Solving Checkers with Reinforcement Learning

Submitted by: Yiğit İlk, Roman Mishchuk @5 February 2025
Technical University of Munich, Heilbronn, Germany
{yigit.ilk, roman.mishchuk}@tum.de
Supervised by: Han Zhou
Professorship of Business Analytics, TUM Campus Heilbronn

Abstract

Our group created a Reinforcement-Learning agent-based Checkers player project along with this report for Introduction to Reinforcement Learning WS2024-2025 Course at TUM Campus Heilbronn. The goal was to create an overall utility with multiple types of players including Q-Learning, Random player and human players where each can play checkers against each other. Therefore, overall, we created a user interface for playing checkers of board size 6×6 and created different Reinforcement Learning algorithms capable of playing on our board.

We have tested and trained over 30 Reinforcement Learning models for our task, taking around 2-3 hours of training time per model, and compared the best contenders here in this paper. We also discuss the methods we used to compare different algorithms and how we created and utilized our own graphical user interface meant for excessive debugging and data visualization, allowing us to create great-performing RL models.

All of this work can be found in the Github repository: [1].

Contents

List of Figures

List of Code Snippets

1. Introduction

Reinforcement Learning is a rapidly developing and ever more popular way to play checkers to find new winning game moves. Yet, due to the nature of this approach, the resulting win rates are never perfect and may often be disappointing based on the algorithm used. This poses a significant challenge when deploying such agents where the sole goal is to win as many games as possible and where the possible moves and states of the playable field go beyond 5 million possibilities.

That made us curious to test out the methods of making the machine-learning approach higher win rates within our setup of the checkers' board game with a 6×6 play area. Hence, we set the following goals for the project:

given a trained reinforcement-learning (later RL) model capable of navigating the playing board with moves that are considered good and bad based on how

close it brings us to victory, construct a great performing RL structure with award functions that would improve the win rates of the model before the games are actually executed (real-time), ensuring the highest win rate we can get against all agents we have.

For the algorithms to create our RL, we were given a checkers board (Figure 1) where the American Checkers rules apply, which can be found here:

1 Board size is a 6×6 matrix.
2 Brown-colored squares start in the bottom right corner, as can be seen in (Figure 1).
3 Checkers are placed on the dark squares.
4 The game goes until neither one of the players has valid moves left.
5 Simple checkers can only move forward diagonally one step.
6 If, after a capture, the same piece can be captured again, it must do so.
7 If a checker reaches the opposite side of the board, it becomes a king.
8 Kings can move and capture diagonally in all directions in only one step.
9 We have decided that after 50 moves of not capturing any pieces, we should tie the game, or else the game can go on forever.

As the play area is smaller than usual, our main focus slowly starts from finite moves. Still, it then shifts to infinite moves as we recognise along the development of our RL agents that we do not have enough computing power to create an agent with >55% win rate over 50 thousand games played against a randomly playing agent. Therefore, the goal is for us to experiment with already existing RL approaches, as will be touched further in (*2. Theory Overview*), (*3. Methodology*) and then improve upon their performance as we standardize a good methodology for testing their performance.

Overall, we aim to provide a graphical user interface and a back-end, both written in Python programming language for American Checkers with a board standardization of 6×6 (which the size of can technically be changed for agents created in the future), where multiple agents can be deployed against each other, trained and then compared for their performances.

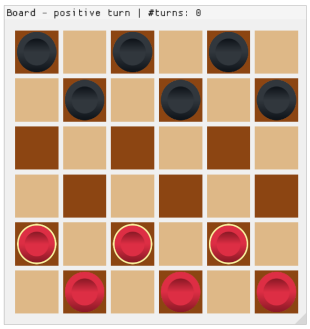

Figure 1. The checkerboard of size 6×6 in our GUI

2. Theory Overview

In this section, we will discuss the relevant theory that we used to solve the posed problem, along with the reasoning behind our choices.

2.1. Markov Decision Process

Markov Decision Process (MDP) – is a tuple $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{P}, \boldsymbol{R}, \ldots)$ (Figure 2), where

- S is the state space,
- A is the action space,
- P is the transition matrix space, (because MDP follows *Markov Property*, we can write $P_{i,j}(a,t) = \Pr(s_{t+1} = j | s_t = i \wedge a_t = a)$)
- R is the reward space.

The game of checkers, from the point of view of our agent, can be modelled well mathematically using the MDP: it has discrete time steps when the agent takes turns, and at each turn, the agent can observe the current state of the board (configuration of the pieces) and the reward for his previous action (whether agent's piece got promoted to king, whether capture has happened, whether agent won or lost). Afterwards, the agent can take his own action (moving a piece from one place to another), which would somehow influence the environment.

That allows us to define a *run* within this model – an ordered collection of tuples $[\ldots, (R_t, S_t, A_t), \ldots]$, which would describe a path of our agent inside the MDP.

We note here that the environment in this context includes both the board and the opponent of the agent.

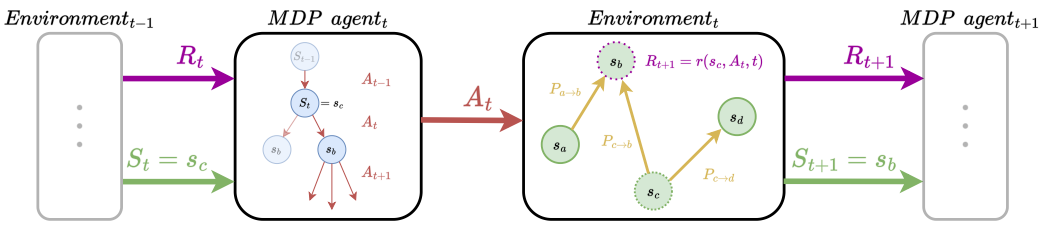

Figure 2. Markov Decision Process (MDP)

Here $\boldsymbol{R_t}$ and $\boldsymbol{S_t}$ represent current reward and state of the environment that the MDP agent observes, after which agent takes an action $\boldsymbol{A_t}$, advancing model to $\boldsymbol{t+1}$.

Given that during the same turn, both agent and its opponent can move the piece several times, later, we simulate such setting in the following way: if, after submission of the agents' action, we notice that another move is possible (that is, the agent has captured opponent's piece and now can capture another one with the same piece), the state of the environment would be changed only by that one capture move of the agent, and afterwards returned to the agent to decide on the next capturing action. Otherwise, not only the agent's move is applied to the environment, but also the opponent's move (and, in the same way as in the agent's case, directly following capture moves). The environment response at the next timestep would include the result of all the enemy moves together with their reward.

2.1.1. MDP horizons.

There are two types of MDPs: the ones with a *finite horizon* and those with an *infinite horizon*. In the first case, the agent has a constant number of moves to make, after which the game always ends. In the second case, the number of moves can vary or even be infinite.

Because of Rule 9 we can be sure the game would always end. Yet, there are no guarantees on the number of moves it would take to end the game. Our experience shows that it could be between 14 and 100 moves. Therefore, we can say that the game of checkers has an infinite horizon, with *episodic tasks*, marking the end of the game once reached.

In such a case, the full description of our model would look as follows: $(S, A, P, r(s,a), \gamma)$, where $r(s,a)$ is a value of the reward for action a taken in state s, and $\gamma \in [0;1]$ is the *discount rate*. It is used in the formula of *total reward*, where value of γ close to 1 would lead to prioritization of future rewards, and value of γ close to 0 would lead to prioritization of immediate rewards.

2.2. Mathematical goal

That being said, *the goal of our agent is to maximize* the *total reward* – a cumulative measure of the potential run's success. It is defined as follows:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1}, \tag{1}$$

where G_t is the total reward if some run with the sequence of rewards $\{R_t\}$ is selected at time t.

Given that we will always reach an episodic task (the end of the game) at some $t = T$, we can simplify the formula to:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k. \tag{2}$$

This way, the agent's goal would be to maximize the value of G_t by selecting an appropriate sequence of actions.

2.3. Value function

Given the model of the environment, we can now start defining the model of our agent. The simplest way to achieve the set goal of maximizing the total reward would be to somehow use the expected value of total reward from upcoming states, to choose the best one. In order to compute the expected value, we will use the *value function*:

$$\begin{aligned} v^\pi(s) &= \mathbb{E}\left[G_t | S_t = s\right] = \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r(s_k, \pi(s_k)) \middle| S_t = s\right], \end{aligned} \tag{3}$$

where $\pi : S \to A$ is the *policy of the agent*, that defines what action to take in each state.

Another way to look at the expected total reward is to consider both the current state as well as the action taken in it. Such function is called *action-value function*:

$$\begin{aligned} q^\pi(s,a) &= \mathbb{E}\left[G_t | S_t = s \wedge A_t = a\right] = \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r(s_k, \pi(s_k)) \middle| S_t = s \wedge A_t = a\right]. \end{aligned} \tag{4}$$

2.4. Finding an optimal policy of the agent

The task of finding a policy that would satisfy our goal of maximizing the total reward is a non-trivial part of our problem. The available solutions of this task generally fall into two categories: *model-based* and *model-free* methods. The first ones assume that we have at least approximate knowledge of transition matrix space and reward space to estimate the $v^\pi(s)$, and the second ones do not. For that reason, before proceeding to implementation of the agent, we have to decide on the method is applicable to our problem.

2.4.1. Model-based methods.

The two relevant model-based methods that we have reviewed during the course are *Policy Iteration* and *Value Iteration*. Both are iterative algorithms that aim to find the optimal policy of the agent by estimating the value function for all the states.

The first problem with such an approach is that we need to know the value of $p_{ss'}$. In case our enemy is a real person, or another agent, we would not know how does $p_{ss'}$ look like (otherwise we would need to be able to predict actions of the opponent ahead of time). Only when fighting against a Randomized player could we determine the value of $p_{ss'}$: in case of a uniform distribution it would be 1/#available_states.

The second problem is that the number of valid states on the board is simply too high, and it takes too long to enumerate them. The following naive approach Listing 1 was run on AMD Ryzen 7 8845HS for >40 minutes. The enumerated amount of states was 5 450 314, which, when saved in compressed form, took up around 257 MB of memory.

Algorithm 1 Value Iteration

Initialization: $v(s)$ is set arbitrarily for all $s \in S$
repeat
    $\Delta \leftarrow 0$
    for each $s \in S$ **do**
        $v \leftarrow v(s)$
        $v(s) \leftarrow \max_a \sum_{s'} p_{ss'}[r(s,a) + \gamma v(s')]$
        $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
    end for
until $\Delta < \theta$ (a small positive number)
Output a deterministic policy π, such that

$$\pi(s) = \arg\max_a \sum_{s'} p_{ss'}[r(s,a) + \gamma v(s')]$$

Listing 1. DFS states enumeration

```
arr = set()
q = [int(Board())]

while q:
    cur = q.pop()
    cur_b = Board.from_int(cur)
    if cur in arr:
        continue
    arr.add(int(cur))
    for start in cur_b.get_possible_pos():
        for end in cur_b.get_correct_moves(start):
            new_board = Board.from_int(cur)
            new_board.make_move(start, end)
            if new_board.game_state == GameState.NOT_OVER:
                q.append(int(new_board))
```

According to more precise enumeration published by Jan Jaap [2], the number of possible states of the checkers board is 63 838 220 543.

With that in mind, we concluded that it would be highly impractical to iterate over all the states of the board and estimate the value function for each of them. Therefore, we decided to use the model-free methods.

2.4.2. Model-free methods.

During the course we have reviewed the following three model-free methods:

Method	Policy	Backup
Sarsa	On-policy	TD(0)
Q-learning	Off-policy	TD(0)
Monte-Carlo	On/Off-policy	TD(∞)

Here *on-policy* means that the method learns the correct policy while following it, and *off-policy* means that the method learns the optimal policy while following a different one; TD(0) means that the method uses only data from two consecutive states to update the value function, and TD(∞) means that the method uses all the data from the run to update the value function.

Firstly, we eliminated Sarsa from the list of potential methods, because of it being on-policy and given that we can and should do mistakes during the learning process to find an optimal policy quicker [3].

Next, we also eliminated Monte-Carlo method, because of high variance and potential instability [4].

That left us with Q-learning as the only method that we could use to solve the problem.

2.5. (Deep) Q-learning

Generally Q-learning method is pretty similar to the Value Iteration method, but instead of iterating over all possible states, we iterate only over the pairs of states that are the part of the existing run set. Yet, classical Q-learning method still uses table to store the model parameters, which is not feasible in our case.

Therefore, we decided to use the Deep Q-learning method, which uses a neural network to approximate the contents of the table. Here is an overview of the initial algorithm we have based our implementation on [5]:

Algorithm 2 initial Deep Q-learning approach

Require: τ – target network update rate,
    α – learning rate,
    γ – discount rate,
    ϵ – exploration rate,
    s_0 – initial state
Initialization: $D \leftarrow \{\}$,
    $\mathbf{w}_p = \mathbf{0}$,
    $\mathbf{w}_t = \mathbf{w}_p$,
    $t = 0$
loop
    Sample action a_t given ϵ-greedy policy
    for current $\hat{Q}(s_t, a; \mathbf{w}_p)$
    Observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1})
    in replay buffer D
    Sample random minibatch
    of tuples (s_i, a_i, r_i, s_{i+1}) from D
    for (s_i, a_i, r_i, s_{i+1}) in minibatch **do**
        if episode terminated at step $i+1$ **then**
            $y_i = r_i$
        else
            $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}_t)$
        end if
        Do gradient descent step loss for $\mathbf{w}_p$
    end for
    $\mathbf{w}_t = \tau \mathbf{w}_p + (1 - \tau)\mathbf{w}_t$
end loop

3. Methodology

In this section, we will go overall the methodology we followed throughout the project for the implementation (*4. Implementation*) of our RL agents, GUI, and the overall project.

3.1. Overview

The methodology of this research focuses on developing and evaluating RL agents for playing checkers on a 6×6 board. The primary goal is to improve the win rate of the RL model by optimizing reward functions and training methods. This involves multiple phases, including environment setup, agent development, training, and evaluation which will all be explained in depth in further sections.

3.2. Environment Setup

To ensure consistency and reproducibility in training and testing, we developed a standardized checkers environment with the following characteristics:

- **Game Rules:** American Checkers, with modifications for a 6×6 board as explained in the beginning(*1. Introduction*)..
- **Board Representation:** Implemented using Python, storing board states in matrix form which is further explained in (*4. Implementation*).
- **Game Engine:** Created using Python to handle game logic, move validation, and win conditions, again further explanations in (*4. Implementation*).
- **Graphical User Interface (GUI):** Developed for real-time observation and debugging of agents (*3. Overall View of Our GUI*).

3.2.1. RL Algorithm Implementation. As explained in (*2. Theory Overview*), we have selected a Deep Q-learning approach, and so were mainly experimenting with the network architecture. Each agent was trained under different configurations, analyzing performance improvements through modifications in reward functions and exploration strategies.

3.3. Training Methods

- **Random Play Baseline:** Training agents against a purely random opponent to establish initial performance metrics.
- **Self-Play Training:** Allowing agents to learn by competing against progressively improved versions of themselves.
- **Reward Function Optimization:** Testing different reward structures, including:
  - Incremental rewards for advantageous board positions.
  - Penalties for ineffective moves causing faster losses or losing pieces.
  - Bonuses for piece captures and king promotions.

3.4. Performance Evaluation

To assess agent performance, we used the following metrics which are explained in detail in (*5. Results*)..

- **Win Rate:** Percentage of games won against different opponents (mostly the random bot for fairness).
- **Average Game Length:** Number of moves taken before game terminated (limited to 50 when no pieces are captured).

3.5. Fairness in Performance Metrics

To ensure the reliability of our results, we conducted multiple test runs under varied conditions:

- Training agents with different hyperparameters and observing learning stability.
- Comparing our RL agents against ourselves (real human players) for general qualitative assessment while training to achieve faster overview of a model.
- Performing statistical analysis on win rates across 50,000 game iterations (explained in further topics).

By systematically refining our RL models through iterative training and evaluation, we aimed to maximize win rates while keeping an eye on our development cycle timeline.

4. Implementation

In this section, we will discuss about the implementation of our whole project with the basis that was explained in (*3. Methodology*)

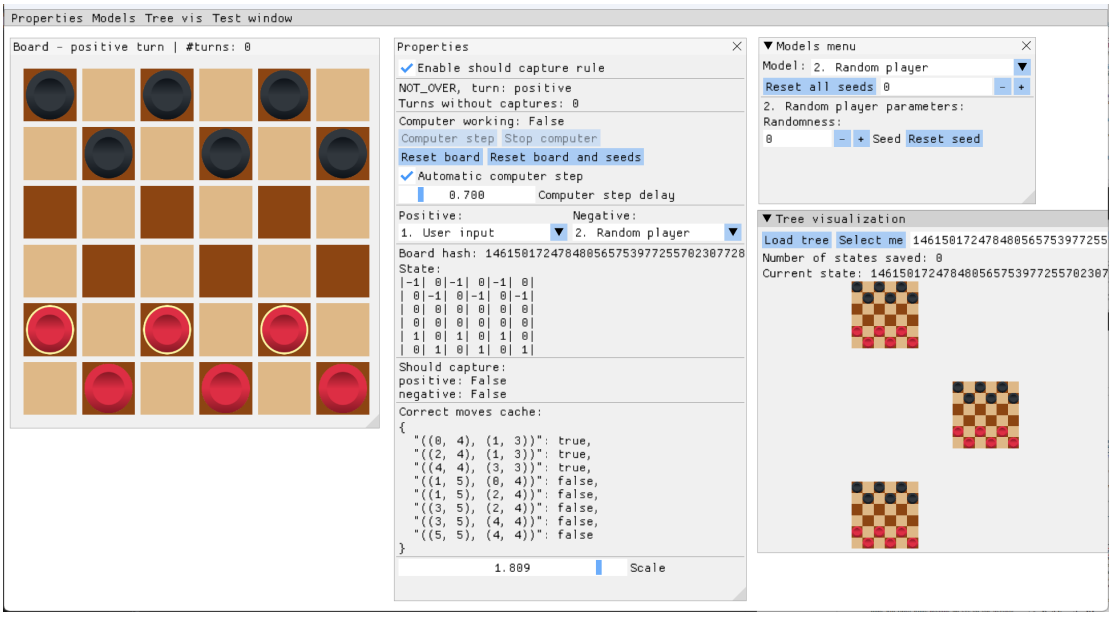

Figure 3. Overall View of Our GUI

4.1. Building the game logic

We have created our agents according to the game rules of American Checkers as explained previously. We started with the game logic of how a board should function and came up with the initial board setup:

Listing 2. Game Board setup

```
self.__board: list[list[int]] = list(map(list,
zip(*[
    [-1, 0, -1, 0, -1, 0],
    [0, -1, 0, -1, 0, -1],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0, 1]
])))
```

with this setup, we declared 1s as positive side checker pieces and -1s as negative side checker pieces where 2 and -2 would be their kings.

Inside of the same class, we decided to add actual gameplay logic allowing for simpler implementations in our RL agents and possibly allowing us to change game rules without changing the agents themselves majorly:

Listing 3. Game Board functions

```
def is_move_correct(self, start: tuple[int, int], end: tuple[int, int]) -> bool:
    return self.__is_move_correct(_c(*start), _c(*end))
# ------------------------------------------------------------------
def get_correct_moves(self, start: tuple[int, int]) -> Iterator[tuple[int, int]]:{...}
# ------------------------------------------------------------------
def get_possible_pos(self) -> Iterator[tuple[int, int]]:
    return self.__get_possible_pos(self.__turn_sign)
# ------------------------------------------------------------------
def make_move(self, start: tuple[int, int], end: tuple[int, int]) -> MoveResult:
    """
    **Warning**: No checks are performed and no turn change is made!

    Would update the board state, check if the piece should be promoted or capture again

    Returns: 2 if king was captured, 1 if simple piece was captured, 0 otherwise
    """
# ------------------------------------------------------------------
def __int__(self) -> int:
    return int.from_bytes(
        self.__bare_byte_repr(),
        byteorder="big"
    )
```

as can be seen in (*3. Game Board functions*) we have used a method where every agent can get the possible playable checker pieces and then also ask the board itself what are the legal moves. After making a choice within the board status, the agent passes the move to the board with the make_move function, moving the game along until the game state either ends or the turn is up to the next player.

4.2. Building the RL agents

In our implementation, instead of approximating the $q(s, a)$ function, we decided to approximate the $v(\hat{s})$, where $\hat{s}$ is the state of the board after agent has made its move but not the enemy. In such a way we still receive information about the board and the action done, but may not deal with several inputs to the neural network, reducing the complexity of the model. We then emulate the value of the $q(s, a)$ function using q_s function in the code:

Listing 4. Deep Q-learning network class

```
class DQN(nn.Module):
    """
    Using structure similar to NNUE:
    https://www.chessprogramming.org/File:StockfishNNUELayers.png

    Observation space: hot-encoded board:
    for each of 18 cells we can be -2, -1, 0, 1, 2 (5 possibilities).
    In total it gives 18 * 5 = 90 possible inputs, out of which at most 12 are on.

    # Action space: 4 possible actions.
    Value function: 1 output.
    """

    def __init__(self):
        super(DQN, self).__init__()

        # layer_sizes = [ 90, 52, 1 ] for other model type
        layer_sizes = [ 90, 50, 50, 1 ]

        layers = []
        prev_size = layer_sizes[0]
        for cur_size in layer_sizes[1:]:
            layers.append(nn.Linear(prev_size, cur_size))
            prev_size = cur_size

        self.layers = nn.ModuleList(layers)

    def forward(self, board: Board) -> torch.Tensor:
        state = board.to_tensor(device)
        for layer in self.layers[:-1]:
            state = F.relu(layer(state))
        return self.layers[-1](state)
```

Listing 5. Using q_a instead of Q-value

```
GAMMA = 0.99 # discount rate

@dataclass
class Action:
    action: tuple[tuple[int, int], tuple[int, int]]
    value: torch.Tensor

def q_s(dqn: DQN, current_state: Board) -> list[Action]:
    """
    Return: list[(new_state, action, immediate_reward, value)]
    """
    ret: list[Action] = []
    for s in current_state.get_possible_pos():
        for e in current_state.get_correct_moves(s):
            next_state = copy.deepcopy(current_state)
            immediate_reward = torch.tensor([next_state.make_move(s, e) * next_state.turn_sign], device=device)
            value = dqn(next_state) * GAMMA + immediate_reward
            ret.append(Action((s, e), value))
    return ret
```

4.3. Training the RL agents

4.3.1. Our hardware. We have used three different computers to train these models and here are our findings.

At first, when we started off with basic award function, we were just playing the games on the cpu and writing the trained models into actual files using Pickle for Python[6], which is a library that allows us to serialize class variables into files and also compresses the data and thus only used CPU power.

For all of these, we had access to computers which all had 32 GBs of DDR5 ram@5600MT/s, then either an AMD 7600x Desktop cpu or an AMD 7735HS

laptop processor or AMD 8845HS laptop processor with single threading all gave us about an episode every 1.6 seconds while training and allowed us to run up to 12 python instances at once when multi threaded.

After creating the first Q-Learning agent via PyTorch[7], we wanted to try out GPU power.

On the desktop computer, we had an Nvidia RTX2060 GPU and on the last laptop an Nvidia RTX3060 GPU for training which unfortunately was giving us around 2 seconds per episode which went below our expectations to beat the CPU time of 1.6 seconds thus making us move forward with training the RL agents with just CPU power.

4.3.2. How we collected data.

Listing 6. Environment response code

```
def make_environment_step(state: Board, action: tuple[tuple[int, int], tuple[int, int]], enemy: IPlayer) -> tuple[Board, torch.Tensor]:
    """
    Returns new state and reward for the given action.
    """
    state = copy.deepcopy(state)
    cur_sign = state.turn_sign
    we_captured = state.make_move(*action) * cur_sign
    reward = 0

    for pos, piece in state:
        if (cur_sign == 1 and piece == 2 and pos[1] > 3) or \
                (cur_sign == -1 and piece == -2 and pos[1] < 2):
            reward -= 1

    enemy_captured = 0
    while state.game_state == GameState.NOT_OVER and state.turn_sign != cur_sign:
        enemy_captured += state.make_move(*enemy.decide_move(state)) * cur_sign * (-1)

    reward += we_captured - enemy_captured
    if state.game_state != GameState.NOT_OVER:
        our_pieces = 0
        enemy_pieces = 0
        for _, piece in state:
            if piece == cur_sign:
                our_pieces += 1
            elif piece == -cur_sign:
                enemy_pieces += 1
            elif piece == 2 * cur_sign:
                our_pieces += 2
            elif piece == -2 * cur_sign:
                enemy_pieces += 2

        reward += 3 * our_pieces / (enemy_pieces + 1)

        if state.game_state == GameState.DRAW:
            reward -= 40
        elif state.game_state == GameState(cur_sign):
            reward += 40
        elif state.game_state == GameState(-cur_sign):
            reward -= 40
        else:
            raise ValueError("Unexpected game state")

    return state, torch.Tensor([reward])
```

4.3.3. How we trained the models.

We wanted to implement training utility per RL agent basis as some agents need excess data and access into the board game logic and decided that creating a method just for this purpose suited for our training requirements:

Listing 7. Q-learning training rutine

```
@dataclass
class TransitionRecord:
    current_state: Board
    next_state: Board
    immediate_reward: torch.Tensor

def optimize_model(memory: list[TransitionRecord]):
    if len(memory) < BATCH_SIZE:
        return

    state_action_values = []
    expected_state_action_values = []

    for r in random.sample(memory, BATCH_SIZE):
        state_action_values.append(max(q_s(policy_net, r.current_state), key=lambda x: x.value.item()).value)
        next_state_value = 0
        if r.next_state.game_state == GameState.NOT_OVER:
            with torch.no_grad():
                next_state_value = max(q_s(target_net, r.next_state), key=lambda x: x.value.item()).value
        expected_state_action_values.append((next_state_value * GAMMA) + r.immediate_reward)

    criterion = nn.SmoothL1Loss()
    loss = criterion(
        torch.cat(state_action_values),
        torch.cat(expected_state_action_values)
    )

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()
```

4.4. Testing the RL agents' performances

We wanted to find a standardized way of comparing the performance of our models and at first came up with the idea of making our agents fight against a random bot and we though 10 thousand games against the bot we were trying to test would be enough.

Our initial findings after that was that the value would change when we would run the tests where the same model would get results varying up to ±30%.That is when we noticed that the 98.52% win rates we were achieving (the inaccurate data we achieved is never shown in this paper) were all incorrect data. That is why we moved over to random bots where we give a fixed initial seed for each one of the 10 thousand games played thus allowing for a more accurate testing across all the models we had instead of possibly lucking out in the "random" bots we had. Which is why later on we also increase the amount of games played to 50 thousand games as it gave us 5 times more seeds to start with and seemed to gave us a lot of data points for an accurate testing.

More details on the code used for testing can be found in (*5. Results*) with the actual numbers.

5. Results

The final phase of our project involved implementing a way of comparing the trained models. We conducted incrementing amounts of tests to see the results. We started by comparing the win rates between 10 thousand games against random bots and incremented it by 10 thousand until we noticed the average was not changing more than ±1% at 50 thousand games. The results demonstrated that our test

approach effectively got us more accurate results and that a testing over games in counting 50 thousand gave us plenty of data points.

And finally, we have used the following test suite in a jupyter notebook with Python to test our RL models where "modelName" in the code is the file name of the model:

Listing 8. RL testing structure

```
stats = []
test_net = DQN()
test_net.load_state_dict(torch.load("modelName.pth"))
------------------------------------------------------------------
for i in range(50000):
    enemy = RandomPlayer(i)
    board = Board()
    while board.game_state == GameState.NOT_OVER:
        while board.game_state == GameState.NOT_OVER and board.turn_sign == 1:
            board.make_move(*enemy.decide_move(board))
        while board.game_state == GameState.NOT_OVER and board.turn_sign == -1:
            with torch.no_grad():
                state_values = max(q_s(test_net, board), key=lambda x: x.value.item())
                board.make_move(*state_values.action)

    pieces = 0
    for _, piece in board:
        pieces += piece != 0

    result = (1 if board.game_state == GameState(-1) else -1) * abs(pieces)

    stats.append(result)
------------------------------------------------------------------
sum(1 for x in stats if x > 0) / len(stats) # win rate
plt.hist(stats) #matplotlib output
```

which returned us the values which can be found at (*1. Win rates of RL agents in 50,000 games against a random bot*)

TABLE 1. Win rates of RL agents in 50,000 games against a random bot

#	Agent Name	Win Rate (%)
1	DQN 90_50_50_1	76.908
2	DQN 90_50_20_1	59.670
3	DQN 90_50_50_1	77.732
4	**DQN 90_50_50_1 t.g. #3**	**84.448**
5	DQN 90_52_1	78.628
0	Random Bot	50.028

*t.g. meaning trained against

Apart from the percentile win rates, we wanted to also visualize how the games concluded using another method. We wanted to see the amount of checker tiles left on the board when the game has concluded. The tile count left on the boards are summed up and the ones for the agent competing against the random bot are counted as positive while the random bot being counted as negative counts.

To better explain this visualization, we wanted to give the following examples:

The worst performing agent can be seen on (*4. The matplotlib of model #2*) where the leftover count of the items on the board is around 40% lose rate in the negatives while around 60% win rate in the positives summing up to 100% of the game play.

The best performing agent can be seen on (*5. The matplotlib of model #7*) where the leftover count of the items on the board is around 12.5% lose rate in the

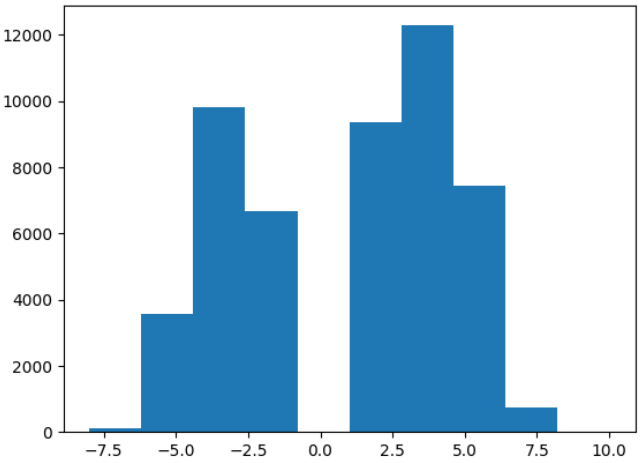

Figure 4. The matplotlib of model #2

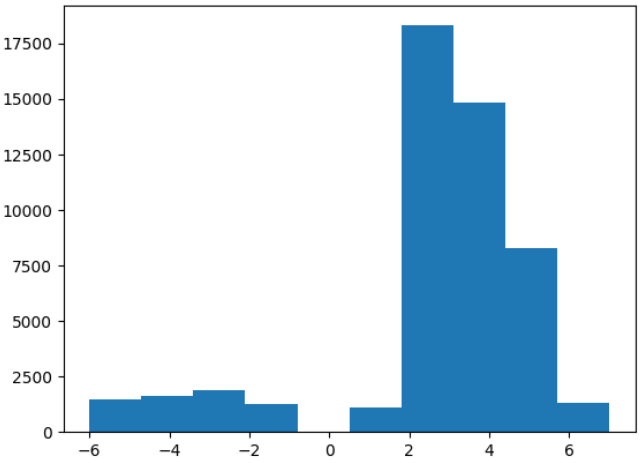

Figure 5. The matplotlib of model #7

negatives while around 87.5% win rate in the positives summing up to 100% of the game play.

In summary, our testing results have encompassed the following key outcomes:

- 📖 Training multiple types of RL agents allowed us to compare their outcomes and allowed us to go deeper into Reinforcement Learning and Q-Learning in specific.
- 💥 Seeing how simple algorithms without neural network performed poorly allowed us to push our research with Q and Deep Q Learning forward granting us really good results.
- 👷 The extensive logic we put into the backend of the board logic simplfied and standardised the freshly implementation of RL agents with new complex RL algorithms inside them
- 😎 The use of an expandible GUI framework allowed us to visualize the gameboard, training and allowed for extensive debugging.
- ✅ We have achieved our goal with a >87% win rate deep q learning agent

All of our achievements in the Github repository with all the models and the win rate data calculated can be found here:[1].

6. Conclusion

Our GUI implementation and extensive prorotypnig done on different RL logics and training methods have proven to do a quite successful job even given the low amount of computer power and the limitations of Python we had.

We were able to raise the win rates of agents from 59% to >84% within only 2 weeks of the development cycle from 15 January 2025 to 30 January 2025 as we planned on our lecture midterm date.

Beyond the limitations of the programming language and our improvised multi threaded training methods, we believe we have come quite close to the limitations of the RL algorithms we have used and received quite good performance.

We have gone from Monte Carlo basic award functions and finite game plays to Neural networks with extensive award functions and even tested their impacts on performances over large quantity of game plays creating large amounts of data points to prove that we have really achieved a Reinforcement Learning agent capable of playing 6x6 checkers.

References

[1] Github Repository holding all the code and data explained in this report. (), [Online]. Available: https://github.com/YeetTheFirst21/checkersRL.

[2] Jan Jaap, *Complexity of checkers and draughts on board sizes 6x6/8x8/10x10/12x12/14x14*. [Online]. Available: https://damforum.nl/bb3/viewtopic.php?t=7817.

[3] Dennis Soemers, *What is the difference between q-learning and sarsa?* [Online]. Available: https://stackoverflow.com/questions/6848828/what-is-the-difference-between-q-learning-and-sarsa/49390009#49390009.

[4] N. S. (https://stats.stackexchange.com/users/43775/neil slater), *When are monte carlo methods preferred over temporal difference ones?* Cross Validated, URL:https://stats.stackexchange.com/q/336976 (version: 2020-05-16). eprint: https://stats.stackexchange.com/q/336976. [Online]. Available: https://stats.stackexchange.com/q/336976.

[5] Adam Paszke, Mark Towers, *Reinforcement learning (dqn) tutorial*. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

[6] Pickle Documentation for Python code serialization. (), [Online]. Available: https://docs.python.org/3/library/pickle.html.

[7] PyTorch Documentation. (), [Online]. Available: https://pytorch.org/docs/stable/index.html.

[8] Imgui Documentation for GUI. (), [Online]. Available: https://pypi.org/project/imgui/.

[9] OpenGL rendering Documentation. (), [Online]. Available: https://pypi.org/project/glfw/.