Sergey Pastukhov, omikad@gmail.com April 21, 2024

Abstract

This paper introduces a novel algorithm for two-player deterministic games with perfect information, which we call PROBS (Predict Result of Beam Search). Unlike existing methods that predominantly rely on Monte Carlo Tree Search (MCTS) for decision processes, our approach leverages a simpler beam search algorithm. We evaluate the performance of our algorithm across a selection of board games, where it consistently demonstrates an increased winning ratio against baseline opponents. The results highlight the algorithm's robustness in learning and adapting to various game dynamics.

1. Introduction

In the domain of artificial intelligence, two-player board games have historically served as pivotal 'toy problems' for exploring and advancing search and planning algorithms within vast decision spaces. The outstanding algorithm AlphaZero achieved superhuman performance in the game of Go, chess, and other board games without the use of human expertise in these games. In this work, we introduce a new approach to solving such games, similar to AlphaZero, but with a key difference in how the policy network is trained. The main idea is that the algorithm iterates through possible moves using beam search, and then learns to predict the outcome of this search. This concept gives rise to the name of the algorithm, PROBS - Predict Result of Beam Search. This approach shows promising results—it demonstrates an increase in the winning percentage during the training process and shows improvement with the use of greater computational power. Although this new approach to solving board games does not improve upon state-of-the-art approaches, it demonstrates a new working concept that may inspire researchers to develop new methods in other areas.

The foundation of the PROBS algorithm is the iterative training of two neural networks. The first network is a value function, V(s), which predicts the expected utility from the current state. V(s) approximates the optimal value function V*(s), which exists for all games of perfect information and determines the outcome of the game under perfect play by all players.

The agent's action selection is modeled by a function, Q(s, a), which predicts the outcome of a min-max beam search in the game tree from state s with move a. Conducting a full traversal of the entire game tree from state s is unfeasible; therefore, the algorithm only iterates over a limited subtree and replaces the values of the leaves of this tree with V(leaf_state). A key result of this work is that the PROBS algorithm operates effectively even if this subtree is quite small, demonstrating improvement during the training process even with a searched subtree of only two moves in depth.

These two neural networks are trained iteratively through a cycle of the following steps:

- The agent plays games against itself using Q(s, a), selecting both optimal and suboptimal moves with a certain probability to ensure exploration.
- Using the games played, the agent trains the value function V(s), which predicts the expected utility of being in state s with the policy derived from applying Q(s, a).
- For each observed state from the same played games, a beam search is initiated to explore a subtree, utilizing a fixed version of Q(s, a) to prioritize the expansion of each state. When the search limit is reached, the value of the leaf states is replaced using V. The results of this exploration are used to improve the function Q(s, a).

Each iteration guides the policy represented by Q(s, a) towards one where decisions are made by exploring a small game sub-tree, and the leaves of this tree are replaced with V(s). Thus, Q(s, a) becomes a slightly deeper estimation compared to the function V(s), and V(s) in turn becomes an estimate of this new version of Q(s, a). In the subsequent iteration, Q(s, a) is trained on game sub-trees where the values of the leaves are replaced with this more accurate version of V(s). As a result, in each iteration, Q(s, a) begins to incorporate information about good moves deeper in the game sub-tree than the depth of the beam search.

The PROBS algorithm can be intuitively understood by comparing it to how chess players improve their game. Initially, the function Q(s, a) is randomly initialized, so in the very first step of the first iteration, the agent plays randomly. However, the first iteration of training V(s) can already begin to form some understanding of the game — it might recognize that material advantage leads to victory, and that checks and threats against strong pieces are also advantageous.

Then, like chess players who "calculate the best move in a position," the agent begins to ponder each of its moves. Similar to chess players, for each position, the agent initiates a search for its possible best moves and those of the opponent. As a chess player spends many hours contemplating their moves, they learn to improve the process of calculation itself — focusing more on better moves and seeing benefits even before calculating all combinations. For instance, an experienced chess player might see that a certain move leads to a favorable situation in three moves simply because they have calculated such advantageous scenarios many times and seen the benefits extend even three moves further. This experience allows the chess player to find advantages six moves ahead.

2. Related work

Outstanding success in board games was achieved by AlphaZero [4], which reached a superhuman level in the game of Go. Similar to our work, AlphaZero iteratively optimizes both V(s) and Q(s, a); however, the optimization of Q(s, a) is achieved through the use of a Monte Carlo Tree Search (MCTS) tree, the outcomes of which are used as estimates of move probabilities. These probabilities serve as the target for training Q(s, a). The more moves the agent evaluates while creating the MCTS tree, the more accurately these probabilities are estimated. In contrast, the PROBS algorithm does not evaluate move probabilities but rather assesses the outcomes of tree exploration. Consequently, in PROBS, there is no MCTS tree but a simpler beam search mechanism is used instead. In this work, we demonstrate that even traversal of an extremely small subtree allows each iteration to enhance the policy, showing that limited yet focused exploration can effectively contribute to strategy refinement in deterministic games with perfect information.

In addition to board game strategies, advancements in planning algorithms have been explored in the context of puzzle games, such as demonstrated in "Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping" [2]. This study focuses on puzzles like Sokoban, where the authors predict the entire path from the initial state to a goal state using A*. The fundamental difference between board games and puzzle games lies in the nature of the objectives. In puzzle games, the task is to solve the puzzle, and all paths that solve the puzzle are equally valid. In contrast, board games involve two players with opposing goals, making the objective to develop a policy that remains unbeaten by any player. Here, any discovered path might be dominated by another, rendering the board game scenario a moving target problem, where A* may not be applicable directly. However, A* could potentially serve as a useful operator for improving policies in board games, but this hypothesis requires further investigation in new research.

3. The PROBS algorithm

Our method uses two independent deep neural networks:

- $V_\theta(s)$, parameterized by θ, takes the raw board position s as input and produces its value. The value of a terminal state is 1, -1, or 0, reflecting a win, loss, or draw, respectively. The output of the value model is a number between -1 and 1, representing the long-term expected reward from following a policy derived from $Q_\varphi(s, a)$.
- The network $Q_\varphi(s, a)$, with parameters φ, takes the raw board position s as input and produces a vector of q-values. By applying a softmax function to the predicted q-values, we derive action probabilities, thereby enabling Q_φ to represent the policy of a trained agent. This vector of values represents the outcome of a beam search used to select an appropriate action from the state s.

The training of these two networks follows an iterative process, starting with self-play using the Q_φ model. This is followed by refining Vθ using the outcomes of the played games, and then enhancing Q_φ with beam search. Every iteration in the process acts as an improvement operator for the policy encoded by Q_φ. The following is a more detailed overview of each iteration:

- Execute a predefined number of self-play games, selecting moves based on the q-values derived from Qφ(s, a). Action probabilities are obtained by applying the softmax function to the vector of q-values, followed by the selection of a random action using these probabilities. To increase exploration, Dirichlet noise is added to the action probabilities, following the approach outlined in [4], with parameters ε = 0.25 and α tailored to each game:
  $$p_a = \exp(Q_\varphi(s, a)) / \text{sum}(\exp(Q_\varphi(s, a)))$$
  $$P(s, a) = (1 - \varepsilon)\, p_a + \varepsilon\eta_a$$
  $$\eta_a \sim \text{Dir}(\alpha)$$
- Parameters θ of the value model V_θ are optimized via gradient descent, with a loss function that computes the mean-squared error between the predicted and actual terminal rewards at the conclusion of each game episode, with each state s being drawn randomly from an experience replay.
- In each observed state s, we deploy a beam search to generate a limited sub-tree of the game, starting from s. The breadth and depth of this sub-tree, critical parameters of our model, will be discussed further. The leaf states of this sub-tree are either terminal states or the limits of tree expansion. Values for terminal leaf states are provided by the emulator, typically set to 1, 0, or -1; values at the limits of the beam search game sub-tree are estimated using V_θ. The value of any non-leaf state is the maximum of the negative values of its child nodes."
- The parameters φ of the q-value model Q_φ are optimized via gradient descent, with a loss function that computes the mean-squared error between the predicted q-values for each action a within a state, and the corresponding q-values obtained through beam search from state s upon selecting action a.
- Clear experience replay.

Algorithm 1: Beam search

```
Data:
    ● Board state s
    ● $V_\theta$ - state value estimator function
    ● $Q_\varphi$ - q-values estimator function
    ● $N_0$ - number of expanded nodes in game sub-tree
    ● $N_1$ - max depth of game sub-tree
Result: Q-values for every valid action in $s_0$
Function beamSearch(s, $V_\theta$, $Q_\varphi$, $N_0$, $N_1$)
    1. tree ⟵ empty list
    2. beam ⟵ empty priority queue
    3. tree.add( { value=∅, state=s, children=∅ } )
    4. beam.add( { priority=∞, nodeIndex=1, depth=0 } )
    5. For expand = 1,$N_0$ do
        a. If beam is empty End For
        b. priority, nodeIndex, depth ⟵ pop item with highest priority from beam
        c. value, state, children ⟵ tree[nodeIndex]
        d. actionValues ⟵ $Q_\varphi$(state)
        e. For Each action in emulator.getValidActions(state) do
            ● nextState, reward, done ⟵ emulator.step(state, action)
            ● childIndex ⟵ length of tree + 1
            ● children.add( { action=action, child=childIndex } )
            ● If done do
                ● tree.add( { value= -reward, state=nextState, children=∅ } )
            ● Else do
                ● tree.add( { value= ∅, state=nextState, children=∅ } )
                ● If depth < $N_1$ do
                    ● priority ⟵ (∞ if depth = 0 else actionValues[action])
                    ● beam.add( { priority, childIndex, depth+1 } )
    6. For i in range from length of tree to 1 do
        a. value, state, children ⟵ tree[nodeIndex]
        b. If value = ∅ do
            ● If children is empty do
                ● tree[i].value ⟵ $V_\theta$(state)
            ● Else do
                ● tree[i].value ⟵ max( -tree[child].value for child in children )
    7. QValues ⟵ (action, -tree[child].value) for (action, child) in tree[1].children
    8. Return QValues
```

Algorithm 2: PROBS - Predict Result of Beam Search

```
Data:
    ● $N_0$ - number of expanded nodes in game sub-tree for each beam search
    ● $N_1$ - max depth of game sub-tree for each beam search
    ● $N_2$ - capacity of the experience replay memory
    ● $N_3$ - number of iterations
    ● $N_4$ - number of episodes to play in each iteration
    ● $N_5$ - number of maximum turns to play in each episode
    ● $\varepsilon$ - exploration coefficient (0.25)
    ● $\alpha$ - Dirichlet noise parameter to boost exploration
Result: $Q_\varphi$ - q-values estimator function
Function PROBS($N_0$, $N_1$, $N_2$, $N_3$, $N_4$, $N_5$, $\varepsilon$, $\alpha$)
    1. Initialize experience replay memory $D_{ER}$ to capacity $N_2$
    2. Initialize value function V with random weights $\theta$
    3. Initialize q-value function Q with random weights $\varphi$
    4. For iteration = 1,$N_3$ do
        a. For episode = 1,$N_4$ do
            ● Reset environment emulator and observe initial state $s_0$
            ● Store $s_0$ in $D_{ER}$ as a beginning of a new episode
            ● For t = 1,$N_5$ do
                ● Compute action probabilities using
                  $p_a = \exp(Q_\varphi(s, a)) / \mathrm{sum}(\exp(Q_\varphi(s, a)))$
                  $P(s, a) = (1 - \varepsilon) p_a + \varepsilon\eta_a$
                  $\eta_a \sim \mathrm{Dir}(\alpha)$
                ● Draw a random valid action $a_t$ using probabilities $p_a$
                ● Execute action $a_t$ in emulator and observe reward $r_t$ and state $s_{t+1}$
                ● Store $s_{t+1}$ in $D_{ER}$
                ● End loop if environment is terminated
            ● Associate a final reward $r_T$ with all the states in the episode: ($s_i$, $\delta_i r_T$), where $\delta_T$ =
              1; $\delta_{T-1}$ = -1; $\delta_{T-2}$ = 1; $\delta_{T-3}$ = -1 and so on.
        b. Put all the observed pairs ($s_i$, $\delta_i r_T$) into dataset $D_V$
        c. For Each random minibatch from $D_V$ do
            ● Perform a gradient step on $(\delta_i r_T - V(s_i; \theta))^2$ with respect to the network V
              parameters $\theta$
        d. Initialize dataset $D_Q$ as empty
        e. For Each state $s_i$ in $D_{ER}$ do
            ● QValues ← beamSearch($s_i$, $V_\theta$, $Q_\varphi$, $N_0$, $N_1$)
            ● Put ($s_i$, QValues) into dataset $D_Q$
        f. For Each random minibatch from $D_Q$ do
            ● Perform a gradient step on $(\mathrm{QValues}[a] - Q(s_i, a; \varphi))^2$ with respect to the network
              Q parameters $\varphi$
    5. Return $Q_\varphi$
```

4. Empirical evaluation

We evaluate the PROBS algorithm on the game of Connect Four, a classic two-player deterministic game with perfect information, featuring a board size of 6x7 and a maximum of 7 actions per turn. The algorithm was compared against four distinct agents:

- Random agent, which performs any valid move at random.
- One-step lookahead agent, which analyzes all potential moves to either execute a winning move, if available, avoid immediate losing moves, or otherwise select randomly from the remaining moves
- Two-step lookahead agent that evaluates the game tree up to two moves ahead with similar decision criteria
- Three-step lookahead agent that extends this evaluation to three moves ahead, maintaining the same strategic approach.

In our experiment, the average game lasted for 19 turns, which demonstrated the effectiveness of the Three-step lookahead agent. To illustrate the learning progression of the PROBS algorithm, we evaluated each iteration checkpoint against these four agents and reported its Elo rating. Before the experiment, we determined the Elo ratings of these four agents, using the Elo approach described here, to be 1000, 1183, 1501, and 1603, respectively.

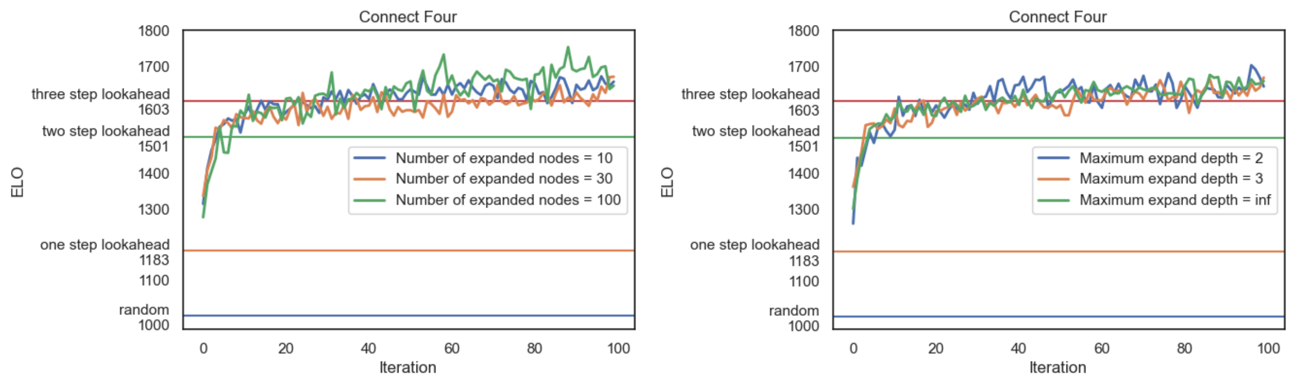

Figure 1: **a:** Training PROBS algorithm on Connect Four board game. **b:** PROBS algorithm training for a different depth limit of beam search.

Figure 1a (left) illustrates the outcome of the training process for two different models. It shows multiple lines, each representing the mean Elo rating for every iteration, aggregated over multiple parallel training runs initiated from scratch under different parameter settings. Specifically, each run encompassed 100 iterations, involving 1,000 games per iteration. The training runs varied in terms of node expansions (10, 30, or 100) and the maximum depth allowed for beam search (2, 3, or 100).

Figure 1b (right) demonstrates that the PROBS algorithm performs effectively with various depth limit values for beam search. Notably, even with beam search constrained to a maximum depth of 2, the PROBS algorithm can be trained to win significantly against a "three-step lookahead" agent (Elo rating 1603), which performs a full scan of all actions for the game sub-tree at depth 3. It is generally improbable for a player trained only up to a depth of 2 to defeat a player who performs optimally with a depth of 3 search, unless it can leverage information beyond this 3-step lookahead. This suggests that during its iterative training process, the PROBS agent learns to utilize information exceeding its beam search constraints.

References

- [4] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of go without human knowledge. nature, 550(7676), 354-359
- [1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815.
- [2] Lehnert, L., Sukhbaatar, S., Mcvay, P., Rabbat, M., & Tian, Y. (2024). Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping. arXiv preprint arXiv:2402.14083.
- [3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.
- [5] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), 484-489.
- [6] Allis, L. V. Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis, University of Limburg, Maastricht, The Netherlands (1994).
- [7] van den Herik, H., Uiterwijk, J. W. & van Rijswijck, J. Games solved: Now and in the future. Artificial Intelligence 134, 277–311 (2002)
- [8] Elo, Arpad E. (August 1967). "The Proposed USCF Rating System, Its Development, Theory, and Applications" (PDF). Chess Life. XXII (8): 242–247.
- [9] Krizhevsky, A., Sutskever, I. & Hinton, G. ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems, 1097–1105 (2012).
- [10] Mnih, V. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).
- [11] Lowerre, Bruce T. (1976). The Harpy Speech Recognition System (PDF) (PhD). Carnegie Mellon University.
- [12] LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. Nature 521, 436–444 (2015).
- [13] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning 8, 229–256 (1992).
- [14] Sutton, R., McAllester, D., Singh, S. & Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In Advances in Neural Information Processing Systems, 1057–1063 (2000).
- [15] Mandziuk, J. Computational intelligence in mind games. In Challenges for Computational Intelligence, 407–442 (2007).
- [16] Luckhart, Carol and Keki B. Irani. “An Algorithmic Solution of N-Person Games.” AAAI Conference on Artificial Intelligence (1986).
- [17] Minimax algorithm Russell, Stuart J.; Norvig, Peter. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Hoboken: Pearson. pp. 149–150. ISBN 9780134610993. LCCN 20190474.
- [18] Gemp, Ian M. et al. “Sample-based Approximation of Nash in Large Many-Player Games via Gradient Descent.” Adaptive Agents and Multi-Agent Systems (2021).