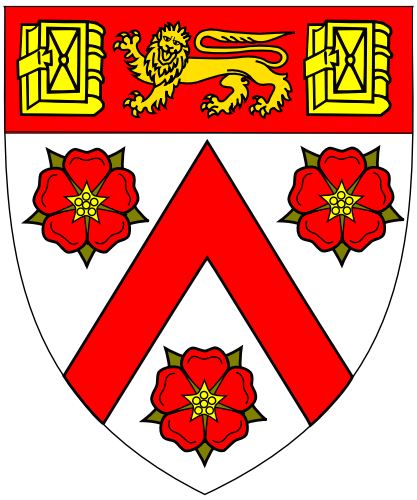
UNIVERSITY OF CAMBRIDGE

Mario Pariona



TRINITY COLLEGE

UNIVERSITY OF CAMBRIDGE

Blokus AI Agents

This dissertation is submitted as part of the requirements for

Part II of the Computer Science Tripos

May 2025

Declaration of Originality

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 2327D, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Date: *19 May 2025*

Proforma

Candidate Number:	**2327D**
Project Title:	**Blokus AI Agents**
Examination:	**Computer Science Tripos – Part II, 2025**
Word Count:	**12,266**[1]
Code Line Count:	**7243**[2]
Project Originator:	**The Candidate**
Project Supervisor:	**Dr Petar Veličković**

Original Aims of the Project

Develop a Blokus environment compatible with the Gymnasium API and implement Blokus AI agents capable of playing the game effectively. The project began with small boards and progressively moved to larger ones. The aim was for the Q-learning agent to defeat a random agent and a minimax-optimized agent at least 75% of the time on a 7×7 board.

Work Completed

The project was a success. The project implemented a Blokus environment compatible with the Gymnasium API and developed several agents capable of playing the game, including the MinimaxAgent,ABPruningAgent, and TabularQLearningAgent for small boards, as well as the QNetworkAgent for larger boards. The project also included a custom approach to training the agent, which was successfully implemented and tested.

Special Difficulties

None.

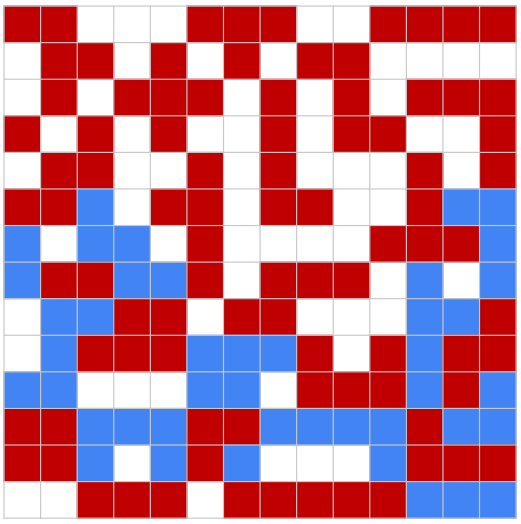
[1] Estimated using `texcount` with `%TC:macro \footnote [include]`, `%TC:group tabular 1 1`, and `%TC:group table 0 1`.
[2] Estimated using `cloc`.

Contents

1 Introduction

Game	Game-Tree Size	State-Space Complexity
Tic-Tac-Toe	10^4	10^3
Connect Four	10^{21}	10^{13}
Checkers	10^{40}	10^{20}
Blokus Duo	$\mathbf{10^{93}}$	$\geq \mathbf{10^{54}}$
Chess	10^{123}	10^{46}

(a) *Estimated **Game-Tree Size**[1] and **State-Space Complexity**[2] of Selected Two-Player Perfect Information Games (rounded to the nearest power of 10). The version of Blokus Duo considered here is the two-player variant played on a 20×20 board.*



(b) *A completed 14×14 Blokus board: the red player has placed all their 21 pieces, while the blue player has only managed to place 11. The red player won by blocking the blue player, leaving them unable to make any further moves.*

This dissertation presents the development of a highly efficient Blokus game environment, running at least $6\times$ faster than previous implementations. A diverse set of agents were trained using reinforcement learning algorithms such as Q-Learning and Deep Q-Networks (DQN), as well as search-based methods including Minimax and Alpha-Beta Pruning, across multiple board sizes. On the 7×7 board, with an estimated game-tree size of $8 \cdot 10^{14}$, the Alpha-Beta Pruning agent discovered a winning strategy for the first player, guaranteeing victory when playing first. On the 10×10 board, with an estimated game-tree size exceeding 10^{32}, the DQN agent achieved a 90% win rate against strong heuristic opponents. A novel approach to agent training is also introduced. These contributions provide a robust platform for future research in game AI and reinforcement learning.

1.1 Overview

Blokus is a strategy board game for 2 to 4 players. Each player begins with 21 uniquely shaped pieces of a distinct color and takes turns placing them on a square board, starting from their corner. Each new piece must touch at least one corner of a piece of the same color, without sharing edges, and pieces may be rotated or reflected. The goal is to cover as much area as possible with pieces. The game ends when no valid moves remain; the player with the most area covered wins. See Figure 1.1b for an example of a finished game.

[1] Number of possible games (i.e., leaf nodes in the game tree) from the initial position.
[2] Number of legal game positions from the initial position.

1.2 Motivation

Why Blokus? Blokus is a game that is easy to learn but difficult to master. While simple heuristics such us playing largest pieces early or blocking the opponent's corners are intuitive, finding a winning strategy remains challenging.

To understand why finding such a strategy is challenging, we begin by estimating the **game-tree size**. For simplicity, we consider the two-player version of the game. An approximate upper bound on the total number of possible states is $(91 \cdot 20 \cdot 20)^{42} \approx 4 \cdot 10^{191}$. This is because there are 91 shapes (including rotations and reflections) and up to 20×20 positions available per turn. Consequently, the branching factor is at most $91 \cdot 20 \cdot 20$, and a total of 42 pieces can be placed during the game. In practice, however, the actual **game-tree size** for small boards (where exhaustive search is feasible) is much smaller than this upper bound. For example, for a 5×5 board (where at most 10 pieces can be placed), about $1.5 \cdot 10^7 \ll 4 \cdot 10^{33} \approx (91 \cdot 5 \cdot 5)^{10}$ states, when using the same pieces and rules.

To obtain a tractable estimate of reachable end-states, we use **Knuth's method** [9], which samples random trajectories and computes an unbiased estimator of tree size based on observed branching factors.

Algorithm 1.2.1 (Knuth's Method for Estimating Search Tree Size) — Sample random paths, observing b_i legal continuations at depth i. The expected search tree size is estimated as:

$$\text{Estimated states} = \mathbb{E}\left[1 + b_1 + b_1 b_2 + b_1 b_2 b_3 + \cdots\right].$$

This algorithm was run with $10,000$ samples, yielding an estimated value of $2.49 \cdot 10^{93}$ states, which is used in Figure 1.1a. The cumulative distribution function (CDF) of the estimated number of states is shown in Figure 1.2.

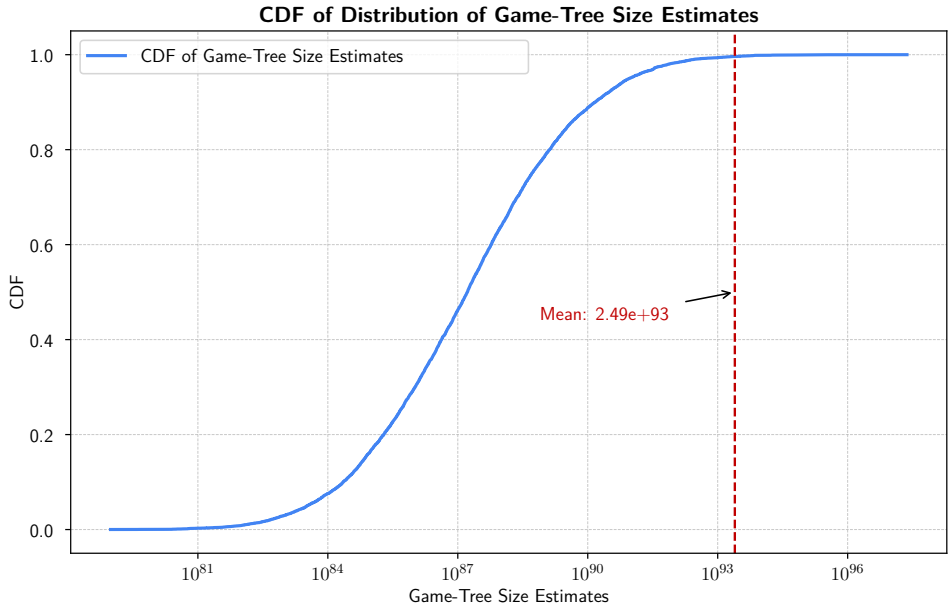


Figure 1.2: *CDF of the estimated game-tree size for a* 20×20 *Blokus board using Knuth's method.*

Note that this estimate is not exact: rare large values may be missed during sampling, which can affect the expected value. Nevertheless, this provides a useful reference for the

game's complexity.

Using this game-tree size estimate $L \approx 2.5 \cdot 10^{93}$ (number of leaf nodes), and noting that each unique end state can be reached by at most $(21!)^2$ distinct sequences (due to ordering of piece placements), we have:

$$L \leq E \times (21!)^2,$$

where E is the number of unique end states. Thus,

$$E \geq \frac{L}{(21!)^2} = \frac{2.5 \cdot 10^{93}}{(21!)^2}.$$

Since the state-space complexity is at least the number of unique end states,

$$\text{State-space complexity} \geq \frac{2.5 \cdot 10^{93}}{(21!)^2} \approx 10^{54}.$$

This gives an upper bound of $\frac{2.49 \cdot 10^{93}}{(21!)^2} \approx 10^{54}$, which is larger than the state-space complexity of Chess, estimated at 10^{46} [18].

1.3 Existing Work

Classical search methods like Minimax and alpha-beta pruning have been widely used for 2-player Blokus [13, 8, 4, 3, 16]. While these methods are effective for smaller games where exhaustive search is feasible, they are not sufficient for game with large state spaces.

Monte Carlo Tree Search (MCTS), on the other hand, estimates the value of an action by simulating random games from the current state. It incrementally builds a search tree by performing numerous rollouts of possible future game states and uses the outcomes of these simulations to guide its exploration toward the most promising moves. This approach has been explored in various works [20, 16, 3, 4, 10]. However, it is computationally expensive, as it requires simulating multiple games before making a decision. Additionally, quick decisions may not always be optimal if insufficient simulations have been performed.

Reinforcement Learning (RL) methods have also been explored. One early example applied tabular Q-learning to a small 5×5 board [2]. More recent efforts have adapted advanced methods like AlphaZero [19], which combines MCTS with deep neural networks to learn optimal play. AlphaZero has also been applied to Blokus [10], although with much smaller compute budgets.

1.4 Objectives

AlphaZero's training primarily used $5,000$ first-generation Google's custom TPUs for game generation and 64 second-generation TPUs for network training, and run for 24 hours to achieve superhuman level in games like Chess, Shogi and Go. It cost $3e22$ FLOPs[3] [19]. While this approach has great potential for this type of games, it is not a practical solution when computing resources are limited.

[3] Floating-Point Operations

Let us revisit our problem by only considering the two-player version of the game. A finite two-player zero-sum game with perfect information is theoretically solvable, meaning that either the first or second player has a guaranteed winning strategy, or the game will end in a tie if both players play optimally. This outcome represents the game's *Nash Equilibrium*[4].

The main objective then is to solve the following problem:

> ***Problem.*** Assuming there is a winning stretegy for the first player, as that seems more likely, find the agent that will always win against any agent.

We begin by exploring small boards, such as 4×4 and 5×5, to understand the game dynamics and develop effective strategies. We then scale up to larger boards, such as 10×10, to test the robustness of most advanced approaches. The main objective of this dissertation is to explore the application of reinforcement learning techniques to the game Blokus.

1.5 Contributions

The contributions of this dissertation are as follows:

- ◇ Computational estimation of the state-space complexity and game-tree complexity for the two-player version of the game.
- ◇ Developed highly efficient Gymnasium-compatible environments: `BlokusEnv`, and `SingleAgentBlokusEnv`. The latter allows specifying a set of agents to operate in the background, with only one agent interacting directly with the environment.
- ◇ Created the `BlokusPiece` class, managed by the singleton `BlokusPieceManager` instance, along with the `BlokusAction` and `BlokusTheme` classes, representing the pieces, actions, and visual themes of the game.
- ◇ A modular `Agent` class serving as a base class for all agents, enabling users to easily create new agents.
- ◇ Implemented several agents, all subclasses of `Agent`, including:
  - `RandomAgent` class.
  - `MinimaxAgent` and `AlphaBetaPruningAgent` classes, both of which support configurable search depth and an optional `use_cache` parameter for caching actions by state.
  - `HeuristicAgent` class, which accepts a custom function to evaluate state-action pairs.
  - `QAgent` class, a tabular Q-learning agent designed for small boards up to 7×7, where the game-tree size is estimated to be around $8 \cdot 10^{14}$ states.
  - `QNetworkAgent` class, which estimates Q-values for state-action pairs using a neural network. The network can be trained with various methods.
  - `MixedAgent` class, which combines multiple agents and selects one to act at each step based on a user-defined probability distribution specified during initialization.

[4] A **Nash Equilibrium** is a game theory concept where no player can gain by changing their strategy if the other remains unchanged, ensuring both play optimally.

- ◇ Introduced the `QNetworkTrainer` class, responsible for training `QNetworkAgent` instances. The `QNetworkAgent` utilizes a configurable neural network architecture for estimating Q-values. Multiple `QNetworkTrainer` implementations are provided.
- ◇ Evaluation of various agents against each other.

2

Preparation

*In this chapter, I outline the necessary preparation for our approach to solving the Blokus problem. I begin by presenting the **rules of Blokus**, followed by an overview of the theoretical background required for our implementation, including **Tree Search Algorithms** and **Reinforcement Learning**. Next, I introduce the **implementation frameworks** used in this work, namely **Gymnasium** and **PyTorch**. I then present the **requirement analysis** and provide a description of the **software engineering tools and methods** used. Finally, I describe our **starting point**.*

2.1 Blokus Rules

To begin, let us understand the Blokus game and its rules. The rules are as follows:

Players: 2 to 4. Each gets 21 Tetris-like pieces of their own color (more precisely, all the free polyominoes[1] made of 1 to 5 squares). Refer to Figure 2.1, which illustrates the 21 pieces.
Board: 20×20 grid.
Placement Rules: Place one piece per turn. Each new piece must touch your own color only at a corner. Your pieces can't touch your own color edge-to-edge. You can touch or block other players freely.
Start: Your first piece must cover your corner of the board.
Turns: Take turns placing one piece. Skip if no legal move.
End: Game ends when no player can place a piece.
Goal: Place pieces to cover the largest total area (i.e., the most squares). Highest area wins.

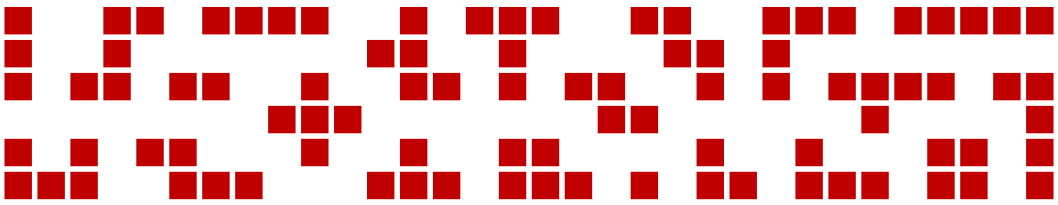


Figure 2.1: *The 21 "pieces" of Blokus each player can use. These shapes can be rotated and reflected.*

After discussing the rules of the game, we proceed to present Tree Search Algorithms, beginning with the Negamax algorithm—a perspective-based variant of Minimax—and its Alpha-Beta pruning enhancement.

[1] A *free polyomino* is a shape formed by joining one or more equal-sized squares edge to edge, where rotations and reflections are considered identical.

2.2 Tree Search Algorithms

2.2.1 Negamax: A Symmetric Perspective on Minimax

Traditional Minimax alternates between MAX and MIN nodes to simulate optimal play for both players. However, in two-player zero-sum games, the value for one player is simply the negative of the other's—$V_{\text{opponent}} = -V_{\text{player}}$. Negamax takes advantage of this symmetry by using a single recursive function that always chooses the maximum from the current player's perspective.

This replaces the classical Minimax form:

$$V(n) = \begin{cases} \max\limits_{e \in \text{Children}(n)} V(e), & \text{if player}(n) = \text{MAX} \\ \min\limits_{e \in \text{Children}(n)} V(e), & \text{if player}(n) = \text{MIN} \end{cases}$$

with the symmetric Negamax expression:

$$V(n) = \max_{e \in \text{Children}(n)} \left(\delta(n,e) \cdot V(e)\right), \quad \delta(n,e) = (-1)^{\text{switch}(n,e)}$$

where $\text{switch}(n,e) = 1$ if the player changes between node n and node e, and 0 otherwise.

Negamax simplifies the logic of game-tree search by using a single function for both players, eliminating the need to alternate between max and min operations. This formulation aligns naturally with reinforcement learning, where agents consistently evaluate states from their own perspective. Although traditional approaches assume alternating turns, Negamax remains valid even when this is not the case. Most importantly, this perspective is more intuitive: players focus on maximizing their own score, while minimizing the opponent's score emerges as a natural consequence of optimal play.

2.2.2 Alpha Beta Pruning on Negamax

As it is the case with Minimax, Negamax can be optimized using alpha-beta pruning, though the implementation is slightly different. In Negamax, instead of alternating between maximizing and minimizing at each node, the same function is applied for both players by negating the values. Alpha and Beta values in Negamax represent the best possible score the current player can guarantee for themselves (α) and the worst possible score the opponent can force them into (β).

Alpha-beta pruning works by comparing these values during the search process. If at any point $\alpha \geq \beta$, it means that further exploration of this branch is unnecessary, as it cannot yield a better result than what has already been found. This condition indicates that the current branch cannot improve the best guaranteed outcome for the current player, as the opponent would avoid any moves leading to this state. Consequently, the algorithm prunes the branch to save computational effort. Refer to Algorithm 1 for the pseudocode of the alpha-beta pruning algorithm.

Algorithm 1 Alpha-Beta Pruning Algorithm (Negamax Version)

```
function NegamaxWithPruning(node, depth, α, β)
    if depth = 0 or node is terminal then
        return evaluate(node)
    end if
    best_value ← −∞
    for each child in node.children() do
        if switch(node, child) = 1 then
            value ← −NegamaxWithPruning(child, depth-1, −β, −α)
        else
            value ← NegamaxWithPruning(child, depth-1, α, β)
        end if
        best_value ← max(best_value, value)
        α ← max(α, best_value)
        if α ≥ β then
            break                                   ▷ prune remaining branches
        end if
    end for
    return best_value
end function
```

2.3 Reinforcement Learning

Reinforcement learning (RL), at its core, involves teaching an agent to choose an action in an environment to maximize cumulative rewards. This process is guided by the interaction between the agent and the environment, where the agent learns from feedback in the form of **rewards**.

RL can be categorized by *action/state spaces* (continuous or discrete), use of *environment models* (model-based or model-free), or *learning approach* (on-policy or off-policy). Additionally, RL methods can be **value-based**, **policy-based**, or a combination of both (**actor-critic**). Here, we focus on value-based methods for *finite, discrete spaces* with *limited rewards*.

Firstly, we introduce the concept of **Markov Decision Processes (MDPs)** in Section 2.3.1, a formal framework for modeling decision-making problems in RL. Next, we define some core terms before formally defining RL in Section 2.3.2. We then introduce **Value Functions** and **Bellman Equations** in Section 2.3.3 and Section 2.3.4, key components in most RL algorithms, followed by an explanation of **Q-learning** and **Deep Q-Networks (DQN)** in Section 2.3.5 and Section 2.3.6 respectively, two widely used RL algorithms.

2.3.1 Markov Decision Process (MDP)

Definition 2.3.1 (Markov Property) — A process is said to have the **Markov property** if the *future state* of the process depends only on the **current state** and **current action**, and not on the *previous states* or *previous actions*. Formally, a process satisfies the Markov

property if:

$$\Pr(s_{t+1}, r_t | s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = \Pr(s_{t+1}, r_t | s_t, a_t) \tag{2.1}$$

for all time steps t.

Here, s_t denotes the state at time t, a_t the action taken at time t, and r_t the reward received after taking action a_t in state s_t.

A **Markov Decision Process (MDP)** is a mathematical framework used to model decision-making problems in situations where outcomes are partly random and partly under the control of a decision-maker. More specifically, a reinforcement learning problem in which the environment dynamics satisfy the Markov property is called a Markov Decision Process. Formally, an MDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R)$, where:

- $\mathcal{S}$ is the *state space*, representing the finite set of all possible states that describe the configuration of the system.
- $\mathcal{A}$ is the *action space*, representing the set of all possible actions. For each state $s \in \mathcal{S}$, the subset $\mathcal{A}_s \subseteq \mathcal{A}$ denotes the actions available in state s.
- $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the *state transition distribution function*, where $\Delta(\mathcal{S})$ denotes the set of probability distributions over $\mathcal{S}$. For all $s \in \mathcal{S}$, $a \in \mathcal{A}_s$, and $s' \in \mathcal{S}$, we define:

$$P(s' \mid s, a) := \Pr(S_{t+1} = s' \mid S_t = s,\ A_t = a).$$

- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function*, where $R(s, a, s')$ represents the immediate reward (or expected immediate reward) received after transitioning from state s to state s' due to action a, for all $s \in \mathcal{S}$, $a \in \mathcal{A}_s$, and $s' \in \mathcal{S}$.

2.3.2 The Agent-Environment Interaction

Before delving into the details, let us define some key terms that will be essential for understanding the rest of the chapter.

Agent The learner or decision-maker that interacts with the environment.
Environment The external system with which the agent interacts.
Action A choice made by the agent that affects the state of the environment.
State A representation of the current situation of the environment.
Reward A scalar value that provides feedback on the immediate benefit of the action taken.
Policy A strategy used by the agent to determine its actions based on the current state. It can be deterministic or stochastic.

With these key terms defined, Reinforcement Learning (RL) can be formally described as a type of machine learning where an agent interacts with an environment in discrete time steps. At each time step, the *agent* receives a **state** from the *environment*, selects an **action** based on a *policy*, and receives a **reward** from the environment. The goal of the agent is to learn a policy that maximizes the expected cumulative reward over time.

More specifically, at each discrete time step t, the agent receives some representation of the environment's state, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space, and on that basis selects an action, $A_t \in \mathcal{A}_{s_t}$, where $\mathcal{A}_{s_t}$ is the set of actions available in state s_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in \mathbb{R}$, and finds itself in

a new state, $s_{t+1} \in \mathcal{S}$. The interaction between the agent and the environment is illustrated in Figure 2.2.

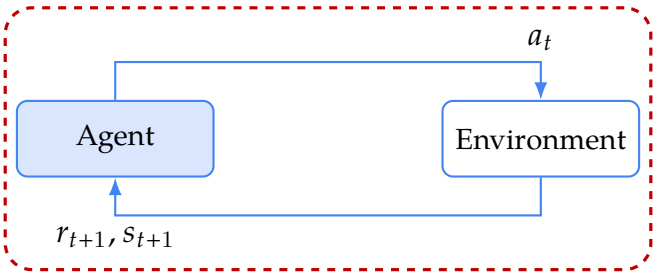


Figure 2.2: *Interaction between Agent and Environment in Reinforcement Learning*

The agent's behavior is defined by a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, which maps each state $s \in \mathcal{S}$ to a distribution over available actions $\mathcal{A}_s \subseteq \mathcal{A}$. We denote:

$$\pi(a \mid s) := \Pr(a_t = a \mid s_t = s),$$

so that $\pi(\cdot \mid s)$ represents the full distribution over actions in state s.

RL methods aim to optimize the agent's policy to maximize the expected cumulative reward based on its interactions with the environment.

2.3.3 Value Functions & Bellman Equations

To accomplish the goal of maximizing cumulative rewards, the agent must estimate the quality of being in a particular state or taking a specific action within a state. This is achieved through the use of value functions, which quantify expected future returns.

We assume that the agent follows a fixed policy π, meaning that for all time steps t, actions are sampled according to $a \sim \pi(\cdot \mid s)$, unless stated otherwise.

Definition 2.3.2 (Value Functions $V(s)$ and $Q(s,a)$) — The value function for a state s, denoted as $V^\pi(s)$, represents the expected return when starting from state s and following a policy π. It is defined as:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \,\middle|\, S_0 = s\right]$$

The action-value function, denoted as $Q^\pi(s,a)$, represents the expected return when taking action A in state S and then following a policy π. It is defined as:

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \,\middle|\, S_0 = s, A_0 = a\right]$$

Here, $\gamma \in [0,1]$ is the *discount factor*, which controls the trade-off between immediate and future rewards.

The Bellman equations provide a recursive definition of value functions, expressing the value of a state or state-action pair in terms of the values of its successors.

Definition 2.3.3 (Bellman Equations for $V^\pi(s)$ and $Q^\pi(s,a)$) — The Bellman equation for the state-value function is given by:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s),\, s'\sim P(\cdot|s,a)}\left[R(s,a,s') + \gamma V^\pi(s')\right]$$

and the Bellman equation for the action-value function is given by:

$$Q^\pi(s,a) = \mathbb{E}_{s'\sim P(\cdot|s,a),\, a'\sim\pi(\cdot|s')}\left[R(s,a,s') + \gamma Q^\pi(s',a')\right]$$

where $P(s' \mid s,a)$ is the probability of transitioning from state s to state s' after taking action a.

2.3.4 Optimal Value Functions

Optimality in value functions refers to the concept of finding the best possible value function that maximizes the expected cumulative reward for an agent in a reinforcement learning problem. The optimal value functions are defined as follows:

Definition 2.3.4 (Optimal Value Functions) — The optimal value functions represent the maximum expected return achievable under any policy π. They are defined as follows:

$$V^*(s) = \max_\pi V^\pi(s), \quad Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

The optimal value functions satisfy the Bellman optimality equations, which are recursive relationships that define the value of a state or state-action pair in terms of the values of its successors:

Definition 2.3.5 (Bellman Optimality Equations) — The Bellman optimality equations for the optimal value functions are given as follows:

For the optimal state-value function $V^*(s)$:

$$V^*(s) = \max_{a\in\mathcal{A}_s}\left[\mathbb{E}_{s'\sim P(\cdot|s,a)}\left[R(s,a,s') + \gamma V^*(s')\right]\right]$$

For the optimal action-value function $Q^*(s,a)$:

$$Q^*(s,a) = \mathbb{E}_{s'\sim P(\cdot|s,a)}\left[R(s,a,s') + \gamma \max_{a'\in\mathcal{A}_{s'}} Q^*(s',a')\right]$$

The optimal policy π^* can then be defined as $\operatorname{argmax}_{a\in\mathcal{A}_s} Q^*(s,a)$, which selects the action that maximizes the expected return from state s.

As we can see, having those value estimates enables us find the optimal policy, but how does an agent actually learn these estimates in practice? In most cases, the agent does not have access to the transition probabilities and rewards of the environment. Instead, it must learn these values through **experience**—interaction with environment and updating estimates based on observed rewards and transitions. We will next introduce the concept of Q-Learning, a popular value-based RL algorithm that learns the action-value function directly from experience.

2.3.5 Value-Based Methods: Q-learning

Imagine you are learning to solve a maze. At every step, you remember how good it was to take a certain direction (action) in a particular location (state). If it led you closer to the goal (higher reward), you adjust your memory to value that decision more. Q-learning formalizes this intuition.

Q-learning [22] is a model-free[2], off-policy[3] RL algorithm that enables the agent to learn Q^* through iteration.

During an episode, the agent collects transitions of the form (s_t, a_t, r_t, s_{t+1}). At each step, the Q-value is updated using

$$Q_{i+1}(s_t, a_t) \leftarrow Q_i(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \max_{a'} Q_i(s_{t+1}, a') - Q_i(s_t, a_t)\right)$$

Here, α is the learning rate, and γ is the discount factor. Each update slightly shifts our Q-value toward the better estimate based on the new reward and future value.

This iterative process becomes clearer when rewritten as:

$$Q_{i+1}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q_i(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \max_{a'} Q_i(s_{t+1}, a')\right)$$

2.3.6 Scaling Up with Deep Q-Networks (DQN)

Tabular Q-learning, which stores a Q-value for every state-action pair (s, a), is effective only in environments with small, discrete state and action spaces. Moreover, it does not generalize across state-action pairs, making it unable to estimate Q-values for previously unseen pairs. Instead, a function approximator is typically used to estimate the action-value function: $Q(s, a; \theta) \approx Q^*(s, a)$, where θ represents the parameters of the approximator. Neural networks are often used for this purpose because of their inductive bias and their capacity as *universal approximators*[4]. However, introducing non-linear function approximators means that convergence is no longer guaranteed. Despite this, such approximators are essential for learning and generalizing in large state spaces. When neural networks are used in this context, they are called Q-networks.

The core idea remains the same: the agent interacts with the environment, observes transitions, and updates the Q-values (now approximated by the network) to get better at predicting long-term rewards.

[2] It does not require a model of the environment.

[3] It learns the value of the optimal policy independently of the agent's actions.

[4] The **Universal Approximation Theorem** states that a feedforward neural network with a single hidden layer can approximate any continuous function on a compact domain, given a sufficient number of neurons.

As described by the original paper [15], we aim to minimize the loss function:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right]$$

where $y_i = \mathbb{E}_{s'\sim P(\cdot|s,a)}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right]$ and $\rho(\cdot)$ represents the distribution of state-action pairs found during interaction with the environment, also referred to as the *behaviour distribution*.

By taking the gradient of the loss function with respect to the parameters θ_i, we derive the update rule:

$$\theta_i \leftarrow \theta_{i-1} + \alpha \nabla_{\theta_i} \mathcal{L}(\theta_i) \tag{2.2}$$

where α is the learning rate.

This again in practice is not possible since we do not have access to the distribution $\rho(\cdot)$, and we cannot compute the expectation over the next state s'. To overcome these challenges and stabilize training, two fundamental techniques are used.

Experience Replay: This technique, introduced by Mnih et al. [15], involves storing past transitions in a replay buffer and sampling random batches from it during training. By breaking the correlation between consecutive transitions, it stabilizes the learning process. Specifically, transitions at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$, are stored in a dataset $D_t = \{e_1, \ldots, e_t\}$, which is pooled over multiple episodes (an episode ends upon reaching a terminal state).

Target Networks: This technique was introduced in the DQN architecture by Mnih et al. [14]. It employs a separate target network, which is updated less frequently than the main network, to compute the target Q-values. This reduces oscillations and mitigates the risk of divergence during training. Specifically, every C updates, the current network is cloned to create a target network, which is then used to generate the Q-learning targets y_j for the subsequent C updates. During training, the target network Q^- remains fixed, and no gradients are propagated through it.

An adapted version of the DQN algorithm from [14] is presented in Algorithm 2.

As demonstrated with DQN, neural networks are used to approximate Q-values when dealing with large or continuous state spaces. To efficiently process image-like data—such as the board representation in Blokus—we require neural network architectures designed for this purpose. In the following section, we introduce **Convolutional Neural Networks (CNNs)**.

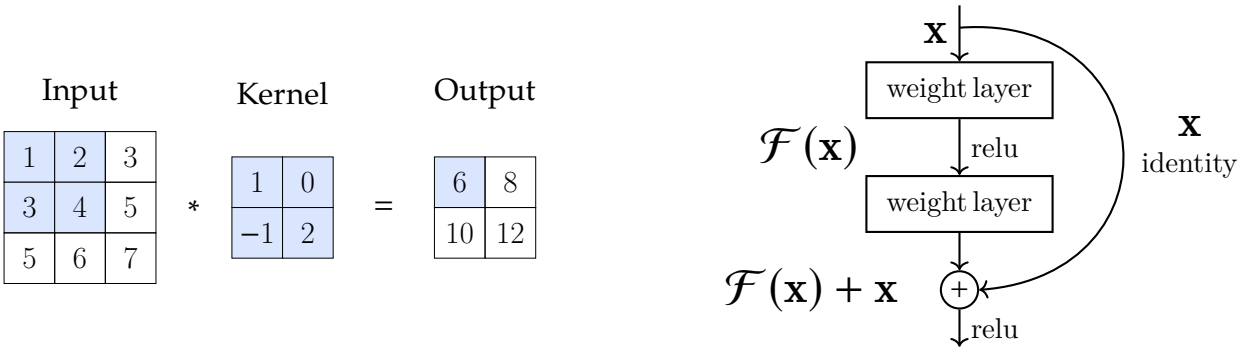
2.3.7 Convolutional Neural Networks (CNNs)

Designing effective features manually for Fully Connected Neural Networks becomes hard for image-like inputs. Convolutional Layers [11] address this by capturing local patterns such as edges, corners, and shapes through the application of small filters across the input matrix (including its channels[5]). This process, known as the convolution operation, is illustrated in Figure 2.3a with a simple example.

This inherent *inductive bias* [6] makes CNNs very effective at learning from images or grid-like data (like frames in video games).

[5] A single layer of data in a multi-layered input, such as RGB channels in an image.
[6] Set of assumptions a model makes to generalize from the training data to unseen data.

Residual Networks (ResNet). Stacking multiple Convolutional Layers enables the extraction of increasingly complex features, allowing the network to approximate more intricate functions. However, this can make training challenging due to issues like vanishing gradients. Residual Networks (ResNet) [6] address this by introducing **skip connections**—shortcuts that bypass one or more layers. These connections facilitate easier training, even for networks with hundreds of layers. The basic structure of a ResNet block, including the skip connection, is illustrated in Figure 2.3b.



(**a**) *Convolution operation.*

(**b**) *ResNet block with skip connection. Adapted from* [6].

Figure 2.3: *Illustrations of the convolution operation and a ResNet block with skip connection.*

2.4 Implementation Frameworks

Python was selected as the programming language because of its compatibility with Gymnasium and PyTorch, which are well-suited for this project. These frameworks will be introduced in this section.

2.4.1 Reinforcement Learning Environment: Gymnasium

The **Gymnasium framework** provides a standardized API for reinforcement learning, simplifying the *development*, *testing*, and *logging* of RL algorithms. It seamlessly integrates with popular RL libraries such as **Stable-Baselines3** and **RLlib**.

The framework defines the `Env` interface, which includes an `observation_space` and an `action_space`. It also introduces two essential methods: `reset()` and `step()`. Additionally, the `Wrapper` class serves as a modular tools to modify the environment's behavior (e.g., preprocessing, reward shaping) without altering the environment itself.

The main API methods used to interact with a Gymnasium environment are summarized in Table 2.1:

2.4.2 Deep Learning Framework: PyTorch

Most Deep Learning algorithm rely on a technique called backpropagation, which calculates the gradient of a loss function (to be minimized) with respect to the network's weight

Method	Description
`step()`	Performs action in environment. Returns the next observation, reward, terminated flag, truncated flag, and extra info.
`reset()`	Resets the environment to an initial state and returns the first observation and info.

Table 2.1: *Key Methods of Gymnasium `Env` class.*

parameters. PyTorch is a widely used Python framework that provides an intuitive interface for neural networks and automatic differentiation.

PyTorch's core is the `torch.Tensor` class, which enables GPU-accelerated operations and automatic gradient tracking via `autograd`. The `torch.nn` package offers modular tools like layers, loss functions, and optimisers for building and training neural networks.

2.5 Requirement Analysis

The requirements stay the same as in the Project Proposal (Appendix B): the core project aims to build a Gymnasium-compatible environment. We then aim to implement tabular Q-learning agents that can perform well on a 7×7 board against random and minimax-optimized opponents.

For the extensions, I aim to address the challenge of larger boards using Deep Q-learning agents. Additionally, I plan to explore the development of agents capable of performing well against any opponent, ideally discovering optimal strategies, using a similar approach to DQN. The extension are exploratory in nature, requiring substantial experimentation on possible solutions.

In Table 2.2, I break down the project into implementation modules and present a dependency analysis.

2.6 Software Engineering Tools and Techniques

2.6.1 Language, libraries, tools

The project was developed using Python 3.12.9 with `venv` 20.27.1 for environment management. The third-party libraries used in the project are summarized in Table 2.3.

ID	Task Name	Priority	Depends On
1	Estimate game-tree size and state-space complexity	Medium	–
2	Implement `BlokusPiece` & `BlokusAction`	High	–
3	Implement `step()` method	High	2
4	Finish `BlokusEnv` implementation	High	2, 3
5	Implement `SingleAgentBlokusEnv`	High	4
6	Implement `Agent` (base class)	High	4
7	Implement `RandomAgent`	High	6
8	Implement efficient `step()` method	Medium	3
9	Implement `MinimaxAgent` and `AlphaBetaPruningAgent`	High	6
10	Implement `HeuristicAgent`	High	6
11	Implement `TabularQLearningAgent`	High	6
12	Implement `CacheManager` for Minimax/$\alpha\beta$ Agents	Medium	9
13	Train `TabularQLearningAgent` on small boards	High	5, 9, 12
14	Define `QNetwork` interface	Medium	13
15	Implement `QNetwork`s	Medium	16
16	Implement `QNetworkAgent`	Medium	14, 16
17	Implement `DQNTrainer` (subclass of `QNetworkTrainer`)	Medium	7, 16
18	Run `DQNTrainer`	Low	5, 17, 18, 19
19	Evaluate agents	Medium	7, 9, 10, 13, 18
20	Try to implement a different `Trainer` for `QNetworkAgent`	Low	15, 16
21	Implement `BlokusTheme`	Low	4
22	Make UI for playing	Low	5, 6

Table 2.2: *Dependency analysis of the implementation components. Points* 14 *onward—except for point* 19*—are extensions.*

#	Library	Version	Purpose
1	`torch`	2.3.1	Deep learning framework.
2	`torchvision`	0.18.1	Image transforms and CNNs.
3	`gymnasium`	1.0.0	RL environment standard.
4	`pygame`	2.6.1	Game rendering and interaction.
5	`matplotlib`	3.9.2	Data visualization.
6	`numpy`	2.1.2	Numerical computing.
7	`pytest`	8.3.4	Unit testing framework.
8	`wandb`	0.19.4	Experiment tracking.

Table 2.3: *Third-party libraries used in the project.*

2.6.2 Development methodology

This is a Machine Learning project by nature, and the development methodology followed **spiral** principles [1], focusing on identifying and managing risks, particularly concerning the time performance of the environment and agent decisions. In each iteration, I concentrated on a specific goal to develop and test. Afterward, I assessed the results and identified potential risks before defining the next goal. While I set **initial guiding milestones**—such as the

implementation of the environment, the Alpha-Beta pruning agent, the tabular Q-learning agent, and performance improvements for the environment—the direction of the project was also shaped by continuous **evaluation and refinement** at each stage, in line with the iterative nature of the Spiral model.

For memory profiling, tools such as `tracemalloc`, `memory_profiler`, `psutil`, and Python's garbage collector were used. Additionally, PyTorch's native tracking utilities were employed to monitor memory usage during training. For performance analysis, `cProfile` was used to profile the code and identify CPU performance bottlenecks.

I adhered to good practices, keeping the code modular and well-organized. To maintain high-quality, readable, and reusable code, Python's built-in docstring conventions (Google style) and type hints were consistently used throughout the project.

2.6.3 Hardware, version control, backup

I developed the code with Visual Studio Code on my laptop (Apple MacBook Pro M2 Pro 16GB RAM, 512 GB SSD). Model training and testing were performed on Google Colab using GPUs (T4, L4, A100) to enable exhaustive experimentation. Git was used for version control. The Google Drive sync feature was used for daily backups.

2.7 Starting Point

Prior to this project, I had no experience with reinforcement learning or deep learning. To prepare, I studied Q-learning and related methods using books and videos during the summer. Additionally, I used the Gymnasium API to develop a basic tic-tac-toe agent with tabular Q-learning and followed PyTorch's introductory tutorial. Although I was familiar with Git, I had not used it extensively for large-scale projects.

3

Implementation

In this chapter, we will discuss the implementation of our approach to solving the **Blokus problem for the 2-player version**. We will begin by describing the environment we created for the game in Section 3.1, followed by a detailed explanation of the agents I implemented for it.

Our approach to solving the Blokus problem is incremental. We begin by addressing the problem for small boards and gradually progress to larger boards. That being said, in Section 3.2 we focus on agents that perform well the 2-player version of Blokus on small boards, introducing the `MinimaxAgent`, `ABPruningAgent`, and `TabularQLearningAgent`. In Section 3.3, we focus on agents that perform well on larger boards, specifically the 10×10 case which can be extended to any board size. We also describe the `QNetworkAgent`, which is trained using two different methods: the DQN algorithm and a custom approach we developed.

3.1 Blokus Environment

The Blokus environment is the core component of this project, responsible for managing the game state and interacting with the agents. Unlike typical Gymnasium environments designed for a single agent, our environment supports multi-player gameplay with 2 to 4 competing players. To address this, I implemented the `BlokusEnv` environment to take a single action input at each step and internally manage the player turn, applying the action to the appropriate player.

To explain the implementation clearly, this section begins by first define key terminology that will be used throughout the rest of the dissertation. Next, we describe the main methods required, as described in Section 2.4.1—`reset` and `step`—before detailing the internal mechanisms and optimizations that ensure efficient operation of the environment. Finally, we introduce the `SingleAgentBlokusEnv` subclass, which allows a single agent to play against built-in opponents whose actions are hidden.

Before diving into the implementation, we start by defining some important terms.

3.1.1 Blokus Terminology

We start by defining the **game elements** in real world, in natural language. Then we go on to defining additional notation to help us model useful cells. We finalise by defining these same definitions in a more formal way that will allow us to work with the later in an abstract way.

▶ **Game Elements**

Notation 3.1.1 (Board)**.** A *board* is the grid on which the game is played. It is made up of a collection of **cells** arranged in a two-dimensional array.

Notation 3.1.2 (Cell)**.** A *cell* is a single space on the game board where a **unit square** can be placed. It is the basic unit of the game board.

Notation 3.1.3 (Unit Square)**.** A *unit square* is the smallest component that makes up a **piece**. It is a square of size 1×1 that can be placed on the game board.

Notation 3.1.4 (Piece)**.** A *piece* is a collection of one or more **unit squares** arranged in a particular shape. It represents the player's movable object on the board. Refer back to the Figure 2.1 for a visual representation of the pieces.

We will then define a couple of cells that will help us model the concept of "placement feasibility."

Notation 3.1.5 (Legal Cell)**.** A *legal cell* is a **cell** where a **unit square** can be placed according to the game's rules. It is a cell that is not occupied by another piece and is not a **locked cell**.

Notation 3.1.6 (Corner Cell)**.** [1] A *corner cell* is a **cell** that allows a legal connection, such as a corner touch, with an existing **piece**. It is a cell that can be used to expand the player's piece on the board.

Notation 3.1.7 (Locked Cell)**.** A *locked cell* is a **cell** that is adjacent to a cell ocuppied by a **square** of the player's color, therefore not a **legal cell**, by the game's rules.

▶ Formal Definitions

In order to understand this game in a more low-level, let's define the game mathematically. We define this in an abstract language.

- ◇ $\mathsf{Players} \subset \mathbb{Z}$ is the set of **players** in the game, each identified by a unique integer representing their turn order. For example, in the Blokus Duo version, $\mathsf{Players} = \{1, 2\}$.
- ◇ $\mathsf{Cells} \subset \mathbb{N}^2$ is the set of all **cells** on the board. For a 14×14 Blokus Duo board, this is defined as:
$$\mathsf{Cells} = \{(x, y) \in \mathbb{N}^2 \mid 0 \leq x < 14,\ 0 \leq y < 14\}.$$
- ◇ Pieces is the set of all **pieces** available in the game, where each piece is a specific instance of a **shape** being placed on the board. The game includes 91 distinct pieces (considering rotations and reflections) but only 21 unique shapes, each composed of one or more connected **unit squares**.
- ◇ S denotes the set of all valid game states — that is, every configuration the board and players can be in at any given time.
- ◇ A is the **action space**, representing all valid moves a player can make from any given state.
- ◇ δ is a (partial) **transition function** that defines how the state of the game evolves after a player takes an action. Formally:
$$\delta : \mathsf{S} \times \mathsf{A} \rightharpoonup \mathsf{S}, \quad s_{i+1} = \delta(s_i, a_i) \text{ if defined.}$$

  This function is partial because not all actions are valid in every state.

[1] The term "expander" was used in the implementation, but "corner" is used here for easier explanation.

3.1.2 Reset and Step Functions

The `reset()` function initializes the board state as an empty 2D `numpy` array. Each piece is defined using the `BlokusPiece` class, which essentially stores a list of coordinates of the **unit square** that compose it, relative to the top-left corner of the smallest rectangle enclosing it—i.e., with minimum x and y coordinates set to 0. An action is represented using the `BlokusAction` class as a tuple of the form (piece, x, y).

We create a singleton class called `BlokusPieceManager` to efficiently handle all pieces and their transformations, including rotations and reflections. Essentially, a dictionary of the form `shape_id`$\rightarrow$`List[BlokusPiece]` is constructed, where `piece_id` is a unique identifier for each shape.

When an action is passed to the `step()` function, the board state is updated by placing the piece at the specified coordinates. For each coordinate in the piece, the board is modified as `board`$[x + $`coordinate.`$x][y + $`coordinate.`$y] =$ `player_id`, and returns the new state of the board, along with the size of the piece as the reward.

Not too hard, right? Looking ahead, as part of the `step()` function, we will also compute the valid actions for the next player and return them in the observation. This approach eliminates the inefficiency of agents individually verifying the validity of each action. This is particularly important in environments with a large action space where many actions are invalid such as our 20×20 board with an approximate action space of $20 \times 20 \times 91$. On average, fewer than 0.5% of these actions are valid. For instance, in this case, a Random agent would be expected to choose one of the **valid** actions randomly, making it essential to precompute and provide these actions. Additionally, the environment must inform the agent when the game is over, which occurs when no further actions are possible. Therefore, the environment should be capable of detecting this condition inherently.

This introduces a new challenge to address:

> ***Problem.*** Find all possible actions given the current state of the board efficiently.

3.1.3 Compute Possible Actions

Blokus is unique in that the number of valid actions in any given state can range from 0 to over 1000. In this section, we will explore efficient strategies to address this problem. See Figure A.1 for a visual representation of the possible actions.

Strictly speaking, the possible actions can be computed in constant time since all the parameters are upper bounded; however, for the sake of this discussion, we will use a loose definition of time complexity to compare the algorithms.

For all these algorithms, we assume that the board is a square of size $N \times N$. Let SquaresPerPiece denote the maximum size of a piece, and let CornerCells represent the number of **corner cells** for the current player whose turn it is.

▶ First Idea: Naive Brute Force

The naive approach to computing possible actions is to try placing each piece in every possible position on the board and verify whether it meets the placement criteria. For each of the 91 pieces, if the shape hasn't been used, we try all valid board positions—specifically, $(N - \text{piece.x}) \cdot (N - \text{piece.y})$. When placing a piece, we ensure that:

- ◇ **No overlap** with any previously placed piece.
- ◇ At least **one** **unit square** of the piece must touch a piece of the **same color** at a corner.
- ◇ **No** **unit square** of the piece may touch any piece of the **same color** along an edge.

Complexity: $O(N^2 \cdot |\text{Pieces}| \cdot \text{SquaresPerPiece})$

In Section 3.2.1, we will see how an exhaustive search over $1.5 \cdot 10^7$ states for a 5×5 board was performed using this version of the function, taking 12 hours. Clearly, a more efficient approach is needed.

▶ Second Idea: Found in most open-source implementations

An improved approach is to only consider positions that cover a **corner cell**. This reduces the number of possible positions to check to $O(\text{CornerCells} \cdot \text{SquaresPerPiece})$. We perform this check for each of the $O(|\text{Pieces}|)$ pieces, with each check requiring $O(\text{SquaresPerPiece})$ time.

Complexity: $O(\text{CornerCells} \cdot |\text{Pieces}| \cdot \text{SquaresPerPiece}^2)$

▶ Third Idea: My Approach

By combining the idea of using the direction of the **corner cell** (from an intermediate approach I explored) with a divide-and-conquer strategy and precomputation, we arrive at a significantly faster and more efficient algorithm. *To the best of my knowledge, this method is novel and has not appeared in previous implementations.*

A natural starting point is to break down the problem of finding all possible actions into smaller subproblems: determining the possible actions that cover a specific **corner cell**. If this can be done efficiently, we can then combine the results to achieve a quick and effective solution.

> ***Problem.*** Given a **corner cell**, return the possible actions that cover it, assuming all the pieces are available.

Assuming all 21 shapes are available, out of the 40 cells surrounding the **corner cell** (within a distance of less than 5 in all directions), only the **legality** of 28 of these cells is relevant in determining the actions that can cover this cell. This is because 12 of the 40 cells cannot be covered by any piece that also covers the **corner cell**, according to the rules of the game (see Figure 3.1). This splitting of cells depend on the **diagonal direction** in which the **corner cell** is connected, as it is diagonally adjacent to a square of a piece on the board. See Figure 3.1 for a visual representation of the possible actions.

We can then use the idea of precomputing the possible placements (with respect to the **corner cell**'s coordinates and direction) for each bitmask (where 1 represents a **legal cell**nd 0 represents a non-legal cell) formed by these neighboring cells. While this is a promising approach, it would require approximately 6 GB of storage. We can adopt a mixed approach by reducing the number of neighboring cells considered. We can dividfe these 28 into two groups: 22 **core cells** and 6 **peripheral cells**. Then the possible placements will be defined as the union of the two independent sets: A_c^{core}, which is the set of actions that can be performed using only the 22 **core cells**, and $A_c^{\text{peripheral}}$, which is the set of actions that can be performed using at least one of the 6 **peripheral cells**.

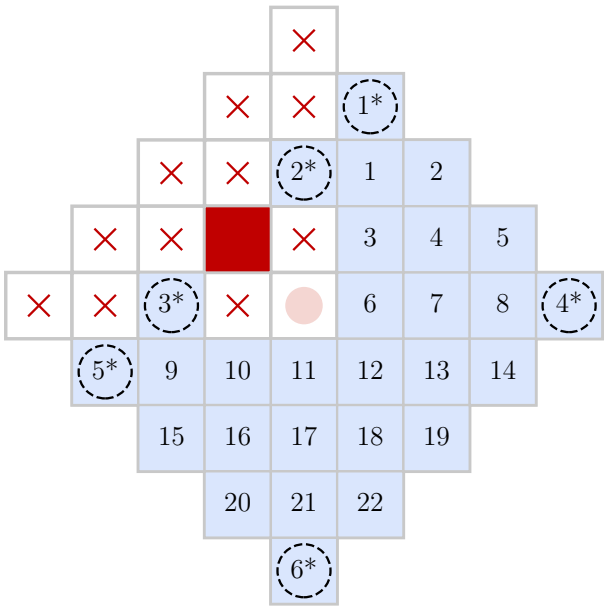


Figure 3.1: *The 28 cells surrounding the corner cell that are relevant for the computation of possible actions. The 22* ***core cells*** *are shown in blue, while the 6* ***peripheral cells*** *share the same color but are indicated by dashed outlines and labeled 1*, 2*, 3*, etc. The 12 cells that are not relevant for the computation are crossed out.*

Mathematically, this can be expressed as:

$$\text{Actions} = \bigcup_{c \in \text{CornerCells}} A_c, \quad A_c = A_c^{\text{core}} \uplus A_c^{\text{peripheral}}, \tag{3.1}$$

where Actions is the set of possible actions, assuming all pieces are available.

We observe that the actions involving the 6 peripheral cells will each cover exactly one of these cells. Furthermore, for each of these 6 cells, there is only one possible action that can cover both that cell and the corner cell. Consequently, the set $A_c^{\text{peripheral}}$ in the equation above will have a maximum size of 6, making it computationally efficient to determine dynamically.

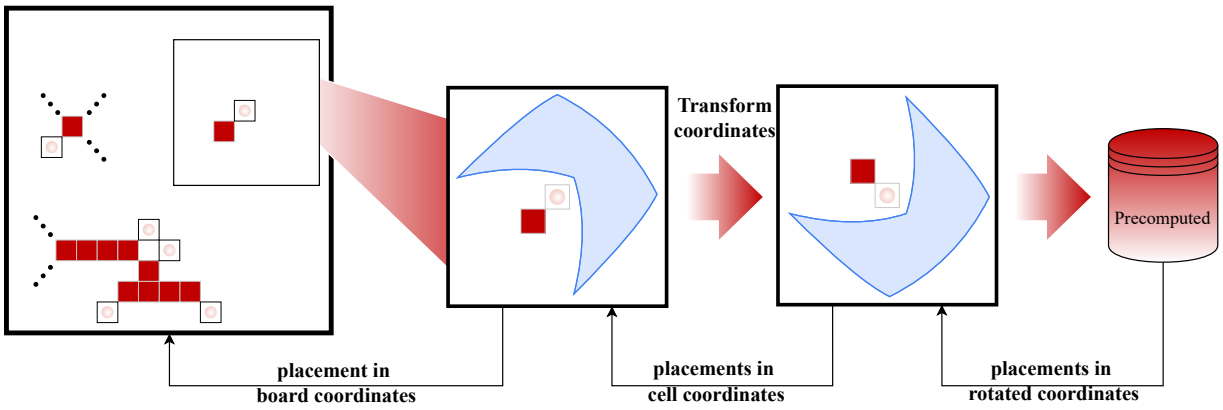


Figure 3.2: *Pipeline for computing possible actions. The blue shadow indicates the 28 cells that are relevant for the computation. The red-shadowed circle indicates the corner cell for the current player, while the dark-red colotred cell is the cell the corner cell needs to touch, defining the direction.*

For its complexity, each corner cell, we use the precomputed dictionary to return the set of

all the actions that contain the cell. This is a linear operation on the size of the output. Then, we just need to return the union of all the sets, while removing the actions use pieces that have already been used. Therefore the complexity would be $O(\text{CornerCells} + \sum \text{output sizes})$. We can see that $\sum \text{output sizes} < \text{SquaresPerPiece} \cdot |\text{output size}|$ since each action can cover at most SquaresPerPiece corner cells. Therefore, we can say that the complexity is $O(\text{CornerCells} + \text{SquaresPerPiece} \cdot |\text{output size}|)$.

In this section, we have seen three different ways to compute the possible actions for a given board state. In all these algorithms but the first one, we need a way to compute the **corner cells**. In Section 3.1.4 we will explore a way to do this efficiently.

3.1.4 Internal Interaction

To maintain an efficient set of **corner cells** for each player, we avoid recomputing it every time `possible_actions` is called. Instead, we update it incrementally.

This is achieved by maintaining a set of **corner cells** and **locked cells** for each player i, denoted as Corner_i and Locked_i respectively, and updating them at every move using a fast routine. The procedure `FastUpdateAttributes` efficiently updates the game state after a player makes a move: it removes the used piece from the player's available set, marks the newly **locked cells** (which are now occupied or adjacent to the piece), removes those from the corner set Corner_i, and adds any new **legal corner cell** created by the placed piece. Finally, it updates the board and ensures that no **corner cell** remains on a now-occupied cell for any player. See Algorithm 3 for the corresponding pseudocode.

3.1.5 Single-Agent Blokus Environment

The `SingleAgentBlokusEnv` class is a subclass of `BlokusEnv` that enables a single agent to play against built-in agents whose actions are hidden. When the agent takes an action with `step()`, the environment processes the move, then the internal agents take their turns until it is again the external agent's turn. The reward is defined as the difference between the area covered by the learning agent's action and the total area covered by the opponent's pieces since the last time the agent placed a piece. The `step()` method of the `SingleAgentBlokusEnv` class is shown in Listing 3.1. A comparison of the interactions in `BlokusEnv` and `SingleAgentBlokusEnv` is provided in Figure 3.3.

(a) *BlokusEnv-agents interaction* (b) *SingleAgentBlokusEnv-agent interaction*

Figure 3.3: *Comparison of high-level logic and interaction in multiple-agent (left) and single-agent (right) Blokus environments. This example assumes a 2-player game, but any number of players can be used.*

```python
    def step(self, action_id):
        obs = self.base_env.step(action_id)
        return self._process_hidden_agents(obs)

    def _process_hidden_agents(self, obs):
        while self.base_env.current_player != self.player_turn:
            agent = self.hidden_agents[self.base_env.current_player]
            action_id = agent.get_action(self, obs)
            obs = self.base_env.step(action_id)
        return obs
```

Listing 3.1: *The `step()` method of `SingleAgentBlokusEnv`. This method calls `BlokusEnv.step()` internally, then repeatedly queries the internal agents for their actions until it is the learning agent's turn again. For clarity, details about reward calculation and the `terminated`, `truncated`, and `info` parameters are omitted.*

From this point onward, we will use the term **environment** *to refer to either a* `BlokusEnv` *or* `SingleAgentBlokusEnv` *object, and* **state** *to refer to a state within such an environment. The specific context will make clear which environment or state is being referred to.*

The following sections—Section 3.2 and Section 3.3—focus on building agents that perform well in the 2-player version of Blokus across various board sizes. There are no official rules for playing the 2-player version on boards other than 20×20 (where each player controls two opposite colors, starts in opposite corners, and follows the standard rules) and 14×14 (known as Blokus Duo, where players start on opposite corners of the inner 6×6 section). For this version of the game, we establish the rule that players always start in opposite corners, regardless of board size.

Before describing the implementation of the various agents, we introduce the `Agent` class, which serves as a common interface for all agents. This class defines the `get_action(env: BlokusEnv, obs: ObsType) -> ActionType` method, which is called by the environment to obtain the agent's next move. Any **agent** must implement this method to interact properly with the environment.

3.2 Small Boards

In this part, we will discuss the implementation of our approach to solving the Blokus problem for small boards, beginning with Tree Search Algorithm and then moving on to tabular Q-learning.

3.2.1 MiniMax Agent

We take a score-based approach to the game, where the score is defined as the difference in area between our agent and the opponent, which could suggest a magnitude of victory or defeat. We apply the Minimax algorithm on those scores, and then the agent selects one of the optimal actions at random. This design choice to make the agent **stochastic** ensures that the agent does not always play the same action, preventing it from being exploited by an opponent.

Caching. A caching mechanism was implemented to store all optimal actions from previously evaluated states, allowing for quick retrieval during subsequent searches. After a search from state, the agent saved the list of optimal actions in the cache. If the cache reached its capacity, the entries with the greatest depth were prioritized for removal first. If multiple entries had the same depth, the least recently used (LRU) entries among them were removed to make space for new entries. The cache is saved to a file and reloaded (including board size and depth information) when the agent was recreated.

3.2.2 Alpha Beta Pruning Agent

The `ABPruningAgent` class extends the `MiniMaxAgent` class and implements the Negamax algorithm with Alpha-Beta pruning, as detailed in Section 2.2.2.

Caching. The caching here is done in a similar way as in the `MiniMaxAgent` class. Only the first found optimal action is stored in this way, this is done to take advantage of the speed-up of the search.

Search Heuristic. The order in which the actions are evaluated can significantly impact the performance of the search algorithm. A couple of heuristics were attempted to ensure the search prunes as much as possible. However, we would like to ensure that the sorting is also done quickly. For example, a heuristic that needs me to place each of the actions in the board to then decide on something of the new state is slow.

A way of specifying the heuristic could be passed to the `ABPruningAgent` class. A list of heuristics is shown in Table A.1.

Using **alpha-beta pruning**, we were able to deduce that there is a *winning strategy* for the **first player** on boards ranging from $\mathbf{5} \times \mathbf{5}$ up to $\mathbf{7} \times \mathbf{7}$.

3.2.3 Tabular Q-Learning Agent

Let us briefly review how Q-learning works, as discussed in Section 2.3.5. In tabular Q-learning, the function $Q(s, a)$ estimates the expected cumulative reward obtained by taking action a in state s and following the optimal policy thereafter. Formally, it satisfies the Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]$$

In this case, since `BlokusEnv` (described in Section 3.1) was designed for multi-agent simulation, it cannot be directly used to train a single agent. As discussed in Section 3.1.5, I implemented the `SingleAgentBlokusEnv` class, a subclass of `BlokusEnv`, which enables training a single agent against a built-in opponent. This environment is specifically designed to be compatible with the Q-learning algorithm.

Q-table. We store the Q-value $Q(s, a)$ for each state-action pair (s, a) in a dictionary. Due to the huge number of states, we only store the pairs encountered during training, rather than all possible pairs. For each first visited pair (s, a), we set the Q-values to 0.

Exploration Strategy. In each iteration, we select an action using the **epsilon-greedy strategy**, which balances *exploration* and *exploitation*. The agent chooses a random action with probability ϵ and the best action (the one with the highest Q-value) with probability $1 - \epsilon$. The design choice was for the ϵ value to linearly decay from 1.0 to 0.01 over the first 100,000 steps, promoting early exploration and later exploitation.

Q-learning Update Rule. In each iteration, the agent gets the (s_t, a_t, r_t, s_{t+1}) tuple from the environment. The agent then updates the Q-value for the state-action pair (s_t, a_t) using the reward r_t and the maximum Q-value for the next state s_{t+1}.

We then use the Q-learning update rule to update the Q-values based on the agent's experience. The update rule is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

where s_t is the current state, a_t is the action taken, r_t is the reward received, and γ is the discount factor. The learning rate α determines how much we update the Q-value based on the new information.

Training. During each episode, the agent interacts with the environment and updates its Q-values immediately using the Q-learning update rule. Training continues until the agent reaches a satisfactory performance level.

3.3 Big Boards

Larger boards result in an exponential growth of game states, making exhaustive search impractical. In these scenarios, while each state may theoretically have a single optimal move to avoid a loss against a perfect opponent, this is rarely true in practice. Often, there are multiple actions that can lead to *winning states*[2]. Additionally, since opponents are also imperfect, even if our agent makes a suboptimal choice, it is unlikely that the opponent will exploit it optimally. Our approach, therefore, focuses on developing agents that select "good enough" actions rather than perfect ones. In this section, we discuss the implementation of such agents that perform effectively, even if not perfectly, on large boards.

3.3.1 Q-Network Agent

The `QNetworkAgent` is defined as an agent that accepts a neural network architecture. Its core method, `get_action`, selects the action $a = \arg\max_a Q(s, a; \theta)$, where $Q(s, a; \theta)$ is the *learned estimate* of the action-value function, parameterized by the neural network weights θ, for a given state s and valid action a.

Inspired by previous work [12], the neural network was designed to take the state–action pair (s, a) as input and learn to approximate the *optimal action-value function* $Q^*(s, a)$, as defined in Section 2.3.4.

This design choice contrasts with the more **standard approach**, where the **neural network receives only the state s and outputs Q-values $Q(s, a)$ for all possible actions a in the**

[2] States where the agent wins if both the agent and the opponent play optimally.

action space, regardless of whether a is a valid action in state s. In our case, the action space is extremely large (for the 10×10 board, there are approximately $9,000$ possible actions in the game), making it impractical for the network to output Q-values for every action simultaneously. By providing the action as an input, the network can better generalize across action components such as piece shape and position.

To implement this modularly, we define a customizable interface extending `torch.Module`, called `QNetwork`. This interface is responsible for encoding the state–action pair (s, a) with the method `encode` into a unified representation and returning the corresponding Q-value.

I defined the `Trainer` interface, which is responsible for the training logic: for example, in DQN, it handles the collection of trajectories, the optimization step, the experience replay buffer, and hyperparameters—essentially everything except the main training loop. This design allows us to experiment with different architectures and training strategies without modifying the core training loop. The DQN algorithm described in this section—`DQNTrainer`—extends `Trainer`.

3.3.2 Neural Network Architecture

The forward pass begins with the current game state, encoded as a 44-channel tensor. This includes:

21 channels per player: Each corresponding to a specific shape, indicating whether it has been played (drawn on the board) by player 1 or player 2.

2 additional channels: Representing the current **corner cells** for each player, which indicate where new pieces can legally be placed.

This 44-channel state is concatenated with the proposed action, drawn as a single binary 10×10 channel (i.e., a one-hot mask of the selected shape at the chosen position), resulting in a 45-channel input.

This tensor is passed through a stack of **three ResNet blocks**, each with **two convolutional layers** (kernel size 3, padding 1). **No pooling layers**[3] are used, preserving the 10×10 spatial resolution throughout. **Batch normalization is omitted** due to the small batch size and non-stationary state distributions from replay buffer sampling.

The resulting tensor is flattened into a vector representing the high-level features extracted from the CNN layers.

In parallel, the discrete components of the action—the **shape ID**, **piece ID**, and **board position**—are each embedded into a 24-dimensional vector and summed to form a single action embedding. This embedding is then processed by a multilayer perceptron, producing a 256-dimensional output to enhance its representational capacity.

The network also incorporates a **mask of available shapes** for both players, represented using embeddings of the *precomputed* **shape IDs** and **player IDs**. This mechanism encodes which shapes remain available to the current player. The corresponding embeddings are masked, flattened, and concatenated with both the ResNet output and the action embedding.

Finally, this combined vector is then passed through a small fully connected network to predict the **Q-value** of the state–action pair.

A code snippet of the forward pass is shown in Listing A.1.

[3] Pooling layers are typically used to reduce the spatial dimensions of the input tensor, which can help decrease the number of parameters and computational requirements in the network. For example, pooling can reduce a tensor from $\mathbb{R}^{C \times H \times W}$ to $\mathbb{R}^{C \times \frac{H}{2} \times \frac{W}{2}}$.

In the following sections, we describe how the `QNetworkAgent` model is trained, beginning with our implementation of the DQN algorithm.

3.3.3 Deep Q-Network

Here we describe the implementation of the `DQNTrainer`, which trains the `QNetworkAgent` using the Deep Q-Network (DQN) algorithm [15], first introduced in Section 2.3.6.

For training, we use the single-player environment `SingleAgentBlokusEnv`, as was previously used for the tabular Q-learning agent.

▶ Experience Replay Buffer

The experience replay buffer is implemented as a fixed-size circular buffer that stores the agent's experiences as tuples of (s, a, r, s', d), where s is the current state, a is the action taken, r is the reward received, s' is the next state, and d is a boolean indicating whether the episode has ended. When the buffer reaches its maximum capacity, the oldest transition is replaced by the newest one. During training, transitions are sampled uniformly at random[4] from the buffer in batches of size N (the **batch size**), which is a hyperparameter specifying how many transitions are used for each update step.

▶ Target Network

The target network was implemented as a separate neural network—a fixed, non-trainable copy of the current network—used to compute target Q-values without propagating gradients through it. The target value for the Q-learning update is computed as follows:

$$y_i = r_i + \gamma \max_{a'} Q(s'_i, a'; \theta^-)(1 - d_i)$$

where y_i is the target Q-value for the i-th transition, r_i is the reward, γ is the discount factor, $Q(s'_i, a'; \theta^-)$ is the Q-value predicted by the target network for the next state s'_i and action a', and d_i is a boolean indicating whether the episode has ended.

As in the original paper[14], the target network was updated every C steps, where C is a hyperparameter. We can see how this is done in Pytorch in Listing 3.2.

```
1 if step_count % target_update_freq == 0:
2     target_net.load_state_dict(agent.policy_net.state_dict())
```

Listing 3.2: *Target network update.*

▶ Exploration Strategy

We used the same **epsilon-greedy strategy** described for Q-Learning in Section 3.2.3, where the agent selects a random action with probability ϵ and chooses the action with the highest Q-value with probability $1 - \epsilon$. The value of ϵ decays over time to encourage exploration in the early stages of training and exploitation in later stages.

[4] A prioritized approach could also be used, for example, to prioritize transitions with larger *temporal-difference errors* (i.e., the difference between the estimated and target values).

▶ **Training Details**

Loss Function. The loss function used for training the neural network was the mean squared error (MSE) between the predicted Q-values and the target Q-values. The MSE loss is defined as:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i; \theta))^2$$

where N is the batch size, y_i is the target Q-value for the i-th transition, and $Q(s_i, a_i; \theta)$ is the predicted Q-value for the i-th transition.

Optimizer. The optimizer used for training the neural network was the Adam optimizer [7], which is an adaptive learning rate optimization algorithm. The Adam optimizer is defined as:

$$\theta_{t+1} = \theta_t - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}$$

where θ_t is the parameter vector at time step t, α is the learning rate, m_t is the first moment estimate, v_t is the second moment estimate, and ϵ is a small constant to prevent division by zero. The terms m_t and v_t help stabilize the learning process by adapting the learning rate for each parameter based on the first and second moments of the gradients[5]. This per-parameter adaptation allows the optimizer to converge more quickly and reliably.

▶ **Reward Shaping**

We mostly used a simple reward function: the reward is defined as the difference between the size of the piece placed by the agent and the size of the piece placed by the opponent. For a more detailed discussion of alternative reward shaping strategies, see Section A.5.

While the DQN approach described above proved effective—enabling the agent to consistently achieve a win rate of at least 80% or higher against a fixed opponent after sufficient training—this setup is inherently limited to scenarios where the opponent's behavior is known and unchanging. However, our broader objective is to develop an agent that can generalize and compete effectively against **any** opponent, regardless of their strategy. In the following section, we address this challenge and present a more general approach.

3.3.4 A Different Approach

As seen in picture Figure 3.3b, the standard approach to training an agent is to have a single agent that plays in the environment formed by the core environment and the opponent. The agent then treats that as a new environment and learns as usual.

Although we've seen that this approach works well, it has some limitations. The agent learns to play against a fixed opponent, and the training process is not guaranteed to generalize to other opponents.

[5] The first and second moment estimates are computed as exponentially weighted moving averages of the gradients: $m_t = (1-\beta_1)\sum_{j=1}^{t}\beta_1^{t-j}g_j$, $v_t = (1-\beta_2)\sum_{j=1}^{t}\beta_2^{t-j}g_j^2$, where g_j is the gradient at step j, and β_1, β_2 are hyperparameters controlling the decay rates.

This is a reasonable goal, since we typically train an agent to maximize cumulative reward, which is usually defined with respect to the opponent's policy. However, in our case, we do not merely want to maximize the expected score difference—we do not want an agent that can achieve an expected score of 10 but still has a chance to lose due to the opponent's stochasticity. Instead, we want to ensure that, regardless of the opponent's actions, our agent can always secure a positive score—even if it only wins by a single point each time.

Motivated by the principles of negamax and DQN, we propose a novel approach that combines the strengths of both methods. Instead of having a NN that estimates the $Q(s, a)$ in `SingleAgentBlokusEnv`, it will learn the negamax value of s' in the original `BlokusEnv`, where s' is the state after taking action a in state s, i.e., $Q(s, a)$ will be the exact value I can guarantee if I and the opponent play optimal actions.

In the ideal world, we would like to have an agent that can beat any opponent always. So we would like the oppponent agent to perform all actions as much as possible, so the idea of making it Random comes to mind, however this is not the best idea, as the estimates won't be good. We use the a second neural network to estimate these optimal negamax values, while allowing the agent to take random actions.

Instead we have two neural networks Q_{agent} and $Q_{opponent}$, which represent the Q-values for the agent and the opponent, respectively, seen from their perspective.

The new Bellman equations are as follows:

$$Q^*_{agent}(s_i, a_i) = r_i - \gamma \cdot \max_{a'} Q^*_{opponent}(s'_i, a')$$
$$Q^*_{opponent}(s_i, a_i) = r_i - \gamma \cdot \max_{a'} Q^*_{agent}(s'_i, a')$$

where s_i is the state at time step i, a_i is the action taken at time step i, r_i is the reward received at time step i, and γ is the discount factor. The state s'_i is the state after taking action a_i in state s_i.

Thus, we can define a unified Q-function $Q(s, a)$, where $Q(s, a) = Q_{agent}(s, a)$ if it is the agent's turn, and $Q(s, a) = Q_{opponent}(s, a)$ if it is the opponent's turn. This allows us to use the same training procedure as described in Section 3.3.3.

$$y_i = \begin{cases} r_i, & \text{if the episode terminates at step } i \\ r_i - \gamma \cdot \max_{a'} Q(s'_i, a'; \theta^-), & \text{otherwise} \end{cases}$$

▶ Experience Replay Buffer

We will an experience replay buffer that willl be shared by both agent and oppponent's models. Similar to the DQN implemented in Section 3.3.3, everything about the replay we will be kept the same.

▶ Target Network

Unlike the DQN implementation described in Section 3.3.3, here we maintain two separate target networks, one for each player. Each target network is updated every C steps, where C is a hyperparameter.

▸ Exploration Strategy

Learning Agent. For our learning agent, we continue to use the ϵ-greedy strategy as in the DQN approach. Other exploration strategies, such as the top-3 method described in the DQN section, are also valid and may yield good results.

Controlled Opponent Policy. As mentioned before, we will allow our opponent to try lots of actions, being almost always random in order for the learning agent to explore the state space. We found that a purely random opponent was insufficient for meaningful exploration, so we made the opponent slightly more greedy by using a fixed ϵ-greedy policy. This encourages exploration of more relevant parts of the state space while still allowing the agent to encounter diverse scenarios. This stands in contrast to traditional self-play, where both agents co-adapt simultaneously, often leading to instability.

▸ Training Details

We keep the same `Trainer`–`QNetwork`–`NetworkAgent` structure as in Section 3.3.3. The only difference is that we have two Q-networks, one for each player. The training process is similar to the one described in Section 3.3.3, with the following differences:

Loss Function. If $|Q(s,a) - (r - \max_{a'} Q(s',a'))| < \tau$, and $\tau \cdot d < 0.5$, where d is the maximum depth of the search tree, then this estimate is sufficiently accurate and performs comparably to the optimal one. For a proof, see Section A.6.

This implies that if the Q-value estimates are close enough to the target, small errors are acceptable. To robustly handle both small and large errors, we use the Huber loss between the predicted $\hat{y} = Q(s_i, a_i; \theta)$ and the target $y = r_i - \gamma \cdot \max_{a'} Q(s_i', a'; \theta^-)$. The Huber loss is defined as:

$$\text{Huber}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & \text{if } |y - \hat{y}| < 1 \\ |y - \hat{y}| - \frac{1}{2}, & \text{otherwise} \end{cases}$$

Optimizer. We use Adam [7] as the optimizer for both policy networks, updating the parameters of both the learning agent and the opponent jointly.

```
self.optimizer = optim.Adam(
    chain(self.agent.policy_net.parameters(), self.opponent_agent.policy_net.parameters()),
    ↪ lr=lr
)
```

Listing 3.3: *Joint Adam optimizer for both agents*

In this chapter, we presented the development of the Blokus game environment and the agents designed to interact with it. In Section 3.1, we detailed the implementation of the `BlokusEnv` class, a subclass of `GymEnv`, which supports 2–4 players, as well as the `SingleAgentBlokusEnv` class, which enables a single agent to play against built-in agents.

We introduced the necessary terminology for understanding the `BlokusEnv` implementation, briefly described the `reset()` and `step()` functions, and outlined an efficient method for finding possible actions in the game. The following sections described several implemented agents, with further details provided in the next chapter. In Section 3.2, we discussed our approach to solving the Blokus problem for boards up to 7×7, covering the stochastic `MinimaxAgent`, the deterministic `ABPruningAgent`, and the `TabularQLearningAgent`. In Section 3.3, we explained our approach for larger boards, focusing on the 10×10 case, which can be extended to any board size. We also described the `QNetworkAgent`, which is trained using two different methods: `DQN` and `MyApproach`.

In the next chapter, we will evaluate the efficiency of our approach for finding possible actions, as well as assess the agents and compare them against the core success criteria.

4

Evaluation

In this chapter, we will evaluate the performance of the environment we have built, `BlokusEnv`, and of the trainable agents: `TabularQLearningAgent` and `QNetworkAgent`.

Since there are not any well-known publicly available strong Blokus agents online, we rely on evaluating against heuristic agents.

4.1 Core Evaluation

Remembering the success criteria from Appendix B, we will evaluate the environment and the agents based on the following criteria:

- **Build** and **set up** the simulation environment using OpenAI Gymnasium.
- **Implement** and **test** the A1 agent: Tabular Q-learning against a `RandomAgent`.
- **Implement** and **test** the A2 agent: Tabular Q-learning against a minimax-optimised opponent.
- **Achieve a 75% win rate** against a `RandomAgent` on 7×7 boards using tabular Q-learning.
- **Achieve a 75% win rate** against a minimax-optimised opponent on 7×7 boards using tabular Q-learning.

4.2 Environment Evaluation

4.2.1 Environment Correctness

To heuristically verify correctness, I first ensured that the piece placement logic was correct using unit tests. For each piece passed to `step()`, I checked whether the piece could legally be placed. If an invalid action was detected, an exception was raised, allowing us to identify and resolve the underlying issue. The environment has been stable and free of bugs for the past five months, indicating a robust and correct implementation. Additionally, unit tests were written to verify the correctness of the environment in basic scenarios.

4.2.2 Speed of Possible Actions Computation

Here, we compare our method for computing possible actions with the other two approaches described in Section 3.1.3.

Our approach requires 102 seconds to precompute the possible actions for each of the 2^{22} possible bitmasks and save them to a file. This precomputation is performed only the very first time the project is run. Afterward, loading the precomputed actions into RAM takes just 3 seconds. When multiple environments are initialized and played together in the same session, only a single copy of the precomputed actions is needed, so this loading step is only performed once per session. Let us now evaluate how worthy this method actually is.

We take $10,000$ steps in the environment, and for each method, we will measure the time it takes in total. To make this fair, we set the `seed` to a fix value (so that each method is

applied to the same set of states always) for each of these methods. We will keep the fast interaction as described in Section 3.1.4 for all the methods. See Table 4.1 for the results. We ran $10,000$ steps, but for ease of comparison, the values in the table are scaled by a factor of 10, representing the equivalent of $100,000$ steps. Our method is approximately $10\times$ faster than the second method on boards ranging from 4×4 to 10×10, and about $6\times$ faster on the 20×20 board.

Method	4×4	7×7	10×10	20×20
inefficient	8.12	36.15	76.09	261.23
efficient	5.43	13.37	22.20	48.71
precomputed	**0.57**	**1.38**	**2.62**	**8.93**

Table 4.1: *Evaluation of different methods for computing possible actions in the Blokus environment. It was run for* $10,000$ *steps, but the values in the table are scaled by a factor of* 10*, representing the equivalent of* $100,000$ *steps.*

4.3 Agent Evaluation

We mostly focus on win-draw-loss rates, as well as score distribution.

4.3.1 Tabular Q-Learning

We now evaluate the performance of the `TabularQLearningAgent` using the Q-Learning algorithm, first introduced in Section 2.3.5 and implemented in Section 3.2.3. The agent was trained in the `SingleAgentBlokusEnv` environment, embedding either a random or a minimax opponent. After training, its performance was evaluated against both opponents.

All experiments are conducted on a 7×7 board, where the state-space complexity is approximately $8 \cdot 10^{14}$.

▶ Against Random Opponent

In just 20 minutes of training and $500,000$ iterations, the agent achieves a win rate of 90% against a `RandomAgent`, and $634,010$ states saved in the Q-table. This was run in my personal computer.

Given that the `RandomAgent` may take actions leading to states not encountered during training, and considering that $500,000$ is much less than $\sqrt{8 \cdot 10^{14}}$, achieving a 90% win rate is a strong result. However, a notable limitation is that if the agent encounters a state it has never seen before, its behavior becomes effectively random, which can lead to unpredictable outcomes unless the agent is already in a highly advantageous position.

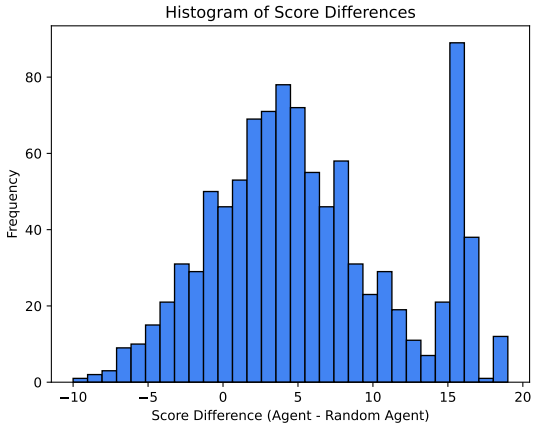
See [figure] for the distribution of scores. For further context, we also compare the performance of other heuristic agents against the `RandomAgent`. See Figure 4.1a for these results.

The results for the Tabular Q-Learning agent against the RandomAgent are: 78% wins, 17% draws, and 5% losses in $1,000$ games.

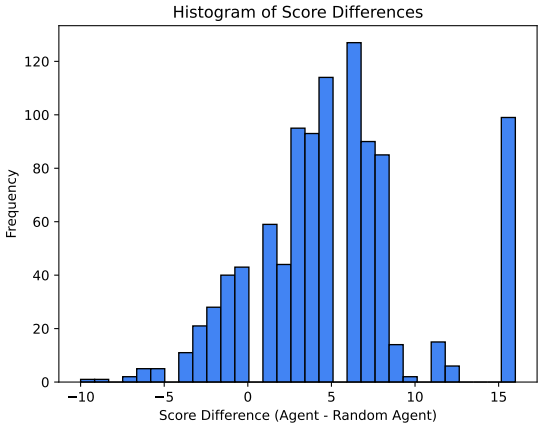
▸ Against Minimax-Optimized Opponent

To evaluate performance against a minimax-optimized opponent, we require an agent that uses the minimax algorithm. The `MinimaxAgent` itself is slow: at depth 3, it takes approximately 1.5 s per move, and at depth 4, around 2.5 minutes per move. Even with caching, this approach is not practical, as the required cache size grows exponentially. To address this, we employ the `AlphaBetaPruningAgent`, which optimizes the search process. However, this agent is deterministic, which poses challenges when used as the second player. If the `TabularQLearningAgent` is trained against this deterministic alpha-beta pruning agent, it quickly learns to defeat it 100% of the time within just 1–3 minutes of training.

A more robust and interesting approach is to use an agent that follows an epsilon-greedy strategy: with probability ϵ, it selects a random action, and with probability $1 - \epsilon$, it selects the best action, given by the alpha-beta pruning agent with maximum depth. We tried $\epsilon = 0.1$ and trained the `TabularQLearningAgent` against this agent. To do this, it was really hard, as it takes a lot of time to compute the best action for the alpha-beta pruning agent. A small CPU server provided by my supervisor was used to run this experiment. During the process, caching was enabled, allowing us to store actions for $10,000,000$ of the states encountered by the opponent. However, whenever the agent encounters a state whose value is not cached, it takes significantly longer to compute the action. The results are shown in Figure 4.1b. The results for the Tabular Q-Learning agent against the minimax-optimized opponent are: 84% wins, 11% draws, and 5% losses in $1,000$ games.



(**a**) *Performance of QLearning against* `RandomAgent` *after being trained against it on a* 7×7 *board.*



(**b**) *Performance of QLearning against a minimax-optimized agent, after being trained against it on a* 7×7 *board.*

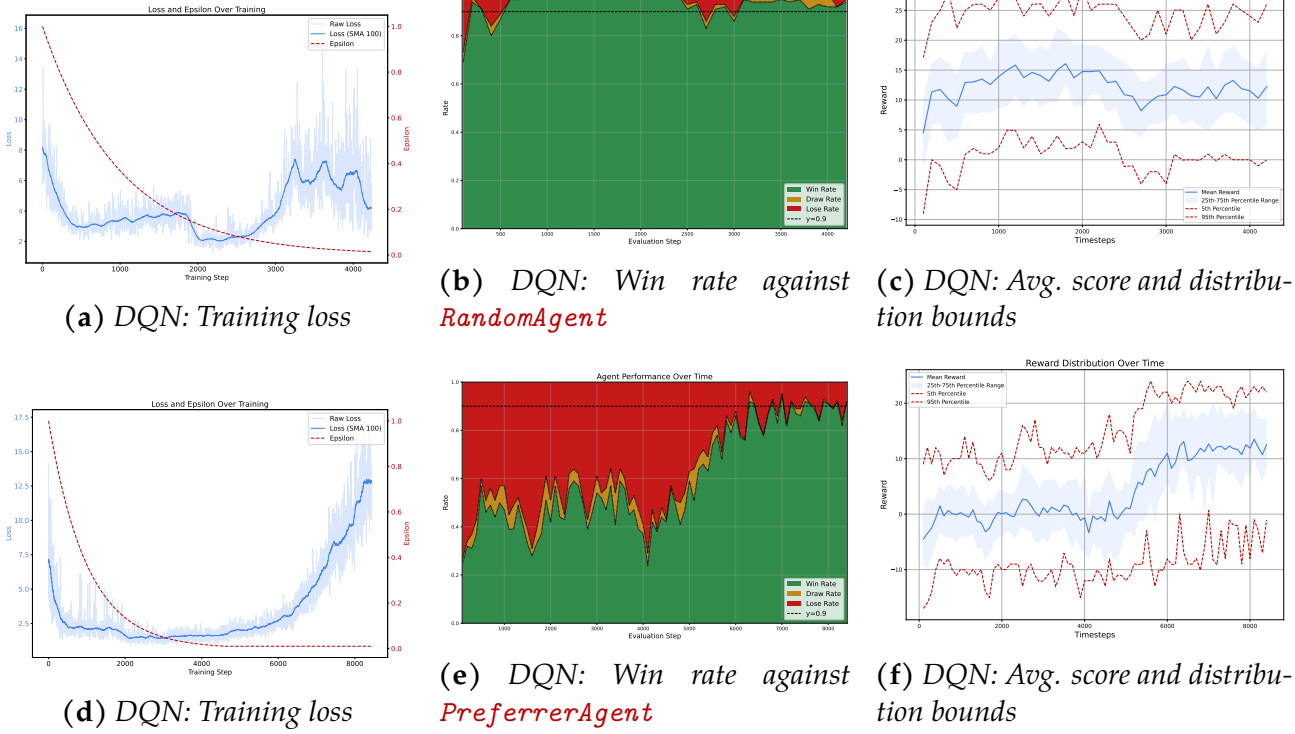
4.3.2 DQN algorithm

For the 10×10 board, the replay buffer was set to a size of $50,000$ transitions, which was sufficient to store a significant amount of experience. The buffer was implemented as a circular buffer.

Batch size used 128, learning rate of $1e^{-6}$, and the discount factor was set to 0.99. The epsilon decay was set to 0.999, with a minimum value of 0.01. We trained the `QNetworkAgent` against two types of opponents: the `RandomAgent` and the `PreferrerAgent`. The latter is a heuristic agent that selects pieces in a predefined order, which we believe to be effective.

While not optimal, this agent demonstrates strong performance and maintains fast decision-making times comparable to the `RandomAgent`. For reference, the `PreferrerAgent` achieves approximately a 95% win rate against the `RandomAgent` when playing first, and about 88% when playing second. See the accompanying figures for detailed performance results. I have used NVIDIA A100-SXM4-40GB in Google Colab for training the DQN agent.

We test how it performs in the embedded environment while training.



(**a**) *DQN: Training loss*

(**b**) *DQN: Win rate against* `RandomAgent`

(**c**) *DQN: Avg. score and distribution bounds*

(**d**) *DQN: Training loss*

(**e**) *DQN: Win rate against* `PreferrerAgent`

(**f**) *DQN: Avg. score and distribution bounds*

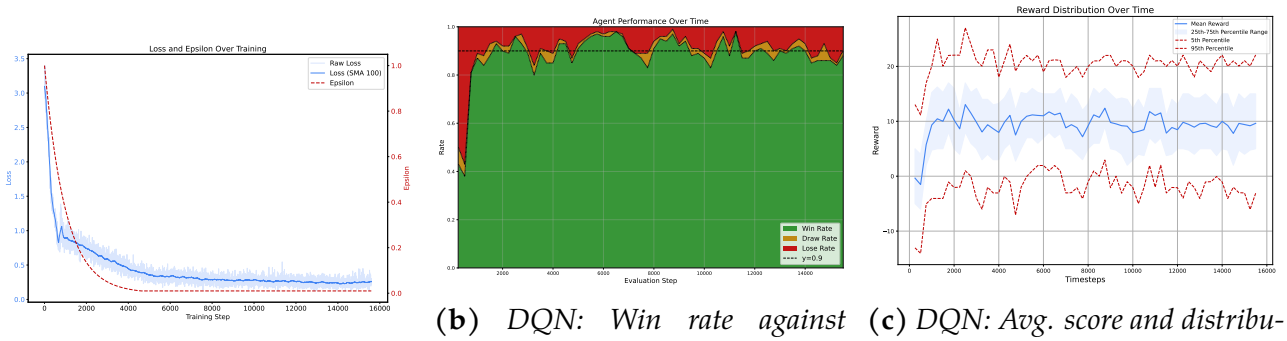
Figure 4.2: *Training metrics for both agents: (top row) DQN agent when being trained against a* `RandomAgent`*, (bottom row) DQN agent when trained against a* `PreferrerAgent`*. Columns show (a) loss, (b) win rate, and (c) average score and distribution bounds.*

It is evident that training against the `PreferrerAgent` provides a more meaningful challenge than training against the `RandomAgent`. The DQN agent learns more efficiently and achieves a higher win rate, as the `PreferrerAgent` is a significantly stronger opponent.

4.3.3 My Approach

Here, we test how the agent performs against the `RandomAgent` after it has been trained.

We can see that it wins by a large margin against the `RandomAgent`, even though it did not encounter this opponent during training. See Figure ?? for the results. While there were instances where the agent achieved promising results—such as winning approximately 80% of games against a greedy agent and 60% against a minimax agent at depth 3—I was not able to consistently reproduce this level of performance against more advanced agents.



(**a**) *DQN: Training loss*

(**b**) *DQN: Win rate against* `RandomAgent`

(**c**) *DQN: Avg. score and distribution bounds*

Figure 4.3: *Training metrics* `QNetworkAgent` *when trained against a* `RandomAgent` *using the auxiliary algorithm I proposed. Columns show* (a) *loss,* (b) *win rate, and* (c) *average score and distribution bounds.*

4.4 Experimentation

Over 15 hyperparameters needed to be tuned, and the training time is quite long, so no brute force hyperparameter search was done. Manual tuning was used. Checking the loss, win rate, osciallation of the values. Also a little UI was coded to play against it...the built-in `RecordEpisodeStatistics` wrapper is used. Also `RecordVideo` is used to evaluate manually the performance of the agent in real time. All this was done to better tune the hyperparameters.

For the so many hyperparametes: batch size, learning rate, epsilon (how to do the epsilon scheduler), discount factor, optimizer, loss function, architecture, sampling, exploration strategy,

The Minimax, AlphaBetaPruningAgent and the `TabularQLearningAgent` didn't need much experimentation as they only work for up to 7×7 boards, and a solution with the `AlphaBetaPruningAgent` has been found, and cached in a file.

Here we focus on the `QNetworkAgent` and the two training algorithms I've implemented: the `DQN` and the `MyAlgorithm`.

Loss function. All these were tried: Huber loss (with $\delta = 0.5$), L1 loss, and L2 loss, with and without clipping (e.g., clipping gradients at 10). Ultimately, Huber loss was preferred for its stability.

Optimizers A couple of optimizers were attempted: Adam, RMSprop, and SGD. Adam was ultimately chosen for its stability and faster convergence in this setting.

Exploration Instead of simple DQN, we take the best action and random otherwise, take the first 3 actions, instead of just one... (this smooths the thing in our case), otherwise if it's just one, if for some reason, the thing moves and the estimate says to use a different action, then it is as if it had overfitted the other and needs to relearn the other path that hasn't been covered as much!!! This was done since the estimations were not very good.

Evaluation: Focusing only on the loss does not always correlate with improving agent's performance in the environment. If the loss decrease, and the agent isn't accumulating higher rewards, it could mean that the model is learning to predict better, but not taking actions that lead to greater success.

5 Conclusions

This project has been a total success. We implemented an efficient Blokus environment and several agents that can plat the game very well.

5.1 Work completed

As expressed in Section 1.5, the project implemented a Blokus environment compatible with the Gymnasium API and developed several agents capable of playing the game, including the `MinimaxAgent`, `ABPruningAgent`, and `TabularQLearningAgent` for small boards, as well as the `QNetworkAgent` for larger boards. The project also included a custom approach to training the agent, which was successfully implemented and tested.

5.1.1 Novel Contributions

Possible Actions Precomputation. I introduced a method for efficiently determining possible actions, as described in Section 3.1.3, by precomputing. To the best of my knowledge, this approach has not been previously explored.

Dual Q-value Estimation. I introduced a technique for estimating the Q value of a state-action pair, assuming both players play optimally, and optimized it using a unified loss function, as described in Section 3.3.4.

5.2 Limitations

While the implemented architecture is relatively shallow and training times are reasonable, the resulting performance is moderate—neither outstanding nor poor. The proposed method does not always converge reliably; although the loss generally decreases, indicating that average Q-values approach optimality, the agent does not consistently select the best actions.

5.3 Future Work

This project has opened the doors to a wide range of future investigations. The following list outlines some potential areas for further exploration:

- Optimize the backend of the environment to C++ to speed up the training process.
- Combine the proposed method with stabilization techniques such as Double DQN [5] or Dueling DQN [21].
- Incorporate an inductive bias for symmetry, ensuring that $Q(s, a) = Q(s', a')$ whenever s' and a' are the reflections of s and a with respect to the main diagonal of the board. This approach was previously attempted, but did not give good results.

- ◇ Use Proximal Policy Optimization (PPO) [17] to train the agent. This method was tried in earlier stages but did not yield good results, possibly due to the lack of a good encoding.
- ◇ Explore transfer learning approaches. For example, solve the problem for an $N \times N$ board and reuse the learned embeddings of shapes, pieces, and players to accelerate learning on larger boards.
- ◇ In the proposed method:
  - Prioritized sampling could be beneficial, as the loss is generally small, but occasional outliers may hinder convergence.
  - Regularizing the replay buffer to maintain a balanced number of experiences for each agent could improve training stability.
- ◇ A more rigorous hyperparameter search could be conducted.
- ◇ Design the network to take s and output the Q-value for each possible action could offer some advantages.
- ◇ Explore the 4-player version of the game.

5.4 Lessons Learned

Hyperparameter tuning. I learned that hyperparameter tuning is a challenging and time-intensive process, involving the careful exploration of a large search space (with over 15 parameters to consider). Identifying effective combinations required substantial experimentation and iterative refinement.

Model management. I learned that managing the model files is a complex task, as it involves versioning, organizing checkpoints, and ensuring consistency between training and evaluation settings.

Do not underestimate. I underestimated the complexity of the 7×7 board, which delayed progress for nearly two weeks as I struggled to develop a fast enough minimax agent for training the Q-learning agent. Starting with a smaller board and scaling up would have been more efficient. I also underestimated the time required for polishing the dissertation and evaluation. Although much of the dissertation was written by February, I only began refining it in April, and improving my writing skills took longer than expected.

Bibliography

[1] B. W. Boehm. "A spiral model of software development and enhancement". In: *Computer* 21.5 (1988), pp. 61–72. DOI: 10.1109/2.59 (cit. on p. 16).

[2] Peter Boström and Anna Maria Modée. "Changing the Random Behavior of a Q-Learning Agent over Time". Bachelor's Thesis in Computer Science (15 ECTS credits), Supervisor: Johan Boye, Examiner: Mads Dam. Stockholm, Sweden: Royal Institute of Technology (KTH), 2011. URL: https://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/rapport/bostrom_peter_OCH_modee_anna_maria_K11041.pdf (cit. on p. 3).

[3] Derek Gloudemans. *Blokus Reinforcement Learning*. A Blokus game interface implementing heuristics, search strategies, and learning agents. Deep reinforcement learning state estimation planned for future updates. 2018. URL: https://github.com/DerekGloudemans/Blokus-Reinforcement-Learning (cit. on p. 3).

[4] Niko Hass. *rust-socha2021*. Accessed: 2025-05-09. 2021. URL: https://github.com/nikohass/rust-socha2021 (cit. on p. 3).

[5] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fcPaper.pdf (cit. on p. 38).

[6] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 14).

[7] Diederik P Kingma. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on pp. 29, 31).

[8] Matt Knapper. *Machine Learning Blokus*. Accessed: 2025-05-09. 2014. URL: https://github.com/mknapper1/Machine-Learning-Blokus (cit. on p. 3).

[9] Donald E. Knuth. "Estimating the Efficiency of Backtrack Programs". In: *Mathematics of Computation* 29.129 (1975), pp. 121–136. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2005469 (visited on 05/08/2025) (cit. on p. 2).

[10] Jakub Kubiak. *Blokus-RL: Reinforcement Learning Environment for Blokus*. Accessed: 2025-05-09. 2021. URL: https://github.com/KubiakJakub01/Blokus-RL/ (cit. on p. 3).

[11] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541 (cit. on p. 13).

[12] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015) (cit. on p. 26).

[13] Sharon Mathew. "Develop an AI Algorithm to Play the Board Game Blokus". Submitted in accordance with the requirements for the degree of MSc in Mobile Computing and Communication Networks. MSc Thesis. Leeds, United Kingdom: University of Leeds, 2017. URL: https://teaching.bb-ai.net/Student-Projects/project-reports/Games/Sharon-Mathew-Blokus-project-report.pdf (cit. on p. 3).

[14] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (cit. on pp. 13, 28).

[15] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2013). URL: https://arxiv.org/abs/1312.5602 (cit. on pp. 13, 28).

[16] Sofia Maria Nikolakaki. *Algorithm Modeling for Hardware Implementation of a Blokus Duo Game*. Accessed: 2025-05-09. Chania, Greece, 2014. URL: https://artemis.library.tuc.gr/DT2014-0060/DT2014-0060.pdf (cit. on p. 3).

[17] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347 (cit. on p. 39).

[18] Claude E. Shannon. "Programming a Computer for Playing Chess". In: *Philosophical Magazine*. Series 7 41.314 (Mar. 1950), pp. 256–275. DOI: 10.1080/14786445008521796. URL: http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf (cit. on p. 3).

[19] David Silver et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017) (cit. on p. 3).

[20] João Vieira. "Playing BlokusDuo in a ZYNQ Device: A Quest for an Efficient Algorithm". In: *XVI Jornadas sobre Sistemas Reconfiguráveis (REC'2020)*. Accessed: 2025-05-09. Lisbon, Portugal: Instituto Superior Técnico, University of Lisbon, 2020. URL: https://web.tecnico.ulisboa.pt/~joaomiguelvieira/public/docs/papers/playing_blokus_duo_in_a_zynq_device_a_quest_for_an_efficient_algorithm.pdf (cit. on p. 3).

[21] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. arXiv: 1511.06581 [cs.LG]. URL: https://arxiv.org/abs/1511.06581 (cit. on p. 38).

[22] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College, Cambridge, May 1989. URL: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf (cit. on p. 12).

A

Appendix A

A.1 Deep Q-Learning pseudocode

Algorithm 2 Deep Q-learning with Experience Replay and Target Networks

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- \leftarrow \theta$
for episode = 1 to M **do**
    for $t = 1$ to T **do**
        With probability ϵ select a random action a_t
        Otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t in emulator and observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
        Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{for non-terminal } s_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ
        Increment num_steps by 1
        if num_steps mod C == 0 **then** ▷ Every C steps
            Reset target network: $\theta^- \leftarrow \theta$
        end if
    end for
end for

A.2 Possible Actions in Circular Layout

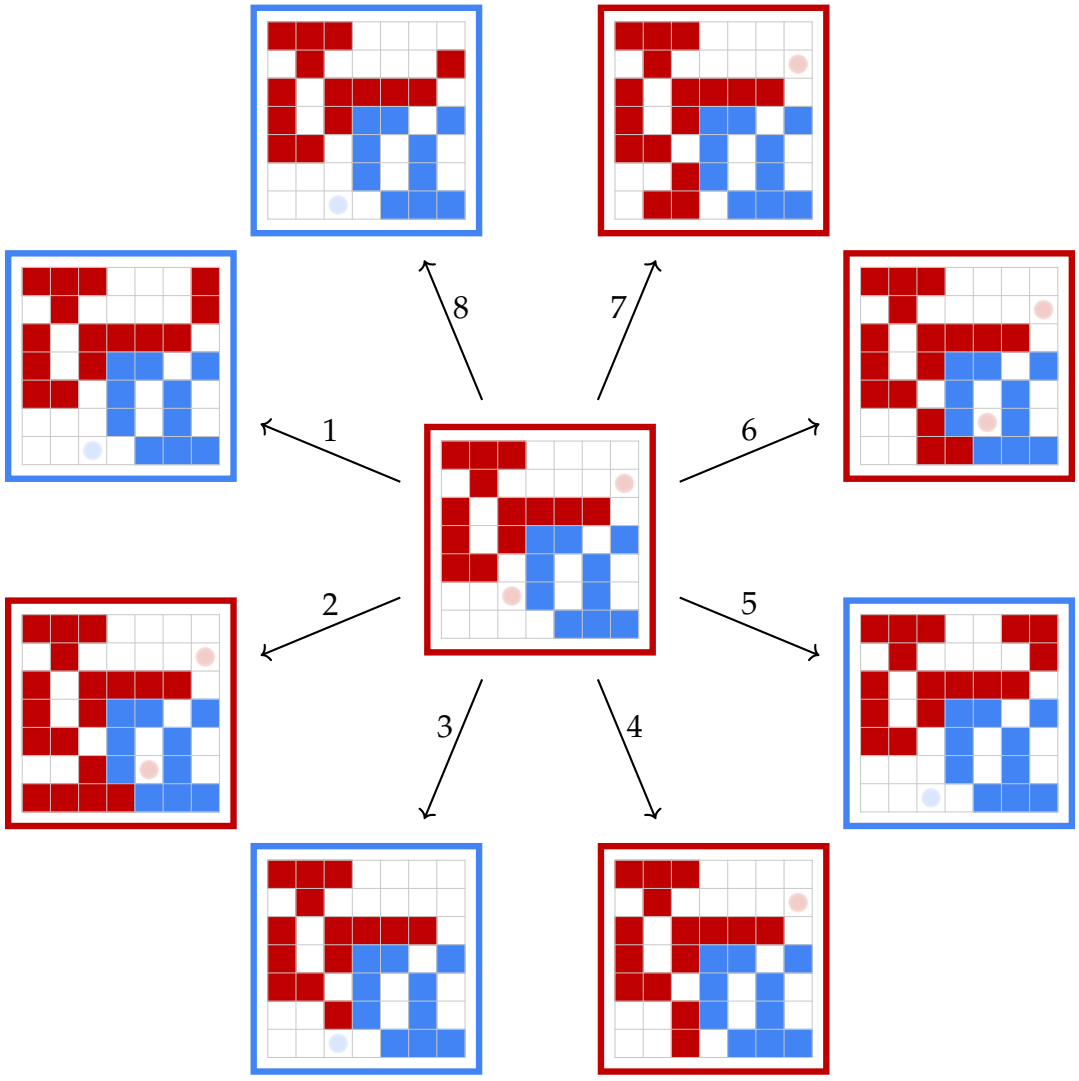


Figure A.1: *Possible Actions in a Circular Layout. The central board represents the current state, while the surrounding boards illustrate the possible next states after each of the 8 potential actions from the current state. The frame around each board indicates the color of the player whose turn it is. The circular-shaded cells highlight the corner cells for the player whose turn it is to play.*

A.3 Fast Update Attributes for Blokus

Algorithm 3 Fast Update Attributes for Blokus

procedure FastUpdateAttributes(player = p, action = a) ▷ Remove placed piece from available pieces
  p.available_shapes $\leftarrow$ p.available_shapes $\setminus$ $\{a$.shape_id$\}$
  ▷ Update Locked_p and remove from Corner_p
  for each $(x, y) \in a$.locked **do**
    $\text{Locked}_p \leftarrow \text{Locked}_p \cup \{(x, y)\}$
    if $(x, y) \in \text{Corner}_p$ **then**
      $\text{Corner}_p \leftarrow \text{Corner}_p \setminus \{(x, y)\}$
    end if
  end for
  ▷ Add expanders Corner_p from piece expanders
  for each $((x, y), (dx, dy)) \in a$.expanders **do**
    if legal_cell$((x, y), p)$ **then**
      $\text{Corner}_p[(x, y)] \leftarrow (dx, dy)$
    end if
  end for
  ▷ Update board and remove occupied expander squares
  for each $(x, y) \in a$.body **do**
    board$[x, y] \leftarrow p$
    if $(x, y) \in \text{Corner}_p$ **then**
      $\text{Corner}_p \leftarrow \text{Corner}_p \setminus \{(x, y)\}$
    end if
    for $p' \neq p \in P$ **do**
      if $(x, y) \in \text{Corner}_{p'}$ **then**
        $\text{Corner}_{p'} \leftarrow \text{Corner}_{p'} \setminus \{(x, y)\}$
      end if
    end for
  end for
end procedure

A.4 Heuristic Functions

A.5 Reward Shaping

One approach is to estimate the quality of a state using a function $h_p(s)$, which evaluates the value of a given state for player p. This function can be used to shape the reward signal,

#	Description	Requirement
1	Piece size.	None
2	Maximise my **corner cells**.	Requires placing the piece to evaluate
3	Minimise opponent's **corner cells**.	Requires placing the piece to evaluate
4	Maximise difference of **corner cells**.	Requires placing the piece to evaluate
5	Minimise number of possible actions for opponent.	Requires placing the piece to evaluate
6	Rank the shapes by predefined order (e.g., F, W, Y).	None
7	Combine: rank in first 3 depths, then piece size, then minimise opponent's **corner cells**.	Requires placing the piece to evaluate

Table A.1: *Heuristics attempted for action evaluation in the Alpha-Beta Pruning Agent.*

making it more informative for the agent. The goal is to provide the agent with a reward signal that accelerates and enhances the learning process.

The reward obtained from taking action a in state s is defined as:

$$r_p(s, a) = h_p(s') - h_p(s)$$

where s' denotes the next state resulting from taking action a in state s within the `BlokusEnv` environment, rather than the `SingleAgentBlokusEnv` environment.

Ultimately, when using a discount factor $\gamma = 1$, the cumulative reward over an episode corresponds to the value of the terminal state, $h_p(s^*)$, where s^* is the final state.

The function $h_p(s)$ must satisfy the following conditions: $h_p(s_0) = 0$, where s_0 is the initial state; $h_p(s) > 0$ for an end-state s where the game is won; $h_p(s) < 0$ for an end-state s where the game is lost; and $h_p(s) = 0$ for an end-state s where the game ends in a draw.

A.6 Loss Function Proof

If $|Q(s, a) - (r - \max_{a'} Q(s', a'))| < \tau$, and $\tau \cdot d < 0.5$, where d is the maximum depth of the search tree, value then then if the negamax estimate is off by at most 0.5 from the real value. Therefore, if $|Q(s, a) - Q^*(s, a)| < 0.5$ for every (s, a), then whenever $Q(s, a) \geq Q(s, a')$, it cannot be the case that $Q^*(s, a) < Q^*(s, a')$. This is because, if it were, then

$$\begin{aligned} 0.5 + 0.5 + 0 &> (Q^*(s, a') - Q(s, a')) + (Q(s, a) - Q^*(s, a)) + (Q(s, a') - Q(s, a)) \\ &= Q^*(s, a') - Q^*(s, a) \geq 1 \end{aligned}$$

which is not possible.

A.7 QNetwork Forward Pass Code

```python
def forward(self, encoded_actions):  # (B,)
    (state, mask), (action, flat_action) = encoded_actions
    batch_size = state.shape[0]

    # Combine board state and action mask
    x = torch.cat((state, action), dim=1)  # (B, (21+21+1+1)+(1), 10, 10)
    x = F.relu(self.conv1(x))              # Initial conv
    x = self.resblock1(x)                  # Residual block 1
    x = self.resblock2(x)                  # Residual block 2
    x = self.resblock3(x)                  # Residual block 3
    x = x.view(batch_size, -1)             # Flatten spatial features

    # Action info (pos, shape, piece) embedded + processed by MLP
    pos_emb = self.position_embedding(flat_action[:, 0]) # (B, D)
    shape_emb = self.shape_embedding(flat_action[:, 1]) # (B, D)
    piece_emb = self.piece_embedding(flat_action[:, 2]) # (B, D)
    action_feat = self.action_to_bigger(pos_emb + shape_emb + piece_emb) # (B, D)

    # Encode possible shapes with player info
    shape_ids_emb = self.shape_embedding(self.shape_ids)          # (21*2, D)
    player_ids_emb = self.player_embedding(self.player_ids)       # (21*2, D)
    shapes = (shape_ids_emb + player_ids_emb).unsqueeze(0).expand(batch_size, -1, -1)  # (B,
    ↪  42, D)

    # Mask invalid shapes
    shapes *= mask.squeeze(1).unsqueeze(2)  # (B, 42, D)
    shapes = shapes.view(batch_size, -1)    # Flatten shapes (B, 42*D)

    # Final MLP
    x = torch.cat((x, shapes, action_feat), dim=1)
    x = self.fc1(x)
    x = F.relu(self.fc2(x))
    return self.fc3(x).squeeze(1)  # Output Q-value: (B,)
```

Listing A.1: *Forward pass of the neural network used in the DQN agent.*

B

Project Proposal

Introduction

Motivation

Blokus was the first board game I played in Cambridge, at the end of my first year with some compsci friends from Trinity. I couldn't help but admire its simplicity alongside the challenge of devising a winning strategy. As I played, I found myself picturing various heuristics and useful criteria that could help players evaluate the quality of a board state. Knowing that I now have the time to explore these ideas is exciting, as it's something I would have loved to dive into earlier.

Game Overview

Blokus is a board game in which each player (2-4 players) receives a preset set of 21 pieces in a specific colour. Players take turns placing pieces on the board starting by touching a designated starting position. Each piece must touch at least one other piece of the same colour but only at the corners. The game ends when no more pieces can be placed, and the player with the fewest remaining tiles wins. Blokus Duo (the 2-player version) uses a 14x14 board, while the standard Blokus game (2-4 players) uses a larger 20x20 board.

Project Objective

Develop a library to build and deploy AI agents for Blokus. Focus on solving the 2-player version of Blokus for small boards. Since the game has finite states and an acyclic state transition graph, a deterministic non-losing strategy exists. For simplicity, we assume the first player has the non-losing strategy, although future exploration may reveal this is not the case. The goal is to approximate this optimal strategy by developing an AI agent capable of beating any player.

Starting Point

During the summer, I familiarised myself with Q-learning and various other methods with books and videos. I also gained experience with the OpenAI Gymnasium API, where I developed a basic agent using tabular Q-learning for the game of tic-tac-toe. Additionally, I had some limited exposure to PyTorch, where I trained simple CNNs on the MNIST dataset.

Description

This project focuses on creating a Reinforcement Learning (RL) agent capable of mastering a strategy game on small boards, with potential for scaling. Using OpenAI Gymnasium,

the first phase involves developing a simulation environment where an RL agent learns optimal strategies by training against an "ideal" opponent created through minimax. This pruning technique efficiently calculates winning moves, enabling the project to embed a strong opponent as the agent's adversary. The agent itself will be trained using Q-learning, a method that refines action choices by learning which moves maximise rewards. Once trained, the agent's performance will be assessed based on its ability to counter the optimal opponent's moves effectively. Key data structures and algorithms include alpha-beta pruning for efficient search and Q-learning for action-value learning.

Potential extensions to the project include:

- ◇ Implementing advanced methods such as deep Q-learning (DQN).
- ◇ Define a policy as a probability distribution for different strategies/heuristics, and use policy-based techniques like Proximal Policy (PPO) to learn and optimise this policy.
- ◇ Monte Carlo Tree Search (MCTS).
- ◇ Incorporating heuristic-based rewards to enhance the agent's learning process.
- ◇ Developing self-play mechanisms to enable the agent to improve by competing against itself.
- ◇ Applying transfer learning techniques to leverage knowledge from smaller boards to larger ones.

Success Criteria

The project's success will be defined by the following criteria:

- ◇ **Build** and **set up** the simulation environment using OpenAI Gymnasium.
- ◇ **Implement** and **test** the A1 agent: Tabular Q-learning against a random agent.
- ◇ **Implement** and **test** the A2 agent: Tabular Q-learning against a minimax-optimised opponent.
- ◇ **Achieve a 75% win rate** against a random agent on 7×7 boards using tabular Q-learning.
- ◇ **Achieve a 75% win rate** against a minimax-optimised opponent on 7×7 boards using tabular Q-learning.

Possible Envisaged Evaluation Metrics

Evaluation metrics for this project will include win rates against random and heuristic-based agents, performance consistency as board sizes scale from 2x2 up to 14x14, and time to convergence. Core metrics will focus on win rates against an optimal opponent, while extension- specific metrics will assess win rates against random or greedy-policy agents on larger boards where finding an optimal opponent is impractical, as well as comparative win/loss rates between various agents.

Work Plan Timetable

We detail a work plan for this project. Note that *this work plan is provisional*, and indeed some work items may be completed earlier or later than anticipated.

Note: There are 8 weeks during which no specific work is allocated. Based on the research period that I will make in the last two weeks of December, I will identify which extensions are most feasible and beneficial within the given timeframe. I will then update the timeline accordingly.

Michaelmas Term

Time Period	Milestone	Details
21 Oct - 03 Nov (2 weeks)	OpenAI Gymnasium Env Compatibility	Ensure compatibility of the project with the OpenAI Gymnasium framework, setting the foundation for the simulation environment.
04 Nov - 10 Nov (1 week)	Minimax for Small Boards	Implement the minimax algorithm to determine optimal moves on small boards, aiding in the development of a strong opponent for training the RL agent.
11 Nov - 17 Nov (1 week)	Research and LaTeX Setup	Conduct a literature review on previous attempts in similar projects and establish a LaTeX environment for dissertation writing to streamline documentation.
18 Nov - 01 Dec (2 weeks)	A1: Tabular Q-learning against Random Agent	Develop and implement tabular Q-learning to train the agent against random and heuristic-based agents, gathering baseline performance metrics.

Michaelmas Vacation

Time Period	Milestone	Details
02 Dec - 15 Dec (2 weeks)	A2: Tabular Q-learning against Optimal Opponent	Enhance the Q-learning implementation by training against the optimal opponent derived from the minimax strategy, assessing the agent's adaptability and strategy improvement.
16 Dec - 29 Dec (2 weeks)	Catch Up and Research	Use this period to catch up on delays and review papers on scaling strategies. This coincides with holidays, allowing for a balance of work and rest.

30 Dec - 12 Jan (2 weeks)	A3: Develop DQN-style Agent for Small Boards	Begin the development of a Deep Q-Network (DQN) agent specifically designed for small board configurations, incorporating advanced reinforcement learning techniques.

Lent Term

Time Period	Milestone	Details
13 Jan - 26 Jan (2 weeks)	Extension-Focused Dev	Focus on further extending the agent's capabilities, exploring additional strategies and enhancements.
27 Jan - 02 Feb (1 week)	Progress Report	Prepare a comprehensive progress report outlining achievements, challenges faced, and next steps in the project timeline.
03 Feb - 09 Feb (1 week)	Catch Up and Rest	Allocate time for any catch-up work – e.g., finishing the report – and rest.
10 Feb - 23 Feb (2 weeks)	Extension-Focused Dev	Continue enhancing the agent's features and exploring new strategies, ensuring robust development. (+ Exam Revision + Dissertation Writing)
24 Feb - 04 Mar (2 weeks)	Extension-Focused Dev	Continue enhancing the agent's features and exploring new strategies, ensuring robust development. (+ Exam Revision + Dissertation Writing)
10 Mar - 23 Mar (2 weeks)	Extension-Focused Dev	Continue enhancing the agent's features and exploring new strategies, ensuring robust development. (+ Exam Revision + Dissertation Writing)

Lent Vacation

Time Period	Milestone	Details
24 Mar - 06 Apr (2 weeks)	Finishing Dissertation Draft	Focus on finalising the draft of the dissertation, ensuring comprehensive coverage of all project aspects.

06 Apr	Submit 1st Dissertation Draft	Submit the first draft of the dissertation to the supervisor for review and feedback.
07 Apr - 20 Apr (2 weeks)	Waiting for Feedback	Await supervisor feedback while preparing for revisions.

Easter Term

Time Period	Milestone	Details
21 Apr - 27 Apr (1 week)	Apply Feedback and Submit 2nd Dissertation Draft	Implement the feedback received on the first draft, refining the dissertation content, and submit the second draft of the dissertation to the supervisor for further evaluation.
28 Apr - 7 May (1.5 week)	Wait for Feedback	Await further feedback from the supervisor.
8 May - 14 May (1 week)	Apply Feedback + Dissertation Completion	Revise the dissertation based on feedback from the second draft, finalize the document, and prepare for submission.
15 May	Submit Dissertation and Code	Complete the submission of the dissertation and source code by the deadline.

Computing Resources and Backup Plan

Own Machine for Coding

Specifications: MacBook Pro (16-inch, M2 Pro, 2023), 16 GB RAM, 512 GB SSD

Backup Plan: **Version Control and Redundancy**. All code will be version-controlled through GitHub, with daily checkpoints uploaded to Google Drive. In case of hardware or software failure, I will utilise the Computer Laboratory (CL) facilities or SSH server.

Responsibility Clause: *I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.*

University's High-Performance Computing (HPC) Resources

Usage: **Model Training and GPU-Accelerated Experiments**. The HPC resources might be used to train models on large-scale simulations and for GPU-accelerated experiments for my extensions.

Contingency Plan: If the University's HPC resources are unavailable or heavily loaded, I will use Google Cloud or AWS for computationally intensive tasks. I do not currently have access to cloud services, but I will purchase a subscription (e.g., Google Cloud)

if necessary and ensure they are properly configured for the project. However, it is always possible to find a board state small enough such that any computing resources I need will be within reach.