Bachelor Thesis

Board Game AI Using Reinforcement Learning

Linus Strömberg, Viktor Lind
Computer Science

Studies from the School of Science and Technology at Örebro University
Örebro 2022

Studies from the School of Science and Technology at Örebro University

Linus Strömberg, Viktor Lind

Board Game AI Using Reinforcement Learning

Supervisors: **Jennifer Renoux, Örebro University**
Ilkin Huseynli, Piktiv AB
Examiner: **Johannes Andreas Stork, Örebro University**

© Linus Strömberg, Viktor Lind , 2022

Acknowledgement

We would like to thank Piktiv AB for the opportunity and thesis idea, as well as our supervisor Ilkin Huseynli for the support. A special thanks to Simon Johansson at Piktiv AB for several valuable suggestions within the field of Machine Learning.

We would also like to thank our supervisor Jennifer Renoux at Örebro University for the positive encouragement and guidance through this project. Without her support this thesis would not have been possible.

Abstract

The purpose of this thesis is to develop an agent that learns to play an interpretation of the popular game Ticket To Ride. This project is done in collaboration with Piktiv AB.

This thesis presents how an agent based on the Double Deep Q-network algorithm learns to play a version of Ticket To Ride using self-play. This is the documentation of how the game and the agent was developed, as well as how the agent was evaluated.

Keywords

Reinforcement Learning, Game Development, Computer Engineering, Q-learning

Contents

Chapter 1
Introduction

Many multiplayer games today involve artificial players. This can enhance the game play experience or it can enable people to play games that normally would have not been possible without other people. This thesis documents an implementation of an artificial player learning to play a version of the popular board game Ticket To Ride.

1.1 Problem Formulation

The purpose of this thesis comes from an idea from the company Piktiv AB. They see opportunities in validating reinforcement learning in discrete board games. The exact wording from Piktiv AB was:

"Validate reinforcement learning for discrete board games by modelling a moderately complex board game and build an AI agent that learns to play the game through self-play"

This task will be attempted by implementing a version of Ticket To Ride with an artificial player using Q-learning based methods to learn the game.

1.1.1 Project Goals

The thesis idea is going to be split into several sub-goals in order to realize the project. The project starts with the design and implementation of the game Ticket To Ride (TTR) as a playable video game, then an agent is going to be implemented. Then the work of interconnecting the agent and the game starts. When this is done the agent needs to learn to play the game. The whole process and design decisions are going to be documented through the course of the thesis.

1. Create a discrete board game as a video game.

   Since we do not have any previous experience with game development, and particularly no previous experience in developing artificial players playing games, the game has first been developed with an interface for people to play it. The game has later been modified so that artificial players can play it.

2. Create a Reinforcement Learning Agent (RLA)

   We first implemented an artificial player that takes random actions in the game. In parallel with this the game was modified so the agent could play the game. Then a new agent was developed and trained using Reinforcement Learning and self-play since Piktiv AB required this.

3. Documentation

   The process of the research and development has been documented and presented as a thesis at the end of the project.

1.1.2 Evaluation

To evaluate the project some core components need to be achieved. A version of the game TTR needs to be in place, where different agents are going to be able to compete against each other. If there is an RLA in place at the end of this project, it is going to compete against another agent that does completely random actions. If the RLA wins a majority of the games against a player taking random actions, this is to be considered a success.

1.1.3 Delimitations

Due to the time and resource constraints of this project the following limitations have been set.

Ticket To Ride Implementation

The end goal is not for a human to have a great time playing this game, meaning that no time will be put into having a good user interface or graphical design. The goal is to implement a discrete board game, meaning that modifications of the game logic is allowed in order to complete the project within the given time frame.

Reinforcement Learning Agent

The end goal is not to have an agent that has achieved a level of super human game play, or even an agent that makes for a fun opponent to play against.

1.2 Division of Work

The thesis was co-written and pair-programmed equally by both authors.

1.3 Outline

The outline of this bachelor thesis is as follows:

Chapter 1 gives an introduction to the reader. It presents the problem formulation, how the work was distributed between the contributors, and this outline.

Chapter 2 gives the reader an introduction to the game TTR and some background on Reinforcement Learning. At the end of this chapter the reader will know the rules and content of the game, as well as the Reinforcement Learning topics relevant for this project.

Chapter 3 gives an overview of relevant previous work. It starts with Google DeepMinds leap in Reinforcement Learning with their Atari playing agent and their now famous AlphaGo agent, followed by an implementation of the game Risk. The chapter ends with previous work done on TTR.

Chapter 4 contains the description of the implementation of both TTR and the Reinforcement Learning Agent. The chapter will guide the reader through architectural decisions made as well as pseudo code. The chapter concludes with a explanation of how the agent was trained with accompanying data from training sessions.

Chapter 5 presents resulting match and performance data from the trained agents.

Chapter 6 addresses thoughts arisen during the interpretation of the results and from the research and development of this project. An analysis of the results accompanied with reflections are provided.

Chapter 2
Background

The purpose of this chapter is to introduce the reader to the board game TTR as well as giving an overview of Reinforcement Learning (RL) concepts. The game will be explained in such a manner that the reader will be able to easily distinguish differences in the rules of the original game and this projects implementation. The method used to implement the RLA builds on the Q-learning algorithm. In the end of this chapter the reader has been introduced to Reinforcement Learning and its concepts used in the agent implementation. The full implementation can be seen in chapter 4

2.1 Ticket To Ride

TTR is a railway-themed discrete board game designed and released in 2004, by Alan R. Moon. The game is recommended to be played by two to five players, and for players from the age 8 and up. The game is played on a board that represents a map of a region. The original version of the game represents the United States and parts of southern Canada. The map has 35 cities that are connected by railroads, where the railroads are of different lengths. The goal is to collect score by claiming railroads [1].

2.1.1 Game Contents

The contents of the game box consists of a variation of components depending on what version is used. In the original game box the below content can be found.

- A board map representing cities from the United States of America, and parts of southern Canada.
- 240 Plastic Train Tokens distributed evenly between 5 different colors.
- 110 Train Cards distributed over 8 different colored Train Cards, and the Locomotive card. 12 for each and every color and 14 locomotive cards.
- 40 Regular Distance and 6 Far Distance Destination Tickets Cards.
- 1 Express Bonus card used to give a bonus to the player that has the longest continuous railroad.
- 15 Train Station Tokens.
- 5 Score Markers.

2.1.2 Game Setup

The game board is placed on a flat surface and the card piles are shuffled. 45 colored Plastic Trains and 4 Train Cards are being dealt to each player. The players draw the five top cards from the Train Cards pile and place them adjacent to each other and so that they all are facing upwards from the table. The remaining Train Cards form one pile facing down. These are the two different Train Card piles that the player can draw from during the game.

Each player receives three Destination Ticket Cards and must choose to keep one, two or three of the ones being dealt. If a player chooses two cards, the third is put back to the Destination Ticket pile. This card type is not being discarded during game play. Whether a Destination is completed or not, the card needs to be kept for the purpose of counting scores in the end.

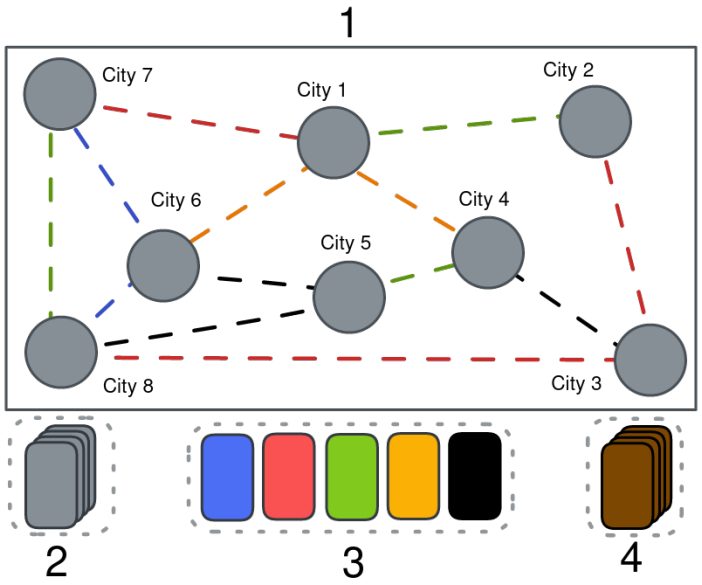


Figure 2.1: An image representing contents of *TTR*. Object 1 is the board, object 2 are the face down train cards, object 3 are the face up train cards and object 4 are the destination tickets.

2.1.3 Object of the Game

The goal of the game is to claim railroads that are interconnecting cities of the map. A player can receive a score at three different situations as well as losing score at one situation. The player with the highest score in the end of the game wins.

The players can take one of three actions each turn. The player chooses between drawing two types of cards - the Train Cards and the Destination Ticket cards, or to claim a railroad. The Train Cards are used for claiming railroads and the Destination Tickets are objectives that the player can complete.

2.1.4 Actions

Drawing Train Cards

The players draw Train Cards of 8 different colors from two different draw piles. One pile consists of 5 cards facing upwards so that the player can see what these cards are, and one pile facing down. The player can draw two cards from any pile under certain conditions. A Locomotive card represents every color in the game and is considered a trump card. If the player draws the Locomotive card present in the face up pile in the first part of the two actions of the turn, the turn is ended as seen in figure 2.2.

If the player chooses to draw from either the face down pile, or a card from the face up pile that isn't a Locomotive in the first draw action, the present Locomotives of the face up pile is considered an illegal action in the next draw action, where the player now can choose from the face down pile or from any other card that is not a Locomotive from the face up pile. For the sake of clarity two scenarios of this type of behaviour have been visualized in figure 2.3 and 2.4.

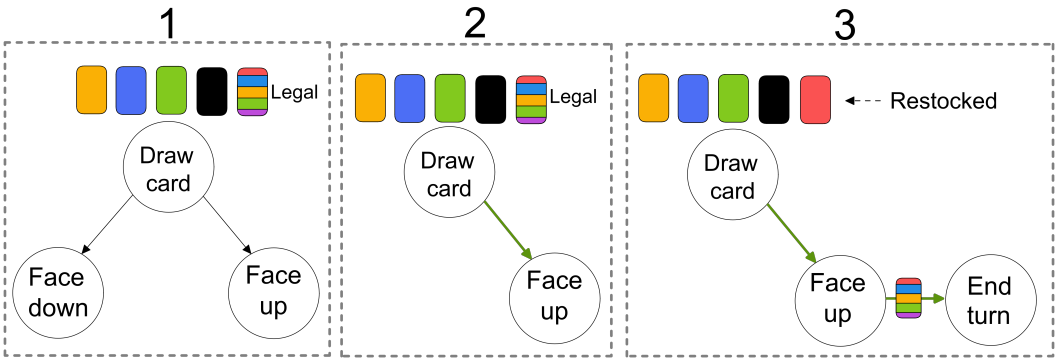


Figure 2.2: A three step visualization showing a decision tree of drawing a Locomotive in the face up pile.

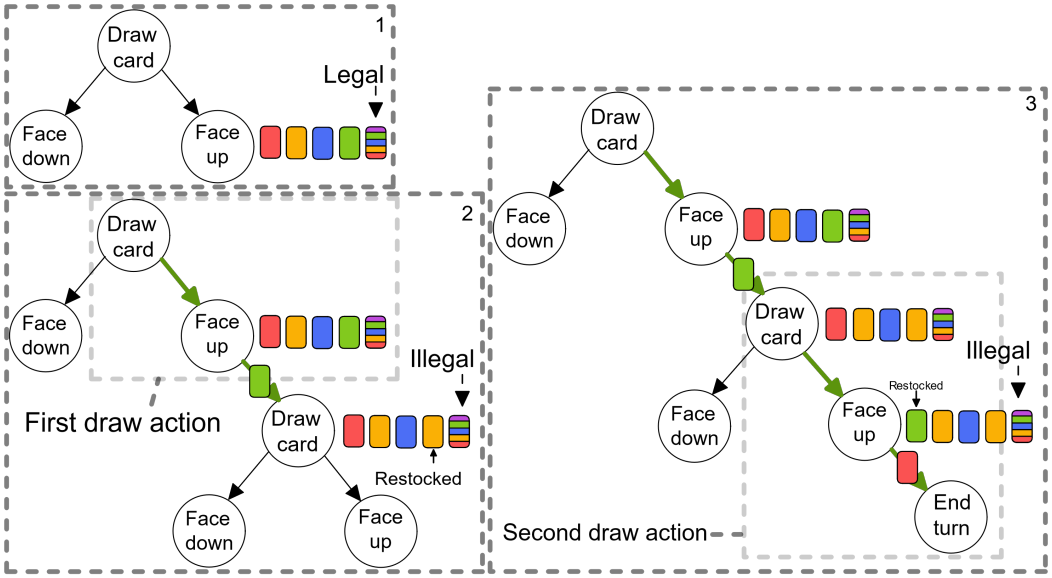


Figure 2.3: A three step visualization showing a decision tree of a player drawing two non-Locomotive cards, both from the face up pile.

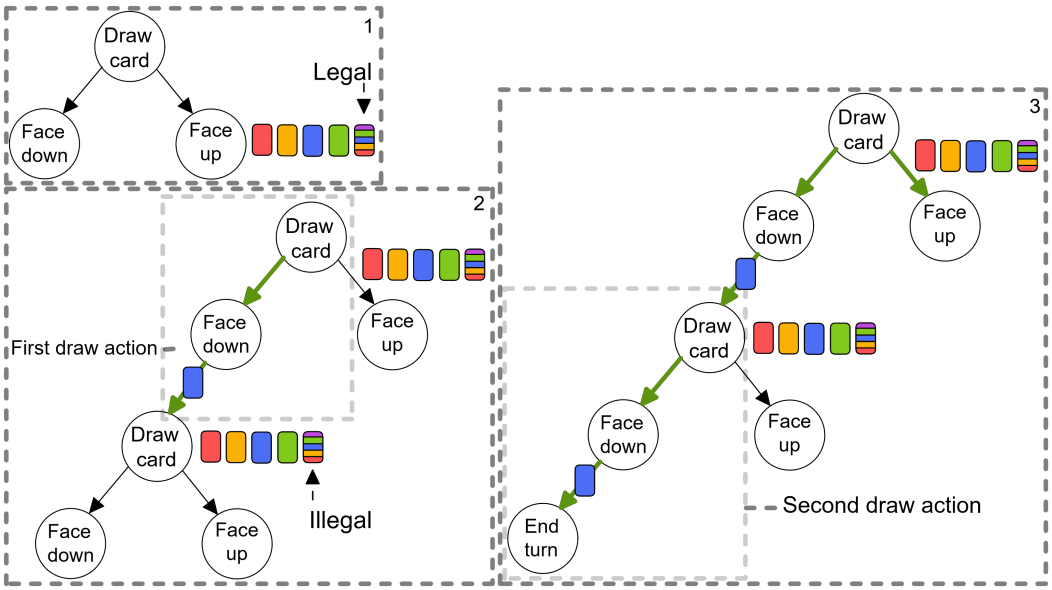


Figure 2.4: A three step visualization showing a decision tree of a player drawing two non-Locomotive cards, both from the face down pile.

Drawing Destination Tickets

When a player chooses to draw from the destination ticket pile, the player takes the top three cards, looks at them and then decides to keep one, two or all three of them. This results in up to seven different choices each time the player wants to draw from that pile as shown in the below figure.

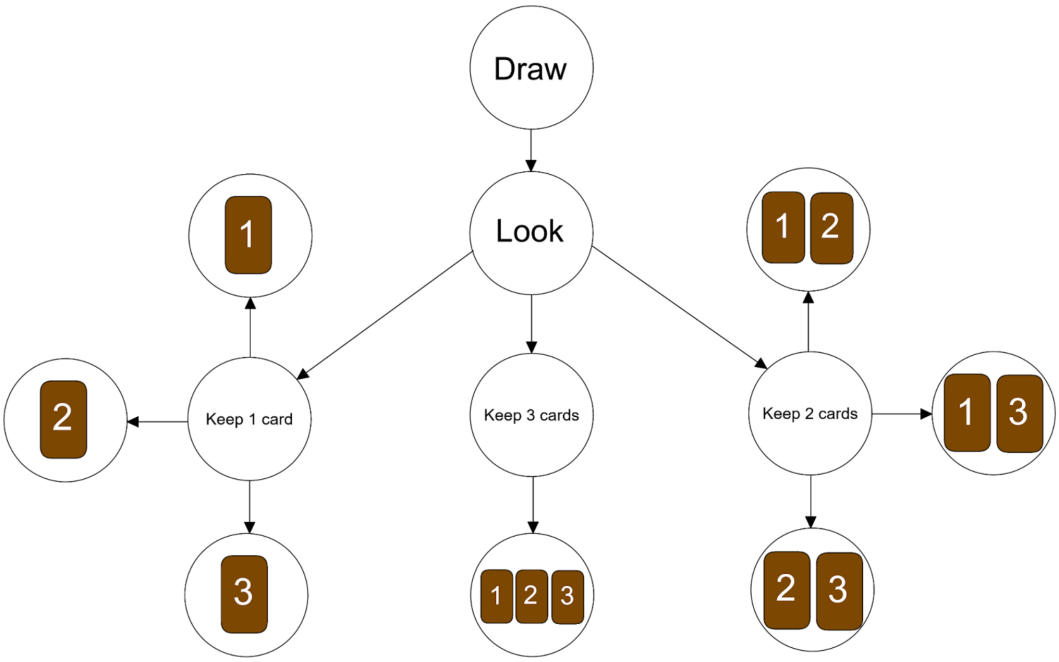


Figure 2.5: A decision tree of the event of picking destination tickets.

Claiming railroads

All railroads in the game have a length and a color. To claim a specific railroad the player needs to spend an amount of colored Train Cards corresponding to the railroads length and color. Locomotives and Train Cards can be combined in order to meet the railroads requirement. A railroad can only be owned by one player, and it can not be claimed from another player at a later stage.

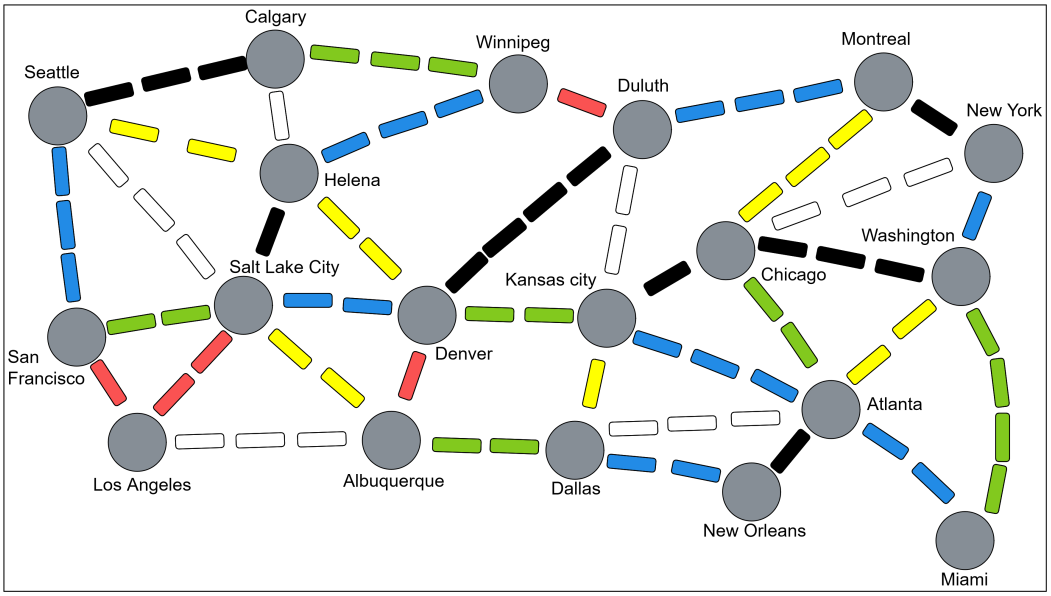


Figure 2.6: The game board used in the project implementation.

2.1.5 Scoring and Game end

Claiming Railroads

The player receives a score corresponding to the length of the railroad that is in the players possession.

Completing Destination Tickets

Destination tickets act as an objective and display the name of two cities and a score. The objective is for the player to connect the two cities by claiming railroads. The distance between the cities is always so that the player needs to claim several railroads in order to connect the cities By completing the destination ticket the players score increases by the amount of points that is given by the card. If the player fails with the objective of connecting the cities the players score is deducted by the amount of points written on the card, i.e a penalty is given. The score given by destination tickets are calculated at the end of the game.

Building the Longest Continuous Railroad

The player that has built the longest continuous railroad is rewarded with 10 points. The railroad is allowed to contain self loops and pass through the same city several times but is not allowed to use the same railway more then once.

Ending the Game

When a player has claimed enough railroads to only have three or less plastic Trains left in its inventory, the final round is triggered. This means that each and every player, including the player who triggered the final round, gets to play a final round. The final score is being calculated after all players have taken their turn.

2.2 Reinforcement Learning

The agent in this thesis is based in the Machine Learning (ML) paradigm Reinforcement Learning (RL).

RL is based on practices where intelligent *agents* learn to navigate *environments* by executing possible *actions*. Prior knowledge of the domain or environment is not necessary as the agent builds its understanding of the environment through trial-and-error [2]. When an agent interacts with an environment it sends an action to the environment, where a *feedback signal*, also known as a *reward*, is returned to the agent alongside an *observation* of how the *state* of the environment was changed.

The agent strives to maximize the total amount of reward during the task, whether it is playing a video game or any other problem suited. For instance, if a RL agent plays a board game, it strives to win the game. Figure 2.7 presents an image of what is called the *interaction loop*, which is a closed loop of interactions between an agent and an environment that continues until a task is completed.

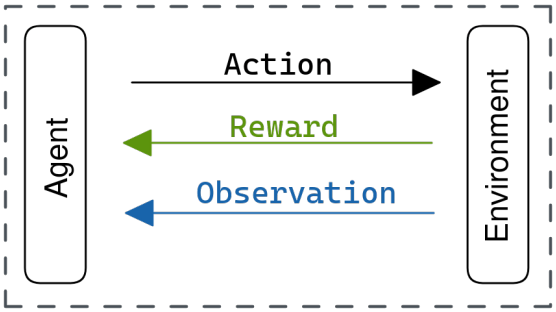


Figure 2.7: An image of the *interaction loop*.

The concept of RL in ML is not completely different from the concept of Positive Reinforcement, where both humans and animals can be motivated by being treated with rewards upon the completion of tasks [3, 4].

RL is based on the theories of Markov Decision Processs (MDPs) and Dynamic Programming (DP) [5].

2.2.1 Markov Decision Processes

A MDP is a way to mathematically represent a sequential decision-making system. A MDP can be defined as $\mathrm{MDP}(S, A, r, P, \gamma)$ where each attribute is defined as the following:

- S is a set of all the possible *states* an environment can have.
- A is a set of all possible *actions* that can be taken within the environment.
- r is the *reward* returned for each action taken.
- P is a *probability transition matrix*.
- γ is the discount factor.

An action $a \in A$ is a move or an activity that an agent can do to interact with an environment. If the environment represents TTR, an action is the agent drawing a card from a card pile or claiming a railroad.

The action and the accompanied reward r is remembered by the agent where they are later used to predict future states, actions and rewards [6].

For each time step t in a MDP, an environment is in some state $s_t \in S$, an agent takes some action $a_t \in A$. The action is based on P, where P is the probability that the environment will end up in a certain state $s_t \in S$, based on the taken action. After the action has been taken a reward r_t is received. The discount factor γ is an attribute used to make the agent prioritize immediate or later rewards.

At each time step t, the agent chooses a policy π, where the policy is a mapping between states and actions. It can be defined as $\pi_t(s_t) = a$, where the action a will be selected based on state s_t. The goal is for the agent to find the optimal policy in order to maximize the cumulative reward.

Ticket To Ride can be modeled based on the theories of MDPs, where at a certain time t, the current state of the game s_t contains all the necessary information (railways, card piles and cards on hand) for a player to decide on an action $a_t \in A$, where a reward (score if railway was claimed for example) r_t is given to the player.

2.2.2 Q-learning

Watkins introduced an RL algorithm named Q-learning [7]. Q-learning is a simple algorithm that enables agents to learn how to navigate optimally in a Markov Decision Process.

The Q-learning algorithm is known as an action-value function which maps values from rewards to actions, where the action-values are called Q-values. Actions mapped to high Q-values are considered being of high quality and should be governed by the agent. The Q-learning algorithm predicts how good certain actions are in given situations. These predictions improve over time.

Agents using Q-learning to solve Markov Decision Processes only remember the quality of certain actions made in certain states. This is done by saving and updating Q-values stored in a table called the Q-table.

The Q-learning algorithm calculates the new Q-value Q_{t+1}^{new} by adding the former Q-value Q_t^{old} with the learning rate α multiplied by the reward r_t received from the latest action taken added with the discounted value γ of an estimate of the highest future value $max_a(Q_t(s_{t+1}, a) - Q_t^{old}(s_t, a_t))$ as seen in equation 2.8.

$$Q_{t+1}^{new}(s_t, a_t) \leftarrow Q_t^{old}(s_t, a_t) + \alpha \times \left(r_t + \gamma \times max_a(Q_t(s_{t+1}, a) - Q_t^{old}(s_t, a_t)) \right)$$

Figure 2.8: The Q-learning algorithm

1. Gamma - γ

   Gamma is called the discount factor which determines the action selection of the agent. A value closer to 1 will make the agent cater for actions that yield later rewards, in contrast to having the value closer to 0, where the agent will value immediate rewards higher.

2. Alpha - α

   Alpha is called learning rate, and is a scalar that determines how much the Q-values are being incrementally updated. A lower value means it updates with a slower pace, where setting it to 0 would make the algorithm stop updating, hence it would stop learning.

2.2.3 Double Q-learning

In [8], it is explained that the Q-learning algorithm performs poorly in some environments due to Q-learning overestimating the Q-values. Hasselt introduced a method using two action-value functions, both the Q-learning algorithm, which together reduced the overestimation of the action-values that the original Q-learning algorithm suffered from. This method is called *Double Q-learning*.

2.2.4 Deep Q-Network

In [9] Q-learning was combined with Deep Learning [10], creating the algorithm Deep Q-Network (DQN). This approach introduces an Artificial Neural Network (ANN) (*sometimes referred as Neural Network (NN)*) to create an agent that is capable of learning policies directly from large amounts of sensory input. In this case it is an agent that learns to play several Atari games by feeding the agent all the pixels from the monitor screen.

A ANN [11] consists of several layers of cross-connected Artificial Neurons (ANs), where different calculations take place in different layers. A representation of a typical ANN can be seen below.

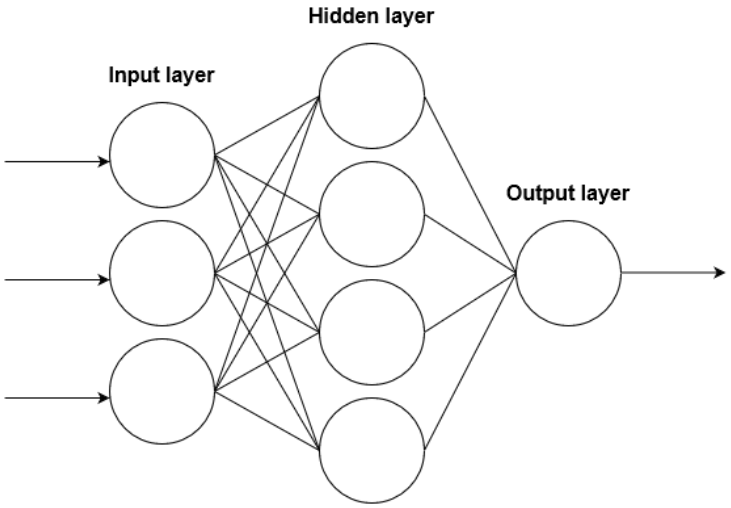


Figure 2.9: Architecture of a typical artificial neural network.

Here follows an explanation of each component of the ANN seen in figure 2.9.

- Artificial neuron - Processes data it receives as an input, either from being in the input layer receiving from the environment, or receiving from other neurons. The data processed is then outputted to another neuron or out from the network.
- Input layer - Takes input data from an environment.
- Hidden layer - Receives output data from a previous layer and processes it. A network can have any number of hidden layers.
- Output layer - Receives processed output from the previous layers and outputs it out of the neural network. This output data is the product of the neural network.

The addition of the ANN combined with Q-learning makes the DQN capable of learning policies from large inputs to solve complex problems using RL.

2.2.5 Double Deep Q-Network

In [12], the shortcomings of DQN was addressed were the authors showed that the DQN is overestimating the Q-values in some of the Atari games the agent played. An algorithm using Double Q-learning together with Deep Learning is introduced which reduces the overestimation observed in DQN. The new algorithm is called Double Deep Q-Network (DDQN) and uses two ANNs together alongside Double Q-learning.

One ANN estimates all action-values of all possible actions, where a second ANN evaluates what action is best to take based on the first ANNs predictions. This is a method of combining two DQNs to solve the problem in overestimation that one DQN suffers from. The agent in this thesis is based on a DDQN.

Chapter 3
Related Work

This chapter shows noticeable work that has been the basis used for inspiration of this thesis.

3.0.1 Atari games

In [9] a team at Google DeepMind took a great step in the RL field as they developed an agent called the *Deep Q-network agent* that learned to play 49 Atari games using RL. The agent is based on prior research in deep learning and Q-learning. The agent learned how to play the Atari games with performance on par with professional human players and with this outperforming all earlier attempts on playing games with RL using other algorithms [9].

3.0.2 The game of Go

Google DeepMind [13] made an even bigger step presenting the results of their AlphaGo agent. According to Google DeepMind, the game of Go has been considered the most challenging classic game for artificial intelligence. This is due to the humongous search space the game has.

With this agent DeepMind managed to defeat the European champion of the game Go as well as achieving a 99,8% win rate against other Go agents. DeepMind achieves this by combining a neural network that evaluates the positions of markers on the board with a neural network that evaluates moves. The training was done using a combination of supervised learning , RL and self-play.

Furthermore, in 2016 AlphaGo faced one of the best Go players of all time, *Lee Sedol* [14], where AlphaGo won 4 to 1. The event was later released as a documentary [15]. DeepMind did further work building upon AlphaGo, where they in [16] DeepMind showed that AlphaGo could learn to play Go *tabula rasa* [17], meaning it learned without any guidance and experience it got from the supervised learning.

DeepMinds idea of using RL and self-play to train an agent in order to play a board game is the base of this bachelor thesis.

3.0.3 Risk

Other work on board games using ML has been done in the war strategy game of Risk. In this game players compete with armies of plastic figures to attack each other with the goal to conquer regions of the map.

In Erik Blomqvists thesis *Playing the game of risk with an AlphaZero agent* [18] the goal is to use a zero learning algorithm like the AlphaZero or the EXIT-algorithm for training an agent to play the game of Risk by using self-play and to reach superhuman game-play level. Blomqvist designed an Agent that uses Monte Carlo Tree Search (MCTS) [18] for online decisions and is aided by a neural network which learns offline action policies and a state evaluation function.

In 2005 Michael Wolf [19] implemented the game of Risk with two different artificial players. One basic player that was choosing the highest rated action based on several different rated actions, and one enhanced player that used heuristic functions for selecting its actions.

3.0.4 Ticket To Ride

In the article [20] by Dinjian and Nguyen they tried to implement the game of TTR together with an agent to play the game, but did not manage to do so. Instead they create a good platform for further works where their report investigates topics that can be useful when attempting to develop TTR such as featurizing the environment, determining reward and curriculum learning [2].

In [21], Huchler implements a MCTS agent for TTR. Huchler calculated the state space complexity of the original TTR USA map for a two player game where the result was state-space complexity of $\sim 9.45 * 10^{54}$ different board configurations.

In the report [22], search algorithms were used to find bugs in the game of TTR. The authors of the report tried to develop a search based Artificial Intelligence (AI) that used A-Star and MCTS but were not able to. Instead they developed four different agents with heuristic functions for commonly used strategies in TTR.

Chapter 4
Implementation

This chapter will explain how we implemented the game TTR and how we approached the task of training the RLA. The following sections will show why certain design decisions and architectural choices were made during the realization of the parts of this project.

The recommended ML frameworks for this thesis was PyTorch [23] and Keras [24], where PyTorch builds upon Torch [25] and Keras builds upon TensorFlow [26]. The recommendations came from the supervisors of this thesis based on their experience with the frameworks. Ultimately the decision fell on Keras to be used when implementing the RL agent.

Due to this, the decision was made to implement the game environment in Python as well. Keeping the code base homogeneous can enable the project to have a higher chance of success given the time constraints where some immediate gains are no serialization of data structures, one code formatting standard and minimal implementation of the interaction between the environment and the agent.

4.1 Ticket To Ride

The initial implementation of the TTR environment was designed for human players and contained all game logic from the original game, as described in Chapter 2. The environment underwent several reworks during the course of the project.

Due to the constraints in time, choices were made to keep the time complexity of the RL Agent training phase low. To accomplish this a set of changes to the original game rules needed to happen in order to fit this requirements. Inspiration to these changes was found by investigating a simpler version of TTR, namely the junior version of the game, *Ticket To Ride First Journey* [27]. This game has a smaller board map, uses less card colors and has simpler game play rules.

The final version implemented is combined by parts from both the original game [1] and the junior version [27], as well as some custom implemented logic to further shrink the action space and the time complexity. The measures taken to adapt the final game do not affect the core gameplay of the game. Human players of this version will still recognize TTR. The following section will show what changes have been made.

4.1.1 State-Space Complexity

The same method has been used as Huchler used in *An MCTS agent for Ticket To Ride* [21] to calculate the State-Space Complexity (SPC). This shows how many different states the game board can have at the end of a game. The board state is the ownership status railroads have together with the Destination Ticket Cards are in possession of certain players at certain times.

Railroads

There are a total of 39 railroads in this implementation. This thesis has been based around a two player game and because of this, each railroad can be under three different ownership statuses - *not owned*, *owned by player 1* or *owned by player 2*. This gives the modified game an upper bound of 3^{39} railroad combinations that can be built [21].

Destination Tickets

There are two different Destination Ticket types in the game, the *far distance destination tickets* which are 5 cards in total, and the *short distance destination tickets* which are 12 cards in total. Each player receives one card of each type in the beginning of the game. The remaining far distance cards are discarded for the rest of the game, while the short destination cards form a new pile that is being used in the game.

Since there are two players that draw one card each from a pile of 5, and one card each from a pile of 12, the following equation shows how many different states the destination tickets can be distributed in between the draw piles and the players under the course of the game.

$$20 \times \left(\sum_{n=1}^{11} \binom{11}{n} \times \sum_{l=1}^{12-n} \binom{12-n}{l} \right) \tag{4.1}$$

Figure 4.1: The equation results in 6963020 different combinations of the destination tickets.

The number 20 in the equation represents the different combinations the far distance cards can be in, whereas the rest of the equation represents the ways the short distance cards can be in.

Total State-Space Complexity

The total state-space complexity is the number of railroad combinations 4.05×10^{18} and the destination ticket combinations 6963020. This calculation can be seen in the below equation.

$$4.05 \times 10^{18} \times 6963020 \approx 2.82 \times 10^{25} \tag{4.2}$$

2.82×10^{25} are the number of different board states that exist in this version of TTR. Huchler states in [21], that her calculation is an overestimation as it is not possible to build all railroads because the players do not have enough train tokens and there are more than one player.

4.2 Environment Architecture

The following subsections will give an overview of the game environment architecture.

4.2.1 Emulator

The emulator will run the main game loop, instantiate the environment, the agents, and the logic related to rules and scoring in the game. This is done in a monolithic manner where everything is run from one file.

4.2.2 Game

This class holds methods for the majority of the game logic such as score calculations when claiming railroads, completing destination tickets, check status of card piles and building the longest railroad. It will also give rewards (Section 2.2.1) to an RLA if one is in play. The class takes player objects (Section 4.3) and works with the internal data of those instances when calculating scores.

4.2.3 Environment

The class Environment will serve as an interface between the environment and the agents playing the game. A model of this interaction can be seen in figure 2.7. The Environment consists of its own data members and methods, but it is also inheriting from the classes Board and Card, which respectively contain methods to create the map and the cards needed to play the game. The three classes form the environment together. A visualisation of this can be seen in figure 4.2.

The environment needs to be represented in a way so that a NN [28] can use the environment as an input. This has been done by encoding the environment and storing it using ordinal encoding [29] (Section 4.2.6).

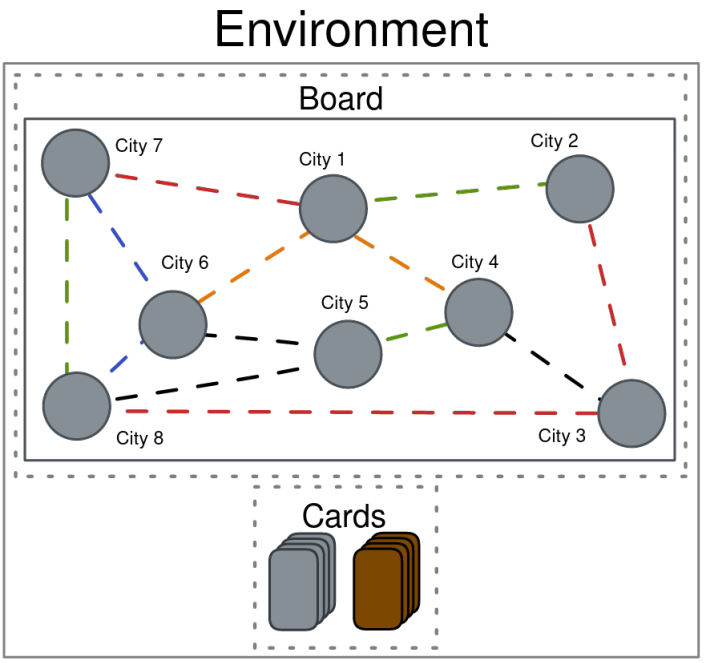


Figure 4.2: An image representing the Environment in the game implementation. As explained in (Section 4.2.3) it uses the contents of Board and Cards.

The content represented in the final game environment:

- A representation of the board map like TTR First Journey, with 18 cities and 39 railroads connecting the cities.
- 20 Plastic Trains distributed to each player in the game.
- A total of 72 Train Cards. 10 Train Cards in 6 different colors and 12 Locomotives cards.
- 12 Regular Distance and 5 Far Distance Destination Tickets Cards.
- 1 Express Bonus card used to give a bonus to the player that has the longest continuous railroad.

4.2.4 Board

This class holds the data structure that makes up the game board, also referred to as the *map*. The map of the game was realized with a graph theory framework called NetworkX [30]. NetworkX made it possible to easily create a graph that holds the information of cities and the railroads that connect them. The cities correspond to nodes and the railroads are edges in the graph.

The cities hold their own names and are necessary when creating the railroads. The data relevant to the game and the players are held in the railroads. The railroad attributes are what color it has, how long it is and who is owning it. These attributes are used to match what the players have in their hands when claiming railroads. The class includes a method that translates and encodes the NetworkX map. The encoded map is later used and interpreted by the RL agent. An elaborated explanation of this follows in (Section 4.2.6).

4.2.5 Cards

The class Cards holds data and methods related to card handling in the game. It contains the face up train pile, face down train pile, and the destination tickets. This class holds methods that create the card piles, deals cards and restocks empty piles. It also contains a method that encodes the cards similar to the method in the Board class using ordinal encoding. The next section will explain this further.

4.2.6 The Encoded Environment

The project plan (Section 1.1.1) included that the game was to be developed for humans to play initially. The purpose of this was to speed up the development process by having a one-to-one relationship in mind when implementing the rules and logic of TTR. This entailed the use of qualitative variables [31] when creating the attributes of the map. See figure 4.3.

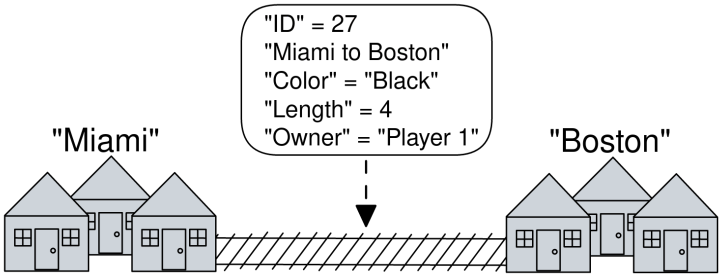


Figure 4.3: A visualization of the qualitative attributes of two cities and a railroad connecting them (Section 4.2.4).

The NN that is used in the RLA, later explained in (Section 4.3.4), can not parse the data from the map since it is represented with an Euler Graph using categorical attributes [31]. Because of this the entire environment including the map and the cards needs to be encoded using ordinal encoding.

Each railroad on the map has a unique ID. All IDs are extracted from the map and sent to the NN. The idea is that the RLA forms an understanding of which railroads can be claimed through trial-and-error, rather than reading the attributes of the railroad directly.

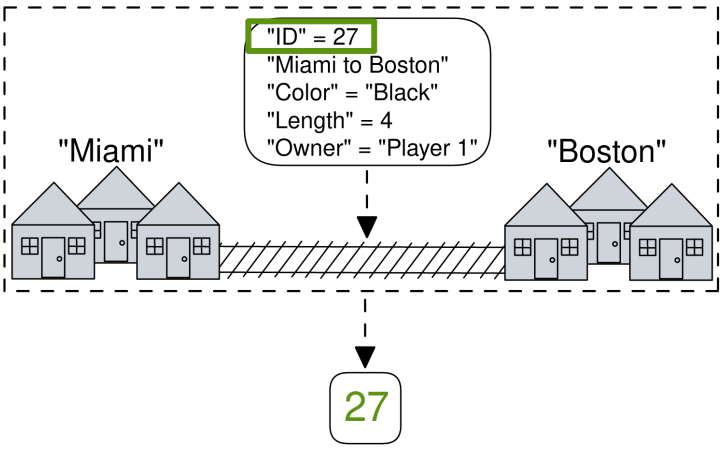


Figure 4.4: The encoding of two cities and one railroad can be done by simply using the unique ID of the railroad.

Observation State

The environment is encoded from the graph representation into an integer representation which the Reinforcement Learning Agent uses when interpreting the environment. The encoded environment is called the *observation state*. It contains all the railroads of the map in the encoded form as shown in figure 4.4, as well as the destination tickets pile and the Train Cards pile.

There are a total of 41 objects in the encoded environment, where the railroads range from ID 0 to 38 (39 railroads in total), the destination tickets pile has ID 100 and the Train Card pile has ID 200. The ID of a railroad is changed if it is claimed. If *Player 1* claim railroad 32, the ID 32 will be replaced with the integer *-1*. Railroads claimed by *Player 2* will have its ID replaced by the integer -2. ID 100 and 200 representing the card piles are strictly used to show if the piles are empty or not. If any pile is empty it will be replaced by the integer 0.

The observation state needs to be extended with the agents own cards on hand before it can be used as an input in the NN. The observation state will together with the agents cards on hand form an array that will be used by the agent when making decisions. The observation state combined with the agents hand is called *the agent state*. A visualization of this can be seen in figure 4.5.

The Agent State

As explained above, the agent state is the observation state combined with the agents hand. The following list will explain the contents of the marked array in figure 4.5.

1. Railroads

   The ID's and ownership status of the railroads, as previously explained.

2. Status of the card piles

   The ID's of the two card piles, as well if they are empty or not.

3. Train Cards

   Each index of the marked section represents a Train color. Each integer in each index represents how many of said color is in the agents hand. The figure shows that the agent has the following cards:
   [*0 wildcards, 1 green, 2 blue, 3 red, 0 yellow, 0 black, 6 white*]

4. Destination Tickets

   Each destination card in the game is unique. Each unique card has its own index in this subsection of the array. Integer 1 indicates that a particular card is in the agents hand, and if not it is indicated by the integer 0. There is a total of 17 destination cards and the indexes range from 48 to 65.

Figure 4.5: The top figure shows the Agent state and the bottom figure shows its content.

Action Space

A player can choose to either claim a railroad, draw a Train card, or draw a ticket. In this implementation there are 41 different actions. The player can choose to claim one of 39 railroads, draw cards from the Train Card pile or draw from the ticket pile. Each action is a tuple containing the type of action to be taken paired with the ID tied to this action which can be seen in figure 4.6.

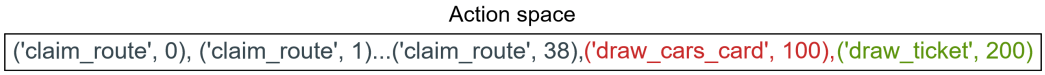


Figure 4.6: The contents of the action space array.

The agent will match the agent state with the action space and in this way filter out the unavailable actions, such as non-claimable railroads. The actions of drawing cards will be present in the action space array as long as the draw piles are not empty.

4.2.7 Action Limitations

Each player is faced with many choices during the course of the game, with some choices providing more complex outcomes than others. To reduce the number of choices and keep the complexity to a minimum, some game mechanics have been simplified without affecting the core game play.

The following sections will begin by describing how choices are made in the original edition of TTR, and will later show how these choices have been modified to fit the scope of this project. Different masking mechanisms have been implemented with the purpose to filter out invalid actions. This is to nudge the player into only choosing available actions, keeping the space complexity of all possible combinations of total amount actions at a minimum.

Selecting Actions

The three original actions of TTR have been kept. A player can choose between *Drawing Train Cards*, *Drawing Destination Tickets* or *Claiming Railroads* in a turn. Although how these choices work has been modified and adapted to fit the scope of this project.

Drawing Train Cards in the Adapted Game

As stated (Section 4.2.7) the action complexity is to be kept small due to the time constraints of the project. Figure 2.3 and 2.4 shows by intuition that a player can try to pick illegal Locomotive cards, while being denied to do so by the game logic, forcing the player to retake the action until a legal action has been made.

To remove the possibility of making illegal actions during the *draw Train Card* sequence, the logic has been modified so that if a player requests to draw Train Cards the player receives the top two cards from the face down pile. The face up pile has been removed in this implementation, limiting the decisions the players can make. After the adaptation the player only needs to decide if it wants to get cards or not. Removing any further decision making as seen in the below figure.

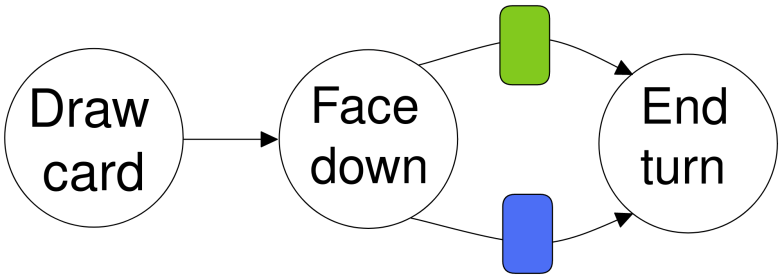


Figure 4.7: The draw Train Card action after its adaptation.

Drawing Destination Tickets in the Adapted Game

If the player wants a destination ticket, the player receives the top card from that pile. This also applies to the game mechanic where the player needs to select destination tickets in the beginning of the game.

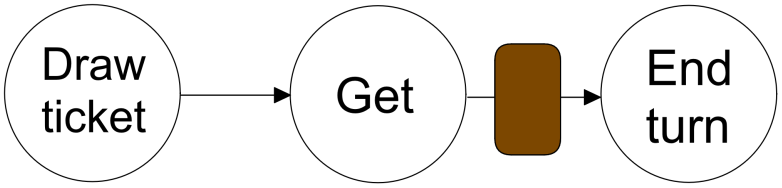


Figure 4.8: Illustration of the adapted draw destination tickets action.

Claiming Railroads in the Adapted Game

If the player chooses to claim a railroad, the player will be presented with a filtered map instead of the whole map. This is only for decision making purposes and, the players of the game will have full observability of the map. It will not affect the choices the players make other than nudging the players to claim valid railroads. An example of this can be seen in figure 4.9.

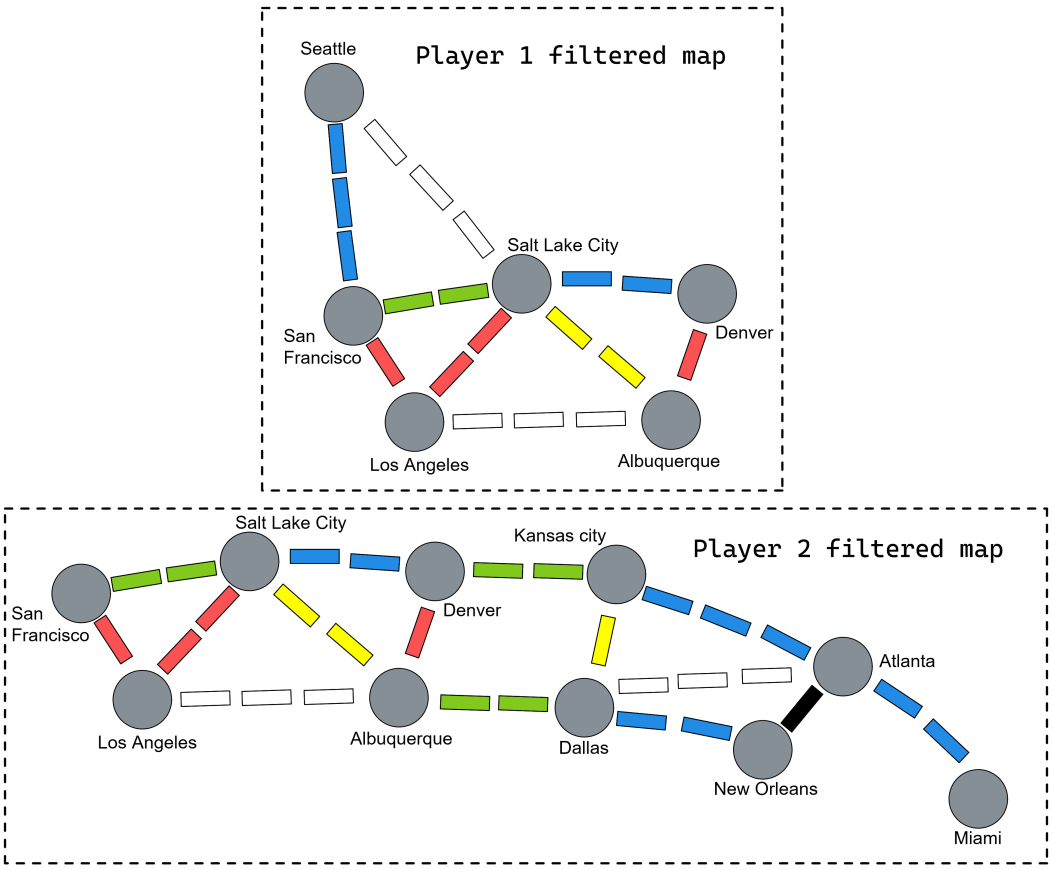
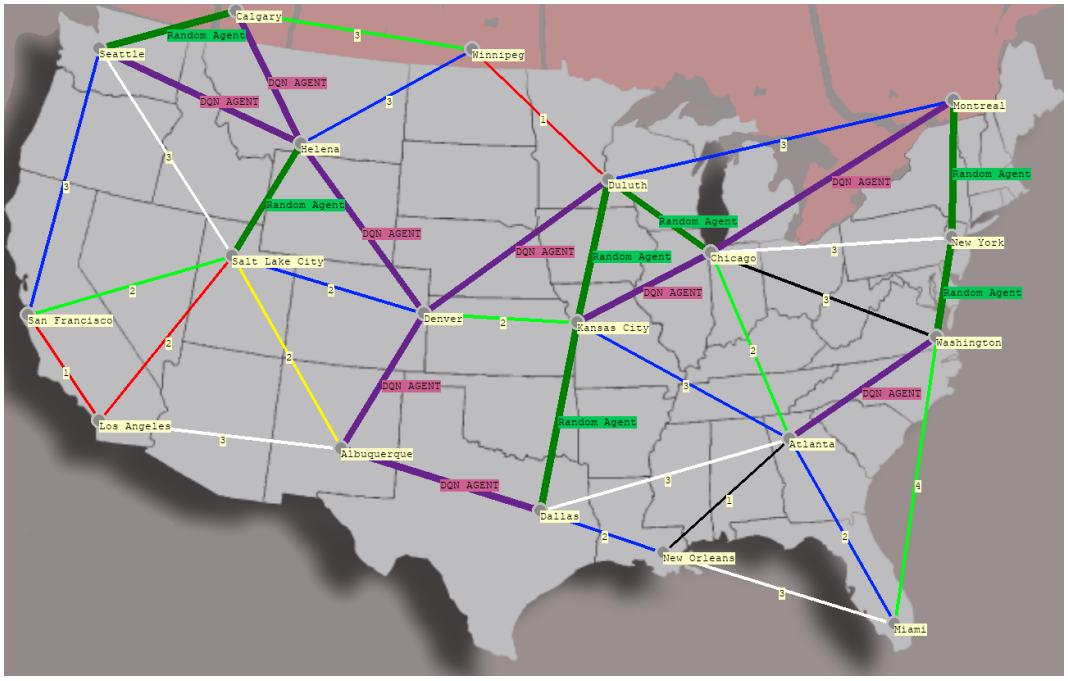


Figure 4.9: A view of what the players see if they choose to claim a railroad.

4.2.8 Graphical Representation

A graphical view that updates in real time has been added so that two artificial players can be observed during game play. The interface is not interactive, people playing the game need to use the command line interface. The view is updated when an agent makes a move. It was built using Pygame [32] and can be seen in the below figure.



4.3 Agent Architecture

This section will give an overview of the player classes functionality.

4.3.1 AbstractPlayer

The class AbstractPlayer is the base of all player classes. It holds information like name, id, score, list of cards on hand, railroads claimed and similar. There are also methods for the player to interact with the game, such as getting available actions or drawing cards from the card piles.

In this implementation there are three different player types that can interact with the game: The *Humanplayer*, the *RandomAgent* and the *DQNAgent*. All player types inherit the AbstractPlayer class.

4.3.2 Humanplayer

The HumanPlayer is at its core a command line interface of the game which is designed for humans. It shows a main menu containing actions, it takes keyboard inputs and it displays the game TTR in a sufficient way. The menu will show the three possible actions of TTR, and after a choice has been made relevant information will be displayed. If the player chooses to claim a railroad, the player is presented with a list of all available railroads.

If the player chooses to draw Train Cards the player receives the two top cards from the pile, as long as it is not empty. If the player chooses to draw a destination ticket the player receives the top card from that pile.

4.3.3 RandomAgent

The class RandomAgent is an agent that only takes random actions. The agent has a list containing the three actions *claim railroad*, *draw Train Card* and *draw destination ticket*.

The agent will first make a random choice from this list, and if the agent manages to pick the action *claim railroad* it will make another random choice from all the available railroads that the agent can afford to claim. If the choice is either *draw Train Card* or *draw destination ticket*, the game logic and rules will apply.

4.3.4 DQNAgent

The DQNAgent has been implemented by using and customizing the works of Yash Patel from the following article [33]. The agent has been modified in order to interface with the TTR implementation developed in this project. The agent in the article uses a Double DQN approach based on DeepMinds Double DQN [12]. This approach uses two NNs to perform actions within the environment. One primary network $\mathbb{Q}_\theta$ for action evaluation, and a target network $\mathbb{Q}_{\theta'}$ for action selection.

Algorithm 1

Algorithm 1 shows how the DQNAgent is being used in the main game loop. The pseudo code will be further explained in the following sections. The game starts by triggering all of the setup events (Section 4.1) and then getting a player to take an action.

Algorithm 1 Main Game Loop

Ensure: Initialized Players and Environment
Require: $n \leftarrow$ number of games
while Game is running **do**
  Player receives observation state S_t from environment.
  Agent state $A_t^S \leftarrow S_t$ + Player hand.
  Player executes action a_t based on A_t^S
  Player receives reward r_t.
  Player observes change in environment S_{t+1}.
  Agent state $A_{t+1}^S \leftarrow S_{t+1}$ + Player hand.
  if current Player is DQNAgent **then**
    Remember(A_t^S, a_t, r_t, A_{t+1}^S).
    Replay().
  end if
  if Player wins game **then**
    End the game.
  else
    End player turn and switch Player.
  end if
end while
Train target network $\mathbb{Q}_{\theta'}$ after n games

DQNAgent Architecture

The main attributes of the class consists of hyperparameters [34] such as *epsilon*, *gamma* and other constants related to the algorithm (Section 2.2.2). This class is inheriting from the class AbstractPlayer. It is built using Keras, which is a Python binding to the ML framework TensorFlow.

The Models

Since Double Q-learning is being used, two models need to be created. The two models, a primary network and a target network, are created with the same architecture.

The two NNs consists of one input layer of 65 neurons, one hidden layer of 55 neurons and one output layer of 41 neurons. Both the hidden layer and the output layer are dense layers meaning that each neuron in one layer is fully connected to all other neurons in the previous layer.

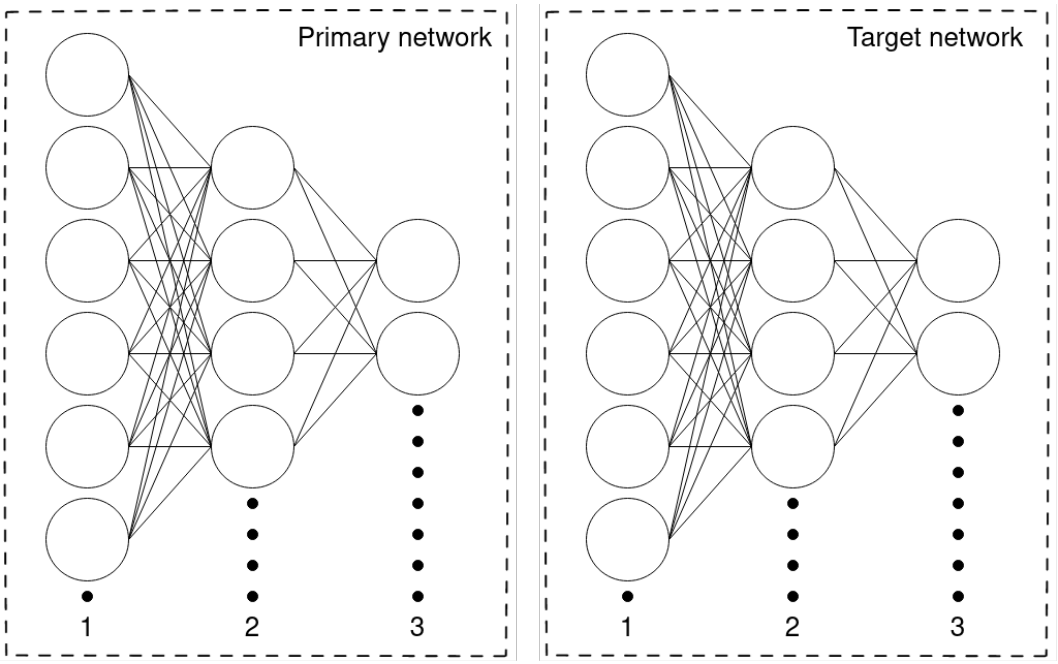


Figure 4.10: The two neural networks of the agent

1. Input layer

   Consists of 65 neurons. This layer is the agent state (Section 4.2.6). Each index in the array corresponds to one neuron in the layer.

2. Hidden layer

   Consists of 55 neurons. It is an intermediate layer between the input layer and the output layer, the size is based on [35] which states that the hidden layer should be a number between the input layer and the output layer. This layer uses the activation function *ReLu* [36].

3. Output layer

   Consists of 41 neurons where each neuron corresponds to one action in the action space (Section 4.2.6). It uses the activation function *linear* [37].

Both NNs use the loss function *Mean Squared Error* [38] and the optimizer *Adam* [39]. Adam (A Method for Stochastic Optimization) is an optimization algorithm that updates the weights in the NN. The Mean Squared Error loss function produces an estimate (loss) of how the values in the primary network differs from the target networks.

Action Masking

Illegal actions such as claiming railroads that the agent can not afford or are already claimed are masked in order to reduce a potentially high time complexity while choosing actions to take. The agent will always have access to the entire map, meaning the agent will be aware of what railroads have been claimed by opponents, the action masking will happen when it is time to choose an action.

Taking Steps

At each turn the agent will take one step. In this step the agent will select an action in one of two ways, either by picking a random valid action in the same manner as the RandomAgent (Section 4.3.3), or the primary network $\mathbb{Q}_\theta$ will return a Q-table where the agent will choose the action associated with the highest Q-value [7].

The epsilon ϵ value of the agent is reduced at the start of each turn, and then the current value of epsilon will be matched against a randomly generated floating point number between 0 and 1. If epsilon is greater than this random number, the agent will make a random action. If epsilon is lesser than the generated number, the agent will make a choice based on previous experiences. This behaviour is referred to as *exploration* and *exploitation* [2].

When a step has been taken the results are remembered by the agent. Algorithm 2 shows how a step is being taken.

Algorithm 2 DQNAgent Step()

Require: Primary network $\mathbb{Q}_\theta$, ϵ, ϵ_{decay} and $\epsilon > \epsilon_{decay}$

1: $\epsilon_{decay} \leftarrow 0.999$
2: $\epsilon * \epsilon_{decay}$
3: **if** $\epsilon >$ random number **then**
4: Return random action
5: **else**
6: q $\leftarrow$ get Q-table from $\mathbb{Q}_\theta$
7: $q_{masked} \leftarrow$ perform action masking on q
8: Return action with highest Q-value from q_{masked}
9: **end if**

Remembering

The agent remembers experiences (actions and their outcomes) in a similar manner as DeepMinds experience replay function. It will save what action was taken and what the results were in terms of rewards, effect on the environment, and if the action taken triggered a terminal state ending the game. This can be seen in Algorithm 3.

Algorithm 3 DQNAgent Remember(A_t^S, a_t, r_t, A_{t+1}^S)[33]

Require: A_t^S, a_t, r_t, A_{t+1}^S

1: Store (A_t^S, a_t, r_t, A_{t+1}^S) in Agents memory.

Action Evaluation

After each step the agent will perform a replay seen in Algorithm 4. The purpose of this replay is to update the values of the Q-table in the primary network $\mathbb{Q}_\theta$.

Algorithm 4 DQNAgent Replay()

Require: Primary network $\mathbb{Q}_\theta$, Target network $\mathbb{Q}_{\theta'}$, γ, *batch_size*
batch_size $\leftarrow$ int
if *batch_size* is greater than the agents memory **then**
  Return
else
  sample_list $\leftarrow$ Get *batch_size* random sample objects from agents memory.
  sample_object $S_{obj} = (A^S_t, a_t, r_t, A^S_{t+1})$
end if
for each S_{obj} from sample_list **do**
  if state is *terminal* **then**
    Update Q-table in $\mathbb{Q}_{\theta'}$ with r_t
  else
    Update Q-table in $\mathbb{Q}_{\theta'}$ with r_t + (max ($\mathbb{Q}_{\theta'(A^S_{t+1}+a_t)}$)* γ)
  end if
  Train primary network $\mathbb{Q}_\theta$.
end for

Updating Target Network $\mathbb{Q}_{\theta'}$

After a fixed number of games played, the target network $\mathbb{Q}_{\theta'}$ is updated. The weights of the primary network $\mathbb{Q}_\theta$ are being copied into the target network $\mathbb{Q}_{\theta'}$, replacing the target networks own weights. Algorithm 5 shows how the target network is updated.

Algorithm 5 DQNAgent Target_train()

Require: Primary network $\mathbb{Q}_\theta$, Target network $\mathbb{Q}_{\theta'}$
primary_weights $\leftarrow$ Get weights from $\mathbb{Q}_\theta$
target_weights $\leftarrow$ Get weights from $\mathbb{Q}_{\theta'}$
Set target_weights $\leftarrow$ primary_weights
Update $\mathbb{Q}_{\theta'}$ with target_weights

Reward Function

The reward functions are designed to be used when giving feedback to the agent during its training. Rewards are given to the agent whenever the agent performs an action, as well as at the end of the game when the final score is being calculated. In this implementation there are four different reward functions that the agent can use.

1. **Integer rewards with bonus**
    - Claiming railroads: 1, 2, 4 or 7 reward value.
    - Drawing Train Cards or destination tickets: 0 reward value.
    - Completing drawn destination tickets: 3, 4, 5, 9, 10 or 11 reward values.
    - Failing to complete drawn destination tickets: -3, -4, -5, -9, -10 or -11 reward values.
    - Building the longest railroad: 10 reward value.
    - Bonus reward for winning the game: 20 reward value.
2. **Integer rewards with no bonus**
    - Same as the above reward function, except the bonus reward if winning the game.
3. **Floating point rewards with bonus**
    - Claiming railroads: 0.01, 0.02, 0.04 or 0.07 reward value.
    - Drawing Train Cards or destination tickets: 0 reward value.
    - Completing drawn destination tickets: 0.03, 0.04, 0.05, 0.09, 0.1 or 0.11 reward values.
    - Failing to complete drawn destination tickets: -0.03, -0.04, -0.05, -0.09, -0.1 or -0.11 reward values.
    - Building the longest railroad: 0.1 reward value.
    - Bonus reward for winning the game: 1 reward value.
4. **Floating point rewards with no bonus.**
    - Same as the above reward function, except the bonus reward if winning the game.

4.4 Training DQNAgent

This section will explain how the four different agents were trained and what tools were used under the process.

Parallelized Training

Four different instances of the game TTR were run in parallel using Docker containers [40]. Each Docker container contained a DQNAgent using one of the four reward functions. Each container ran 250 games at a time, were the DQNAgents trained using self-play. Different performance data is saved during the sessions. The data saved contained the trained model of the DQNAgent along with performance metrics. After each training session of 250 games, the data collected were extracted from the containers and analyzed.

The DQNAgent that won the most games during the self-play training were saved and used as a base for the next training session. This method was used until 750 games had been played and trained. One training session of 250 games took approximately 16 hours, resulting in a total of 64 hours of training. This excludes the number of wasted training sessions where faulty parameters or other mistakes were made. Training was done using a *AMD Ryzen 5 3600* Central Processing Unit (CPU) [41].

Normalizing the Agent State

The input layer in the neural network (Section 4.3.4) was not normalized [42] from the beginning of the project. Since the ReLu function changes all inputs in the neural network of negative values into zeros, it appeared like it could have a negative impact on the results due to the use of negative numbers in the playerÂ´s ID. Hence the agent state (Section 4.2.6) were normalized using unsigned floating point numbers. The ID of the agent passed to the neural network was changed from -1 and -2, to 0.1 and 0.2. The other values in the agent state were changed in a similar manner.

Analysing Performance Data

Tensorboard is a visualization tool provided in the ML framework TensorFlow used when analyzing the performance metrics collected during training. The plotted lines shows data collected during the agents individual training sessions. The data does not show the result of agents playing against each other. The data analysed was total reward and score per game, as well as how the loss, the accuracy and the epsilon value evolved during the training session.

The lines in the following graphs show one DQNAgent using the reward function *integer rewards with bonus* (blue), and one DQNAgent using the reward function *integer rewards with no bonus* (purple). Figure 4.11 displays how the *epsilon* value (Section 2.2.2) is decreasing in both the agents for each game played.

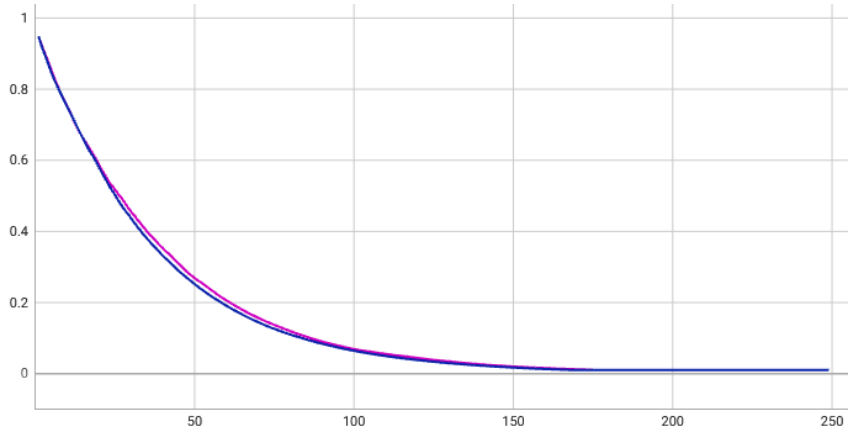
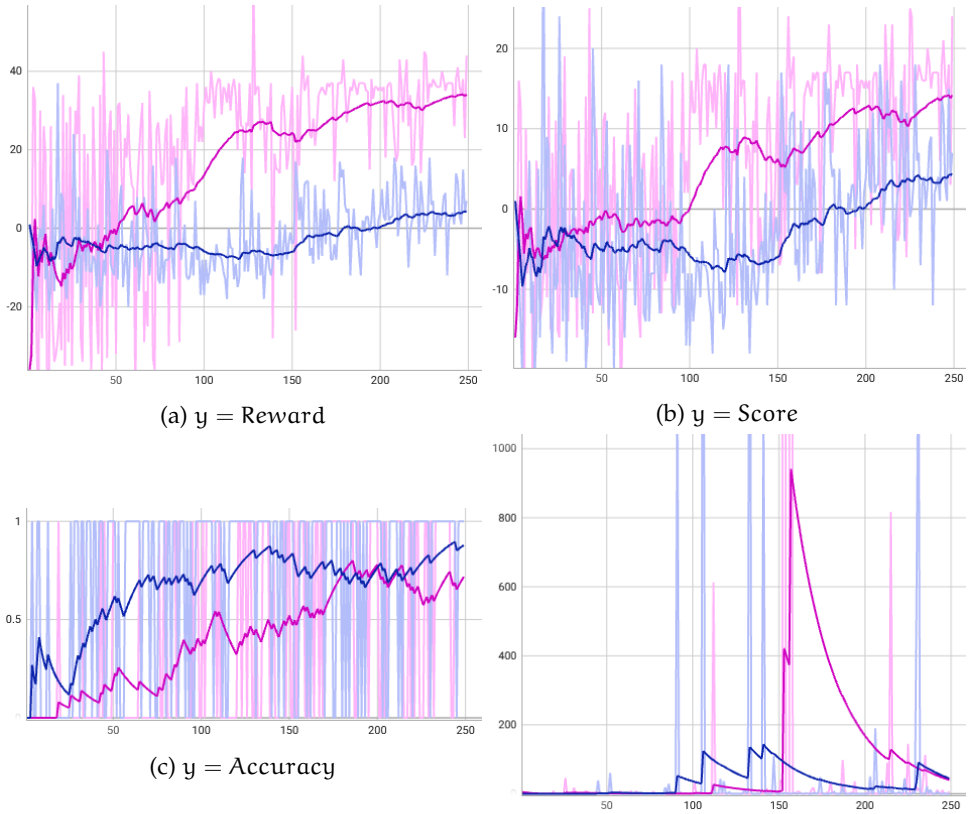


Figure 4.11: $y = Epsilon$

Figure 4.12a shows the total reward for each game played accompanied with figure 4.12b that shows the total score.

The data from figure 4.12c displays the accuracy and figure 4.12d displays the loss calculated each time the agent trains the primary network Q_θ. Loss [43] represents how good or bad the primary network Q_θ is performing. The value of accuracy [44] is a measurement of how well the model predicted the action-values from the primary network Q_θ with the values of these actions in the target network $Q_{\theta'}$.



(a) $y = \text{Reward}$

(b) $y = \text{Score}$

(c) $y = \text{Accuracy}$

(d) $y = \text{Loss}$

Figure 4.12: Data showing that reward, score, accuracy and loss improves over time.

Chapter 5
Results

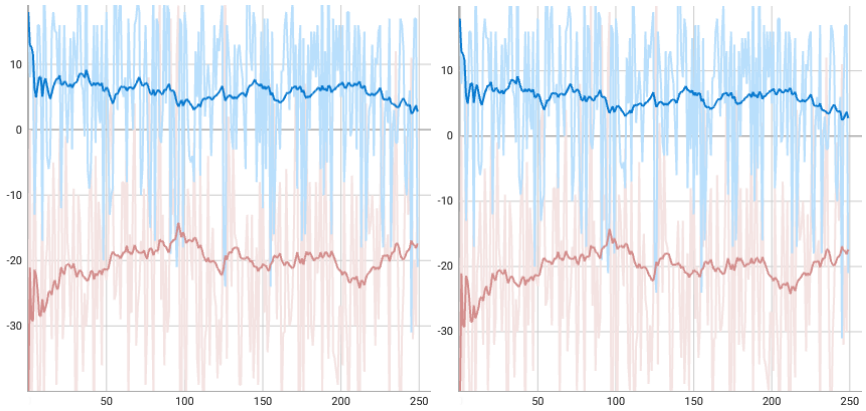
This chapter shows the performance results of the trained DQNAgents (Section 4.3.4).

Four different reward functions have been used during the training phase of the agents, *Integer rewards with bonus*, *Integer rewards with no bonus*, *Floating point rewards with bonus* and *Floating point rewards with no bonus*. After the training was done each agent competed against a RandomAgent (Section 4.3.3) for 250 games. Performance metrics from these games are plotted in the below graphs. The results will be discussed in chapter 6.

5.1 Game Results

The x-axis in all of the below graphs shows the number of games played. Each graph is supplemented with a tabular summary of game data. Games that resulted in a tie were deducted from the summarized data.

250 games of experience, integer rewards with no bonus.

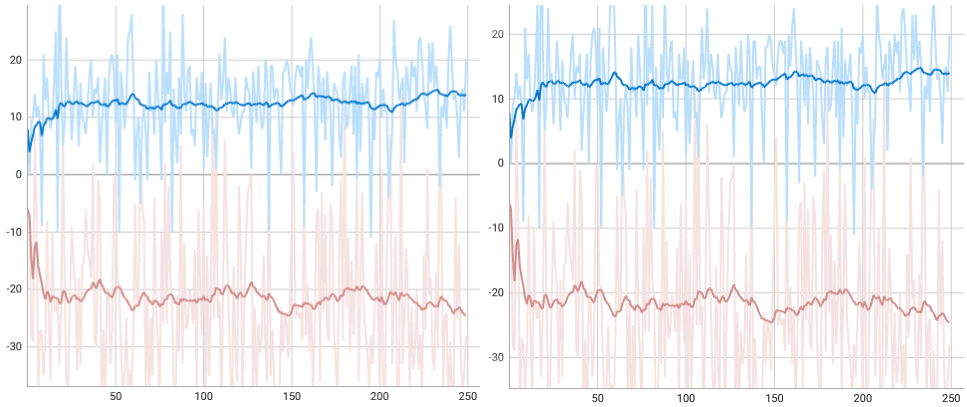


(a) $y = reward$ (b) $y = score$

Figure 5.1: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	209/250 (83.6%)	5.46	5.46
RandomAgent	41/250 (16.4%)	-19.676	-19.676

750 games of experience, integer rewards with no bonus.

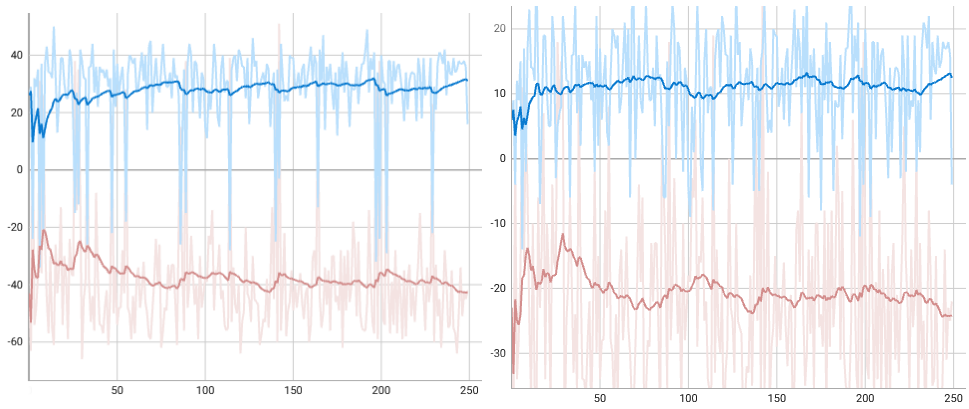


(a) $y = reward$ (b) $y = score$

Figure 5.2: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	240/250 (96.0%)	12.724	12.724
RandomAgent	10/250 (4.0%)	-21.848	-21.848

250 games of experience, integer rewards with bonus.

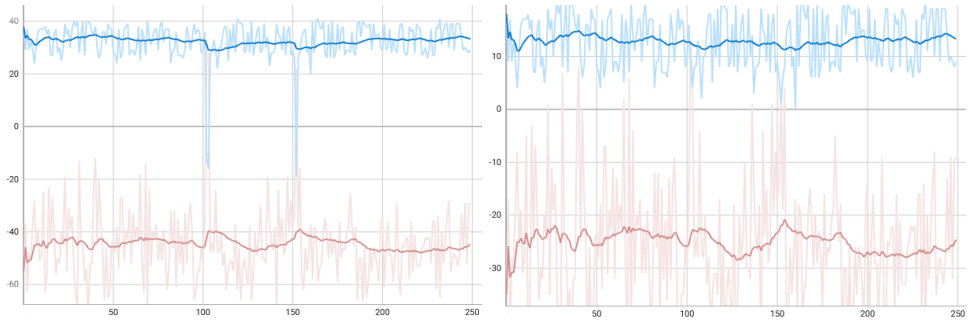


(a) $y = reward$ (b) $y = score$

Figure 5.3: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	232/250 (92.8%)	28.516	11.396
RandomAgent	18/250 (7.2%)	-38.124	-21.004

750 games of experience, integer rewards with bonus.

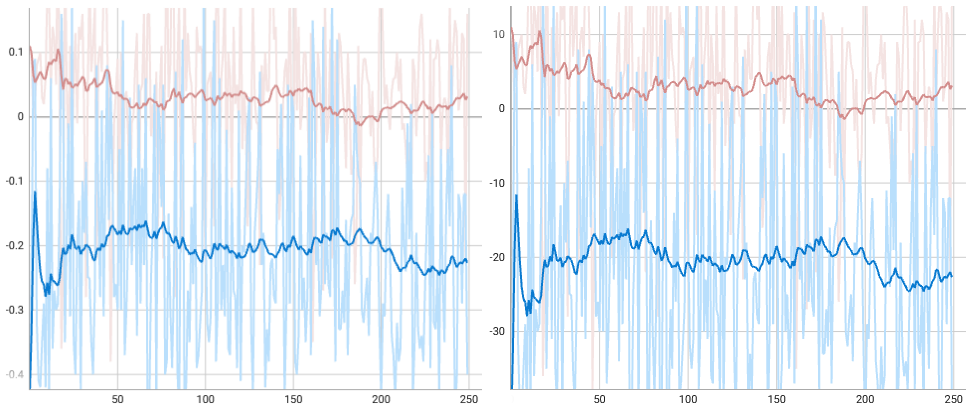


(a) $y = reward$ (b) $y = score$

Figure 5.4: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	247/250 (98.8%)	32.436	12.916
RandomAgent	3/250 (1.2%)	-44.56	-25.04

250 games of experience, floating point rewards with no bonus.

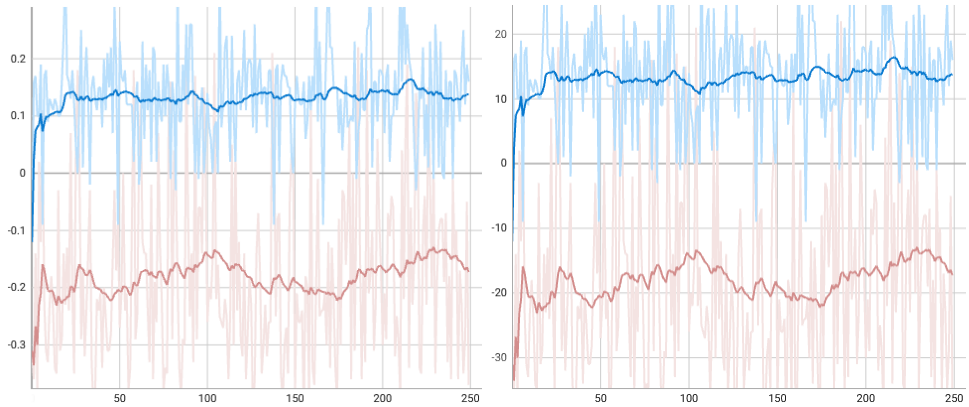


(a) $y = reward$ (b) $y = score$

Figure 5.5: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	56/250 (22.4%)	-0.2064	-20.644
RandomAgent	193/250 (77.2%)	0.0272	2.724

750 games of experience, floating point rewards with no bonus.

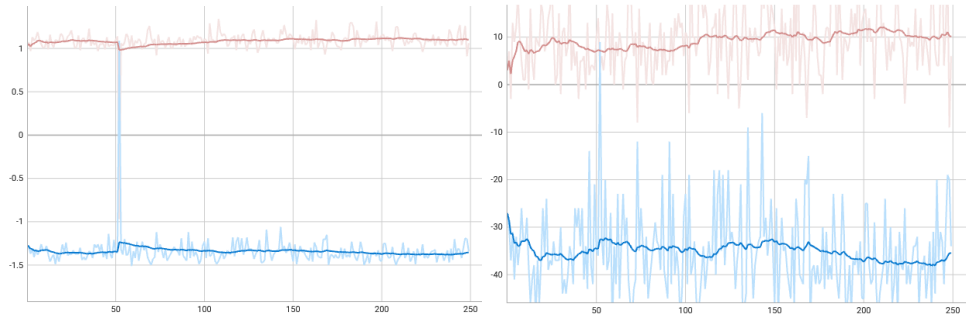


(a) $y = reward$ (b) $y = score$

Figure 5.6: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	224/250 (89.6%)	0.134	13.408
RandomAgent	25/250 (10%)	-0.177	-17.788

250 games of experience, floating point rewards with bonus.

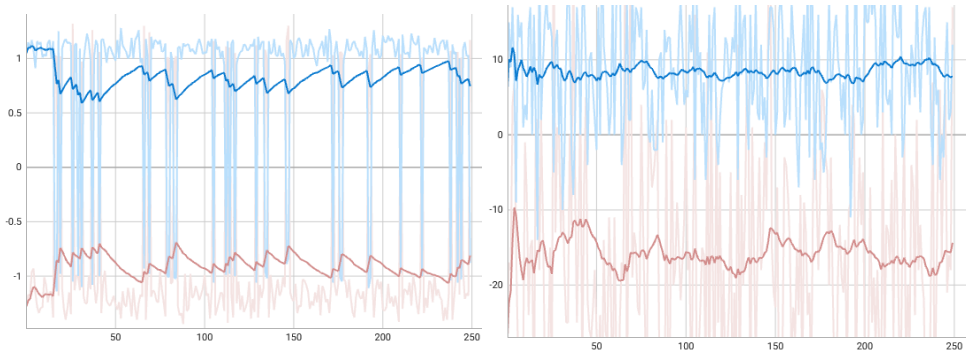


(a) $y = reward$ (b) $y = score$

Figure 5.7: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	1/250 (0.4%)	-1.344	-35.204
RandomAgent	249/250 (99.6%)	1.087	9.512

750 games of experience, floating point rewards with bonus.



(a) $y = reward$ (b) $y = score$

Figure 5.8: DQNAgent (blue) vs RandomAgent (red)

Summary of 250 games played			
Player type	Games won	Average reward	Average score
DQNAgent	216/250 (86.4%)	0.811	8.364
RandomAgent	33/250 (13.2%)	-0.893	-15.704

5.1.1 Summary of all Games

The below table shows all the above results together. Again, games that ended in a tie are omitted from the data.

Summary of 250 games played						
Exp.	Player	Games won	Avg reward	Avg score	Type	Bonus
250	DQN	(92.8%)	28.516	11.396	Int	Yes
250	Random	(7.2%)	-38.124	-21.004	Int	Yes
750	DQN	(98.8%)	32.436	12.916	Int	Yes
750	Random	(1.2%)	-44.56	-25.046	Int	Yes
250	DQN	(83.6%)	5.460	5.460	Int	No
250	Random	(16.4%)	-19.676	-19.676	Int	No
750	DQN	(96.0%)	12.724	12.724	Int	No
750	Random	(4.0%)	-21.848	-21.848	Int	No
250	DQN	(0.4%)	-1.344	-35.204	Float	Yes
250	Random	(99.6%)	1.087	9.512	Float	Yes
750	DQN	(86.4%)	0.811	8.364	Float	Yes
750	Random	(13.2%)	-0.893	-15.704	Float	Yes
250	DQN	(22.4%)	-0.2064	-20.644	Float	No
250	Random	(77.2%)	0.0272	2.724	Float	No
750	DQN	(89.6%)	0.134	13.408	Float	No
750	Random	(10%)	-0.177	-17.788	Float	No

Chapter 6
Conclusions

In this chapter the results from the previous chapter are discussed and reflected upon. The project is then out into different contextual aspects and is followed by conclusions made by us. The chapter ends with a summary of future ideas to try going forward.

6.1 Discussion

Integer rewards with bonus

When comparing the agent with 250 games of experience with the agent with 750 games of experience, an increase in average reward by 12.1% and score by 11.8% can be seen. Games won by the agent increased by 6%.

Integer rewards with no bonus

When comparing the agent with 250 games of experience with the agent with 750 games of experience, an increase in average reward and score by 57.1% can be seen. Games won by the agent increased by 12.4%.

Floating point rewards with bonus

When comparing the agent with 250 games of experience with the agent with 750 games of experience, an increase in average reward by 265% and score by 520% can be seen. Games won by the agent increased by 86%.

Floating point rewards with no bonus

When comparing the agent with 250 games of experience with the agent with 750 games of experience, an increase in average reward by 253.7% and score by 253.9% can be seen. Games won by the agent increased by 67.2%.

Conclusion

An increase in performance can be seen across all agents. Agents with integer type reward functions seem to perform well after only 250 games of experience, were agents with floating point number rewards perform a lot worse at the same level of experience.

Agents given 500 additional matches of experience shows that agents with floating point rewards are starting to catch up with the integer reward agents. This suggests that those who perform worse in the beginning might perform even better if given more experience. An overview of the increased performance can be seen in the table below.

Performance development						
Exp.	Player	Games won	Avg reward	Avg score	Type	Bonus
250	DQN	92.8%	28.516	11.396	Int	Yes
750	DQN	98.8%	32.436	12.916	Int	Yes
+500		+6%	+12.1%	+11.8%		
250	DQN	83.6%	5.460	5.460	Int	No
750	DQN	96.0%	12.724	12.724	Int	No
+500		+12.4%	+57.1%	+57.1%		
250	DQN	0.4%	-1.344	-35.204	Float	Yes
750	DQN	86.4%	0.811	8.364	Float	Yes
+500		+86%	+265%	+520%		
250	DQN	22.4%	-0.206	-20.644	Float	No
750	DQN	89.6%	0.134	13.408	Float	No
+500		+67.2%	+253.7%	+253.9%		

Tactical Repertoire

Even though there is no particular goal in the project to evaluate what exactly an agent has learned in terms of strategies or knowledge in general, some observations were made where it looks like the agents might have learned strategies. Reflections regarding tactics are discussed in the following subsections.

Completing Destination Tickets

An observation made is that the DQNAgentÂ´s number of Destination Tickets on hand per game decreases over time. This might be a result from the players receiving negative rewards for all uncompleted Destination Tickets left in hand at the end of the game. The DQNAgents might have developed a strategy to avoid this potential final negative reward signal.

Longest Railroad

The event of DQNAgents completing the longest railroad mission can be seen in some agents behaviour more than others. The occurrence of this event is too rare to draw any concrete conclusions that the agents are knowingly building the longest railroad on purpose.

Claiming Railroads in General

A strategy could be that the agent simply tries to claim the most railroads since each claim gives an intermediate reward.

Using the graphical interface we could see that each DQNAgent often started building railroads in particular areas of the map. For instance, one DQNAgent claimed railroads from Calgary to Helena, and from Helena to Denver very often. Why this happened a lot is unclear, but one reason could be that there is a short destination ticket that has a mission to claim railroads connecting Calgary and Albuquerque, and a long destination from Calgary to Atlanta.

Conclusions

Within the scope and time frame of this thesis it is hard to tell if the observations really are strategies developed by the agents.

Training Data

Figure 4.12d displays a graph of the loss. According to [45] spikes in loss are common when using too small batch sizes, non-optimal loss functions or too high learning rates. The observation in loss spikes is something that can be addressed in future research and will be discussed further in (Section 6.4).

Figure 4.12c displays the accuracy of two DQNAgents. A point where the accuracy is plateauing or decreasing has not been reached during the training. At this point we can not draw any conclusions from this.

Ticket To Ride Implementation

The agents could be trained in different environments in order to increase the agents resilience, and in the end be able to play different expansion sets of TTR The scope and time constraints of this project did not allow for a full implementation of TTR. As stated before, the map and the action sequences have been limited.

Given more time a full implementation of the game could be used when training the agent. This would open up the opportunity to yield a more resilient agent that was not nudged during training, recalling that this implementation did not allow the players to take illegal actions.

6.2 Contextual Aspects

RL is based on the training of programs to perform certain tasks, were some programs need more training than others. This requires a high amount of computing power, and the more training a program needs the more energy it will consume.

There are many examples where RL yield positive long term impact on society. One example is the use of RL and AI in cancer detection [46]. One can argue what worth an agent who learnt to play a board game has in the same context.

This project has been a learning experience, and having more people getting into the field of ML and RL could in the long run result in even more tools to aid humanity in solving complex problems. In the short term, this project could be considered a huge waste of energy since the result is an agent playing a custom version of TTR.

Using RL to make an agent learn how to play a game raises ethical questions regarding data collection and profiling of human behaviour. Recalling that Google DeepMind [13] combined data collected from professional Go players with RL, Google DeepMind managed to create an AI agent capable of beating a human in the game of Go, a game that relies heavily on interpreting and predicting an opponents behaviour.

This area of technology could be of interest in the military industry or in the hands of nation states striving for automated surveillance. There are cases where algorithms have been involved in putting people on probation [47], which raises questions regarding bias in AI. Since humans are interpreting the law and also designing the algorithms, the question can be asked if people sentenced by AI are treated equally and respectfully. Bias in AI must always be considered when letting AI algorithms into real life situations, affecting peoples lives.

6.3 Conclusion

The purpose of the thesis was to attempt an implementation of an intelligent agent learning and playing a discreet board game. This was demonstrated by implementing a custom version of the board game Ticket To Ride, with an agent playing the game after it had been trained using Reinforcement Learning methods.

The level of success can be seen in the performance comparison where an agent that takes completely random actions loses in the majority of games facing different iterations of agents trained using Reinforcement Learning. We deem that this project has been successfully carried out. A review of the project goals from Chapter 1 can be seen below.

1. Create a discrete board game as a video game.

   The full game of TTR, playable by people, was successfully implemented in the start of the project. After this it was modified in order for artificial players to interact with it.

2. Create an RLA.

   A RandomAgent was created and could play against a human player, thus the goal of creating an artificial agent that could play our game was successful. The work continued where an RLA using DDQN was implemented. It was trained using Reinforcement Learning methods and was able to beat a RandomAgent at a vast majority of games played. This is to be considered a success.

3. Documentation

   A thesis has been written.

6.4 Future Work

All areas of this project can be further explored. The architecture of the neural network could be further experimented with using different loss functions, optimization algorithms or using a different number of layers. Hyperparameter tuning is a big part of the RL area and more experiments using different hyperparameter configurations can be explored.

Future research could be done using a Graphics Processing Unit (GPU) to see if more effective calculations can be achieved. Training can also be distributed over several workstations where several more agents could be trained with different hyperparameter settings.

In [20] Dinjian and Nguyen discussed Curriculum Learning, where an agent is trained in certain areas, incrementally building its experience. The idea of using this approach would be interesting to try. Combining RL and supervised learning in a similar manner as Google DeepMind did with AlphaGo is another possible approach.

Acronyms

A

AI Artificial Intelligence

AN Artificial Neuron

ANN Artificial Neural Network

C

CPU Central Processing Unit

D

DDQN Double Deep Q-Network

DP Dynamic Programming

DQN Deep Q-Network

G

GPU Graphics Processing Unit

M

MCTS Monte Carlo Tree Search

MDP Markov Decision Process

ML Machine Learning

N

NN Neural Network

R

RL Reinforcement Learning

RLA Reinforcement Learning Agent

S

SPC State-Space Complexity

T

TTR Ticket To Ride

References

[1] Alan R. Moon. Ticket to ride usa. https://ncdn0.daysofwonder.com/tickettoride/en/img/7201-T2R-Rules-EN-2019.pdf. Accessed: 2022-04-27. (Cited on pages 8 and 20.)

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press Cambridge, Massachusetts London, England, 2 edition, 2018. (Cited on pages 14, 19, and 32.)

[3] Jennifer L. Diedrich. Motivating students using positive reinforcement, 2010. (Cited on page 14.)

[4] L Thorndike and Darryl Bruce. *Animal intelligence: Experimental studies*. Routledge, 2017. (Cited on page 14.)

[5] Ronald A Howard. Dynamic programming and markov processes. 1960. (Cited on page 14.)

[6] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. 2015. (Cited on page 15.)

[7] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989. (Cited on pages 15 and 32.)

[8] Hado Hasselt. *Double Q-learning*. https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf, 2010. (Cited on page 16.)

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. *Human-level control through deep reinforcement learning*. *Nature*, 518(7540):529–533, February 2015. (Cited on pages 16 and 18.)

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015. (Cited on page 16.)

[11] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007. (Cited on page 16.)

[12] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*, 2015. (Cited on pages 17 and 30.)

[13] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya

Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. *Mastering the game of Go with deep neural networks and tree search*. *Nature*, 529:484–489, 01 2016. (Cited on pages 18 and 45.)

[14] Wikipedia. *Lee Sedol*. `https://en.wikipedia.org/wiki/Lee_Sedol`. (Cited on page 18.)

[15] DeepMind. *AlphaGo - The Movie*. `https://www.alphagomovie.com/`. (Cited on page 18.)

[16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. *Mastering the game of Go without human knowledge*. *Nature*, 550:354–359, 10 2017. (Cited on page 18.)

[17] Wikipedia. *Tabula rasa*. `https://en.wikipedia.org/wiki/Tabula_rasa`. (Cited on page 18.)

[18] Erik Blomqvist. *Playing the Game of Risk with an AlphaZero Agent*, 2020. (Cited on page 19.)

[19] Michael Wolf. *An Intelligent Artificial Player for the Game of Risk*. Master's thesis, University of Technology Department of Computer Science, Darmstadt, 2005. (Cited on page 19.)

[20] Daniel Dinjian and Cuong Nguyen. The difficulty of learning ticket to ride, 2010. (Cited on pages 19 and 46.)

[21] Carina Huchler. *AN MCTS AGENT FOR TICKET TO RIDE*. Master's thesis, Maastricht University Department of Knowledge Engineering Maastricht, The Netherlands, 2015. (Cited on pages 19 and 21.)

[22] Fernando de Mesentier Silva, Scott Lee, Julian Togelius, and Andy Nealen. *AI-based Playtesting of Contemporary Board Games*. Technical report, New York University, Tandon School of Engineering. (Cited on page 19.)

[23] Pytorch. `https://pytorch.org/`. (Cited on page 20.)

[24] Keras. `https://keras.io/`. (Cited on page 20.)

[25] Torch. `http://torch.ch/`. (Cited on page 20.)

[26] Tensorflow. `https://www.tensorflow.org/`. (Cited on page 20.)

[27] Alan R. Moon. Ticket to ride first journey. `https://www.daysofwonder.com/tickettoride/en/first-journey/`. Accessed: 2022-04-27. (Cited on page 20.)

[28] Jürgen Schmidhuber. *Deep learning in neural networks: An overview*. *Neural networks*, 61:85–117, 2015. (Cited on page 22.)

[29] Jason Brownlee. Ordinal and one-hot encodings for categorical data. `https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/`. (Cited on page 22.)

[30] Networkx developers. `https://networkx.org/`. Accessed: 2022-04-27. (Cited on page 23.)

[31] Categorical variables. `https://en.wikipedia.org/wiki/Categorical_variable`. (Cited on page 24.)

[32] Pygame. Pygame. `https://www.pygame.org/news`. (Cited on page 29.)

[33] Yash Patel. Reinforcement learning w/ keras + openai: Dqns. https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c. (Cited on pages 30 and 32.)

[34] Wikipedia. Hyperparameters (machine learning). `https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)`. (Cited on page 31.)

[35] Sandhya Krishnan. *How to determine the number of layers and neurons in the hidden layer?* https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3. (Cited on page 31.)

[36] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*, 2018. (Cited on page 31.)

[37] Tomasz Szandala. *Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks. CoRR*, abs/2010.09458, 2020. (Cited on page 31.)

[38] Hossein Pishro-Nik. *Introduction to probability, statistics, and random processes*. Kappa Research, LLC, 2016. (Cited on page 32.)

[39] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. (Cited on page 32.)

[40] Docker. Docker. `https://www.docker.com/`. (Cited on page 34.)

[41] amd. *AMD*. `https://www.amd.com/en/products/cpu/amd-ryzen-5-3600`. (Cited on page 34.)

[42] Data normalization. `https://deepchecks.com/glossary/normalization-in-machine-learning/`. (Cited on page 35.)

[43] Google. Training and loss. `https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss`. (Cited on page 35.)

[44] Google. Accuracy. `https://developers.google.com/machine-learning/crash-course/classification/accuracy`. (Cited on page 35.)

[45] Jeffrey M Ede and Richard Beanland. *Adaptive learning rate clipping stabilizes learning*. *Machine Learning: Science and Technology*, 1(1):015011, mar 2020. (Cited on page 45.)

[46] Konstantina Kourou, Themis P. Exarchos, Konstantinos P. Exarchos, Michalis V. Karamouzis, and Dimitrios I. Fotiadis. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, 13:8–17, 2015. (Cited on page 45.)

[47] NY Times. An algorithm that grants freedom, or takes it away. `https://www.nytimes.com/2020/02/06/technology/predictive-algorithms-crime.html`. (Cited on page 45.)