Reinforcement Learning agent for Ashtachamma game with varying opponent strategies

University of Maryland, College Park

Advait Kinikar
Directory id: kinikara
UID:120097997
kinikara@umd.edu

Hitesh Kyatham
Directory id: hkyatham
UID:120275364
hkyatham@umd.edu

Sumedh Chinchmalatpure
Directory id: sumedhc
UID:120408618
sumedhc@umd.edu

Arden Matikyan
Directory id: amatikya
UID: 117250909
amatikya@umd.edu

Abstract

The main idea behind this project was to investigate, study and model a game agent, which would be robust enough to perform well in dynamic environments and make decisions and actions based on varying opponent strategies. This is important since real world conditions are dynamic in nature and developing such an agent would address some of the limitations that Reinforcement Learning faces, which restricts its use. The game environment is based on the ancient Indian game called Ashta Chamma. Multiple methods of reinforcement learning were looked at and Proximal Policy Optimization and Deep Q Networks were selected for training the algorithm. The reason for selection of these algorithms over simplistic ones such as Q-learning and TD-λ was that, they can handle complex environments with large state spaces like our game much better, they are robust enough to handle stochasticity and randomness, they are stable and can balance exploration and exploitation effectively. After training the agent, it was found that the agent performs better than the baseline, which means, it was "learning" to play the game effectively

1. Introduction

Reinforcement Learning (RL) has caught the eye of many researchers and developers in recent times. This might be because of its ability to learn from experience by trial and error which makes it more appealing over traditional supervised learning algorithms, which often struggle with the lack of supervised data. Furthermore, unlike supervised and unsupervised learning, which focus on static tasks like classification or pattern discovery, RL focuses on sequential decision-making, where each action affects future outcomes, making it more human-like than other algorithms. However, even though RL can handle exploration (trying new actions) and exploitation (choosing the best-known actions) well, one of its limitations is that it is not robust enough to adapt to changing conditions in its environment. For example, an RL agent may not be able to adapt to varying strategies its opponents may use in a game. This is interesting because in dynamic game environments, agents often face opponents with diverse strategies and shifting conditions, which mirrors any real-world scenario and creating adaptive agents can significantly improve performance and decision-making in real-time. In this project, we would attempt to investigate this very problem i.e. how to model a game agent using reinforcement learning (RL) to adapt to varying strategies. Varying strategies are introduced in terms of a: random, aggressive and defensive player. Each opponent type introduces unique challenges, from unpredictable, random moves to targeted, aggressive actions. We would use Proximal Policy Optimization (PPO) and Deep Q Networks (DQN) algorithms, to learn how the RL agent would adapt to different moves and respond to varying strategies. We would use win-rate as our evaluation metric for the algorithms and we would measure the agent's success using performance of a random-move player as the baseline. However, other metrics such as reward accumulated in each episode of evaluation, training loss, explained variance, policy gradient loss, value loss, entropy loss etc. would also be used as metrics of evaluation. This allows us to analyze the effectiveness of PPO and DQN in making the agent "learn" to handle dynamic play styles and strategic variation. The environment in which the agent would play would be the game of '*Ashtachamma*'.

After training, evaluating and testing the agent for both PPO and DQN, we found that, although the agent doesn't have a significantly high win rate, it still has more wins than the random player. Having more wins than the random player means that the agent is performing better than the baseline. Furthermore, from the plots mentioned

above, it could be seen that the agent was accumulating rewards and the loss were reducing, which means the agent was able to learn and adapt to the game, the different strategies that other players play with and the general randomness and stochasticity involved and then make appropriate moves. This marks the success of our project.

2. Literature Review

The development of reinforcement learning (RL) agents for strategic board games like Ashta-Chamma could be of significant interest due to the dynamic and multi-player interactions involved in such games. Ashta-Chamma, similar to Ludo, involves strategic decision-making, cooperation, and competition among multiple agents. In this literature review, we would explore, the relevant RL techniques and their applicability to Ashta-Chamma, focusing on Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN), while comparing them with classical methods like Q-learning and Temporal Difference (TD) learning.

PPO has emerged as a robust policy gradient method for RL, offering stability and sample efficiency. Schulman et al. [1] introduced PPO with a clipped surrogate objective to ensure reliable policy updates without overfitting to recent experiences. Its adaptability to stochastic (random) environments and high-dimensional state spaces makes it a strong candidate for games like Ashta-Chamma, where dynamic board states and varying opponent strategies are prevalent. Yu et al. [2] demonstrated PPO's effectiveness in cooperative multi-agent games, highlighting its ability to handle both cooperative and competitive dynamics with minimal hyperparameter tuning. In Ashta-Chamma, where player interactions with each other and their behaviors are crucial, PPO's robustness and scalability make it a practical choice.

DQN extends traditional Q-learning by approximating the action-value function with deep neural networks, enabling it to handle large and complex state spaces. Van Hasselt et al. [3] addressed the overestimation bias in standard Q-learning by introducing Double Q-learning, which improves the accuracy of Q-value predictions. Fan et al. [4] provided theoretical justifications for DQN's stability through techniques like experience replay and target networks. In Ashta-Chamma, where state transitions are dynamic and influenced by stochastic events like dice rolls, DQN's ability to generalize value functions across diverse states makes it a viable method for training agents.

Next, we researched learning techniques, particularly Temporal Difference (TD) learning in [5], which has been a subject of intense research since Tesauro's TD-Gammon achieved remarkable success in Backgammon. TD learning is a method that enables an agent to learn from the environment by making predictions about future rewards and adjusting those predictions based on actual outcomes. Unlike other methods that wait until the end of an episode to update values (like Monte Carlo methods), TD learning updates estimates at each step, immediately after each action. Now, TD learning, as explored by Tesauro in TD-Gammon, is effective for board games with smaller state-action spaces. However, it struggles with the high variance and complexity present in multi-agent environments like Ashta-Chamma. Similarly, while Q-learning is a foundational RL method, its tabular approach becomes computationally infeasible in large state spaces. Its variants like Strategies-Oriented Q-tables, which incorporate game-specific heuristics, offer some improvement but lack the generalization capabilities of deep RL methods.

PPO and DQN address these limitations effectively. PPO's policy-based approach is particularly advantageous in dynamic environments where exploration-exploitation trade-offs are critical. As shown by Rolf et al. [6], DQN excels in handling complex decision-making tasks by using experience replay to decouple state transitions and stabilize learning. In the context of Ashta-Chamma, where strategies must adapt to opponent behavior and stochastic events, PPO offers policy stability, while DQN provides value approximation for optimal decision-making.

Galway et al. [7] highlighted the importance of adaptive and learning-based strategies in games with dynamic interactions. Ashta-Chamma requires agents to balance aggression (killing opponent pawns), defense (avoiding risky moves and prioritizing safe places), and progress toward victory. Hybrid strategies that adapt based on game states have been shown to yield the most balanced gameplay. Self-play, as suggested in [8], can further enhance learning by exposing agents to diverse strategies, improving their resilience in multi-agent environments.

Thus, the combination of PPO and DQN provides a robust framework for addressing the strategic depth and dynamic interactions in Ashta-Chamma. While classical methods like TD learning and Q-learning offer foundational insights, their limitations in scalability and adaptability are evident. By integrating advanced techniques like experience replay and clipped surrogate objectives, PPO and DQN enable agents to effectively

navigate complex state spaces and dynamic scenarios. This hybrid approach is well-suited for the strategic nature and multi-player interactions involved of Ashta-Chamma, promising improved performance and adaptability in training RL agents for such games.

3. Data/Environment

Following, is the brief description of the environment, our agent, would play in. It involves the description of the board, different player strategies and the moves that they would make.

3.1. Problem Statement

The objective of this project is to develop a reinforcement learning (RL)-based agent capable of adapting to diverse strategies employed by opponents in the game of Ashta-Chamma. This endeavor addresses a critical limitation of traditional RL approaches, which often struggle to adapt effectively to dynamic and unpredictable environments.

The problem is inherently multi-faceted, requiring an in-depth understanding of the game environment, the various strategies opponents may employ, and appropriate metrics for evaluating agent performance. Additionally, establishing a baseline for comparison is essential for assessing the effectiveness of the proposed solutions.

3.2. Environment

The game consists of 3 elements namely, the board, play pieces and the special dice

3.2.1 Board –

Size and Shape: The board typically consists of a 5x5 grid with a total of 25 squares, although variations can be 7x7 or larger depending on the region and tradition. But in this project, we consider a 7x7 grid board

Starting and Home/Win Squares: Each player has a starting zone and a home zone (goal area). The objective is to navigate from the starting zone, around the board, and into the Home/Win zone. The Home/Win zone, is usually, at the center of the grid. In our game, the Home/Win zone is at the position (4,4). The staring positions are:

Player 0 (Agent): (0, 4), (1, 4)
Player 1 (Random): (4, 0), (4, 1)
Player 2 (Defensive): (8, 4), (7, 4)
Player 3 (Aggressive): (4, 8), (4, 7)

Here, the first tuple is the coordinates of the first pawn for a player and the second tuple is the coordinates of the second pawn.

Safe Zones: Some squares are designated as safe zones where other tokens cannot be captured. In our game, there are 9 safe zones. They are located at the following positions:

(1, 4), (2, 2), (2, 6), (4, 1), (4, 4), (4, 7), (6, 2), (6, 6) and (7, 4)

3.2.2 Pieces –

Each player has four tokens. The aim is to move all four tokens from the starting area, around the board, and into the home column. The tokens move based on the roll of cowry shells or special dice, with the number of shells opening upwards indicating the number of spaces to move. Now, for the sake of reducing the complexity of the game and making it a bit easier for the agent to learn, while training, evaluation and testing, we used only 2 pawns or pieces. Even when using only 2 pawns, it took us almost 3 hours to train and evaluate our agent. With 4 pawns, it would have been significantly more.

Special dice/cowry shells –

Roll Outcome: Typically, 1-4 cowry shells are used as dice. The number of shells that land with the opening facing up determines the roll result.

Special Rolls: Some rolls may allow a player to take another turn or have other special rules, such as skipping opponent's tokens or moving extra spaces.

3.2.3 Rules –

The player moves the tokens based on the output of the dies the possible outcomes of the dies could be 1,2,3,4,8 houses or squares, the moves can be applied to one single token or multiple tokens as the player wishes. If player makes a move which ends up on the same square as the opponent token the opponent token is considered captured and should return back to the home position of the opponent. The game is won when all the player token reaches the goal position. For the sake of simplicity, in our code, we programmed to randomly select a number from 1-6, to mimic dice roll. Furthermore, Ashtachamma has a rule, wherein, a player cannot start unless, the player rolls a 1,4 or 8. We did not implement this, since, it increases the randomness in the game and when we tried to implement this, the penalties accumulated by the agent were too high and it was getting difficult for the agent to learn. Additionally, there is one more rule wherein, if a player "Kills" another player's

pawn, the player gets another chance to roll. We did not implement this, since, it gave an advantage, to aggressive type players, which made the agent more focused on them, ignoring the defensive type players. Hence, the agent was struggling to play against defensive type players.

3.3. Agents

Following are the types of agents or players or opponents that would be in the game. Each has its own strategy to play the game. It is inspired from what is described in [9].

3.3.1 Random Agent

A Random agent employs a strategy where moves are chosen entirely at random, without any consideration for strategy or game advancement. This type of agent is used as a baseline in this project, allowing researchers and developers to measure the effectiveness of more strategic agents by comparing their performance against a purely random approach.

3.3.2 Aggressive Agent

An Aggressive agent focuses on offensive strategies, specifically choosing to move pieces that can eliminate or "knock out" or “Kill” opponents' pieces based on the dice results. This aggressive tactic forces the attacked player to restart with the affected piece, effectively setting them back in the game. While an aggressive strategy aims to provide a tactical advantage by reducing the number of opponent pieces on the board, it may blend with other strategies like random moves when no direct attack opportunities are present.

3.3.3 Defensive Agent

A Defensive agent prioritizes the safety of its pieces, always moving to protect them from potential attacks. It considers a piece to be in danger if it is within a "knocking range," defined as being within one dice roll's distance (1-6 squares) from an opponent's piece. Defensive moves aim to avoid losses by moving endangered pieces out of reach of opponents, thereby maintaining a strong position on the board. The defensive player, would prioritize, moving into a safe square, over killing an opponent’s pawn, if ever such condition, comes forth. This strategy focuses on long-term preservation and positioning over immediate offensive gains.

3.4. Evaluation Metrics and Baseline

The evaluation metric chosen for assessing the performance of the reinforcement learning (RL) agent is win rate, which measures the percentage of games the agent wins out of the total games played. This metric is closely connected to the baseline, which is the performance of an agent that makes random moves. The win rate is calculated as the ratio of the number of games the RL agent wins to the total number of games it plays, expressed as a percentage. The baseline performance is established using an agent that performs random actions. This means that the baseline agent has no strategy or learning algorithm applied to it—it simply chooses random moves at each step. The strategies could be killing the opponent’s pawn over going to a safe zone etc. Furthermore, to really understand whether the agent is actually learning or not, plots such as reward, training loss, explained variance, policy gradient loss, value loss, entropy loss for PPO and reward accumulated and training loss for DQN would be studied. If the plots suggest that the agent is successful in accumulating rewards and the losses, while training are reducing and then stabilizing, then it can be said that the agent is learning to play the game, with opponents of varying strategies.

As discussed before, RL agent, will be trained to play the game of Ashtachamma against various opponents with different strategies (random, aggressive, defensive). The RL agent will play a set number of games against each type of opponent. For each game, the agent will take actions based on its learned policy and receive rewards or penalties based on the outcome. After the RL agent has completed its training and has played a set of games against each opponent, its win rate is calculated. For example, if the agent won 60 out of 100 games against a random-move opponent, its win rate would be 60%. The win rate of the random-move agent (baseline) is also calculated. The performance of both i.e. the win rate would be compared. If the RL based agent has a higher win rate than the random-move agent, it would be concluded that the algorithm has worked well. The evaluation will also highlight how well the RL agent adapts to different strategies. For example, the agent might perform better against certain types of opponents (like the random player) but struggle against more sophisticated strategies (like the aggressive player). This can provide insights into how the agent’s learning algorithms handle varied strategies and how adaptable they are in dynamic environments. This and the plots would be the only metric under consideration under now, as per the mentor (TAs) suggestion, since, the problem statement is quite new and much work has not been done on it.

However, as future scope, once the RL agent achieves good results in terms of win rate, evaluating it by using move efficiency as a metric, which assesses the quality of each move by considering how many optimal moves are made during a game could be thought of.

4. Methods

Following, would be a brief description of how the 2 methods we implemented i.e. PPO and DQN, work. It would be followed by, an explanation of our actual implementation i.e. How we created the board? How we designed players with different strategies? How we trained the agent using the said methods? etc.

4.1. DQN

DQN is a value-based RL algorithm. Its goal is to learn a Q-value function, which estimates the expected total reward for taking an action '*a'* in a state *'s'* and following the best policy afterward. Q-Learning uses a Q-table to represent the action-value function *Q(s, a)*. For large state-action spaces, storing a Q-table becomes infeasible, so DQN replaces it with a neural network that approximates *Q(s, a)*. The neural network takes the state s as input and outputs Q-values for all possible actions a. The action with the highest Q-value is selected as the best action.

During training, the agent interacts with the environment and stores transitions *(s, a, r, s')* in a replay buffer. The neural network is trained on random minibatches of these transitions, breaking correlations in the data and stabilizing learning. DQN uses two networks: the Q-network (current policy) and the target network (provides stable Q-value targets). The target network is updated less frequently, which reduces instability.

Now, the Q-value is update using the Bellman Equation:

$$Q(s, a) = r + \gamma \cdot max\ Q(s', a)$$

Here, *r* is the reward for taking action *a* in state *s*, *s'* is the next state and γ is the discount factor. It decides how much preference must be given to accumulate future rewards in future states *s'*, while taking current actions *a.*

The loss function minimizes the difference between the predicted Q-value and the target Q-value. It is given as:

$$Loss = (Q(s, a) - target)$$

For the exploration-exploitation trade-off, DQN uses an ϵ-greedy policy, where the agent explores random actions with probability ϵ and exploits the learned Q-values otherwise

4.2. PPO

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm designed to train agents to make decisions in an environment while maximizing long-term rewards. Unlike older RL methods, PPO doesn't focus on estimating how good an action is (like in Q-learning). Instead, it directly learns a policy, which is a strategy that maps states to actions

PPO represents the policy $\pi(a, s)$, which gives the probability of taking an action a when the agent is in state s. A neural network is used to approximate this policy. The network's goal is to maximize the total reward by choosing better actions over time.

PPO also uses an advantage function *A(s, a)* to determine how good an action a is compared to the average action in the same state s. It is calculated as follows:

$$A(s, a) = Q(s, a) - V(s)$$

Here, *Q(s, a)* is the total reward if action *a* is taken in state *s*, *V(s)* is the expected reward for being in state *s*, regardless of action. If *A(s, a)* is greater than zero, then the action is better than average. If *A(s, a)* is less than zero, then the action is worse than average.

PPO updates the policy using a policy gradient, which encourages the agent to take actions with higher advantages and discourages weaker ones.

A key innovation of PPO is its clipped objective function, which prevents the policy from changing too much in one update. If there is no clipping, the agent might make large, unstable changes to its strategy and forget what it learned earlier. PPO uses a ratio *r(θ)*, which compares the new policy π_θ to the old one π_{old}

$$r(\theta) = \frac{\pi_\theta}{\pi_{old}}$$

Finally, we calculate the objective function *L(θ)*. If the policy changes too much (outside the range $1-\epsilon$ to $1+\epsilon$), the advantage is clipped to keep updates stable.

$$L(\theta) = min(r(\theta).\ A(s, a),\ clip(r(\theta),\ 1-\epsilon, 1+\epsilon).\ A(s, a))$$

PPO adds an entropy bonus to encourage exploration. Entropy is a measure of randomness in the agent's actions. A higher entropy means, the agent tries different actions (exploration). A lower entropy means, the agent sticks to current actions (exploitation). Early in training, exploration is prioritized to learn about the environment. Later, the agent focuses more on exploiting the best strategies.

PPO uses the same batch of experiences multiple times to improve learning efficiency. This makes PPO sample-efficient compared to older methods.

4.3. Comparison between DQN and PPO

Following, is a comparison between DQN and PPO

	DQN	PPO
Type	Value-based	Policy-based
Policy	Deterministic (chooses max Q-value)	Stochastic (chooses highest probability)
How does it stabilize?	Experience replay/target network	Clipped objective
Computation Exploration vs Exploitation	Lighter ϵ-greedy	Higher Stochastic
Action space	Discrete	Discrete or continuous
Learning focus	Q-value approximation	Direct policy optimization
Environment type	Complex	Complex

Table1. Comparison between DQN and PPO

From the above table, it is evident that each algorithm has its own advantages and limitations, but still, they are better for training an agent for a complex game environment like Ashtachamma than regular RL algorithms

4.4. Reward/Penalty Structure

Following is the reward/penalty structure; we have used for the agent:

Progress towards win → **1.5** → Encourages the agent to make moves that bring pawns closer to the goal and prevent the agent from wandering aimlessly

Reaching the win position → **5** → Rewards the agent for successfully bringing a pawn to the central square (4,4) and keeps the agent focus on the ultimate goal

Winning the game → **10** → Rewards the agent significantly for achieving the primary objective, encouraging strategic planning

Killing opponent pawn → **2** → Rewards the agent for eliminating competition, creating opportunities to secure the board

Entering Safe zone → **1** → Encourages the agent to move pawns to safe squares, minimizing the risk of being killed by opponents.

Staying idle → **-0.1** → Penalizes inaction or unnecessary waiting, ensuring that the agent remains active and engaged in gameplay

making risky moves → **-0.8** → Discourages the agent from making moves that unnecessarily place pawns in danger (e.g., near opponent pawns)

invalid pawn selection → **-1** → Penalizes the agent for selecting pawns that cannot be moved legally, forcing the agent to adhere to the rules.

The above reward structure is good because, it creates a balance between offensive and defensive strategies the agent might use, making the agent effective in a dynamic gameplay. It also aligns with the game's objectives of winning, facilitates learning and discourages counterproductive behavior.

4.5. Implementation

As discussed above, DQN and PPO were used to train the agent to play the game with opponents using varying strategies. Before training the agent, the environment was created, the different actions that player with each strategy (defensive and aggressive) were decided. The board was first programmed, then the strategies for each player were programmed and then final a test play (without the agent) including the random, offensive and defensive player was programmed and run. After observing the test play and confirming that all the players are playing as they are supposed, an environment, based on the board and the strategies previously programmed was made using OpenAI Gymnasium library for training the RL agent. Then in another script, the environment was imported and the RL agent was trained. The entire flow can be seen in Figure1.

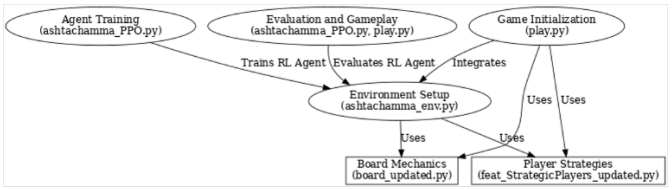

Figure1. Flow of the entire project

As seen in figure, the game is first initialized in "play.py". It uses the board programmed in board_updated.py and the player strategies from feat_strategic_players.py. The graphical interface for the game is obtained by using 'Pygame'. The RL environment is setup in 'ashtachamma_env.py' and the agent is trained in 'ashtachamma_ppo.py'. For DQN, the structure is same, just there is another script 'ashtachamma_dqn.py'.

The board_updated.py script manages the core

mechanics of the Ashta-Chamma game, including defining the game board, player paths, safe zones, and home squares. It handles dice rolls, pawn movements, and special interactions like killing opponents and checking win conditions. Figure 2. shows the board generated by 'board_updated.py'.

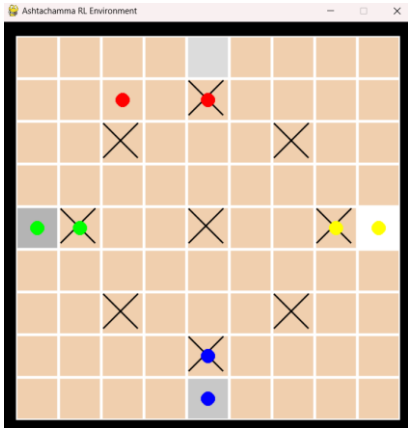

Figure2. Board generated in board_updated.py

The feat_StrategicPlayers_updated.py script implements various player strategies—aggressive, defensive, and random—allowing different decision-making behaviors that influence the flow of the game. The play.py script is responsible for the main gameplay loop, simulating moves for both human and scripted agents based on their strategies, and rendering the game state using Pygame. The ashtachamma_env.py defines the reinforcement learning (RL) environment, structuring it for use with Stable-Baselines3, and it manages game states, rewards, transitions, and turn-based mechanics for RL agents. Finally, ashtachamma_PPO.py trains a reinforcement learning agent using Proximal Policy Optimization (PPO), configuring the model, logging training progress, and evaluating the trained policy for autonomous gameplay.

We used Stable-Baselines3 – a library for RL from PyTorch for implementing both DQN and PPO, as it provides robust and easy-to-use implementations of reinforcement learning algorithms. Stable-Baselines3 abstracts the complex mechanics involved in these algorithms, such as managing replay buffers and target networks for DQN or implementing the clipped surrogate objective for PPO. Additionally, it provides flexibility in controlling hyperparameters like learning rates, exploration strategies, batch sizes, and clip ranges, enabling fine-tuned training for specific environments.

For DQN, Stable-Baselines3 efficiently handles the mechanics of experience replay and maintains both the main and target networks, which are critical for stable training. It also allows customization of hyperparameters, such as:

A. Exploration Fraction (**exploration_fraction**): Determines the fraction of the total training process during which the agent transitions from fully random actions to more deterministic ones.
B. Initial & Final Exploration Rates (**exploration_initial_eps /exploration_final_eps**)**:** control the epsilon (ϵ) value in the epsilon-greedy policy, dictating the probability of taking random actions.
C. Learning rate: Controls how quickly the model adapts to new information.
D. Buffer size: Determines how many past experiences are stored for training.
E. Exploration strategy: Configures how the agent balances exploration and exploitation during training.

For **PPO**, Stable-Baselines3 simplifies the implementation of policy optimization by managing the clipped surrogate objective and multiple epochs of minibatch updates. It also supports crucial hyperparameters such as:

A. Clip range: Limits the changes to the policy during updates, ensuring stability.
B. Learning rate: Adjusts the step size during optimization.
C. Batch size and number of epochs: Configures the number of updates per data sample to improve sample efficiency.
D. Entropy coefficient: Balances exploration by adding randomness to the policy.

Stable-Baselines3 integrates seamlessly with custom Gym environments, making it straightforward to train agents for the Ashta-Chamma game. Its modularity and flexibility ensure that both DQN and PPO can be tailored to the specific requirements of the game environment, facilitating efficient and effective learning.

5. Experimentation and Results

We performed multiple experiments for each algorithm. The agent was trained, evaluated and tested with PPO for almost 32 times and for 19 times with DQN. We got the following results. As discussed above, we performed the experiments with only 2 pawns, because, it was taking significant too much time to train the agent (about 3 hours for 2 pawns). With 4 pawns, the action space and the state-space would have become even more complicated and hence would have resulted in more time to train. Our main objective was to train the agent, so that it learns the game and plays well against other players (aggressive and

defensive) and better than the random player. This could be observed with only 2 pawns as well. Following were our observations for both PPO and DQN. It should be noted that all the plots were obtained by using 'tensorboard'

5.1. PPO:

Here the reward structure was the same as above. Here, the agent uses, the 'MlpPolicy' i.e. a multi-layer perceptron to make decisions. The learning_rate controls how much the model adjusts during training—smaller rates ensure steady learning without drastic changes. The 'n_steps' parameter specifies how many game moves are collected before the model updates, balancing between frequent updates and having enough data for learning. 'batch_size' determines how much data is processed at once during training, while 'n_epochs' indicates how many times the same batch is reused to improve learning. gamma focuses on how much the agent values future rewards, with a value close to 1 prioritizing long-term gains. 'gae_lambda' helps balance stability and accuracy in estimating rewards. The 'clip_range' prevents the agent from making overly large updates to its policy, ensuring stable progress. Lastly, 'ent_coef' encourages exploration by adding randomness to the agent's actions, helping it discover better strategies.

Of all the 32 tests, 1 is explained in this report (all results cannot be shown, but they are improving over time). Following were the hyperparameters set for th experiment. The hyperparameters were found using 'Optuna'. It uses a process called *'automated hyperparameter tuning'*, where it intelligently tests different combinations of hyperparameters to find the best ones for your model. Optuna works by creating a "study" where it tries out different values for parameters like learning rate, batch size, or clip range, guided by algorithms that focus on finding the optimal settings quickly.

Policy: MlpPolicy, **learning_rate:** 1.4401226335484468e-05, **n_steps** = 6144, **batch_size** = 128, **n_epochs** = 10, **gamma** = 0.96, **gae_lambda** = 0.9500000000000001, **clip_range** = 0.35, **ent_coef** = 0.0004724873659303969, **timesteps** = 7500000, **n_evals** = 500

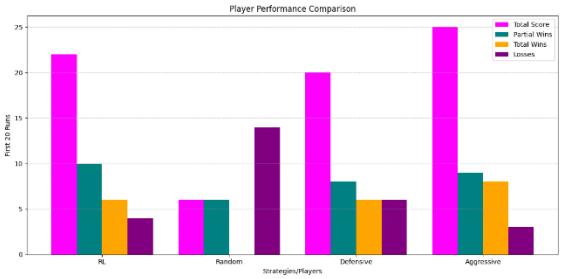

Figure3. Win/Loss Plot for all the Players

Figure3. shows all the win/loss statistics for all the players. A player is said to have a complete win, if the player has 2 (all) pawns in the win position. A player is said to have a partial win if the player has 1 pawn in the win position. The total score is the sum of all complete and partial wins (1 point for partial win and 2 points for complete win). Losses means that the player did not have any pawns in the win position in a particular game.

Now, it can be seen that the aggressive player has the most impressive statistics (Total score: 25, No of partial wins: 9, No of complete wins: 8, No of losses: 3). The agent has the statistics (Total score: 22, No of partial wins: 10, No of complete wins: 6, No of losses: 4), which is better than the statistics of random player (Total score: 6, No of partial wins: 6, No of complete wins: 0, No of losses: 14). Here, the RL agent is better than the random player in every aspect. This satisfies our baseline condition of the agent playing better than the random player. Infact, here it is playing as good as the defensive player (Total score: 20, No of partial wins: 8, No of complete wins: 6, No of losses: 6) with win rate of 33%, showing that it is utilizing some strategy.

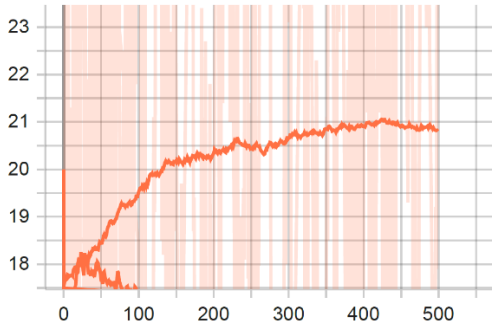

Figure4. Episode Reward for the agent

Figure7. is a plot showing the rewards accumulated by the agent over the number of evaluation episodes. The trend is increasing over time. An increase reflects the agent learning to maximize its rewards by improving its strategy. If the plot shows a steady rise, the agent is effectively learning the task. If the reward plateaus, the agent may have converged or might require additional exploration to refine strategies.

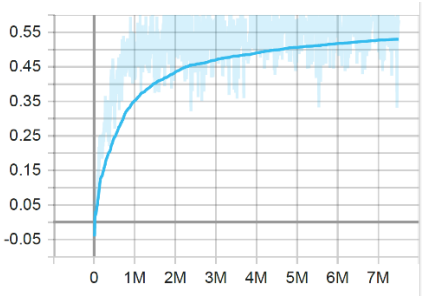

Figure5. Explained variance for the agent

Figure5. is a plot showing the explained variance over the number of evaluation episodes. The trend is increasing towards over time. An increasing trend suggests the that the predictions are improving. Explained variance, evaluates the reliability of the value function. High explained variance (>0.5) means the value network accurately predicts rewards.

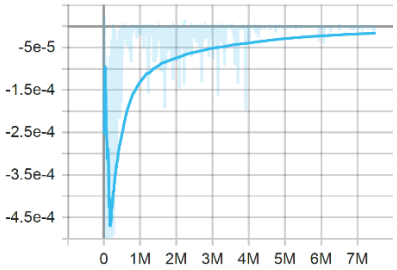

Figure6. Policy gradient loss for the agent

Figure6. is a plot showing the policy gradient loss over the number of evaluation episodes. The policy gradient loss graphs show a clear learning progression in the reinforcement learning model. Starting from a negative loss value , the model experiences a sharp initial improvement in the first million episodes. After this initial phase, the loss gradually converges toward zero, with consistent oscillations that indicate ongoing learning and policy refinement.

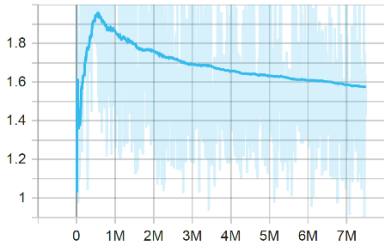

Figure7. Training loss for the agent

Figure7. is a plot showing the training loss over the number of evaluation episodes. The training loss should decrease over time. It can be seen that the loss is gradually decreasing. If the train loss decreases steadily, the optimization process is working as expected.

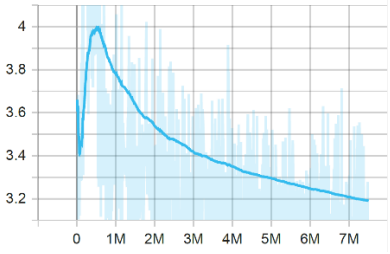

Figure8. Value loss for the agent

Figure8. is a plot showing the value loss over the number of evaluation episodes. The trend is supposed to decrease over time. A consistent decrease in value loss shows the agent is learning to predict future rewards effectively. It can be seen that the value loss is decreasing over time.

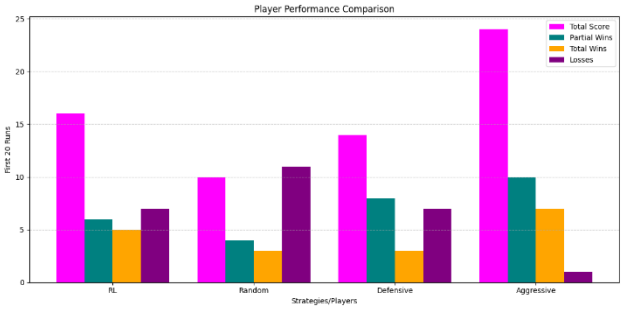

Figure9. Win/Loss Plot for all the Players

Figure9. shows the win loss plot from another experiment. Even here it can be seen that the agent (Total score: 16, No of partial wins: 6, No of complete wins: 5, No of losses: 7) is performing better than the random player (Total score: 10, No of partial wins: 4, No of complete wins: 3, No of losses: 11) and as good as the defensive player. (Total score: 14, No of partial wins: 8, No of complete wins: 3, No of losses: 7)

Thus, from the above win/loss statistics and the metric plots, it can be inferred that the agent is learning to play the game and is performing better than the baseline (better than random player) after being trained using PPO

5.2. DQN

DQN shares the similar hyperparameters and reward structure. As discussed earlier there were about 19 experiments performed using DQN.

Trial	# Pawns Per Player	Total Timesteps	Exploration Fraction	Exploration Start ε	Exploration End ε
DQN12	2	1M	0.1	1.0	0.05
DQN4	2	1M	0.3	1.0	0.05
DQN3	2	1M	0.2	1.0	0.05
DQN15	2	5M	0.2	1.0	0.05
DQN17	4	1M	0.2	1.0	0.05
DQN19	4	5M	0.2	1.0	0.05

Figure10. Statistics for 5 experiments

Figure 10. shows the statistics for 5 experiments performed using DQN. It shows the no of pawns, timesteps, Exploration fraction, Exploration start value and exploration end value. It should be noted that experiment 17 and 19 were performed using 4 pawns. They yielded positive results (agent performed better than random player), but took significantly large time to be computed. With 2 pawns although it took a long time, it was still less than with 4 pawns.

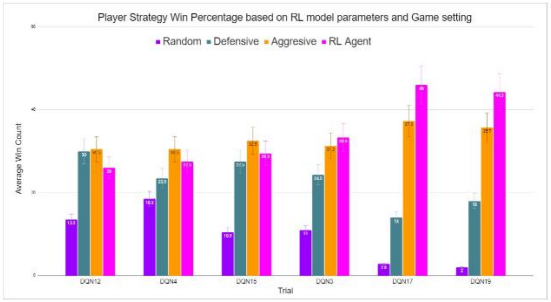

Figure11. Win count for all the players

Figure 11. is the win plot for each player from experiment 12 to 19. It can be seen that the RL has significantly higher win rates than the random player in all experiments.

Figure12. Training loss for DQN_17

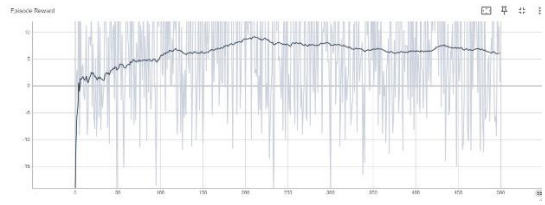

Figure12. Accumulated rewards for DQN_17

Now, we can see the environment and model for DQN_17, the highest win rate was achieved out of all trials. Furthermore, the gradual increase and then stability in the accumulated rewards plot Figure 12. shows that the agent is effectively learning. The training loss Figure13. Is similar, wherein in the loss first decreases then stabilizes. DQN_17 has the greatest loss. This is likely because DQN_17 was run on an environment where each player had 4 pawns, as opposed to 2. With this change, the state space and action space became larger, allowing for more decisions. With more pawns, there were more opportunities for the agent to make decisions that lead to rewards or penalties. Because of this the agent was actually able to learn more and perform better, despite showing a greater loss function.

Thus from the results obtained from both PPO and DQN, it is safe to say that the agent effectively learned to play the game and performed better than the baseline condition i.e. better than the random player. Infact, it performed as good as the defensive and aggressive player showing that it is using a balanced strategy.

6. Limitations

Despite the project's successes in developing a reinforcement learning agent for Ashta Chamma, several limitations were identified. The computational intensity of training, especially with 4 pawns, restricted the number of experiments possible. The agent faced a limited variety of opponent strategies, potentially constraining its real-world applicability (only aggressive, defensive and random). Although outperforming the random player, the agent's win rate did not significantly surpass opponents like aggressive player, indicating room for improvement. Scalability challenges emerged when increasing from 2 to 4 pawns, suggesting potential difficulties with more complex variants. Lastly, the lack of explainability in deep reinforcement learning models made it challenging to interpret the agent's decision-making process, restricting insights into its learned strategies

7. Conclusion

The project successfully developed a RL agent for playing Ashta Chamma, demonstrating significant adaptability to varying opponent strategies. Utilizing Proximal Policy Optimization (PPO) and Deep Q-Network (DQN) algorithms, the agent consistently outperformed the baseline random player, showcasing improved performance and adaptive gameplay. Analysis of key metrics, including accumulated rewards and various loss functions, revealed clear learning trends and decision-making improvements over time. Notably, the agent's performance was comparable to both aggressive and defensive players, indicating the development of a balanced strategy. Experiments with different numbers of pawns demonstrated the agent's ability to handle increased complexity, albeit with longer training times for larger state spaces. These results validate the effectiveness of advanced RL techniques in complex, multi-agent game environments and underscore the potential for developing adaptive AI agents for strategic board games

8. What is different in our project? /Above and beyond

The project is distinguished by its innovative application of reinforcement learning to a complex, dynamic environment modeled after the ancient Indian game Ashta Chamma. While RL agents have been extensively studied in various game environments, developing a robust RL agent capable of playing against multiple strategies—random, defensive, and aggressive—sets this project apart. The ability of the RL agent to adapt dynamically to diverse strategies is a significant and novel contribution. This

versatility reflects real-world scenarios, where unpredictability and variation in behavior are commonplace.

Additionally, implementing an RL agent specifically tailored for Ashta Chamma is a unique aspect of this work. The game's mechanics, enhanced by elements such as safe zones, strategic risk assessment, and reward structures for specific actions like killing an opponent's pawn or entering safe zones, create a challenging and dynamic environment. The use of Proximal Policy Optimization (PPO) and Deep Q Networks (DQN) further exemplifies the project's creativity, as these advanced algorithms enable the agent to effectively learn and adapt in stochastic and high-dimensional state spaces.

Moreover, the use of performance metrics like win rate, rewards, and losses, coupled with visual analytics using TensorBoard, provided valuable insights into the agent's learning process. By achieving better-than-baseline performance against varying opponent strategies, the project demonstrated the agent's ability to "learn" and respond effectively to dynamic challenges. These aspects make the project not only academically significant but also creatively outstanding.

9. Contribution and Acknowledgement:

Each member of the group, equally contributed to the project. Hitesh programmed the game mechanics, while Arden and Sumedh worked on developing player strategies. Advait programmed the play.py script, wherein the game was initially tested. Then all members contributed in creating the RL game environment and the models (ashtachamma_env and Ashtachamma_ppo, ashtachamma_dqn respectively). Then the group was further divided into 2, in which each group worked on a model (training, evaluation and testing), to get the desired results

In the development of this project, some assets and images for the Ashta Chamma board were sourced from a GitHub repository. This decision allowed us to ensure a visually appealing and functional design for the project, aligning with the standards required for our experiments. By utilizing publicly available, high-quality assets, we focused more effectively on the development and optimization of the reinforcement learning (RL) agent, instead of spending excessive time creating visual elements from scratch. Furthermore, this approach demonstrated efficient resource utilization while adhering to appropriate licensing and attribution requirements, a critical consideration in academic work.

Nomenclature:

a. RL: Reinforcement Learning
b. AC: Ashtachamma
c. PPO: Proximal Policy Optimization
d. DQN: Deep Q Networks
e. TD-λ: Temporal Difference λ
f. TA: Teaching Assistant
g. AI: Artificial Intelligence

References

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

[2] Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A. and Wu, Y., 2022. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, *35*, pp.24611-24624.

[3] Van Hasselt, H., Guez, A. and Silver, D., 2016, March. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 30, No. 1).

[4] Fan, J., Wang, Z., Xie, Y. and Yang, Z., 2020, July. A theoretical analysis of deep Q-learning. In *Learning for dynamics and control* (pp. 486-489). PMLR.

[5] Matthews, Glenn & Rasheed, Khaled. (2008). Temporal Difference Learning for Nondeterministic Board Games.. 800-806. A. Alpher. Frobnication. Journal of Foo, 12(1):234–778, 2002.

[6] Rolf, Benjamin & Jackson, Ilya & Müller, Marcel & Lang, Sebastian & Reggelin, Tobias & Ivanov, Dmitry. (2022). A review on reinforcement learning algorithms and applications in supply chain management. International Journal of Production Research. 61. 10.1080/00207543.2022.2140221.

[7] Galway, L., Charles, D. & Black, M. Machine learning in digital games: a survey. *Artif Intell Rev* **29**, 123–161 (2008). https://doi.org/10.1007/s10462-009-9112-y

[8] F. Alvi and M. Ahmed, "Complexity analysis and playing strategies for Ludo and its variant race games," 2011 IEEE Conference on Computational Intelligence and Games (CIG'11), Seoul, Korea (South), 2011,

[9] https://github.com/jaya-shankar/Ashta-Chamma_Board_Game.git