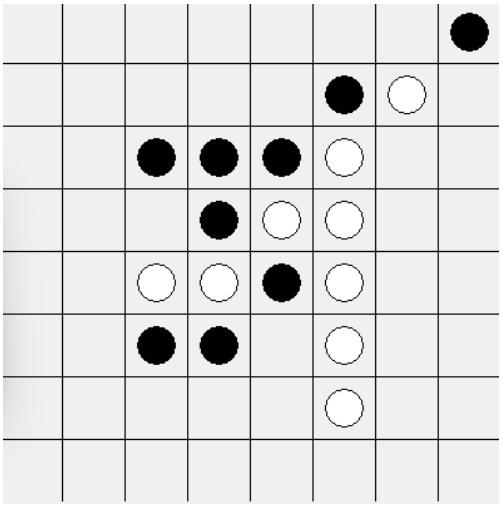
EE435 Project: Game of Go



Presentation video link: https://youtu.be/lM5a-Op4RkI

1. Introduction

Gomoku, also known as Five in a Row, is a classic strategy board game that involves two players taking turns to place stones on a grid. The objective is to be the first to form an unbroken chain of five stones horizontally, vertically, or diagonally. Due to its simple rules yet complex strategies, Gomoku serves as an excellent testbed for artificial intelligence (AI) research.

We have successfully completed the following tasks as part of this project:

- **Training Neural Network:** We trained a neural network with a comprehensive set of parameters using historical game data. The network was optimized to identify winning patterns and make strategic decisions.
- **Using MCTS:** The Monte Carlo Tree Search algorithm was employed to simulate numerous possible game states. This allowed the AI to select the most promising moves by evaluating the potential outcomes.
- **User Interface:** A user-friendly interface was developed to enable human players to interact seamlessly with the AI. The interface provides real-time feedback and allows users to play against the AI in an intuitive manner.

By combining MCTS with a trained neural network, our AI makes smart decisions and plans its moves well. This report covers the methods, implementation, and results of the project, showing how effective these advanced AI techniques are in playing strategic games.

2. Methodology

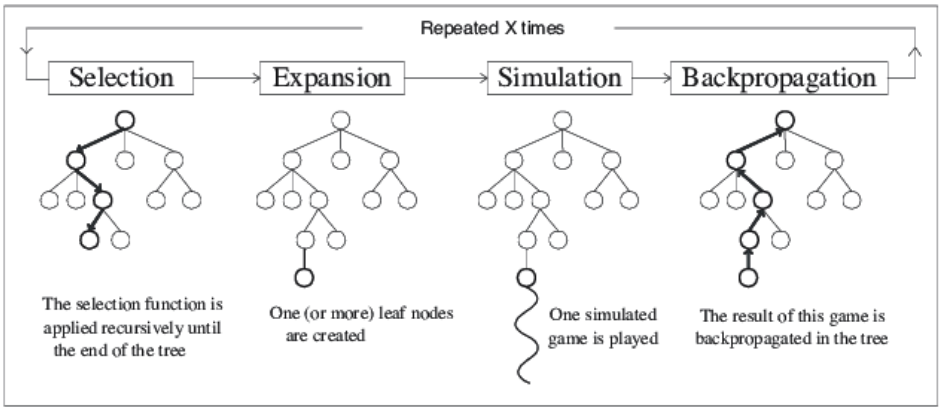


Figure 1. Monte Carlo Tree Search (MCTS) Process

2.1. Monte Carlo Tree Search (MCTS).

- **Selection**:
  - Starting from the root node, successive child nodes are selected until a leaf node is reached.
  - Selection is based on the Upper Confidence Bound for Trees (UCT) algorithm, which balances exploitation (choosing moves that have been successful in the past) and exploration (choosing moves that have not been tried as much).
  - The UCT formula used to select child nodes is:

$$Q(s,a) + c \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}}$$

    where:
    - $Q(s,a)$ is the average reward of taking action a from state s.
    - $N(s)$ is the number of times state s has been visited.
    - $N(s,a)$ is the number of times action a has been taken from state s.
    - c is a constant that determines the level of exploration.
- **Expansion**:
  - If the selected leaf node is not a terminal state (i.e., the game is not finished), it is expanded by adding one or more child nodes corresponding to possible moves.

  - Each child node represents a possible action that can be taken from the current state.
- **Simulation**:
  - From the newly expanded node, a simulation (also known as a rollout) is run to a terminal state.
  - Moves are selected randomly during the simulation.
  - The outcome of the simulation (win, lose, or draw) is used to estimate the potential of the node.
- **Backpropagation**:
  - The results of the simulation are propagated back up the tree, updating the nodes' statistics.
  - The visit count $N(s)$ and the average value $Q(s, a)$ for each node are updated.
  - This process ensures that the information gained from the simulation is used to inform future decisions.

2.2. **Neural Network Training.**

- **Network Architecture**:
  - The neural network used in this project is a policy-value network designed to predict the move probabilities and the value of the game state.
  - It consists of three main types of layers:
    * **Convolutional Layers**: These layers extract spatial features from the board state. The network has three convolutional layers with ReLU activation, which progressively capture higher-level patterns in the game board.
    * **Policy Head**: This part of the network outputs a probability distribution over all possible moves. It includes a convolutional layer followed by a fully connected layer with a softmax activation function.
    * **Value Head**: This part evaluates the board state, predicting the likelihood of winning from that state. It includes a convolutional layer, followed by fully connected layers, and ends with a tanh activation function to output a value between -1 and 1.
- **Training Process**:

  - The network is trained using self-play data generated by the MCTS algorithm. Each game played by the AI provides data in the form of board states, move probabilities, and game outcomes.
  - The loss function used for training combines two components:
    * **Value Loss**: Measures the difference between the predicted value of the board state and the actual game outcome using mean squared error.
    * **Policy Loss**: Measures the difference between the predicted move probabilities and the probabilities generated by MCTS using cross-entropy.
  - An L2 regularization term is added to the loss function to prevent overfitting.
  - The optimizer used for training is the Adam optimizer, which adjusts the learning rate during training to improve convergence.
- **Training Data**:
  - The training data consists of board states, the move probabilities generated by MCTS, and the outcomes of the games.
  - Data augmentation techniques, such as rotation and flipping of the board, are used to increase the diversity of the training data and improve the network's generalization ability.

3. Implementation

3.1. Monte Carlo Tree Search (MCTS).

- **Initialize**
  - Initialize the root node: `root = TreeNode(parent=None, prior_p=1.0)`
- **Playout**
  - For each playout (total *n_playout* times):
    - Copy the current state: `state_copy = copy.deepcopy(state)`
    - Set the current node to the root: `node = root`
    - **Selection**: Traverse the tree until a leaf node is reached:
      - While `node` is not a leaf:
      - Select the next node: `action, node = node.select(c_puct)`
      - Update the state with the selected action: `state_copy.do_move(action)`
    - **Expansion**: If the leaf node is not a terminal state:
      - Expand the node: `action_probs, leaf_value = policy_value_fn(state_copy)`
      - Add child nodes for each possible action: `node.expand(action_probs)`
    - If the leaf node is a terminal state:
      - Determine the leaf value: `leaf_value = determine_leaf_value(state_copy)`
    - **Simulation**: Run a random playout from the expanded node:
      - Simulate the game to the end: `leaf_value = simulate(state_copy)`
    - **Backpropagation**: Update the nodes' statistics:
      - Update the current node and its ancestors with the simulation result: `node.update_recursive(-leaf_value)`
- **Final Move Selection**
  - Select the move with the highest visit count: `best_move = argmax(child._n_visits for child in root._children)`
  - Return the best move: `return best_move`
- **Helper Functions**
  - `determine_leaf_value(state)`
    - Check if the game has ended: `end, winner = state.game_end()`
    - If the game has ended:
      - If it is a draw, return 0.0
      - If the current player has won, return 1.0
      - If the opponent has won, return -1.0

    * If the game has not ended, return the network evaluation: `policy_value_fn(state)[1]`
  - `simulate(state)`
    * While the game has not ended:
      · Select a random action: `action = select_random_action(state)`
      · Update the state with the selected action: `state.do_move(action)`
    * Return the leaf value: `return determine_leaf_value(state)`

3.2. **Neural Network with PyTorch.**

- **Initialization**
  - Initialize the neural network with convolutional and fully connected layers.
  - Set the optimizer (Adam) with L2 regularization.
  - Load model parameters if a pretrained model is provided.
- **Forward Pass**
  - **Input**: A batch of states.
  - **Output**: A batch of action probabilities and state values.
  - Pass the input through the common convolutional layers:
    * `x = F.relu(self.conv1(state_input))`
    * `x = F.relu(self.conv2(x))`
    * `x = F.relu(self.conv3(x))`
  - **Action Policy Head**:
    * `x_act = F.relu(self.act_conv1(x))`
    * `x_act = x_act.view(-1, 4*self.board_width*self.board_height)`
    * `x_act = F.log_softmax(self.act_fc1(x_act))`
  - **State Value Head**:
    * `x_val = F.relu(self.val_conv1(x))`
    * `x_val = x_val.view(-1, 2*self.board_width*self.board_height)`
    * `x_val = F.relu(self.val_fc1(x_val))`
    * `x_val = F.tanh(self.val_fc2(x_val))`
  - Return action probabilities and state values.
- **Policy Value Function**
  - **Input**: Current state of the board.
  - **Output**: A list of (action, probability) tuples and the value of the state.
  - Compute the legal positions and current state.

  - Pass the current state through the policy-value network to get action probabilities and state value.
  - Return the action probabilities and state value.

- **Training Step**
  - **Input**: Batches of states, MCTS probabilities, winners, and learning rate.
  - **Output**: Loss and entropy.
  - Zero the parameter gradients.
  - Set the learning rate.
  - Perform a forward pass to compute action probabilities and state value.
  - Compute the loss:
    * Value loss: `value_loss = F.mse_loss(value.view(-1), winner_batch)`
    * Policy loss: `policy_loss = -torch.mean(torch.sum(mcts_probs * log_act_probs, 1))`
    * Total loss: `loss = value_loss + policy_loss`
  - Perform backpropagation and optimization step.
  - Compute policy entropy for monitoring.
  - Return the loss and entropy.

- **Save and Load Model**
  - Save model parameters to a file.
  - Load model parameters from a file.

```
batch i:198, episode_len:7
kl:0.03053,lr_multiplier:0.444,loss:2.5433380603790283,entropy:1.9665303230285645,explained_var_old:0.362,explained_var
_new:0.439
batch i:199, episode_len:11
kl:0.02727,lr_multiplier:0.444,loss:2.453291416168213,entropy:1.949414610862732,explained_var_old:0.405,explained_var_n
ew:0.478
batch i:200, episode_len:7
kl:0.02108,lr_multiplier:0.444,loss:2.420064926147461,entropy:1.91493821144104,explained_var_old:0.417,explained_var_ne
w:0.499
current self-play batch: 200
num_playouts:1000, win: 6, lose: 4, tie:0
New best policy!!!!!!!
batch i:201, episode_len:10
kl:0.03402,lr_multiplier:0.444,loss:2.537860631942749,entropy:1.9045612812042236,explained_var_old:0.334,explained_var_
new:0.413
batch i:202, episode_len:7
```

FIGURE 2. Training Process

4. Results

4.1. **Example 1.** Starting from an evenly balanced position, no matter which direction we attempted to expand or block, the AI consistently found optimal moves to extend in the opposite direction.

Black : Human , White: AI

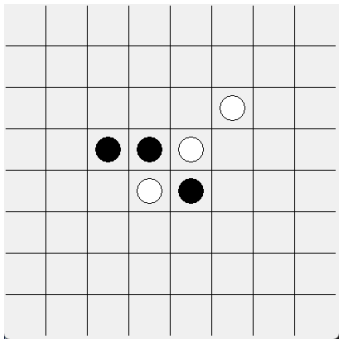
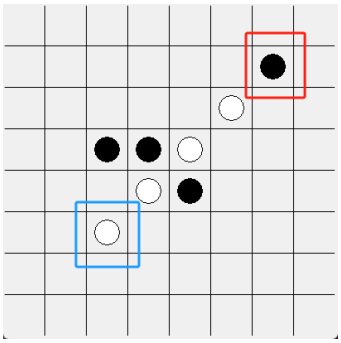
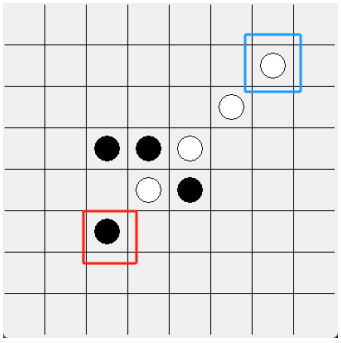
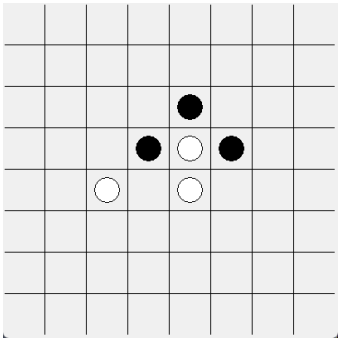


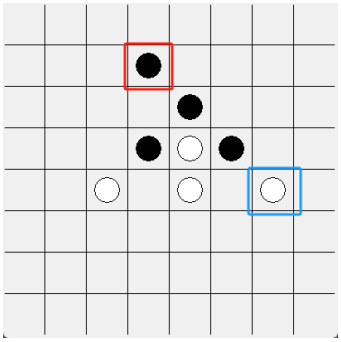
Figure 3. Initial State





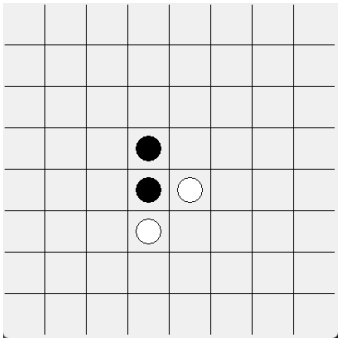
4.2. **Example 2.** The AI effectively blocks the player's strategic expansion.
Black : Human , White: AI

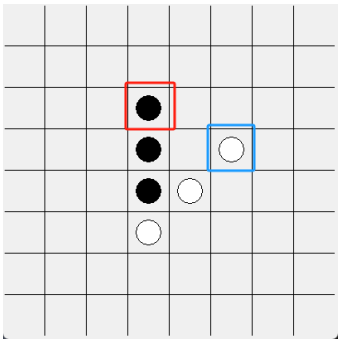




4.3. **Example 3.** The AI has demonstrated a strategic approach to enhance its position by placing a stone that not only disrupts the player's potential gains but also fortifies its alignment on the board.

Black : Human , White: AI





5. Limitation

In the given example(Figure 4), the game ends in a tie, which highlights a significant limitation of playing on an 8x8 board. When both players, or in this case, the human player focus mainly on defensive strategies without aggressive advancements, the board quickly becomes filled without a clear victor.

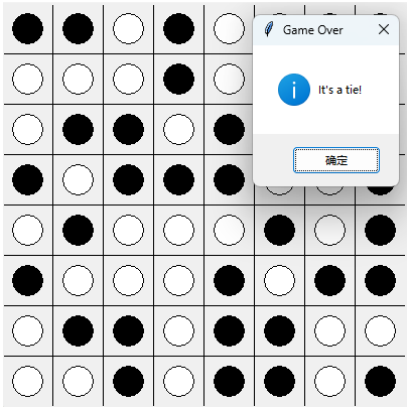


Figure 4. Tie example

One reason causes this is the limitation of training resources. Training a larger board may consume many times more time.

Also, Integrating neural networks more deeply into the selection and evaluation phases could provide more accurate assessments of game states without needing extensive playouts.