An Intelligent Battle Snake

Paras Malhotra **Pablo Rodriguez**

Abstract

Simultaneously playing games against multiple opponents, whose policy is unknown a priori, showcases the AI alignment problem. An overly conservative strategy leaves points on the table, while an overly aggressive one leads to premature death. Algorithmic alternatives are put to the test in Battlesnake.

1 Introduction

1.1 The Battlesnake Game

Battlesnake is a multiplayer programming framework for developers. Controlled by a web server to build, and the code to write, Battlesnakes move to find food, avoid others, and stay alive. Battlesnake is an open-ended playground to explore multi-agent competitive games.

Snakes compete against each other for food and survival. Each player maximizes its own reward while minimizing opponent rewards. The rules of the game are simple, though the best strategies may be sophisticated. Competitors may develop heuristics to solve sequential problems, such as in the A^* algorithm.

1.2 Game Definition

A Battlesnake game consists of a number of snakes on a grid world. It is forbidden that a snake eat its own body. Eating food, a snake can restore its health, as well as grow in size. A snake eats another by hitting its head, when the shorter one dies while hitting any snake's body leads to death.

Battlesnake is specified as a Markov game, with the tuple $M = (N, S, A, T, R)$. Respectively: the number of snakes, the state space observed by all agents, the action space, the transition probability, and the reward function.

1.3 Problem Specification

Let s refer to states, a to actions, and r to rewards. Transitioning to a new state s', a piece of experience consists of (s, a, r, s').

The starting state:
s_{start} = Random locations of all competing snakes and food pellets, starting health (100) for all snakes

The available actions:
$Actions(s) = \{Up, down, left, right\}$ (avoiding collisions with other snakes or boundaries)

The successor state:
$Successor(s, a)$ = Updated locations of competing snakes and pellet locations, healths of all snakes

The cost:
$Cost(s) = 1$ health for each move and boost to max health (100) on eating food

The end state:
$IsEnd(s)$ = One or no snakes left alive. Death is caused by depleting health, collisions with other snakes/walls or if trapped (no more legal moves left)

2 Literature Review

The original Battlesnake paper publishes results, where each snake's policy is trained independently with Proximal Policy Approximation (PPO). For illustration purposes, several human engineered heuristics are identified.

- Avoid hitting the walls
- Avoid making forbidden moves
- Move towards eating food
- Favor eating or trapping competitors

They observer different performance during offline training versus online inference. HILL agents performed better than those without it.

3 Dataset

3.1 Generating Data

As in reinforcement learning, a sequence of states, actions and rewards are generated by simulating game play.

Since the Battlesnake game is coded using the programming language Go, we coded our own game simulator in Python that generates food at random locations and announces the winner for each game episode.

Using the simulator, we generated datasets that were then used for training the neural network via reinforcement learning.

When using our Minimax and Alpha Beta approach, we assumed that the opponent's policy is to minimize our agent's score:

$$\pi_{agent}(s; \mathbf{w}) = arg \max_{a \in Actions(s)} V(Succ(s, a; \mathbf{w}))$$

$$\pi_{opponent}(s; \mathbf{w}) = arg \min_{a \in Actions(s)} V(Succ(s, a; \mathbf{w}))$$

This may not always be true, especially for games with more than 2 players. In such games, the opponent's strategy is likely to be maximizing his/her own score rather than minimizing our score.

A generated episode may be: $s_{start}; a_1, r_1, s_1; a_2, r_2, s_2; ...; a_n, r_n, s_n$

3.2 Size of Dataset and Pre-Processing

Since the data is generated by self-play, there is no limit to the size of the dataset. However, we trained the model by generating 100 episodes of game play. Each episode contained 150-200 turns per player on average. On each training, we assumed a grid/map size of 11 x 11, the standard used by Battlesnake for its major tournaments and rating system. We also assumed 3 players (or 2 opponents) per game.

3.3 Transformation for Input to Convolutioanl Neural Network

Battle state was converted into 15-channel tensors representing a grid each, for input to a CIFAR10-type of network:

- Objects: food, hazards
- Players: adversary and self location of head and body, as well as health and length
- Actions: left, right, up, down

4 Baseline approach

4.1 Random Agent

To establish a performance baseline, we use the ELO rating system by Battlesnake. Battlesnake uses TrueSkill's rating system and then transforms it into an ELO rating by applying a logistic function to convert the rating to a number between 0 and 1, then multiplying by 10,000 and rounding down to the nearest integer.

The baseline is established by using a random agent - an agent that chooses a random move (among all legal moves) every time. We created such an agent and then entered into a tournament to establish the ELO. We achieved a rating of 2,020 for such a random agent, and this will act as our performance baseline, since intelligent agents are expected to achieve better performance than a random agent.

5 Main approach

5.1 Game Algorithms

We used multiple game algorithms to approach the problem and compare performance. The algorithms include the following:

- Depth Limited Minimax Search
- Depth Limited Minimax Search with Alpha-Beta Pruning
- Depth Limited Expectimax

For the evaluation function of these algorithms, we used the following hand-crafted evaluation heuristics, such as:

- Breadth-first serarch: min-distance to food
- Min-distance to enemy
- Player health

5.2 State Evaluation via Q-Learning

Off-policy, Q-Learning learns the optimal value function, on each (s,a,r,s'):

$$prediction = \hat{Q}_{opt}(s, a; \mathbf{w})$$

The value is based on an estimate of the optimal policy:
$target = r + \gamma \max_{a' \in A(s)} \hat{Q}_{opt}(s', a'; \mathbf{w})$

An error can quantify the distance between example's target and the model's prediction:

$$E = prediction - target$$

The objective function could be the square loss, for one example:
$Loss(s; \mathbf{w}) = \frac{1}{2}E^2$

$$\nabla_{\mathbf{w}} Loss(s; \mathbf{w}) = E \nabla_{\mathbf{w}} \hat{Q}_{opt}(s, a; \mathbf{w})$$

$$\mathbf{w} \leftarrow w - \eta \nabla_{\mathbf{w}} Loss(s; \mathbf{w})$$

Therefore, no knowledge of the Markov Decision Process' transitions $T(s, a, s')$ is necessary.

5.3 State Evaluation via Temporal-Difference Learning

On-policy, TD-Learning training can be comprehensive of immediate rewards r, as well as the γ discounted value of future states:

$$target = r + \gamma \hat{V}_{\pi}(s'; \mathbf{w})$$

The value is based on an exploration policy estimated from $\hat{V}_{\pi}(s; \mathbf{w})$:
$prediction_{linear} = \hat{V}_{\pi}(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$

$$E = prediction - target$$

Likewise the objective function could be the square loss:
$Loss(s; \mathbf{w}) = \frac{1}{2}E^2$

The loss' gradient, for a linear network:
$\nabla_{\mathbf{w}} Loss(s; \mathbf{w}) = E \nabla_{\mathbf{w}} \hat{V}_{\pi}(s; \mathbf{w}) = E\phi(s)$

The stochastic weights update rule, with η learning rate:
$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} Loss(s; \mathbf{w})$

Therefore, applying TD-Learning requires knowedge of the rules of the game, such as $Successor(s, a)$.

6 Performance Metric

6.1 Battlesnake TrueSkill

The BattleSnake game is scored by a transformed version of TrueSkill, originally developed by Microsoft as an alternative to ELO. Values are normalized to the range 0-1. The Rating inputs the Trueskill value, as well as the player's score mean and variance. In order to be competitive in global player rankings, a player needs to have a high ranking in every leaderboard.

6.2 Model Evaluation

The model can be evaluated through generic evaluation methods, such as K-fold cross validation. Over-fitting may be prevented via: regularization, early stopping, dropout, among many other alternatives.

7 Result & Analysis

7.1 Baseline approach performance

Our baseline approach (random agent) achieved an ELO rating of 2,020.

7.2 Discarded approaches

Because TD-Learning is an on-policy algorithm, we did not pursue it. It appeared to be not appropriate for the problem statement at hand, namely learning the optimal agent policy.

7.3 Computing constraints

Battlesnake has a timeout of 500 ms, which became on issue given the available hardware. This limited inference depth as well as timely Convolutional Neural Network predictions.

7.4 Computing complexity

State Evaluation via Q-Learning is expensive. In terms of memory, CNN input channel representation increase exponentially depending the game grid's dimensionality, to the power of the number of available agent actions.

7.5 Analysis of results

The top leaderboard position has an ELO rating of 8,000+ and a random agent baseline achieved a 2,020 rating.

Depth limited minimax search gave us an Elo rating of 3,050, thereby beating the random agent (Elo of 2,020). However, we could only achieve a max depth of 2.

By using minimax with alpha-beta pruning, we were able to increase our max depth to 3. This allowed us to reach an Elo rating of 4,400. Also, on testing against random agents using our game simulator, we were able to hit a 100% win rate (against random agent opponents) across 100 game episodes.

We couldn't benchmark the Q-learning approach because of the timeout issues. The ELO rating is a closed system owned by the Battlesnake team and so it was impossible for us to bypass the timeouts and derive an Elo rating. Inference of the CNN was taking too long especially when combined with alpha beta pruning.

8 Error Analysis

8.1 Transfer Learning

Loss function oscillations can make learning difficult for a regression model to predict the sum of the immediate reward and the optimal value of the follow-on state:

$$target = r + \gamma \max_{a' \in A(s)} \hat{Q}_{opt}(s', a'; \mathbf{w})$$

This is because the target label is not stable until learning of $\hat{Q}_{opt}$ has converged.

The approach we engineered is to train in stages:

1. Learn weights that minimize loss with $target = r$
2. Then fine-tune that model with a the full target including $\gamma \max_{a' \in A(s)} \hat{Q}_{opt}(s', a'; \mathbf{w})$

8.2 Evaluation Function Learnings

After several manual fine tuning attempts, we realized that the min distance to the enemy was not helping. This heuristic made our snake much less aggressive in competing for food pellets, and so, we ultimately concluded to remove this heuristic.

We also had a score for number of alive opponents on the board (the lower the better) to motivate the snake to move aggressively in order to trap the opponents. However, this didn't work and ultimately, we removed this factor as well. Instead, we may have tried to score moves that are likely to result in a rapid kill and we suspect that may have worked better.

9 Future Work

We can further improve performance by adding a score based on player mobility (using a flood-fill algorithm) to the evaluation function for the alpha-beta and minimax agents. This would help in avoiding situations where our snake was getting stuck by spiralling itself into corners that didn't have a way out, and ultimately dieing.

Additionally, we could incorporate an algorithm for maxN depth limited search, where each opponent tries to maximize his/her own score rather than minimizing our score. Currently, we are using that opponents try to minimize our score when in fact, most opponents will try to maximize their own score as a primary objective and possibly minimize ours as a secondary objective. However, we are not completely sure if this would improve performance given the timeout constraints because we would then have to compute and store in memory each opponent's score. So, this approach is likely to increase the response time.

10 Code

Our code is uploaded here on Github.

References

[1] Jason Brownlee. 2023. Building a Convolutional Neural Network. *Machine Learning Mastery*.

[2] Jonathan Chung, Anna Luo, Xavier Raffin, Scott Perry. 2020. Battlesnake Challenge: A Multi-agent Reinforcement Learning Playground with Human-in-the-loop. *arXiv:2007.10504*.

[3] A. L. Samuel. Some studies in machine learning using the game of checkers. *doi: 10.1147/rd.441.0206*.