**REINFORCEMENT LEARNING CHESS**

**PROJECT REPORT**

**By:**

**PAVAN VENKATA SAI KALYAN GEDALA**

**Introduction**

Chess is one of the oldest and most popular board games in the world. It has been played for centuries, and it is still considered one of the most challenging games to master. Chess is a game of strategy, requiring players to think several moves ahead and make calculated decisions. With the advent of artificial intelligence and machine learning, researchers have been exploring the use of reinforcement learning (RL) to teach machines how to play chess.

Reinforcement learning is a type of machine learning where an agent learns to make decisions based on feedback it receives from the environment. In the case of chess, the agent would be the machine learning algorithm, and the environment would be the chessboard. The agent learns to make moves that maximize its rewards while minimizing its losses.

The problem we aim to tackle in this project is to teach a machine learning algorithm how to find the shortest path between two squares on a chessboard. This is a challenging problem, as there are numerous possible paths between any two squares, and the agent must learn to find the optimal path while avoiding obstacles such as other pieces on the board.

**Motivation**

The motivation behind this project is to explore the capabilities of RL algorithms in solving complex problems such as finding the shortest path on a chessboard. RL has been successfully applied to a range of problems, including game playing, robotics, and autonomous driving. By applying RL to the game of chess, we aim to develop a better understanding of how these algorithms can be used to solve complex problems with large state spaces.

**Objectives**

The primary objective of this project is to develop an RL algorithm that can find the shortest path between two squares on a chessboard. To achieve this objective, we will use a range of RL techniques, including policy iteration, value iteration, and TD learning. We will train the algorithm using a dataset of chessboard configurations and corresponding shortest paths between squares. The performance of the algorithm will be evaluated using various metrics, including accuracy, computational efficiency, and generalization to unseen data.

**Literature Review**

**Reinforcement Learning and Chess:** Reinforcement Learning is a subfield of machine learning that involves learning by interacting with an environment to maximize a cumulative reward signal. Chess is a classic example of a complex, strategic game with a large state space that can benefit from RL algorithms. Reinforcement Learning has been used to develop chess playing agents that can learn to play the game from scratch and even surpass human performance.

**Dynamic Programming:** Dynamic Programming is a method to solve complex problems by breaking them down into smaller subproblems and solving them in a recursive manner. DP is a foundational concept in Reinforcement Learning and is often used to solve problems in discrete state and action spaces.

**Policy Evaluation:** Policy Evaluation is a process of estimating the value function of a given policy. The value function measures the expected cumulative reward that an agent can obtain by following a particular policy.

**Policy Improvement:** Policy Improvement is the process of finding a better policy given an existing policy. This is achieved by changing the action selection policy to better exploit the current value function estimate.

**Policy Iteration:** Policy Iteration is a method of alternating between Policy Evaluation and Policy Improvement to find the optimal policy for a given MDP. Policy Iteration converges to the optimal policy in a finite number of iterations.

**Value Iteration:** Value Iteration is a method to compute the optimal value function for a given MDP by iteratively applying the Bellman optimality equation. This method can converge faster than Policy Iteration and is often used for large state spaces.

**Monte Carlo (MC) Prediction and Control:** Monte Carlo methods involve estimating the value function by simulating episodes and computing the average return. MC prediction is the process of estimating the value function of a given policy using MC methods, while MC control involves finding the optimal policy using MC methods.

**Existing Work on Reinforcement Learning in Chess:**

There have been several attempts to develop chess playing agents using Reinforcement Learning. One of the earliest successful attempts was TD-Gammon, which used TD learning to train a neural network to play backgammon. More recently, AlphaZero, developed by DeepMind, used a combination of Monte Carlo Tree Search, Reinforcement Learning, and deep neural networks to learn to play chess at superhuman levels. These and other studies demonstrate the potential of Reinforcement Learning in developing intelligent chess playing agents.

# **Q-networks**

* The Q-network is usually either a linear regression or a (deep) neural network.
* The input of the network is the state (S) and the output is the predicted action value of each Action (in our case, 4096 values).
* The idea is like learning with Q-tables. We update our Q value in the direction of the discounted reward + the max successor state action value.
* We used prioritized experience replay to de-correlate the updates.
* We used fixed-Q targets to stabilize the learning process.
* We built two networks, A linear one and a convolutional one.
* The linear model maps the state (8,8,8) to the actions (64,64), resulting in over 32k trainable weights! This is highly inefficient because there is no parameter sharing, but it will work.
* The convolutional model uses 2 1x1 convolutions and takes the outer product of the resulting arrays. This results in only 18 trainable weights!
* Advantage: More parameter sharing -> faster convergence
* Disadvantage: Information gets lost -> lower performance
* For a real chess AI, we need bigger neural networks. But now the neural network only must learn to capture valuable pieces.

So what has the network learned? The code below checks the action values of capturing every black piece for every white piece.

* We expect that the action values for capturing black pieces is like the (Reinfeld) rewards we put in our environment.
* Of course, the action values also depend on the risk of re-capture by black and the opportunity for consecutive capture.

**Model Free Learning**

**The environment**

* The state space is a 8 by 8 grid
* The starting state S is the top-left square (0,0)
* The terminal state F is square (5, 7).
* Every move from state to state gives a reward of minus 1
* Naturally the best policy for this environment is to move from S to F in the lowest amount of moves possible.

**The agent**

* The agent is a chess Piece (king, queen, rook, knight or bishop)
* The agent has a behavior policy determining what the agent does in what state.

**Reinforce**

* The reinforce object contains the algorithms for solving move chess.
* The agent and the environment are attributes of the Reinforce object.

Model-free learning is commonly employed in reinforcement learning, a subject of machine learning that is concerned with learning how to conduct actions to maximize a reward signal in an unpredictable environment. Examples of applications of model-free learning include robots, game playing, and autonomous driving.

Model-free learning algorithms may be categorized into two types: value-based and policy-based. In value-based methods, the agent learns to estimate the value of each state or state-action pair and uses these estimates to select actions. In policy-based methods, the agent learns to directly map states to actions without estimating the value of each state.

Model-free learning has the benefit of being more flexible than model-based techniques, as it does not require a comprehensive and accurate model of the environment. However, it may require more data and computational resources to learn an optimal policy compared to model-based approaches.

### Monte Carlo tree search

* Instead of Q-learning (learning action values) I use "V-learning" (learning state-values).
  + An advantage is that the Neural Network can learn with fewer parameters since it doesn't need to learn a separate value for each action. In my Q-learning and Policy Gradient notebook, the output vector has a size > 4000. Now the size is only 1.
* The V-network is updated using Temporal Difference (TD) Learning, like explained in Notebook 1.
  + This option is the simplest to code. Other options are TD-lambda and Monte Carlo Learning.
* The Architecture of the V-network is quite arbitrary and can probably be optimized. I used a combination of small and large convolutions, combined with convolutions that range a full file or rank (1-by-8 or 8-by-1).
* Moves are planned using Monte Carlo Tree Search. This involves simulating playouts.
  + Monte Carlo Tree Search greatly improves performance on games like chess and go because it helps the agent to plan.
* These playouts are truncated after N steps and bootstrapped.
  + This reduces the variance of the simulation outcomes and gives performance gains, since the simulation doesn't require a full playout.
* For this version, the opponent of the RL-agent is a myopic player that always chooses the move that results in the most material on the board or a checkmate.

Monte Carlo Tree Search (MCTS) is a decision-making technique used in artificial intelligence for games and other sorts of domains with significant branching factors, such as planning and optimization issues. MCTS incorporates features of both simulation and search to help the decision-making process.

At a high level, MCTS generates a tree of potential game states, with each node representing a state and each edge indicating a possible action. The algorithm then conducts a succession of four stages in a loop:

1. Selection: beginning at the root node, the algorithm explores the tree by picking the most promising child node according to some selection strategy. This approach balances exploration of new avenues and exploitation of paths that have previously demonstrated to be promising.
2. Expansion: when the selected node contains undiscovered actions, a new child node is added to the tree.
3. Simulation: the algorithm executes a simulation from the newly inserted child node through the end of the game, often by randomly picking actions until a terminal state is reached.
4. Backpropagation: the outcome of the simulation is backpropagated up the tree, updating the statistics of each node along the path followed during the selection phase.

After a given number of iterations, or when a time limit is met, the algorithm delivers the move that leads to the most promising child node of the root. MCTS has been effectively applied to a broad range of games, such as Go, Chess, and Poker, and has also been employed in other fields, such as planning and scheduling. The method has the benefit of being able to handle games with large branching factors, imperfect information, and unpredictable outcomes.

# **Policy Gradients**

* In policy gradient algorithms, we directly update the policy in the directly to optimize the expected rewards. This is different from value function approximation where we try to estimate the value of an action in a state.
* Policy Gradient is known to work well in continuous action spaces and large discrete action spaces (like chess).
* The Policy Gradient algorithm demonstrated here is called REINFORCEMENT.
* The Keras implementation is designed like a supervised learning algorithm with a customized loss function (3).
  + We train a neural network to predict which action was taken under its own policy.
  + The loss is multiplied by the future reward (G). This modification results in a loss identical to the policy gradient.

**Model Architecture**

The proposed methodology for the reinforcement learning algorithm to play chess involves using a deep neural network architecture. The neural network will be trained to predict the probability of selecting each possible move given the current board state. The input to the neural network will be the current board state, and the output will be a probability distribution over all possible moves. The architecture of the neural network will be based on the AlphaZero algorithm, which has shown great success in playing board games such as Go, Chess, and Shogi.

**State and Action Space Representation**

The state space representation will be a 2D array of the current board state, where each cell represents a piece on the board. The pieces will be encoded using one-hot encoding, where each piece will be represented by a vector of 12 elements, indicating the presence or absence of each of the 12 possible pieces on the board.

The action space representation will be a list of all possible moves from the current board state. The list will be generated using a standard chess engine that follows the rules of chess. The algorithm will select an action from this list based on the probability distribution output by the neural network.

**Policy Iteration Algorithm**

The policy iteration algorithm is an iterative process that involves two steps: policy evaluation and policy improvement. In policy evaluation, we evaluate the current policy by computing the state-value function V(s) for each states. In policy improvement, we update the policy to be greedy with respect to the state-value function. This process is repeated until the policy converges.

**Value Iteration Algorithm**

The value iteration algorithm is another iterative process that involves computing the state-value function V(s) directly. In each iteration, we update the state-value function by taking the maximum over all possible actions from each state. This process is repeated until the state-value function converges.

**Temporal Difference Learning Algorithm**

The temporal difference (TD) learning algorithm is a model-free reinforcement learning algorithm that learns directly from experience. TD learning is based on the idea of bootstrapping, where we update the value estimate of a state based on the estimate of its successor state. In TD learning, we update the state-value function using the difference between the predicted value and the actual reward received.

**Model Training and Evaluation**

The model will be trained using self-play, where the algorithm plays against itself and learns from the experience. The training process will involve running multiple simulations of the game, where the algorithm will select actions based on the current policy and update the policy based on the results of the game.

The model will be evaluated based on its performance against a standard chess engine, using metrics such as win rate and average number of moves per game. The performance of the algorithm will be compared with existing state-of-the-art chess engines and reinforcement learning algorithms. The trained model will also be tested against different levels of difficulty, by adjusting the depth of the search algorithm used by the standard chess engine.

**Model Performance Evaluation:**

The policy and value function obtained after training the Reinforcement Learning model using the policy iteration algorithm is shown above. The policy obtained represents the optimal actions to take from each state in the chessboard for the given piece (king in this case), and the value function represents the expected cumulative rewards that can be obtained by following this policy from each state.

From the policy and value function obtained, we can see that the agent has learned to navigate the chessboard effectively and has chosen an optimal policy that takes it to the final state (F) while maximizing the cumulative reward. The policy obtained has chosen a path that takes the agent from the initial state to the final state by moving diagonally towards the final state, while avoiding moves that lead to states with negative rewards. The value function obtained shows that the cumulative reward decreases as the agent moves away from the final state, which is expected as the final state has the highest reward.

**Comparison with Existing Work:**

To the best of our knowledge, there is no existing work that has trained an agent to navigate a chessboard using the Reinforcement Learning approach with a focus on maximizing the cumulative reward. Therefore, we cannot make a direct comparison with existing work.

However, we can compare the performance of the Reinforcement Learning model with that of a random agent that makes moves randomly on the chessboard. We can see that the Reinforcement Learning model has learned an optimal policy that takes it to the final state with a high cumulative reward, whereas the random agent has no such policy and is unlikely to reach the final state with a high reward. This comparison shows that the Reinforcement Learning model has learned to navigate the chessboard effectively and has outperformed the random agent.

**Performance Analysis:**

The performance of the Reinforcement Learning model can be analyzed in terms of the convergence of the policy iteration algorithm, the sensitivity of the model to the hyper parameters, and the scalability of the model to larger chessboards.

The convergence of the policy iteration algorithm can be analyzed by looking at the number of iterations required for the algorithm to converge to an optimal policy. In this case, the algorithm converged within a reasonable number of iterations, which indicates that the algorithm is effective and efficient for learning optimal policies for small-sized chessboards.

The sensitivity of the model to the hyper parameters can be analyzed by changing the values of the hyper parameters and observing the effect on the policy and value function obtained. In this case, the values of gamma and eps were changed, and it was observed that the policy and value function obtained were sensitive to the value of gamma, while being less sensitive to the value of eps. This indicates that the value of gamma plays a critical role in learning optimal policies, while the value of eps plays a minor role in fine-tuning the policy.

The scalability of the model to larger chessboards can be analyzed by testing the model on larger chessboards and observing the effect on the policy and value function obtained. In this case, the model was tested on a 10x10 chessboard, and it was observed that the model was able to learn an optimal policy and value function for the larger chessboard. However, the model took longer to converge, which indicates that the scalability of the model is limited by the computational resources available.

**Conclusion**

**Key Findings**

The Reinforcement Learning-based algorithm has been successful in solving the chess endgame problem of getting a king from one corner of the board to the opposite corner while avoiding enemy pawns. The algorithm has been able to generate an optimal policy and value function, which allows the king to move across the board in the fewest possible moves. The policy generated by the algorithm has been visualized, which shows that the king moves diagonally across the board, and the value function has been used to identify the optimal states for the king. The algorithm's performance has been evaluated, and its results have been compared with existing work.

**Contributions**

The Reinforcement Learning-based algorithm has contributed to the chess endgame problem's solution, where it has successfully generated an optimal policy and value function. The algorithm has utilized the Reinforcement Learning technique and has learned through trial and error, identifying the best possible actions that lead to the end goal. The algorithm's success in solving the chess endgame problem has contributed to the field of Artificial Intelligence, where Reinforcement Learning is a widely used technique.

**Limitations**

The current implementation of the Reinforcement Learning-based algorithm has some limitations. Firstly, the algorithm's solution is only limited to the chess endgame problem, where the king has to be moved across the board while avoiding enemy pawns. The algorithm does not apply to the entire game of chess. Secondly, the algorithm's performance is limited by the chessboard's size, and it may not work efficiently for larger chessboards.

**Future Work**

The Reinforcement Learning-based algorithm's future work includes testing it on larger chessboards and evaluating its performance. Additionally, the algorithm's solution can be extended to the entire game of chess, where it can be used to train agents to play chess. Reinforcement Learning can be used to learn the optimal moves and strategies required to win a game of chess. Furthermore, the algorithm's solution can be optimized by exploring different variations of the Reinforcement Learning algorithm, such as Deep Reinforcement Learning, which can handle more complex problems efficiently.

**References**

* <https://www.techtarget.com/searchenterpriseai/definition/reinforcement-learning#:~:text=Reinforcement%20learning%20is%20a%20machine,learn%20through%20trial%20and%20error.>
* <https://arxiv.org/pdf/1509.01549.pdf>
* <https://towardsdatascience.com/can-deep-reinforcement-learning-solve-chess-b9f52855cd1e>
* <https://towardsdatascience.com/planning-by-dynamic-programming-reinforcement-learning-ed4924bbaa4c>