5-2022

# Playing Cassino with Reinforcement Learning

Edmund Yong

# Abstract

Reinforcement learning algorithms have been used to create game-playing agents for various games—mostly, deterministic games such as chess, shogi, and Go. This study used Deep-Q reinforcement learning to create an agent that plays a non-deterministic card game, Cassino. This agent's performance was compared against the performance of a Cassino mobile app. Results showed that the trained models did not perform well and had trouble training around build actions which are important in Cassino. Future research could experiment with other reinforcement learning algorithms to see if they are better at training around build actions.

# Contents

# 1 Introduction

Over time, researchers have developed AI-based agents that play computer games against human opponents with varying degrees of skill. While some agents base their actions on pre-defined rules, others use machine learning techniques such as reinforcement learning to achieve better results through emulated games and reward calculations. Reinforcement learning has been used repeatedly to train applications to play deterministic games: games where a set input always yields a set output. One such application, AlphaZero, learned to play complex games such as chess, shogi, and Go at a championship level through self-play after only being taught these games' basic rules. (Silver, Hubert and Schrittwieser)

In contrast with most previous studies, this study used reinforcement learning techniques to create an agent for a non-deterministic game, Cassino. Cassino is a fishing card game that originated in Italy and spread to English speaking countries. Due to the game's modest popularity, digital apps for playing Cassino are largely unavailable. Cassino has multiple rulesets and optional rules, which can alter the game's optimal strategy, and allows for multiple types of actions to be taken based on a player's hand and cards currently on the table.

Deep-Q Learning was used to train a model to play Cassino. This model's performance was compared against the performance of a Cassino mobile app. Results showed that the trained model did not perform well and had trouble training around build actions which are important in Cassino. Further studies that experiment with other reinforcement learning algorithms are needed to improve the model's performance.

# 2 Literature Review

## 2.1 Reinforcement Learning with Deterministic Games

Prior studies on the use of reinforcement learning to train agents for playing deterministic games includes work by Finnsson and Björnsson (2008), Yang (2009), Zhu and Kaneko (2018), and Silver et al. (2018).

A 2008 paper by Finnsson and Björnsson describes experiments with a General Game Playing (GGP) agent that uses tree search and Monte-Carlo (MC) simulations to select moves (Finnsson and Björnsson). A GGP agent is an artificial intelligence (AI) that can learn to play multiple games at a high level without game-specific knowledge from programmers. Monte-Carlo methods are a class of computational algorithms used for estimating value functions, finding optimal policies, and simulating mathematical systems using repeated random sampling. These methods are usually implemented as computer simulations since they depend on random numbers and repeated computations.

GGP research is promoted by the Association for the Advancement of Artificial Intelligence (AAAI). Since 2005, the AAAI has sponsored an annual GGP competition. Agents that enter the competition play matches by connecting to a game server. The server gives agents a set start-clock, which limits the time that an agent can analyze a game's description, and a play-clock, which limits the time an agent has to make each move. Games are specified in Game Description Language (GDL), a language that can accurately describe different games with varying players. For an agent to compete in the competition, it must have an HTTP server to interact with the competition's game server, an understanding of GDL, and an AI that can play the games it gets.

Prior to Finnsson and Björnsson's study, most GGP agents used game-tree searching with a heuristic evaluation function to choose moves. Finnsson and Björnsson's agent, CADIAPLAYER, used their Upper Confidence-bounds applied to Trees (UCT) algorithm to eliminate the need for heuristic evaluation. UCT, which builds a game tree in memory, is a variant of the Upper Confidence Bounds (UCB) algorithm, which combines policy gradient ascent with approximate policy prediction. UCT's tree

records the result of a game's actions, together with the current game state and the number of times each action has been used at the current state. If action being recorded hasn't been used at a given state, UCT defaults to that action to increase tree exploration. Each time an action that hasn't been simulated is used, all parts of the tree in memory above the current state are deleted to limit the program's use of memory.

Unlike other high-level game-playing programs that use game-specific features to evaluate moves, GGP agents use a smaller set of generic features that can apply to multiple games. This use of generic features can prevent agents from learning key game mechanics. This, in turn, can lead to inaccurate heuristic evaluations or cause agents to play towards the wrong objectives.

Finnsson and Björnsson tested CADIAPLAYER by entering it in the 2007 GGP Competition, which the program subsequently won. This event required agents to play a large number of matches in 40 different types of games. For single play games, CADIAPLAYER used a modified version of the Memory Enhanced IDA* search algorithm. When the play-clock begins, this algorithm immediately searches for a satisfactory next move. If the algorithm finds such a move, the program continues to use Memory Enhanced IDA* to search for better moves until the play-clock expires. Otherwise, CADIAPLAYER switches to the UCT algorithm. For multiplayer games, CADIAPLAYER uses MC and the UCT algorithm for decision making. For two-player games, the agent can either maximize the difference between the players' score or maximize its own score. For games with three or more players, CADIAPLAYER only considers its own score for simplicity. To improve CADIAPLAYER's performance, the UCT algorithm also creates separate game-tree models for opponents.

During the 2007 GGP competition, Finnsson and Björnsson noticed that CADIAPLAYER sometimes played too optimistically when choosing between actions that hadn't been explored in simulations. The authors found that CADIAPLAYER made a random choice among unexplored actions when it couldn't identify an option with a higher chance of success. The agent would eventually identify a better choice, but only after running more simulations. Finnsson and Björnsson subsequently improved their agent by causing it to track the effectiveness of its actions over multiple game states; this improvement exploited the fact that actions that are good in one game state are often good in others.

Finnsson and Björnsson ran three further experiments that matched different variants of their agent against each other. Data was gathered from 250-game matches between pairs of agents while alternating turn order between games. Each match had a start-clock and play-clock of 30 seconds and was run on a Linux based dual processor computer with each agent using a single core. The authors chose to use Connect-4, Checkers, Othello, and Breakthrough for their experiments.

In their first experiment, the authors compared the performance of their main agent that used UCT with two variants: one that used MC and a uniform random distribution to choose the action with the highest average success rate, and a second that also built a game tree. The UCT agent outperformed the other two agents in all four tested games with a win rate ranging from 77% to over 91%. In the second experiment, which tested the tracking of moves across multiple games, the improved CADIAPLAYER showed better results in all four games. Finnsson and Björnsson's third experiment featured a competition between two identical agents in which one agent was allowed twice as much thinking time as the other. In this last experiment, the agent with more thinking time won every match with no apparent decrease in performance.

In 2009, Yang et al. experimented with creating an adaptive game AI for Dead End (Yang, Gao and He). Dead End is a two-dimensional game that places 2 AI dogs and a player-controlled cat in predetermined locations on a grid with a single exit. The player tries to either move the cat to the exit or to avoid both dogs for 20 simulation steps, while the dogs try to catch the cat. During a simulation step, the cat and each dog must move one space up, down, left, or right. Yang et al. treated the player's reaching the exit as a win, the dogs' catching the player as a loss, and the player's avoiding the dogs for 20 simulation steps as a draw. To balance gameplay, the player can move at twice the speed of the dogs; hence, the dogs must work together to catch the player.

Initially, Yang et al. used MC methods to program their AI. At the time when their work was published, game AIs were usually programmed using scripted rules. Yang et al. sought to use Monte-Carlo methods to improve their AI's adaptability, in order to increase the difficulty of playing against AI-controlled opponents. To apply MC to Dead End, Yang et al. created a search tree with all legal moves for

one dog at each time step (Figure 1). Their algorithm then ran a simulation for the search tree's four parent nodes, each of which represented a dog's possible moves. After the simulations for each parent node were successfully run, each parent node recorded its results and the program chose the move with the highest chance of winning.

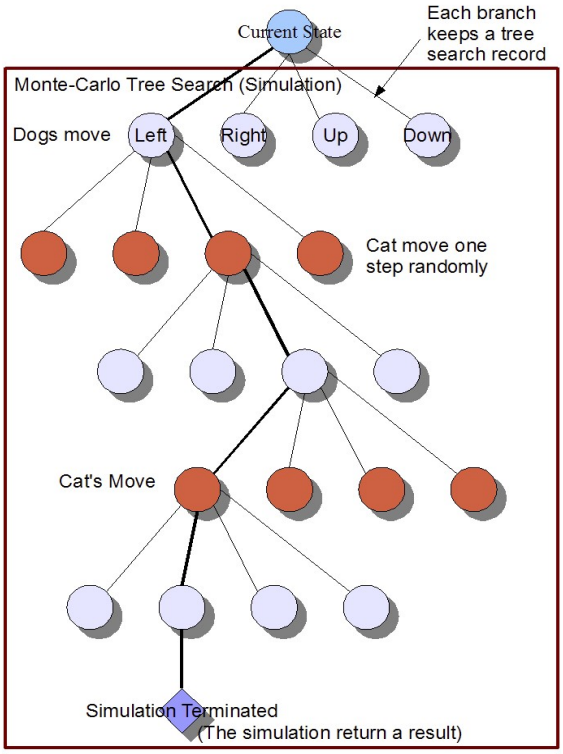

Figure 1. Dead End Game Tree (Yang et al.)

For the experiment, each set of dogs competed against a cat that used one of three different strategies. The experiment limited computations to 500ms to avoid boring human players. The performance of Yang et al.'s implementation improved as it was allotted more computation time, up to an unspecified threshold value. After this value, its performance decreased then stabilized.

Subsequently, Yang et al. switched from MC-based models to artificial-neural-network- (ANN-) based models in order to increase the rate at which their AI moved. The authors determined that the ANN-based models could be trained to emulate their MC model's performance in less time. For all games played, each ANN saved all moves by the player and the dogs, then discarded all data from games the player won. The remaining data was separated into two groups, to be used to train the ANNs and to check their weights.

The authors found that the ANN-based AIs, while more efficient and faster than MC-based AIs, couldn't effectively compete against strategies that the AIs hadn't been trained to counter. The authors' findings were obtained through an unspecified testing procedure that involved building 1000 threads running Dead Game. Yang et al. concluded an adaptive game AI should use both models: an ANN model to play against known strategies and an MC-based model to play against unknown strategies.

A 2018 paper by Zhu and Kaneko discusses the use of different loss functions to train deep neural networks (DNNs) to play shogi (Zhu and Kaneko). Evaluation functions for training DNNs use loss functions to select optimal moves; this is crucial when training programs to play games at a high level. Shogi poses a more difficult training problem than (e.g.) chess, due to its larger board size and rules that allow players to use pieces captured from an opponent and to place them almost anywhere on the board. Many high-level shogi programs such as Elmo, the previous world champion of computer shogi, used linear-based models as evaluation functions because those models require less computation to train.

Programmers started experimenting with using DNNs to play shogi after AlphaGo used DNNs to play Go, a less complex game than shogi. AlphaZero eventually defeated Elmo at the World Computer Shogi Championship, winning 90 of the 100 games they played.

Zhu and Kaneko observed that DNN-based game playing programs like AlphaZero could still be improved. Their loss functions at the time of their work had only applied cross entropy for training. To assess the effectiveness of shogi-playing DNNs that were trained with other loss functions, Zhu and Kaneko combined two different loss functions with the function used in Bonanza, the first shogi program to play competitively against a top-level player. Zhu and Kaneko's functions minimized the error of Bonanza's loss function, which mainly focused on win prediction, while also minimizing the evaluation function's temporal difference error function.

The authors compared the results from their modified loss function with Bonanza's original function. To train the DNNs with the experiment's functions, Zhu and Kaneko used a dataset that consisted of winning players' moves from 868,161 games of shogi, obtained from a computer shogi server. 85% of this data was used to train the DNNs and the remaining 15% was used to test the DNNs' performance. The data was shuffled before it was used since moves played in the same game have a strong correlation with one other. The DNNs initialized their weights by first training an initial network using an independent dataset and random weights chosen with a normal distribution. After 500,000 unspecified time steps, the main DNNs used the weights from the trained network. After this, the DNNs were first updated after 600,000 time steps, then at intervals of 200,000 time steps.

Zhu and Kaneko evaluated the functions' performance by checking how often the trained program's move was better than its opponent's move, and by comparing the trained program's move against all other legal moves. Zhu and Kaneko found that the combined loss function was more accurate than Bonanza's function in the earlier stages of learning and achieved similar values to the base function towards the later stages of learning.

In 2018, Silver et al. introduced AlphaZero, a generic version of AlphaGo Zero, and tested its performance with chess, shogi, and Go (Silver, Hubert and Schrittwieser). AlphaGo was the first program to play games such as chess or shogi without the help of game-specific features or alpha-beta search. Like AlphaGo, AlphaZero chose moves using a DNN that was trained through self-play and reinforcement learning. AlphaZero's DNN was initially given random values for its parameters; these parameters were then updated through self-play games created with MC tree search (MCTS). The MCTS algorithm chose moves based on whether they had been used before. After each game, the DNN updated its parameters to minimize the error between its predicted and the actual outcomes.

While their algorithms are similar, AlphaZero differs from AlphaGo Zero in several key ways. AlphaGo Zero, which was created for Go, exploited the lack of draws in Go by optimizing DNNs by the probability of winning. AlphaZero on the other hand estimates and optimizes its predicted and actual outcome since games such as chess and shogi can result in a draw.

AlphaGo Zero and AlphaZero also differ by how they train their DNNs. AlphaGo Zero generated self-play games from the current best DNN, which would be replaced if a new DNN outperformed it with a win rate of 55%. By contrast, AlphaZero generated and updated a single DNN. AlphaGo Zero exploited Go's inherent positional symmetry, rotating and reflecting data for each position to model eight possible board configurations. AlphaGo Zero also randomly rotated or reflected board positions during MCTS before evaluation and averaged the results between them. Since different sides are important in other games such as chess and shogi, AlphaZero can't exploit symmetries like AlphaGo Zero.

Silver et al. trained three different instances of the program: one each to play against top-level programs in chess, shogi, and Go. The authors used 5000 first-generation TPUs to generate self-play

games and 16 second-generation TPUs to train the DNNs. The different instances of AlphaZero were trained for 9 hours in chess, 12 hours in shogi, and 13 days in Go. Each program was run on the hardware for which it was designed with 3 hours per game and 15 seconds per move.

In chess, AlphaZero outperformed Stockfish, the 2016 Top Chess Engine Championship season 9 world champion, after 4 hours of training. After being fully trained, AlphaZero won 155 out of 1000 games against Stockfish, losing 6 . The authors also tested AlphaZero with extra matches against Stockfish using commonly used chess openings and AlphaZero won all matches convincingly.

In shogi, AlphaZero outperformed Elmo, the 2017 Computer Shogi Association world champion, after 2 hours of training. AlphaZero had a 98.2% win rate when playing as black and a 91.2% win rate overall.

In Go, after 30 hours of training, AlphaZero outperformed AlphaGo Lee, an older version of AlphaGo that defeated Lee Sedol, the 18-time world champion of Go. When playing against AlphaGo Zero, AlphaZero had a win rate of 61%—even with AlphaGo Zero taking advantage of game specific features for eight times more data.

Silver et al. found from the matches against Stockfish and Elmo that AlphaZero searched 60,000 positions per second in chess and shogi, while Stockfish searched 60 million and Elmo searched 25 million. This shows that AlphaZero could choose moves better during training. Through additional testing, the authors found that AlphaZero could defeat Stockfish while using $1/10^{th}$ as much thinking time and also had a win rate of 46% against Elmo when given $1/100^{th}$ as much thinking time.

## 2.2 Reinforcement Learning with Non-Deterministic Games

Prior studies on the use of reinforcement learning to train agents for playing deterministic games includes work by Yen et al. (2015), Mealing and Shapiro (2015), and Hsueh et al. (2018).

In (Yen, Chou and Chen), Yen et al. discuss the use of nondeterministic Monte-Carlo tree search (NMCTS) to generate moves in Chinese Dark Chess (CD-Chess). CD-Chess is a two-player zero-sum stochastic game that is played on a 4×8 board. Each player controls a set of 16 red or black pieces,

consisting of a king, two guards, two ministers, two rooks, two knights, two cannons, and five pawns.

At the start of a game, all pieces are placed face-down at random positions on the board. The first player turns over a face-down piece to determine the color that he or she controls. The players then take turns either revealing a face-down piece or moving a piece they control. Every piece can move left, right, up, or down. Pieces can either move into empty spaces or into spaces occupied by opposing pieces that those pieces may capture. In order to capture an opponent's piece, a piece must outrank its opponent's piece. The one exception to this rule involves cannons: a cannon can capture any piece that is separated from it by a third piece in the same row or column. A player wins when the opponent has no legal moves or captures all of the opponent's pieces. A game is considered a draw if neither player captures or reveals a piece within 40 turns.

Due to CDC's complexity, most CD-Chess programs before the authors' research used alpha-beta search (ABS) to select moves. Yen, Chou, and Chen proposed using NMCTS to avoid how CD-Chess's large branching factor limits ABS's exploration of possible moves. NMCTS is a variant of MCTS that uses deterministic state nodes to represent moving actions and nondeterministic nodes to represent revealing actions. Each nondeterministic state node represents possible piece types for face-down pieces and subsequent move sequences, using a visit count, win count, and subtree (Figure 2). When NMCTS finds a nondeterministic state node, it selects an inner node using roulette wheel selection and chooses that node's best child node. If the selected node hasn't yet been visited, NMCTS runs simulations using the chosen node; otherwise, it chooses a different child node.

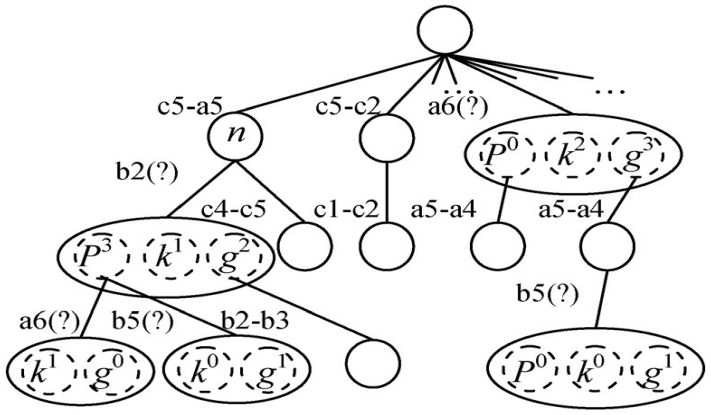

Figure 2. NMCTS tree with inner nodes (Yen et al.)

The authors simulated three different strategies for playing CD-Chess. One, capture-first (CF), captures an opponent's piece when possible and randomly selects a legal action otherwise. The second,

capture strongest piece first (CSPF), extends CF by assigning each capture move a weight based on possible pieces to capture. The third, capture and escape strongest piece first (CESPF), extends CSPF by accounting for and avoiding opponent's possible capture moves.

Yen et al. tested NMCTS with their CD-Chess program, Diablo, by conducting multiple experiments that involved playing 400 games and running 10,000 simulations per move against ABS-based programs. The simulations were run on an Intel based computer using a single CPU core. After each simulation, their program updated the nondeterministic state node's win count and visit count. The win count was incremented by 1 if the authors' strategy won and 0.5 if it played to a draw. To decrease the number of draws, the authors gave a bonus to a node's win count based on a simulated game's length; this bonus served to decrease the number of inefficient moves. The bonus was calculated by subtracting the simulation's duration from the average length of 500 simulations run before starting NMCTS and multiplying the result by a constant $d$ to control the bonus's influence on the win count.

In the first experiment, the authors tested Diablo with various values for $d$ to observe how $d$ affected its win rate. Diablo's performance always improved when using a win rate bonus and the average game length decreased as $d$ was increased. Diablo's performance proved optimal $d$ when 0.01, which increased its win rate by 8.5%; higher values caused Diablo to focus on shortening game length and ignoring the game's result.

In the second experiment, the authors compared different variants of Diablo that used CF, CSPF, and CESPF. CESPF performed the best of the approaches with a loss rate of 4.75%.

In the third experiment, Yen et al. determined that increasing the simulations per move improved Diablo's win, up to 60,000 simulations per move. Running more than 60,000 simulations per move greatly increased the computation time for each move and gave little benefit.

The authors also tested NMCTS's performance by entering five computer CD-Chess tournaments with Diablo: TCGA 2011, TCGA 2012, TAAI 2011, TAAI 2012, and Computer Olympiad 2013. Throughout all five tournaments, Diablo won 33 games, lost 8 games and ended 15 games with a draw. DarkKnight, another NMCTS Chinese Dark Chess program, also earned good results, winning TCGA

2013 and Computer Olympiad 2013. These results showed that CD-Chess programs using NMCTS can play on par with or better than other advanced CD-Chess programs.

Yen et al. tested NMCTS's commercial potential by having Diablo play over 50,000 games a day against randomly chosen players on UNIQUE CHINESE DARK CHESS, a popular Internet Chinese Dark Chess web server. For this experiment, the authors limited Diablo's performance to motivate human players to continue playing against it. For example, Diablo was limited to using 1 second of computation time for each move and only ran 10,000 simulations per move. Even with these limitations, Diablo could still compete with human players of various skill levels.

In 2017, Mealing and Shapiro used expectation maximization and sequence prediction to create an agent for simplified variants of poker (Mealing and Shapiro). Unlike Go, chess, and backgammon, poker requires players to account for hidden information, including an opponents' cards and strategy for playing them: e.g., bluffing and changes in strategies over time. To address these needs, the authors' agent infers hidden game information using expectation-maximization; anticipates an opponent's strategy with sequence prediction, which infers an opponent's actions from past interactions with the opponent in similar situations from past games; and creates a model of an opponent to play against in between actual games.

The expectation-maximization algorithm can infer a game's hidden information based on an opponent's actions. The algorithm first creates a function that returns likelihood estimates for the expectation of the log-likelihood of unobserved variables. It then iteratively maximizes this function and uses the updated parameters to create a new function.

Mealing and Shapiro's agent was trained using a variant of counterfactual regret minimization (CFR). CFR generates a strategy through self-play in two-player, zero-sum, imperfect information games: it repeatedly plays itself, then analyzes the result, finding ways to improve its strategy. The authors eventually chose to use outcome-sampling MC counterfactual regret minimization (OS-MCCFR) since CFR is too computationally costly to be used online. Unlike CFR which produces exact calculations, OS-MCCFR produces estimates, which reduces computation and allows for its application to larger games.

To determine whether their agent's performance would improve if it played against a model of the opponent between games, Mealing and Shapiro played their agent against three other agents: one that used CFRX, a variant of CFR that runs for X iterations; one that used the UCB algorithm; and one that used OS-MCCFR without opponent modelling. Each agent played 100,000 games against the authors' agent in each of two different variants of poker: Dice-Roll Poker and Rhode Island Hold'em.

Dice-Roll Poker uses dice instead of cards. Initially, each player pays a required bet of 1 chip into the pot. Each player then rolls a six-sided die and hides the result from other players. After all players have rolled their die, betting starts. In the first round of betting, each player can give up a chance to win the pot (fold), match an opponent's current bet (call), or raise to exceed an opponent's bet by a specified amount. If all bets are equal, a player can also pass their turn (check). If no players fold, each player then rolls their second die, hiding the result from other players before the second betting round starts. If no one folds during the second betting round, the player with the highest dice sum wins the chips in the pot.

Rhode Island Hold'em uses a standard 52 card deck. Its betting rounds are identical to Dice-Roll Poker. Initially, each player pays a required bet of 5 chips into the pot. Then, each player is dealt 1 card from a standard 52 card deck that is hidden from other players. After all players have been dealt a card, the first betting round begins. If no players fold, a "flop" card is dealt that is visible to all players and a second betting round begins. If no players fold, a second "flop" card is dealt and a final betting round begins. If no players fold during the third round, the player with the best three-card hand wins the pot.

Mealing and Shapiro tested four strategies for modeling opponents: one without using expectation-maximization or sequence prediction (UN), one that used expectation-maximization (EM), one that used sequence prediction (SP), and one that used both (EM+SP). EM outperformed UN in all cases; this shows that predicting hidden game information with expectation-maximization improves performance. Results also showed that SP outperformed UN and EM in all cases; this shows that using sequence prediction to anticipate an opponent's changing strategy improves performance. Finally, EM+SP outperformed other strategies in all cases.

In 2018, Hsueh, Wu, Chen, and Hsu experimented with the AlphaZero algorithm to determine

whether AlphaZero could learn to play non-deterministic games optimally.

AlphaZero is a generalization of AlphaGo Zero, an algorithm plays Go using deep neural networks (DNNs). AlphaGo Zero's DNNs are trained using a three-step system. The first step uses MCTS to create self-play games and runs 1,600 simulations with each move. With each simulation, the algorithm traverses the tree to a leaf, expands the tree from the chosen leaf, then updates the tree's weights based on the odds of winning. After running all simulations, it chooses a move based on the number of times a root's child nodes were visited during the simulations. In the second step, the algorithm randomly chooses different positions from the last 500,000 self-play games and uses those to optimize the DNNs. The optimized DNN is then tested by playing 400 games against the current best DNN. The optimized DNN becomes the new best DNN if its win rate is greater than 55%. In the third step, the algorithm assesses whether the self-play games it created were either equal in quality to past games or better.

AlphaZero generalizes AlphaGo Zero by removing the third step that verifies self-play game quality and considering draws as well as wins and losses. AlphaZero also selects values for parameters differently, with one parameter's value scaling based on the number of legal moves in a typical position in the game. The algorithm was tested in other games in addition to Go like shogi and chess; it played at the same level as AlphaGo Zero in the 3 games.

While AlphaZero had shown success in deterministic games, the algorithm hadn't been applied to non-deterministic games. Hsueh et al. tested the algorithm's performance using Chinese Dark Chess (CD-Chess). Hsueh et al. ran four experiments, with each experiment focusing on one of three AlphaZero parameters that control the algorithm's level of exploration with MCTS. Due to the complexity of 4×8 CD-Chess, Hsueh et al. conducted their experiments using 2×4 CD-Chess, which has a smaller board size and fewer pieces. Hsueh et al. also used lookup tables instead of DNNs for the experiments due to 2×4 CD-Chess's smaller number of game states. The experiments were conducted by setting test values for the chosen parameter to test. Default values were set for each of the remaining parameters to create self-play games and 800 simulations were run per move. After each iteration, the current lookup table played 10,000 games against the optimal lookup table to see how each parameter affected how well the program

learned to play 2×4 CD-Chess.

From the experiments, Hsueh et al. found that AlphaZero learned optimal play in 2×4 CD-Chess while maintaining a win rate close to 50% against optimal play as long as the parameters did not use extreme values.

# 3 Methodology

## 3.1 Cassino

Cassino is a lesser known two-player game that uses a standard 52 card deck. The goal of Cassino is to score 21 points by capturing cards. Initially, each player is dealt 4 cards that are hidden from the other player. Four cards are then placed face up in the center. Players take turns either capturing cards or placing a card from their hand in the center. Players capture cards by either pairing a card in their hand with a card in the center or playing a card whose value is the sum of multiple cards in the center. The 2♠ and each ace are worth 1 point and the 10♦ is worth 2 points. The player with the most spades is awarded 1 point and the player with the most overall cards is awarded 3 points.

Cassino allows players to capture cards by assembling build and call combinations. For a build combination, players can combine a card in their hand with a card on the field if the cards' sum equals a card in that player's hand. When two cards are combined, they become equivalent to their combined sum. For a call combination, players can combine a card in their hand with a card on the field if the cards have equal value and the player's hand contains another card of equal value. After the two cards are combined, their value remains the same. In addition to these rules, Cassino also has other optional rules that can be included in gameplay. Due to the large number of different rulesets, the optimal strategy may not always be obvious.

## 3.2 RLCard

For this study, RLCard 1.0.4 (Zha, Lai and Huang) was used to implement an application for playing Cassino as well as for training an AI opponent to use for testing. RLCard is an open-source toolkit for reinforcement learning research in card games. The toolkit allows users to implement reinforcement learning algorithms using Python and PyTorch. RLCard includes multiple card game environments such as Blackjack, Texas Hold'em, Mahjong, and Uno and allows users to add new card environments. RLCard also provides several baseline algorithms for training such as Deep Monte-Carlo

(DMC), Deep-Q Learning (DQN), Neural Fictitious Self-Play (NFSP), and Counterfactual Regret Minimization (CFR).

## 3.3 Limitations

The device used for this study was a laptop with an Nvidia 1050Ti 4G and a 6-core Intel i7-8750H. This device was not suitable for training opponents to play games with high action spaces such as Cassino. When RLCard attempted to train on Cassino using this device, training lasted for approximately 4 months without pause and never completed.

## 3.4 Limit Cassino

Since better hardware was not available for this study, a modified version of Cassino, referred to as Limit Cassino, was used to reduce the game's action space. One major change made to Limit Cassino removes all cards with a rank ranging from 7-10, lowering the deck size from 52 to 36. This change reduces training times by reducing the number of potential build actions. A side-effect of this change is that the 10♦ was removed from the deck, which reduced the number of potential points to earn in a round.

Another change that was made to improve training times was the abstraction of the action space. This change ignored cards in the action, save for the card from the player's hand, due to the hand card's status as the most important part of the action. For example, if a player were to attempt to capture 3♠ and a 3♣ with a 6♦, RLCard would recognize this as a capture action with the 6♦ and determine what cards to capture based on table.

## 3.5 Training

To ensure that reducing the number of earnable points per round doesn't greatly affect the result of the study, two models were trained; a base model which plays Limit Cassino as previously described, and another model which plays an altered variant of Limit Cassino that awards 2 points for capturing the 6♦. The model that plays the altered variant of Limit Cassino will be referred to as the 6♦ model. Out of the baseline reinforcement learning algorithms included with RLCard, Deep-Q Learning (DQN) (Mnih, Kavukcuoglu and Silver) was chosen for this study. The default model and the 6♦ model were trained

with DQN for 10000 episodes. Every 100 episodes, their performance would be evaluated with 4000 self-play games. A reward of 1 was returned for each win and a penalty of -1 for each loss. The values were then averaged to get the models' cumulative reward. (Figure 3) shows the cumulative reward over time for both models, with a timestep being a single turn in a game.

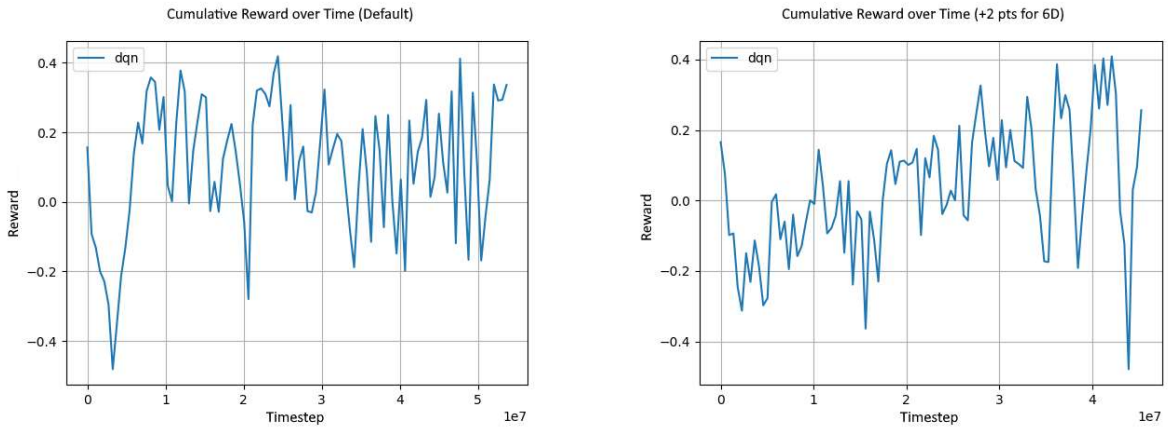


Figure 3. DQN Model's cumulative reward over time

## 3.6 Testing Procedure

The performance of each model was evaluated by comparing them to the performance of a Cassino mobile application (Dokken). To gauge their performance, test games were played against both DQN models and the Cassino mobile app. The win rates of each opponent would then be compared to assess the DQN models' capabilities.

Since Limit Cassino is a modification to Cassino created for this study, the deck size in the mobile app could not be adjusted to match RLCard. Since this study sought to create a high-level AI opponent for Cassino, the change in deck size should not impact the study's final results. The Cassino mobile app was set to the highest difficulty and optional settings were adjusted to match the DQN models' action rules. The rules that were set for the Cassino mobile app can be found in Appendix A.

# 4 Results

50 games were played against each RLCard model and their win rates were compared to the win rate of the mobile app over 50 games (Figure 4). The model that trained and played on default Limit Cassino rules had a 2% win rate over all 50 games, gaining approximately 7.22 points per game. The 6♦ model on the other hand did not win any games but had a higher average point gain per game of 9.47. The mobile app had a win rate of 82%, gaining approximately 23.54 points per game.

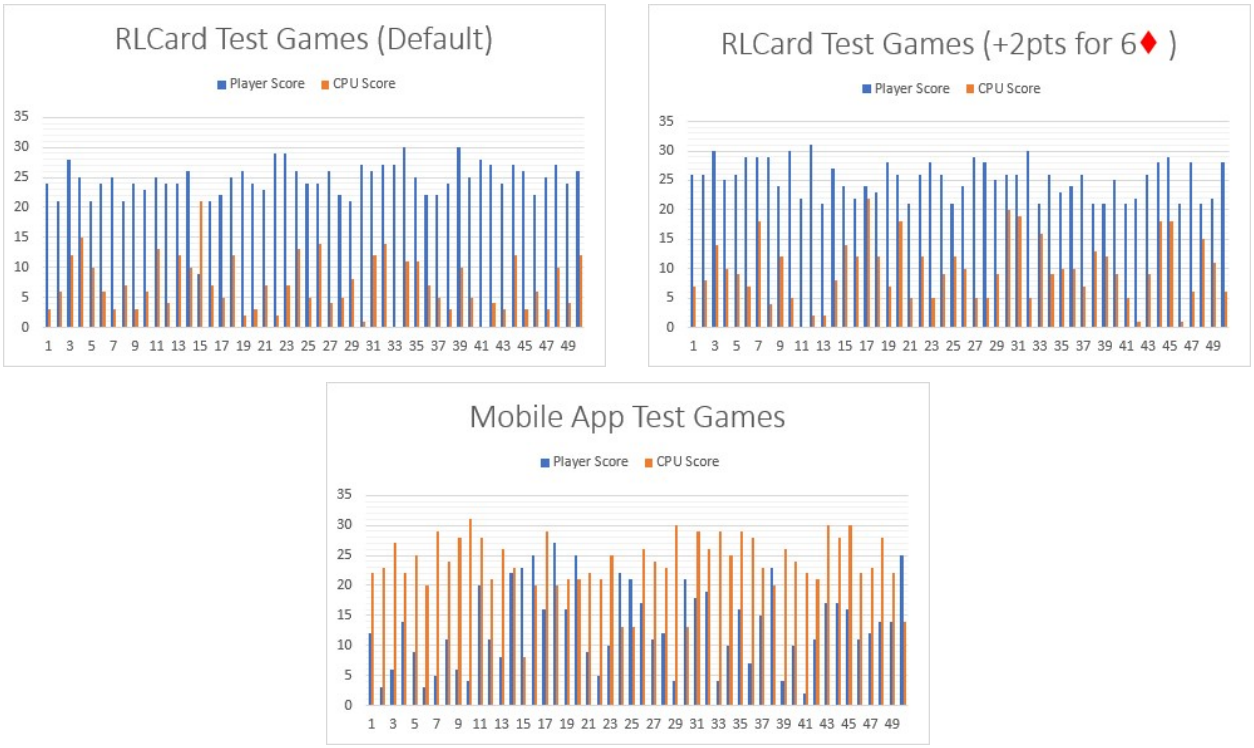


Figure 4. A summary of the test games against each AI opponents. Tables with all scoring data can be found in Appendix B.

# 5 Conclusions

In test games between the DQN models and the Cassino mobile app, the model which used default Limit Cassino rules had a win rate of 2%. During these test games, the model's gameplay regularly avoided using build actions. This implies that DQN had issues with training around build actions. This could be a reason for the model's low win rate since build actions are important in Cassino. The 6♦ model had a lower win rate of 0% and shared the same issues as the default model but was able to earn more points per game on average. The better point average per game shows that the model was able to adjust its gameplan to prioritize the 6♦. Future studies could experiment with other reinforcement learning algorithms such as DMC, NFSP, and CFR to see if they are better at training around build actions. Another potential path for future research could be to adjust parameters for DQN since in (Hsueh, Wu and Chen), AlphaZero was could not learn Chinese Dark Chess when extreme values were used for parameters.

# References


Dokken, Michael. *Cassino! (2.3.3) [Mobile App]*. 2021. App Store. <https://apps.apple.com/us/app/cassino/id527702079>.

Finnsson, Hilmar and Yngvi Björnsson. "Simulation-based approach to general game playing." *23rd national conference on Artificial intelligence (AAAI'08)*. AAAI Press, 2008. 259-264. <https://dl.acm.org/doi/10.5555/1619995.1620038>.

Hsueh, Chu-Hsuan, et al. "AlphaZero for a Non-Deterministic Game." *Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. Taichung, Taiwan: IEEE Computing Society, 2018. 116-121. <https://doi-ieeecomputersociety-org.iris.etsu.edu:3443/10.1109/TAAI.2018.00034>.

Mealing, Richard and Johnathan L. Shapiro. "Opponent Modeling by Expectation–Maximization and Sequence Prediction in Simplified Poker." IEEE Computing Society, 2017. 11-24. <https://doi-ieeecomputersociety-org.iris.etsu.edu:3443/10.1109/TCIAIG.2015.2491611>.

Mnih, Volodymyr, et al. "Playing Atari with Deep Reinforcement Learning." 2013. *arXiv.* <https://arxiv.org/abs/1312.5602>.

Silver, David, et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play." *Science* 7 December 2018: 1140-1144. <https://science.sciencemag.org/content/362/6419/1140/tab-article-info>.

Yang, Jiajian, et al. "To Create Intelligent Adaptive Game Opponent by Using Monte-Carlo for Tree Search." *International Conference on Computing, Networking and Communications (ICNC)*. Tianjian, China: IEEE Computer Society, 2013. 603-607. <https://doi-ieeecomputersociety-org.iris.etsu.edu:3443/10.1109/ICNC.2009.710>.

Yen, Shi-Jim, et al. "Design and Implementation of Chinese Dark Chess Programs." *IEEE Transactions on Computational Intelligence and AI in Games* 7.1 (2015): 66-74. <https://www-computer-org.iris.etsu.edu:3443/csdl/journal/ci/2015/01/06826513/13rRUwvT9jC>.

Zha, Daochen, et al. "RLCard: A Toolkit for Reinforcement Learning in Card Games." *AAAI Conference on Artificial Intelligence*. 2020. <https://arxiv.org/abs/1910.04376>.

Zhu, Hanhua and Tomoyuki Kaneko. "Comparison of Loss Functions for Training of Deep Neural Networks in Shogi." *Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. Taichung, Taiwan: IEEE Computing Society, 2018. 18-23. <https://www-computer-org.iris.etsu.edu:3443/csdl/proceedings-article/taai/2018/122900a018/17D45WK5ArU>.

# Appendix A – iOS App Information

## Table A1 – iOS App Settings

| Game Settings | |
|---|---|
| Players | 2 |
| Face Cards | Off |
| Aces | Off |
| 2 & 10 | Off |
| Captures | On |
| Builds | Off |
| Trails | Off |
| Build to Anything | Off |
| Draw | Off |
| Sweep | On |
| Difficulty | Insane |

# Appendix B – Test Results

## Table B1 – Deep-Q Learning Model Test Games

| Game Number | Player Score | Opponent Score |
|---|---|---|
| 1 | 24 | 3 |
| 2 | 21 | 6 |
| 3 | 28 | 12 |
| 4 | 25 | 15 |
| 5 | 21 | 10 |
| 6 | 24 | 6 |
| 7 | 25 | 3 |
| 8 | 21 | 7 |
| 9 | 24 | 3 |
| 10 | 23 | 6 |
| 11 | 25 | 13 |
| 12 | 24 | 4 |
| 13 | 24 | 12 |
| 14 | 26 | 10 |
| 15 | 9 | 21 |
| 16 | 21 | 7 |
| 17 | 22 | 5 |
| 18 | 25 | 12 |
| 19 | 26 | 2 |
| 20 | 24 | 3 |
| 21 | 23 | 7 |

| | | |
|---|---|---|
| 22 | 29 | 2 |
| 23 | 29 | 7 |
| 24 | 26 | 13 |
| 25 | 24 | 5 |
| 26 | 24 | 14 |
| 27 | 26 | 4 |
| 28 | 22 | 5 |
| 29 | 21 | 8 |
| 30 | 27 | 1 |
| 31 | 26 | 12 |
| 32 | 27 | 14 |
| 33 | 27 | 0 |
| 34 | 30 | 11 |
| 35 | 25 | 11 |
| 36 | 22 | 7 |
| 37 | 22 | 5 |
| 38 | 24 | 3 |
| 39 | 30 | 10 |
| 40 | 25 | 5 |
| 41 | 28 | 0 |
| 42 | 27 | 4 |
| 43 | 24 | 3 |
| 44 | 27 | 12 |
| 45 | 26 | 3 |
| 46 | 22 | 6 |
| 47 | 25 | 3 |
| 48 | 27 | 10 |
| 49 | 24 | 4 |
| 50 | 26 | 12 |

## Table B2 – iOS App Test Games

| Game Number | Player Score | Opponent Score |
|---|---|---|
| 1 | 12 | 22 |
| 2 | 3 | 23 |
| 3 | 6 | 27 |
| 4 | 14 | 22 |
| 5 | 9 | 25 |
| 6 | 3 | 20 |
| 7 | 5 | 29 |
| 8 | 11 | 24 |
| 9 | 6 | 28 |
| 10 | 4 | 31 |
| 11 | 20 | 28 |
| 12 | 11 | 21 |
| 13 | 8 | 26 |

| | | |
|---|---|---|
| 14 | 22 | 23 |
| 15 | 23 | 8 |
| 16 | 25 | 20 |
| 17 | 16 | 29 |
| 18 | 27 | 20 |
| 19 | 16 | 21 |
| 20 | 25 | 21 |
| 21 | 9 | 22 |
| 22 | 5 | 21 |
| 23 | 10 | 25 |
| 24 | 22 | 13 |
| 25 | 21 | 13 |
| 26 | 17 | 26 |
| 27 | 11 | 24 |
| 28 | 12 | 23 |
| 29 | 4 | 30 |
| 30 | 21 | 13 |
| 31 | 18 | 29 |
| 32 | 19 | 26 |
| 33 | 4 | 29 |
| 34 | 10 | 25 |
| 35 | 16 | 29 |
| 36 | 7 | 28 |
| 37 | 15 | 23 |
| 38 | 23 | 20 |
| 39 | 4 | 26 |
| 40 | 10 | 24 |
| 41 | 2 | 22 |
| 42 | 11 | 21 |
| 43 | 17 | 30 |
| 44 | 17 | 28 |
| 45 | 16 | 30 |
| 46 | 11 | 22 |
| 47 | 12 | 23 |
| 48 | 14 | 28 |
| 49 | 14 | 22 |
| 50 | 25 | 14 |

Table B3 – Deep-Q Learning Model Test Games (+2 points for 6D)

| Game Number | Player Score | Opponent Score |
|---|---|---|
| 1 | 26 | 7 |
| 2 | 26 | 8 |
| 3 | 30 | 14 |
| 4 | 25 | 10 |
| 5 | 26 | 9 |

| | | |
|---|---|---|
| 6 | 29 | 7 |
| 7 | 29 | 18 |
| 8 | 29 | 4 |
| 9 | 24 | 12 |
| 10 | 30 | 5 |
| 11 | 22 | 0 |
| 12 | 31 | 2 |
| 13 | 21 | 2 |
| 14 | 27 | 8 |
| 15 | 24 | 14 |
| 16 | 22 | 12 |
| 17 | 24 | 22 |
| 18 | 23 | 12 |
| 19 | 28 | 7 |
| 20 | 26 | 18 |
| 21 | 21 | 5 |
| 22 | 26 | 12 |
| 23 | 28 | 5 |
| 24 | 26 | 9 |
| 25 | 21 | 12 |
| 26 | 24 | 10 |
| 27 | 29 | 5 |
| 28 | 28 | 5 |
| 29 | 25 | 9 |
| 30 | 26 | 20 |
| 31 | 26 | 19 |
| 32 | 30 | 5 |
| 33 | 21 | 16 |
| 34 | 26 | 9 |
| 35 | 23 | 10 |
| 36 | 24 | 10 |
| 37 | 26 | 7 |
| 38 | 21 | 13 |
| 39 | 21 | 12 |
| 40 | 25 | 9 |
| 41 | 21 | 5 |
| 42 | 22 | 1 |
| 43 | 26 | 9 |
| 44 | 28 | 18 |
| 45 | 29 | 18 |
| 46 | 21 | 1 |
| 47 | 28 | 6 |
| 48 | 21 | 15 |
| 49 | 22 | 11 |
| 50 | 28 | 6 |