ASEN 6519 Final Project REPORT

Nathan Foote

TOTAL POINTS

(not graded)

QUESTION 1

1 Final Project Report

- 0 pts Correct

1 Final Project Report

- 0 pts Correct

AlphaZero (Almost) Applied to a Beautiful Game

Nathan Foote
Ann and H.J. Smead Aerospace Engineering Sciences
University of Colorado at Boulder
Boulder, Colorado
nathan.foote@colorado.edu

***Abstract*—Games provide an easy, understandable way to test the capabilities of reinforcement learning algorithms. General game-playing algorithms like AlphaZero have proven to achieve super-human levels of play and have been adapted to solve real-world problems. It is important to test these algorithms on many different environments to understand their limits and behavior. Tak, a self-proclaimed "beautiful game," offers an appealing testing environment. It offers a range of complexity through permitting varying board sizes and, like Chess and Go, has simple rules but a complex strategy. In this paper, a framework for creating an AlphaZero agent to learn and play the game of Tak was created.**

***Index Terms*—OpenAI, AlphaZero, neural networks, Monte Carlo Tree Search, Tak**

I. Introduction

Formulating problems as games have been a ubiquitous way to test and push the limits of reinforcement and machine learning. Games are quantifiable, understandable, and offer clear comparisons between the capabilities of humans and algorithms. Board games, like Chess and Go, are of particular interest as they are almost universally popular. Chess and Go also have the benefit of being simple to represent in terms of their states and actions, but the best strategies are very complex. However, there has been a series of successes in implementing machine learning techniques to achieve super-human levels of play from computer agents. In particular, the AlphaZero algorithm has proven effective at mastering these games [1]. To continue to explore the capabilities of AlphaZero, this paper attempts to create a framework for the board game Tak that will be used to train an AlphaZero Agent.

II. Background and Related Work

A. Foundation

Alan Turing built the foundations of computers playing games when he attempted to create a chess-playing algorithm that implemented alpha-beta pruning mini-max tree search [2]. The alpha-beta tree search would go on to become the standard framework for game-playing programs.

The previous state-of-the-art chess-playing programs before AlphaZero were represented by the likes of Deep Blue and Stockfish [3] [4]. Both programs implemented mini-max search and alpha-beta pruning and used handcrafted parameters and carefully tuned weights derived from human experience to achieve super-human levels of play.

B. AlphaZero

AlphaZero was built upon the framework of AlphaGo to be an expert, general game-playing algorithm without the need for personalized or game-dependent tuning. AlphaZero is fundamentally an implementation of a neural network to replace the rollout policy of a Monte Carlo Tree Search (MCTS). In normal MCTS, a search tree is constructed of nodes representing state-action pairs that hold statistics used to inform decision-making [5].

A node in the search tree has the following representation:

- N(s, a) - Visit Count
- W(s, a) - Total Action Value
- Q(s, a) - Mean Action Value
- P(s, a) - Prior Probability (Prior Value)

An MCTS algorithm has the following format:

Algorithm 1 Basic Monte Carlo Search Algorithm

```
procedure MonteCarloSearch(simulations)
    Starting from a root node
    for i ← 1 to simulations do
        Select the best child node from the current node
        Expand the selected node if it is a leaf node
        Estimate the value of the leaf node
        Backpropagate the statistics up the tree
    end for
    return action of child with highest visit count
end procedure
```

In AlphaZero, a trained neural network is used to evaluate the value of the leaf node instead of a rollout policy. The network is trained in two steps. First, the network generates data from self-play. The data for each game contains the game state, the MCTS probability distribution over the possible action, and the outcome of the game. Data from many games are collected and used to train the network.

AlphaZero uses a single network with two heads: a policy head and a value head. The network takes in the current state of the game and outputs a probability distribution over the total action space and a value estimating the value of the state.

$$(p, v) = f_\theta(s) \tag{1}$$

The network is trained using the following loss function:

$$l = (z - v)^2 - \pi^T \log p + ||\theta||^2 \tag{2}$$

Where the loss minimizes the value loss, which is the difference between the network's estimated outcome (v) and the actual outcome (z), and maximizes the similarity of the network's policy estimate (p) and the search probabilities from MCTS (π). Also, c is a parameter that controls L2 weight regularization to prevent over-fitting.

One other important aspect of AlphaZero is that its MCTS algorithm implements a version of the upper confidence bound (UCB) called probabilistic upper confidence bound (PUCB), which achieves a smaller regret than normal UCB [6]. More detail on the variant of the PUCB implantation will be discussed in the Solution Approach section.

AlphaZero achieved a super-human level of play in Chess and Go. The AlphaZero structure has also been used to solve real-world problems. AlphaDev used an AlphaZero implementation to find new state-of-the-art sorting algorithms where writing assembly code was treated as a game [7]. Continuing to test the limits and capabilities of AlphaZero's general game-playing ability, this paper aims to implement an AlphaZero agent on the game Tak.

C. Tak

Tak is a board game created by fantasy author Patrick Rothfuss and game designer James Ernest. In Tak, players take turns either placing a stone or moving a stack of stones on the board. The goal of the game is to create a "road" between two adjacent sides of the board. The details of the rules are discussed in more depth in the Problem Formulation section. Notably, though, Tak can be played with different board sizes ranging from 3x3 to 8x8. An example of a road is shown below in Fig. 1.

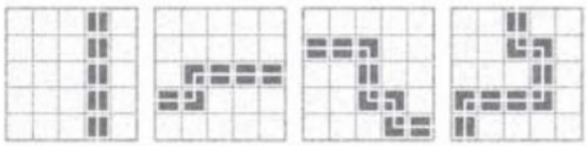

Fig. 1. Examples of Winning Roads

III. Problem Formulation

Tak can be formulated as a perfect information, alternating Markov game. For this paper, a custom OpenAI gymnasium environment was created to represent Tak.

A. State Space

The state of the game is represented as a dictionary containing the current player and board. The player is represented as a +1 or -1 for Players 1 and 2, respectively. The board is an N by N array of empty lists where each list represents the stack of stones in each space. An empty space can be thought of as a stack of size 0. A single stone in a space is a stack of 1, and so on. Each stone is represented as a positive integer. Odd numbers represent Player 1's stones, and even numbers represent Player 2's stones, as shown in Table I.

TABLE I
Stone Representation

Player 1 Stones	Player 2 Stones
1: Denotes a flat stone	2: Denotes a flat stone
3: Denotes a standing stone	4: Denotes a standing stone
5: Denotes a Capstone	6: Denotes a Capstone

There are three types of stones the player can interact with. First are "flat stones," which are used to construct the player's road and can be stacked on by any other piece. Second, "standing stones" block the path of a road for both players and cannot be stacked upon. Lastly, "capstones" are unique stones analogous to a queen in chess. The capstone can "flatten" a standing stone if it moves into its space, and cannot be stacked on. Integers are inserted into the front of the list to represent stacking a stone into the space. A rendering of a Tak game is shown in Fig. 2. Focusing on the space in the second row, third column, the list that represents the stack of stones would be: [3,1,1,2,1].

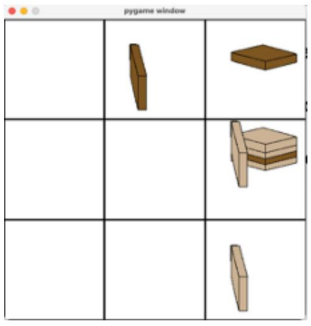

Fig. 2. Examples of a Tak Game

B. Action Space

There are two types of actions in Tak: place actions and move actions. The number of possible place actions is equal to the number of spaces on the board times the number of unique stone types. For example, a 3x3 board only has flat and standing stones, so the total number of place actions is 18. For a 5x5 board, the capstone is added to the game, so there are a total of 75 place actions.

Move actions are more complex because of the rules placed on them. First, there is a limit to the maximum number of stones that can be moved, which is dependent on the size of the board. A 3x3 game only allows for the player to move a maximum of 3 stones, a 5x5 game allows 5 stones, and so on. A move can only occur in one direction: left, right, up, or down. Lastly, the player must leave at least one stone in each space moved in, effectively always leaving a trail of stones when moving a stack.

The number of stones moved can be used to classify move actions. "Move 1" actions contain every possible action that can be taken when moving one stone. "Move 2" and "Move 3" contain every possible action for moving 2 and 3 stones, respectively. As the board space grows, more "Move X" actions need to be added to account for the greater number of

stones moved and the possible trails that correspond to each action. For the 3x3 board explored in this paper, Tak has a total of 126 possible actions. Table II details the total action space for different sizes of Tak as well as for Chess and Go.

TABLE II
ACTION SPACES OF TAK, CHESS, AND GO

Game	Action Space
Tak 3x3	126
Tak 4x4	480
Tak 5x5	1,575
Tak 6x6	4,572
Tak 8x8	32,704
Chess	4,672
Go	11,259

In the environment, the action is represented as two different tuples for each type of action. For a place action, the tuple contains the row and column where the stone will be placed and the type of stone that is to be placed. Notably, the actions are taken from Player 1's perspective; that is, the stone types are represented as odd integers. If Player 2 chooses a place action, then 1 is added to the stone type to represent the corresponding stone type for Player 2.

For move actions, the tuple contains the row and column of the stack that is to be moved, the row and column of the last space in the move, and the trail that is left from the move.

Place Action Example:

(identifier, (row, column), stone type, discrete action index)

3x3 Example Place Action:

(2, (2, 0), 1, 85) = place a flat stone in row 3, column 1.

Move Action Example:

(identifier, (from row, from column), (to row, to column), (trail,), discrete action index)

3x3 Example Move Action:

(1, (0, 1), (2, 1), (2, 1), 27) = move 3 stones from row 1, column 2 to row 3, column 2. Leave 2 stones in the first space of move, and 1 stone in the second space of move.

C. Rewards

To match the architecture used in AlphaZero, the reward is +1 if the current player wins, -1 if the current player loses, and 0 for a tie. This is important for the implementation of the AlphaZero network into the MCTS algorithm, which will be discussed in the Solution Approach section. The reward is collected when the game reaches a terminal state. If a player connects two adjacent sides of the board with a road, then that player wins, and the game ends. If a road is not created, the game will end if the board is filled or one of the players runs out of stones. In either case, the player that controls the most flat stones at the top of each stack wins.

IV. SOLUTION APPROACH

A few constraints were placed on the environment due to limited time and hardware capability. First, only a 3x3 board was evaluated to reduce the state and action space. Second, there is a special rule in Tak that each player places their opponent's first piece. This was avoided to move the project along but should be implemented in the future.

With a completed Tak environment, three Tak-playing agents were created. First, a Random Play agent that takes a random valid action given the state of the board. Second is an MCTS agent that uses a simple epsilon greedy rollout policy to estimate the value of leaf nodes and serves as the structure of the AlphaZero implementation. Finally, the AlphaTak agent that uses an AlphaZero-like network to replace the rollout estimation of the MCTS agent.

A. Random Play Agent

The Random Play agent takes a valid action given the current board state. To do this, the total action space is evaluated to determine what actions are possible given the following rules:

- A stone cannot be placed into an occupied space.
- A player cannot move stones from an empty space.
- A player can only move stones from stacks they control.
- A player cannot move more stones than are in a stack.
- A player cannot move stones past a standing stone or a capstone.

B. MCTS Agent

The MCTS agent implements an MCTS with PUCB using the POMDPs package for Julia [8].

The POMDPs solver function shown in Fig.3 uses a simulate function to run MCTS as defined in *Algorithms for Decision Making* [9].

```
function solve(solver::MCTS_Solver, env)

    # Initialize the root node of the tree
    root = MCTSNode(
        env.state,          # The state from the environment
        nothing,            #  No parent, as this is the root
        Vector{MCTSNode}(), # Initially, no children
        0,                  # No visits yet
        0.0,                # Initial value (could be zero or based on a heuristic)
        0.0,                # Mean value
        0.0,                # Prior value
        nothing,            # No inital action
        1                   # Root node at level 1
    )

    for _ in 1:solver.simulations
        scratch_env = env.clone()
        simulate!(
            root,
            scratch_env,
            solver.exploration_constant_init,
            solver.exploration_constant_base
        )
    end

    best_child = root.children[1]

    for child in root.children
        if child.visit_count > best_child.visit_count
            best_child = child
        end
    end
    return best_child.action
end
```

Fig. 3. MCTS solve function in Julia

To discuss the implementation of PUCB, the best action is selected according to the child node that has the highest visit count. The action that maximized the mean action value Q(s, a) plus a value U(s, a) is used to determine which node is visited when stepping through the search tree. U(s, a) is calculated using PUCB as defined as:

$$U(s,a) = C(s)P(s,a)\sqrt{N(s)}/(1+N(s,a)) \qquad (3)$$

where C(s) is the exploration rate defined as $C(s) = \log((1 + N(s) + c_{init})/c_{base})$ where c_{init} and c_{base} are exploration constants, P(s, a) is the prior probability of selecting action a in state s, N(s) is the parent visit count, and N(s, a) is the node visit count.

The hyperparameters for the MCTS agent are slightly different from AlphaZero's implementation. The MCTS solver runs for 50 simulations instead of 800 and has a depth of 20 instead of 512. These changes were implemented due to the smaller board size and limited hardware running the environment. Otherwise, the exploration parameters c_{init} and c_{base} are the same as AlphaZero: c_{init} = 1.25, c_{base} = 19,652.

C. AlphaTak

The AlphaTak agent uses the same structure as the MCTS agent, except a neural network is used to evaluate the value of the leaf nodes. The neural network for AlphaTak is similar to AlphaZero's structure in that the body of the network is composed of layers of rectified batch-normalized convolutional layers followed by skip connections that output to two heads. The policy head outputs a probability distribution over the total action space, and the value head outputs a value estimate of the given state. The input to the network is an array that represents the state of the board. The 3x3 board is represented as a 91-element array where the first element represents whose turn it is, and the rest of the array represents a possible stack of 10 stones for each space. When the network produces a probability distribution over the total action space, invalid actions have their probabilities set to 0, and the distribution is re-normalized over the remaining valid actions. Theoretically, incorporating this action masking into the training data will teach the network to learn what actions are valid given the game state. One aspect of future work will be to evaluate and refine the neural net structure to find what the appropriate size and configuration to use in Tak would be.

V. Results

Unfortunately, the AlphaTak agent was unable to be trained. Therefore, only the Random Play and MCTS agents were able to be evaluated.

A. Random vs Random

First, 100 games were played were both Player 1 and Player 2 played according to the Random Play policy. The results are shown in Fig. 4. After 100 games, it appears that Random Play results in roughly equal chances to win, lose, or draw, which makes intuitive sense.

B. MCTS vs Random

Next, the MCTS agent was evaluated by playing 100 games against the random agent. Notably, the MCTS agent played 50 games as Player 1 and 50 games as Player 2 to mitigate any player-sided bias. The results of those games are shown in Fig. 5. These games show the MCTS agent provides a significant improvement over the Random Play agent. However, an almost 50% win rate is not an overwhelming success. The MCTS agent would be more compelling if it won a majority of games played against the Random Play agent.

C. MCTS with 100 simulations

The lack of overwhelming success over Random Play is likely due to the MCTS simulations being too low. The number of simulations for the MCTS agent was set to 50, which was the highest number that could play 100 games in a reasonable time frame (less than 10 minutes). In an attempt to verify if this is indeed the case, another 100 games of MCTS vs Random Play were conducted, but with MCTS performing 100 simulations. The results shown in Fig. 6 indicate some improvement at larger simulations. Refining the MCTS agent to win convincingly will be left for future work.

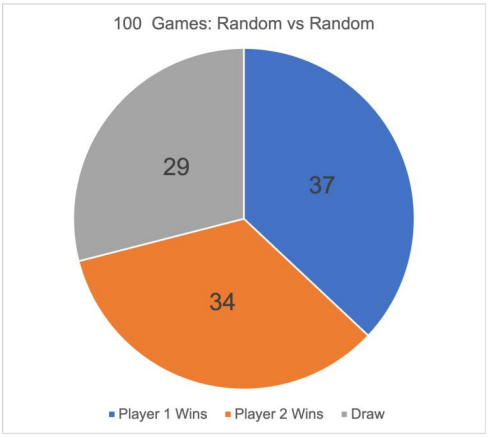

Fig. 4. 100 Games: Random vs Random

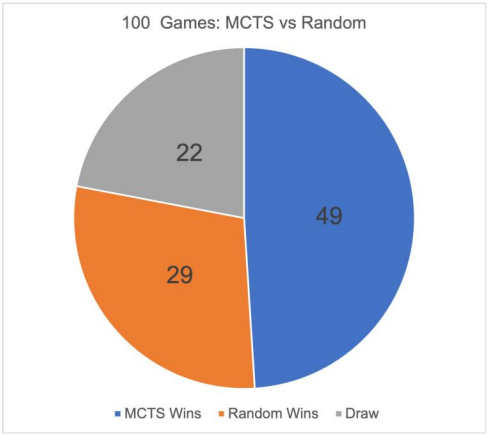

Fig. 5. 100 Games: MCTS vs Random

D. MCTS vs MCTS

Some interesting insights into the behavior of the MCTS agent are seen when it is played against itself, as shown in Fig. 7. First, the number of draws is dramatically reduced when compared to Random Play. This might imply that it is much more difficult to force a draw in Tak when both players are playing to win. Second, it appears that there is an advantage to being Player 2 when both players are playing to win.

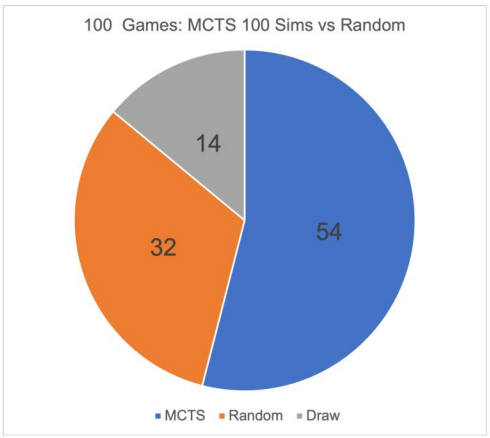

Fig. 6. 100 Games: MCTS w/ 100 sims vs Random

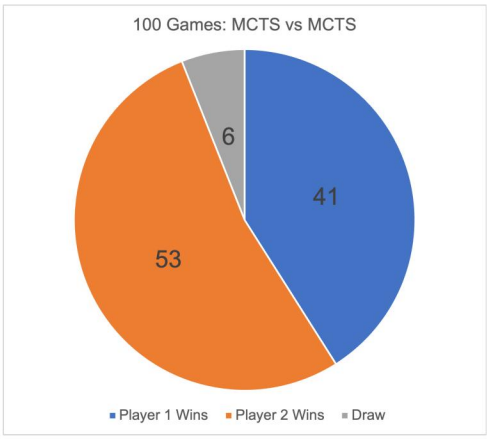

Fig. 7. 100 Games: MCTS vs MCTS

VI. Conclusion and Future Work

While, unfortunately, the AlphaTak agent was unable to be trained, the framework for proceeding with generating self-play data and training the agent is ready to be implemented. The MCTS agent already shows promising improvement over the Random Play agent, which indicates the implementation of a trained network in place of the rollout policy would also exhibit some learned behavior. The following steps for this implementation are detailed below.

A. Optimize the environment

While the Tak environment follows the rules of Tak and accommodates a range of board sizes, there are many opportunities for optimization. Some of the functions that take the longest time to execute are the ones that determine what actions are valid. This is because the functions enumerate through each action to determine if it violates the rules of the game. This could be improved by implementing a cache of valid actions given a board state to avoid having to enumerate through actions that have been checked already.

B. First action implementation

As stated previously, the rule that the player places the opponent's first piece was separate from this implementation. However, the current framework may be able to handle this rule. Starting with the empty board as the root node, Player 1's first turn would be the action corresponding to the child with the smallest visit count. Enacting that action on the environment would create the state that is the root node for Player 2 to do the same, after which the agent would then select actions corresponding to the most visited child node.

C. Agent Optimization and Network Training

The MCTS agent should be optimized until its number of simulations and MCTS depth are as close to AlphaZero's implementation as possible. Much of this work can probably be done by optimizing the Tak environment, but there is undoubtedly room to improve in both the MCTS and AlphaTak agents. Also, AlphaZero adds Dirichlet noise to the prior probability of the root node during training that is yet to be added.

After these optimizations, the AlphaTak agent can be trained in the same manner as AlphaZero and evaluated against the other agents and real players.

VII. Contributions and Release

Most of the work conducted was creating, testing, and verifying the Tak environment. This is because while existing Tak environments are available, they are outdated in that they do not follow the format OpenAI gym requires, or they are wrong. Therefore, the Tak environment created here was created using the general environment template provided by OpenAI Gym and was created from scratch. The MCTS implementation in this paper follows the MCTS algorithm as described by Algorithms for Decision Making and was modified to incorporate AlphaZero's PUCB implementation. Nathan was responsible for all aspects of the project, including the code and the paper that was written. The author grants permission for the report to be posted publicly.

References

[1] Hongming Zhang and Tianyang Yu. Alphazero. *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 391–415, 2020.
[2] George W Atkinson. *Chess and machine intuition*. Intellect Books, 1998.
[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.
[4] https://stockfishchess.org/.
[5] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.
[6] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.
[7] Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.
[8] https://github.com/JuliaPOMDP/POMDPs.jl.
[9] Mykel J Kochenderfer, Tim A Wheeler, and Kyle H Wray. *Algorithms for decision making*. MIT press, 2022.

VIII. Appendix

The code and game data for this paper can be found at: https://github.com/nathanjfoote/footeTak.git