Reinforcement Learning for Cards: Durak

Student Name: Nicholas Antony Crampton

Supervisor Name: Peter Davies

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract — This paper presents a comprehensive exploration of the application of Reinforcement Learning to the strategic card-playing game Durak. We meticulously investigate the capabilities of two RL models, Q-Learning and Deep Q-Networks, within a designed Durak environment purposefully built to emulate Durak's mechanics and flow. The Q-Agent demonstrates significant strategic development, exhibiting thinking ahead, and adaptability to different opponents. Due to its generalized state space, it familiarized itself with every position following an exhaustive training process. Interestingly, the DQN-Agent showed less effective performance, suggesting a need for further refinement in state space complexity and neural network design. The findings in this paper illustrate a balance between a model sophistication and learning efficiency is essential, and sometimes, simpler models yield more impressive results. This study contributes to RL knowledge by investigating a less explored environment in Durak, leaving the door open for future development.

1 Introduction

Artificial intelligence has seen significant advancement and has been at the forefront of science and academia. One field of AI is Reinforcement Learning (RL), where agents make informed decisions by learning from interactions within their environment. RL has been utilized in multiple facets, such as autonomous vehicle navigation and the finance industry, optimizing risk/reward in trading. RL has profoundly impacted the gaming domain, a space rich with complex strategic environments. Games that utilize a standard deck of playing cards involve a mixture of chance and skill, requiring agents to develop strategies to handle uncertainty and adapt to new situations.

In exploring RL's application to card games, this project concentrates on Durak, a popular card game originating in Russia with the strategic complexity that makes it an ideal candidate for testing RL methods. Durak's gameplay comprises hidden information and probabilistic elements, offering a rich environment for developing, nurturing, and analyzing RL models [1]. By tackling Durak's challenges, this research not only contributes to understanding RL's potential and provides insights into its applicability in complex, uncertain environments, but also improves algorithms capable of sequential decision-making and learning with imperfect information, which are limitations in numerous real-world applications.

I. Durak

Durak, also known as "Fool", is a Russian card game traditionally played by 2 – 6 players using 36 cards (removing ranks 2-5 for each suit) from a standard deck. The objective of Durak is to remove all of one's cards and avoid becoming the "Durak" or "Fool". Each player draws six cards. The leftover cards form the talon. The card at the bottom of the talon is revealed, and its suit determines the Trump suit for the game. Each player takes turns attacking and defending. An attacker attacks with a card of their choice so that the next player in clockwise order – the defender - must defend by placing a card of the same suit with a higher value. The defender may also use a card of the trump suit to defend any card so long as the card used for the attack is a trump suit card of a higher rank. The defender's neighbours - original attacker 1 and new attacker 2 - can now place any card whose rank is already in play, either as a card used to attack or defend. One of three conditions must hold to consider a successful defence: a) the attacker chooses not to play additional cards. b) The defender successfully defends six attacking cards. c) The defender defends all attacks by equalling the number of the cards in their hand if they start with fewer than six cards. Given a successful defence, all cards on the table are removed from the game.

If cards remain in the talon, players must draw until they have six cards or the talon is depleted. The order in which attackers play cards, followed by the defender, determines the drawing order. If the defender cannot defend all cards, they must pick up every card on the table and omit their turn to attack. To avoid being the Durak, depleting one's hand after the talon is empty will result in zero cards at the start of the next round, resulting in survival. The last player with a card is the Durak. The implication is that there are no winners but only a single loser [2]. This paper will focus on two-player Durak, where there will be one attacker and one defender at every turn. The terms' survival' and 'winning' will be used interchangeably, both referring to avoidance of being the "Durak".

II. Contribution

While the RL community has extensively mapped the environments of games like poker, chess and go [3], Du-

rak remains relatively uncharted. This project aims to fill the void and venture into unexplored territory by cultivating RL models capable of understanding and mastering advanced Durak strategy. After exhaustive self-play and trial-and-error, the intention is for two highly sophisticated models, a Q-Learning model and a DQN model, to emerge. The primary contribution of this research lies in its focus on the strategy of Durak. The development of sophisticated Durak models will discover advanced tactics that could cause discourse on optimal Durak gameplay. These models analyze and learn the game's intricacies, such as hand management, opportunistic moments, and devising long-term strategy. By documenting and analyzing the strategies these models adopt, there is an aim to provide effective insights for Durak players and enthusiasts. Another aspect of this work will be the comparative performance of two RL techniques: Q-Learning (Q) and Deep Q-Networks (DQN). By simultaneously experimenting with these two models, we aim to evaluate their respective efficacy in learning and strategy development. This side-by-side study should provide concrete evidence as to which model is more successful given the environment.

III. Deliverables - Basic:

Durak Environment Design & Implementation: From the ground up, build a Durak environment in Python, tailoring to the game's specific rules.
Simple Agent Creation: Establish two baseline agents, RandomBot and LowestValueBot. These will serve as initial training opponents for the RL agents. RandomBot is designed to perform actions randomly, providing a baseline of unpredictable play. In contrast, LowestValueBot plays the lowest-valued legal action, introducing a heuristic strategy for the agent to learn against.

Intermediate:
Q-Learning Implementation: Initiate experimentation and training with a Q-Learning model. Utilizing a Q-Table, the agent will discern optimal actions from a Q-Value assigned to all state-action pairs.
RL Agent Training: Train an RL agent to win in Durak against RandomBot and LowestValueBot successfully. The agent must learn to adapt and optimize its strategy to overcome both opponents.

Advanced:
Deep Q-Network Implementation: Implement a DQN agent, enhancing a basic Q-learning algorithm with neural networks. This allows the model to approximate a Q-value given the environment. This approach is more sophisticated and is expected to improve Q-learning models' capabilities.
Strategy and Decision-Making Analysis: Analyze the strategies learned throughout training. Document the evolution of tactics and milestones achieved in gameplay.

IV. Project Overview

This project centers on the question**:** *How can reinforcement learning be effectively applied to master the strategic complexities of Durak?* The work undertaken in this project is multifaceted. Initial progress revolves around constructing a robust, yet versatile simulation environment for Durak, designed to replicate the rules and variables of the game. Within this simulated environment, we deploy two distinct RL models: A conventional Q-Learning model, and a more resourceful DQN model.

Both models are subject to thorough parameter optimization to fine-tune their learning. An important element of RL learning is finding the right balance between exploration and exploitation to ensure that models not only exploit successful strategies but also explore enough of the state space to discover superior strategies. Rewards are the signals that shape the agent's strategy over time. They are carefully adjusted to encourage short-term tactical wins but also the development of long-term strategic gameplay that, in some cases, may sacrifice immediate gains for a delayed, larger reward. Reward functions are designed to encapsulate the objectives of Durak.

Using these models, we experiment in the analysis phase, where the models' decision-making processes and strategic evolutions are recorded and assessed. The aim is to understand the learning in complex games and digest this understanding into strategies both beneficial to AI development and human gameplay. Moreover, we conduct a comparative analysis to determine the strengths and weaknesses of Q-Learning and DQN, respectively, given the unpredictable, challenging game space Durak provides. This comprehensive analysis extends beyond merit in Durak, it provides learnings and strategic proficiency necessary for success in Durak. Through refinement and evaluation, this project aims to further advance RL by demonstrating applicability in a less-studied domain and proposing insights that inform future AI applications in similar scenarios.

2 Related Work

I. Early Foundations of RL

Richard Sutton's seminal paper in 1988 established Temporal Difference (TD) Learning, a ground-breaking development in reinforcement learning. This approach took elements from dynamic programming and Monte Carlo methods, setting itself apart by learning predictions about future rewards and updating predictions based on the difference between consecutive predictions, known as the 'temporal difference' [4]. Unlike previous Monte Carlo methods, where learning is derived from the error between predicted and actual outcomes, TD methods are driven by the error between temporally successive predictions, updating its value estimates of states based on new experiences and gradually improving these estimates through iterative training [4]. This bootstrapping method enables agents to learn more efficiently as they don't have to wait until the end of an episode to update their value estimates.

The principal component of TD learning is the

TD error, which is the difference between the estimated value of a state and the estimated value of the subsequent state, adjusted by the reward received (fig. 1). The intention is to minimize the error, as this guides the agent to adjusting its value estimates towards more accurate predictions of future rewards [4]. Several variants of TD learning were introduced in this paper: TD(0) and TD(λ) [4]. TD(0) is the simplest form of TD Learning, with '0' denoting that updates are based on the immediate next state and reward. TD(λ) (Fig. 3) is a more sophisticated approach to TD learning, extending the concept of temporal differences from multiple future steps rather than the immediate next step. This is achieved through eligibility traces (Fig. 2) [4], which allow the algorithm to weigh multiple states based on their recency. λ (ranging from 0 – 1) controls the decay rate, tuning the influence of previous experiences, with $\lambda = 1$ indicating updates based on entire sequences of states and rewards, factoring the consequences of an entire training episode [4].

II. TD-Gammon

Gerald Tesauro's TD-Gammon is the first application of TD-Learning to complex gaming [5]. Utilizing TD(λ), Tesauro developed a self-teaching neural network capable of reaching expert levels in backgammon, a game like Durak incorporating both deterministic and probabilistic elements. The network achieved expertise through self-play, playing numerous iterations against itself, starting with no strategic knowledge and gradually refining game strategy to an advanced level. This paper demonstrated AI's capability of developing an understanding of game strategy without human intervention [5]. The TD-Gammon model used a multilayer perceptron (MLP) architecture, a neural network suited for function approximation. The MLP was able to output a non-linear function of inputs and to assess the variable states encountered in backgammon. Learning progressed via adjusting weights in the network's connections, driven by temporal differences between the predicted outcomes of successive game states [5].

By incorporating TD(λ), the network can make estimations not only to immediate actions, but also to a series of preceding actions, capturing long-term strategy in gameplay. In the RL field, TD-Gammon showcased the practical application of TD-Learning but pioneered an era where AI could match and surpass human expertise in complex gaming. Respected backgammon analyst and player rated #3 in world rankings (at the time of the paper) Kit Woolsey concluded that TD-Gammon holds an edge over humans in positional judgement [5].

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Figure 1: Based on the concept of TD error described by Sutton (1988) [4], here δ_t is the difference between the predicted value of the next state and the current state value, adjusted by the reward received, r_{t+1} and discounted by γ.

$$e_t(s) = \gamma\lambda e_{t-1}(s) + \Delta V(s)$$

Figure 2: Figure inspired by the concept of eligibility traces from Sutton (1988) [4], showing the update of the eligibility trace at time t for state s, where $e_{t-1}(s)$ is the trace value from the previous step, λ is the decay parameter, and $\Delta V(s)$ is the change in value function for s.

$$V(s) = V(s) + \alpha \cdot \delta_t \cdot e_t(s)$$

Figure 3: Figure adapted from TD(λ) model by Sutton 1988 [4]. Value of state s $V(s)$ is updated by incorporating δ_t and $e_t(s)$, scaled by the learning rate α.

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t)\Sigma_{k=1}^{t}\lambda^{t-k}\nabla_w Y_k$$

Figure 4: Weight update rule for TD-Gammon [5]. $w_{t+1} - w$ is the difference in network weights at time t, $Y_{t+1} - Yt$ is the temporal difference in predicted outcomes, λ^{t-k} scales the gradient $\nabla_w Y_k$ across each step k leading to t.

III. Atari with Deep Reinforcement Learning

The innovation of Deep Q-Networks by Deepmind Technologies in their paper *Playing Atari with Deep Reinforcement Learning* demonstrated another significant leap in applying deep learning to RL problems [6]. Deepmind curated a Convolutional Neural Network (CNN), trained through a Q-learning algorithm, which could effectively learn control policies through the pixels from the video output of an Atari 2600 console [6]. The work was important as it was the first implementation to merge Q-Learning RL with deep neural networks [6]. Instead of searching the Q-Table for a unique state-action pair and its associated Q-value, the neural network could estimate the Q-value for any given state directly from the visual input.

This capability addressed one of the primary challenges in applying Q-learning to complex, high-dimensional environments: The impracticality of maintaining a discrete Q-table for each possible state-action pair. By utilizing CNN's capabilities of extracting features from the pixels, the DQN model could generalize learning across states with a similar pixel layout, drastically reducing the problem space.

Furthermore, this DQN approach featured other innovations to improve training; most notably, the use of experience replay. Inspired by Long-Ji Lin [7], experience replay allows the network to learn from a diverse set of past experiences, breaking a correlation between sequential samples. Experiences are stored as a tuple, consisting of a state, the action taken at that state, the associated reward, and the next state $(s_t, a_t, r_{t+1}, s_{t+1})$ [6] [7]. Experience replay involves storing transitions in a replay buffer and sampling a random batch for training. Additionally, sampling batches is more efficient than training on individual experiences, increasing model convergence speeds [6] [7].

IV. RL Durak Implementations

Regarding RL applications for Durak, two papers, from Jan Ebert [8] and Azamat Zarlykov [9] have contributed notable insights. Their works shed light on environment design, state representation, and reward selection specifically for Durak, becoming the most relevant works for this project. The progress and insights yielded from their research have been instrumental in guiding the direction and refinement of the algorithmic approaches adopted in this paper [8] [9].

Ebert's environment used a continuous action

space, where the agent is not restricted to a finite list of moves each turn and is instead granted free reign to any legal and illegal action plausible. Despite a continuous space having infinite possible actions, learning algorithms use approximation methods to estimate the optimal action directly from the state, as opposed to learning the value of every possible discrete action. Given paramaters, algorithms that utilize policy gradients search through this space, aiming to optimize the expected return. Given the potentially overwhelmingly large discrete action space, Ebert used a Deep-Deterministic Policy Gradient algorithm with its deterministic actor-critic policy. The actor-network in DDPG maps states to actions, allowing the agent to determine an optimal action. The actor's network facilitates the learning of policies that return the highest value and is subsequently updated using policy gradients, computed based on the critic's feedback [8]. The critic network estimates the values of state-action pairs. The critic's role is to assess chosen actions and provide feedback on adjusting the actor's policy parameters. The critic utilises TD-Learning, with the difference between predicted Q-values and target Q-values driving the learning process. DDPG also integrates a replay buffer, that stores transitions experienced by the agent. After extensive hyperparameter tuning and training, Ebert developed a model capable of playing a standard game of 2-player Durak with a win/survival rate of 80% with his DDPG model [8]. Given the nature of a continuous action

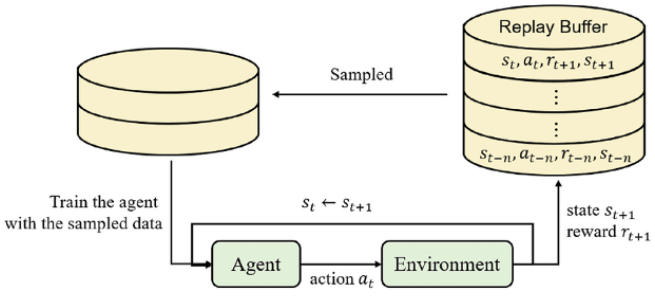

Figure 5: Diagram showing DQN sampling experiences from replay buffer [10].

space, a significant portion of the training was confined to teaching the agent to avoid erroneous moves, a disadvantage of the continuous space.

Contrastingly, Zarlykov's environment utilized a discrete action space, mirroring a more classic approach, where potential actions are defined and limited. Zarlykov implemented five different agents, each with distinct decision-making strategies [9]. There is a Random agent, which selects an action at random. A Greedy agent conserves powerful cards for the endgame by prioritizing playing lower-valued cards. A Smart agent, which 'cheats' by gaining information about the opponent's hand, aiming for moves rendering the opponent vulnerable [9]. The Minimax agent uses a recursive algorithm which explores a game tree where nodes represent game states and branches represent possible moves. Its goal is to minimize the maximum loss (hence the name) by predicting the opponent's best possible move and adjusting their strategy to maximize their minimum gain [9]. The fifth and final agent uses Monte Carlo Tree Search, which expands upon the minimax algorithm by integrating elements of randomness and statistical sampling to explore the game tree [9] [11]. Unlike the minimax agent, which attempts to thoroughly analyse the game tree to a certain depth, MCTS focuses on building a more selectively deep tree using a four-step process: 1. Selection: MCTS selects successive child nodes till it reaches a leaf node, using an exploration/exploitation selection policy. 2. Expansion: At the leaf node, the tree is expanded by adding more child nodes, representing possible moves. 3. Simulation: A simulation (rollout) is conducted for each expanded node, which passes from the selected node to a terminal state using random moves. This step estimates the potential value of each possible move. 4. Backpropagation: The simulation results are backpropagated up the tree, updating a win/loss ratio associated with each node going back to the root [9] [11].

Zarlykov experimented with two scenarios, an open world, and a closed world, providing perfect and imperfect information to each agent, respectively. Since Durak is an imperfect information game by nature, this project concentrates on the outcomes achieved in the closed-world experiments [9]. A tournament was conducted between the five agents, with the tree-based agents consistently outperforming the more elementary agents, with the MCTS proving to be more successful than its minimax counterpart (table 1). It was of note that the most successful non-learning agent was the Greedy agent, which outperformed both the Random and Smart agents (table 1).

Table 1: Results of Zarlykov's agents' matchups in a closed-world environment [9]

	Random	*Greedy*	*Smart*	*Minimax*	*MCTS*
Random		5.6% - 11.3%	3.8% - 8.7%	4.3% - 9.5%	0.4% - 2.7%
Greedy	88.7% - 94.4%		49.4% - 51.9%	45.7% - 49.2%	14.9% - 23.2%
Smart	91.3% - 96.2%	48.1% - 50.6%		50.1% - 57.5%	22.2%- 31.6%
Minimax	90.5% - 95.7%	50.8% - 54.3%	42.5% - 49.9%		12.9% - 20.6%
MCTS	97.3% - 99.6%	76.8% - 85.1%	68.4% - 77.8%	78.4% - 87.3%	

Values indicate the minimum and maximum win-rates achieved in the tournament format, with the values being the win rate of the row player against the column player.

V. Summary of Related Works

RL has progressed significantly over the years, with TD-Learning paving the way for TD-Gammon and modern approaches to gaming [4] [5]. Deep Q-Networks also played a vital role in merging two fields of AI to solve

complex problems [6]. In Durak, Ebert and Zarlykov's contributions have provided a solid foundation, warranting further exploration, by introducing two further algorithms to the table. This project aims to experiment with Q-Learning and DQN, addressing a void in Durak research and exploiting strengths while mitigating weaknesses from their work [8] [9].

3 METHODOLOGY

I. Environment Design

The Durak environment is custom-built in Python for agent development, creating a learning, decision-making, and evaluation platform. It simulates Durak's rules and mechanics, including deck management, attack and defence logic, card transactions, role assignment and player outcomes. The environment design was loosely inspired by the structure of the RLCard module, which utilises Game, Player, and Round as Python classes to simulate various card games within a reinforcement learning context [12]. Six main classes constitute the game's logic: Card, Deck, Player, Round, Game, and GameState. Subclasses of Player are Randombot, LowestValueBot, HumanPlayer, and Agent – which itself has two subclasses: Q-Agent and DQN-Agent – which store the methods for their strategies and variables for learning.

Card: Defines the playing cards with their unique suit and rank, including a method getCardPower() that evaluates card strength in gameplay.

Deck: This class replicates a standard Durak deck of 36 cards, providing necessary functionality such as shuffling and drawing cards. It also handles Card objects.

Game: The Game class coordinates Durak matches, setting up the deck and players whilst overseeing the rules and game progression. It initializes with a player list and shuffled deck from Deck, manages player hands, and facilitates rounds via the Round class. In newGame(), a trump suit is determined, and a player is assigned to attack first. This method also oversees each round, ensuring the game progresses logically, with players dropping out as they run out of cards until a final Durak remains.

GameState: This class encapsulates the current state of the game, monitoring the trump suit, cards in play on the table, and the talon and discard piles. It includes methods to fetch the undefended cards, attack cards, and defence cards to inform players of the cards for strategic decision-making. The class serves as the game's collective memory, ensuring all players have the context for action selection.

Round: Round controls each game round, managing the player's turns and roles via DetermineRoles(). It also holds possibleMoves(), which provides all possible legal actions to the player, given the current state of play. PossibleMoves() is an important precursor to chooseAction() – the method players use to decide their action. PlayRound() oversees the turn sequence, adjudicating attack-defence outcomes, and concluding the round, with defenseCheck() verifying a successful defence or failure in doing so. TalonDraw() replenishes hands from the talon post-round (if applicable) and returns a list of finished players to Game.

Player: This superclass manages common attributes for all player types, including their unique ID, their hand, and access to GameState. It handles game interactions with methods like addCard() and playCard() for hand management. Player serves as the backbone for all specialized player types.

HumanPlayer: This subclass of Player adds interactivity for users in Durak games through CLI prompts. Its iteration of chooseAction() presents available moves, taking keyboard input for action selection. Users can also view information regarding the game state using printGameState() to make an informed decision.

RandomBot: RandomBot is a bot which, as the name suggests, makes decisions randomly within the game of Durak. This bot serves as a baseline agent whose performance can be used as a benchmark against more sophisticated agents [9].

LowestValueBot: An automated bot that employs a heuristic strategy by consistently playing the lowest power card from its hand [9]. This is to conserve stronger cards for the late game, a common strategy amongst Durak players. It uses getCardPower() to evaluate and choose the least powerful card for both attacking and defending, making it more calculated than RandomBot.

(Fig. 6) is a UML diagram that demonstrates all relevant attributes and methods in the environment and their relationships.

II. Agents and RL Methodology

To effectively introduce the RL agents and their methodologies, we first establish the foundational concepts of these agents. The state refers to the agent's observation of the environment, providing the context within which decisions are made [13]. Actions are the set of all possible moves an agent can take in each state within the environment [13]. Actions transition the agent from one state to another. In this implementation, there is a discrete action state, meaning a finite list of legal moves is provided to every player. Rewards are feedback provided to the agent from actions in the environment. Rewards can be positive or negative and shape the learning process, guiding the agent toward desirable outcomes [5] [6] [13].

The Markov Decision Process (MDP) is a framework for modelling decision-making in situations where the outcomes are either random or under the control of a player [4] [5] [14]. An MDP includes a set of states S, a set of actions A, a transition function T(s, a, s') which defines the probability of moving from state s to state s' after taking action a. A reward function R(s, a, s') that gives the immediate reward after transitioning from state s to state s', due to action a. A discount factor \gamma, a value between 0 and 1 which weighs the importance of future rewards versus immediate rewards [14]. MDP also importantly assumes the Markov Property, that the future state depends only on the current state and action, not on the sequence of events that preceded it [4] [14]. An MDP aims to find an optimal policy – a mapping from states to actions that maximize the expected cumulative reward over time [14].

In RL, a policy is an agent's rule or strategy to decide which action to take in each state [13]. It can be deterministic – where the policy prescribes a specific action for each state, or stochastic – where the policy provides probabilities for choosing among possible actions. For instance, the heuristic policies for RandomBot and LowestValueBot are examples of simple, deterministic policies within our RL framework. The core of many RL algorithms is value functions, such as the state-value function V(s), which assesses the agent's position in a given state, and the action-value function Q(s, a) (known as the Q-function), which measures the quality of performing a specific action in a state. Value functions help agents measure their expected return from a particular state and the return from selecting an action within that state.

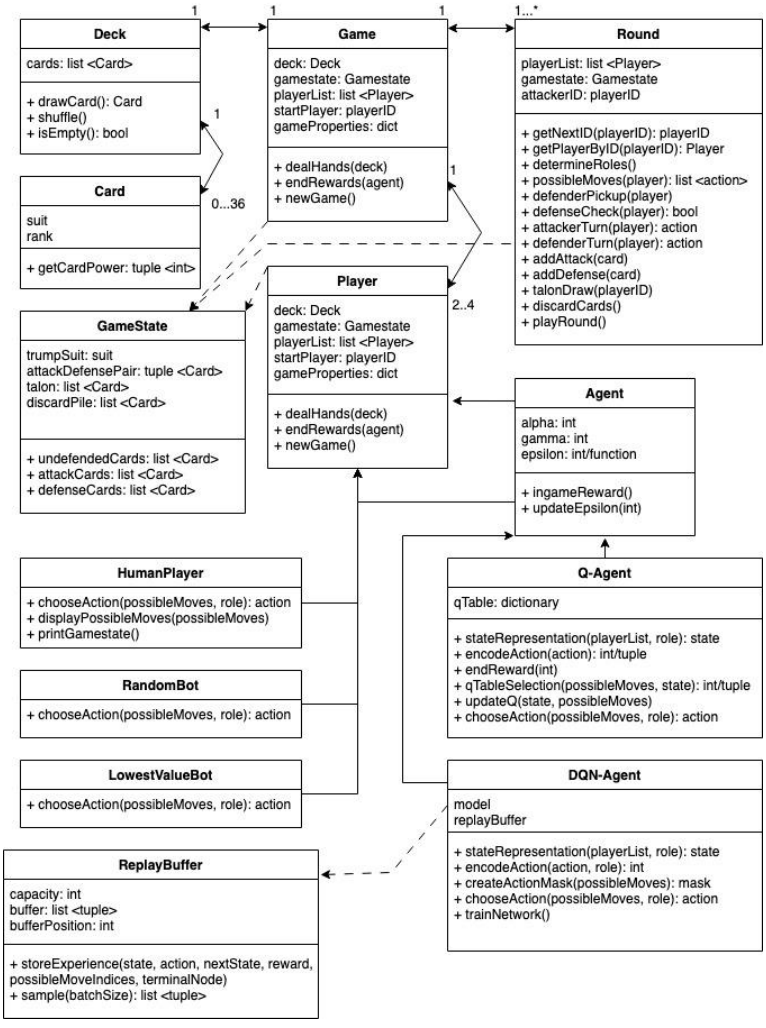

Figure 6: **UML Diagram of the Durak Environment**. Shows the relationships between different classes in the python project. Solid one-sided arrows point a sub-class to their parent class. Dotted lines point classes to class that they have a dependency towards. Two-sided arrows indicate the multiplicity between the two classes (There is 1 deck for a card, there are [0-36] cards in a deck).

As discussed in Section 2, TD-learning updates its value estimates based on other learned estimates rather than waiting for an outcome [4]. Updates are done via the TD error, which is the difference between the current state's estimated value and the resulting state's estimated value after taking an action [4]. Merging all this foundational knowledge allows us to discuss the two principal algorithms used in this project: Q-Learning and

Deep Q-Networks.

Q-Learning: Q-Learning is a model-free RL algorithm that seeks to find the best action given the current state. It is model-free because it does not require a model of the environment; it does not need to know the transition probabilities and reward functions beforehand. Instead, it learns these values by exploring the state-action space [15]. The most crucial component of Q-Learning is the Q-function, which estimates the long-term cumulative reward of taking action a in state s and following the subsequent optimal policy [15]. The Q-Table stores the Q-values for each state-action pair discovered in the environment, allowing the agent to decide their optimal action [15]. Q-Learning involves updating this Q-function in the following procedure: At each time step t, the agent observes the current state s_t, selects an action a_t, observes the reward r_{t+1}, as well as the new state s_{t+1}. Then, it updates for the Q-value for the state-action pair (s_t, a_t) using the Bellman equation (fig.7). α is the learning rate $(0 < \alpha \leq 1)$ – which determines to what extent newly acquired values will override old values. γ is the discount factor $(0 \leq \gamma < 1)$, which balances the importance of immediate and future rewards r_{t+1}, which is the reward received after taking action a_t in state s_t. $max_a Q_{(s_{t+1},a)}$ is the estimated optimal future value. The term $r_{t+1} + \gamma Q_{max}(s_{t+1}, a)$ is the learning target; the difference between this and the current Q-value is the TD error [4]. Several policies are applicable for a Q-learning agent. However, the primary strategy implemented will be the ϵ-greedy policy, whereby with probability ϵ, the agent will select an action at random given the state-action pair (exploration), and with probability $1 - \epsilon$ it will refer to the Q-table for the highest Q-value given the state-action pair (exploitation) [4] [8] [9].

Deep Q-Networks: DQN takes the foundations of Q-Learning and enhances it with deep learning [6]. A significant limitation of Q-Learning is the Q-table, which becomes impractical in environments with large state spaces. DQN addresses this by utilizing a neural network to approximate the Q-function, allowing the agent to generalize over a space of state-action pairs [6] [16]. This implementation uses a replay buffer, which stores the agent's experiences at each time step. The experiences, stored as a tuple (see section 2), are used in a random mini-batch to train the network. This random sampling dissolves the correlation between consecutive learning steps and stabilizes the training process [6] [16]. The loss function that trains the Q-network is based on the TD error, like Q-Learning. However, due to the Q-function now being approximated by a neural network, the loss is computed as the Mean Squared Error between the predicted Q-values and target Q-values (fig. 8) [6] [16] [17]. θ represents the weights of the Q-network. Q_{target} represents the target Q-network's output. s' and a' represent the next state and action. The expectation E is approximated by sampling mini batches from the experience replay buffer [6] [16].

We will delve into their respective Python implementations with these concepts in mind. The Agent class holds the core attributes and behaviours shared by all agent-based players within the Durak game environment. It extends Player, inheriting its card-handling capabilities and state awareness. Agent stores learning-related parameters such as the learning rate, discount factor, and epsilon value for the greedy strategy.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q_{(s_{t+1},a)} - Q(s_t, a_t)\right]$$

Figure 7: The Bellman Equation used in Q-Learning [14] [15] [18]

$$Loss = E\left[\left(r + \gamma\, max_{a'} Q_{target}(s', a') - Q(s, a; \theta)\right)^2\right]$$

Figure 8: The loss function in Deep Q-Networks, computed as the Mean Squared Error between predicted Q-Values and target Q-Values. [16]

Q-Agent: A subtype of Agent, Q-Agent is designed to implement the Q-Learning algorithm within the Durak environment. The key, unique attribute is a Q-table storing state-action values [15]. A state-action counter is included to monitor how often the agent encounters state-action pairs. A central element of Q-learning is state representation. The state representation is how an agent perceives the environment within. Given that every unique state needs to be represented in the Q-table, there needs to be a certain degree of generalization so the agent can revisit states and learn from previous encounters and their corresponding Q-values when choosing an action. Therefore, careful consideration has been taken to condense the state space whilst retaining as much vital information as possible to maximize learning potential for the agent.

Hand: The agent's hand represents the most critical component of its state representation, directly influencing the game's available actions and potential strategies. Hypothetically, a unique starting hand occurs with $1/6^{36}$ probability. Each unique hand requires a separate entry, making the state space too expansive. The state space was reduced by encoding the hand to a more manageable representation. The encoding categorizes cards into groups based on their ranks and whether they are of the trump suit, both of which are relevant in determining the strength of a card. The rank-based grouping is defined as follows: Low = [6, 7, 8], Middle = [9, 10, Jack]. High = [Queen, King, Ace]. Cards are then divided based on their inclusion in the trump suit, recognizing the significant strength that these cards possess, being only weaker to higher-ranked trump suits. Therefore, we have six encoding groups and their associated values: RankLow- 0, RankMiddle - 1, RankHigh - 2, TrumpLow - 3, TrumpMiddle - 4, and TrumpHigh - 5. The function encodeHand() iterates through the hand and calls encodeCard() for each card, which returns the index of their corresponding encoding group. In the encoding, each group is stored as a value, whose value is the number of cards that belong to that group.

Opponents' Hands: The next component in the state measures the relative size of the opponents' hands. Initial attempts to encode this information resulted in an unnec-

essarily expansive state space, given that the number of cards held by each opponent is not crucial for agent decisions. Instead, a binary representation was used, where '1' indicates that an opponent has more cards than the agent, and '0' denotes fewer or equal. This binary encoding, produced by encodeHandLengths(), simplifies the state space while giving the agent a comparative view of its position. However, the agent is unaware of the value of these cards.

Role: Encoding an agent's role within the game is a straightforward yet essential component of the state. It captures the agent's current function as either an attacker or defender. EncodeRole() achieves this by creating a binary array where the index corresponds to the role, with index '0' indicating the agent is an attacker and index '1' indicating a defender.

Table Cards: In initial testing, the state involved encoding the cards on the table – both attacking and defending – creating respective vectors to represent each dimension of the game's current round. The EncodeTableCards() function assigns an encoding of six slots for both vectors. This approach, while informative, again led to an excessive expansion of the state space, with the Q-table becoming bloated. Addressing this, encodeUndefendedCards() streamlined the state space by encoding only the cards that still need a response from the defender. Nevertheless, this approach remained more granular than necessary. The final state representation used by the agent counts the undefended cards.

Talon: The encoding of the talon within the state representation aims to capture a shift in the game's dynamics; initially, the state included the exact counts of the cards remaining in both the talon and the discard pile. Again, this encoding was refined due to state space constraints, and the discard pile encoding was removed. Encoding the talon was shortened to a binary presence indication of talon, with the value set to '1' if the talon contains cards. Otherwise, it is changed to '0'. This change highlights the strategic shift once the talon is empty, whereby players no longer draw cards, and Durak enters the endgame where the objective is to empty one's hand. The culmination of all these encoding methods creates a comprehensive array representing the state. This state array forms the basis upon which the Q-agent perceives the environment and makes decisions. It aims to generalize whilst preserving essential details about the agent's situation.

Action Encoding: An agent learns from its actions' consequences, meaning the action encoding method is as vital as state representation [13]. EncodeAction() is responsible for translating all the agent's possible moves into a format suitable for Q-table entries. EncodeAction() also uses encodeCard(), associating a value depending on the card's rank and its inclusion in the trump suit. When attacking, every card in the hand is viable, as is opting to pass on attacking (except for the initial attack in a round) represented by '-1'. When defending, the function creates a tuple (encodedDefense, encodedAttack), maintaining an association between the card used to defend a specific attack. Picking up all defensive cards is also represented by '-1'. This encoding of actions, paired with the state encoding, completes the state-action pair, forming a key that allows the Q-agent to access the corresponding Q-values [15].

Like the bots, Q-Agent possesses a chooseAction() function. The method integrates the state representation with the Q-Learning policy. When it is the agent's turn during a round, chooseAction() is called, and the process begins by encoding the current state [9] [13]. The agent then faces a choice determined by the ϵ-greedy policy: explore or exploit. A random value x is generated, and if $x < \epsilon$, the agent randomly selects from possibleMoves, discovering new strategies. Otherwise, the agent refers to the Q-table to find the best move for the current state. QTableSelection() is called, and it returns the action with the estimated reward using their Q-values [15]. The function handles two formats for every action: the encoding, which is how the agent perceives their action, and the standard format, which is returned to the environment to continue gameplay. UpdateQ() is also called in this function, which updates the Q-table.

When QTableSelection() is called, the function encodes all the agent's potential actions, passed as a parameter 'possibleMoves' and combines them with the current state to form a list of state-action pairs. Then, it looks up these state-action pairs and finds their corresponding Q-values from the Q-table to ascertain the expected rewards. It then identifies the action or actions (if there is a tie) with the highest associated Q-value. If more than one action is returned, an action is randomly selected among these top-valued actions. The chosen action is then translated back into the game's terms through encodingMapping(), providing the agent with both the original action for the game to process and the encoded action for updating the Q-table, evolving the table based on the success of previous actions [15].

The updateQ() function allows the agent to integrate the new information from interacting with the environment into the Q-table. This function is called in chooseAction and takes the last action selected (lastAction), the state in which that action was taken (lastState), and the reward given after the action (lastReward) as parameters, in addition to the possible moves, and the new state that the agent is in. Like QTableSelection, it finds the maximum possible Q-value from the agent's potential moves and updates the (lastState, lastAction) entry in the Q-table using the Bellman equation (fig. 7) [15] [18].

Deep Q-Networks Agent: The other subtype of Agent, DQN-Agent, is tailored to implement the DQN algorithm for Durak [6]. The key component of this class is the neural network used to approximate Q-values. Other invaluable components, such as the replay buffer, state representation, and action encoding, facilitate the training of

this neural network [6]. The neural network's architecture is a necessary component in the agent's learning process for the game of Durak. After several design experimentations and state representations, the approach was condensed to a single DQN model designed to handle the complexity of the entire game environment efficiently. The model features an input layer that handles a state representation tensor of 153 dimensions. It progresses through three fully connected layers with 128, 64, and 42 neurons, respectively, all utilizing ReLU activation functions for non-linearity [16] [17]. The final layer produces an output corresponding to 37 partially generalized possible actions in the game, aligning with Durak's action space. During the forward pass, the network applies an action mask to the output, ensuring the actions' validity [19]. For DQN, state representation takes a different role than Q-Learning's Q-table.

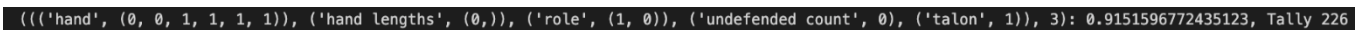

Figure 9: A state-action pair stored in the Q-Table. This example shows the agent has a strong hand, the opposing player has less cards in their hand, the agent is attacking, and the talon is not depleted. The agent chose to attack with a card of value 3, with an associated Q-value of 0.915. This state-action pair has been visited 226 times.

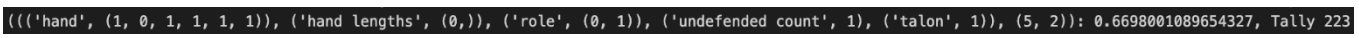

Figure 10: Another state action pair in the Q-Table. This instance shows an agent defending a card of value 2 with a card of value 5.

While the Q-Agent requires a generalized state representation to handle the finite entries of a Q-Table, the DQN-Agent approximates the Q-values, allowing for a far richer state representation. The DQN's ability to process a higher dimensionality of inputs means the state can provide more context surrounding the environment. This contextual depth is vital, as it allows the network to distinguish differences between states that appear similar but have different implications for the agent's strategy. The state representation vector must be fixed for all states, as the neural network requires a consistently sized input layer, the state-action vector [17].

Cards: encodeCard() assigns a unique index to each card, transforming its suit and rank into a single inte-ger that the neural network can interpret. Each suit has a base index: Hearts start at 0, Diamonds at 9, Spades at 18, and Clubs at 27. The method first retrieves the card's suit and rank. The rank – a value between 0 and 8, is added to the suit's base index to calculate the unique index within a flattened array representing every unique card in the 36-card deck. For example, the Ace of Diamonds has an index of 9 + 8, resulting in an index of 17. This index sets a corresponding position in state representation arrays for the agent's hand, the discard pile, and table cards.

Hand: The hand remains the most important factor for decision-making. EncodeHand() translates the agent's hand into a DQN-compatible format by mapping each card to its corresponding index using encodeCard(). It generates a 36-element array representing all possible card indices in a standard Durak deck. As each card is encoded, the function assigns a value of '1' to the position in the array corresponding to the card's unique index, creating a one-hot encoding representation of the hand. The cards not present are represented by '0' in their index, maintaining a consistent array size to satisfy the neural network's input layer [17].

Discard Pile: Encoding the discard pile gives players insight into which cards are no longer in play. encodeDiscardPile() in the DQN captures information omitted in the Q-Learnings state due to Q-Table constraints. Similarly to the hand, the discard pile is represented by a 36-element array. When the player discards a card, its index in the array is marked by a '1', signifying its absence from play. This method communicates to the DQN which cards have been played and are no longer available for selection, which is an upgrade from Q-Learning. Cards which remain in play are left as '0', again to preserve consistency for the neural network.

Table Cards: encodeTableCards() translates the cards in play into a structure that identifies both attacking and defending cards. It constructs two 36-element arrays, one for attacking cards and one for defending cards. Unlike Q-Learning, this encoding sacrifices information on associations between attacking cards and the card used to defend them. That is, there is no clear indication as to which defence cards are used to defend the attack cards, and there is no indication as to which attack cards still need to be determined. However, the agent should be able to discern that undefended cards exist due to more 1s in the attacking array than in the defensive array.

Opponent's Hands: encodeHandLengths() is more contextual than Q-learning's. The agent can discern the comparative strength of hands, informing the agent whether it is standing in the environment. The encoding function generates an array with an entry for each player, where the value stores the length of that player's hand.

Role: encodeRole() is the same function used for the Q-Agent, as there is no additional context to extract from the game.

Talon: encodeTalon() represents the proportion of cards remaining in the talon. It divides the current number of cards in the talon by the maximum possible number, normalizing this state feature to a value between 0 and 1. This normalized value offers the agent a sense of how far the game has progressed and helps it adjust its strategy as the talon depletes, drawing towards the endgame.

Once all components are individually encoded, getStateRepresentation() constructs a state vector, extend-

ing the vector with each encoding until we are left with a vector of length 153. This vector is transformed into a PyTorch tensor, which formats the state for input into the DQN model.

The output layer of the neural network forecasts Q-values for each potential action, correlating each output index to a particular move and its estimated value based on the current state. The layer's size reflects a simplified action space, with each neuron corresponding to an action's Q-value. This streamlining was achieved by reducing the number of actions – originally, the model considered every rank and suit for attacks and every possible defence response. However, only a fraction of actions were applicable in each state, so having a large action space resulted in inefficient training. Therefore, actions were consolidated to improve model convergence and training efficiency [6]. Now, when the action is a card, the index is determined by the card's adjusted for trump cards to reflect their elevated value. For defences, an index is computed based on the rank difference between attacking and defending cards.

A mask tensor was also created using getActionMask(), which filters out illegal or irrelevant actions from the network's considerations given a state [19]. It fills a tensor with negative infinity values, signifying actions not applicable to the agent given the state [19]. For each legal move present in possibleMoves, the corresponding index in the mask is set to '0', indicating a valid option the network can select given the state. This selective masking is important, so the agent's decisions are constrained to permissible actions only [19]. ChooseAction() in the DQN agent again is the core method that is called when the player is obliged to choose a move. The state is converted to a tensor to pass as an input to the neural network, which predicts Q-values for all possible actions given the current state [16]. Action selection is done using an ϵ-greedy policy [13]. Despite all actions receiving a Q-value, a mask is applied to restrict the agent from selecting actions belonging to the indices relating to possibleMoves [19]. After an action is selected, an experience is stored in the replay buffer [6]. The action in its original format is returned to the game to continue gameplay. Training the model occurs after accumulating enough experiences in the replay buffer to feed a batch to the neural network [6]. This functionality happens in trainNetwork(). During training, trainNetwork() is called a certain number of games, determined by parameter *analysisIntervals*, and the number of training iterations is determined by *TrainingIterations*. Therefore, during training, the overall number of training iterations is $\frac{matchesPlayed}{analysisIntervals} trainingIterations$. Training is done is in segments rather than all at once, so the agent can improve the model gradually by learning from the previous training segment.

TrainNetwork() uses experiences stored in the replay buffer to train the network [6]. If the batch size is met, the model samples experiences from the replay buffer and processes the components of each experience (states, actions, rewards, next states, and terminal node flags) into tensors compatible with PyTorch. The function computes the current predicted Q-values from the network for the sampled states and actions, $Q(s, a'; \theta)$ and determines the maximum Q-value for the next states, $max_{a'} Q(s', a'; \theta)$, while setting this value to 0 for terminal states [6] [16] [17]. The target Q-values are calculated as $y = r + \gamma\, max_{a'} Q(s', a'; \theta)$ for non-terminal states and $y = r$ for terminal states [5] [6] [16] [17]. The loss, $L = \frac{1}{N}\Sigma\big(y - Q(s, a; \theta)\big)^2$ is the MSE between the predicted Q-values and target Q-values. Gradient descent is performed on the loss to update the network parameters θ [16] [17]. Through training, the DQN agent incrementally adjusts its Q-value predictions to align with future states' observed rewards and estimated values, refining its policy.

4 Results

In the results section of this study, we conduct detailed experiments to evaluate the performance of our Q-Agent and DQN-Agent within a modified Durak environment. Our approach involves two phases: optimization of the agent's parameters through training and competitive evaluation through structured matchups. The performance metrics used to evaluate performance will be the average rewards, game lengths, and survival/win rates. The initial reward structure assigns +1 for survival in a game of Durak and -1 for becoming the Durak [9]. During gameplay, the agent's performance is assessed through the ingameReward() function, which measures the change in hand strength after an action is played. This function calculates the average difference in hand strength before and after an action, rewarding the agent with +0.1 for improvement and penalizing with -0.1 for a negative change.

For initial Durak experiments, our first opponent to overcome is the RandomBot. Experiment 1 featured a Q-Agent with a fixed $\epsilon = 0.1$ strategy with parameters $\alpha = 0.1$ and $\gamma = 0.99$. The agent made easy work of randomBot, maintaining an average win-rate of above 80% throughout training (fig. 11), as well as maintaining an average reward over 2.0, whilst also significantly decreasing the average game length from 40 rounds to 30 (fig. 12), indicating its competence in defeating the bot in less turns. Following training, the agent played randomBot in 1,000 games. It achieved a compelling win rate of 91.3% (table 2).

Experiment 2 trained an agent against lowestValueBot. In this experiment, ϵ-decay was used, starting at 1.0 and linearly reducing to 0.1 in the first 40,000 iterations. LowestValueBot is a more competent opponent, so the action space must be explored to search for more robust strategies. The results of training showed that LowestValueBot posed a threat, with initial win rates as low as 12-13%. After around 40,000 iterations, the win rate surpassed 50%, hovering in the region of 60% by the

end of training (fig. 13). Interestingly, the average game length did not change much, staying between 27 and 26.5. The games were far shorter, most likely indicating a higher competence from the opponent, which resulted in faster-flowing gameplay. However, the evidence of improvement was noticed with the average rewards drastically improving, with initial rewards as low as -1.0 and plateauing at around 1.5 by the 50,000-iteration mark, when the agent maintained its 60% survival rate. Post-training, the agent played 1,000 games against LowestValueBot, winning 65.7% of games, an impressive result against a coherent opponent (table 2).

Experiments 1 and 2 were replicated using a DQN-Agent instead. 50,000 matches were played, with training taking place every 500 matches and 1000 training episodes, resulting in 100,000 training episodes. Batches taken from the replay buffer were size 128, meaning 1,280,000 experiences were sampled randomly. Like Q-Learning, RandomBot was an easy opponent for the agent to overcome, maintaining an 80% win rate after 10,000 games. However, the underlying statistics were interesting. The average reward reached only 1.0, almost half its Q-Agent sibling, but the game lengths decreased from 32 to 26. Perhaps the average reward was lower because fewer hands were played due to shorter matches. Contrastingly, LowestValueBot proved a significant obstacle for DQN, achieving a sub-50% survival rate overall. Near training completion, average rewards only reached 0-0.2, which indicates weak performance.

After training completion, the four trained agents, randomBot and lowestValueBot, competed in a round-robin style tournament, where every player played over 1,000 matches, and the results were stored to establish a hierarchy of competence [9]. The results of every matchup are stored in Table 2. Post-tournament, the Q-table proved superior to its DQN counterpart given this environment, with the agent trained against LowestValueBot being the strongest of all players. Several games were played against this agent (Agent 2C) to understand their tactics during gameplay. The following was observed: When attacking, the agent learned to prioritize playing weaker cards, saving stronger cards for defense. However, the agent would occasionally inefficiently defend cards, using strong trump cards over non-trump cards. To improve upon this, a Q-Agent and DQN-Agent were trained against this strongest agent, intending to develop an agent to overcome the strongest player thus far.

Experiment 3 introduced a penalty of -0.05 for agents defending inefficiently. This penalty was applied if they had a weaker card that could defend the card they defended, encouraging smarter defending. Experiment 3 also featured a lower $\gamma = 0.5$, so the agent could focus on the immediate consequences of their actions. The training of experiment 3 proved successful, maintaining a 60% survival rate from the 40,000-iteration mark. Even with the new reward penalty, the agent averaged 1.5 reward-sper game, starting at -1.0. The reward scheme worked well; in several games, it had become evident that defending cards had become more intelligent, with the agent learning that it was sometimes smarter to pick up cards instead of playing a valuable trump card. Unfortunately, the DQN agent did not yield the same results. By the end of training, the agent only met a 30% survival rate, showing it could not compete with its Q-Learning counterpart. This was reinforced in the matchups between the three agents, with a second tournament indicating that the Q-Agent, with new rewards, trained against 2C, was the strongest agent, boasting a 66.6% win rate against 2C. DQN again came short, with a mere 37.2% win rate (table 3).

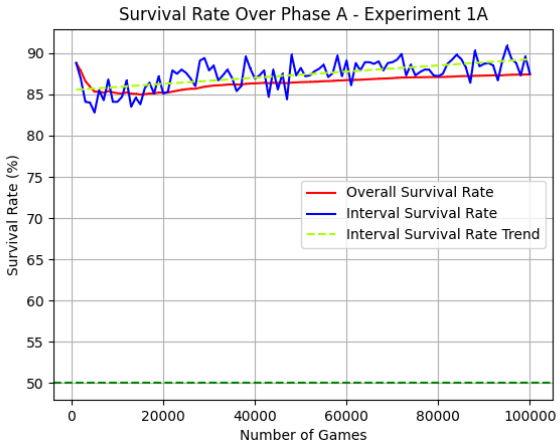

Figure 11: Survival Rate % of experiment 1A trained against RandomBot. Interval Survival Rate refers to the average win rate of the last 100 matches played, with the trend being the line of best fit.

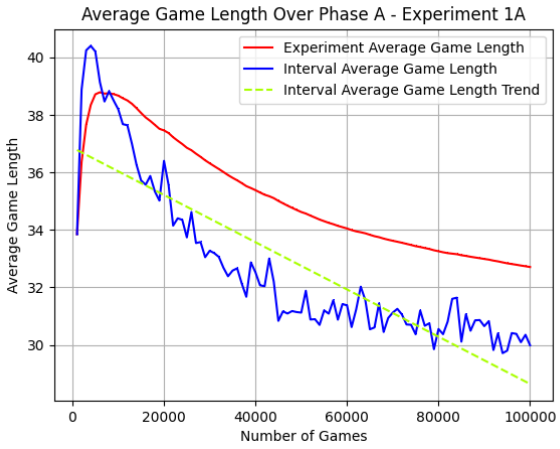

Figure 12: Average Game Length of experiment 1A against RandomBot.

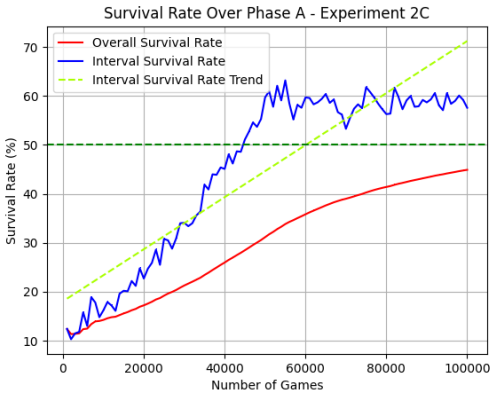

Figure 13: Survival Rate of experiment 2C against Lowest-ValueBot.

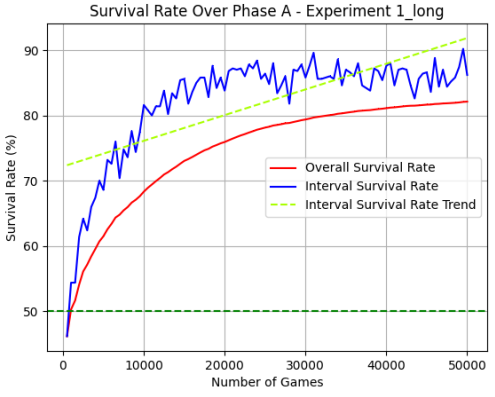

Figure 14: Survival Rate of DQN Experiment 1 against RandomBot

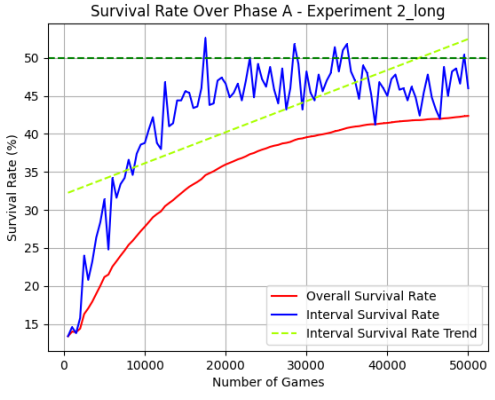

Figure 15: Survival Rate of DQN Experiment 2 against LowestValueBot

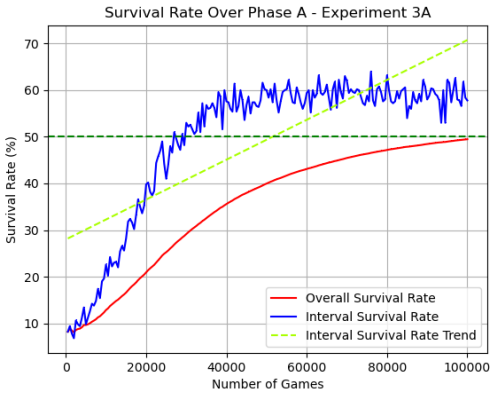

Figure 16: Survival Rate of Q-Agent Experiment 3A trained against Q-Agent Experiment 2C.

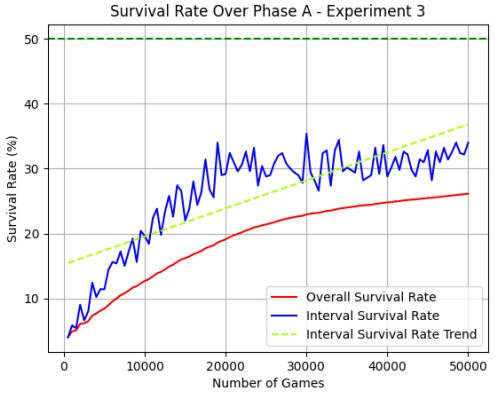

Figure 17: Survival Rate of DQN Agent Experiment 3 trained against Q-Agent Experiment 2C

Table 2: Round Robin Style Tournament of Selected Durak Players

	Random Bot	*Lowest ValueBot*	*Q-Agent 1A*	*Q-Agent 2C*	*DQN Agent 1*	*DQN Agent 2*
Random Bot		10.5%	8.7%	6.2%	10.1%	10.7%
Lowest Value Bot	89.5%		43.7%	34.3%	48.8%	49.6%
Q-Agent 1A	91.3%	56.3%		46.5%	58.4%	55.4%
Q-Agent 2C	93.8%	65.7%	53.5%		65.4%	68.0%
DQN Agent 1	89.9%	51.2%	41.6%	34.6%		52.1%
DQN Agent 2	89.3%	50.4%	44.6%	32.0%	47.9%	

Values indicate the win-rate of the row player against the column player after 1,000 matches played (not training). Inspired by Zarlykov [9]

Table 3: Round robin style tournament of selected Durak players

	Q-Agent 2C	*Q-Agent 3A*	*DQN Agent 3*
Q-Agent 2C		33.4%	62.8%
Q-Agent 3A	66.6%		71.1%
DQN Agent 3	37.2%	28.9%	

Values indicate the win-rate of the row player against the column player after 1,000 matches played (not training). Inspired by Zarlykov [9]

5 EVALUATION

In evaluation, we assess our Q-Agent and DQN-Agent performance within the Durak environment. This analysis aims to quantify the success of the two agents in terms of strategic proficiency and determine the effectiveness of the reinforcement learning models employed. Here, we scrutinise the outcomes against our predefined objectives and research questions, considering the metrics of win rates, reward trends and game lengths. In addition, this evaluation reflects on the overarching methodology, examining the route from initial planning to the execution and subsequent results. By examining these aspects, we aim to offer a comprehensive review of the agents' capabilities and the extent to which they advance the field of AI in strategic game-playing. In examining the results of this study, we evaluate them against the initially set deliverables, consisting of foundational milestones for creating a robust Durak environment and developing intelligent agents within it.

Durak Environment Design & Implementation: Our fundamental deliverable was to build a Durak environment from scratch, capable of supporting the core game mechanics, plus the flexibility to accommodate slight rule variations. A modular and adaptable environment in Python was successfully created, fulfilling the deliverable and creating a solid foundation for agent training and evaluation.

Simple Agent Creation: Two baseline agents, RandomBot and LowestValueBot, were created as opponents serving as measurable benchmarks for the RL agents. RandomBot provided an element of unpredictability, and its simple strategy intended to pose little threat. LowestValueBot utilised a commonly used, heuristic strategy approach in Durak, with its only drawback being its lack of variety. Both agents functioned as expected, satisfying the deliverable.

Q-Learning Implementation: The Q-Agent's performance in the Durak environment exceeded expectations, demonstrating a level of proficiency suggesting a highly effective learning process. One of the most contributing factors to its success was implementing a generalised state space. The state space adopted for the Q-Agent facilitated the learning by condensing the vast number of possible spaces into a more manageable number. In experiment 3A, after 100,000 matches, the agent had only experienced 3932 states, visiting its most encountered state up to 58,422 times. The abstraction allowed the agent to quickly learn basic strategies, such as playing weaker attack cards and learning to defend, which allowed the agent time to learn more advanced concepts, such as trump card retention and long-term strategy. An example of a long-term strategy would be opting not to defend against weaker attacks with powerful trump cards, preserving their stronger cards till the talon is depleted, where they now have the strongest cards in the most critical situation. The generalisation also prevented overfitting. For the Q-Agent, being less sensitive to individual game states meant it did not 'memorise' every state but instead 'understood' its position. After visiting the state numerous times, the agent could relate to its position and apply its knowledge more flexibly. Introducing an additional penalty whilst reducing the discount factor only contributed further to sharpening the agent's strategy, with it being able to ignore the penalty in certain circumstances if it went on to win.

DQN-Learning Implementation: The DQN-Agent's learning journey in the Durak environment was marked by a mixed bag of successes and shortcomings. While the agent did achieve favourable win rates against both bots, it fell short of the impressive performance set by its supposedly 'inferior' sibling, revealing certain areas where its learning strategy could be refined. Unlike the Q-Agent, the DQN-Agent operated with a less generalised state space, which may have contributed to its comparative underperformance. The more granular state representation introduced a higher level of complexity, which the neural network found challenging to navigate. A highly detailed state space could lead to difficulty in discerning the essential strategic elements from the input tensor. The design of the neural network itself may have played a role in DQN's struggles. The number of layers and number of neurons in each layer may have been inadequate to capture the level of abstraction required to make these decisions.

Furthermore, the DQN may have required more training iterations or a larger batch size to converge effectively towards smarter policies. The action masking aimed to focus the training on legible moves may have affected the generalisation process, as legible moves, particularly for defence, hinge on factors outside the agent's control. During gameplay, DQN-Agent exhibited inefficiencies in both attack and defence, with instances of initial attacks being high-ranked trump cards, a questionable choice. Even after parameter tuning and reward structure changes, the DQN did not significantly improve. This may suggest the

agent struggled to find actions leading to better rewards, as evidenced by consistently having lower average rewards than Q-learning. Despite these shortcomings, some positives remain to take away from DQN experimentation. It still met the 50% threshold against both bots, indicative of skill-based gameplay. It also averaged a lower game length against the randomBot than Q-Learning, indicating the potential for a more successful DQN agent.

Strategic Scope: In the initial outline of this project, there was an ambition to explore Durak environments with higher player counts, including games with 3, 4, 5, and 6-player configurations. As research and experimentation progressed, the decision was made to pivot the focus solely on the 2-player variant of Durak. This decision was a calculated adjustment to concentrate our investigative energy and resources on one game mode. The decision to streamline our efforts on the 2-player variant has left an avenue open for expanding the research to include games with more players. With the established foundational strategies and learning algorithms, we can extend our models to handle the increased complexity and dynamic interactions that additional players would feature.

Previous Works & Contribution: Our work in this project reinforces the contributions of Jan Ebert and Azamat Zarlykov. While Ebert's employment of continuous action spaces and policy gradients is a unique approach, Zarlykov's discrete environment provides a framework from which our environment is inspired [8] [9]. Our project plugs the gap by evaluating two further RL models that have yet to be implemented for Durak. Our Q-Agent's performance stands tall amongst other works, but it is evident that our DQN agent did not perform to the same standard as Zarlykov's deep learning MCTS counterpart [9]. MCTS does not require an explicit representation of all possible actions beforehand. Instead, it builds a search tree incrementally, deeply exploring the most promising moves [9]. In contrast, the DQN-Agent learnt its policy mapping states directly to actions. As mentioned above, the DQN-Agent struggled due to a possibly oversaturated state representation and a failure to approximate Q-values, which led to better rewards. MCTS inherently provides a form of value approximation well-suited to games with hidden information and randomness, which may have provided an edge in adapting to Durak's unpredictability [9].

Research Question: The results of this project provide an answer to the research question: Reinforcement learning can be effectively applied to master the strategic complexities of Durak, evidenced by the resourceful performances of the Q-Agent, which successfully developed and applied sophisticated strategies iterative 'trial and error' learning and adaptation. The Q-Learning model demonstrated its efficacy against multiple models, with our final model building on the success of the previously most successful model, proving to be a formidable opponent for players, agents, and bots. The success of Q-Learning proved its relevance in modern RL techniques, boasting superior performance to its more sophisticated counterpart and highlighting the benefits of a generalised state space.

While the DQN-Agent did not match Q-Agent's level of proficiency, it still achieved respectable results, indicating that neural network reinforcement learning still has the potential to be applied more effectively. Focusing exclusively on a single type of agent - in this case, the Q-Agent - rather than dividing efforts between two distinct models may have led to the development of a superior agent. Concentrating resources and research on deepening a single algorithm, state space representation, and reward structure may have allowed for a more thorough and nuanced understanding of Durak.

6 Conclusions

Given the findings and insights gained from exploring RL within the Durak environment, this conclusion aims to encapsulate the project's premise, reinforce its significance, and discuss its implications for the broader domain of AI. This research's objective was to explore an area of RL application that remained unexplored, applying both Q-learning and Deep Q-network methodologies to Durak. The project bolstered the assumption that RL could navigate complex, uncertain gaming environments and expanded upon the discourse in gaming strategies through AI. The designed Durak environment was a playground for the models to familiarise themselves with and learn from. The environment's capability to simulate mechanics such as card transactions and attack/defence logic was pivotal. The Card, Deck, Game, and Round classes formed the backbone of the Durak simulation, emulating the elements and flow of the game, while the GameState class served as a memory all players relied on for decision-making. Each subclass of Player was given the necessary tools to navigate the environment. This environment fostered different strategies, creating a system where diverse approaches were developed. As our project focused on the two agents, both were subjected to rigorous training against bots - one with randomised actions and one with a basic heuristic strategy.

Q-Agent emerged as a strategist who comfortably outmanoeuvred the bots whilst showcasing an evolution in adaptive and sophisticated strategic play. This evolution, showcasing evidence of long-term strategy and comprehension of hand capabilities, was a testament to the Q-Agent's learning capabilities. Adapting the generalised state-space allowed the Q-Agent to avoid overfitting and become familiar with certain situations, allowing the agent to revisit scenarios thousands of times, navigating gameplay with experience and ease. The results yielded a formidable Durak player, as shown by the results of experiment 3. The introduction of a new penalty refined the decision-making. The Q-Agent exploited the strategic subtleties, recognising the importance of talon depletion and conserving trump cards for this occasion. This depth was reflected in high survival rates against all other play-

ers and the maintenance of high rewards despite the new penalty. The agent's evolution from a basic rule-following entity to a sophisticated player capable of intricate strategy exemplifies the project's overall success.

Conversely, the DQN-Agent, whilst similarly capable against RandomBot, appeared weaker than Q-Learning, potentially overcomplicating the problem. This suggested certain limitations to neural network-based RL for Durak. The agent's experimentation results revealed that while rich in detail, the granular state space may have hindered the DQN's ability to discern and generalise essential strategic elements necessary for superior gameplay. Additionally, the neural network architecture may have struggled to approximate Q-values given the inclusion of action masking, meaning factors out of the agent's control restricted the learning of all actions, resulting in rigid convergence and inefficient reward allocation. The DQN's ability to surpass the benchmarks set by Q-learning highlighted that complexity does not always infer efficacy. This venture into DQN design has provided learning opportunities, implying that even in a controlled environment, AI requires careful calibration to comprehend an environment's strategy.

Given these endeavours, the research question posed can be answered affirmatively. RL has proven itself in mastering the strategic complexities of Durak, exemplified by the Q-Learnings adaptability. The comparative analysis of Q-Learning and DQN contributes to the discourse on model development. It reminds us that Q-Learning still has its benefits in a field abundant with revolutionary algorithms. As we close this project, it is evident that the value extracted from this research extends beyond Durak and gaming. The methodologies refined here carry implications for future applications of RL that require strategic acumen and decision-making under uncertainty.

REFERENCES

[1] É. Bonnet, "The Complexity of Playing Durak," *International Joint Conference on Artificial Intelligence*, vol. 25, pp. 109–115, Jul. 2016.
[2] "Durak - Card Game Rules," *www.pagat.com*. https://www.pagat.com/beating/podkidnoy_durak.html (accessed Apr. 24, 2024).
[3] M. Wiering and Martijn Van Otterlo, *Reinforcement Learning : state-of-the-art*. Heidelberg ; New York: Springer, 2012, pp. 539–577.
[4] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug. 1988, doi: https://doi.org/10.1007/bf00115009.
[5] G. Tesauro, "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, Mar. 1994, doi: https://doi.org/10.1162/neco.1994.6.2.215.
[6] Volodymyr Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," *arXiv (Cornell University)*, Dec. 2013, doi: https://doi.org/10.48550/arxiv.1312.5602.
[7] L.-J. Lin, "Reinforcement learning for robots using neural networks," *Machine Learning*, vol. 8, no. 3/4, pp. 293–321, 1992, doi: https://doi.org/10.1023/a:1022628806385.
[8] J. Ebert, "Reinforcement Learning for Durak Bachelor's Thesis on Reinforcement Learning," 2017.
[9] A. Zarlykov, "Artificial Intelligence for the Card Game Durak," 2023.
[10] D. Lee and Seung Jae Lee, "Motion predictive control for DPS using predicted drifted ship position based on deep learning and replay buffer," *International journal of naval architecture and ocean engineering*, vol. 12, pp. 768–783, Jan. 2020, doi: https://doi.org/10.1016/j.ijnaoe.2020.09.004.
[11] O. Baykal and F. Alpaslan, "Reinforcement Learning in Card Game Environments Using Monte Carlo Methods and Artificial Neural Networks," 2019.
[12] D. Zha *et al.*, "RLCard: A Toolkit for Reinforcement Learning in Card Games," Feb. 2020.
[13] R. S. Sutton and A. Barto, *Reinforcement Learning : an Introduction*, 2nd ed. Cambridge, Ma ; Lodon: The Mit Press, 2018, pp. 1–13.
[14] R. S. Sutton and A. Barto, *Reinforcement Learning : An Introduction*, 2nd ed. Cambridge, Ma ; Lodon: The Mit Press, 2018, pp. 47–68.
[15] C. J. C. H. Watkins, "Learning from Delayed Rewards," King's College, 1989.
[16] V. Mnih *et al.*, "Human-level Control through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: https://doi.org/10.1038/nature14236.
[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, Massachusetts: The Mit Press, 2016.
[18] F. G. Foster and R. Bellman, "Dynamic Programming.," *Economica*, vol. 26, no. 104, p. 364, Nov. 1959, doi: https://doi.org/10.2307/2550876.
[19] Z. Wang, X. Li, L. Sun, H. Zhang, H. Liu, and J. Wang, "Learning State-Specific Action Masks for Reinforcement Learning," *Algorithms*, vol. 17, no. 2, Jan. 2024, doi: https://doi.org/10.3390/a17020060.