Trinity College Dublin

The University of Dublin

Reinforcement Learning and Minimax in a Simple Game

Hugh Langan

Final Year Project

Trinity College Dublin

Contents

Abstract

This project investigates how a reinforcement learning approach compares with minimax, a tree-search method in the game of *Tic Tac Toe*. The primary goal is to determine under what conditions a reinforcement learning agent can reproduce minimax's optimal play and whether it can outperform the minimax algorithm when facing a suboptimal opponent. Three different training setups are evaluated: training a reinforcement learning agent against a minimax opponent, training through self-play, and training a reinforcement learning agent against various suboptimal opponents.

Report Structure

- **Section 1** introduces *Tic Tac Toe*, the decision-making algorithms minimax and Q-learning, and states project goals.
- **Section 2** discusses related work, and the gap in the literature that this project aims to fill.
- **Section 3** gives a high-level overview of the components that were developed for this project.
- **Section 4** goes into details on the implementation of components developed for this project.
- **Section 5** discusses the solutions to further issues encountered during the development.
- **Section 6** presents and discusses empirical results of policy analysis and game simulations
- **Section 7** concludes the findings of this project and discusses directions for further work.

1 Introduction

This section will introduce the game of Tic Tac Toe, the concepts of the Minimax algorithm, reinforcement learning, Q-learning, and discuss the goals of this project.

~~X~~	~~X~~	~~X~~
O	O	
O		X

Figure 1: Player wins with three Xs in a row.

1	2	1
2	2	1
1	1	2

Figure 2: Tied board: all spaces filled with no complete line.

1.1 Tic Tac Toe

Tic Tac Toe is a simple, two-player, turn-based game. The game is fully deterministic and fully observable: there is no random chance involved in gameplay, and both players have complete knowledge of the game state at all times. Players take turns placing their pieces on empty spaces of a 3×3 grid, with the objective of completing a horizontal, vertical, or diagonal line of their own pieces.

Figure 1 shows a game state where player 'X' has won by completing a horizontal line. In this project, the traditional 'X' and 'O' markers are replaced with the integers '1' and '2' to ensure that the order of moves is always clear; player 1 moves first. If the board fills without either player completing a line, the game is considered tied, as shown in Figure 2.

The state space of Tic Tac Toe is relatively small: there are $9! = 362,880$ possible move sequences and 26,830 distinct states. This small state space makes Tic Tac Toe feasible for approaches that involve exhaustive search, such as the Minimax algorithm, as well as for reinforcement learning algorithms.

1.2 Minimax

Minimax is a decision-making algorithm used for two-player zero-sum games. As discussed in Russell and Norvig (2020), for any given game state, the minimax algorithm performs an exhaustive recursive search of all possible subsequent game states, until it reaches a terminal state. These terminal states are assigned a score based on the outcome (1.0 for a win, 0.5 for a draw, 0 for a loss), and the algorithm will select an action that provides the best guaranteed score, ie, the action with the maximum payoff in its worst outcome. In Tic Tac Toe, minimax can always guarantee a draw in the worst case, regardless of whether it moves first or second.

The formal recursive definition of the Minimax algorithm is shown in Equation 1. Nodes are divided into "MAX" nodes, where the algorithm aims to maximise its own reward, and "MIN" nodes, where the algorithm aims to minimise the opponent's reward. Terminal states are directly assigned their value based on the result of the game (defined as Utility(s) in the equation), acting as the base case of the recursion.

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s) & \text{if } s \text{ is a terminal state} \\ \max_a \min_{s'} \text{Minimax}(s') & \text{if } s \text{ is a MAX node} \\ \min_a \max_{s'} \text{Minimax}(s') & \text{if } s \text{ is a MIN node} \end{cases} \tag{1}$$

Minimax's behaviour of optimising for the best minimum payoff allows it to guarantee the maximum possible reward against an optimal opponent: since the opponent's goal is to minimize the other player's payoff, an optimal opponent will lead to a minimal payoff from any action.

Against a suboptimal opponent, the minimax algorithm will still achieve a minimum result of a draw. However, it is not necessarily the optimal strategy in this case. There is a possibility for a different model to achieve a win where minimax would only achieve a draw.

In a situation where the algorithm has to choose between two actions: one which could lead to either a win or a draw, and one which can only lead to a draw, minimax will assign both the same value. Against an optimal opponent this is a valid approach, as the opponent will guarantee the minimum payoff from each action, and both actions will lead to a draw. Against a suboptimal opponent, however, the chance for the opponent to make a suboptimal move and allow a win means the first action should be assigned more value. Since minimax searches based on the worst-case outcome, branches where the opponent makes a mistake are ignored.

As also discussed in Russell and Norvig (2020), minimax has limitations that make it inapplicable in many situations. It relies on an exhaustive search of all possible future game states. The number of game states grows exponentially in more complex games, and performing a minimax search quickly becomes infeasible. It requires a full knowledge of the game state so it cannot handle games with any hidden information (like many card games), and it cannot handle games where actions have probabilistic outcomes (e.g. dice rolls). Many real-world applications also involve probabilistic outcomes, imperfect observability or infeasibly large state spaces, making minimax inapplicable.

In games with larger state spaces, alpha-beta pruning can be used to reduce the number of game sequences the minimax algorithm searches, without affecting its optimality (Knuth and Moore, 1975). Due to the small and solvable state space of Tic Tac Toe, this was not necessary in this project.

1.3 Reinforcement Learning

Reinforcement Learning is an approach to decision-making in which an agent learns to select actions that maximise its long-term reward through interacting with an environment and receiving rewards based on the outcomes of its actions (Sutton and Barto, 2018). Reinforcement learning allows an agent to learn optimal behaviour through trial and error in the environment. Compared to a decision-making approach like minimax which requires a complete model of the environment and is based on assumptions of an opponent's behaviour, reinforcement learning provides more flexibility. As well as making fewer assumptions about its environment, reinforcement learning can function under probabilistic actions and partial observability.

Formally, a reinforcement learning problem is defined as a Markov Decision Problem (MDP), a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ where:

- $\mathcal{S}$ is the set of possible states,

- $\mathcal{A}$ is the set of available actions,
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ specifying the possible subsequent states, and their probability distribution if the environment is non-deterministic
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and
- $\gamma \in [0, 1]$ is the discount factor, impacting the weight of future rewards

At each step, the agent observes the current state $s_t \in \mathcal{S}$ of the environment, selects an action $a_t \in \mathcal{A}(s_t)$ which causes a transition to a new state $s_{t+1} \sim T(\cdot \mid s_t, a_t)$, and receives a reward $r_{t+1} = R(s_t, a_t)$ based on the new state. The goal of the agent is to maximise the expected return, which is defined as the sum of future rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

In Tic Tac Toe, the state is the position of all pieces on the board, and the actions are the valid moves from the current state. Transitions are mappings from the current game state to possible subsequent game states based on the action selected. The reward would be granted based on reaching a terminal winning or losing state, for example, +1 or -1 for winning or losing respectively, and 0 in any other state. The output of this reinforcement learning model is a "policy", $\pi : \mathcal{S} \to \mathcal{A}$; a mapping between states and actions, representing an optimal move to make from any given game state. In a two player game, the transitions between states are based not only on the reinforcement learning agent's own actions, but also the actions of the opponent. As a result, the output of the reinforcement learning model can be influenced by the opponent it is trained against.

Reinforcement learning has been proven to be effective in problems where an exhaustive search approach like minimax is computationally infeasible. An example is AlphaZero developed by DeepMind, which was applied to games including Chess and Go, which both have state spaces too large for a full tree search to be possible, and outperformed previous programs in these games (Silver et al. (2018)). AlphaZero was trained through self-play, where both players are independent reinforcement learning agents. This approach relies purely on trial and error to find an optimal policy; no initial knowledge of the game is required.

1.4 Q-Learning

Q-learning is a reinforcement learning algorithm introduced by Watkins in 1989 and formalised by Watkins and Dayan in 1992 (Watkins and Dayan, 1992). Q-learning aims to estimate the expected cumulative reward of taking a given action from a given state, formalised as the *action–value function* $Q^*(s, a)$. It is *model-free*, meaning it does not require prior knowledge of the environment's transition function $T(s, a)$. It is also *off-policy*, meaning it can learn an optimal policy even when suboptimal actions are taken during training.

Q-learning operates by assigning each state-action pair (s, a) a *Q-value*, which represents the expected cumulative reward for taking that action from that state. This value is constantly updated during training as the model explores its environment. The model maintains a *Q-table* containing a Q-value for each possible action from each seen state in the environment. After each step, the Q-table is updated using the formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

In this formula, $\alpha_t \in (0, 1]$ is the learning rate, which determines to what degree new information will override existing estimates. r is the reward gained from taking action a. $\gamma \in [0, 1]$ is the discount factor, which controls the amount to which an immediate reward is valued over a future reward. Once training is completed, the Q-table can be used to create a policy by selecting the highest Q-value action for each state. With sufficient time for training and exploration, Q-learning will converge to the optimal policy $Q(s, a) \to Q^*(s, a)$ with a probability of 1 in a finite MDP (Sutton and Barto, 2018).

The exploration vs exploitation problem is a central challenge in Q-learning, and reinforcement learning more generally. Since the agent has limited knowledge of its environment, generating an optimal policy depends on the agent effectively exploring its environment to find actions that lead to reward states. At every step, the agent has to make a choice between taking the action that it believes to be the best based

on its current knowledge of the environment ($\arg\max_a Q(s,a)$), or to explore a suboptimal or unknown action, which could potentially lead to a higher reward in subsequent states. An *exploration policy* is used to make this decision. If this exploration policy is too conservative (ie, does not explore sufficiently, takes early rewards) the Q-learning agent can converge at a local maximum without finding a truly optimal policy. An exploration policy that causes excessive exploration will lead to inefficient training.

1.5 Goals of the Project

The goal of this project is to investigate the effectiveness of Q-learning as a method for learning strategies in Tic Tac Toe, and to compare a reinforcement learning approach to minimax, a more traditional tree search approach. Specifically, the project aims to:

1. **Replicate Minimax's Optimal Play** Train a Q-learning agent that achieves the same guaranteed outcome as minimax (ie, never worse than a draw against any opponent) and test this through (i) minimax–agreement scores and (ii) game simulation

2. **Evaluate self-play as a training method** Investigate the use of self-play as a training method, to see if a reinforcement learning approach will reach the same optimal strategy when not trained using an opponent with hard-coded behaviour

3. **Measure adaptability and generalization** Train and evaluate Q-learning agents against various suboptimal opponents with minimax as a benchmark for performance.

4. **Compare Computational Efficiency** Compare the computational cost of a minimax approach compared to a Q-learning approach, in a common environment

Demonstrating that a Q-learning agent can both match the performance of minimax under optimal play, and also outperform it under suboptimal play would indicate that a reinforcement learning approach can have a clear practical advantage, even when an exhaustive search is feasible.

2 Literature Review

A relevant study to this project is Hussain (2023), which also aims to compare a reinforcement learning approach and a minimax approach to optimally playing Tic Tac Toe. This evaluates Q-learning specifically and demonstrates that a Q-learning agent trained to play Tic Tac Toe can outperform a random player. However, the details of the training procedure are limited, and the results suggest that the Q-learning agent did not converge to an optimal policy and could not match the performance of the minimax agent. The study supports the feasibility of Q-learning as a general approach to strategy learning in Tic Tac Toe but does not explore how the choice of opponent during training affects the final learned policy. In addition to this, although the study evaluates the Q-learning agent's performance against a suboptimal opponent (a random player), there is no baseline provided to give context to this result. This project aims not just to compare a Q-learning agent directly to a suboptimal opponent, but to compare its performance to a minimax agent's performance against that same suboptimal opponent.

Another relevant study is He et al. (2016), which explores the effects of opponent behaviour during training on the final policy generated by reinforcement learning models. The paper explains that creating a training environment with a static opponent can lead to a policy that is unable to adapt to changing or suboptimal opponents. It introduces the Deep Reinforcement Opponent Network (DRON), which aims to produce more adaptable policies through opponent modelling during training. The findings support this project's investigation into using different opponents during training to generate more effective and general policies. The paper evaluates the impact of opponent modelling in environments with state spaces too large for exhaustive search-based algorithms like minimax to be feasible. This project builds on that idea by asking whether a reinforcement learning approach can still provide an advantage even in domains with small, fully solvable state spaces where exhaustive search is feasible.

Also relevant to this project, Korkmaz (2024) explores the limitations and some proposed solutions around the generalization of reinforcement learning policies. The paper discusses how poor generalization

often comes from overfitting to limited training experiences. Although this analysis focuses on significantly more complex environments than Tic Tac Toe, many of the ideas discussed here are relevant to this project.

One relevant idea from Korkmaz (2024) is that the generalization ability of a reinforcement learning agent is directly linked to its exposure to diverse states during training. This highlights another aspect of the importance of opponent selection during training of a Q-learning model in this project. Since Tic Tac Toe is a two-player game, when training a Q-learning agent to play the game, the diversity of states which that agent will see during training is influenced not only by its own actions but also by the actions of the opponent. An overly consistent opponent may lead to a lack of diversity of states in training and could have similar effects to a poor exploration policy, ie, convergence on suboptimal policies or poor generalization. In this project, the relationship between the diversity of training and the generalization of the output policy will be further explored through evaluating training with opponents that will allow for varying degrees of state exploration.

Given this review of literature, existing studies on this topic either: assess Q–learning against a single weak opponent without a minimax benchmark (Hussain, 2023), or study opponent modelling in domains too large for exact search (He et al., 2016). Existing work does not evaluate how opponent diversity can affect the outcome of reinforcement learning training with the minimax algorithm as a baseline. This project fills this gap by:

- training Q-learning agents under 3 primary conditions: against an optimal opponent, through self-play, and against sub-optimal opponents
- evaluating the performance of these agents, with a focus on comparison to minimax
- exploring the potential benefits of a reinforcement learning approach, ie, exploitation of opponents and computational cost

3 Overview of Problem Situation

This section provides an overview of the components that were developed in order to achieve the goals of the project. This project required the development of several key components: a game environment that allowed for different player types including a human player with visualisation of moves and game states, a minimax agent, a Q-learning agent and an environment for training this Q-learning model, and methods for evaluating policies or algorithms, including an environment for simulating large numbers of games between different agents.

3.1 Game Environment

A "Main" environment for simulating a game between any two player types, including a human player, was implemented. This includes visualisation capabilities and is crucial for debugging and ensuring correctness of results as it allows a user to see the interactions between different agents, or manually input moves to see how they react. While other environments created in this project primarily output raw results without visualisation, this tool proved to be valuable in helping to interpret those results and understand different agent's behaviour.

A separate environment for training a Q-learning agent was created. This environment allowed for selection of hyperparameters, training intervals, and the opponent which the agent should be trained against. The policy generated from this training is saved for evaluation and testing purposes.

A "Matchup" environment was created for simulating a large number of games between any two selected players. These include Q-learning agents, minimax, and various suboptimal opponents which will be introduced later in this report. This will run a specified number of games and output result statistics. Since many agents involved in this project behave non-deterministically, some matchups between 2 agents can result in a large number of different game sequences. For this reason the statistical output from this environment complements the Main visualisation environment, giving information on the consistency of different strategies which would have been difficult to observe through simulating individual games.

Finally, a self-play environment was created for training two Q-learning agents simultaneously. In this environment the two agents start with no prior knowledge, and build up their policies through trial and error

in repeated simulated games against each other. This environment is used to investigate if optimal policies can be generated through self-play, as opposed to through training against a hard-coded opponent.

3.2 Minimax Agent Development

An implementation of the minimax algorithm was a central component of this project. This serves as a performance benchmark for other strategies to be compared to, and a basis for evaluation of policies. In a fully solvable environment like Tic Tac Toe, minimax is a strong reference point for analysis of other strategies.

The minimax agent is used in the Main environment, the Q-learning training environment, and the Matchup environment. Given a board state, it returns a move which is optimal under the conditions of minimax, ie, a move which maximises the minimum achievable reward.

In addition to this functionality, the minimax implementation is capable of scoring a move which is passed to it. This follows the same minimax reward logic, assigning the move a value based on the worst possible outcome from the perspective of the acting player. This is used as a method of analysing Q-policies, to measure how many moves within a policy are optimal or suboptimal according to minimax.

3.3 Q-Learning Agent Development

The Q-learning agent is designed to create an effective policy for game-play through trial and error and interaction with its environment. Unlike minimax which has a complete knowledge of the game's state space, the Q-learning agent begins with no prior knowledge of the game, and must develop this policy through exploration and maximising expected cumulative rewards.

In this project, a Q-learning agent will be required to be able to interpret a Tic Tac Toe board state, recognise winning or losing states, and recognise legal moves from any board state. In training it will have to generate and maintain a persistent Q-table, assigning a value to each move from each board state.

A Q-learning agent will also be created for use in the Main and Matchup environments, which takes a previously generated Q-table and uses this as its policy for playing the game, ie, selecting the highest Q-value move in each position according to that Q-table.

An implementation of Q-learning also requires an approach to solving the exploration vs exploitation problem as discussed in Section 1.4. To effectively learn a strategy to play the game, thorough exploration of the game's state space must take place during training. This exploration comes with drawbacks as excessive exploration of actions that are known to be suboptimal can lead to wasted computational resources with no payoff. An inappropriate approach to this problem can lead to convergence on a suboptimal strategy or inefficient training.

3.4 Evaluation Strategy

Two primary methods were used for evaluation of the performance of trained Q-learning models: agreement with the minimax algorithm and performance in game simulations. A state in a policy that is in agreement with minimax is defined as one whose greedy action is including in the minimax arg-max set for that same state. These methods each provided valuable and complementary insights on different models' performance.

The agreement with minimax metric provided an almost instant assessment of a Q-learning model, without the need for extended simulations. When a model significantly underperformed it was generally immediately clear from the output of this program that this was the case. However, as discussed in Section 1.2, there is a possibility for a policy generated by a reinforcement model to both uphold the minimum reward guarantee of the minimax algorithm, and also outperform the minimax algorithm against a suboptimal opponent. In this case, this method of evaluation is not sufficient, as the output from this program would only state that the policy has a 100% agreement with minimax; there is no capability for it to assign any higher value to these actions, even if they lead to better results in practice.

For evaluating these policies that were *as good or better* than minimax, practical simulations were used. Through large numbers of simulated games, it was clear when a policy was outperforming the minimax algorithm with statistical significance. These simulations were also valuable in backing up the findings from the agreement with minimax metric. For example, if a policy showed a 100% agreement with minimax, then

it should either win or draw every game against any opponent according to the definition of minimax. The Matchup environment allowed this to be tested and confirmed through simulation.

4 Design and Implementation

This section details the implementation of each major component developed for this project and the key design decisions made during the development process. These components include the game environments, agent implementations, and evaluation strategies. The full C++ source code, including some trained models and simulation outputs, is available at: `https://github.com/hughjsl/final_year_project`.

4.1 Game Environment Implementation

The game environments were implemented from scratch in C++, which provides good performance and memory control for intensive model training or simulations. The primary logic of the game Tic Tac Toe is defined in `tic_tac_toe.cpp`. This includes functionality for defining and storing the game board, validating moves, checking for terminal states or winning states. The functions implemented here, such as `makeMove()` for placing a piece, or `checkWin()`, for detecting a winning state, are used across all other components of this project. A method for printing a board state for visualisation within the command-line is also provided.

Game states are represented here as a two-dimensional 3×3 matrix:

```
board(3, std::vector<int>(3, 0));
```

Each index in this matrix is initialised to 0 to represent an empty cell, and updated with the integers 1 or 2 to denote a piece placed by that player. As well as helping to clarify which player moves first, this simplifies the implementation of minimax and Q-learning agents, by allowing them to work directly with integers instead of placing characters ('X' and 'O') and translating these to integers.

The first method of interacting with this Tic Tac Toe implementation was implemented in `main.cpp`. This is a command-line interface for allowing a user to directly test different agents. The user is prompted to select a player type (ie, human, Q-learning agent, minimax, or other suboptimal agents) for each player, and that game will then be simulated. After each turn the board state is printed to provide visualisation of the game.

Also included in `main.cpp` is the environment used for training Q-learning agents. This environment includes configuration for all training variables required for the scope of this project, including number of episodes, learning rate, discount factor, and the opponent to be used in training. The exploration strategy is also implemented here, as this is only needed in training and not during testing.

The training loop for Q-learning is implemented as a sequence of Tic Tac Toe games between the learning agent and its opponent. Training is split into episodes, which in this case represent one complete game of Tic Tac Toe. In each episode, the game board is reset, and the two players take alternating turns until a terminal state is reached. After each action by the Q-learning agent, the agent receives a reward based on the new state of the board which is immediately used to update the Q-table. The logic for updating this table is contained within the Q-learning agent itself, which will be discussed later. A simple ϵ-greedy exploration strategy was implemented here: on each turn the Q-learning agent will make a choice between choosing the action it currently thinks is optimal (the *greedy* action), or exploring a random other action. This decision is based on a random chance with the probability ϵ, where ϵ is taken as a hyperparameter of the model. In testing, ϵ is set to 0, so the agent will always take the greedy action.

A timer function was also implemented in the training loop. This tracked the time taken for each 1,000 episodes of training to complete, and output a live estimate of the total time training would take to finish. This proved valuable in practice as training could take many hours depending on the number of episodes, and which opponent was selected.

For practical evaluation of the performance of trained Q-learning agents against a variety of opponents, a dedicated simulation environment was implemented in `matchup.cpp`. This environment is designed to simulate a large number of games between two specified agents and output performance metrics based on these results.

This program also operates through the command-line. When the program is executed, the user is prompted to select an agent to act as player 1, and then player 2. The user can input a Q-learning policy (in the form of a `.dat` file), select a minimax agent, or various suboptimal agents. The user then selects the number of games to simulate, and the simulation starts. The output is recorded to a `.txt` file as well as the command-line, to avoid data loss when running long simulations. Similar time estimation functionality was added here, to help when running simulations that would take many hours.

Agent behaviour is abstracted through a function pointer, allowing the simulation logic the flexibility to work with many different player types, without requiring switch statements at every step. Q-learning agents act entirely greedily in the simulation environment, ie, there is no exploration taking place, it will select the move it believes to be optimal on every turn.

The primary output from this program is simply the number of wins for player 1, draws, and wins for player 2. In most cases this was the only information needed from these simulations. The environment also tracked the number of unique terminal board states that were seen during the simulation.

4.2 Minimax Agent Implementation

The minimax agent implemented in this project is capable of selecting moves in Tic Tac Toe by performing a full recursive search of the game tree from any given position. This agent guarantees an optimal outcome against an opponent who is also playing optimally, and ensures a worst possible outcome of a draw against any opponent. This provides a useful training tool for Q-learning models, and also a valuable benchmark of "optimal" play for evaluating the performance of reinforcement learning agents.

The implementation of the minimax agent has two main methods: `scorePosition()` and `getBestMove()`. The `scorePosition()` function is a recursive function that evaluates all possible game outcomes from a given board state, for the specified player. If the board state is terminal, the function directly returns a score: `1.0` if the player has won, `0.0` if the player has lost, and `0.5` if the game is a draw. This acts as the base condition for the recursion.

In a non-terminal state, the function starts to iterate over all valid actions possible from the current board state. For each valid action, it will simulate placing that piece to generate the resulting board state, and then it will recursively call `scorePosition()` with that updated board state, but from the opponent's perspective. This recursion will continue until a terminal state is reached. While `scorePosition()` is iterating over the valid actions, a best possible score that the opponent can achieve given each action is taken is tracked. This represents the worst outcome for each action, the value that minimax aims to optimise. The final value returned by `scorePosition()` is `1.0 - bestOpponentScore`, the score that minimax would assign to the given board state.

This exhaustive approach to minimax would be computationally infeasible in many other simple two-player games. The relatively small state space of Tic Tac Toe and the extremely small depth (given a board has 9 spaces and one piece is placed each move, the maximum length of a game is 9 moves) make it an ideal environment for minimax to be applied without limitations. In a game with a larger state space, limitations would have to be applied to the minimax algorithm in order to make it feasible; for example, alpha-beta pruning or limiting the search depth. Either of these would reduce the computational complexity of the algorithm, but also risk sacrificing the guarantee of mathematical optimality that a full minimax search provides.

Pseudocode describing the recursive function of `scorePosition()` is provided in Listing 1.

```
function scorePosition(board, player):
    if player has won:
        return 1.0
    if opponent has won:
        return 0.0
    if board is full:
        return 0.5

    bestScore = 0.0
    for each valid move:
        simulate move
        score = 1.0 - scorePosition(newBoard, opponent)
```

```
        bestScore = max(bestScore, score)
        undo move

    return bestScore
```

Listing 1: Pseudocode for `scorePosition()` in the minimax agent

The `getBestMove()` function uses `scorePosition()` to evaluate all possible moves for the current player. For each move, the move is simulated to get the subsequent board state, that board position is scored according to `scorePosition()`, and the move is reverted. All moves that would achieve the maximum score are returned as a list.

In many positions, there is no single best move according to minimax, but a larger equivalence class of moves that achieve the same score. An option was added within the minimax class for configuration of how the minimax agent will select a move from this equivalence class. Two selection policies were implemented: the agent can take the first move from the set, or select randomly from the set. Choosing the first move from the set is mostly arbitrary, but it provides the ability to make the minimax agent act fully deterministic which is valuable for repeatability. Choosing a random move from the set makes the minimax agent act non-deterministically.

For most applications within this project, the minimax agents were set to choose randomly, as it was preferable for them to act non-deterministically. This gives a few key benefits; when using a minimax agent as an opponent in the training of a Q-learning algorithm, if the minimax agent only ever chose the first action within its set of optimal actions, it would leave many actions unseen during training. Also, when simulating games between a Q-learning agent and a minimax agent, the Q-learning agent already acts deterministically. If the minimax agent was also deterministic, these two agents would only ever produce a single, identical series of moves. Making the minimax agent act non-deterministically means there are many distinct games possible, which gives a more useful indication of the general performance of that Q-learning agent.

4.3 Q-Learning Agent Implementation

The operation of the Q-learning agent developed for this project is based on the standard Q-learning value update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

This formula is applied after each move made by the Q-learning agent to update the values in the Q-table. The current state s, the performed action a, the subsequent state s', and the reward r are passed to the update rule which calculates the new Q-value for that entry in the table, accounting for the agent's learning rate α and discount factor γ. In a terminal state this formula is not used, the reward for that state (`1.0` if the player has won, `0.0` if the player has lost, and `0.5` if the game is a draw, similar to minimax) is directly written to the table.

The ϵ-greedy exploration strategy is used by the Q-learning agent during training. ϵ is selected as a hyperparameter of the model. For each action taken during training, the agent will have an ϵ probability of selecting a random valid action to take, and a $1 - \epsilon$ probability of selecting the move it currently believes to be optimal, ie the highest Q-value action for the given state.

The Q-table is implemented in the form of an unordered map: `std::unordered_map<std::string, std::array<double, 9>>`. This is a mapping of encoded board states to arrays of Q-values. The arrays have a fixed size of 9, as each Q-value represents an action, and there are a maximum of 9 possible actions from any given board state. The board state is encoded as a 9-character string, where each character represents a space on the board (0 for an empty space, 1 for player 1, 2 for player 2).The board is encoded in row-major order. This approach is efficient in terms of storage, and allows board states to be used directly as keys for a state lookup in the Q-table. In a worst case scenario this, where every possible board state was included in the policy, this would have a memory use of $3^9 \times 9 \times 8$ bytes $\approx$ 1.4MB. In practice this is significantly lower as not all states can be explored.

When training is completed, the Q-table is saved to a binary `.dat` file. Each entry in this table is the string of the encoded board state, followed by 9 double precision values storing the Q-values for each possible

move. When a Q-learning agent is loaded in another environment, this binary file can be easily reconstructed into the unordered map.

Due to the way Q-learning has been implemented here, no further changes are required to enable self-play. The agents' behaviour when being trained against a hard-coded opponent or when being trained against another Q-learning agent are identical, the only difference is in the environment itself and the implementation of the training loop (discussed in Section 4.1).

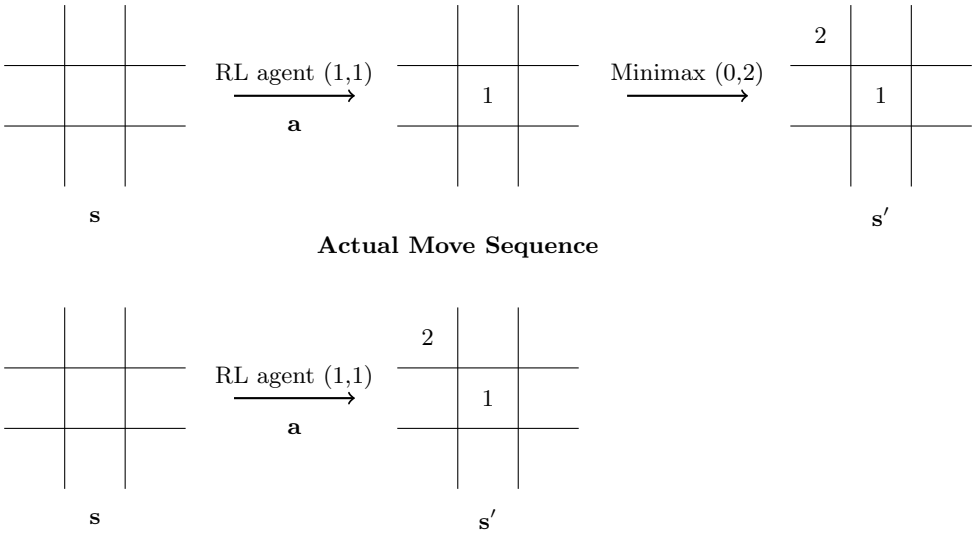

Figure 3: From the Q-learning agent's perspective, both its own move and the opponent's response are seen as a single transition from state **s** to state **s′** via action **a**.

Figure 3 represents the approach taken to applying Q-learning to a two player environment in this implementation. In Q-learning, an agent learns by observing transitions in the form (s, a, r, s'), where s is the current state, a is the action performed, r is the reward received, and s' is the resulting state. However, in the case of a two-person game, there is not a direct mapping from one state to the next when an action is taken, because there is an opponent who alters the environment between each of the Q-learning agent's turns.

The solution to this problem was to combine both players' moves into a single transition. That is, when the Q-learning agent takes action a from state s, both its own move and the opponent's move are performed before the new state s' is returned to the Q-learning function. This essentially reduces the opponent to a part of the environment, the Q-learning agent will learn to see the placement of both of these pieces as direct result of the action it performed. The reward r is also applied after both pieces have been added to the board state, so it represents to combined outcome of both players actions.

The diagram illustrates one of the effects of this abstraction: there is a intermediate state between the Q-learning agent's own action and the opponent's action that the Q-learning agent never sees. Since two pieces are placed between each seen state, if a Q-learning agent is trained to play as player 1, it will only ever have seen board states with an even number of pieces, or similarly only an odd number of pieces if trained as player 2. For a complete policy that contains all board states, a Q-learning agent would have to be separately trained as both player 1 and player 2.

4.4 Evaluation Strategy Implementation

Two separate evaluation strategies were implemented for the purposes of evaluating the effectiveness of Q-learning policies generated. The first performs an evaluation by directly reading the policy and scoring its actions according to the minimax algorithm, and the second performs a more practical evaluation of the policies by allowing for repeating simulation of a Q-learning agent against various opponents.

The first evaluation strategy is implemented in `analyze_policy.cpp`. This iterates through the board states found in a policy (output from the training of a Q-learning model). For each state in this policy, the program decodes the board state and its 9 associated Q-values. It finds the move with the highest Q-value out of these, ie, the move that the Q-learning agent would select from that board state. Then, it iterates through all possible moves from the board state, simulates each of these moves and uses the `scorePosition()` function of minimax (described in Section 4.2) to score the resulting state. If the move preferred by the Q-learning policy is in the set of optimal moves according to minimax, the policy is considered "in agreement" with minimax for that board state. This analysis is performed for each board state in the policy, and the program returns the number of board states where the policy is in agreement with minimax, out of the total number of board states in the policy. Pseudocode explaining the operation of this program is given in Listing 2.

```
for each board_state in Q_policy:
    q_values = getQValues(board_state)
    q_action = argmax(q_values)
    count_agree = 0

    max_minimax_score = 0
    best_moves = []

    for each valid move in board_state:
        simulated_state = applyMove(board_state, move)
        score = scorePosition(simulated_state, opponent)

        if score > max_minimax_score:
            max_minimax_score = score
            best_moves = [move]
        elif score == max_minimax_score:
            best_moves.append(move)

    if q_action in best_moves:
        count_agree += 1

agreement = count_agree / total_states
```

Listing 2: Pseudocode for `analyze_policy.cpp`

This tool was useful for evaluating Q-learning policies because it gave a near-instant result, whereas evaluation through simulation could take many hours, depending on the number of games to simulate and the type of opponent selected. Agreement with minimax roughly represents the policy's ability to uphold minimax's main characteristics: performing optimally against an optimal opponent, and guaranteeing a minimum result of a draw. A policy with low agreement will always under-perform in simulation, so for the purposes of finding reinforcement learning models that can perform optimally, this can save a lot of simulation time.

The total number of states within a policy is also a valuable metric for evaluating the quality of that policy. Since new states are added only when they are encountered in training, the total number of states gives an indication of the effectiveness of that model's exploration policy. It also gives some insight into how that model will perform. A Q-learning agent has no Q-values for any actions in a game state that is not in its policy, and if it is presented with these board states it will have to choose an action completely arbitrarily. This means a Q-learning agent with a small number of states in its policy may perform no better than a random player in some situations, even if it shows high agreement with minimax within the policy.

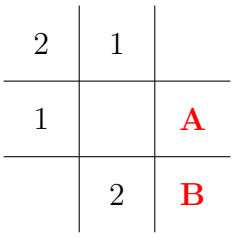

Figure 4: Illustration of two moves (A, B) that minimax rates equally although A offers a potential for a win against a suboptimal response

The limitation of this approach to evaluating a policy comes up when trying to evaluate whether a policy will outperform minimax against a suboptimal opponent. Consider the example board state shown in Figure 4. It is player 1's turn to place a piece. From this game state, if both players are playing optimally, placing a piece in either position A or B will lead to a tie. Based on this, according to the minimax algorithm, either of these moves would be given an equal score of 0.5 (representing a guaranteed minimum outcome of a draw). Against an optimal opponent this is a valid approach, as these moves are guaranteed to lead to the same result. However, against a suboptimal opponent there is a non-trivial difference between these two moves: if player 1 places a piece at A, and player 2 places anywhere apart from the centre space on the next turn, player 1 can complete a line through the middle and win. In comparison if player 1 places a piece at B, there is no possibility of a win for player 1. The difference between these moves (ie, a move that guarantees a draw but also has a possibility of winning, vs a move that guarantees a draw with no possibility of winning) is something that minimax fundamentally cannot measure. However, against a suboptimal opponent, a policy with the ability to take advantage of situations like this one could lead to a significant performance increase. This is where a simulation approach is necessary to further evaluate Q-learning policies.

To allow for further simulation-based evaluation of Q-learning policies, a dedicated environment was implemented in `matchup.cpp`. This program allows for simulation of a large batch of games between two selected opponents. This was implemented as a command-line based program. When executed, the user is prompted to input a player "type" for each player. The options are to input a `.dat` file containing a Q-learning policy, a `minimax` player, or one of three suboptimal players (`random`, `buggy`, or `buggy2`) which will be introduced in a later section. The user is then asked to specify the number of games to simulate.

Due to the number of different opponent types in use in this program, many whose `getMove()` or equivalent function existed in a different class, some abstraction was used in the form of function pointers. The declaration of these is shown in Listing 3:

```
Move (*p1MoveFn)(TicTacToe&, int) = nullptr;
Move (*p2MoveFn)(TicTacToe&, int) = nullptr;
```

Listing 3: Function pointer declarations in `matchup.cpp`

These pointers are assigned in runtime when the agent is selected. This allowed for a significantly less complex game loop and allows for easy expansion of this program to accommodate new player types if needed.

Once opponents and the number of games to simulate are selected, the program begins the simulation, recording both the results of each game, and the terminal board state of each game. The number of wins for player 1, draws, and wins for player 2, as well as number of unique terminal board states are output to the terminal and saved to a `.txt` file.

5 Further Notes on Implementation

This section details the solutions found to further problems encountered during the development of this project. It will discuss the reasoning behind the hyperparameters used in Q-learning training, the process of creating and testing suitable suboptimal opponents for the purposes of this project, and the decision to implement all components from scratch instead of using existing libraries.

5.1 Hyperparameter Selection

Tuning the training hyperparameters of a reinforcement learning agent can have a significant effect on the performance and learning speed of that model (Sutton and Barto, 2018). However, in this project, the primary goal was not to optimise an agent's performance in a specific setup, but instead to investigate the effects of changing the agent's environment (specifically through changing its opponent) on its performance. Because of this, the decision was made early in the project to avoid extensive hyperparameter tuning which could distract from this goal. A set of reasonable default hyperparameter values were chosen to be used consistently throughout the project. This avoids the risk of making it unclear which effects are a result of changing hyperparameters and which are a direct result of opponent modelling.

The selected hyperparameters were: learning rate $\alpha = 0.1$, discount factor $\gamma = 1.0$, and exploration rate $\epsilon = 0.2$. These values were chosen as they are used as common defaults in the literature and performed well across different environments in testing. A learning rate of 0.1 provides a balance between stability of existing values and ability to adapt to new information. The discount factor γ represents the rate at which future rewards are discounted compared to an immediate reward. In Tic Tac Toe, since a piece is placed every turn and there are 9 spaces on the board, there is a strict maximum game length of 9 turns. This means there is no risk of an infinite game loop or excessively long episodes, which the discount factor would normally be used to address. Due to this, there is no reason to incentivise shorter games, a discount factor $\gamma = 1.0$ (ie, future rewards and immediate rewards are assigned the exact same value) is suitable. An exploration rate of $\epsilon = 0.2$ allowed for effective exploration of the state space without massively increasing training times.

5.2 Selection of Sub-Optimal Opponents

One of the primary goals of this project was to explore the possibility of developing a reinforcement learning agent that could outperform a minimax agent under the right conditions, when playing against a suboptimal opponent. For this, developing a suboptimal opponent with the right characteristics became a crucial part of this project. Since it was not immediately clear what characteristics were needed in an opponent to demonstrate this outcome effectively, the design of suboptimal opponents was an iterative process.

Two types of suboptimal opponents were implemented in the first iteration. These were a seeded random player and a "buggy" minimax player. The random player selects a move from the available valid moves. This is selected uniformly, ie, all moves have the same probability of being selected. A set seed was used for the randomiser selecting these moves. This meant it would act deterministically; when presented with an identical board state, it would always choose the same action. The rationale behind the choice to use a set seed here, and make the random opponent act deterministically, was that a Q-learning agent will learn faster and more effectively (ie, converge more quickly) when faced with a consistent opponent.

The "buggy" minimax player makes use of the earlier minimax implementation, and acts optimally according to minimax in most situations. However, it is hardcoded with one exception which is illustrated in Figure 5. The buggy minimax player in this scenario is player 2. When its opponent, player 1, has placed pieces in the top right and bottom right spaces, and the middle right space between them is empty, the condition for it to act suboptimally will trigger. The optimal move in this case would be to play in the middle right space, as it prevents the opponent from winning immediately. However, the buggy minimax will first attempt to place in the centre space if it is free, then attempt to place in the bottom left if it is free, and finally place a piece at random if neither is successful. This means, if the reinforcement learning agent can lead the game to a position which triggers this condition, it can win on the next turn.

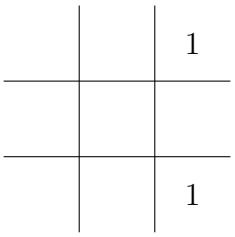

Figure 5: Illustrating the conditions required for suboptimal opponent `buggy` to play suboptimally

Experimentation with this first iteration of suboptimal opponents showed that these opponents were too easily exploitable. Even after short training periods, Q-learning agents were able to generate policies that consistently beat these opponents in simulation, achieving a 100% win rate. This does demonstrate a clear advantage of reinforcement learning for this application, as minimax achieved a significantly lower win rate against both opponents, but these opponents were very well-tuned to the strengths of Q-learning, and this did not give much insights into the limits of a reinforcement learning approach. The policies produced in these tests also showed very poor generalisation, since a policy that could guarantee a win was found early in training, and no further exploration was needed.

To address this, a new random player was implemented that did not have a fixed seed. This opponent still plays suboptimally in most cases but it will act non-deterministically; for any given board state, it can pick any valid action with no consistent behaviour. This provides a more challenging environment for Q-learning: instead of learning to exploit consistent behaviour, this opponent has to be exploited by taking actions that maximise the possibility of it making a suboptimal play which allows the Q-learning agent to win (an example of this is discussed in Figure 4 in Section 4.4). As well as providing a more challenging environment for evaluation of the capabilities of Q-learning, it also gives an opportunity for the Q-learning agent to learn a policy with better generalization, since a non-seeded random opponent will lead to a huge number of different states being seen in training.

An updated version of the buggy minimax opponent was implemented, "buggy2". This works similarly to the first iteration: in most situations, it will call functions from the standard minimax opponent and act optimally, however its hardcoded with a condition where it will act suboptimally. The difference in this iteration are the conditions required, which are illustrated in Figure 6. There are two key changes from the original implementation.

First, the condition requires not only player 1's pieces to be in the top right and bottom right spaces, and the middle right space to be empty, but it requires the buggy2 agent's own piece to be placed in the centre space. This significantly increases the difficulty of reaching this position, since the buggy2 agent acts non-deterministically in most situations due to minimax's random selection of a move from the set of optimal ones. For example, in a position where the centre space is optimal, but there are 4 different moves in the "equivalence set" of optimal moves according to minimax, there is only a probability of 0.25 that the buggy2 agent will actually place a piece in that position.

To exploit this flaw and achieve a greater win rate, the Q-learning agent would have to either lead the game to a position where the centre space is the only optimal space for the buggy2 agent (and even then, reaching this board state would still be probabilistic), or play the game in a way that gives the buggy2 agent multiple chances to place in that centre square throughout the game. All of this must be done while avoiding losing the game. This will provide a more meaningful challenge than the first iteration, to test the Q-learning agent's ability to explore the state space and discover exploitable behaviours in the opponent.

Second, when the condition is met to trigger this behaviour, instead of attempting to place in a hardcoded suboptimal position, the buggy2 agent will randomly place a piece in one of the available spaces. The result of this is that even when the correct board state is reached, there is a significant probability that an optimal move will still be selected by random chance.

A Q-learning agent's ability to learn to exploit an opponent's weakness depends on the rewards available with specific actions. In the first iteration of this buggy player, triggering the flaw led to a guaranteed win, the highest possible reward available in this environment. In this final iteration, reaching the correct board state to trigger the flaw only offers a probabilistic reward, not a guaranteed win. This provides an extra challenge for the exploration ability of the Q-learning agent.

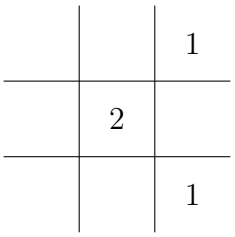

Figure 6: Illustrating the conditions required for suboptimal opponent `buggy2` to play suboptimally

In their final iterations, these opponents were used to test the capabilities of Q-learning to explore and exploit a weakness in a non-trivial environment, and to explore the effects of opponent selection on the generalization ability of the generated policy.

5.3 Decision To Implement From Scratch

Core components of this project, including the Tic Tac Toe environment, the minimax algorithm, and the Q-learning algorithm, were implemented from scratch rather than using existing libraries. This decision was made to provide full control over the behaviour over each component, to ensure optimal performance, and to provide better transparency (ie, to avoid using pre-made functions that operate like a "black box").

An example of the benefits of this is the minimax agent. In many board states, the minimax agent finds multiple "optimal" moves that provide an identical reward, and a method of selecting one of these moves needs to be implemented. In many use cases of the minimax algorithm, the goal is only to achieve optimal behaviour, and it is irrelevant which move from this set it selected. However, when the minimax agent is acting as part of the environment in training of a Q-learning agent, the process used in this selection can be the difference between creating a deterministic or non-deterministic environment, and fundamentally changes the behaviour of the Q-learning agent.

Performance was another concern motivating the choice to implement from scratch. Some aspects of this project involved large-scale simulations taking many hours. For example, in training a Q-learning agent against a random opponent, a training interval of 100 million episodes was used, leading to up to ≈ 900 million total game states simulated depending on game length. A small amount of overhead in the Tic Tac Toe implementation could have massively increased training times here. The minimax algorithm was the most computationally intensive component of this project, as every turn involved multiple searches of the entire game tree (one for each possible move). In both cases, implementing these components from scratch allowed for complete control over memory usage and ensuring performance was not slowed by any features which are not directly relevant to this project.

The choice to implement the Q-learning algorithm manually instead of using an existing library was motivated primarily by wanting to better understand the algorithm, instead of any performance benefits. Building this manually gave a clear understanding of the update rule, the process of exploration and updating the Q-table, the effects of hyperparameters, and other aspects of its functionality. Although a custom implementation had small benefits (eg, complete control over the state encoding formats and `.dat` file policy storage, and implementation of a timer to estimate the run-time of longer simulations), the main factor was learning the operation of this algorithm to help analysis of later results.

6 Results

This section presents the results of experiments evaluating the performance of Q-learning agents under a range of different environments. Different aspects of policy effectiveness are discussed including optimality according to the minimax algorithm, performance in simulated games, and generality.

For clarity, in this section a Q-learning policy will be referred to based on the opponent it was trained against, for example Q_{minimax} refers to a Q-learning policy trained against a minimax opponent.

Policy	Total States	States Matching Minimax	Agreement %
2,000 Episodes	219	188	85.84%
100,000 Episodes	521	498	95.58%
2,000,000 Episodes	569	552	97.01%

Table 1: Q-learning policies trained against a minimax opponent, data on agreement with the minimax algorithm

6.1 Evaluation of Q-Policy Trained Against Minimax

The Q-learning agent trained against a minimax opponent effectively learned to play optimally within that environment, and achieved a draw in every game against a minimax opponent in simulations. This is the best possible outcome against an optimal opponent such as minimax, demonstrating that under these conditions Q-learning is capable of producing an optimal policy for Tic Tac Toe. As shown in Table 1, both the number of states in the policy and the percentage agreement with the minimax algorithm increased as training time increased. After 2,000,000 episodes of training, the policy reached an agreement of 97.01%, showing a strong convergence towards the minimax strategy. Due to the diminishing returns in agreement as training length is increased it is possible that 100% agreement would never be achieved. It is worth noting that an agreement with minimax lower than 100% would suggest that the policy is capable of losing a game against minimax, but this did not occur even during a simulation of 200,000 games. A possible explanation for this is that the states which are not in agreement with minimax were only reached due to the Q-learning agent's suboptimal "exploration" behaviour in training, and these can not be reached in testing where exploration does not occur.

While the Q_{minimax} agent performed optimally in the environment it was trained in, it was not able to generalise effectively, and significantly underperformed compared to the minimax algorithm against other opponents. As shown in Table 2, against a random opponent in 200,000 simulated games, while minimax was able to win 193,622 and lose none, Q_{minimax} won only 163,249. Q_{minimax} also lost a significant (>10%) number of its games, showing that minimax's ability to always prevent a loss was not picked up in training. This result is not surprising given the policy contains only 569 states. When a Q-learning agent is faced with a board state it has not seen in training, it has no Q-values to direct its choice of action and chooses an action completely arbitrarily, performing no better than a random player. A Q-learning agent like Q_{minimax} which has only seen optimal moves from its opponent during training will therefore start to act as a random player as soon as its opponent makes a suboptimal move.

These results show that while training a Q-learning agent against a minimax opponent is effective for producing optimal behaviour within that environment, it is not an effective way to produce a more general policy for playing Tic Tac Toe optimally, and does not come close to matching or outperforming minimax once taken out of its training environment. The agent learns to replicate the decision making of minimax but fails to handle any suboptimal opponent. This also demonstrates a limitation of agreement with minimax as an evaluation metric, as high agreement in this case did not equate to comparable performance to the minimax algorithm.

Player 1	P1 Wins	Draws	P2 Wins	Player 2
Q_{minimax}	0	200,000	0	Minimax
Q_{minimax}	163,249	14547	22,204	Random
Minimax	193,622	6,378	0	Random
$Q_{\text{self-play}}$	174,959	3,105	21,936	Random
$Q_{\text{self-play}}$	0	0	200,000	Minimax
Q_{minimax}	200,000	0	0	$Q_{\text{self-play}}$
$Q_{\text{self-play}}$	200,000	0	0	$Q_{\text{self-play}}$
Random	116,857	25,442	57,701	Random

Table 2: Results from 200,000 simulations between Q-learning agents trained in different environments and standard opponents.

Policy	Total States	In Agreement With Minimax	Agreement %
Self-Play (Player 1)	4,433	3,072	69.30%
Self-Play (Player 2)	4,432	2,947	66.49%

Table 3: Q-policies generated through self-play, with agreement data compared to the minimax algorithm

6.2 Evaluation of Q-Policies Trained Through Self-Play

Training through self-play involves two Q-learning agents learning simultaneously, one acting as player 1 and one acting as player 2. Unlike training against a fixed opponent, exploration is possible from both sides of the game. Two separate policies are generated, one for each player. Each player's policy is constantly being updated to adapt to the opponent's strategy, which is also being constantly adapted. This gives a significantly different training challenge to a static opponent.

One notable result from the testing of self-play-generated policies is that the policy for player 1 consistently beats player 2 in a simulation. This same result, shown in Table 2, was observed in multiple different training scenarios, testing different hyperparameters and training intervals. After training, both policies contain Q-values that specify a single move to choose in each given board state. Because of this, both players act fully deterministically and simulating a game between these players will reproduce the exact same game every time, in this case a win for player 1. There is an advantage to the player 1 position in Tic Tac Toe (as shown in the Random vs Random simulation in Table 2, player 1 achieves around 58.5% win rate compared to around 29% for player 2), however with optimal play player 2 should still have the ability to force a draw, and as studies discussed in Section 2 have found, training through self-play is able to lead to optimal play, though this was not tested in Tic Tac Toe. It is not clear in this instance if the inability of the self-play agents to develop an optimal policy is due to a property of Tic Tac Toe as an environment, or due to insufficient hyperparameter tuning, though attempts were made to address this through hyperparameter tuning without success.

As shown in Table 3, the policies generated through self-play observed a large number of states; each of these policies individually is significantly larger than any other policy generated during this project. This is expected as the exploration from both sides means there is no limit on what states can be seen. It is likely that this training environment would eventually lead to the agents seeing every reachable state possible in Tic Tac Toe, though the self-play policies used in these final results have already been trained for 100 million episodes. Despite this effective exploration, the agreement with the minimax algorithm stayed low: 69.30% for player 1 and 66.49% for player 2. This indicates that the policies were far from converging to an optimal strategy.

The effectiveness of self-play policies in this environment is clear when evaluating their performance through simulation, as shown in the results in Table 2. The self-play agents (player 1 and player 2 were both used in testing, the appropriate policy to the position that agent was in) were completely ineffective, losing every single game against both Q_{minimax} and the minimax agent itself. This is not a surprising result considering the agreement percentage of the policies: in the minimax algorithm, there are only 3 possible scores that can be assigned to an action, 1.0 for an action that will lead to a guaranteed win, 0.5 for a possible win or guaranteed draw, and 0.0 for a possible loss. A move that is not in agreement with minimax means it is at least a tier lower on this ranking system. In practice, the huge majority of actions possible in Tic Tac Toe are assigned a value of 0.5 by minimax, and this means a huge majority of those 30% of actions in the policy that do not agree with minimax are these 0.5-rated states where minimax could guarantee a draw but $Q_{\text{self-play}}$ underperforms and opens itself to a loss. When the opponent is an optimal player like minimax, a possible loss becomes a guaranteed loss, as the opportunity to maximise this outcome will always be taken. The simulation results indicate that enough of these states exist in the policy that they show up in every game against minimax opponents.

While the self-play policies performed very poorly against optimal opponents, the simulation results indicate that these policies have better generalization than Q_{minimax}. When evaluated against a random opponent, $Q_{\text{self-play}}$ achieved a better outcome than Q_{minimax}, with significantly more wins and less draws, and marginally less losses. This improvement can be attributed to the larger number of states in the policy, over 4400 states in $Q_{\text{self-play}}$ vs 569 in Q_{minimax}. Even though the states in the Q_{minimax} policy almost all contain an optimal action and $Q_{\text{self-play}}$ has been shown to contain a significant number of suboptimal

actions, these actions may still be better than the large number of unseen states where Q_{minimax} will choose arbitrarily. This result demonstrates the importance of opponent modelling for creating a policy that can generalize effectively.

6.3 Evaluation Against Sub-Optimal Opponents

Policy	Total States	In Agreement With Minimax	Agreement %
Q_{random}	2,423	2,422	99.96%
Q_{buggy}	554	517	93.32%
Q_{buggy2}	596	575	96.48%

Table 4: Agreement of Q-learning policies trained against suboptimal opponents with the minimax algorithm

6.3.1 Random Opponent

To further test the effects of the opponent selection on a Q-learning policy, and the potential for a Q-learning agent to outperform the minimax algorithm, a suboptimal random opponent was implemented. Initially this used a fixed seed in every game, this selected a move uniformly at random from the possible moves, but it behaved fully deterministically, ie, it would always select the same move when presented with the same board state. This consistency is an ideal environment for Q-learning to exploit. In training, the Q-learning agent quickly learned this opponent's behaviour and found a sequence of moves that would guarantee a win, and the deterministic behaviour of the opponent meant it would never do anything to interfere with this strategy. In practice, after very short training intervals of $< 5,000$ intervals, the Q-learning agent was able to achieve a 100% win rate in this environment. This already displayed an improvement over minimax, which could not guarantee a win in this situation, but this environment felt too well-suited to Q-learning's strengths, and a more challenging opponent was introduced to further test this. The policy generated through this testing also displayed extremely weak generalization, since this environment incentivised finding a winning strategy and sticking to it.

A random opponent without a fixed seed was implemented to create a more challenging training environment, and one that could potentially lead to a more well-generalized and useful output. This opponent still selected moves uniformly at random however it was non-deterministic, there was no consistent behaviour that the Q-learning agent could learn, it would instead have to learn to take actions that would maximise the chances of the opponent's suboptimal play leading to a possibility of a win (Figure 4 in Section 4.4 explains this behaviour with an example - all "optimal" moves according to minimax are equal against an optimal opponent, but some provide a greater chance for a suboptimal opponent to make a mistake that loses them the game). This setup promotes a wide range of state spaces being seen in training.

After training against the unseeded random opponent, the Q-learning agent Q_{random} demonstrated a significant improvement over the minimax algorithm against a common opponent in simulation. As shown in Table 5, Q_{random} won 197,932 out of 200,000 games (98.97%) compared to 193,622 wins (96.81%) for the minimax agent. Importantly, neither agent lost any games in this scenario, against any opponent, demonstrating that the Q_{random} agent is able to uphold the minimax algorithm's guarantee of avoiding a loss, while still achieving a higher win rate. A z-test comparing these win rates gives a z-score of 47.40, indicating very strong statistical significance, with a p-value of $p < 10^{-16}$.

This result represents one of the primary goals of this project: training a Q-learning agent capable of both outperforming minimax in a situation where the opponent plays suboptimally, while still upholding the minimax algorithm's properties of guaranteeing a win, and guaranteeing optimal play against an optimal opponent. As shown in the simulation results, and confirmed by further testing against other agents, compared to minimax this policy performed *as well or better against all opponents*. The agreement percentage of this policy, shown in Table 4, backs up this finding, with only a single state out of 2,423 total where this policy does not take an action which is optimal according to minimax. As mentioned earlier, a possible explanation for the existence of a suboptimal state while simulation indicates that the policy always plays optimally, is that the suboptimal state was reached through the exploration policy in training, and can not be reached in testing while there is no exploration taking place. The large state space found in this policy

also indicates the model should generalize well, and this was confirmed by the simulations where the policy demonstrated optimal play across many different opponents.

The results of these simulations and policy analysis are strong evidence that this is a valid approach to achieve the goal of creating a policy that outperforms minimax. They also reinforce the importance of opponent modelling for achieving good performance with Q-learning, as the Q-learning agent itself and hyperparameters were not changed throughout any test results shown here, but effectiveness of the final model varies massively.

Player 1	P1 Wins	Draws	P2 Wins	Player 2
Q_{random}	197,932	2,068	0	Random
Minimax	193,622	6,378	0	Random
Q_{random}	0	200,000	0	Minimax
Minimax	0	200,000	0	Minimax
Q_{random}	200,000	0	0	$Q_{\text{self-play}}$
Minimax	200,000	0	0	$Q_{\text{self-play}}$

Table 5: Results from 200,000 simulations involving Q_{random} and standard opponents

6.3.2 Buggy Minimax Opponent

To evaluate whether a Q-learning agent could outperform minimax against suboptimal opponents with more consistent flaws, a 'buggy' version of the minimax algorithm was introduced. This opponent behaves like a standard minimax player in most scenarios, but with a hard-coded flaw that makes it act sub-optimally in certain conditions (the exact conditions and reasoning behind them are discussed in Section 5.2). The first iteration, similar to the first iteration of the "random" opponent, proved to be very well suited to Q-learning's strengths and after training intervals of $< 5,000$ episodes, it was able to achieve a win rate of 100%. This significantly outperformed the minimax algorithm, which was only able to achieve a win rate of 13.33%. These simulation results are shown in Table 6.

For a more challenging environment, a new model Q_{buggy2} was trained against an updated opponent, `buggy2`. The condition required to trigger this opponent's suboptimal behaviour depended on both player's pieces being in the correct spaces, instead of just the Q-learning agent's own pieces as in the first iteration. This made triggering the condition significantly harder, as it was impossible to "force" the non-deterministic opponent to place in a specific location. Additionally, the payoff (of a suboptimal play) for reaching this position was only probabilistic, not guaranteed as in the first iteration. These updates prevented Q_{buggy2} from achieving a 100% win rate, but it still significantly outperformed minimax with a win rate of 74.99% compared to 10.06%.

The policies Q_{buggy} and Q_{buggy2} generated here had a similar number of state spaces and agreement with minimax to the Q_{minimax} policy, as shown in Table 4. The generalization ability of these policies was fairly weak, similar to Q_{minimax}. The similarity here is expected given the opponents act identically to minimax given most board states. An interesting note about these results, is that even though the Q_{buggy2} policy had a greater number of states and higher agreement compared to Q_{buggy}, it demonstrated slightly worse generalization when simulated against a random opponent. This is the opposite of what the results in Table 4 would suggest. A possible explanation for this is that the strategies needed to trigger the more specific condition of the `buggy2` opponent are less effective at giving a random opponent opportunities to make a game-losing mistake.

The simulation results of the Q_{buggy} and Q_{buggy2} again show that Q-learning can outperform minimax, though in this case only within the specific environments they were trained in, not achieving the generalization ability seen in Q_{random}.

Player 1	P1 Wins	Draws	P2 Wins	Player 2
Q_{buggy}	200,000	0	0	Buggy
Minimax	26,666	173,334	0	Buggy
Q_{buggy2}	149,980	50,020	0	Buggy2
Minimax	21,923	178,077	0	Buggy2
Q_{buggy}	156,343	20,269	23,388	Random
Q_{buggy2}	139,178	21,802	39,020	Random
Minimax	193,622	6,378	0	Random

Table 6: Simulation results for Q-learning policies trained against `buggy` and `buggy2` opponents, compared to minimax and random baselines.

6.4 Computational Efficiency

An important practical consideration when comparing minimax and Q-learning as an approach to a problem is their computational efficiency. Even in a simple game with a small state space such as Tic Tac Toe, there are significant differences in the runtime between these approaches.

The minimax algorithm evaluates all future move sequences from a given board state, resulting in a time complexity of $O(b^d)$, where b is the branching factor and d is the depth. In this case, the branching factor b is the number of moves possible from any state, and the depth d is the maximum length of a game. In Tic Tac Toe, both of these numbers decrease by 1 after each move is completed, so the total number of possible games is $9! = 362,880$, representing the number of game sequences minimax will have to search in the worst case. The tree search is performed at runtime, meaning every turn where minimax has to select a move will have high computation costs.

Q-learning in contrast, has a high computational cost only during training. Once a Q-table has been generated, selecting an action from a given state is performed with an average time complexity of $O(1)$ due to the implementation of Q-tables as a hash map with board states as indices. This means in practice move selection is near instant, and the time taken to select a move will not increase as the number of states in the policy is increased.

The difference between these approaches at runtime was very noticeable: running a batch of 200,000 simulations when one player was a minimax agent took 660 minutes, while the same 200,000 simulations when both players were Q-learning agents or random players took 0.05 minutes, in the region of $10,000\times$ faster. Due to the time complexities of these two approaches, the difference between these runtimes will grow exponentially larger if applied to a problem with a larger state space than Tic Tac Toe. The scalability of reinforcement learning has been shown in the literature, such as Silver et al. (2018) where reinforcement learning is used to approach games such as Chess and Go where minimax is infeasible.

This difference in computational efficiency also impacts the choice of an opponents for Q-learning: a minimax agent used as an opponent in training will have to perform these intensive searches in every episode of training, and significantly slow down the training time. In practice the difference in speed here was similar to the difference in speed in simulation, in the order of $10,000\times$ faster. This meant a Q-learning agent trained against a random opponent could easily be trained for hundreds of millions of episodes, while training a Q-learning agent against a minimax opponent for only hundreds of thousands of episodes could take many hours. The training length, as shown in Table 1, significantly affects both the number of states explored (and therefore the ability to generalize), and the optimality of the trained Q-learning agent.

In practice this difference in computational efficiency at runtime and ability to scale to larger problems makes Q-learning better suited to a much wider range of problems. In applications where fast decision-making is important, such as robotics or real-time AI opponents in games, this makes Q-learning a clear choice over search-based approaches such as minimax.

7 Conclusions and Future Work

This project aimed to answer the question of whether a Q-learning agent trained under the right conditions can match the draw guarantee of a minimax search in *Tic Tac Toe*, and given a suboptimal opponent,

even improve past that benchmark. All components including the game environment, Q-learning algorithm and minimax search algorithm were written from scratch to ensure full control over their behaviour. The experiments consisted of three main training strategies: training against a full minimax agent, training through self-play, and training against intentionally handicapped suboptimal opponents.

The results show that training against a minimax opponent produces a policy Q_{minimax} which after 2,000,000 episodes of training matches the performance of minimax in almost all (97.01%) seen states. In practice this allowed the agent to force a draw against a non-deterministic minimax opponent in simulation every time, indicating fully optimal strategy within its explored state space. However, outside of that environment, for example against an opponent which plays at random, Q_{minimax} generalizes poorly and significantly underperforms compared to minimax.

Training against an unseeded random opponent gave very different results. The Q-learning agent Q_{random} explored over 2,400 game states during training, and wins 197,932 out of 200,000 games (98.97%) against the same random player. This win rate is higher than that of minimax, which achieved 193,622 (96.81%), and a two-proportion z-test ($z = 47.4$, $p < 10^{-16}$) confirms this is a significant result and not a sampling issue. The Q_{random} policy achieves this increased win rate *while also producing an optimal policy.* The policy produced here had a near perfect agreement with minimax, and through simulation it upheld the minimax algorithm's guarantee of a minimum outcome of a draw against any opponent, and performed *as well or better* than minimax in every test.

Self-play was not effective at producing an optimal policy. After 100 million episodes of training, the policy $Q_{\text{self-play}}$ for player 1 only had an agreement of minimax of 69.3%, and lost every game against a minimax opponent in simulation. The policies generated during self-play seem to indicate that the training has settled on a local maximum which has prevented finding an optimal policy, given the length of the training interval, and the literature indicating that self-play can find an optimal policy in the right conditions (Silver et al., 2018).

The difference in the computational efficiency of the reinforcement learning approach and a tree-search approach like minimax was evident in practice. In both training and simulation, involving a minimax in the game environment increased episode times by roughly a factor of $10,000\times$. Given the state space of Tic Tac Toe is so small it should be an ideal environment for minimax, but it still comes with a huge increase in computational cost compared to lookup in a Q-table. This makes it clear why minimax is infeasible in larger games, and why reinforcement learning would be the more attractive choice.

A follow up from this project would be to explore the possibility of using similar training methods to those used to generate Q_{random} in larger games. In Tic Tac Toe, this policy showed very promising results. However, given the combinatorial explosion that occurs as the size of a game increases, it is possible that the state exploration caused by the random opponent would be prohibitively slow, and the training method would be ineffective compared to a more focused approach. Further experimentation into the training conditions of self-play to examine why training here did not lead to an optimal policy is another worthwhile follow up.

References

He, H., Boyd-Graber, J., Kwok, K., and Daumé III, H. (2016). Opponent modeling in deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48. JMLR: Workshop and Conference Proceedings.

Hussain, B. (2023). Comparing the performance of q-learning and minimax in tic tac toe. *ResearchGate Preprint*. Available at: `https://www.researchgate.net/publication/371206019`.

Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326.

Korkmaz, E. (2024). Analyzing generalization in deep reinforcement learning. *arXiv preprint arXiv:2401.02349*.

Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.