action selection at a given node:

- **Upper Confidence Bound (UCB)** uses a variation of UCB to select actions rather than children nodes due to transition stochasticity.

$$A_t = \arg\max_a (Q_t(a) + C\sqrt{\frac{\ln t}{N_t(a)}})$$

where t is the total number of visits the node, $N_t(a)$ is the number of times action a has been taken, and $Q_t(a)$ is the empirical action value.

- **ϵ-Greedy with Empirical Action Value (empQ)** utilizes a straight-forward ϵ-greedy policy where the empirical action values $Q_t(a)$ are initialized to be equal to the immediate reward of the action and are updated during MCTS back-propagation.

- **Local Action Level Similarity (LALS)** is an ϵ-greedy policy which utilizes the empirical Q values of the local level spaces, L, instead of $Q_t(a)$. At the current node, every valid action is converted to its corresponding local level space L^*, matched with the stored L with the highest similarity score, and an action is chosen ϵ-greedily from the stored empirical action value for the matched L.

Figure 3 illustrates the iterative training process implemented. We begin by applying the UCB action-selection policy on the target node at the top of the tree, and empQ for the empirical action value at all other tree layers. The ϵ value at the 20th layer is set to 1, and varies between 0.4 and 0.8 for all other layers.

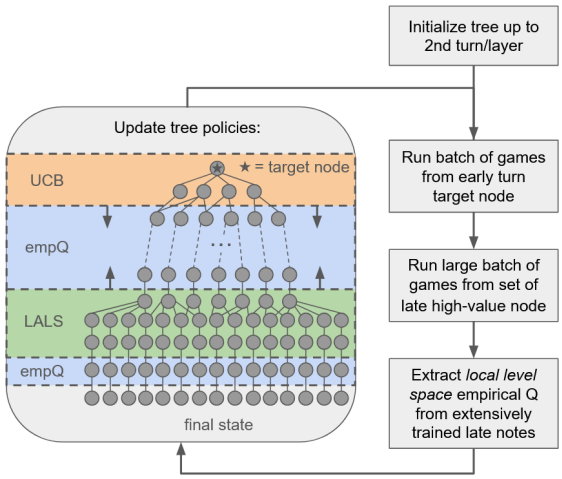

Figure 3: Your caption

We then simulate a batch of games via the set simulation policies starting from an early turn target node (which can be the root node, but can also be within the first few turns). During these simulations, the $Q_t(a)$ values are updating online. Then, we run a much larger number of simulations with the UCB policy from a set of late turn nodes with high final scores. We then add to the stored set of local level state action values by averaging the action value across all instances of each L appearing from the previous batch. Between training batches we update the

tree policies to include more LALS further up the tree and more UCB further down the tree, replacing the initial empQ policy.

Safety Actor-critic MCTS (SAM) SAM is a modified version of the actor-critic algorithm. It incorporates a penalization term $Q^c(s,a)$ for invalid actions and a demonstrator that adds $Q^d(S_{t+1})$ based on the next state generated by the action A_t. The first term penalizes invalid actions, while the second term guides the actor-critic toward trajectories that yield better results.

We implemented several variations of the actor-critic algorithm: a vanilla actor-critic using a linear network, another using convolutional neural networks, a safety critic that incorporates only $Q^c(s,a)$, and finally, SAM, which includes the term $Q^d(S_{t+1})$. Previous proposals for large state spaces with small action spaces have utilized demonstrators (Liu et al., 2021) or guided the process with MCTS (Lu et al., 2021) to help the algorithm escape local minima. Demonstrator approaches have been effective in training policy and value networks based on expert moves before testing; however, we did not have access to such data because the game had not been implemented previously. Therefore, we found it appropriate to use MCTS as a guiding mechanism for the actor-critic.

Additionally, our problem includes constraints on possible actions, represented as $g(v_t, a_t) = \mathcal{A}_{t+1}$. This complexity cannot be addressed simply by modeling the network to follow specific guidelines and making invalid policy estimations. For this reason, we opted for a safety-critic approach that penalizes invalid actions, resulting in lower rewards.

Our actor-critic method is based on the TD(0) actor-critic principle. We have a policy with a baseline such that the policy gradient theorem can be described as:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \nabla \pi(a|s,\theta)(q_\pi(s,a) - v(s))$$

Next, we calculate a one-step update for the weights of the policy estimation function:

$$\theta_{t+1} = \theta_t + \alpha(G_{t:t+1} - \hat{v}(S_t, \mathbf{w}))\frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

Substituting in the return, we have:

$$= \theta_t + \alpha\left(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\right) \times \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \quad (1)$$

This simplifies to:

$$= \theta_t + \alpha\delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

The estimation of the value function is given by:

$$\mathbf{w} = \mathbf{w} + \alpha^{\mathbf{w}}\delta_t \nabla \hat{v}(S,\mathbf{w})$$

The environment generates rewards based on a reward function $r(s,a)$. We argue that the reward obtained for choosing an invalid action a_1 should be nullified. The algorithm must learn not only a policy but also which actions are invalid. The standard approach is to set the factor $Q^c(s,a_1) = 0$, indicating that a_1 violates the constraint, and $Q^c(s,a) = 1$ for valid actions, incentivizing the model to select them (Ma et al., 2021). Therefore, we have:

$$r'(s,a) = Q^c(s,a)r(s,a)$$

This ensures that the reward generated by invalid actions is nullified.

From equation (1), we can implement the safety factor:

$$\delta_t = (R_{t+1} \cdot Q^c(S_t,A_t) + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}))$$
$$\theta_{t+1} = \theta_t + \alpha\delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$$

In principle, this factor decreases the probability of taking an invalid action a_π.

We enhance the safety critic by incorporating a reward factor $Q^D(S_{t+1}^{[d]})$ to guide the actor-critic. Given a trajectory $d = \{(s_0,a_0),(s_1,a_1),\ldots,(s_{19},a_{19})\}$, where $S_{t+1}^{[d]}$ represents the state at timestep $t+1$ along trajectory d, $Q^D(S_{t+1}^{[d]})$ provides additional reward. This reward is calculated as $Q^D(S_{t+1}^{[d]}) = 23\phi$, where ϕ is the cosine similarity between the matrix representations of $S_{t+1}^{[d]}$ and the current state S_{t+1}:

$$\phi = \frac{S_{t+1}^{[d]} \cdot S_{t+1}}{||S_{t+1}^{[d]}|| \cdot ||S_{t+1}||}$$

The constant 23 represents the product of the mean individual value and the mean level (details omitted for brevity). This reward modifies the advantage function during training:

$$\delta_t = R_{t+1} + Q^D(S_{t+1}^{[d]}) + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$$

Note the $Q^c(S_t,A_t)$ term is generally considered part of R_{t+1} in advantage actor-critic formulations. Trajectories d can be generated from expert human play or high-scoring MCTS runs. The reward is conditional on action validity:

$$Q^D(S_{t+1}^{[d]}) = \begin{cases} 23\phi, & \text{if } A_t \text{ is valid} \\ -45, & \text{if } A_t \text{ is invalid} \end{cases}$$

We employed a factor of -45, as it represents the negative of the maximum achievable score at any given step. Our empirical results demonstrate that this term is effective. A detailed description of the full algorithm can be found in Apendix B.

6 Results

Results for the Node-Action similarity MCTS had a very high variation throughout training and did not improve significantly over time (Figure 6). While some games were able to achieve very high scores, the average remained below 20. Given that an amateur player can typically score 50-70 points per game, these results demonstrate significant challenges with the learning. There are many reasons for this. Firstly, the computational power required for this algorithm, despite all of the design choices to reduce computational burden, remains too high to implement effectively without millions of simulations per action. We were only able to train up to three batches, implementing LALS up to the 16th layer. Furthermore, the hyperparameters (i.e. C for UCB and ϵ) used for the training may have been suboptimal, so more work would need to be done to fine-tune these parameters.

For the SAM algorithm, although we generated trajectories, their quantity and quality were insufficient. We achieved satisfactory results by setting $Q^D = 20$ to fixed values when the action was valid.

The implementation involved testing the actor-critic across multiple scenarios to evaluate its ability to generalize and learn from state descriptions. We discovered that, given a specific state, the algorithm could estimate a policy that consistently produces better average results (as demonstrated in Figure 4). Furthermore, we made significant progress in learning up to the first seven valid states without explicitly providing that information to the actor-critic. By simply modifying the reward function, we observed substantial improvements within just 500 episodes (Figure 5).

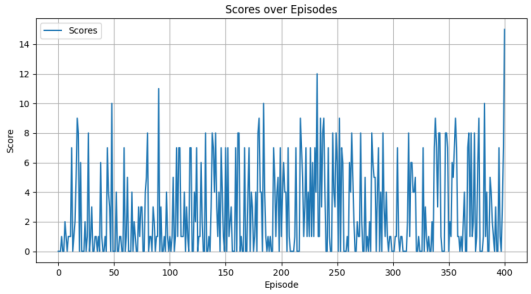

Figure 4: Results for SAM without penalization factors

However, we identified a limitation regarding the algorithm's ability to reach a more optimal state. This limitation is due to the large size of the action space. We can conclude that the action space is significantly smaller than the state space, i.e., $\mathcal{A} << \mathcal{S}$. That is why we couldn't achieve better results, but we believe the demonstrator trajectories will help the algorithm improve.

Figure 5: Results for SAM with constrain and trajectory factors

7 Conclusion

In this paper, we showed the Monte Carlo Tree Search method applied to the board game NMBR 9. In particular, we were successfully able to implement two variations of the MCTS method. The first method utilized node similarity and the second method used a modified actor-critic algorithm. In the future, we would like to implement a MCTS by using a convolutional neural network to function has a value state estimator. Due to time and computational constraints we were unable to finish training and integrating the convolutional neural network. This method would be more in line with AlphaGo and AlphaZero.

References

Gabriel Dulac-Arnold, Daniel Jaymin Mankowitz, and Todd Hester. 2019. Challenges of real-world reinforcement learning. *ArXiv*, abs/1904.12901.

Guoqing Liu, Li Zhao, Pushi Zhang, Jiang Bian, Tao Qin, Nenghai Yu, and Tie-Yan Liu. 2021. Demonstration actor critic. *Neurocomputing*, 434:194–202.

Qiang Lu, Fan Tao, Shuo Zhou, and Zhiguang Wang. 2021. Incorporating actor-critic in monte carlo tree search for symbolic regression. *Neural Computing and Applications*, 33(14):8495–8511.

Haitong Ma, Yang Guan, Shegnbo Eben Li, Xiangteng Zhang, Sifa Zheng, and Jianyu Chen. 2021. Feasible actor-critic: Constrained reinforcement learning for ensuring statewise safety. *ArXiv*, abs/2105.10682.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, L. Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of go without human knowledge. *Nature*, 550:354–359.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*, 2 edition. MIT press.

A Networks architecture

The two network architectures designed for the SAM algorithm both incorporate a single convolutional layer,

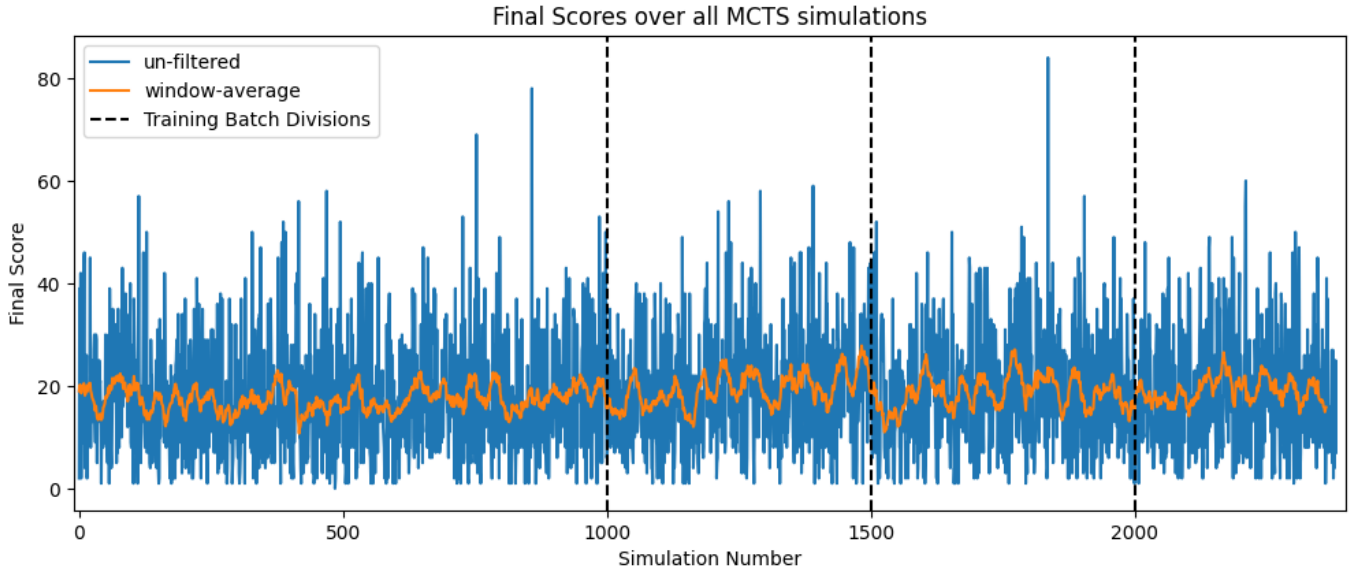

Figure 6

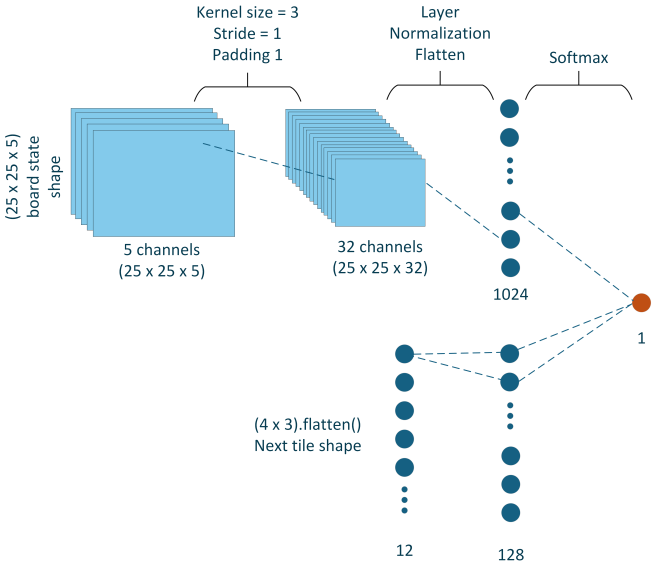

Figure 7: Value Network Architecture

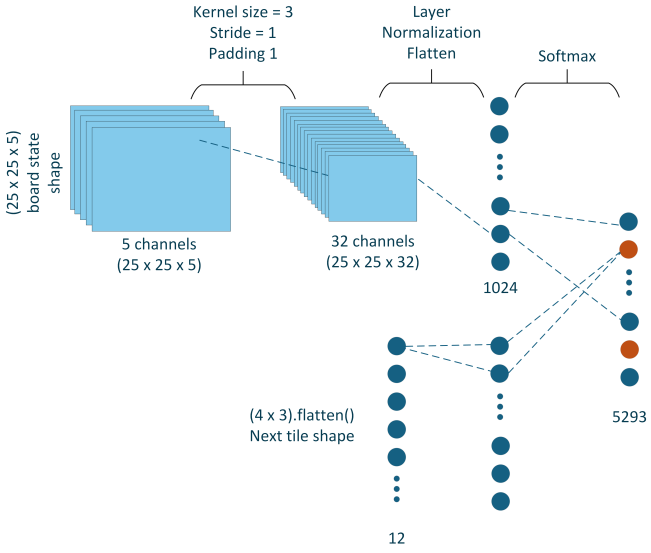

Figure 8: Policy Network Architecture

undergo layer normalization, and feature a linear output layer.

B SAM Algorithm

Algorithm 1 Safety Actor-critic MCTS (SAM)

Require: Actor network $\pi(s|\theta)$, critic network $v(s|\mathbf{w})$, demonstration trajectories $\mathcal{D}$, learning rate α, discount factor γ

Initialize Network parameters θ, $\mathbf{w}$
for each episode **do**
  Initialize state S_0
  for $t = 0$ to $T - 1$ **do**
    Sample action $A_t \sim \pi(S_t|\theta)$
    Execute A_t, observe reward R_{t+1} and next state S_{t+1}
    Sample demonstration trajectory $d \sim \mathcal{D}$
    Calculate cosine similarity $\phi = \frac{S_{t+1}^{[d]} \cdot S_{t+1}}{||S_{t+1}^{[d]}|| \cdot ||S_{t+1}||}$
    Calculate safety reward:
    if A_t is valid **then**
      $Q^D(S_{t+1}^{[d]}) = 23\phi$
    else
      $Q^D(S_{t+1}^{[d]}) = -45$
    end if
    Calculate TD error: $\delta_t = R_{t+1} + \gamma v(S_{t+1}|\mathbf{w}) - v(S_t|\mathbf{w}) + Q^D(S_{t+1}^{[d]})$
    Update critic: $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta_t \nabla_{\mathbf{w}} v(S_t|\mathbf{w})$
    Update actor: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi(A_t|S_t, \theta)\delta_t$
    $S_t \leftarrow S_{t+1}$
  end for
end for