Machine Learning in Games

Harinath Reddy Bijjala
Digital Engineering
Otto von Guericke University
Magdeburg, Germany
Email: harinath.bijjala@st.ovgu.de

Khamar Uz Zama Mohammed
Data and Knowledge Engineering
Otto von Guericke University
Magdeburg, Germany
Email: khamar.mohammed@st.ovgu.de

***Abstract*—Artificial Intelligence in games is a domain that is being heavily researched for decades. Search algorithms were the first to defeat humans in popular games like chess. Recently, Deep Reinforcement Learning (DRL) techniques are used to defeat humans in complex board games like Go. These algorithms, learning only through self-play, started outperforming the search algorithms in most of the board games as well. DRL requires relatively more computation while learning. Search algorithms require relatively more computation while playing. However, most of the current research does not consider computation cost in comparison of DRL and search algorithms, to the full extent. This paper answers the question, what additional computational cost the deep reinforcement learning algorithms require, over the search algorithms. The game Shogi (Japanese chess), which is more complicated than chess, is used to compare Monte-Carlo Tree Search and Deep Q Networks. This comparison considers computation time, the number of games played during training, the final intelligence and the algorithm's setup effort.**

***Keywords*—** Artificial Intelligence, Shogi, Reinforcement Learning, Monte Carlo Tree Search, Deep Q Networks

1. Introduction

For centuries, board games like chess and go, are used as a fun tool to test human's intelligence. These classic board games have also become a medium to test Artificial Intelligence (AI) of computers against humans. Chess is the most researched game in the domain of AI, and IBM's Deep Blue [7] was the first AI to outperform human player in chess.

The board games like chess, shogi and go, are mostly used for AI research because of the following reasons:

- Perfect information.
- Zero-sum rewards.

Perfect information games means, all the information the player needs (about the position of player's pieces and opponent's pieces) is present on the board. Games like Poker restrict the knowledge about opponent's cards. So, Poker is not a perfect information game. The classic board games have zero-sum rewards. It means a positive reward for a player (when a player captures an opponent's piece), becomes a negative reward for the opponent and vice versa. This type of rewards helps in easier development of algorithms which learn through self-play.

Search algorithms go through all possible moves in games, to find the best. Search algorithms have several pruning techniques to minimize the search space and optimization techniques for faster search. For example, Cost Complexity Pruning replace an entire subtree with a leaf node with a similar value which is essential in optimizing the resources. These algorithms often include game specific human knowledge, like assigning certain weights for certain strategic piece positions in chess.

There are also several algorithms successful in playing other games, like Othello, Backgammon, Shogi and recently Go. Some algorithms use DRL techniques to achieve superhuman intelligence in games like Backgammon and Go. These algorithms combine neural networks with Reinforcement Learning (RL) algorithms. Google's DeepMind has developed AlphaGo, a supervised learning neural network algorithm, to play a board game Go, and has defeated the human world-champion in 2016. Go is highly complex, with a solution space of size 10^{170}, and it was thought to be impossible for a search algorithm to defeat humans in the game Go.

AlphaZero is a development over AlphaGo, that uses only reinforcement learning. AlphaZero has neural networks, that can be retrained to play chess, shogi and go, by keeping most of the parameters constant between games. In 2017, AlphaZero defeated the best search algorithms in chess and shogi. Elmo was a computer shogi champion then, and AlphaZero has defeated Elmo with less than 2 hours of training. Elmo is a search algorithm and has decades of human knowledge in pruning and optimizing the search. AlphaZero learns only through self-play and has used more than 1000 Tensor Processing Units (TPUs). This kind of hardware is not available for usage to most researchers and real-world applications. Also, when the computation cost of DRL algorithms is too high to achieve small efficiency gains over search algorithms, then most real-world applications will not use the DRL algorithms. The goal of this paper is to find the computation cost required by modern DRL algorithms over classic search algorithms.

This paper uses Shogi as a medium, to train, test and compare two RL algorithms: Monte-Carlo Tree Search (MCTS), as classical search algorithm and Deep Q Networks (DQN), as a modern RL algorithm. Shogi is similar to chess, with a slightly bigger board and more pieces. The board size of shogi is 9x9, and the game starts with 20 pieces per player. In shogi, when an opponent's piece is captured instead of removing it from the game, the player can bring it back to board on his side later. As a result, the number of pieces on the board does not decrease, as much as in chess, and makes shogi more complex than chess. The solution space of shogi is 10^{81}, compared to 10^{47} of chess. The following factors are considered to compare MCTS and DQN:

- The effort required in setting up these algorithms for shogi

- Number of games played while training
- Number of wins while testing
- Computation time while training and testing

In Section 2, some of the related works in this domain are described. Section 3 explains the working two RL algorithms used to compare. In Section 4, the implementation details of these algorithms, their comparative results are described. The entire summary of the work and possible future works are discussed in Section 5.

2. Related Work

Shogi is a relatively less researched game. Since DRL has gained much research recently, there is even less research that combines shogi and DRL. But, there are a few, and one of them is AlphaZero [22]. David Silver, Thomas Hubert, et al. have developed AlphaZero, to prove "algorithms gain generality, when their specificity (human knowledge) is removed." This is proved as AlphaZero achieved superhuman intelligence in three different games (Go, chess and shogi) with the same Neural Network (NN) parameters. It has only one NN, that is used for both policy evaluation and generating value function. This NN uses MCTS while training and the algorithm prove that MCTS adds stability to the training, which is done only with selfplay. AlphaZero is compared against the top search engines for shogi and chess. This comparison against search algorithms is given with hardware used and games score with Elo ratings. However, there is a vast difference in the hardware used, where AlphaZero uses 5064 TPUs for training and four TPUs for testing. Also, the training time required for search algorithms is not considered.

Wan and Kaneko have made multiple contributions to shogi research. Their works do not use DRL or selfplay learning, but deep NN with supervised training. Wan and Kaneko also address the vast computing resources used for AlphaZero [27]. They aimed to optimize the evaluation NN, so that it requires significantly fewer computation resources, that the resources used in AlphaZero. Their proposed solution uses supervised learning with deep convolutional NNs and available game records, instead of only reinforcement learning. They used two NNs, a value network and discriminator network, with uniform regularization. This solution is tested against a search algorithm called Bonanza. Bonanza uses mini-max search algorithm developed for shogi and is the World Computer Shogi Champion in 2006 and 2013. Their proposed solution outperformed Bonanza by 11% in shogi. They achieved this by training their NNs using two NVIDIA 1080Ti GPUs, and with a batch size of 128 samples. Wan and Kaneko's another contribution to shogi is using imitation learning [26]. Imitation learning is a supervised learning algorithm, which learns from moves of professional human players. Since the data from human players is insufficient to cover the search space, Generative Adversarial Networks (GAN) are used to augment learning data. Imitation learning is trained with different datasets. These datasets have moves that are generated by GAN, moves taken from one top human player (called Habu) and move taken from several professional human players. They tested their model with itself that is trained with different combinations of these training datasets. The model that is trained with GAN impact factor of 3.0 has outperformed all the other models, with 80% wins and draws. Their proposed model is not with any search algorithms.

There are some works that do not use shogi but are similar to this work. Anthony et al. [3] have developed a novel Expert Iteration (EXIT) algorithm, a combination of both deep RL and tree search. Specifically, it combines a variation similar to AlphaGo Zero and MCTS. A board game Hex is used as a test medium for EXIT, against MOHEX, a winner of computer hex tournament. MOHEX uses only MCTS. The two algorithms are tested on the same hardware, where MOHEX is observed to be slightly faster. EXIT outperforms MOHEX by 55 – 75%, based on time given per move. But the computation cost is not considered in comparison to the full extent. Lample and Chaplot [16] have used DQN to play a First-Person Shooter (FPS) game Doom. They used Deep Recurrent Q Networks (DRQN), which includes Long-Short-Term-Memory (LSTM). The inputs image frames are fed to Convolutional Neural Networks (CNNs). CNNs output is split into two NN inputs, a feature extracting dense NN and another NN with LSTM for action selection. Their algorithm is tested against human players in 15 min game-play sessions, and DQN has outperformed humans. The feature extracting NN is trained for the initial few hours until it reaches a detection accuracy of 90%. After this, the training of action selection LSTM NN is started. The total training time is 60 hours, but the hardware used is not mentioned. Adamski et al. [1] have worked on Batch Asynchronous Advantage Actor-Critic (BA3C) algorithm, aiming to show the algorithms implications for very large DRL applications. They Atari games as a testing medium for this algorithm and compared with three different versions of A3C algorithm. They tested complex RL algorithms on games that are less complex than shogi. They tested games on CPU and compared their performance on 12 core to 786 core CPUs. But they find several optimal hyperparameters. They prove that a batch size of 2048 is well suited with Adam optimizer

There are some significant developments made to DQN algorithm. Wand and Schaul have a developed a Dueling DQN algorithm [28]. Dueling DQN has a second neural network that works as a value function. The DQN and Value networks both import their inputs from one CNN layer. This algorithm is tested on multiple Atari games, that are used to test the DQN algorithm. Dueling DQN performs much better on the games, which have high action space, that DQN. On all 40 games, Dueling DQN outperforms DQN by 70%. In 18 games with high action space, Dueling DQN outperforms DQN by 83.3%. This algorithm also uses Prioritized Replay. The samples that have high TD error contribute more to the learning of DQN. These samples are given a higher probability in the Experience Replay of DQN. Hasselt et al. have developed a Double DQN algorithm [25]. They aim to solve the problem of overestimating the action value in DQN. DQN calculates TD error by choosing a maximum Q(s, a) value for the next state. This action with maximum Q value cannot always be the possible action to make in the sext state. Double DQN solves this by changing the equation through which TD error is calculated. This is a very minimal change to the DQN, and with this Double DQN gains a performance of 17% to 617% depending on the game the algorithms are tested.

In [23], they have developed self-play reinforcement learning algorithm for mastering the game GO without any supervision or using human data for training its model. Initially it starts with playing random games and learning from it. In order to accelerate the learning process they introduced a lookahead seach inside the the training loop for improving the accuracy and stability of the model.

The authors of [2] introduced an alternative approach to the conventional MCTS - called Evolutionary Monte Carlo Tree Search (EMCTS) which combines the features of MCTS and Evolutionary Algorithms(EA) to select the best move from all the possible moves. In the terms of EA, the nodes were equivalent to the genomes and each edge of the node in the decision tree is considered as a mutation of that genome. In the standard offline evolutionary algorithms, the parameters of AI agents are evolved by using the performance at playing the games as fitness function. However, the online evolutionary algorithm is a more dynamic approach in which the algorithm can be applied while playing the game itself. In the newly proposed EMCTS, tree search of MCTS and genome-based approach of evolutionary algorithms are combined to form the new structure of the algorithm. While experimenting, the algorithm was run on the game Hero Academy against various approaches like GreedyAction, GreedyTurn, Online Evolutionary Planning, and other flavours MCTS. The results indicate that EMCTS performs significantly better that the others. However, the algorithm has shown few tradeoffs while simulating more phases of the game and evaluating each time phase in the limited time period.

In [20], they have developed a MCTS program solve Shogi game. The program consists of many modifications from the classic MCTS. They have added a few techniques used in computer GO in addition to many enhancements specific to this game. While simulating the game, the playouts were assigned Elo ratings. based on current features of the move, this method estimates probability to that move. The probability that a move i will result in victory is calculated as follows:

$$P(i) = \frac{\gamma_i}{\sum_{j=1}^{n} \gamma_j} \tag{1}$$

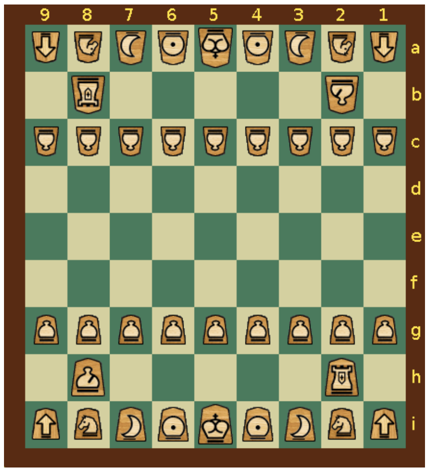
Where,

- γ_i is the strength of the individual i expressed in positive values.

The other significant change is in assigning values to the moves. Apart from the vanilla UCT, additional values namely progressive widening, killer moves, the histoy heuristics were also used. These techniques were also used and proven to be successful in Go.

The implementation is tested in two different phases. One, the progran is given the task to solve the 98 tactical shogi problem positions. To solve each problem, the time limit was set to 30 seconds. The algorithm was able to solve 49 out of 98 problems under the given time. In second phase of testing, the MCTS implementation was tested against the computer program TOHMI which is estimated to have a strength of about 1-dan amateur. The test was contested for 200 games with a time limit of 10 seconds per move for each program. Only 4% of the games were won by the MCTS algorithm.

Figure 1: Initial Shogi board and piece layout [5]



3. Background

3.1. Shogi

Shogi and Chess have same origins from an ancient Indian game. The current variant of shogi has been in use since the 16th century [11]. In Japanese, the word shogi means ‘general’s game.’ Shogi is a two-player board game. The game starts with each player having 20 pieces, of eight types. They are one king, one rook, one bishop, two gold generals, two silver generals, two knights, two lances and nine pawns. The beginning state of the board is shown in Figure 1, with the pieces in international notations. The player, whose pieces are in g, h, i rows, should make the first move. As the game progress, players capture the opponent’s pieces. These pieces can be placed back on the board as the player’s own pieces. This is called ‘drop.’ Because of this feature, there is no color (blacks and whites) for pieces in shogi. The pieces are differentiated based only on their orientation.

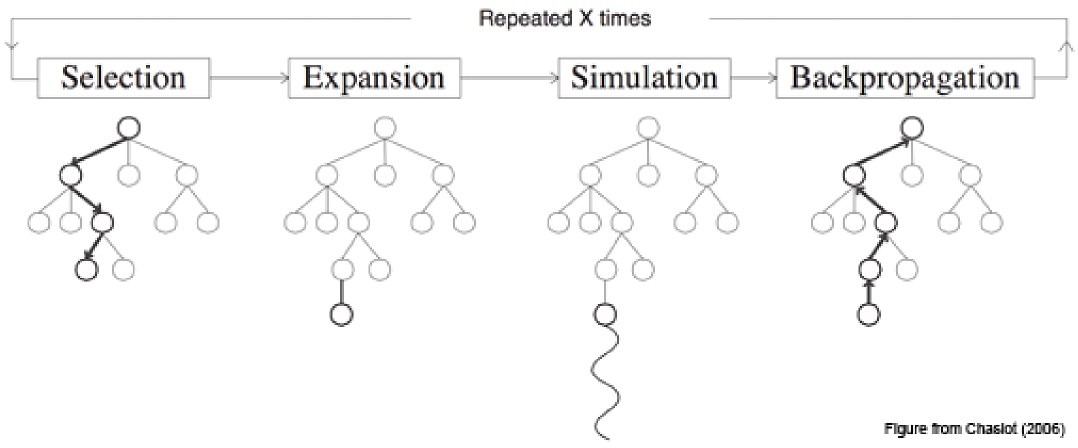
The farthest three rows from a player are ‘promotion zone’ for that player [11]. If a player’s piece passes this ‘promotion zone,’ the piece can be promoted. The promoted piece gets the move of gold general. Promoted lance and promoted knight lose their original movements. Promoted rook and promoted bishop get to retain their original moves and also promoted moves. Promoting a piece is optional, except for two cases. If pawn and lance reach the farthest row, and knight reaches the farthest two rows, then these pieces must promote. Once any piece is promoted, it cannot be demoted. When a promoted is captured, it loses its promotion state and will be dropped on the board with its demoted state.

Multiple reasons can lead to the end of the game. The objective of a player is to capture the opponent’s king. Unlike chess, it is not necessary to announce check before capturing the king [11]. If the king is captured, the game ends. If there is checkmate (similar to chess), the game ends. If any player receives same board state four times in a row, the game ends as a draw. This repetition is called four-fold repetition. However, if this repetition is caused by a player intentionally, that player loses the game, and this is decided by an external referee. In this implementation of shogi, any four-fold repetition is considered a draw. When both the kings are in their promotion zones, and neither of the players can do checkmate to their opponent, it is considered stalemate. At stalemate, the winner is deciding the counting the number of pieces on the board. In this implementation, it is considered a draw.

There are even more rules to drop a piece on board. A piece should be dropped at a location (square), such that it has at least one possible move after drop [11]. Pawn, lance and knight should not be dropped in their compulsory promotion rows. A pawn should be dropped at a location, such that there is no other pawn of the same player in that column. Any piece can be dropped to make a checkmate, except for pawn.

Chess is a heavily researched game since the beginning of AI research. However, the reason to select Shogi is that it is complex compared to chess, with its feature to drop captured pieces. This distinctive feature leads to a higher branching factor in comparison to chess. This feature alone leads RL algorithms to examine more positions and henceforth the highest number of legal positions and actions, and the highest number of possible games against all popular chess variants. All this translates to the difficulty of shogi as it becomes strenuous to reach the highest levels of the game. It is safe to state that, game complexity of shogi is higher than that of chess. Shogi’s game complexity amounts to the number of legal positions and possible games. Shogi and chess are compared in TABLE 1, which explains the complexity of shogi. Algorithms have reached human-level intelligence in this game as recently as

Figure 2: Phases of MCTS



2003. Moreover, algorithms have not defeated human champions until as recently as 2010.

3.2. Markov Decision Process

There is an immense amount of uncertainty involved in games. To make decisions under uncertainty, we need a thorough and formal framework such as *Decision Theory(DT)* [24]. Decision Theory also known as *'Theory of Choice,'* deals with how an agent can make decisions in an unfamiliar decision-making environment with unknown parameters and variables. DT is an interdisciplinary subject researched under economics, statistics, psychology, computer science and even in biology. In DT games such as chess where the state of the game is defined by sequences of choices fall under the category of Markov decision processes.

In circumstances such as games where the outcome of the decision making is partly under control of decision maker and partly random, Markov decision process (MDP) can be used [24]. MDP provides a mathematical architecture for stochastic decision-making scenarios. Not just game theory, this decision process framed almost 7 decades earlier and is used in economics, robotics, manufacturing and automation control.

Idea is to frame the problem of interactive learning to reach the goal and MDP offers an uncomplicated way to do this. In the game environment, the agent constantly interacts with the environment. Whenever an RL agent selects an action, the environment reacts to this and creates a new situation to the agent. MDPs are generally used in reinforcement learning to describe an environment which is fully observable.

MDP is based on Markov property. In order to understand MDP, it is essential first to have a clear idea of Markov property which states that "The future is independent of the past given the present". In Layman's terms, if we get enough information about the current state, you can essentially dispose of the historical information. According to MDP, the current state gives required statics to reach the same decision as in the case of all the history with preceding states. MDP's uses four units to describe the environment and model the decisions of the problem.

- **A:** a set of actions
- **S:** a set of states with s_0 being initial state
- **T((s,a,s'):** a state-action transition model that determines the probability of reaching state s' if action a is applied to state s
- **R(s)** A reward function for reaching desired state s

The decisions are represented as an array of (state, action) pair. The probability distribution of these pairs help in deciding the next state s'. The aim here is to find the best policy Π that yields the highest expected rewards.

3.2.1. Partially Observable Markov Decision Processes.

Depending upon the environment, there will be cases where the game is not fully observable. It means that the current state of the game cannot give complete information then that process is called as Partially Observable Markov Decision Processes (POMDP). In order to gain the necessary information required for selecting the next action, it first determines the current state based upon the probability distribution using a set of observation probabilities The formulation of POMDP is beyond the scope of this research and we will be focusing more on MDP. We suggest the reader to refer [12] for detailed information regarding POMDP.

3.3. Monte Carlo Tree Search

MCTS is elegant in the way that it is a simple algorithm, yet it delivers promising results. It can function effectively with no strategic knowledge to make reasonable decisions. This algorithm has been effectively applied in a variety of areas from the strategy games of Go, Othello, Tic-Tac-Toe, Rubik's Cube, Sudoku, Chess, and Lines of Action, to more Generic Games, planning, and optimization [6]. MCTS has shown significant achievements. However, in games where adversarial planning plays an important, like chess and checkers, minimax search with alpha-beta pruning [15] performs better. All the aforementioned games have the common property, that possible actions increase exponentially as game proceeds.

In an ideal scenario, if one could predict all the possible moves and their future results, chances of winning the game increase significantly. However, the exponential increase in moves requires enormous computational power. MCTS methods based on sampling could easily miss a critical move or underestimate the significance of an encountered terminal state due to averaging value backups. This where Monte Carlo Tree Search comes to the aid. Usually, in RL algorithms used for games, MCT is used to predict moves which would eventually lead the player to victory. However, prior to selecting the right move, all the moves of the present state should be arranged. Congregation of all the possible moves resembles a tree. Hence, this search is called tree search. MCTS could be progressively compelling in games, for example, Go, where terminal

Table 1: Complexity comparison of shogi and chess [2]

Game	Board Size	No: of Pieces	Types of Pieces	Legal Positions	Possible Games	Average Game Length
Chess	64	32	6	10^{47}	10^{123}	80
Shogi	81	40	8	10^{81}	10^{226}	140

positions and potential pitfalls are uncommon or do not happen until the terminal phase of the game. MCTS can here fully play out its strategic and positional understanding resulting from Monte Carlo simulations of entire games.

For example, to understand the inner workings of MCT and to visualize the exponential increase in possible moves, consider a game of Tic-Tac-Toe as shown. At each step, the player will be presented with multiple options which will different end outcomes. The move further increases in next turn resembling tree.

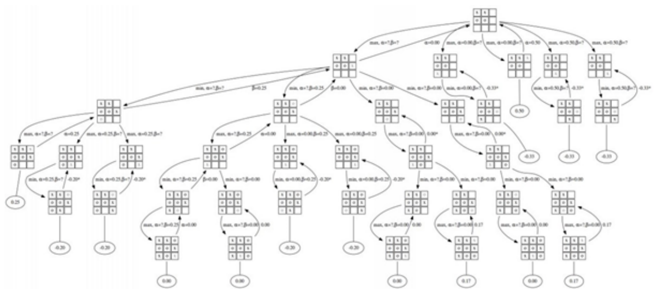


Figure 3: Tree expansion of TicTacToe [21]

If a simple game of Tic-Tac-Toe can have a graph as shown in fugure [x], imagine the possible moves in the game of shogi. Considering every possible output is not realistic and slows down the output significantly. Instead, favoring some nodes over the other and searching their children nodes first makes a faster tree search. This selection process of nodes and detailed explanation of MCT is explained further in the next section.

3.3.1. Overview of MCTS.

In this section, we briefly review the algorithm used in this paper: MCTS with Upper Confidence Bound1 for trees. Monte-Carlo Tree Search (MCTS) [9] is a best-first- search algorithm which uses statistical sampling to assess the game states. MCTS can be broken down to four phases. These phases are repeated in a loop until the computation time runs out or the terminal state has been reached [8]. Each iteration of the algorithm represents one simulated game.

Phase one - Selection: The tree is traversed from the root node to one of the leaf nodes, using a selection policy to move from one state to other. There are various policies for selecting the node. Some of the policies are discussed below.

1) **Max child:** The max child is the child that has the win-score value.
2) **Robust child:** The robust child is the child with the highest visit count.
3) **Robust-max child:** The robust-max child is the child with both the highest visit count and the highest value. If there is no such child with the given criteria, more simulations are run until the criteria is satisfied. [9].
4) **UCB1:** The child with the highest UCB1 value as the balance between exploitation of states with high estimated values and exploration of states with uncertain values is important. In this work, we use UCB1 - the UCT variant of MCTS as selection policy for balancing between the exploration and exploitation of the states in the game. UCB1 is discussed in detail in section 3.3.2
5) **Secure child:** The child with the maximum Lower Confidence Bound is selected.

Phase two - Expansion: When a leaf node has been reached, all the possible moves from the leaf node are expanded. A node is considered expandable if it has unvisited children and it is a nonterminal state of the game.

Phase three - Simulation: A simulation (also called as playout) is run until the desired result has been achieved. The simulation is purely random with some simple heuristics

Phase four - Backpropagation: we begin backpropagation when the simulation reaches the end of the game or until the desired level of depth of the tree has been reached. All the nodes visited during the playout have their playout-count incremented and if the playout has results in a win then the win-count is also incremented which are later used in selecting the promising node. This algorithm may be configured to stop after any desired length of time, or after reaching a certain depth of the tree. As more and more playouts are run, the statistics tree of game grows exponentially in memory. Hence it is crucial to select the optimal terminal condition depending upon the game and also the availability of computing resources.

Algorithm 1 General MCTS

```
1: $state$ ← initialize game
2: while $state$ is not terminal do
3:     $bestChild$ ← FindBestChild($state$)
4:     $state$ ← Perform($bestChild$)
5: end while
6: Input: $S_0$ - Current state of the game
7: Output: Best child of the current state
8: procedure FINDBESTCHILD($S_0$)
9:     $T$ ← initialize empty tree
10:    Create root node $r_0$ of T with state $S_0$
11:    while within computation budget do
12:        $P_0$ ← SelectNode($r_0$)
13:        $child$ ← Expand($P_0$)
14:        $winner$ ← Simulate($child$)
15:        BackPropagate($child, winner$)
16:    end while
17:    return BestChild($T$)
18: end procedure
```

3.3.2. Selection Policies:.

This step is crucial to the algorithm in the sense that it determines the direction of the growth of tree. It has to deal with exploration/exploitation dilemma precisely to avoid getting trapped in a local maxima/minima. Further, remaining steps of MCTS depend upon the selection of the state.

UCB1: *UCB1, or Upper Confidence Bound* for a node, is given by the following formula:

$$UCB = \frac{w_i}{n_i} + C\sqrt{\frac{\log N}{n_i}} \tag{2}$$

Where,

- w_i is the number of wins for the node,
- N is the number of times the parent node of i has been visited,
- n_i is the number of times the child node ith has been visited
- C is the exploration parameter. Theoretically C is equal to $\sqrt{2}$. But in practice this usually varies depending upon the game. Higher values of C lead to more exploration in the game and vice versa

Upper Confidence Bound for Trees (UCB) algorithm is probably the most important aspect of MCTS. The goal of MCTS is to approximate the value of the actions that may be taken from the current state. The success of MCTS, is primarily due to this tree policy. UCB1 has some promising properties: it is very simple and efficient and guaranteed to be within a constant factor of the best possible bound on the growth of regret. It is thus a promising candidate to address the exploration-exploitation dilemma in MCTS: every time a node (action) is to be selected within the existing tree, the selection is considered as an independent tree. A child node j is selected to maximize: UCB1. If more than one child node has the same maximal value, the tie is usually broken randomly. This also ensures that all children of a node are considered at least once before any child is explored.Thus there is an essential balance between the exploitation and exploration in the decision tree of the game. If one child node of the parent node is being visited multiple times, the denominator of the exploration term increases, which decreases its contribution. The exploration term ensures that each child has a nonzero probability of selection, which is essential given the random nature of the playouts. This also gives the peculiar property to the algorithm, as even nodes with low reward estimates are guaranteed to be chosen eventually (given sufficient computation time), and hence different paths of play are explored.

First Play Urgency(FPU):The UCB1 formula handles the exploration-exploitation dilemma effectively. However in the case where the nodes are from the root are visited rarely. In such cases UCB1 favours exploration of the states which have been already visited rather than maintaning the balance. FPU policy can handle such situations by choosing the nodes which have not been explored yet. To achieve this each node is assigned an urgency value using the formula of UCB1-Tuned. In this policy, the nodes by default are assigned a very high intital value(for instance around 10000). If the node has been atleast once, the urgency value of the node will be adjusted using the UCB1. Setting of this high initial value forces the MCTS to select the unexplored nodes first such that each node is visited atleast once before exploiting any further child nodes. UCB1-Tuned was first introduced in [4] and experimented on the Multi-Armed Bandit problem. UCB1-Tuned is represented as:

$$UCB1-Tuned = \frac{w_i}{n_i} + \sqrt{\frac{\min\left(\frac{1}{4},\ \mathrm{V}(t_k,\sigma_k,\mathrm{t})\right)\log t}{t_k}} \tag{3}$$

Where,

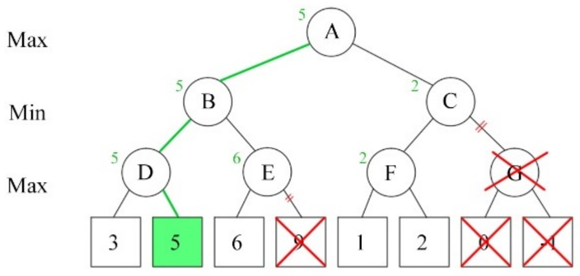
$$\mathrm{V}(t_k,\sigma_k,\mathrm{t}) = \sigma^2 + \sqrt{\frac{2\log t}{t_k}} \tag{4}$$

- w_i is the number of wins for the node,
- n_i is the number of times the child node ith has been visited,
- $t_k = \Sigma t(n')$
- $t(n)$ is the number of simulations involving node n, which is known as the visit count of node n.
- $r(n)$ is the empirical mean of the rewards the agent obtained from these simulations. Note that because it is a one-sum game, the average reward for the opponent is 1-r(n).
- $\sigma(n)$ is the empirical standard deviation of the rewards (which is the same for both players).

UCB1-Tuned is similar to UCB1 except for the parameter C which has been replaced by more efficient upper bound on the variance of the rewards which is either $\frac{1}{4}$ or V(t_k,σ_k,t) (the upper confidence bound computed from samples observed so far).

3.3.3. MiniMax with Alpha-Beta Pruning :. MiniMax is a decision-making strategy, usually utilized in a turn-based 2 player games.Here one player is termed as maximizer while the other player is termed as minimizer. Hence while selecting the states from all the possible states, the maximizer will choose a state which has maximum score and vice versa.

Figure 4: Alpha-Beta pruning

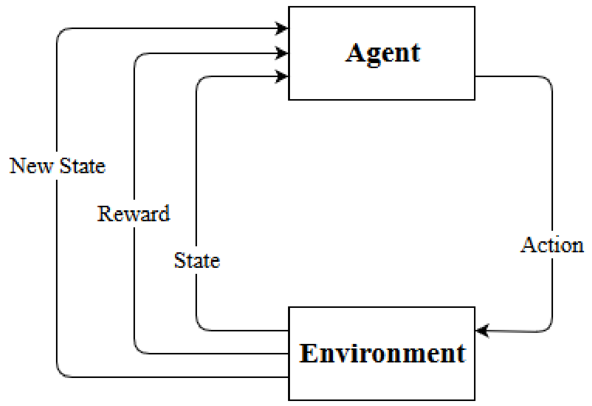


As this strategy makes use of decision trees to evaluate the possibilities, it becomes very time consuming to construct and evaluate each and every possibility of the game. For games as simple as TicTacToe the number of different states is 255168. Now we see the problem. As the complexity of the game increases, the decision tree space increases exponentially. Alpha-Beta Pruning is one such optimization algorithm which helps to cut down the nodes of the tree without even visiting them based upon the scores returned by the evaluation function of the game.

3.4. Deep Q Networks

The working of most RL algorithms can be explained with Figure 5. In this paper, the agent is DQN, the environment is shogi. The agent takes action *a*, based on current state *s* of the environment. For the action, the agent receives a reward *r*, and the environment is in a new state s'. The agent always tries to take actions to maximize its rewards. One of the pre-requisites for an environment to be used for RL algorithm is, it can be convertible to Markov Decision Process (MDP). The gameplay of shogi happens in discrete time-steps. The number of states in shogi is large but finite with a terminal state. Hence, the environment is convertible to MDP and RL algorithms can be applied to this environment.

Figure 5: Working overview of RL with its basic elements.



DQN were first introduced with their success in playing Atari games in Atari 2600 emulator [2]. DQN was able to make strategic decisions and mimic human-like or better moves in more than 25 Atari games. DQN works similar to Q-Learning (QL). QL is a tabular RL algorithm, which does not use any neural networks. It stores its Q values for the state-action pairs in a table. This table is updated iteratively during the learning process and the agent takes actions according to these Q values. DQN replaces this table with a neural network, which works as a function approximator. The reward received by the agent is used to update weights of the neural network, based on the Bellman equation (Equation (5)). γ in this equation is called discount factor. Gamma is a constant, that decides the priority between immediate rewards and future rewards. For the agent to be strategic in shogi, sometimes it must take actions to maximize future reward, not immediate reward. For example, by capturing a pawn, the agent gets a small reward. There can be game state, such that by not capturing the pawn in that state might lead the agent to capture the king in the next consecutive states. If the agent takes action for small immediate reward, it might not get the chance to get more significant future rewards.

$$Q'(s,a) = Q(s,a) + \alpha[r + \gamma max(Q^*(s',a')) - Q(s,a)] \quad (5)$$

DQN is not the only algorithm to combine RL and Neural Networks (NN). Neural Fitted Q-learning (NFQ) [19] is an example. However, the following features make DQN work better at games like shogi.

- Experience Replay (ER)
- Target Neural Network

Even with neural networks, DQN uses a memory. This memory does not store all the (s, a)values as in QL, but works as a buffer to store s, a, r and s'. This buffer works like queue with fixed memory size. The first (s, a, r, s) values to enter the memory, leaves first. The NN is trained on a batch of samples, randomly chosen from this memory. This process is called Experience Replay (ER). ER is used to avoid overfitting, a common problem of deep NN. By choosing a batch size based on the application, ER can also help in faster learning of the agent [17].

The $Q^*(s', a')$ in Equation (5) is the target Q value generated by a separate Target NN. Target NN is a clone of the agent DQN, with its weights updated only for every fixed C steps. The weights of Target NN stay constant during the C number of steps and then cloned again with the DQN. With the usage of a separate target NN, the agent is observed to be more stable than QL and avoids divergence of the policy [18].

$$TD(\Delta) = \alpha[r + \gamma max(W^*(s',a')) - W(s,a)] \quad (6)$$

Since DQN works with NN, the agent must use NN weights instead of Q values. Hence, Equation (5) is modified to Equation (6). DQN weights are randomly initialized. DQN agent predicts based on the input from shogi environment. Error in this prediction is calculated using Equation (6). This error is also called the Temporal Difference (TD) error. $W^*(s', a')$ is the weights of prediction by Target NN for the state s'. Only the maximum of output weights of Target NN is considered while calculating the error. The DQN agent is trained iteratively by trying to reduce this loss function, i.e., TD error, for the samples chosen from ER.

Exploring the state space plays a major role in RL algorithms as they learn only through self-play. This exploration is facilitated by introducing randomness to the actions taken by the agent. Exploration makes an agent try new ways to solve a given problem. After a learning process, if the agent is making actions, for which it already knows that it will get a positive reward, then it is called exploitation. The policy that tells the agent when to explore and when to exploit is called behavioral policy. DQN follows ϵ-greedy policy. Every action agent takes has a ϵ probability for exploration and (1-ϵ) probability for exploitation. Since DQN agent starts with no knowledge about its environment, its initial actions are full exploration. The ϵ is linearly reduced to 0.1, and from then it is kept constant at ϵ = 0.1.

4. Implementation

Both MCTS and DQN algorithms are developed with python programming language. As a game environment, python library *Python-Shogi* is used. This library was written only in python, and it is chosen as it gives the possibility of changing the source code to the algorithms' requirements. As previously said, shogi has many rules and programming the full game will be a difficult task. This library is used mainly for the following features:

- Generate a list of all the possible legal moves for the current player.
- Check if the game has ended, with the possibility to check the reasons that caused the game to end.
- To access the list of all moves that were performed for the current board.
- To check if the current move results in the promotion of the piece to be played.
- To access the list of all attackers for a given piece.
- To get the type of piece at the given position

A wrapper is developed to make this game library compatible with the inputs and outputs of MCTS and DQN.

4.1. MCTS

In the selection phase of the MCTS where UCT values are used to select the best promising node, a value of $\sqrt{1.41}$ was used as this resulted in a balanced exploration-exploitation state. The helper function `Board.legal_moves` is used to generate all the possible states in the expansion of the current state. Shogi is one of the complex game where the number of possible games is as high as 10^{226}. In the figure 5, the graph shows the number of possible states with the increase in depth of the tree during simulation phase of MCTS. We can also observe that the time taken(in seconds) by the algorithm increases exponentially with the increase in the number of moves in the figure 6. In the initial stages of the game MCTS tends to favour random moves. As the game progresses it starts exploiting different powers of the shogi characters.

Figure 6: Depth of the tree vs possible states

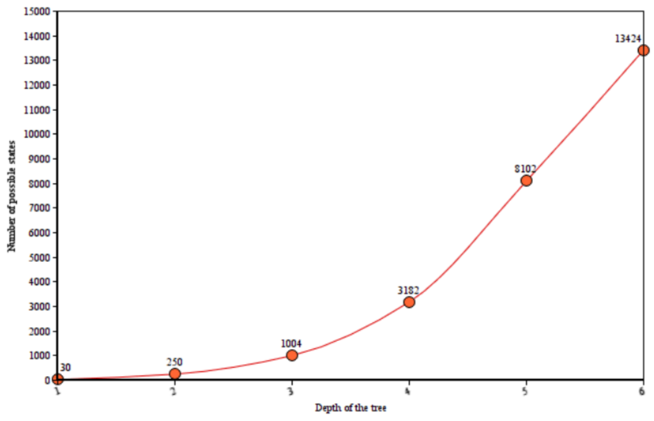
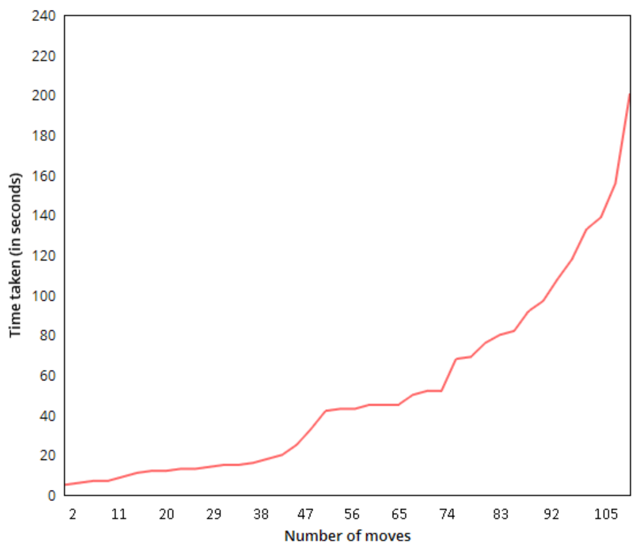


Figure 7: Time taken vs Number of moves



Rewards Policy: Table 2 indicates the rewards given to the moves if the current move captures any opponent piece. A score of 2 is added to the initial score if the move results in promotion of the player's piece irrespective of whether it captures the opponent's piece.

4.2. DQN

The need for different application areas has led to the development of different types of neural networks, from simple stacked auto-encoders to Collaborative Evolutionary Reinforcement Learning [13], that combination evolutionary and RL algorithms. When

Table 2: Values of pieces(+ indicates a promoted piece)

(Pawn) 6.6	(Lance) 7.7	(Knight) 7.7	(Silver) 7.7
(+Pawn) 8.0	(+Lance) 8.0	(+Knight) 8.0	(+Silver) 8.0
(Gold) 8.3	(Bishop) 8.3	(Rook) 8.3	
(No Promotion) 8.7	(+Bishop) 8.7	(+Rook) 8.7	
Check 9.1 Checkmate 10			

DQN was first introduced to play Atari Games, Convolutional Neural Networks (CNN) and Fully Connected Neural Networks are used. CNNs are a family of neural networks, which have demonstrated promising result in the field of image recognition, classification, identification of objects, self-driving cars and in natural language processing tasks [10]. CNNs are majorly used to extract useful features from individual pixels of the image, using the convolution operator. Convolution operator preserves the spatial information of the pixels by learning image features. For instance, every image is considered as a matrix with pixel values. The image can be represented as a greyscale (or single-channel) or image with red, blue and green (three-channel image with each color for one channel). Hence, each channel is a 2d matrix with pixel values. In the convolution step, a filter will be applied on input data to get a feature map (also called convolved features). The filter is initialized randomly with a 0's and 1's to extract essential features from the input. The NN will not learn any features if the filter is initialized with all zeros. The most common size of the kernel is 3x3, 5x5 or 7x7. By sliding the filter by 1 pixel on the input from left to right, it will compute a feature map that is a sum of element-wise matrix multiplication of input. More complex features can be extracted by varying the number of filters and their size. Since shogi has a 9x9 board, CNN can be used to extract features from the piece positions on the board. Like a three-channel image, this board can be converted to multiple planes based on pieces on board and their positions. The output from this CNN is fed to Fully Connected Neural Networks, also called as Dense Neural Networks (DNN).

Shogi begins with 20 pieces for each player. These 20-piece positions and possible legal actions of each player are given as input to the CNN. As the game progress, the pieces that can be dropped on the board increases for both players. So, these captured pieces' moves need to be added to the CNN input. As the game progress, pieces can get promoted for both players. Again, there is a need to add promoted piece positions and their possible moves, to the CNN input. When all these pieces positions and their moves are individually given as CNN input, the computation cost will be too high. So, positions of similar pieces are combined as one input. For example, all pawns' positions are combined in one input. However, the pieces positions and their possible moves are not combined. In total, there are 63 inputs for the CNN, and each input is a 9x9 plane. These 63 inputs are shown in TABLE 3. DQN receives these 63 inputs in the same order every time. The positions of opponent's drop pieces are not considered for CNN input. It is because the game environment library does not allow the current player to see the captured pieces the opponent has. Keeping a previous record of player's pieces on board can solve this problem. But, it will make the algorithm even more computationally expensive.

Table 3: Inputs of CNN.

Player	Input Data	Planes
Player	Positions of non-promoted pieces	8
	Moves of non-promoted pieces	8
Opponent	Positions of non-promoted pieces	8
	Moves of non-promoted pieces	8
Player	Positions of promoted pieces	6
	Moves of promoted pieces	6
Opponent	Positions of promoted pieces	6
	Moves of promoted pieces	6
Player	Drop positions of captured pieces	7
	Total Inputs	**63**

Three layers of CNN are used for DQN, each with filter sizes 5x5, 3x3 and 1x1. The first layer of CNN uses 63 filters to match the inputs. This filter number is doubled in the second layer to 126 to extract more features from the inputs. The third layer does the same as the second layer, but with a smaller filter size. The output filter's width is calculated using the equation shown in Equation (7). The same equation is also used for using an output filter's height. After the three CNN layers, the three-dimensional data is flattened. Flatten layer acts as a bridge between CNN and DNN layers. DNN part of the neural-network layout also has three layers.

$$Output_{width} = \frac{W - F_W + 2P}{S_W} + 1 \tag{7}$$

Where,

- W is the width of the input data sample.
- F_W is the width of the filter
- S_W is a stride in width direction. Stride is the movement of the filter on the input. If S_W = 1, the filter moves one data unit in the width direction.
- P is padding, which is the number of data units one side of the filter can move further than the border data unit

As said before, DQN uses two neural networks: agent model, the one used for prediction and target model, the one used to generate target values. For shogi, the agent's output should be a valid move, that tells the from and to location squares of a piece. AlphaZero [22] solves this problem by using a neural network, that outputs move probabilities, not piece probabilities. The DQN used here, solves this problem by using a second neural network. The first network (called s-model) gives the output, which is used to know the piece to move, and the second network (called a-model) gives the output, which is used to know the destination of the selected piece. These two networks combine to make the DQN agent. Both s-model and a-model have their own targets models, which are used to train these two neural networks. So, this DQN used four neural networks in total. This does not make this algorithm Dueling DQN [28], which uses a second neural network to find the value of a current state. This algorithm uses DQN principles alone and for the shogi application, the second neural network is added. Dueling DQN uses one CNN, whose output is sent to two DNNs. But, this DQN uses two separate CNNs, one to extract piece position information and the other to extract action location information, from the input.

Both s-model and a-model have the combination of CNN and DNN. The outputs of both DNNs are combined to make the DQN agent's action. The DNN of s-model has 88 output neurons. Out of these, 81 represent the 81 squares on the board. The remaining 7 represent the types of captured pieces. Out of the movable pieces, that are already on the board and that can be dropped on the board, the one with the highest weight of corresponding neurons is chosen to be moved. The DNN of a-model has 81 output neurons, representing 81 squares of the board. Out of the possible locations, to which the piece can be moved, the one with the highest weight of corresponding neurons is chosen as piece destination. . The entire NN layout of a-model is shown in Figure 8. s-model differs from a-model only in output DNN layer having 88 neurons, instead of 81. The number of weight vectors in the two NNs are shown in TABLE 4.

Table 4: The number of weight vectors in two NNs of DQN

Layer	s-model	a-model
Convolutional	14238	14238
Convolutional	71568	71568
Convolutional	16002	16002
Flatten	0	0
Dense	168033936	168033936
Dense	1037673	1037673
Dense	38896	35802
Total	169,212,313	169,209,219

When the agent takes action, the neural network's weights are updated according to the reward received for that action. The rewards are given only when an opponent's piece is captured. Otherwise, the reward is zero. This reward values range from 1 to 16, based on the captured piece. By capturing the king, the agent gets a reward of 1000. As DQN uses ER, the weights are updated from the random samples chosen from ER memory. ER uses a queue-like memory. It acts like a buffer and first sample to enter ER memory, leaves first. The ER used here has a memory size of 2000. Training is done using a batch of 10 random samples from this memory.

The behavioral policy used by DQN is ϵ-greedy policy. In this implementation of DQN, the ϵ value is kept constant for one game and is changed only when the new game starts. Since, the agent is trained for only 10 episodes, the ϵ value is reduced exponentially instead of linearly. The DQN algorithm used for Atari Games [17], reduces ϵ from 1 to 0.1. If we provide ϵ=1 for one full game, the DQN algorithm cannot implement what it has learned, in more than 200 moves. So, ϵ is exponentially reduced from 0.8 to 0.1.

In shogi, the player has the choice of whether to promote his piece or not. The decision to promote the piece requires strategic thinking. Because, lance and knight have unique moves in the original state, which will be lost promoted state. For the easier implementation of the algorithm, this DQN always chooses to promote its pieces when there is a choice.

The entire algorithm is developed in python language. The neural networks are developed with Keras python library, using TensorFlow as backend. The neural network weights are updated according to a loss function. In DQN, TD error is a part of this loss function. Gradient descent optimization algorithms are used to know whether the loss function is increasing or decreasing. Often, they were used as black box optimizers for neural networks and widely used for hyperparameter tuning, especially to minimize the loss function. These optimization algorithms are less time consuming,

Figure 8: Layout of NN layers for model used for action selection

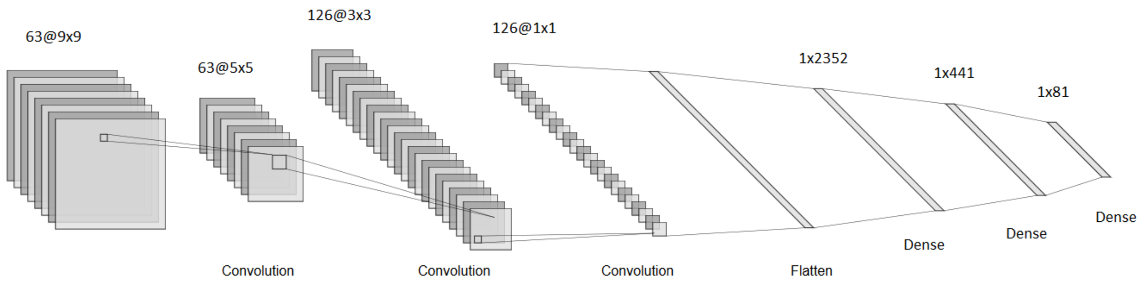
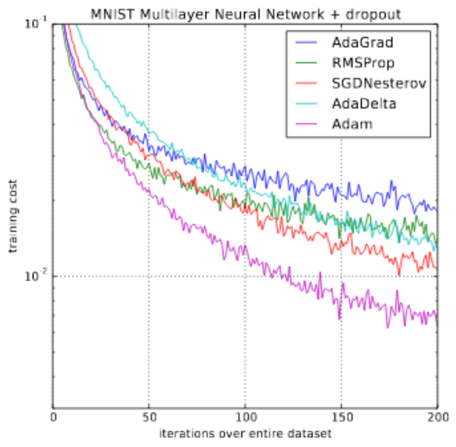


Figure 9: Comparison of training cost of different optimizers [14]



easy to compute, and these converge fast. There are three types of gradient decent optimization techniques:

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

SGD is faster than batch gradient descent. In SGD, the frequent update of parameters causes high variance in parameters update and fluctuations in the loss function. These loss function fluctuations enable SGD to find a local optimum faster. However, it complicates convergence to an exact local minimum. This problem is generally solved by using a small learning rate.

There are many optimization algorithms developed for deep neural networks. One of them is Adaptive momentum estimation (Adam), and this is chosen here. Adam computes adaptive learning rates for each parameter and stores an exponentially decaying average of past gradients [14]. Adam has relatively low training cost and it can be seen in Figure 9. Adam is used for this DQN with a learning rate of 0.0001.

Every neuron in a NN has an activation function. Activation functions define neuron's output for a given input data. There three widely used activation functions: Rectified linear unit (ReLU), Tanh and Sigmoid. There is also linear function, but it is less used, as most of the real-world data is composed of non-linear patterns. Also, all the layers of a neural network should never be activated with linear function for an RL application. The output of ReLU, Tanh and Sigmoid to a neuron map to 0, 1, -1, 1 or 0, max, and outputs of these three activation functions are shown in Figure 10. ReLU is chosen for this DQN, as it has shown better performance than the other activation functions. ReLU output is defined as the following Equation 8 where x is an element from the input or a feature map from CNN.

$$f(x) = max(0, x) \tag{8}$$

4.3. Results

Table 5: Hardware specifications while testing the algorithm:

Processor:	Intel Core i3-5005U CPU
Ram :	12 GB
Ram frequecy:	1600 MHz
System type:	64-bit Windows Operating System

MCTS was tested against a random agent. The algorithm won 70% of the games. Due to vast size of the possible moves and without alpha-beta pruning, the number of tree nodes reached to a massive number 150,000 nodes which resulted in high computation costs and time. The average time required for calculating each moves is 45-50 seconds. Below the top 5 games of MCTS are listed

Table 6: Top 5 games of MCTS against random agent

Game	Total Moves	Time (min)	Game-over by
1	152	95	Checkmate
2	178	104	Checkmate
3	200	118	Checkmate
4	203	109	4-fold repetition
5	220	121	Checkmate

DQN is trained on ten games, and the training metrics are shown in TABLE 7. The hardware used is a 2-core 4-threaded Intel i7-7500U CPU, and the algorithm utilizes all cores and threads of CPU. On average, the DQN algorithm took 277 minutes to train with 336 moves per game on current hardware. So, it took nearly 50 seconds to make a move while learning.

The trained DQN is tested against a player who always makes random moves (called random player). Only random moves are selected for testing because the DQN implemented here can only

Figure 10: Different types of activation functions

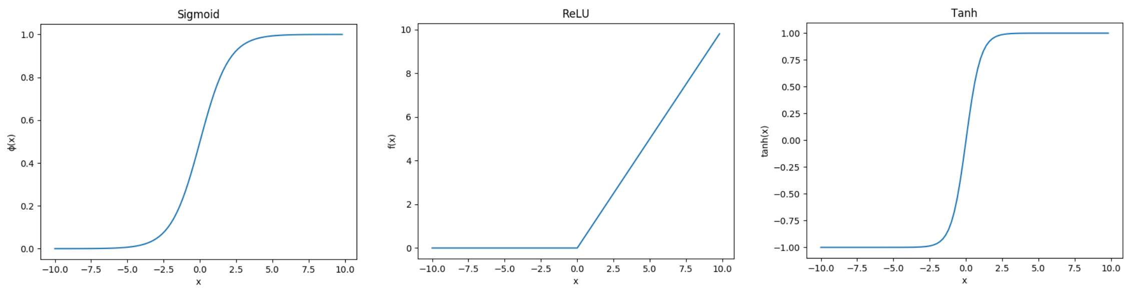


Table 7: Training Metrics of DQN.

Game	Winner	Total Moves	Time (min)	ϵ	Game-over by
1	Player 1	320	214	0.8	Checkmate
2	Player 1	700	437	0.525	Checkmate
3	Player 1	226	205	0.36	Checkmate
4	Player 2	305	275	0.256	Checkmate
5	-	220	205	0.19	4-fold repetition
6	-	370	323	0.15	4-fold repetition
7	Player 2	303	269	0.13	Checkmate
8	-	638	588	0.12	4-fold repetition
9	-	50	49	0.11	4-fold repetition
10	-	226	207	0.10	4-fold repetition

Table 8: Test Results of DQN

DQN vs. Random Player	Random Player vs. DQN
2 wins	2 wins
6 draw	5 draw
2 lose	3 lose

be slightly intelligent. Choosing a very small batch size for training, experimenting with separate a-model and s-model hinder more intelligence of DQN. AlphaZero uses a batch size of 4000+ sample. The DQN used for Atari games uses a batch size of 32 samples. With a small batch size, the agent is trained only on ten games. In the trained ten games, it is observed that the DQN agent never took an action that captures the king, even when there are many chances. So, the agent never knew capturing the king gives it a very high reward. Instead, it went after smaller rewards it receives when it checks the opponent's king. The testing results of 20 games are shown in TABLE 8. Testing is done by changing e-value, to be either 0 or 1 based on the turn of the player. ER and Target NN of DQN are disabled during the testing phase. Most of the tested games ended by reaching a maximum step count, which ends in a draw. The total moves played only DQN algorithm are averaged, and the average time per move while testing is 0.887 seconds, which is significantly less than time per move in training.

In the observed literature, where algorithms learn through self-play, player-1 always has the upper hand over the game as they get to start the game. The testing results of DQN contradict this by having more wins against a random player while playing in the second position. The final two wins by player-2 during training can be the cause of this contradiction.

DQN has several parameters that need to be optimized for better performance. These parameters can be learning rate, activation function, filter size of CNN, output size of CNN layers, and more. On the other hand, MCTS has fewer parameters that need optimization. These parameters are:

1) Exploration Constant - This constant is responsible for maintaining the balance between exploration and exploitation. Higher values of exploration constant lead the monte carlo tree search algorithm to explore the unvisited nodes first and then exploit them. After series of simulations the value of $\sqrt{1.41}$ was found to be optimal for this game and the tests against the random agent were run using this constant.

2) Simulation Lookahead - During the simulation phase, the alogrithm simulates the game in order to determine the current player's win-score. However, if the game is complex then simulating the game until the terminal state has been reached takes a toll on the computation speed of the algorithm. This parameter can be reduces to a limited number of moves during the simulation and evaluate the value of the game till this state only leading to faster computations. Due to the high range of the branching factor(92 for Shogi) the lookahead is set to 15 moves

5. Conclusion and Future work

The results from the previous section prove that DQN is computationally expensive. In the training phase of neural networks requires more computation resources than search algorithms. When the testing phase is considered, DQN performs faster than MCTS. The only advantage DQN has over MCTS is a faster testing time. DQN has several disadvantages over MCTS. They are as follows:

- DQN requires more setup effort with parameter optimization.
- DQN requires more computation resources.
- DQN has significantly high training time compared to no training time for MCTS.

DQN is a simpler algorithm in comparison to the successful algorithms for shogi, like AlphaZero, imitation learning with GAN. When these complex algorithms are compared to MCTS, that will make the above disadvantages even worse.

For search algorithms, to be equal to or more intelligent than neural network (trained with high computation cost) algorithms, they require integration of human knowledge and search optimization techniques. The resources humans use to acquire knowledge in the problem domain, and the resources required to integrate this knowledge into search algorithms, should also be considered as the search algorithm's computation resources. Also, human knowledge cannot be available for every problem.

The implementation of this DQN can be improved in multiple ways. A value network can be added as done in Dueling DQN. Value networks analyze the board and determine who is going to win in the current state. This information is valuable in making actions and assigning rewards. When value network is used, the reward calculation is split as V(s) and V(a), instead of combining them as Q(s,a). When the game is already in a bad state, the actions taken in that state cannot be much better. With this, unnecessary high rewards will be controlled. Double DQN also might help in the performance improvement of DQN with very minimal changes to the current algorithm. Another way to improve DQN performance is to use MCTS for the training of DQN. AplhaZero proves the usage of MCTS during training will result in stable learning of value network. MCTS covers more search spaces in a shorter time than DQN can. The depth of MCTS can be just 3-5 steps to provide valuable information for DQN training.

While simulating the game, general knowledge was used to determine various rewards depending upon the opponent's piece if the current player captures it. These values can be changed to study how the changes in the values of the rewards affect the outcome of the game. Due to higher complexity of the selection-policy First Play Urgency (FPU), UCB1 was chosen. However for strategic and complex games like shogi, FPU has shown to be more effective in choosing the promising nodes. The effects, outcomes, and rewards can be studied further in future for the better understanding

References

[1] I. Adamski, R. Adamski, T. Grel, A. Jędrych, K. Kaczmarek, and H. Michalewski. Distributed deep reinforcement learning: Learn how to play atari games in 21 minutes. In *International Conference on High Performance Computing*, pages 370–388. Springer, 2018.

[2] L. V. Allis et al. *Searching for solutions in games and artificial intelligence*. Rijksuniversiteit Limburg, 1994.

[3] T. Anthony, Z. Tian, and D. Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[4] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[5] H. L. Bodlaender and F. Duniho. Shogi (): Japanese chess. https://www.chessvariants.com/shogi.html. Accessed: 2019-06-25.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[7] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[8] G. M. J. Chaslot, M. H. Winands, H. J. V. D. HERIK, J. W. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[9] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[10] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[11] T. Hosking. *Classic Shogi: Games Collection*. Shogi Foundation, Stratford-upon-Avon, England, 2006.

[12] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[13] S. Khadka, S. Majumdar, S. Miret, E. Tumer, T. Nassar, Z. Dwiel, Y. Liu, and K. Tumer. Collaborative evolutionary reinforcement learning. *arXiv preprint arXiv:1905.00976*, 2019.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[16] G. Lample and D. S. Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[19] M. Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[20] Y. Sato, D. Takahashi, and R. Grimbergen. A shogi program based on monte-carlo tree search. *Icga Journal*, 33(2):80–92, 2010.

[21] S. Sharma. Monte carlo tree search. https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa. Accessed: 2019-06-27.

[22] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[23] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[24] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. 2011.

[25] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[26] S. Wan and T. Kaneko. Imitation learning for playing shogi based on generative adversarial networks. In *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 92–95. IEEE, 2017.

[27] S. Wan and T. Kaneko. Building evaluation functions for chess and shogi with uniformity regularization networks. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[28] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.