Mastering Hex by Self-Play

Project Team:

Dickbauer Alexander, B.Sc.

Chory Matthias, B.Sc.

Langer Antonia, B.Sc.

Supervisor:

Dr. rer. nat. Sharwin Rezagholi, M.Sc.

Abstract

This paper uses a self-play approach to explore the application of learning the game of Hex, a challenging two-player strategy game with perfect information. We combine Monte Carlo Tree Search (MCTS) with a deep learning agent that uses neural networks to represent a reinforcement learning agent's policy and value functions. The training process results show that the best model produced by the approach beat a random player and the MCTS algorithm without neural networks as player one with a score of 100 to 0. However, the model struggled when playing as player two due to overfitting caused by a bug in the data generation script. Still, these results highlight the potential of combining MCTS and deep learning to achieve outstanding performance in complex games like Hex.

Introduction

Hex is a board game played on a diamond-shaped board, where players take turns placing colored pieces on the board. The objective is to create a connected path of pieces from one side of the board to the other while blocking the opponent's path. Hex is a challenging game due to its high branching factor and complex strategic interactions, making it an ideal candidate for AI research.

Monte Carlo Tree Search (MCTS) is a widely used algorithm for playing games[1], which involves building and exploring a search tree, gradually improving its estimates of the values of the different actions and paths. MCTS can be improved by incorporating deep learning agents that use neural networks[2] to learn from experience, potentially achieving superhuman performance in challenging games.

In this paper, we explore the combination of MCTS with a deep learning agent to learn the game of Hex through self-play. The basic idea is to use the neural network to estimate the action or state values at each tree node instead of using the default policy or value function estimates. This is done by training the network on a dataset of game positions and outcomes created using self-play.

Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) consists of four steps selection, expansion, simulation, and backpropagation. These steps are used iteratively to construct and explore a search tree, gradually improving its estimates of the values of the different actions and paths.

The first step of the MCTS algorithm is to traverse the tree from the root node to a final game state, a leaf node. A child node is selected at each level using a tree policy, balancing the tree nodes' exploration and exploitation. The tree policy typically selects nodes with high-value estimates but also considers the number of visits to the node to encourage the exploration of less-visited nodes.

Once a leaf node is reached, the MCTS algorithm creates one or more child nodes, representing the possible actions that can be taken from the current state. The new nodes are added to the tree, and the algorithm continues to the simulation step.

In the simulation step, the algorithm plays out a hypothetical game starting from the newly added child node, using a default policy that selects moves at random. The simulation continues until a terminal state is reached, and the resulting reward is used to update the value estimates of the nodes along the path from the root node to the expanded node.

The final step of the MCTS algorithm is to update the statistics of the nodes along the path from the root node to the expanded node, using the result of the simulation. The node values are updated by incrementing their visit counts and averaging their reward estimates with the new value.[1]

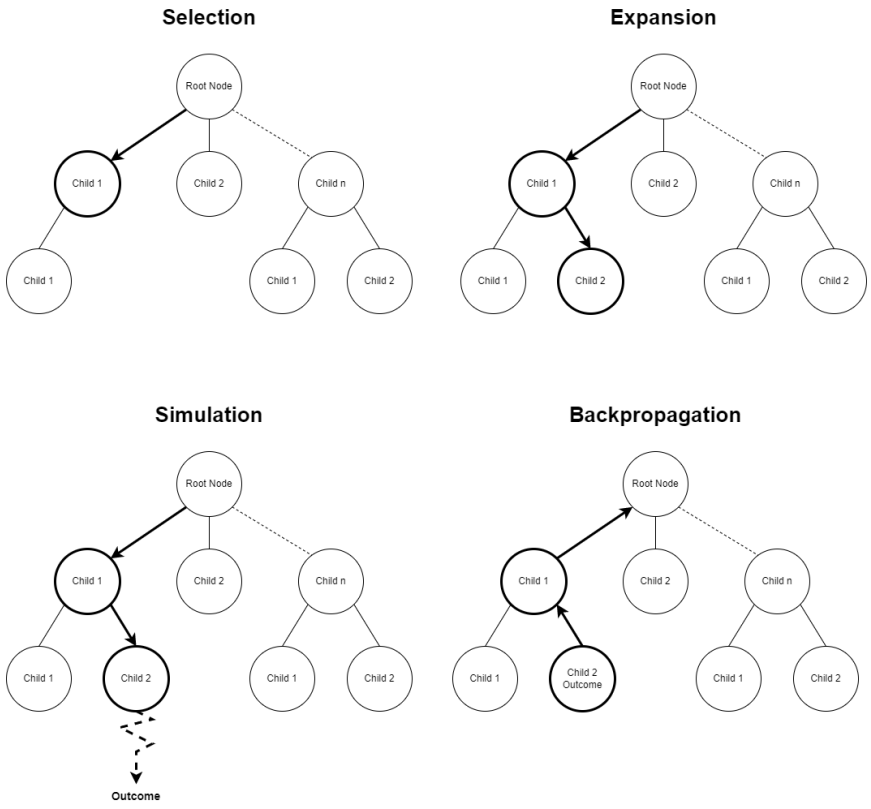

Figure 1 - MCTS Steps

Deep learning Agent

To improve the MCTS, a deep learning agent can be added by replacing the default policy and the value function with a neural network trained using reinforcement learning. This allows the agent to learn from experience and improve its decisions over time, potentially achieving superhuman performance in challenging games.

The basic idea is to use the neural network to estimate the action or state values at each tree node instead of using the default policy or value function estimates. This is done by training the network on a dataset of game positions and outcomes.

During the selection step of the MCTS algorithm, the neural network is used to estimate the value of each child node, selecting the node with the highest value. This policy is then updated after each iteration of the MCTS algorithm based on the new data collected during the simulations.

During the backpropagation step, the neural network is used to update the value function estimates of the nodes along the path from the root node to the expanded node, using the simulation result. The node values are updated by incrementing their visit counts and averaging their reward estimates with the new value predicted by the neural network.[2]

Model

The model is a convolutional neural network (CNN) designed to be used as the deep learning agent in the MTCS. The network takes as input a game board represented as a two-dimensional array of size n x n, where n is the size of the board, and produces as output two tensors: a probability distribution over the possible actions (pi), and a value estimate for the current state of the game (v).

The model's architecture consists of four convolutional layers, two fully connected layers, and two output layers. The convolutional layers are designed to extract local features from the input game board, while the fully connected layers aggregate these features to produce the final output.

During the forward pass, the input game board is first reshaped into a tensor of shape (batch_size, 1, board_x, board_y), where batch_size is the number of input samples, and board_x and board_y are the dimensions of the game board. The convolutional layers process the input tensor, followed by a batch normalization layer and a rectified linear unit (ReLU) activation function. The output of the last convolutional layer is then flattened into a one-dimensional tensor and fed into the fully connected layers.

The first fully connected layer is followed by a batch normalization layer and a ReLU activation function designed to reduce the dimensionality of the input tensor. The second fully connected layer is similarly followed by a batch normalization layer and a ReLU activation function, producing the model's final output. The output consists of two tensors: a probability distribution over the possible actions (pi), which is obtained by applying a softmax function to the output of the second fully connected layer, and a value estimate for the current state of the game (v), which is obtained by applying a hyperbolic tangent (tanh) function to the output of the second fully connected layer.

The dropout parameter controls the amount of regularization applied to the model during training and helps to prevent overfitting.

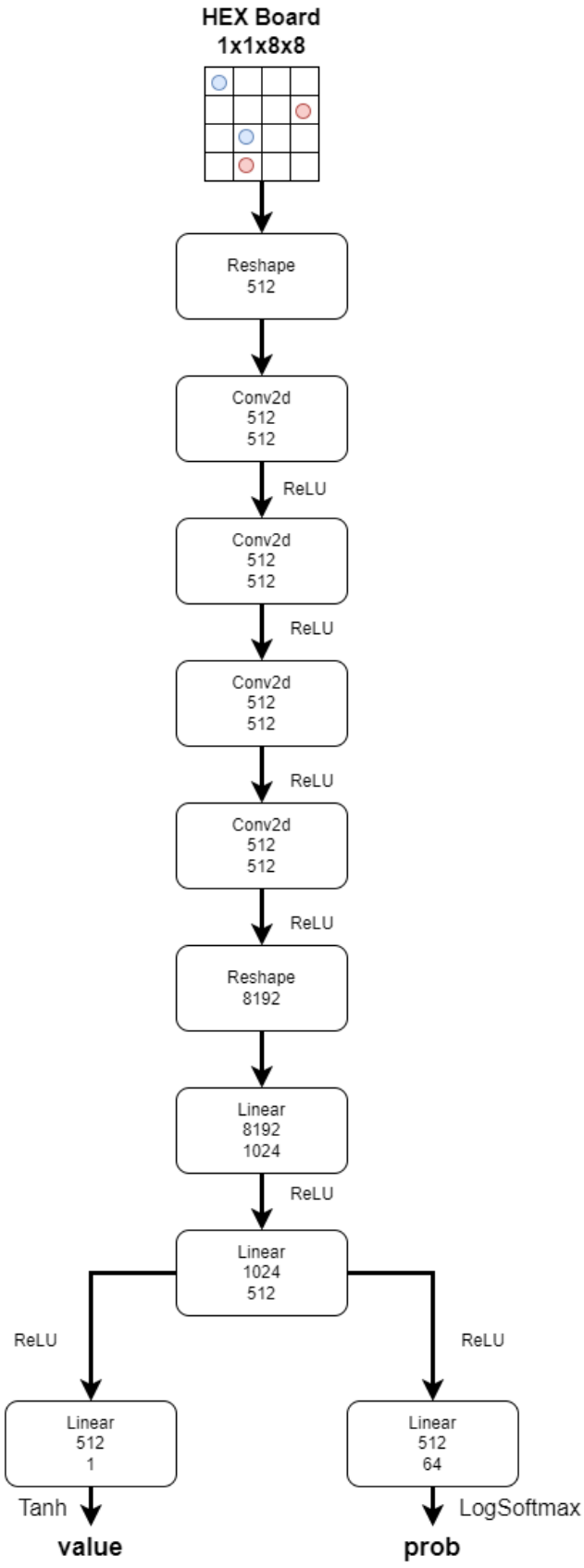

Figure 2 - Model Architecture

Training

Training the deep learning agent for the MCTS algorithm in Hex is a computationally intensive task that requires large amounts of data and significant computational resources. To address this challenge, the training was performed on the Slurm cluster of FH Technikum Wien, which allowed the workload to be distributed across multiple nodes and efficiently managed the computational resources, which reduced the time of one iteration from approximately 5 hours to approximately 45 minutes.

The training process consisted of five steps, orchestrated by a main job that scheduled and monitored the execution of the individual tasks. The five steps were as follows:

Self-play for data generation

In this step, a hundred instances of the MCTS algorithm were run in parallel on different cluster nodes, each playing games against itself and generating training data in game positions and outcomes. Each instance was configured to play two games, and the second game was played with a flipped board.

Collecting and formatting the data

Once the self-play process was complete, the training data was collected from the cluster nodes' multiple outputs and formatted into a single dataset to train the deep learning model. This involved merging the data from the different nodes and storing the resulting dataset in a numpy binary file for later use.

Training the model

The formatted dataset was then used to train the deep learning model. Initially, the training process was configured to run for 50 epochs but was later increased to 100. In an earlier version of the training script, multiple nodes were used for the training, but since the process slowed down significantly using multiple nodes, only one dedicated node was used for this step.

Evaluating the currently trained model vs. the previous best model

Once the model training was complete, the current model's performance was evaluated against the previous best model by playing 40 games on 20 cluster instances between the current and best models.

Selecting the new best model

The final step of the training process was to collect the evaluation results, process them and select the new best model. If the current model had a win rate of 55% or higher, it was selected as the new best model and stored for future use. Otherwise, the previous best model was retained, the iteration was increased by one, and the training process continued with a new round of self-play and data generation if the current iteration was below 50.

Results

After 50 iterations of training the deep learning agent for the MCTS algorithm, the results were auspicious. The best model produced by the training process beat a random player and the MCTS algorithm without using a neural network as player one, with a score of 100 to 0 in each case. This indicates that the deep learning agent could learn practical strategies and patterns for playing Hex and use these strategies to achieve a high level of performance.

However, the model lost most games when playing as player two against a random player and the MCTS algorithm. This indicates that the model was overfitted to a particular starting position. In other words, the model may have learned to play effectively from one position but could not generalize this knowledge to other positions.

The model's poor performance when playing as player two was due to a bug in the data generation script, where only training data was created if player one won. This means that the model was not trained on a sufficiently diverse set of starting positions and game outcomes resulting in the observed overfitting.

Even though the bug was resolved in iteration 15, in an attempt to save the training progress, the overfitting was already significant enough that the best model as player one was consistently winning, thus creating biased train data. This highlights the importance of adequately validating the training data and ensuring it is diverse and representative of the full range of possible game states.

These results demonstrate the potential of combining MCTS and deep learning to achieve superhuman performance in complex games like Hex.

References

[1] Metropolis, N. and Ulam, S. (1949) The Monte Carlo Method. Journal of the American Statistical Association, 44, 335-341. https://doi.org/10.1080/01621459.1949.10483310

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, & Demis Hassabis (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529, 484-489.*

Figures