An investigation into Dice Chess: using adversarial and reinforcement techniques

Ayush Sundriyal, Christophe Raso, Claudia Sanchez Muñoz, Diogo Reis Rio, Dumitru Verşebeniuc, Vikram Venkat

January 2022

Contents

1 Abstract **3**

2 Introduction **3**

3 Methods **4**

4 Implementation **5**
4.1 ExpectiMax 6
4.2 Evaluation Function 6

5 Q learning **7**
5.1 Deep Q Learning 9

6 Markov Decision Process **9**

7 Experiments **10**

8 Conclusion **12**

1 Abstract

The goal of this research is to build a Dice Chess game in which a person can play against an Artificial Intelligence player that is as performing as possible. This was achieved in three steps.

The first consisted in coding the basic logic rules behind the game and implementing a Graphical User Interface that shows the chessboard. The game could be played by two human players and was tested extensively to ensure that each piece behaved properly.

The second was about creating an AI player that would make use of an evaluation function in parallel with Adversarial Search techniques to operate, such as Minimax, Alpha-Beta Pruning, Negamax and ExpectiMax. Here, a human player could play against an AI player by selecting what technique it would follow. The different algorithms were tested against a random baseline agent and against each other and it turned out that the ExpectiMax algorithm is the most performing.

Finally, the last step focused on improving the AI player by making use of several techniques of reinforcement learning, becoming in this way an hybrid agent.

2 Introduction

Since the mid 1950's, a branch of Computer Science has been devoted to imitate human intelligence. It started by challenging the machines to play against human players in one of the most puzzling and complex games: Chess. Among the alternative versions of this game, we can find Dice Chess, whose rules integrate "chance and randomness" into the game.

Computationally speaking, human intelligence can be abstracted by three tasks performed in a specific order. These tasks are grouped in what is called the "Sense/Think/Act cycle". The agent detects or is informed about the state of the environment, it "senses", in this case the position of the different pieces in the chessboard, it "thinks" by making a decision based on the current state that will ensure the best outcome, in other words deciding the optimal movement for a piece given the current state of the game. And it "acts" by putting such the decision into place by moving the piece to the evaluated position.

With this paper, we will illustrate our work on creating an engine that

is capable of "reasoning" during a game of Dice Chess. The goal of this research is to fill a gap in the current literature. Indeed, plenty of research was performed about AI playing Classical Chess, but not that much has been made for this variant of the game. The main difference is that Classical Chess is a fully observable and deterministic game, whereas each movement in Dice Chess is stochastically determined by a "dice roll" that gives the piece to be moved in a turn.

With this research, we wanted to answer the following questions:

1.Does a ExpectiMaxMinimax agent or a Hybrid Agent perform better with respect to the specific variation of parameters used?

2. Does our AI learn better through repeated exploration or through probabilistic determinations?

3.Does the Use of a different evaluation function determine the winning agent given that both have the same logic?

The next section of this report will explain the theoretical methods used to create our engine. Following that, the implementation of the previous techniques will be discussed. Furthermore, a pervasive experimentation upon the different techniques to improve the performance of our agent will be taken part. Then, the results of these experiments will be collected and presented. After that, an analysis and interpretation of these results will be performed in the goal of answering the research questions. Finally, a summary of this study will be presented and the main research question will be answered.

3 Methods

Minimax is a decision rule suitable for games of perfect information and deterministic. The algorithm takes into account two opponents, the max player and the min player. The max player tries to improve its chances of winning, while the min player seeks to minimize the opportunities of the max player to win. Theoretically, the algorithm returns the "perfect play" assuming both players are rational, i.e at every turn they will try to maximize the "benefit" of the evaluated move. Thus, it will return the "highest valued" move[2], given the piece to be moved in the turn and the current state of the board.

Negamax is a variant of the minimax algorithm. The main difference is that for both players the same procedure is implemented, making player A the negative value of player B [5]and maximizing the negative value from its

best current move. Hence, the value of the move played by the opponent is subtracted to us.It engages chance nodes to obtain the expected value for a given play, though both players maximize their score and accordingly the opponent's score will be substracted fromthe other. This is highly convenient for this type of game since the same layout will be applied for both.

ExpectiMaxminimax is another version of the minimax decision rule but adapted for games that involve chance and are hence non-deterministic games[4]. It involves another type of node called "chance nodes" in which the average expected value is computed for this specific move and then compares all chance nodes (there is one per possible piece moved) and backpropagates the highest (or lowest) evaluated node.

The ExpectiMaxmax algorithm is the combination between the negamax and the expected minimax algorithms. It engages chance nodes to obtain the expected value for a given play, though both players maximize their score and accordingly the opponent's score will be the negative of the other.

Alpha beta pruning is a technique used for the minimax decision rules, it adds two bounds to every node that they must exceed or lower (depending on the player we are talking about) in order for the move to be considered desirable[4]. By this manner we are able to avoid exploring those branches that by our analysis, indicate to not be of further interest since there exist other possible moves which would render a better result from the same turn.

4 Implementation

In order for the (expected) minimax to return the best the move given our dice roll it is needed to first obtain the pieces of the same type that are able to move, following retrieve all possible moves for each "movible" piece of the latter type and "evaluate" given the specified parameters, each of the possible moves. And this process has to be repeated given every conceivable throw of the dice for three consecutive plays, the values of each simulated play are compared at every ply, backprograting only the best one, thus returning at the end the "theoretical best move". All this process is very expensive computational wise, therefore it was in our priorities throughout this phase to lighten the whole dynamics of our algorithms resulting in a more efficient performance. At any state, it is possible to retrieve (get the coordinates of the piece in the board), move (change their coordinates) or delete (when it gets killed) any piece in O(1) from the board. A hashmap was implemented,

called PieceMap, allowing to obtain all pieces by their according type in O(1), hence to get the exact piece in O(n) where n is a dynamic variable representing the number of pieces that belong to this group.

In the case that a pawn was rolled, the entire process will be converted to O(n) since we need to iterate through every pawn, n being the number of pawns there are currently on the board. In order to know which pawn we should move, retrieving all legal moves is an O(p) process, p being the total number of possible moves for every pawn, thus the final complexity to retrieve all legal possible moves for our pawns will be O(np)

4.1 ExpectiMax

As found on the slides of the Reasoning Techniques Course, Pruning is less effective, and the minimax doesn't handle chance very well, if at all. Hence, it was decided that an implementation of the expected minimax was important and necessary. It is a variation of the Minimax algorithm. While Minimax assumes that the adversary plays optimal moves, Expected Minimax doesn't. This is especially very important for the game researched, since dice chess is a game based on chance and thus a non-deterministic game. It was decided that instead of using Expected Minimax, ExpectiMax was going to be our decision rule for the AI player. It relies on the same principle as the Expected Minimax tree, the minimizer nodes are replaced by chance nodes where the expected value is computed, yet the score of the min player will be substracted to the opponent's score and viceversa, therefore making it possible to implement the same procedure for both players. Commensurate with the Expected Minimax, it takes the average of all the "values" of the nodes below it and averaging out with the number of possible dice rolls, i.e. 6, returning thus the "expected value" of the play evaluated. It is also more pessimistic when it comes to the optimal moves of the opponents, and hence, is treated as inherent randomness. Chance nodes simply assign a probability of 1 to the lowest value children node and 0 to all the other children nodes. Below is the pseudo code for the ExpectiMax algorithm.

4.2 Evaluation Function

What is the gist of an evaluation function? It retrieves the current position of a piece on the board and returns a value that indicates the sequence of possible moves rated on ranges from 'Good' to 'Bad'. For the project,

we have implemented a dynamic and static evaluation function. A static function assesses the board given only the current positions of pieces. A dynamic evaluation function evaluating the board by taking into account, for example, a potential piece move which can be under attack, or, in the range of, an enemy piece.

Claude Shannon Evaluation function: Although in chess there is no known simple and exact evaluating function f(P), and probably never will be because of the arbitrary and complicated nature of the rules of the game, it is still possible to perform an approximate evaluation of a position. The main objective behind implementing this function was the side to move relativity "values" of the pieces. Examples are a 9 for queens and 5 for rooks. On adding the numbers of pieces for the 2 sides with their respective coefficients, the side with the largest total has the better position. Claude's idea was that instead of factoring in more terms for exchanges, it was better to implement specific variations of a game, and let the computer learn from repeated simulations.

5 Q learning

One of the most investigated approaches[1] to an evaluation function is a sum of weighted parameters that are improved as a function of the game playing performance. These parameters are incrementally refined as the agent plays more games. A similar manner to the classical Reinforcement Learning, which provides for an incremental update of an evaluation function, although in this case it is represented as a table of values.[6] This table is called a Q-Value Table. For our approach, the table was composed by a pair of state and action, to represent the current state of the game and the action evaluated to be carried off, and its corresponding "probability", this number (between 0 and 1) represents the "likelihood of a good move", as the agent plays, these probabilities are updated by the weighted discounted sum of accumulated rewards that the move implied. Policy Gradient QLearning makes use of an action-value function for its policy . A popular choice for the RL tasks is to directly adjust the parameters of the policy function of to maximize the reward by taking steps which minimize the cost. These are Policy Gradient Algorithms. [3]This approach has given raise to many practical algorithms, such as actor-critic or REINFORCE. However, with PGAs there is no target feature to which be compared to estimated cost. The aim of our loss function is to maximize the probability of good actions and reduce the ones of bad ac-

tions. Nevertheless, we want this update to be periodic and thus not increase any probability to 1, since that would make the agent only to that certain action. A loss function works by minimizing hence the most suitable choice, also taking into account the numerical overflow these operations imply given that the range of results is [0,1], is using -log(P(a)) . This will be then multiplied by the discounted return for action 'a'. The first equation is as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^{\boldsymbol{\pi}},a\sim\boldsymbol{\pi}_\theta}[\nabla_\theta \log \boldsymbol{\pi}_\theta(a|s)Q^{\boldsymbol{\pi}}(s,a)]$$

note that the magnitude of the update is proportional to the learning factor gamma. the "reward" given pair of state, action, is calculated by SHANNON'S function, since it was tested it gave a better estimate for "weight" of the action. Although having a multiagent game violates the condition for Markov Decision Process for convergence[6], additionally the agent's policies can be very sensitive to their other learning partners' [3]. A solution would be to take into account all parametrized policies for all agents, and apply the gradient for each agent's policy. Another approach was introduced by Littman, he took standard Q Learning, [Watkins and Dayan, 1992], and modified it to work with Markov games. He replaced the simple update used in standard Q Learning with a mixed strategy (probabilistic) minimax. In order to learn robust policies, it was proposed to update policies considering the worst situation. Hence, we optimize the cumulative reward for the agent under the assumption the other agent acts adversarially. Thus, yielding a minimax learning objective. Since in this research we use Negamax, i.e. both players are opposed maximizers, it matches to the max Q-value given the state and the action. This Qvalue conditions on the current actions of both agents and represents the current reward plus the discounted worst case future reward from the next state s'. This results in the second equation:

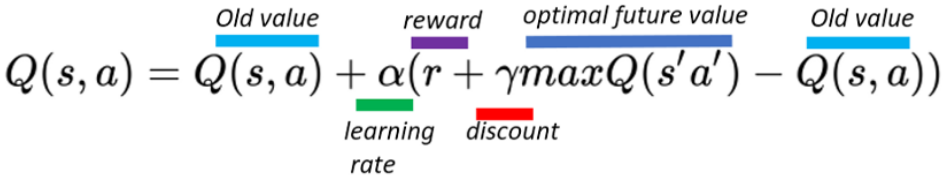
$$\begin{aligned} J_{\mathrm{M}}(\theta_i) &= \mathbb{E}_{s\sim\rho^{\boldsymbol{\mu}}}[R_i] \\ &= \min_{a^t_{j\neq i}} \mathbb{E}_{s\sim\rho^{\boldsymbol{\mu}}}\left[\sum_{t=0}^{T}\gamma^t\, r_i(s^t,a^t_1,\ldots,a^t_N)\big|_{a^t_i=\boldsymbol{\mu}(o^t_i)}\right] \\ &= \mathbb{E}_{s^0\sim\rho}\left[\min_{a^0_{j\neq i}} Q^{\boldsymbol{\mu}}_{\mathrm{M},i}(s^0,a^0_1,\ldots,a^0_N)\big|_{a^0_i=\boldsymbol{\mu}(o^0_i)}\right]. \end{aligned}$$

When our agent begun, it started with an empty table. Progressively, while playing against our agent ExpectiMax, it recorded the set of states and actions and their

corresponding probability in the table. When a given pair has never been investigated, the probability distribution of the action is uniform in order to enhance exploration, resulting in a variety of different actions. Once the game is over we update the probability of every action performed with equation 1 but Ri corresponds to the discounted sum of max rewards as stated in the equation above.

5.1 Deep Q Learning

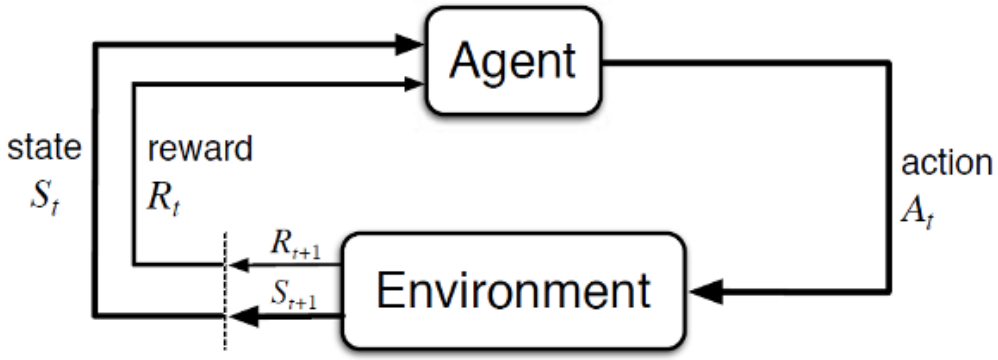
In Deep Q-learning we use a neural network to approximate the Q-value function. As input the state of the board 64 spots and 1 die roll, and the Q-values of all possible moves is generated as output. The process of DQN involves 3 steps, firstly, all past experience originated from already played games is stored in memory, afterwards the next best action is determined by the maximum output of the Q-Network and finally the network is going to update using backpropagation until convergence of optimal policy. The loss function is the mean square error between the predicted Q-value and the target Q*-value, yielding in a regression problem. Nevertheless, we do not have a target value to be compared with, thus the update equation is derived from the Bellman's equation and figure1 , being the following;



6 Markov Decision Process

Markov Decision Process contains five-tuple (S, A, R, P, G), where S - state, A -action, R - rewards, P - probabilities, and G - gamma (discount factor). The purpose of the Markov Decision Process is to find policy, that leads to long-term reward. Policies are simply a mapping of each state S to a distribution of action A. For each state S, the agent should execute an action A with a certain probability. S = Observation space: size equals 65, where

one die and board 8x8, which consists of empty spots and pieces with the type and color. A = Discrete space action: size equals 64, where each action represents the spot on the board. R = Reward system: 1) The basic reward with evaluation of the current move. 2) The ExpectiMax as an advanced evaluation function. P = Probabilities are taken from Deep Q Network G = Gamma (discount factor) equals to 0.96-0.99



7 Experiments

Purpose of experiments was to define the performance of evaluation functions and compare different AI against each other. For more accurate results, we simulated 1000 games = 500 games for the white side and 500 games for the black side

Firstly, we determined which evaluation performed better. 1) Random Agent VS ExpectiMax(with Shannon Evaluation function). Win rate of Random Agent is 5.3% and ExpectiMax(with Shannon Evaluation function) is 94.7% respectively.

2) ExpectiMax(position+threats) VS ExpectiMax(with Shannon Evaluation function) Win rate of ExpectiMax(position, position+threats) is 3.9% and ExpectiMax(with Shannon Evaluation function) is 96.1% respectively. Its learning performance increased rapidly playing against the ExpectiMax.

From results we can say that Shannon Evaluation function performs the most.

Secondly, we figure out which AI agent performs better 1) Minimax(with alpha-beta pruning and with Shannon Evaluation function) VS ExpectiMax(with Shannon Evaluation function) Win rate of Minimax(with alpha-beta pruning and with Shannon Evaluation function) is 36.9% and Expecti-

Max(with Shannon Evaluation function) is 63.1% respectively. 2) Expecti-Minimax(with Shannon Evaluation function) VS ExpectiMax(with Shannon Evaluation function) Win rate of ExpectiMinimax(with Shannon Evaluation function) is 36.7% and ExpectiMax(with Shannon Evaluation function) is 63.3% respectively From results we can say that ExpectiMax performs the most.

Thirdly, we try to figure out if the first move has the impact in the game. In this type of experiment, we simulated 1000 games without changing the side.

1) ExpectiMax(with Shannon Evaluation function) VS ExpectiMax(with Shannon Evaluation function) Win rate of white is 49.6% and black is 50.4% From the result, we can say that the first move doesn't play the crucial part in the game.

The QLearner PG agent was trained throughout 1000 games. The percentage rate for the first 300 games was of only 0.08, performing worse or equally to random baseline agent. For the next 300 games, the winning percentage increased to 0.27 and throughout the last 400 games the winning rate augmented to 0.55., exceeding our ExpectiMax agent.

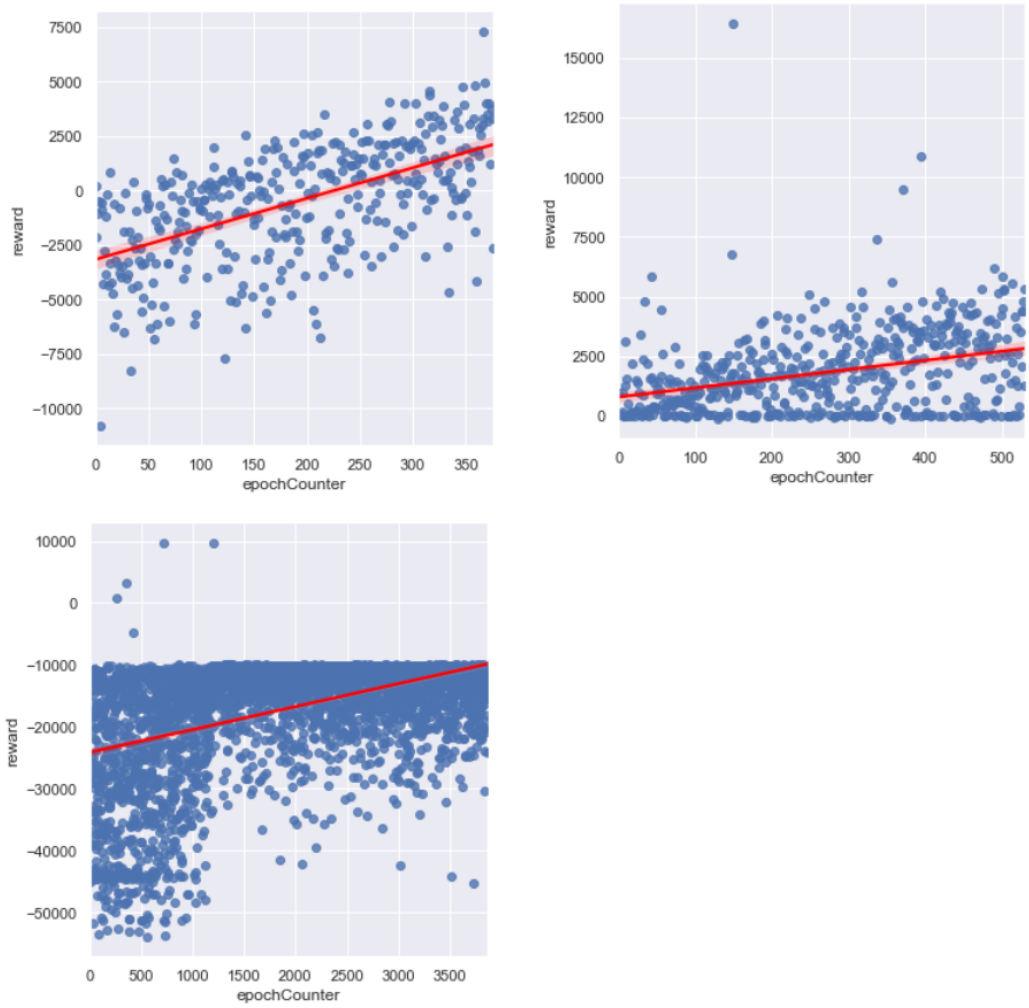


Figure 1: ExpectiMax evaluation function with move punishment (top left graph), without move punishment (top right graph) and considers only move punishment with Shannon's function (bottom left graph).

8 Conclusion

Regarding the implementation of the hybrid agent, it can be inferred that it performed better than the ExpectiMax agent, regardless of a white or black

piece move, with the data from the experiments section.

Even though DQN didn't perform with optimal moves, it learnt how to move the pieces and with further training it will acquire to learn and apply strategies, since at the moment it plays almost like a random agent that knows where to move the given piece. Nevertheless, this can and will be improved with future training, as we already saw its learning performance from the experiments. The probabilistic determination works better, but it has it's limitations and the DQN's will continue improving, making them the more rational choice.

The experiments pitted the 2 evaluation functions against each other with the same agent and it was observed that that Shannon's evaluation function performed the original every time.

Thus, we can conclude that the evaluation function plays a primary role in our agents.

References

[1] Michael Buro. A generalized linear evaluation model. 2002.

[2] Xiyu Kang, Yiqi Wang, Yanrui Hu, et al. Research on different heuristics for minimax algorithm insight from connect-4 game. *Journal of Intelligent Learning Systems and Applications*, 11(02):15, 2019.

[3] Shihui Li, Yi Wu, Xinyue Cui, Honghua Dong, Fei Fang, and Stuart Russell. Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4213–4220, 2019.

[4] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[5] Warren D Smith. Fixed point for negamaxing probability distributions on regular trees. *preprint*, 1995.

[6] William Uther and Manuela Veloso. Adversarial reinforcement learning. Technical report, Tech. rep., Carnegie Mellon University. Unpublished, 1997.