Independent Study Module Report

An Investigation of Reinforcement Learning Algorithms for Mastering the Game of Santorini

Aryan Sarswat - A0200521E
Lim Wei Liang - A0205466E
Roy Chua Dian Lun - A0199930N

University Scholars Programme
National University of Singapore

Contents

1 Introduction

2 Santorini

2.1 Game Rules

Santorini is a 2-player board game on a 5x5 grid, with 3 distinct pieces: building blocks, the worker and the dome. Each turn a player will move his worker to an adjacent tile, then place a building block on a tile adjacent to the moved worker. The building blocks can be stacked up to 3 levels. The worker can only traverse to an adjacent tile that is within 1 level of the worker's current tile (i.e. if the worker is on level 1, he can only move to an adjacent tile of level 0, 1 or 2). When there are 3 building blocks on a tile, a dome can be placed on top of the blocks to prevent another player from reaching level 3. The worker has to traverse from the ground level (level 0) to level 3 in order to win the game. Another way to win is to force the opposing worker to have no adjacent tiles to move to.

2.2 Game Code

An Object-Oriented Programming (OOP) approach was employed to simulate the board-game, as it was deemed as the most "user-friendly" approach, allowing ease of code review and overall game design. The main objects that were instantiated were: 1) Worker, 2) Square, 3) Board, and 4) Player. All of the game's code was done in Python 3.9.

1. Worker Class
   The Worker class is a base class, representing a Worker object on the Board. The worker has two pieces of information stored: the player it belongs to, and its position on the board (represented by a tuple of x-y coordinates). It has a method for movement about the board.

2. Square Class
   The Square class represents a "square" on the board, having immutable x-y coordinates (represented as a tuple) as an attribute. It possesses two pieces of information: the level of buildings (represented as an integer ranging from 0 to 4), and whether a Worker is present (represented by either a Worker object or None).

3. Board Class
   The Board class represents the entire state of the board game, which is the main object used for interaction with the players. The space of the board (a 5x5 grid) is represented as a list of 25 Square objects. The Board class has methods for checking terminal states and validity of player actions.

4. Player Class
   The Player class is a super class for all agents and human players. It possesses information about its Worker locations, as well as two methods: one to place its

2 workers at the start of the game, and one to simulate an action (movement of a worker piece, followed by placement of building).

A simulation of the Board game will start with the creation of a Board object and 2 Player objects (either human or AI), and initialize worker placement depending on the Player object. The players (starting from Player A) will take turns doing an action until a terminal state is reached, where either player wins, of which the script will terminate for a single simulation.

3 Methods employed to create AI Agent

3.1 Linear Value Function Approximation

3.2 Neural Networks (Value Function Approximation)

Using Linear Value Function Approximation with minimax is an extremely effective process. In games where the branching factor is not very large, minimax can lead to optimal gameplay by the AI agent. However, due to the relatively large branching factor of Santorini (128 moves per turn), a reasonably fast AI agent can play by looking up to 4 states maximally. This severely handicaps the performance of the AI. This is one of the main reasons to attempt to use a Neural Network to predict the value of a state instead.

Neural Networks are essentially seen as universal function approximators, which can be used to mimic any function. Thus by using the minimax algorithm we can find the true value of a state and use that as our training data. This training data will be fed into the neural network which will adjust its weights and bias to be able to create a function which can accurately determine how strong a certain state is. This will increase the speed at which our agent can predict the value of a state as the minimax algorithm determines the value of the state by back propagating the value from a terminal state, which is why there is no need to look ahead further states as they are already accounted for. This aspect of Neural Networks as function approximators significantly speed up the “thinking-time” of our agent.

One Hot Encoding was employed to turn the board state into a flattened array of size [1,325] containing only ones and zeros. This was used to ensure that the height of the building is not given priority due to its large numbers (1-4).

ANN Architecture

1. Input
   Consists of a 1,325 layer of Neurons as each board state contains 25 squares and there are 13 unique states thus a hot encoded flattened matrix would have 325 ones and zeros.

2. Neuron Layers
   This is actually a hyperparameter which we can change, for our experiments there was one hidden layer consisting of 256 input neurons and 64 output neurons. This layer was chosen arbitrarily.

3. Output
   The output consists of a tensor with one value. This value is between [-1,1] inclusively. The higher its value is the better the state is for Player 1 and the lower it is the better the state is for Player 2. A value of 0 implies that the state is equally strong for both players.

4. Activation Functions
   The Rectified Linear unit (ReLU) activation function was employed to linearize the function and reduce its computational complexity. The Tanh function was also used in the output layer to ensure that the output was between [-1,1]. This not only squashed the results into this range but also did it proportionally. Thus if the value of a state is extremely large it would be reduced to 1 which can be interpreted as a completely winning state for Player 1.

5. The Loss Function
   The Mean Squared Error loss was employed, this ensured that the state values did not explode as it heavily punishes large differences between the true value and the value predicted by the neural network.

Picture of ANN Architecture

3.3 Neural Networks with Convolutional Layers (Value Function Approximation)

Description
In order to make a value function approximator that is capable of converting board information to a state-value, a Convolutional Neural Network (CNN) was used to attempt to produce an accurate state-value given a board state. A CNN is capable of extracting relevant information from a multidimensional array. It is a type of neural network that is commonly employed in fields of Computer Vision, such as feature detection of images. One common example is facial recognition technology, where a CNN is trained and employed to detect facial features.

Rationale
The rationale of employing a CNN in the determination of the state-value of a board, is mainly due to the capability of a CNN in retaining the 2-dimensional structure of an "image" of a board, whilst generating and preserving 2D spatial features that a normal ANN is not capable of. This is because a board is fundamentally 2-dimensional in nature, and the player is traversing in a 2D-board space (3D if the buildings are included). Therefore, features of the Santorini board, like the grouping of high-level buildings, or the proximity of a player to a building, is better extracted by a CNN rather than a simple

1D neural network structure. The feature extraction capability of a CNN is pivotal in determining how much "value" the board is to a player.

CNN Architecture The neural network was split into two distinct sections, the CNN layers (for feature extraction) and the ANN layers (for state-value determination).

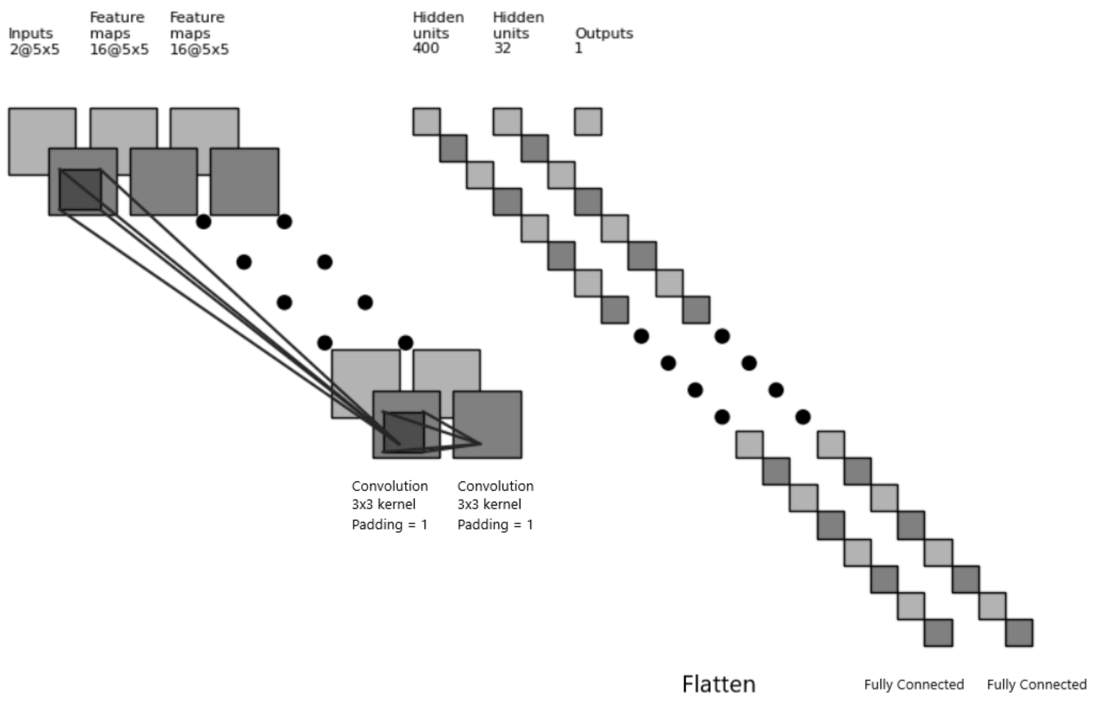

Figure 1: CNN architecture for first value function approximator

1. Input
   From the board, two sources of information were determined to be essential for the neural network to output an accurate state-value: worker position and building level. Hence, the input layer had 2 input features. The board was converted into 2 layers. Each layer is a 2-dimensional 5x5 array, representing the board structure. The first layer consisted of building data, where the possible building levels of 0 to 4 were min-max normalized (0 representing ground level, 0.25 representing level 1 etc). The second later consisted of worker data, where the player's worker was denoted as 1, and the opponent's worker was denoted as -1.

2. Output
   The output was a tensor with a single value that represented the state-value of the board, with the maximum value of 1 being favourable for the player (aka a winning state) and the minimum value of -1 being not favourable.

3. CNN Layers
   The first CNN layer had an input feature value of 2, representing the building levels and worker positions. It outputted 16 feature channels, with a default stride value

of 1. The second CNN layer had similar hyperparameters to the first layer, the only difference being the input features to be 16, corresponding to the out channels of the first layer.

4. Batch Normalization
   Batch normalization was implemented after every convolutional layer to coordinate the updating of weights at the end of every batch.

5. Pooling
   Pooling was not implemented in the CNN as the value function should retain as much information about the board as possible.

6. Kernal Size
   A kernel size of 3 was adopted to capture the possible features of the board, as the worker could move in the adjacent 8 squares, hence the features that are extracted will be more relevant for the player's worker.

7. Padding
   Padding with value of 1 was used to capture the information of the board corners more accurately.

8. Activation Function
   The Rectified Linear unit (ReLU) activation function was adopted for both the CNN and ANN layers (bar the second ANN layer), for less computational complexity and its linear behaviour.

Training of the first CNN Model

The first CNN model was trained on a random agent (who made random worker movements and worker placements) for 5000 iterations. Each iteration represents a complete game of Santorini. For each state, the CNN agent will generate all possible next states, evaluate the states through the CNN, and output the state-value for each possible state. Based on an epsilon-greedy policy, it will then either make a random move, or make a move based on the state with the highest state-value. The state-value of the next state (either the random state or the state with the highest state-value) is saved in an array for backpropagation at the end of the iteration. As the reward given to the agent is only known at the end of the iteration, there is no update of weights to the CNN until the end of one iteration.

Algorithm 1: Training CNN Model

```
Result: Model Trainied
foreach Episode do
    while game is not terminal do
        Generate all possible states;
        Generate all possible states-values for all states;
        r ← generate random number;
        if $r > \epsilon$ then
            Make action with the highest state-value;
        else
            Make random action;
        end
        Store state-value of action made;
    end
    $\epsilon \leftarrow \epsilon \times (\epsilon$ - decay);
    R ← 1 (Agent Win) or -1 (Agent Loss);
    Backpropagate loss based on R;
end
```

3.4 Monte Carlo Tree Search

The Monte Carlo Tree Search algorithm is a variant of the standard Monte Carlo method where a search tree is constructed creating a sort of policy network for an agent to follow. It relies on rollouts to be able to successfully determine the value of a state. This is a very popular algorithm used in many board games; It was successfully implemented by researchers at Deep-Mind to develop AlphaGo and AI which was able to reach grandmaster level ELO at the game of GO. Monte Carlo Tree search is executed by 4 key steps Selection, Expansion, Simulation and Backup. In one run of the algorithm first Selection occurs and if the node is a leaf node it is expanded, after which a number of games are simulated from that leaf node, finally the values obtained from simulating these games are back propagated up to the root node. These four steps will be elaborated upon in detail below:

1. **Selection:** Beginning at the root node, select children node based on a heuristics which involve the visit count and value of the child state. For our experiment the Upper Confidence Bound. More specifically the UCB1 algorithm was used where along with the value of the state the number of visit of the child node and the parent node is used to determine the criteria for selecting a child node. This algorithm allows for the selection of nodes which have not been visited and thus promotes exploration. This algorithm also contains a hyperparameter c which is known as the exploration parameter and it is usually determined empirically but we have used the theoretical value of $\sqrt{2}$.
$$v_i + c\sqrt{\frac{\ln N_p}{n_i}}$$
2. **Expansion:** Once a leaf node is reached, the tree is expanded by adding children to the leaf node, these children are all the possible next states which may follow.

This allows for the growth of the search tree. However if the leaf node is terminal it will not be expanded and in some cases one may choose not expand a leaf node completely.

3. **Simulation:** From the selected node or from a node which has been expanded, a complete episode starting from that node is simulating until the state is terminal. After Which a reward is returned. This is also known as a rollout and the games are simulated with a certain policy; this policy could be a Linear Value Function, a random Policy or even a through a Neural Network which was trained under expert games. Other than the node selected for the simulation, none of the other states in the simulation are stored, this significantly reduces the memory requirements compared to other methods such as Q-tables.

4. **Backup:** The return obtained from the simulated episode is the backed up the path taken to reach the leaf node or if it is the first node in the tree it used to initialize the value for the node. These values are then added up for each simulation in each node in a value sum variable, this will be an indication how good or bad the state is. This is one of the key steps of the MCTS algorithm as instead of using randomly generated episodes, where the goal is to explore as many states as possible, it enriches this data and even though lesser states are explored, the data contains more information than that of a randomly generated episode.

The figure below visually depicts how the algorithm runs

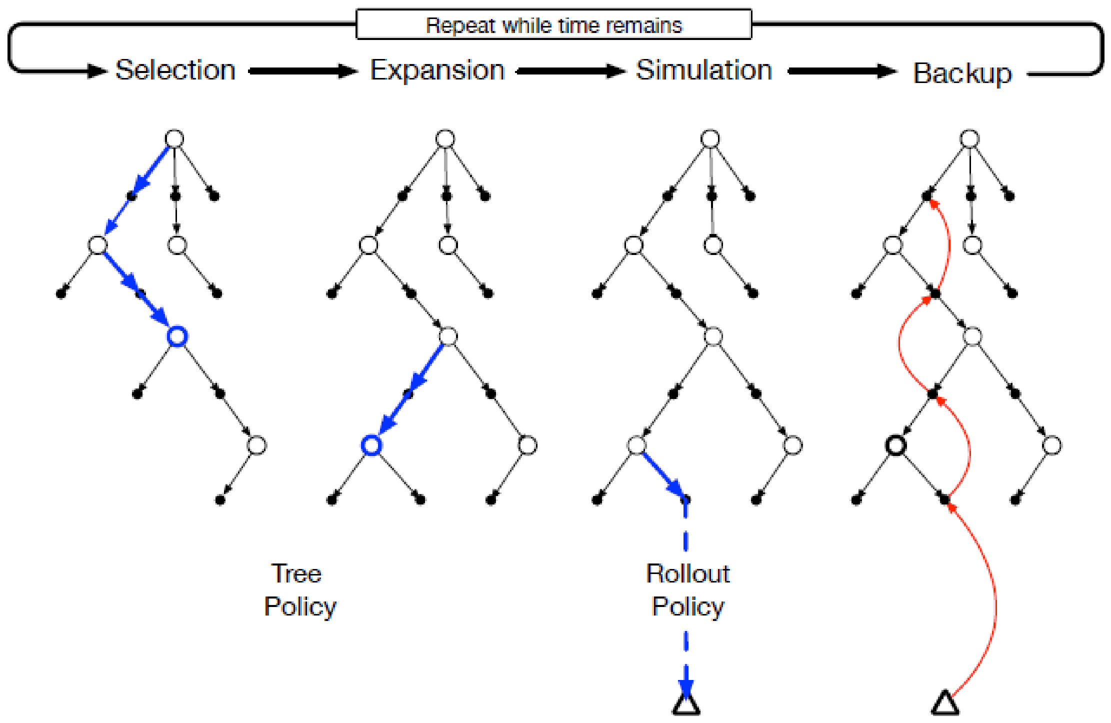

Figure 2: one iteration of MCTS algorithm

Algorithm 2: Monte Carlo Tree Search

```
Input: root
Output: root
Result: One Iteration of MCTS
while time/resource left do
    if root is not expanded then
        Expand Root;
    else
        search_path ← [root];
        current_node ← root;
        while current_node is expanded do
            current_node ← Select child from current_node;
            search_path.append(current_node);
        end
        for Simulation ∈ {1, 2, 3, ....., Number of Simulations} do
            Reward ← Rollout from current_node;
            Backpropagate Reward;
        end
        if Current_node is not terminal then
            Expand current_node;
        end
    end
end
```

This algorithm continues to run until either a time limit is reached or some other computation resource is depleted such as memory. After the algorithm stops running the tree can be used as policy to be able to play games. For each node, the child node with the highest value can be selected, this works because if the visit count and the value of the state is high it implies during the simulation the tree had often selected this node, which means that it often wins in this state.

In addition to the above the previous tree can be improved upon by selecting a leaf node which has not terminated and setting that node as the new root node to construct another MCTS tree. This will allow for greater exploration and better approximation of the value of states.However due to our computational constraints we were unable to run a the MCTS algorithm for a significant upon of time, this is due to the fact that for the value of any node to converge we need a high number of simulation and this is not easily achievable on standard computers as they make take upto 4 to 5 days to complete the construction of one MCTS tree. Thus we developed a variation of the MCTS tree which considers the breadth of the tree instead of a depth.

The Algorithm is as follows: for each node, all the child nodes are selected one by one and for each of these node games are simulated to obtain a value of how good this state is, this value if normalized by the number of games simulated. After all the games have been simulated for all the child nodes, using the UCB1 algorithm a child node is selected which has the highest value, this child node then becomes the new root node. And for each iteration this repeatedly results in a higher depth explored. This algorithm significantly cuts down on the computational resources required, however this comes at a cost; since we are using the new child nodes as the root nodes for another tree, we are essentially doing a one step look ahead, the values are not backed up all the way to the root node, this reduces the chances of the value of the state to converge. Furthermore it constricts the number of nodes which explores as it only searches down the path with the highest UCB1 value.

3.5 Combining the Above

4 Results and Dicussion

5 Conclusion

References

[1] Sutton, R.S., & Barto, A.G. *Reinforcement learning: An introduction.* Cambridge (Mass.): The MIT Press.

[2] Liang, Shiyu, and R. Srikant. *Why Deep Neural Networks for Function Approximation?*. ArXiv:1610.04161 [Cs], Mar. 2017. arXiv.org, http://arxiv.org/abs/1610.04161.