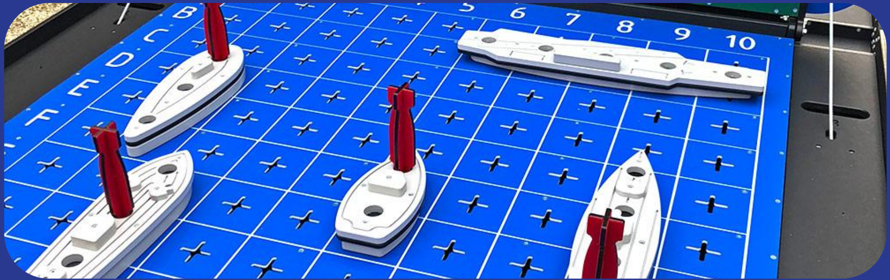
Kahaan Shah | Saptarishi Dhanuka | Suyog Joshi

Bot-tleship

Final Presentation | Artificial Intelligence | Monsoon ‘24



Refresher

- Agent that can play against a human so people can play battleship against a bot
- We chose some baselines
  - Random
  - Human-like
- Previous work includes algorithms like:
  - Monte Carlo Tree Search
  - Probability-based
  - RL algos
- We also wanted to compare the bots against each other to find the best performing one
- Expectations:
  - The system can complete the game in as few moves as possible
  - Reasoning based on current state and hits on board - where to target next shot for highest likelihood of hit
  - Several heuristics exist to optimize for battleship - does the AI learn these?

Baselines

Random Bot

- Chooses un-attacked squares randomly
- On a 10x10 board with 5 ships, takes average of 96 moves to complete the game (destroy all ships), SD ~4
- Meaning the random bot takes takes almost the maximum number of moves to win

Baselines

Heuristics-based human-like bot

- Follows the algorithm used most commonly by human players:

1. **Hunt Mode**: Randomly attacks squares on the board until it hits a ship.
2. **Target Mode**: After a hit, attacks adjacent squares to determine the ship's orientation (horizontal or vertical).
3. **Destroy Mode**: Focuses attacks along the determined orientation to sink the ship completely.
4. **Reset:** Once a ship is sunk, returns to Hunt Mode to find the next ship.

- On a 10x10 board with 5 ships, takes an average of 76.23 moves to complete the game

AI-Based Bots

We implemented three bots:

- MCTS-Bot
- DQN-Bot
- PPO-Bot

Monte-Carlo Tree Search

- We have a very large tree space, so we use a probabilistic approach
- The algorithm has four steps:
  - Selection: We select a child to explore
  - Expansion: We expand the selected node by choosing a new child
  - Simulation: We simulate a potential win and assign a reward based on the validity
  - Back propagation: We send the reward back up the tree

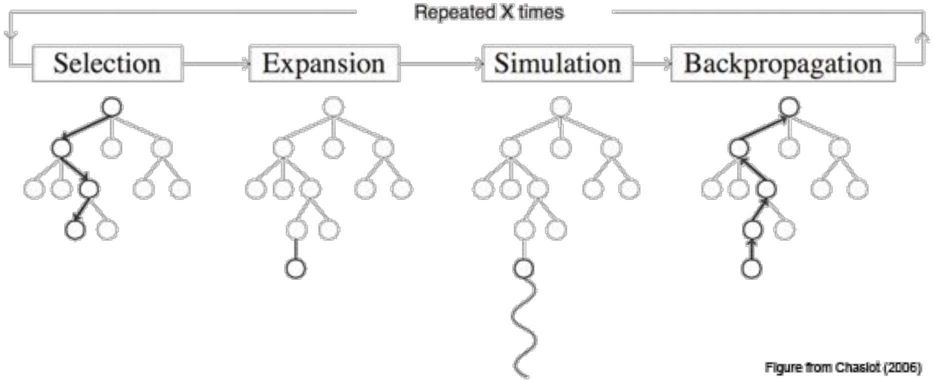


Figure from Chaslot (2006)

Choosing the “Best Child”

- After all the simulations are done we choose the best child
- The “best” child is given by the UCT formula: $UCT(node_i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$
- This formula balances the “wins” that a child has and ensuring that all the expanded children are given a chance to be explored

A Problem

- Battleship is a POMDP, we don’t know what the outcome will be of a move, unlike say Chess
- Solution:
  - We use heuristics like a heatmap to help the bot choose moves (i.e. give more weight to explore moves near known hits)
  - Instead of deciding whether there was a win, we simulate that all the children were hits, then evaluate if the final position is “valid” and based on this give a reward
  - Reward function: (valid squares/ship squares) * (valid ships/total ships)^2
- We backpropagate this reward after each simulation

Q-Learning

- Q learning uses a Q-table which contains all possible states, and the reward when going from State-A to State-B
- It isn’t possible to have a large table for a game like battleship, where there are so many possible states and a probabilistic result
- So we use Deep Q-Learning which does not require the table

Deep Q-Network

- A DQN based AI attempts to “approximate” the Q-table by observing many “episodes” of events and regressing a manifold based on them
    - Reinforcement learning is used to train the neural network used for approximation
- Useful for sequential decision-making problems
- Balances exploration (trying random shots) and exploitation (targeting a ship)
- We trained a CNN to calculate the expected q-values given the state and an action

Why CNNs?

- CNNs can extract spatial patterns (i.e. grid locations of ships)
- The convolutional filters focus on small patches of the board (5x5)
    - Can capture local patterns like clusters of hits or ship alignments
    - Fully connected layers flatten the data, losing spatial information

DQN Architecture

- State-Action Value Function: Q(*state, action)*
  - Given the current state, what is the expected award for *action*?
  - Approximated by CNN
- Replay buffer
  - Stores past experiences - *(state, action, reward, next state, done)*
  - During training, the agent samples a random batch of past experiences to train the CNN
  - Random sampling ensures diverse training data
- Target Net
  - The q-values and *next state* calculated by the main network are fed to a target net (updated less frequently)
  - Used to calculate the expected future reward and train the main network

CNN Architecture

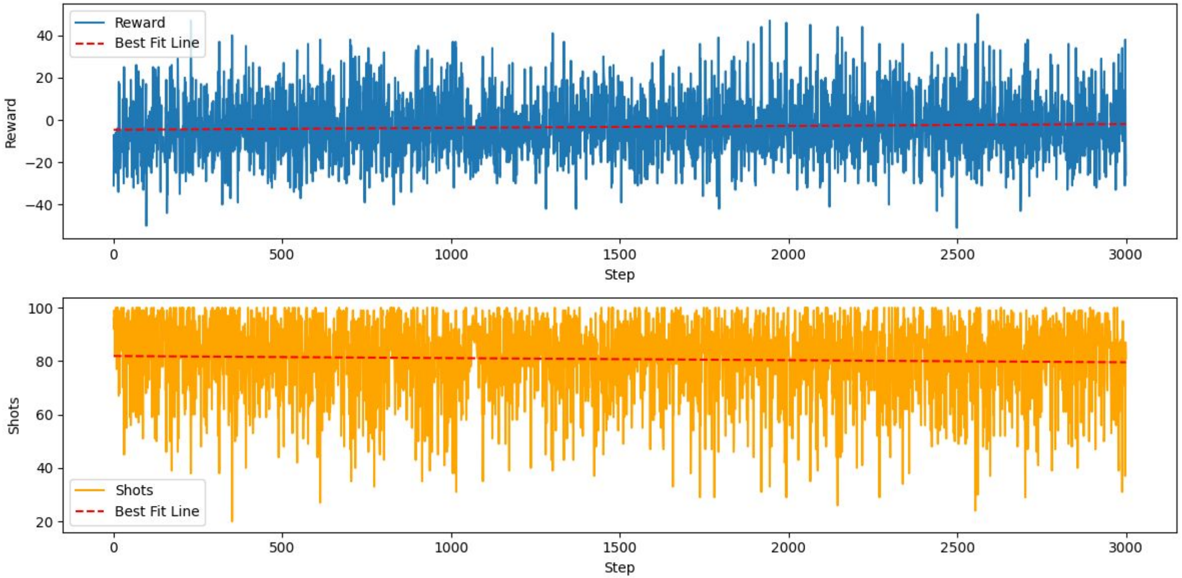
- Our CNN architecture has 7 layers:

1. Conv1: Extracts spatial patterns over the whole board (kernel size = 10).
2. Max Pooling: Reduces spatial dimensions, highlights key features.
3. Conv2: Captures finer spatial details (kernel size = 5).
4. 3 Dense Layers: Encodes the spatial features into q-values.
    a. Outputs a q-value for each grid position (100 values)

Training the DQN

- Attempted to use Gymnasium and stable_baselines with a custom environment for battleship
  - Did not support masking invalid actions, so tried setting reward for invalid shots very low
  - Despite this, bot was not learning to choose valid moves
  - Had to create own environment and training setup
- Could only train over 3000 games because of low computation speed
  - Took 6 hours
- 3000 steps was not enough for the bot to reach the performance we expected
  - Still showed some improvement over time, and better performance than RandomBot
  - With more training, we would expect performance closer to MCTS and PPO

Training the DQN



- Slope for rewards: 0.00089
- P-value for the slope of rewards: 0.003 (statistically significant)

- Slope for shots: -0.00078
- P-value for the slope of shots: 0.009

Proximal Policy Optimization (PPO)

- Learns policy directly in an on-policy manner
- It tries to keep the new policies close enough to the old policies so as to not overshoot
- Clipped surrogate objective comes into play that stops new policy from going too far away, controlled by a hyperparameter ε (this is mainly what handles exploration-vs-exploitation)

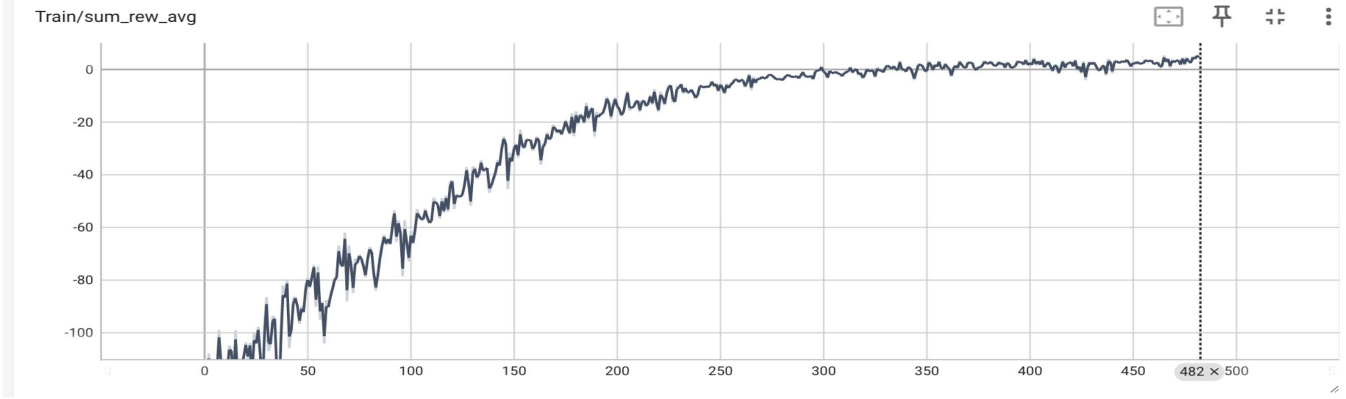
PPO Architecture

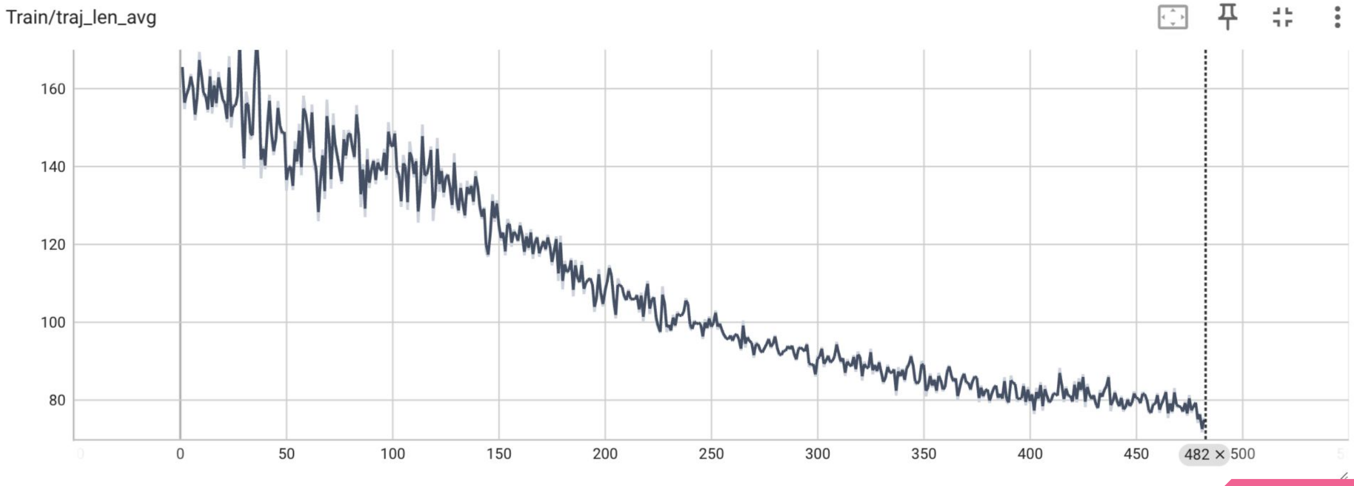
- Actor-Critic Framework:
  - Actor: Proposes actions based on the current policy
  - Actor Network: Probability distribution over actions
  - Critic: Evaluates the actions by estimating the value function
  - Critic Network: Value estimate for given state
- Reward and Penalty formulation:
  - If attacks previously attacked square : **Large penalty**
  - If gets a hit : **Equivalently large reward**
  - If misses : **Small penalty**

PPO Training

- PPO interacts with the environment (a discrete multi-dimensional space)
- Actor updates its policy to choose actions based on feedback of the rewards from the critic
- Policy is clipped to avoid instability and overshooting
- Advantages measure how good a policy is compared to the baseline
- Entropy term encourages exploration
- Discount factor ensures future rewards are discounted

PPO Training



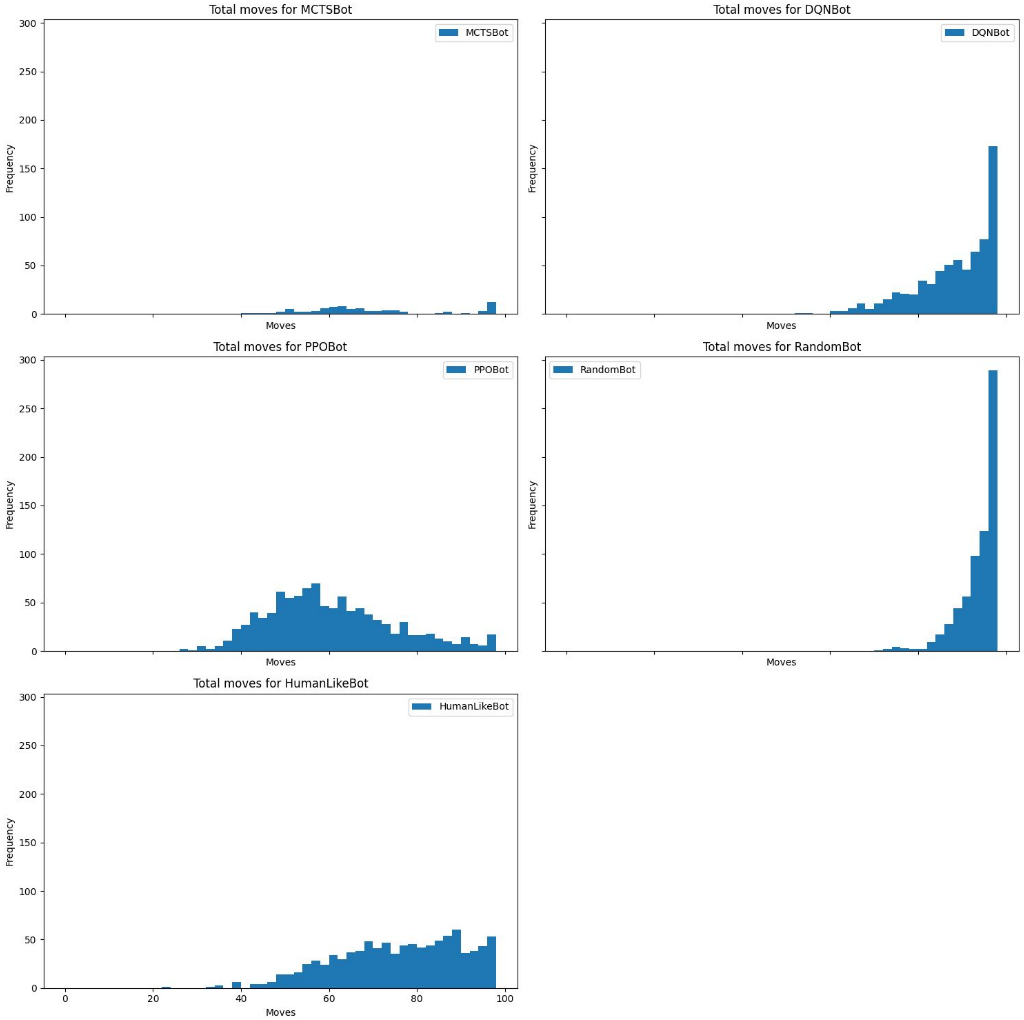


Results

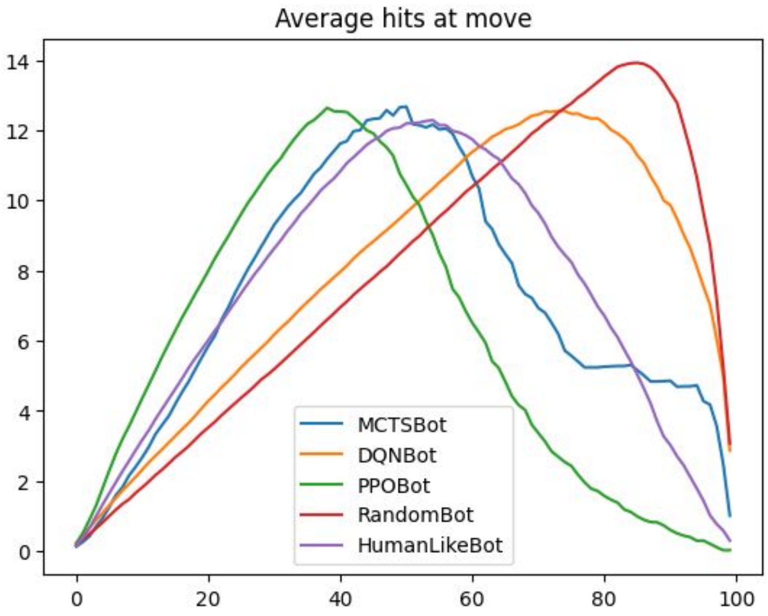
We simulated each bot for a 1000 games on random boards till it won (mcts for 100 due to time constraints):

Bot	Mean Moves	Median Moves	Max Moves	Min Moves
Random	95.47	97	100	70
Human-Like	76.23	77.5	100	22
MCTS (100)	73.55	67.5	100	41
DQN	91.55	95.0	100	52
PPO	<u>60.02</u>	<u>58.0</u>	<u>100</u>	<u>26</u>

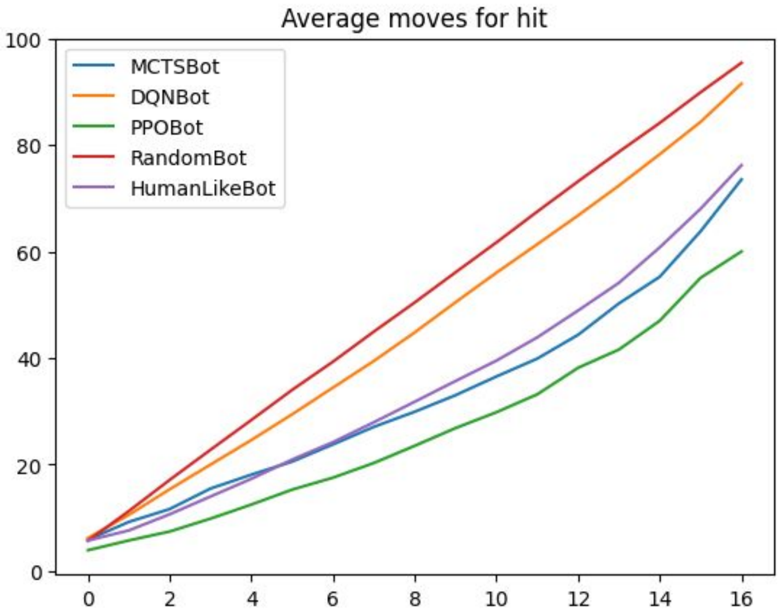
Results


Total moves for MCTSBot
MCTSBot
Total moves for DQNBot
DQNBot
Total moves for PPOBot
PPOBot
Total moves for RandomBot
RandomBot
Total moves for HumanLikeBot
HumanLikeBot
Frequency
Moves
0
50
100
150
200
250
300
0
20
40
60
80
100

Results

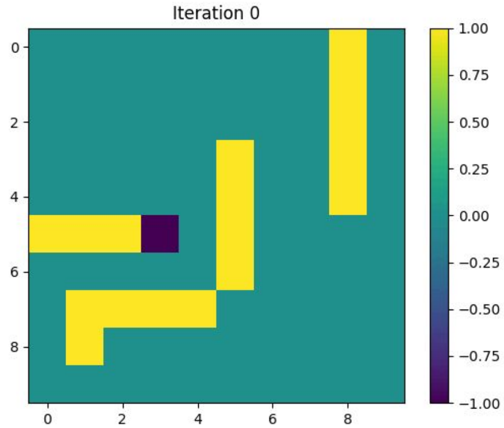


Average hits at move
14
12
10
8
6
4
2
0
0
20
40
60
80
100
MCTSBot
DQNBot
PPOBot
RandomBot
HumanLikeBot

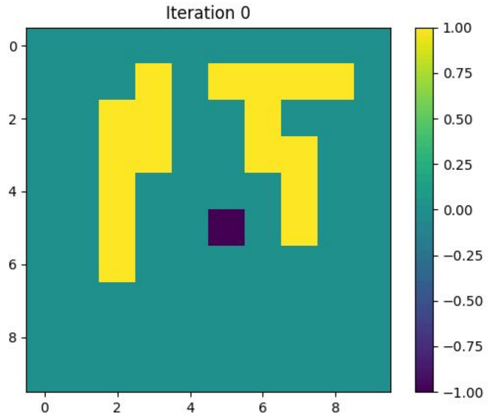


Average moves for hit
100
80
60
40
20
0
0
2
4
6
8
10
12
14
16
MCTSBot
DQNBot
PPOBot
RandomBot
HumanLikeBot

Sample Runs



MCTS



PPO

Head to Head

We made each bot play the others for 1000 games:

	Random	Human-Like	MCTS (100)	DQN	PPO
Random	X	90.4% / 74	85% / 69.2	62% / 88.14	98% / 59.53
Human-Like	9.6% / 91	X	58% / 59.00	19% / 82.71	80.4% / 57
MCTS (100)	15% / 92.60	42% / 73.40	X	26% / 84.88	57% / 56.1
DQN	38% / 91.92	81% / 74.06	74% / 65.90	X	96% / 59.60
PPO	2% / 90	19.6% / 63	43% / 57.86	4% / 78.43	X

Insights

- We learnt about three new algorithms
- It is difficult to train to play partially observable processes
- Battleship is a relatively small game where the winner still has a high degree of luck on their side due to randomness
- Relatively simpler methods like MCTS could perform comparably to Deep RL methods

Demo

Play the PPO-bot!

Thank You