Simulating Artificial Players in Games

Comparison of AI techniques for NPC controllers in video games

Tobias Oliver Jensen
tojen15@student.sdu.dk

Master's Thesis
Fall 2020/Spring 2021

Game Development and Learning Technology
University of Southern Denmark, Odense
June 1, 2021

Contents

Abstract

This project attempts to artificially mimic specific play styles that human player's might use when playing a simple rogue-like game. Through the use of artificial intelligence (AI), the possibilities are explored using both classic forms of AI, in the form of finite state machines and behavior trees, and more advanced methods in the form of neural networks. To get a better understanding of how they compared to actual humans, the AI techniques are implemented, the neural networks are trained, and all the techniques are compared with an actual human through a Game of Imitation. To keep the scope optimal, the project aims at replicating the movements performed by players, when they are playing as speedrunners, explorers, and treasure hunters. The Game of Imitation is executed through a survey containing six recordings of the agent being controlled either by a human or an AI. The participants in the survey are tasked with ranking the recordings from most to least human-like, after which their responses are compared to their demographics as to get an insight into what people might be easier to fool using AI as character controllers. Though the project focuses on simulating humans, the main focus is on neural networks, as they are capable of learning how to act based on training, therefore, in theory, requiring less actual work when trying to simulate humans. The project did have some flaws that made the final results inconclusive, as there are research that points to agents being able to imitate play styles, but the project still highlights certain areas of interest, and certain issues, that should be addressed in future research.

Introduction

Computer technology is quickly reaching new heights for assisting humans in different tasks as the technology is evolving. Especially one area has been of interest for many decades, which is the idea of intelligent computers that can act like and communicate with humans. For this, machine learning has been an area of research since the 1940's, where the artificial neural network was first theorized. Ever since, artificial intelligence has been an important area of research, as it allows computers to operate more intelligently through algorithms or machine learning. To further develop artificial intelligence, many researchers are using video games as a medium for development, testing, and comparing machine's way of performing in certain problem spaces. As video games provide a good platform for experimenting with artificial intelligence, attempting to create more advanced and intelligent agents to interact with inside games could be a milestone towards more general intelligence for the future. Having more intelligent and human-like characters in a game could also be a major improvement for future games, which might improve replay-ability of the games, or make for a better challenge and better entertainment. The main reason for adding artificial intelligence to a game is creating a more interesting experience, in which the world is procedurally generated, and agents can provide a bigger challenge. In working towards more entertaining agents for games, trying to imitate people's way of playing a game by using artificial intelligence could be a way of making games more immersive, thereby increasing the entertainment. To do this, it is important to note what play styles that are available to a certain game genre, as these can be converted into personas that can be simulated by agents. Furthermore, it is important to explore what artificial intelligence techniques that can be utilized for creating these types of agents, as there are many different techniques with different strengths and weaknesses.

Therefore, this project will work with artificial intelligence for games in an attempt to create agents that can simulate players. The aim will be making agents that can analyzing a visible part of an environment and make decisions as to what action will bring it closer to a set goal. For this, the following problem statement will be the focus of the research:

How could artificial agents be created to simulate players in games?

To explore this problem statement, the following research questions will be addressed:

1. **What is the simplest technique that can simulate a player's behavior?**
2. **How could machine learning be used to teach agents based on a player's performance?**
3. **How advanced must agents become to simulate players in a convincing way?**

To address these questions, a simple game is required that can be used to gather necessary information for training the machine learning techniques, and later it can be used to compare the performance of the different techniques. As there are many different algorithms and methods for controlling an AI agent in a game, selecting specific techniques will be necessary, as the scope cannot compare them all against each other. Deciding what techniques to use will play an important role in the process as a whole, as certain techniques are already commonly used, while others are good for map analysis. Lastly, a test to compare the performance of the different techniques will be carried out based on Turing's Game of Imitation, where the agents will be compared by attempting to fool participants into believing that the agents are actually human players.

Background

This section will go through different topics that had relevance for planning and executing the project, as to develop a proper test platform, select methods, and implement the different artificial intelligence techniques used for the agent.

Game Design & Non-Player Characters

Video games are systems designed to present a challenge, where a player voluntarily tries to reach a goal, while constrained by rules, by using feedback from the game's environment (McGonigal, 2011). Defining

proper boundaries, rules, and feedback systems are important for the game to function optimally for players. Usually, the game is build around one specific core mechanic that the player must utilize to beat the system and reach the goal (Burgun, 2015, Section 2). This mechanic is confined by the rules, and the feedback systems tell the player, how its utilization works in the game world. Games are not confined to a single core mechanic, but can utilize more complex combinations of mechanics to generate good gameplay, though other mechanics are often seen as supporting mechanisms to the core (Burgun, 2015, Section 4). The importance of support mechanics allows multiple different actions to be performed, so the game does not become stale with a single mechanic to perform throughout the whole game. However, it is important that the game only contains valuable mechanisms that are required to reach the goal, as they would otherwise be useless or confusing, also known as *feature creep* (Compton, 2018). To avoid feature creep, the different mechanisms must play together to give the player the opportunity to combine these into new *skills* (Cook, 2007) that improve the player's chance of success. As a player interacts with a game to complete the challenge, they accumulate skills that are often transferable to other games. Therefore, video games are continuously improving to give a better immersion, appeal, and challenge to the players. But one mechanic is included in the majority of games, but does not seem to have progressed much since the 90's: computer controlled agents (Yannakakis, 2012; Yannakakis and Togelius, 2018, Chapter 1.4). *Non-Player Characters* (NPC's) are computer controlled agents used to populate a fictive world and create interactions with a player. They can be assistants to the player's efforts (e.g., a teammate, a quest giver, or a pet), an opposition or foil to the player (e.g., a villain, a thug, or a monster), or a neutral party whose role depends on the player's interaction with them (e.g., a trader, a guard, or an animal) (Lim et al., 2012). NPC's are also confined to the same boundaries as the player, but they are often given different rules and mechanics to interact, as they do not have the same way of considering and inputting their actions as a human. As NPC's can play a major role in games, their behavior also plays an important role in the interaction with the player and the fictive world. More believable behaviors could be made through different aspects, such as emotions and personality (Lim et al., 2012), or by modelling the play styles that player's use during interaction with the fictive world (Liapis et al., 2015; Yannakakis and Togelius, 2018, Chapter 5). These different play styles can be categorized, such as *speedrunners*, *explorers*, or *treasure hunters*, which describe different ways that a player plays the game (Guerrero-Romero et al., 2018). What is important about the different play styles is that NPC's could be designed to follow similar models for playing games (Holmgård et al., 2014; Guerrero-Romero et al., 2018; Liapis et al., 2015), thereby creating agents that could improve the immersion and experience of the game by simulating the interaction between a human and the game and offering that to the actual player.

Artificial Intelligence

In order to create NPC's in a game, *Artificial Intelligence* (AI), in some form, is required to control the agents. AI has been utilized for decades, and it is widely used in modern technology. Video games especially relies heavily on AI, and they have done so since some of the first video games were created back in the 1950's, where AI was based on algorithms such as Minimax and later Reinforcement Learning (Yannakakis and Togelius, 2018, Chapter 1.2; Samuel, 1959). The integration of AI is utilized in most games, as it can control agents by giving them a certain behavior (Lim et al., 2012), allow agents to perform advanced tactics (both individually and as a group) (Barriga et al., 2019), and create re-playable experiences in games through *Procedural Content Generation* (PCG) (Smith, 2014). Among the most common AI techniques to include in games, apart from PCG, are *Finite State Machines* (FSM's) and *Behavior Trees* (BT's) to control agent behavior (Yannakakis and Togelius, 2018, Chapter 2.2.1, Chapter 2.2.2, Lim et al., 2012), while search algorithms such as A* are often used for pathfinding (Yannakakis, 2012). But even so, some developers have included more modern AI techniques in their games, though the criteria for a successful inclusion of AI in a game might not be met. A theory for this issue is that of a 'knowledge and skill gap' that exists between researchers and game developers, which keeps the game developers from using newer techniques in games: where developers aim at making experiences that a player can enjoy (Swink, 2007), AI researchers focus on utilizing new models or techniques to play, create content for, or adapt video games (Yannakakis and Togelius, 2018, Chapter 1.2.3; Perez-Liebana et al., 2019; Liapis et al., 2013). Research often attempts to create agents that can make proper decisions in a problem space to reach a goal state. For an agent to succeed, it must be able to imitate certain actions that is required for a human player, such as keeping track of different objects that are the player, enemies, collectables, etc. (Levine et al., 2013). Because of the

improvements made through the decades, NPC's have seen upgrades with newer techniques to improve their behavior and interaction within games (Yannakakis, 2012). Currently, techniques categorized as *Machine Learning* (ML) are used a lot in the General Video Game Artificial Intelligence competition (Perez-Liebana et al., 2016) to compete in playing different games without giving the agent prior knowledge about the games. The inclusion of ML techniques has also been successful in some games (Galway et al., 2008; Justesen et al., 2019; Yannakakis and Togelius, 2018, Chapter 1.2.2), though they are yet to become as common as the classic techniques. Since there is a difference in the purpose of the AI between research and actual implementation in releases, the potential knowledge gap between the two discourages the use of advanced techniques, such as ML, in the game development industry (Yannakakis and Togelius, 2018, Chapter 1.2.3). This is partly due to the internal working of ML agents, as they often require learning before they can act in a certain problem space. For instance, *Artificial Neural Networks* (ANN's) is a ML technique that relies on training, where the network is trained to perform in a certain way within a specific problem space (Yannakakis and Togelius, 2018, Chapter 2.5; Yadav et al., 2015, Chapter 3.7). This is a technique that was proposed in the 1940's and has been explored ever since as a way of solving problems in specific problem spaces (Yadav et al., 2015, Chapter 2). ANN's work by approximating towards an optimal solution by emulating the biological brain's way of learning (Yadav et al., 2015, Chapter 3). Mostly, ANN's are trained using supervised learning with a training set of data, but they can also be trained using reinforcement learning or unsupervised learning (Justesen et al., 2019; Samuel, 1959; Yadav et al., 2015). However, a shipped game, that contains ANN controlled agents that learn over time, might end up having agents that act in weird ways, thereby making the game experience unconvincing and unsatisfactory (Galway et al., 2008). This can make those types of ANN's undesirable for game developers, as they cannot ensure the agent's behavior and the game experience. A variation of ANN's that have gained a lot of popularity in the past decade is *Convolutional Neural Networks* (CNN's). CNN's are a special kind of networks that are useful for finding patterns in a certain context, e.g images, videos, or voice recordings, by analysing and extracting the important informations before running the information through a neural network (Albawi et al., 2017). As these networks can be used to find certain patterns by analysing an input, they could be used in a game, by treating the game like an image (Stanescu et al., 2016; Barriga et al., 2019), where specific features can be extracted and analyzed, instead of passing everything through the network. By giving the CNN a representation of data that reflect the current state of the game, it could be used to control the agent, and even succeed in performing as well as players (Mnih et al., 2015). Furthermore, the CNN would be able to analyse the game in two different ways: by feeding it the whole screen as an input (Mnih et al., 2015) or by giving it the map as an input (Stanescu et al., 2016; Barriga et al., 2019). These different options give the possibility of creating an AI controlled agent that can act in a world only given the same information as a potential player, thereby making its behavior more realistic, and forcing it to play by the same rules as the human player.

Methods

This section goes into more detail of some of the AI techniques discussed in 3.2, how they were used in the project, and the considerations related to the implementation of different AI and ML techniques used for the project. Along with the considerations, highlights of important functionalities are also displayed and explained. The full Unity project can be accessed at bias2402 (2021d).

The Game

In order to test the different artificial intelligence (AI) methods chosen for the project, a game was required, both for gathering data to train the chosen machine learning (ML) techniques, and for seeing how the agents perform after implementation and training. To keep the number of functionalities low (Compton, 2018) and stick to a simple system of rules and mechanics (McGonigal, 2011; Burgun, 2015), a simple rogue-like game with randomly generated maps (Smith, 2014) was made (using an adapted, previously made, map generator (bias2402, 2021b)). Using a rogue-like game for testing AI is a common practice (Holmgård et al., 2014; Liapis et al., 2015; Cerny and Dechterenko, 2015), as they are simple to design and collect data from, yet complex enough for adding different possible play styles. Through adding different objectives to the game, data about different personas can be extracted (Guerrero-Romero et al., 2018), and specifically for this project, three personas were chosen:

- *Speedrunner*: reach the goal as fast as possible
- *Explorer*: explore as much as possible
- *Treasure Hunter*: find all the treasures in the map

These personas fulfill different roles that covers main ways of playing games, where the speedrunner focuses on reaching the goal, the explorer wants to see everything, and the treasure hunter wants to find everything (Guerrero-Romero et al., 2018). Having these three personas covers the full usage of the map, as both players and AI's would require considerations about their moves to complete the current objective.

The Map

To create a proper, though random, test area, a map of 20 by 20 blocks is randomly generated every time the game is started. The process of generating the map consists of five steps (figure 1 shows an example of a randomly generated map being traversed by the agent).

1. Create the border.
2. Position the spawn and goal.
3. Generate a path from spawn to goal using weighted randomness.
4. Fill in the map with platforms and lava.
5. Select five random platform blocks and replace them with treasure blocks.

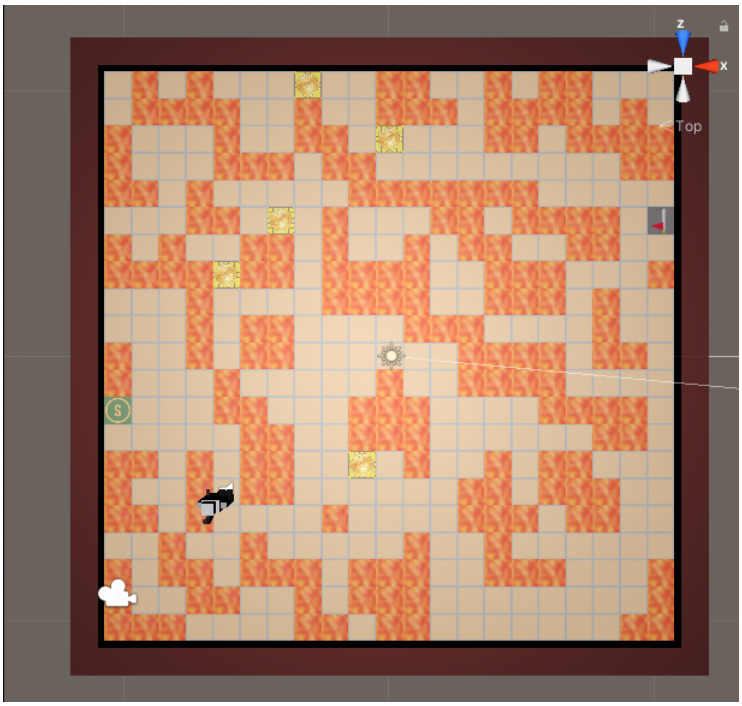


Figure 1: A randomly generated map that is being traversed by the agent

The map consists of five different blocks (figure 2) that are used to generate the main area of the game, while the area is surrounded with a border to confine the agent. The agent starts at the spawn block (always placed randomly on the far left side), and it must find its way to the goal block (always placed randomly on the far right side) along the platforms, while avoiding the lava blocks. If the agent steps into the lava, it will

(a) Spawn (b) Goal (c) Lava (d) Treasure (e) Platform

Figure 2: Blocks used for the walkable area of the game

be reset back to the spawn, but no further punishment is inflicted. This decision plays two roles: it does not hinder the AI's nor player's process, and it can be seen as a hidden mechanic that can be utilized during gameplay to get back to spawn (Cook, 2007). Lastly, there are the five treasure blocks, which randomly replaces platform blocks after the map is generated. These are special blocks that are required by one of the personas (described in Training Data Collection) for completing the level, but are otherwise just walkable blocks.

The Agent

The agent is represented by a block figure that can be controlled either by a human player or an AI. Since the game is based on the design of a rogue-like game, the agent's direction is indicated my a mining pick that it carries in one hand. This was originally meant to be used for another mechanic, where the agent would have to break boulders on the map, but this idea was scraped, as it was deemed irrelevant for the gameplay (Compton, 2018). To make the game harder for the human players, the camera was connected to the agent, so it follows the agent around, and only ever allows visibility of 11 by 11 blocks. This also played a role in the AI implementations (described below), as the AI's are supposed to work with the same information the player was given. Therefore, any search algorithms used were limited to the visible area, and ML networks would only gain information about the current area. The agent was created with one major script that handled inputs, movement, network training, and translation of AI decisions to actions. Using Unity's inspector window, all the different options and setups for the agent were available to the same agent, while the script would differentiate between what parts to use. To differentiate between the different agent control types, a control option called *Agent Type* was added to the top of the agent settings (figure 3). The different types available are:

- Human: used when a human is controlling the agent.
- Playback: used to remake the map from a data set and play back the recorded actions (see Training Data Collection for more information).
- BT: enables control using a behavior tree.
- FSM: enables control using a finite state machine.
- ANN: enables control using an artificial neural network.
- CNN: enables control using a convolutional neural network.

Figure 4 is an example from the agent's script, where the movement handling is separated based on the agent type. Apart from the agent type setting, different settings for the agent itself were added to easily connect and adjust the agent for each specific situation. Important settings were separated into sections in the inspector as AI, ANN, and CNN settings. They can be seen in figure 3. The ANN and CNN contain the different files and scriptable objects (Unity, 2021) that were used to configure and save data from the networks (described in Network Support Classes), while the AI section had the options for toggling training and training emulation. Training was required for the networks, but the initial implementation was time consuming, as the map must be recreated and the agent had to playback the actions recorded in the data

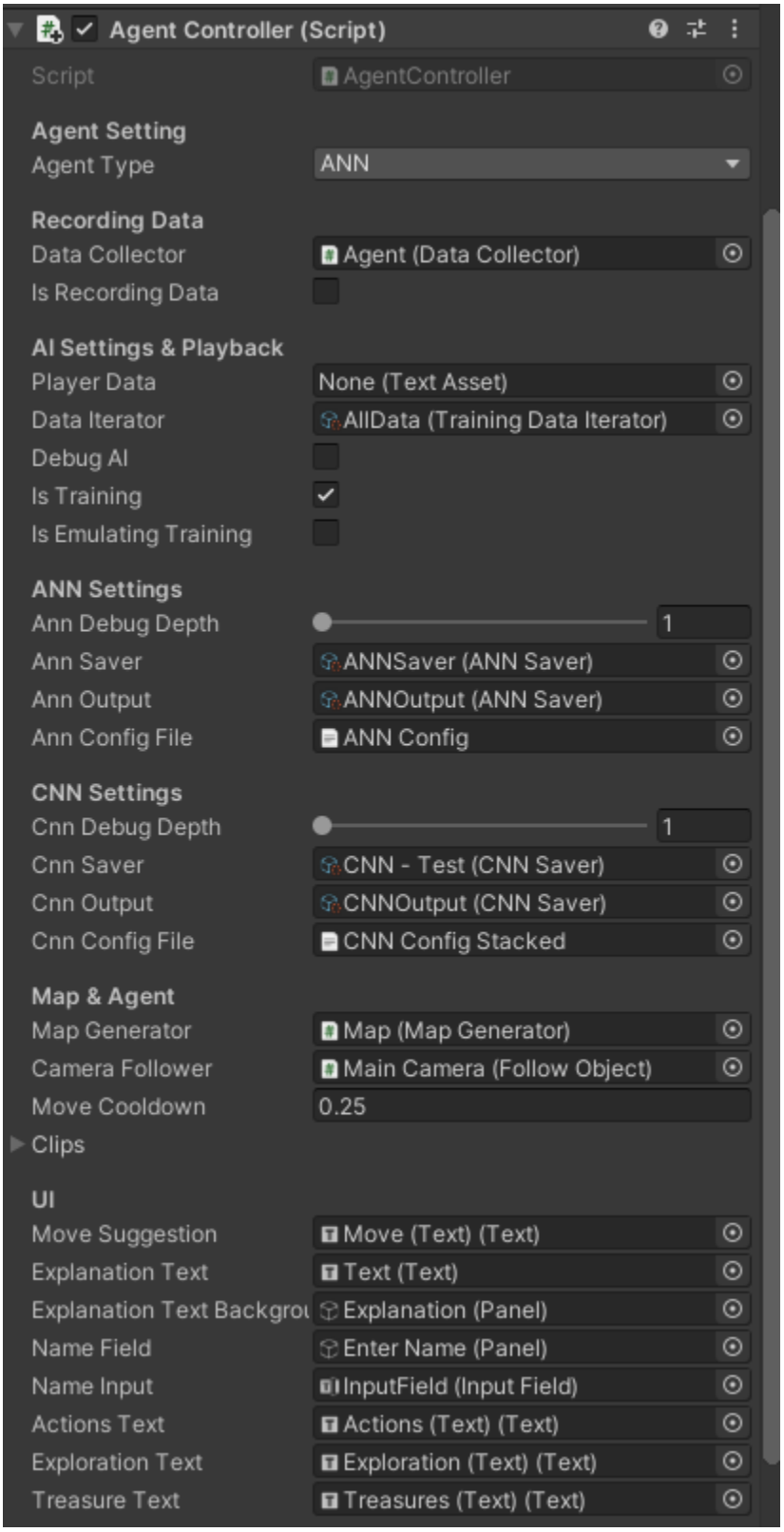


Figure 3: Inspector in the Unity Editor, here showing the settings for the agent

```
void AgentMovementHandling() {
    switch (agentType) {
        case AgentType.Human:
            if (Input.GetKey(KeyCode.W)) {
                WalkForward();
                steps++;
                actionsText.text = "Actions: " + steps;
                if (isRecordingData) dataCollector.AddMoveToRecords("w");
            }
            if (Input.GetKey(KeyCode.S)) {
                WalkBackward();
                steps++;
                actionsText.text = "Actions: " + steps;
                if (isRecordingData) dataCollector.AddMoveToRecords("s");
            }
            if (Input.GetKey(KeyCode.A)) {
                TurnLeft();
                steps++;
                actionsText.text = "Actions: " + steps;
                if (isRecordingData) dataCollector.AddMoveToRecords("a");
            }
            if (Input.GetKey(KeyCode.D)) {
                TurnRight();
                steps++;
                actionsText.text = "Actions: " + steps;
                if (isRecordingData) dataCollector.AddMoveToRecords("d");
            }
            break;
        case AgentType.ANN:
        case AgentType.CNN:
            if (isTraining) Playback();
            else {
                FeedDataToNetwork(null);
                PerformNextMove();
            }
            break;
        case AgentType.BT:
            btHandler.Execute();
            PerformNextMove();
            isReadyToMove = false;
            break;
        case AgentType.FSM:
            FSMExecution();
            PerformNextMove();
            isReadyToMove = false;
            break;
        case AgentType.Playback:
            Playback();
            break;
        default:
            throw new System.NullReferenceException("AIType not properly set!");
    }
}
```

Figure 4: Movement handling of the agent, based on the current agent type setting

set to train the networks. To improve this process, an emulator was implemented, which will not playback the actions visually (it will recreate the map though), but instead playback the actions and simulate the map in pure code without any visual handling. This reduced training time from around 15 minutes to less than one minute for each network. Section ANN Re-Testing goes into more details between live training and emulated training.

Finite State Machine

Between the AI techniques used in games, the inclusion of Finite State Machines (FSM's) is the simplest and most common technique. FSM's utilize three main components for controlling the agent: *states*, *transitions*, and *actions* (Yannakakis and Togelius, 2018, Chapter 2.2.1). FSM's are easy-to-implement graphs that can give agents convincing behaviors through states and transitions. Each state contains a behavior consisting of different actions, which are executed while the agent stays in that state. Depending on the internal and external conditions, the agent can transition to other states, which gives an idea of the agent being intelligent.

FSM Implementation

The FSM implementation used for the project consisted of only three states that were used to traverse the map:

- **Walk Forward**: This state checks for lava in front of and around the agent, walks forward when possible, or transition to one of the other states depending on the blocks around the agent.
- **Turn**: If there is lava in front of the agent, but not around it, transition to this state and turn left or right, depending on where the lava is. Transitions back to Walk Forward.
- **Walk Backward**: If the agent is surrounded by lava, transition to this state to step back and find another direction. Transitions back to Walk Forward.

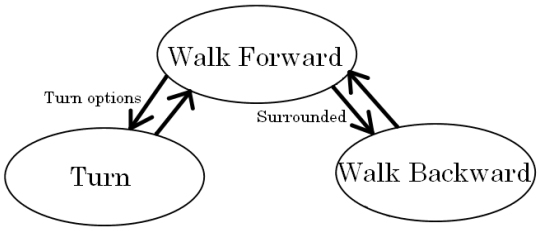


Figure 5: Simple finite state machine that were used by the agent

Figure 5 shows the FSM that was used and an example of the agent being controlled is seen in Jensen (2021d). The FSM was made with only a few states in mind, as the agent only has four different actions to perform: walk forward, turn left, turn right, walk backward. The Walk Forward state is the main state, where the agent stays most of the time. It contains actions to determine whether to walk or transition to another state (the small arrow texts in 5 are the determiners for transitioning) based on the block in front and around the agent. If its path is blocked by lava, it will check for turn options, and if there are none, it will walk backward, as that is the only viable option. Since the FSM is quite simple, the agent would easily get stuck in parts of the map, so to make the agent more convincing, a random chance to turn towards the goal, if possible, would happen after a certain number of forward actions (unless it was walking that way already), and it also got a small memory of the previous five actions to avoid following some patterns continuously and getting stuck.

Behavior Tree

An issue with FSM's is that they can easily become predictable and they can be hard to expand as a game evolves. This can be a problem for players, as it can destroy the immersion of the game. Therefore, an alternative to FSM's is Behavior Trees (BT's), as they have a more dynamic work structure and allows for complex behaviors that can be expanded throughout the development. Where the construction of an FSM is similar to a graph structure, BT's utilize a tree structure, where execution starts at the root (which is usually depicted at the top), traversing down through the tree using the logic of each node that is visited (Yannakakis and Togelius, 2018, Chapter 2.2.2; Simpson, 2014). According to Yannakakis and Togelius (2018), BT's consists of three different nodes: *sequence*, *selector*, and *leaf*. However, according to Simpson (2014), BT's consists of three different types of nodes instead:

- **Composites**: The logic for traversing the tree, as they execute their child nodes based on the results that are returned to them from their children.
- **Decorators**: Can only have a single child, but allows for transformation of the result from their child, such as inverting the child's result.
- **Leaves**: Actions the agent should perform whenever the leaf is reached during the execution (frame). These can complete and return their result in a single call, or keep running if the action is performed across multiple executions (frames).

Composites are different types of logic that guides the traversal through the tree. Two of the nodes described by Yannakakis and Togelius, the sequence and selector nodes, are categorized as composites by Simpson, while leaves are the same in both descriptions. Furthermore, in Simpson's description the decorator type includes nodes that can adjust the output of a single child node, when the child node returns its result to the decorator. This enables the inclusion of nodes to change the traversal of a tree, without changing the behavior of the leaves. For this project, the implementation uses Simpson's definition of tree nodes, and the implementation will reflect the three different categories. It is important to note that there are discussions and descriptions online about other composite and decorator nodes (Simpson, 2014), but they are excluded in this project.

BT Implementation

The implementation was based on a pre-made Unity asset (bias2402, 2021a) for creating and executing BT's. This asset contains the following nodes, which were used for the implementation:

- Selector: A composite node that executes all of its children until one returns true, making the selector return true, or if the last child returns false, it makes the selector return false. The selector is used to execute actions until one succeeds or all fails.
- Sequence: A composite node that executes all of its children until one returns false, making the sequence return false, or the last child returns true, making the sequence return true. The sequence is used to execute an order of actions, as long as they all succeed.
- Inverter: A decorator that inverts the result received from its child, returning the inverted result to its parent. The inverter can be used to change the execution of its parent, depending on the result of its child.
- Succeeder: A decorator that always returns true no matter the result of the child. The succeeder can allow the child to execute and fail without affecting the continued traversal of the tree.
- Leaf: The lowest parts of the tree, leaves contain the actions to perform, whenever they are reached. They will return true or false depending on the success of their action, and can require multiple executions of the tree to complete their action. The tree will follow the same order of nodes down to a leaf each execution if the leaf is not done with its action.

The tree was created using a window in the Unity Editor, where the connections and execution order was created (Figure 6). The asset does not indicate the execution order, but the node connections were made from left to right, thereby creating the order of execution. Every time a new child was added to a node, all connections to children were removed and re-added to ensure the execution order.

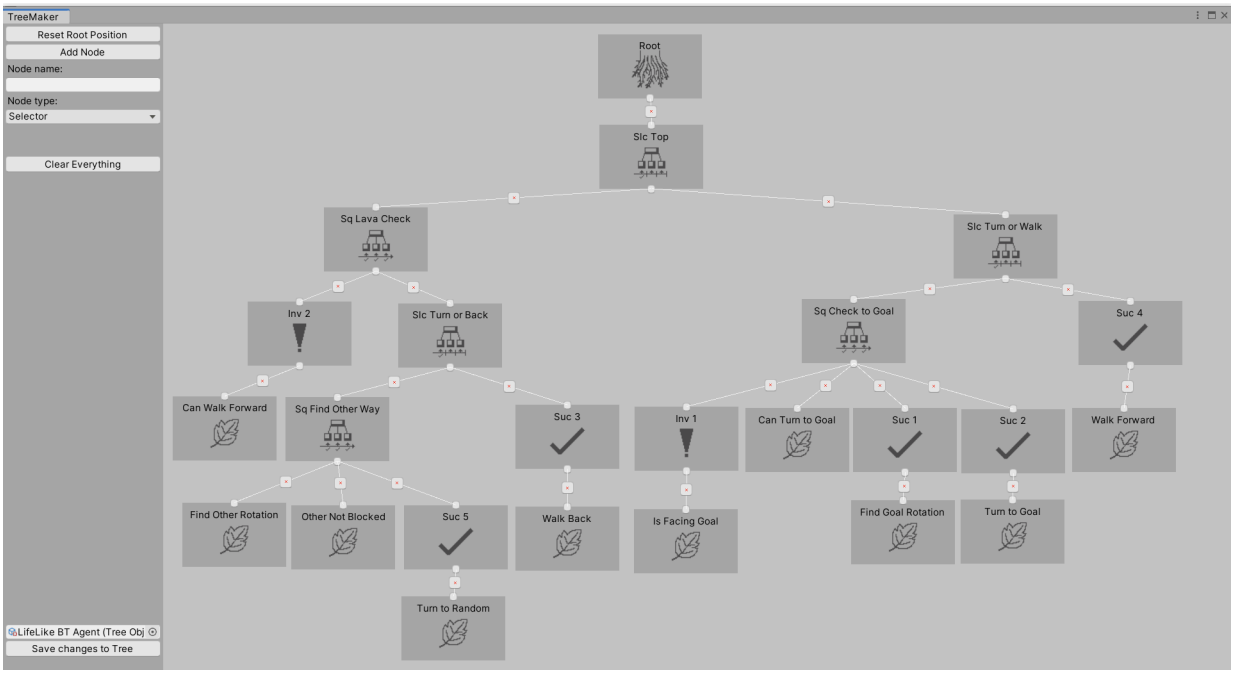


Figure 6: Behavior tree maker in Unity, and the tree used by the agent

As with the FMS, the BT would get stuck at times. Since the BT has a more complex way of making decisions, its turning implementation was enhanced with a bit of map analysis using a *Depth First Search* (DFS) algorithm. As to not give the BT agent an unfair advantage (though the FSM did not have any assistance from a DFS), the DFS was limited to the visible part of the map, because using a search algorithm to traverse the map would make it obvious that the agent is controlled by AI, since it knows where to go even with limited vision. Instead, the DFS algorithm would check the different turn directions from the current position, but only up to five blocks away. This number was chosen, as that would keep the search space within the vision space of the camera (the visible area is 11 by 11 (A randomly generated map that is being traversed by the agent), so five blocks in each direction would reach the edge). Using the DFS allowed the agent to pick better routes around obstacles, but the agent received another enhancement in the form of a feature similar to the FSM's random chance to turn towards the goal. However, the BT agent's version would not randomly turn towards the goal, instead it would always turn towards the goal when possible. These smaller improvements would in theory make it better at traversing the map, which was decided as a good comparison between the two implementations.

Artificial Neural Networks

Artificial Neural Networks (ANN's) are general purpose functions that can be adapted through their graph structure and internal settings, allowing for approximations towards optimal solutions to a variety of problems within the same problem space (Justesen et al., 2019). Based on biological neural networks, ANN's consist of *neurons* that are connected through *connections* similar to biological synapses. The neurons receive *input values* from other neurons, multiplying them with a corresponding connection *weight*, summing the products to a *net input*. The product is adjusted using a *bias*, after which the result is sent through an *activation function*, resulting in the neurons *output value* that can be passed on if the value is above a certain threshold (Yadav et al., 2015, Chapter 3.3). What makes ANN's excellent for finding solutions in a problem space, are their ability to learn by updating the values of the neurons and connections. Note that

the bias can be seen as either an extra weight that is applied to the net input to offset the output (Yadav et al., 2015, Chapter 3.4), but it can also function as the neuron's threshold (Yannakakis and Togelius, 2018, Chapter 2.5.1). This ANN implementation uses the former idea, where the bias is used as an internal weight. Furthermore, the bias can be an overall weight applied to each neuron of the network instead of a value inside each neuron, but here each neuron has its own bias. As ANN's are able to learn by updating the internal values of neurons and connections, all weights and biases of the network can be updated during the learning process, thereby changing their impact on future calculations. The way ANN's learn is through different algorithms, the most common being the *backpropagation* algorithm, which is based on gradient descent (Yannakakis and Togelius, 2018, Chapter 2.5). Backpropagation calculates the *cost function*, also called the network's *error*, based on a given desired output and the network's actual output. The error, along with a given *learning rate* for the network, is then used to update the weights and biases of the network by backpropagating through the network, starting from the output layer, going through all hidden layers, but ending just before reaching the input layer (the input layer passes the inputs without any calculations, so it does not need any updates). The goal of the algorithm is finding the global minimum for the error, and by applying gradient descent throughout the network using the chain rule, every value associated with the network's output can be adjusted by a small amount to move the network closer to that minimum (Yannakakis and Togelius, 2018, Chapter 2.5; Yadav et al., 2015, Chapter 3.8.1).

ANN Implementation

The ANN implementation used a Unity asset that was completed prior to the project start (bias2402, 2021c), which consists of a system of classes that hold the different informations: an ANN class, a Layer class, and a Neuron class. When the ANN is made, it will be given the necessary informations about the network topology, so it can create all the necessary layers, each with the right number of neurons, and ensure the hidden layers and the output layer gets the right activation functions (figure 7, 8, 9). Note that the asset only allows the creation of a feed forward ANN (Heaton, 2005, Chapter 5), where each neuron of each layer is connected to each neuron in the next layer.

```
26 references
public class ANN {
    private int epochs = 1000;
    private double alpha = 0.05;
    private List<Layer> layers = new List<Layer>();
    private static Random random = new Random();
    private bool isDebugging = false;
    3 references
    public List<double> inputErrors { get; private set; }

    /// <summary> Create a new ANN consisting of an input layer with nInputsN neuron ...
    2 references
    public ANN(int nInputsN, int nHiddenL, int nhiddenN, int nOutputN, ActivationFunctionHandler.ActivationFunction hiddenLAF,
        ActivationFunctionHandler.ActivationFunction outputLAF, int epochs = 1000, double alpha = 0.05) ...

    /// <summary> Create a new ANN based on the given annConfig
    3 references
    public ANN(Configuration.ANNConfig annConfig, int nInputNeurons) ...

    /// <summary> Creates a new ANN from an annString by deserializing the ANN
    2 references
    public ANN(string annString) => DeserializeANN(annString);

    /// <summary> Enables debugging. This method should never be called from anythin ...
```

Figure 7: The ANN class showing its internal values, list of layers, and its different constructors used to create or recreate a network

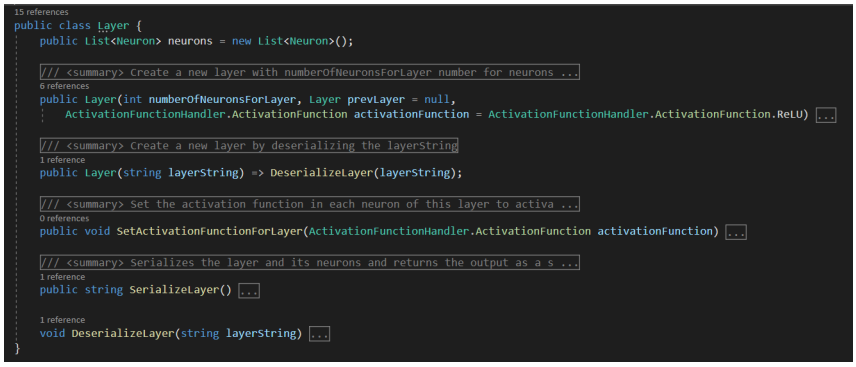


Figure 8: The layer class, showing its internal list of neurons

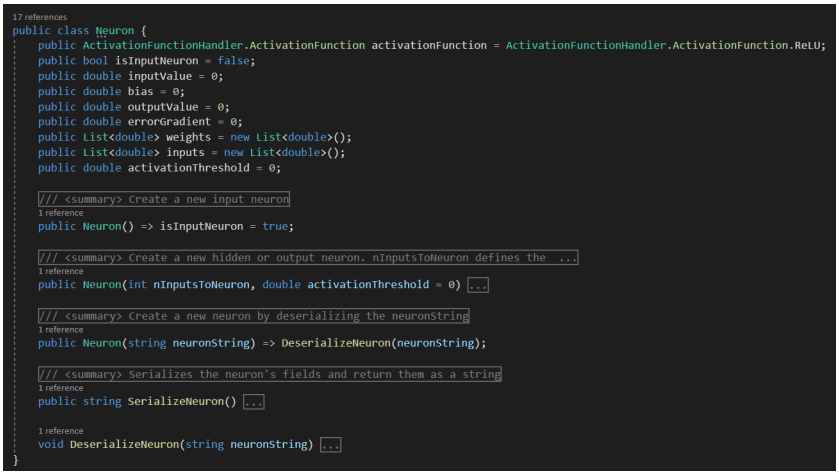


Figure 9: The neuron class, where all the internal values a neuron can contain are shown

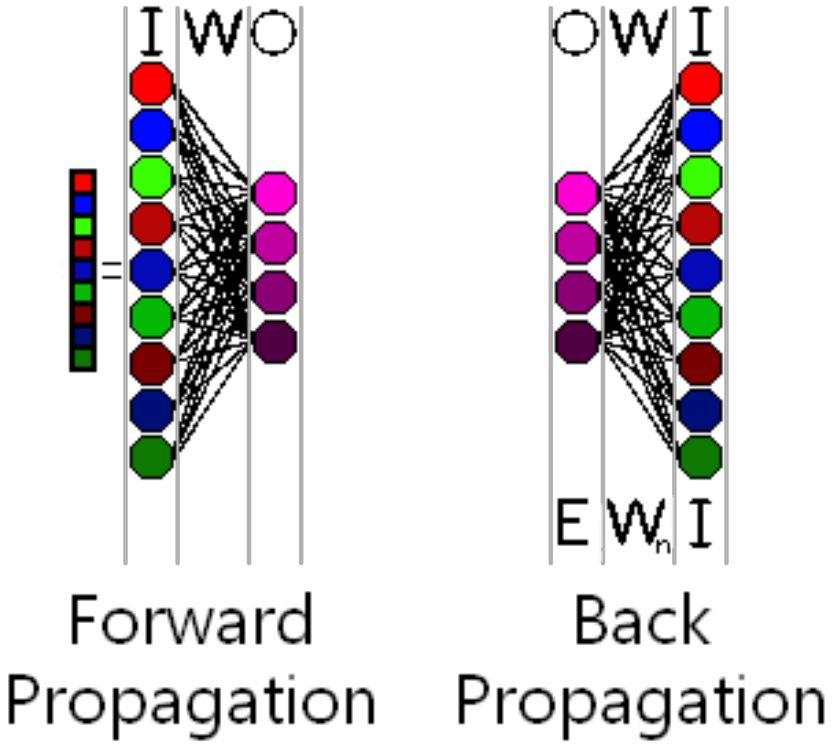


Figure 10: Visualization of the propagations that happen in an ANN between the input and output with no hidden layers

The network contains different methods for running or training the network, training methods being able to calculate, train the different weights and biases, and return an output, whereas the running methods will only calculate and return an output. Each of these methods would run for a set number of times, defined by the *epochs* value seen in figure 7. Some of the methods would use the preset value, while others would allow parsing an epochs value to this specific training session. When the network's training was performed, it would use the backpropagation algorithm. Figure 10 illustrates the forward and backpropagation processes in an ANN, where the processes are color coded to show the relation between a 3x3 2D map, how the values are passed to the network, and how the network calculates the output. The mathematics for calculating the output is described in Yadav et al. (2015, Chapter 3.4). Figure 10 illustrates the map being converted to 1D, then parsed to the input layer (I), which is connected to the output layer (O) through the connections (W) that each contain a weight. When the backpropagation occurs, the process starts at the output layer (O), where the error (E) is found. The weights of the connections (W_n) are then calculated, before the process reaches the input layer (I) and stops. During the backpropagation, the adjustments are controlled using a variable called *alpha* that adjusts the learning rate of the network (can also be seen in figure 7). A high alpha value allows the network to adjust the neuron values faster, but also risk *over-fitting* the values. Over-fitting of values occurs when the values are changed too much during the backpropagation, which leads the network to performing worse or not change at all. Similarly, *under-fitting* is also a problem that might occur if the alpha is too low. As figure 7 shows, the alpha is set to 0.05, which was the base value used throughout the project. In Heaton (2005, Chapter 6), the learning rate is discussed as a value that should be kept below 1, usually around 0.5. However, the value of 0.05 was chosen based on prior experience working with ANN's and its effect on the calculations, and it was not considered further until after the comparison test (see ANN Re-Testing). Apart from the learning rate, the ANN's topology also plays a major role in its success, as more neurons allow for more calculations, which can result in more precision over time. The topology of the ANN can have an impact on the functionality of the network, so having more hidden layers can influence the final result. However, more hidden layers does not mean a better result. The number of hidden layers, and the number of neurons for each layer, is hard to set, as there are no rules determining the proper number. However, there are rules of thumb that can assist, though mostly, the decision is based on knowledge about the necessity of hidden computation and trial-and-error (Heaton, 2005, Chapter 5). This is discussed in more detail in Agent Comparison Test. It is also important to note that the ANN used *discrete* activation instead of *continuous* activation. The difference between the two being that continuous activation allows the network to continuously gather information and change its decision. Because the game was based on the rogue-like genre, the ANN (and in extension, also the convolutional neural network) were not able to use continuous activation, as the game would accept an input, move the agent a specific amount, wait for a certain delay, then await the next inputs. This forced the ANN to only make decisions whenever it is given the permissions.

Convolutional Neural Networks

The Convolutional Neural Network (CNN) is a neural network that is often used in recognition software for images, videos, voice, etc., as it can learn to find specific patterns. Considering the use case of image recognition, the project included a CNN to try and analyze the game like an image, as that could potentially provide the necessary information for controlling the agent. CNN's work by utilizing *filters* and different processing layers to perform a feature extraction from an input (Stanescu et al., 2016). CNN's do not have a specific architecture (CNN's have an architecture of processing layers in contrast to the ANN's topology of neuron layers), so different ideas were considered, from the classic setup of a CNN (O'Shea and Nash, 2015), to a more complicated system using branching (Li et al., 2017), where different branches could handle different input extractions, to utilizing stacking of layers, as there are examples for this type of architecture being used to analyse game states (Stanescu et al., 2016; Barriga et al., 2019). Of the three different types, the classic and stacked architectures were chosen, as those were simpler to work with, compared to the branching architecture. To keep the implementation even simpler, it only allowed the use of 2D filters and 2D input maps. Lastly, just like neurons in an ANN, the different layers of the CNN can contain a bias, but that was not included for this implementation. The power of the CNN comes in its processing layers, which are different layers that perform different actions to extract necessary information from an input. Here the chosen layers were kept to the most common types (O'Shea and Nash, 2015):

- Padding Layer
- Convolutional Layer
- Max Pooling Layer
- Fully-Connected Layer

Furthermore, the order the layers are shown is similar to the order the layers are usually executed, as the padding, convolutional, and max pooling layers perform the data extraction, while the fully-connected layer consists of an ANN for data analysis and decision making. However, a CNN's architecture can be changed around almost freely, though it should end with a fully-connected layer for decision making, while the data extraction layers can be mixed and weaved. This project focused on the basics of the CNN, but there are many different layers and operations that could had been included instead and should be considered in future, similar projects (Khan et al., 2018, Chapter 4).

Padding Layer

The padding layer functions by adding a layer of zeros around the map, increasing both dimensions by two. This is illustrated in figure 11 with the green border of zeros added to the map. This layer enables the network to detect important features along a map's edges, as adding an outer edge of unimportant data allows the network to find important data on the former edge that might otherwise be missed during convolution. When the padding has been added, the map is sent to the next layer.

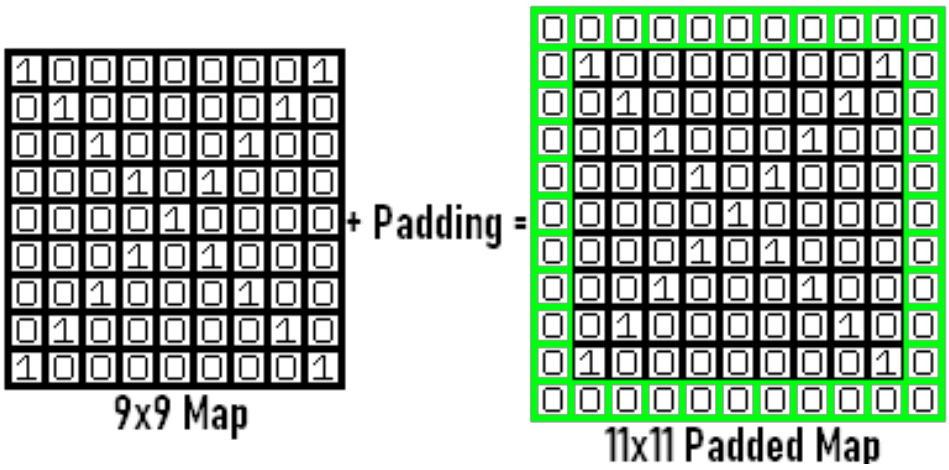


Figure 11: Visualization of the padding layer's operation

Convolutional Layer

The convolutional layer is used to apply filters on the input to search for important features. The process is performed by moving a *kernel* across the map and applying the filters on the input map to compare values and generate a new map. The filter is sometimes called a kernel (O'Shea and Nash, 2015), but to make things simpler, these are regarded as separate things in this context: the kernel is a lens that is moved across the map, while the filter is the information holder that is applied through the kernel (figure 12, 13).

The kernel is moved using a set *stride* value (Albawi et al., 2017), which determines the number of positions to move after applying the filter (the default value of the stride is one). For each position of the kernel, the values of the filter are multiplied with the value on the map below, after which the products are summed up as the feature value for the new map. This is illustrated in figure 13, where the kernel (green) is placed on the input map (black), a filter (blue) is then applied through the kernel, and the operation calculates the value for the new map (red). The operation is usually depicted as starting in the upper left corner, moving horizontally to the right until it reaches the end, after which it jumps back to the left and one

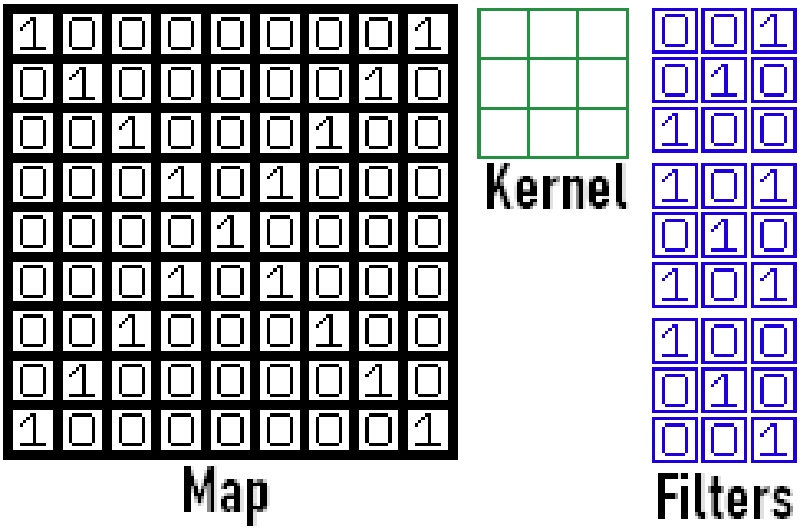


Figure 12: Illustration of the map, kernel, and filters as different things

down. This is the method used for this implementation, and how all illustrations of CNN processes should be seen.

To better describe the convolutional layer's process, figure 14 illustrates the process as sending the input through a neuron of an ANN (similar to the illustrations in Agarwal, 2017). Here, the relation between the input map, the filter, and output map is illustrated with the filter's values being the weights going to the neuron that are multiplied with the values of the input map. The neuron can be seen as having nine connections with weights, but in this neuron, each weight is used multiple times. Looking at the input map, most values are colored using multiple of filter's colors to indicate which filter values are multiplied with that input value. The colors show that the corners of the input will only be multiplied with the filter's corners, while the the center value will be multiplied with each of the filter values. This is important to note, as this illustrates how features near the center and more likely to be extracted than features along the edge, which makes the possibility of adding padding important.

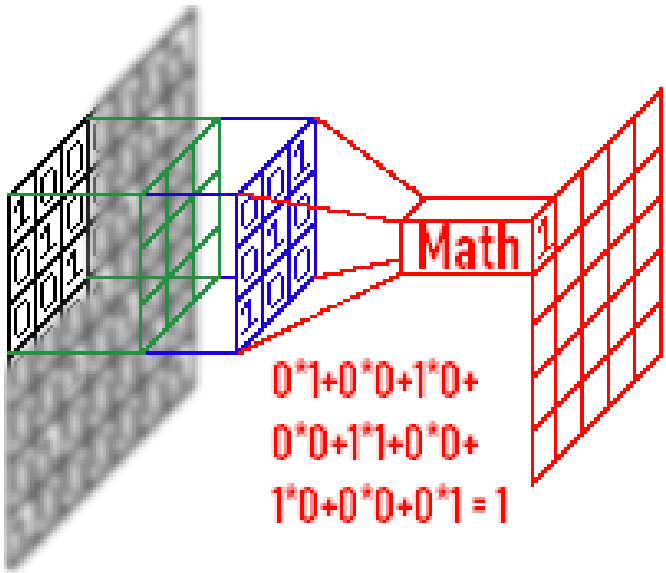


Figure 13: Visualization of the convolutional layer's operation

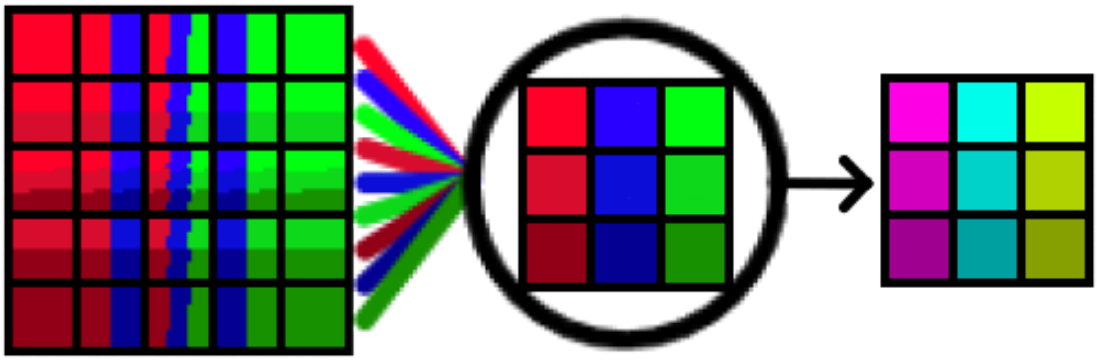


Figure 14: Visualization of the relation between an input map and each value in the filter map

Furthermore, figure 13 and 14 also illustrate that the new map that is generated is smaller than the original; the new map's dimensions are both two smaller. This is due to the convolutional operation, as each kernel position (here being 3x3) will result in one value on the new map (as figure 13 illustrates). The size of the output map changes based on the settings of the convolutional layer, where the dimension of the input, the dimension of the filter, and the stride plays a role. The output map's dimension can be calculated using equation 1 (Albawi et al., 2017):

$$O = 1 + \frac{N - F}{S} \tag{1}$$

where O is the output map's size, N is the input map's size, F is the filter's size, and S is the stride. It is important to node that the equation only works for 2D, rectangular maps and filters. If the input map contains padding, the equation becomes:

$$O = 1 + \frac{N + 2P - F}{S} \tag{2}$$

where P is the padding, and it assumes only one layer of padding is added. Due to the dimensions of the kernel, and the stride that is used to move it, if equation 1 does not return a whole number, the operation will skip over data from the maps. Working with a stride of one, this will never occur, but strides that are higher than one might result in missed information.

When the convolutional layer has completed its operation, it will have a new set of maps, one for each filter that was used. Each value from the new maps will then be sent through an activation function, similar to how neurons calculate their value and apply an activation function in an artificial neural network. Afterwards, the maps will be sent to the next layer.

Max Pooling Layer

The max pooling layer works similar to the convolutional layer, but instead of having filters, it only has a kernel which is moved across the given map. As there are no filters, instead of the convolutional layer's mathematical operation after each stride, it selects the highest value found inside the current kernel area and saves it to a new map. The kernel's default size is often set to 2x2 and its stride to two, where the most important data is saved, while the map's overall size is reduced by two in both dimensions. The point of this layer is down sampling of the map, where the important data is kept, and less important data is discarded. This allows the next layers to work more efficiently, since the important data is given, while also reducing the computational time required as there are less inputs to compute. Similar to the convolutional layer, the max pooling layer also follows equation 1 when generating the new map, but instead of the F being the filter dimension, it is the kernel dimension (since they can be seen as the same thing in some contexts, while in this context, the filters are applied through the kernel, so they must have the same dimensions).

Similar to the convolutional layer, after completing its operation, the values of each new map will be sent through an activation function. However, the max pooling layer does not create more maps, instead it outputs the same number of maps as it was given, which are sent on to the next layer.

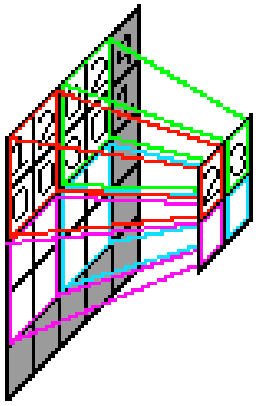


Figure 15: Visualization of the max pooling layer's operation

Fully-Connected Layer

The fully-connected layer is an artificial neural network (ANN), where every neuron in each layer is connected to every neuron in the next layer: a feed forward ANN. Illustrations of the fully-connected layer often depicts it as being only an input layer connected to an output layer (Stanescu et al., 2016), but it can also contain hidden layers. When the fully-connected layer is run, it will use the given maps from the previous layer, extract their values, and pass them to the input layer, where each value in each map will be passed to an input neuron. The layer will then perform the forward propagation as normal.

CNN Backpropagation

Similar to the ANN, the CNN requires a way of learning, and here the backpropagation algorithm can be used again (Jefkine, 2016; Skalski, 2019). As the fully-connected layer consists of an ANN, the backpropagation of this layer is performed using the ANN implementations backpropagation. From there, the backpropagation requires extraction of the error in relation to the original inputs, as to backpropagate through all the layers and do the different updates required. Figure 16 illustrates this, and it shows how similar it is to the propagation from the ANN, but it also shows how the error is extracted from the ANN (the fully-connected layer). Equation 3 describes the extraction process, where each updated weight going to a specific input neuron are multiplied with the neuron's input value, after which the products are summed up, and finally sent though the derivative activation function for the layer connected to the input layer (could be either a hidden layer or the output layer). The resulting values are then stitched together to form the same number of *error maps* passed to the fully-connected layer, with each value having the same index as the related original value (Arbel, 2018).

$$f'(I \cdot \sum W_{n,I}) \tag{3}$$

The generated error map(s) is passed to the previous layer, and depending on the architecture of the network, the backpropagation process handles the error map in different ways. Using the classic CNN as an example, the error map would be passed from the fully-connected layer to the max pooling layer. The max pooling layer performs backpropagation by copying its original input map, zeroing all the values in the map, except the ones chosen during forward propagation, to create an *error mask*, and insert the error map values in the positions they were picked. Figure 17 shows the backpropagation for the max pooling layer, which can be seen as opposite to the forward propagation illustrated in figure 15.

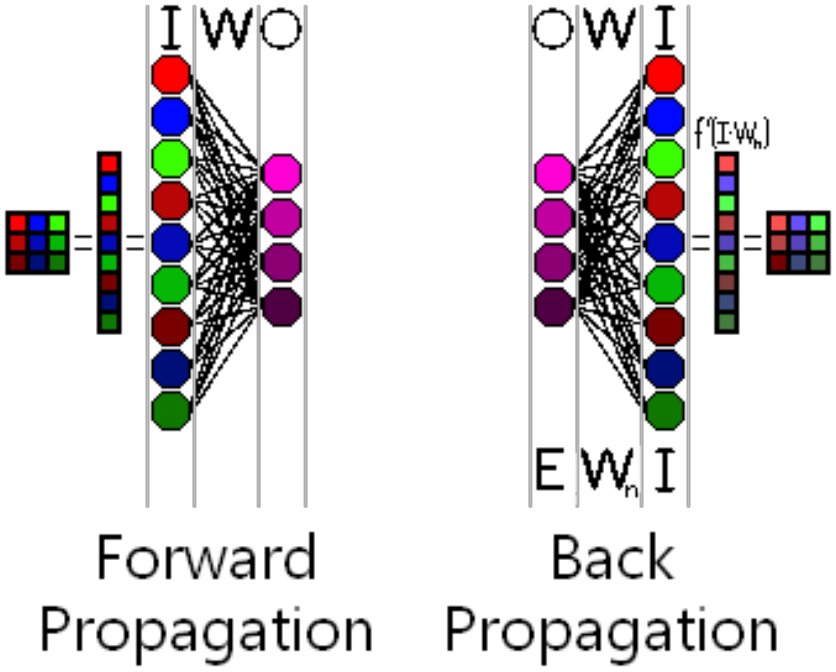


Figure 16: Visualization of the forward and backpropagation in a CNN's fully-connected layer

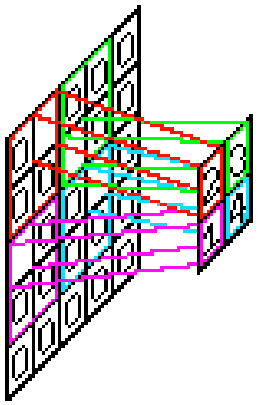


Figure 17: Visualization of the backpropagation in a CNN's max pooling layer, where the error map value are inserted in the error mask

Lastly, the error maps, generated from the error masks, are passed to the convolutional layer, where they are used to update the filter values. As each filter contains a map that is used to extract features, updating these filters is the actual learning step of the network, similar to the weight updates in an ANN.

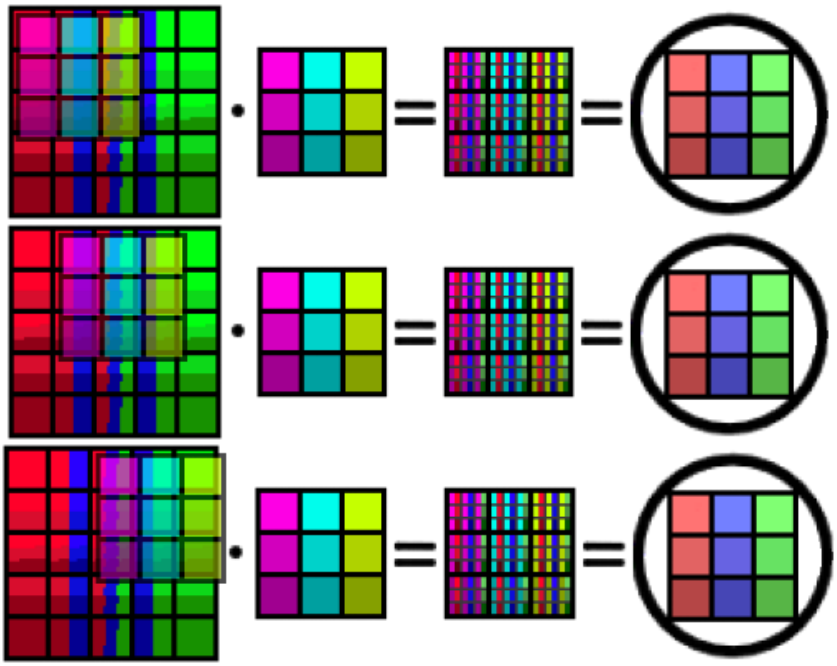


Figure 18: Visualization of the backpropagation in a CNN's convolutional layer, where the error map is used to perform the convolutional process as to update the filter

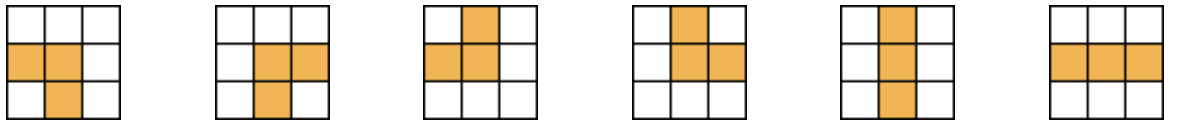
As figure 18 illustrates, to perform the filter update, the error map is used to perform the convolutional operation used in the convolutional layer (see Convolutional Layer). This generates a map of delta values, and similar to figure 14, the delta map is a mixture of colors to represent the multiplications that are performed between the input map and the error map. After generating the delta map, it is used to update the filter values by adding the value from a specific position in the delta map, to the filter value in the same position (the index must be the same for both the delta map and the filter). After completing this operation with all the errors maps, the backpropagation is complete.

If the padding layer was used before the convolutional layer, it is skipped for the backpropagation, as there are nothing to update in the layer. However, if it was included in-between other layers, it would reduce the error map in size, so it fits the map dimensions that was originally passed to the layer. However, the padding layer was only used as the first layer in this project.

CNN Implementation

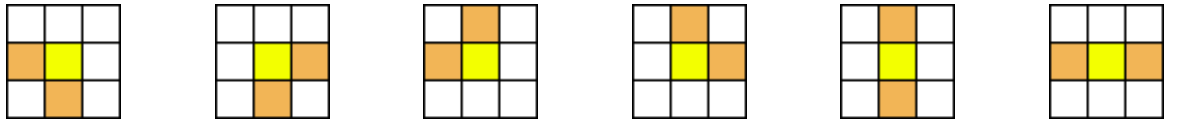
The implementation was performed in two steps, the first being implementing the classic CNN architecture with a specific order of layers, and with different methods that allowed for adjusting the internal settings of the layers through parameters. This included the operation of each layer (explained in Convolutional Neural Networks) and the standard backpropagation that assumes a specific architecture for the network. Afterwards, the implementation was adjusted to allow for different architectures, such as stacking different layers instead of following a specific order. This included a rework of the backpropagation, where the error maps were saved and used in a specific way as to properly perform the backpropagation through the layers. Adjusting the implementation aimed at making a flexible implementation, where different architectures could be utilized to improve performance both computationally and behaviorally. The implementation, similar to the ANN implementation (see ANN Implementation), was given multiple methods for running and training the network, allowing different settings to be changed through parameters. The code was separated into different sections, and different classes were used to store information that was necessary, such as the

filters. Furthermore, to improve the computation, the implementation focused on handling the different operations in an abstract way of representing the different maps through 2D arrays, which were traversed and used through indexing, loops, and data holders. Lastly, the backpropagation was kept as one big method, using a switch to go through the different backpropagation parts. This was handled using a stack containing the order of execution, so the backpropagation would call the different layers the correct number of times, and in the exact opposite order the layers were called during the forward propagation. For analysing the map, the implementation used 24 filters that would test for walkable areas either horizontally, vertically, or as a corner represented in four different combinations. For each type of the four walkable block, six filters were made to test for the different cases. The filters are shown in figure 19, 20, 21, and 22. When the program was started, the filters were parsed to the CNN, where the different block types were given starting values between negative and positive one, based on their importance, e.g., lava was given a value of minus one, while treasures were given a value of one. The filters shown below utilize a main block in the center, with two by platform blocks to form a pattern to check for paths, while the white spaces would be regarded is irrelevant and could be any type of block. These filters would then be updated through the backpropagation as the network learned, so the actual values of them would be adjusted away from their starting value towards what the network learned to be optimal.



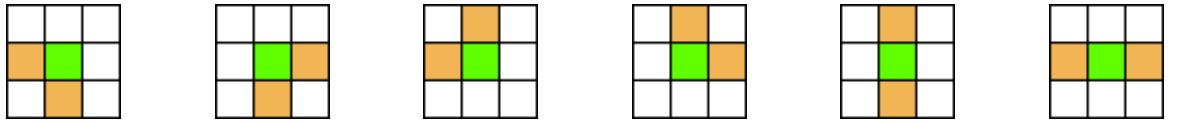
(a) Lower Left (b) Lower Right (c) Upper Left (d) Upper Right (e) Vertical (f) Horizontal

Figure 19: Path filters



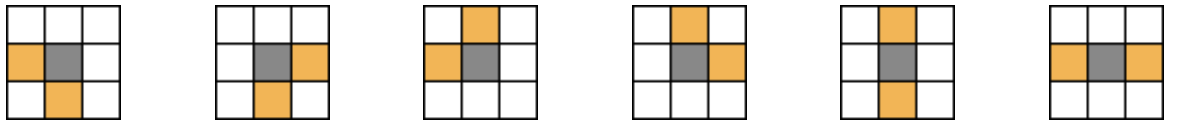
(a) Lower Left (b) Lower Right (c) Upper Left (d) Upper Right (e) Vertical (f) Horizontal

Figure 20: Treasure filters



(a) Lower Left (b) Lower Right (c) Upper Left (d) Upper Right (e) Vertical (f) Horizontal

Figure 21: Spawn filters



(a) Lower Left (b) Lower Right (c) Upper Left (d) Upper Right (e) Vertical (f) Horizontal

Figure 22: Goal filters

Network Support Classes

To improve the workflow with the ANN's and CNN's, different support classes were made to ease the handling of the networks. The first support class was a serialization class, which allows serializing the different networks and saving them as text, to later be deserialized and used again. The class was built to serialize both network types, where the CNN utilize the ANN serialization option to serialize any internal ANN's it contains. An example of a serialized ANN is shown in figure 23.

Figure 23: Example of a serialized ANN as seen in Unity's inspector

Another important support class was the configuration class, as it can read specifically written configuration files to either create an ANN or run a CNN. The ANN configuration file could be used when creating the ANN to set its settings and build its topology, while the CNN file would be used to run the CNN, where the settings are read as execution instructions from top to bottom. This allowed for easily creating different topologies for the ANN and architectures for the CNN, which are explored further in CNN Performance (Pre-Testing), ANN Re-Testing, and CNN Re-Testing.

```
<!---
#ANNConfig: ANN
Alpha: 0,05
Epochs: 10000
NumberOfHiddenLayers: 2
NumberOfHiddenNeuronsPerLayer: 8
NumberOfOutputNeurons: 4
HiddenActivationFunction: Sigmoid
OutputActivationFunction: Sigmoid


---!>
```

Figure 24: Example of an ANN configuration file, showing an ANN with an x-8-8-4 topology (x being the number of inputs, which is set in the code instead)

```
<!---
#CNNConfig:Training
Layer: Padding

Layer: Convolution
ActivationFunction: Sigmoid
Stride: 1

Layer: MaxPooling
ActivationFunction: Sigmoid
Stride: 2
KernelDimension: 2

Layer: FullyConnected
ANNConfig:
#ANNConfig: Internal Output ANN
Alpha: 0,05
Epochs: 1
NumberOfHiddenLayers: 0
NumberOfHiddenNeuronsPerLayer: 0
NumberOfOutputNeurons: 4
HiddenActivationFunction: None
OutputActivationFunction: Sigmoid


---!>
```

Figure 25: Example of a CNN configuration file, showing an architecture for a classic CNN with padding at the top

The last support class (and the most important during implementation) was a debugging class that could debug informations about execution of the networks, and output details about the different parts of the networks during different operations. As the networks handle thousands of numbers in the few milliseconds they run, debugging them step-by-step could take well over an hour. Instead, the debugging class allowed saving data about the execution, which could be saved to a scriptable object and read afterwards. This allowed watching and reading all the data to compare and look for irregularities. The class was given an adjustable debug depth, allowing for more details to be printed:

1. Only print the execution order.
2. Include stats and results from the different operations.

3. Include everything from each operation.

This class proved very useful, as the large amount of numbers could easily be read from a document and compared to each other, instead of sitting for hours going through the execution and writing down important details. Figure 26 shows an example of debugging a CNN at depth one. To see an example of a debug file at depth three, see Appendix 1.

```
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Starting CNN default Training | Total time: 0ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Starting convolution with a map of size 7x7, 3 filters, and a stride of 1 | Total time: 1ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Convolution Layer complete | Duration: 1ms | Total time: 3ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Starting pooling of all maps with a kernel with dimensions of 2, and a stride of 2 | Total
time: 5ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Pooling Layer complete | Duration: 0ms | Total time: 5ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Starting input generation from maps in pooled maps | Total time: 6ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Input generation complete with 12 inputs for the ANN | Total time: 7ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Running ANN | Total time: 8ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Run complete | Duration: 0ms | Total time: 9ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Fully Connected Layer complete | Total time: 9ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Starting backpropagation | Total time: 11ms |
|----------------------------------------------------------------------------------------------------------------------------|
| Operation: Backpropagation completed. 3 x Convolution, 3 x MaxPooling, 0 x AveragePooling, 3 x
InputGeneration, 1 x FullyConnected | Duration: 8ms | Total time: 20ms |
|----------------------------------------------------------------------------------------------------------------------------|
```

Figure 26: Example of a CNN debug file at depth one

Optimizations

This section goes over the optimizations that were made to the different methods, as problems appeared and solutions were found. Furthermore, the section, and the following section (Improvements (Post-Testing))

will describe testing and training set-ups for the ANN and CNN. Note that two different PC's were used during the project: a laptop for development and testing, and a desktop for training the ANN's and CNN's. The PC specs can be found in Appendix 2.

CNN Performance (Pre-Testing)

During implementation of the Convolutional Neural Network (CNN), performance was shown to be an issue that occurs when using the default settings of the network, as this generated a large number of inputs for the Artificial Neural Network (ANN) that is used in the CNN's fully-connected layer. Because of the larger number of inputs, the computation time resulted in lags each time an action was performed, which is undesirable for a game. As implementation progressed, the performance was improved from computation times around 63ms down to around 5ms. However, as the architecture of the CNN might require adjusting to allow for better data extraction, having guidelines for how different combinations of layers affect the computation was deemed important. Since a CNN does not have a specific architecture that must be used, changing the architecture of the CNN and the settings of the CNN's different layers can impact the performance of the CNN as a whole, but also change the resulting behavior of the agent. To properly test the performance of the CNN, different configuration files were created to test different architectures for the CNN. As to not randomly select different architecture and hope to come across something that seemed optimal, printed data from the implemented network's basic training method was used. As the output shows the time each operation takes, these initial times were used, along with definitions of the different layer's functions, to make estimates of the optimal architectures, which could later be tested. Table 1 shows the initial output for a default CNN training run with the following architecture and layer settings:

- Convolutional Layer:
  - Input map: 11x11
  - Filters: 24 of 3x3
  - Stride: 1
  - Activation Function: Sigmoid
- Max Pooling Layer:
  - Kernel: 2x2
  - Stride: 2
  - Activation Function: Sigmoid
- Fully Connected Layer:
  - Alpha: 0.05
  - Epochs: 1
  - Inputs generated: 384
  - Hidden layers: 0
  - Hidden neurons per hidden layer: 0
  - Hidden Activation Function: Sigmoid
  - Outputs: 4
  - Output Activation Function: Sigmoid

The test prints do not include the first run upon starting the application, as this run would be instantiating the network, thereby taking longer! Furthermore, times were rounded to the nearest whole number. The total time will always be higher than the different sections combined, as other processes also run in between certain layers, and the fully connected layer's time also contains the time for generating inputs to the ANN by transforming the maps to a list. As table 1 shows, using the implementations default training method performed efficiently enough to not slow down the game. However, as defined in the Convolutional Neural

Convolutional Layer	Max Pooling Layer	Fully Connected Layer	Backpropagation	Total
842µs	247µs	419µs	1063µs	3173µs

Table 1: Test run of the default training method of the CNN

Networks section, the CNN's architecture can be almost freely changed to include, exclude, add, or remove layers as the developer sees fit. Therefore, four different architectures were tested to compare the efficiency, as different architectures can have different effects on the network outcome. The order of the layers were kept similar to the order of the default run presented above, and the settings of the layers were kept identical to the default run, except for the number of inputs in the fully connected layer(s), which depended on the architecture.

- **Stacked Convolution**: Two convolutional layers in series, followed by a single max pooling layer, and lastly a single fully connected layer. This architecture will focus on extracting data from the input, then extract input from each convoluted map, before continuing to the pooling and decision making.
- **Stacked Pooling**: A single convolutional layer, followed by two max pooling layers, ending with a single fully connected layer. This architecture will focus on extracting the best data and minimize the number of data fed to the fully connected layer.
- **Weaved Convolution**: Two convolutional layers, each followed by a max pooling layer, ending in a single fully connected layer. This architecture combines the two previous architectures to extract, minimize, further extract, and further minimize the data fed to the fully connected layer.
- **Multi-Connected**: A single convolutional layer, a single max pooling layer, and two fully connected layers to analyze the extracted inputs. This architecture will focus on making a more in-depth analysis of the extracted data from the convolution and pooling.

The architectures were kept small (four or five layers), as more layers will increase the computational time (more operations would be required). This also meant that padding was excluded for these tests, as it would add more data to go through. Furthermore, all of the architectures had the time for their backpropagation noted as well, as the backpropagation process took the longest of all the processes during development.

Each architecture was run four times, the first being the instantiation run, which was skipped, while the following three were used. For each of those three runs, the time for each operation was noted and added to table 2. From the table, it is clear to see that backpropagation was the operation that took the majority of time during each process, but as the backpropagation runs through all of the layers to update all weights in the network (see CNN Backpropagation), it cannot be seen as a major time consumer compared to the rest. Instead, the convolutional layer and the fully connected layer spent the most time for their calculations. Comparing the inputs given to the fully connected layers also show a clear connection with the time for completing the layer and doing the backpropagation. Lastly, the max pooling layer is the fastest layer, even if it is stacked. Since the layer does not perform any major calculations, but just compare and pick the highest value in the kernel, it performs the fastest. The max pooling layer also has a major influence on the input generation for the fully connected layer, as it can minimize the maps, while keeping the most important informations. Comparing the different architectures, the Stacked Pooling was the overall fastest architecture, but only by 961µs faster than the Multi-Connected architecture. Meanwhile, problems occurred for the Stacked Convolution and Weaved Convolution architectures, as they would hit infinity during their backpropagation after the initial run. Therefore, the times noted in 2 for those two are the initial times, which were excluded for the other two. Since this was a problem at the time, these two were excluded continuing through the project.

Network Training

After optimizing the CNN and experimenting with different architectures, training the different networks was the next step in the process. Based on the architecture performance tests, two of the alternative

Architecture	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**	**Backpropagation**	**Total**
*Stacked Convolution	Convolutional	Convolutional	Max Pooling	Fully Connected	N/A	Backpropagation	Total
	2223µs	12708µs	7535µs	inputs: 9216, 24586µs	N/A	86167µs	142632µs
Average	2223µs	12708µs	7535µs	inputs: 9216, 24586µs	N/A	86167µs	142632µs
Stacked Pooling	Convolutional	Max Pooling	Max Pooling	Fully Connected	N/A	Backpropagation	Total
	851µs	267µs	106µs	inputs: 216, 294µs	N/A	905µs	3310µs
	1033µs	265µs	106µs	inputs: 216, 249µs	N/A	954µs	3482µs
	906µs	270µs	134µs	inputs: 216, 245µs	N/A	957µs	3272µs
Average	930µs	267µs	115µs	inputs: 216, 263µs	N/A	939µs	3355µs
*Weaved Convolution	Convolutional	Max Pooling	Convolutional	Max Pooling	Fully Connected	Backpropagation	Total
	3938µs	444µs	3969µs	1746µs	inputs: 2304, 15606µs	66140µs	99619µs
Average	3938µs	444µs	3969µs	1746µs	inputs: 2304, 15606µs	66140µs	99619µs
Multi-Connected	Convolutional	Max Pooling	Fully Connected	Fully Connected	N/A	Backpropagation	Total
	958µs	377µs	inputs: 600, 984µs	inputs: 8, 8µs	N/A	1412µs	4849µs
	908µs	269µs	inputs: 600, 642µs	inputs: 8, 6µs	N/A	1342µs	3951µs
	918µs	258µs	inputs: 600, 754µs	inputs: 8, 7µs	N/A	1410µs	4147µs
Average	928µs	301µs	inputs: 600, 793µs	inputs: 8, 7µs	N/A	1388µs	4316µs

Table 2: Architecture test with four different layer combinations.
*These architectures would hit infinity during their backpropagation, so their values were not comparable

architectures (Stacked Pooling and Multi-Connected) were deemed viable for the agent, so they were included in the training process along with the default architecture. Different networks were created based on these three architectures, after which the networks were trained using the same data sets (see Training Data Collection). However, problems with the CNN implementation became apparent at this point, as the three selected architectures started to reach infinity as training went on, just like the problems presented earlier. Especially the two alternative architectures would reach infinity before reaching ten training runs. As this problem occurred mostly in the alternative architectures, all of these were excluded and only the default was kept and trained for the upcoming test.

After completing the training of the CNN, the focus was shifted to train the ANN. As the ANN would require a specific topology, this was made using the ANN configuration setup. The ANN would have 121 input neurons, as that would cover the visible part of the map, and it would have four output neurons, as those are the possible choices of the network. As discussed in ANN Implementation, the inclusion of hidden layers and hidden neurons is hard to decide. To take a common approach to the topology, having two layers of eight neurons each was chosen, as the input to output ratio was high. Furthermore, the alpha was kept at its default value of 0.05, and the network was trained thousands of times to accommodate for a low learning rate. After training the ANN, preparation for the comparison test were made, which is explained in Agent Comparison Test.

Improvements (Post-Testing)

After the agent comparison test was completed (see Agent Comparison Test), time was allocated to look at the implementations to make optimizations, fix issues, and test performance to find possible improvements for the trained networks used in the comparison test.

Network Improvements

As the networks were reaching strangely high values during the training sessions, something with the implementations must have been off. The artificial neural network (ANN) was debugged step-by-step, based on knowledge from the pre-test work, and a few issues were found:

1. A part of the ANN's backpropagation was done incorrectly, where the weight update for the hidden layer neurons was not following the backpropagation algorithm correctly (Rumelhart et al., 1995), which would likely had caused the training to be skewed. Instead of running the delta value through the activation function, the hidden layer's index was sent through, which would offset the weight and bias corrections.

2. Parsing the chosen activation functions from the ANN configuration file into the ANN was not done correctly, and instead of throwing an error, the value that was supposed to go through the activation function was instead returned and used. This was a major issue, as that alone could create spikes during the propagations.

3. For the ANN that had been trained for the agent comparison test, values in the configuration file were swapped, so instead of a network that would be x-8-8-4, it was x-2-2-2-2-2-2-2-2-4; there were eight hidden layers with two neurons instead of two hidden layers with eight neurons. This might have reduced the impact of the decisions made by the network.

Apart from the issues with the ANN implementation, the backpropagation for the CNN was still reaching infinity after this. A closer inspection of the backpropagation raised the question of whether an alpha value should had been added to the delta calculations for updating the filter. As an alpha value is important in an ANN, this was tested by inserting an alpha value for the filter updates which massively improved the CNN's. By including the alpha value, it ensured that the filters were not adjusted too fast, allowing for more precision during training and avoiding the infinity.

After all the issues were addressed and fixed, the networks were trained again to see how they would perform. Using alternative CNN architectures still reached infinity, so they were completely off the table from that point. Also, the agent started acting a bit different when controlled by the networks, as it would

no longer just walk in a straight line (see Agent Comparison Test), but sometimes walk backwards to avoid lava. Since the issues were fixed, new training sessions were executed to see the effects of the changes. To thoroughly test the ANN, a proper in-depth training session was performed to test whether the network, with its improvements, would perform better at all. The network was trained in two different ways: using long training sessions through playback and using short training sessions with the emulator. In both cases, the topology was also adjusted to test if the network was performing too many or too few calculations. For the CNN, the architectures presented and tested in CNN Performance (Pre-Testing) were re-created and tested again to ensure if they had any effect on the agent behavior. The specific set-ups and the results are presented and discussed in the following section.

ANN Re-Testing

The first network to be trained and tested was the ANN, as noteworthy changes in the ANN could result in important behavior changes in the CNN. As the ANN can be configured using a configuration file, the ANN was tested with different variations of settings and topology. Furthermore, to assure the training emulator worked as intended, the tests were performed twice, both emulated and live (discussed in The Agent). The first test would focus on a slightly bigger network, to see if it could improve the behavior. The ANN's topology was adjusted with an extra hidden layer of eight neurons, while its internal settings were changed for each test. Prior to the tests performed and described in the following, a larger test was conducted to test whether a longer training time (approximately six hours of training this single network) would leave a bigger impact on the network's performance. This test setup is presented in table 3.

Topology	**Alpha**	**Epochs**	**Activation Functions**	**Data Sets**	**Iterations**	**Total Runs**	**Training Type**	**Results (3 runs)**
121-8-8-8-4	0.05	10000	Sigmoid (H) Sigmoid (O)	8	10	800.000	Emulated	Left Left Left

Table 3: Test setup for the long training session (six hours) of a single network

As table 3 shows in the 'Results' column, the ANN did not perform any better. The understanding of the noted results are discussed further down.

To properly test different variations of the originally considered ANN, three topology variances were tested: one with a single hidden layer, one with two hidden layers, and one with three hidden layers. In all three variances, the hidden layers would contain eight neurons each. These choices were based on considerations about the required number of hidden calculations that could be necessary in the network's decision making (Heaton, 2005). Having 121 inputs (11 by 11 visible grid transformed to inputs) means a lot of computation could be necessary. Furthermore, having five different blocks (see The Map) and the outside area (defaults to the same value as lava blocks) would give six different types of inputs for consideration. With those considerations, only having six neurons per hidden layer could had been better, but having double the neurons to the output layer for more calculations was deemed better.

Deciding on the best topology based on considerations can be hard, and close to impossible, so future work could utilize neuro-evolution (e.g., NEAT (Stanley and Miikkulainen, 2002)) to find an optimal topology.

Apart from the topology, different values for epochs (internal runs) and different alpha values (learning rate) were also tested for each topology. To ensure the emulator (see The Agent) was working correctly, each test was performed both live (playback of a data set) and emulated (handled by code). Each ANN was trained and tested, and the settings and results can be seen in 4, 5, and 6.

The tables each contain two blocks based on the 'Alpha' value, and both blocks contain two smaller sections based on the 'Training Type', where two different 'Epochs' values were tested. The 'Activation Functions', 'Data Sets' (see Training Data Collection for more detail), and 'Iterations' were kept the same throughout this test, though further testing could test variances in these as well. 'Total Runs' would wary based on 'Epochs', 'Data Sets', and 'Iterations' ($TotalRuns = Epochs \cdot DataSets \cdot Iterations$).

Topology	Alpha	Epochs	Activation Functions	Data Sets	Iterations	Total Runs	Training Type	Results (3 runs)
121-8-4	0.05	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Emulated	Forward Forward Forward
121-8-4	0.05	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Emulated	Forward Forward Forward
121-8-4	0.05	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Live	Forward Right Right
121-8-4	0.05	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Live	Left Left Left
121-8-4	0.5	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Emulated	Right Right Right
121-8-4	0.5	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Emulated	Right Right Right
121-8-4	0.5	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Live	Right Right Right
121-8-4	0.5	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Live	Left Left Left

Table 4: Results of using different alpha values, epochs, and training types for a 121-8-4 topology

Topology	Alpha	Epochs	Activation Functions	Data Sets	Iterations	Total Runs	Training Type	Results (3 runs)
121-8-8-4	0.05	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Emulated	Back Back Back
121-8-8-4	0.05	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Emulated	Right Right Right
121-8-8-4	0.05	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Live	Forward Forward Left
121-8-8-4	0.05	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Live	Left Left Left
121-8-8-4	0.5	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Emulated	Left Left Left
121-8-8-4	0.5	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Emulated	Forward Left Left
121-8-8-4	0.5	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Live	Forward Forward Forward
121-8-8-4	0.5	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Live	Forward Forward Forward

Table 5: Results of using different alpha values, epochs, and training types for a 121-8-8-4 topology

Topology	Alpha	Epochs	Activation Functions	Data Sets	Iterations	Total Runs	Training Type	Results (3 runs)
121-8-8-8-4	0.05	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Emulated	Left Left Left
121-8-8-8-4	0.05	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Emulated	Back Back Back
121-8-8-8-4	0.05	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Live	Left Left Left
121-8-8-8-4	0.05	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Live	Left Left Left
121-8-8-8-4	0.5	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Emulated	Left Forward Left
121-8-8-8-4	0.5	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Emulated	Back Back Back
121-8-8-8-4	0.5	100	Sigmoid (H) Sigmoid (O)	24	1	2400	Live	Left Left Left
121-8-8-8-4	0.5	1000	Sigmoid (H) Sigmoid (O)	24	1	24000	Live	Right Right Right

Table 6: Results of using different alpha values, epochs, and training types for a 121-8-8-8-4 topology

Tests were performed after each training session, and the actions were judged based on the preferred action to perform. The agent would perform an action every 0.1 seconds, and after a couple of seconds, the action performed for that time period would be noted. However, if the agent moved forward just once, this was noted as an action, as it would result in a new position (and new inputs), compared to continuously turning and backing, all of which would keep the agent at the spawn block (backing without walking forward or turning first would attempt to move the agent into the border). Furthermore, to ensure the agent had ample room to act, the map was remade if it contained a lava block within three blocks of the agent's front, as it could hinder moving forward. Several remade maps did show the agent performing other actions based on the generated map, so future experiments could judge the agent's output based on the differently generated maps.

As presented in the tables, there were cases where the agent performed two different actions: the forward action was performed, after which it changed to a turn action. But this was a significant change in comparison to the other entries, as these showed the agent reaching a different decision based on the inputs. Entry three in table 4, entry three and six in table 5, and entry five in table 6 were interesting improvements. In most of them, the agent would take one step forward, and then continuously turn, while for entry three of table 6, the agent would take two steps forward and then start turning. However, in entry five of table 6, the agent would turn to the left for a second, then take a step forward, and then continuously turn left again, which is a weird change in its behavior, as it reached a new decision with the same inputs that made it turn. This could had been an error of some kind, as that, in theory, should not happen. As the priority is moving and turning the agent, choosing one of these topologies would be a step in the right direction. Based on these results, the topology and values for entry three in table 5 were deemed the best based on these considerations: entry five of table 6 could be a bug, so it was excluded, and the majority of the remaining networks that moved forward and then turned had an alpha value of 0.05 and epoch value of 100. Therefore, the ANN settings for that entry were chosen as the ANN settings for the fully-connected layers in the re-testing of the different CNN architectures. Lastly, the ANN re-testing did not show any major differences between the live and emulated training, therefore the emulator was used during the CNN re-training.

CNN Re-Testing

After fixing the issues presented in Network Improvements and testing different settings and topology variations for the ANN, the changes were brought to the CNN default architecture. Just like the performance test, the network used the same settings for the different layers, though with the fully-connected layer's ANN replaced by the ANN found as the optimal ANN in ANN Re-Testing.

- Convolutional Layer:
  - Input map: 11x11
  - Filters: 24 of 3x3
  - Stride: 1
  - Activation Function: Sigmoid
- Max Pooling Layer:
  - Kernel: 2x2
  - Stride: 2
  - Activation Function: Sigmoid
- Fully Connected Layer:
  - Alpha: 0.05
  - Epochs: 100
  - Inputs generated: Architecture dependent
  - Hidden layers: 2
  - Hidden neurons per hidden layer: 8

- Hidden Activation Function: Sigmoid
- Outputs: 4
- Output Activation Function: Sigmoid

As the ANN re-testing did not show any major differences between live and emulated training, the re-training of the CNN's utilized emulated training to speed up the process. Table 7 shows the results of the test, the behavior, and the average calculation time. To further discuss the performance and results, the number of neurons fed to the fully-connected layer for decision making was noted as well. And as described in 5.1, the first run was skipped here as well as to not count the initialization time.

Architecture	**Number of ANN Inputs**	**Calculation Time**	**Results**
Default	864	2784µs	Forward Forward Forward

Table 7

As there were still issues with the other architectures described in CNN Performance (Pre-Testing), only the default CNN architecture was tested in the end. The results of the test did not change from the comparison test, as the CNN would still default to a single action. The only major improvement was the calculation time, as it was down to 2.7ms, even with a large number of inputs. Continued work on the implementation to fix the remaining issues could improve the chances of success in future projects. The code is available from bias2402 (2021d)

Data Collections

This section goes through the different data collections that were used throughout the project, either for collecting data for network training, or for collecting necessary data about the agents and the human-likeliness. In contrast to the Optimizations and Improvements (Post-Testing) sections, this section will only discuss data collected from people.

Training Data Collection

In order to train the Artificial Neural Networks (ANN's) and Convolutional Neural Networks (CNN's), data sets with recordings of human players playing the game were required. Since the project focus was simulating player's behavior, the networks would need information about how players acted when using different play styles (see The Game). Therefore, the test game was expanded to gather information from the player by recording necessary information and saving it to a text file, which the player could send back to the developer afterwards. These text files would record the player's inputs, the delay from when agent was ready to the player actually gave an input, and an encoded version of the map that the player got, as the recorded actions would only work on that specific map. To gather information about different play styles, the player had to play through the map three times, each time with a new main objective to emulate a specific persona (The Game):

1. Speedrunner: reach the goal in as few steps as possible.
2. Explorer: explore as much of the map as possible before reaching the goal.
3. Treasure Hunter: find and step on all the treasures before reaching the goal.

For collecting the data sets, the game was adapted to give short, precise informations to the player about the game and how they should play it. When the game was played, the players would play through the game three times, each time with another play style. To make it easier for the players, and attempt to make a more realistic playthrough, the order of the play styles was given as presented above. That way,

players would try and speedrun the map with no prior knowledge about it, then explore everything, while not necessarily remembering anything but the path to the goal, and then lastly have to explore and reach each treasure, which they might remember having seen, but also might not have realized were even there. The players could had been given three random maps, thereby making the order irrelevant, but to gather proper data about different play styles, it was decided that each play style should be used for the same map, so the networks could learn how to play that specific map in different ways. As the text files were generated, the name of the file would be current play style followed by a name entered by the player. This name could be their actual first name or a gamer tag if they wanted: the name was only used to differentiate between the files.

After preparing the build, it was shared across gaming communities on Discord with the necessary information for handing in the generated text files. After one week, only eight people had sent back text files from their playthrough, which totaled 24 data sets for the networks. These were added to scriptable objects for persona training, each object containing only the files of a specific persona, and the objects could then be used for network training.

Agent Comparison Test

The goal of the project is trying to simulate players in a game, which requires agents that play in a similar fashion to human players. To test whether the implemented artificial intelligence (AI) and machine learning (ML) techniques could be mistaken for humans in this game, a survey was conducted based on Turing's The Imitation Game (Oppy and Dowe, 2020). In the survey, different recordings of the agent would be compared, and in each recording, the agent would be controlled in a different way. The test contained six different recordings of the agent running through the map, with the agent being controlled differently in each video:

1. Artificial Neural Network (Jensen, 2021a)
2. Behavior Tree (Jensen, 2021b)
3. Convolutional Neural Network (Jensen, 2021c)
4. Finite State Machine (Jensen, 2021d)
5. Human player (Jensen, 2021e)
6. Playback (Jensen, 2021f)

The videos were recorded from within Unity, where all relevant details that could reveal the control type were removed, or hidden during post-processing. The persona for this test was based on the speedrunner, as that was the easiest to implement and train agents to perfom. The videos were then uploaded to YouTube and named Agent 1, Agent 2, etc., to hide information about the controller. To gather relevant information for the later analysis, the videos were added to the end of a survey that was made to gather demographic information about the participants, their knowledge of AI, and lastly have them rank the recordings based on human-likeliness. The survey was made using Google Analyse (in Google Drive) consisting of the three sections 'Demographics', 'AI Knowledge', and 'Comparison'. The survey was made as a quantitative survey (Creswell and Creswell, 2018) to focus on comparing the demographics and knowledge of the participants against their perception of the agent in the different recordings. Qualitative information about their perception was excluded, as the general goal was getting an understanding of the different technique's impact on the agent's general behavior. Future studies could utilize a mixed methods study to gather further information about what makes the agent seem human-like. The following sections will go into more details about the three survey categories, and the whole survey, and results, can be found in Appendix 3. The results of the survey were analyzed using Google Analyse's built-in analytics tool (graphs based on the replies), and the answers printed to a Google Sheets (the answers from each participant grouped together). The survey had 15 participants, and their answers are discussed further down in Results and Discussion of Survey Results. Furthermore, the extra materials contain a CSV file with the replies.

Demographics

The demographics section focused on gathering personal informations about the participant's gender, age group, education, and game involvement. The following questions were added to this part of the survey:

1. Gender
2. Age
3. What is your highest completed education?
4. If you answered "University" or "Vocational Education" in the previous, what did you choose?
5. How much do you play video games in an average week?
6. How much do you watch streamers or videos of people playing video games in an average week?

The questions about their involvement in either playing or watching others play video games were added, as to make an assessment of knowledge about AI in games based on their interaction with video games. As almost every video game contains AI in some form (see Artificial Intelligence), playing or watching others was seen as a way of gaining knowledge about game AI that could assist in distinguishing humans from AI's in games. Lastly, the replies to the fourth question were partly unusable, as the question was badly worded ("what did you choose" should had been "what did you study/work with"), so two of the five answers were unusable leaving only three (out of six participants stating 'University' as their highest education) actually useful. Of those three, the replies given were still incomparable, as they had all mentioned the same education.

AI Knowledge

To properly get an idea of the participants knowledge of AI, these questions were added to the second section of the survey:

1. How much, according to yourself, do you know about AI in general?
2. How much, according to yourself, do you know about AI in games?
3. How much do you think AI is used in games?
4. How many games do you think contain AI?

These questions would use rankings, so the participants could rate their self-estimated knowledge of AI. The reason behind these questions was that more knowledge should make it easier to spot the AI's between the recordings. Comparing a participants self-estimated knowledge of AI with their education and video game involvement could show interesting connections for future research and implementations of AI.

Comparison

The last section contained the six recordings, each followed by a question asking whether or not this recording was of a human player, and at the end, a ranking matrix for ranking the videos from most human-like to least human-like. The ranking of the recordings would be useful for judging which techniques seemed to work the best, compared to the actual human recordings. Looking at the participants thoughts about a recording being of a human or not would play well into considerations of whether a certain technique would be better for this type of game or not.

Results

For analyzing the replies, the graphs, made by Google Analyse, were compared with the participants individual answers recorded in a Google Sheets. In general, the majority of the participants were male (figure 27), and most were in their 20's (figure 28). Looking at the different participant's answers in the Google Sheets, the majority of these participants also had a university education (figure 30). Looking at the graphs for playing (figure 31) and watching (figure 32) others play video games, 46.7% spent less than ten hours a week playing games, while 66.7% spent less than ten hours watching streams. Following the idea that interacting more with video games have an impact on their knowledge of game AI, having most participants play more than ten hours a week could indicate that they have an easier time spotting the AI controlled agents.

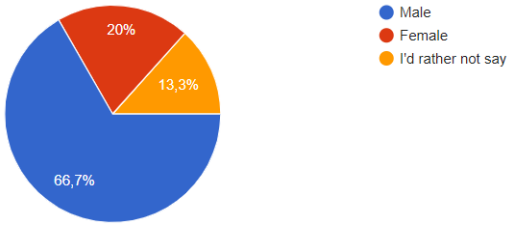


Figure 27: The gender of the survey participants

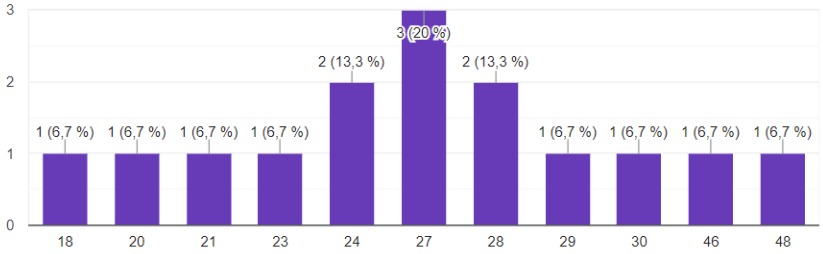


Figure 28: The age spread of the survey participants

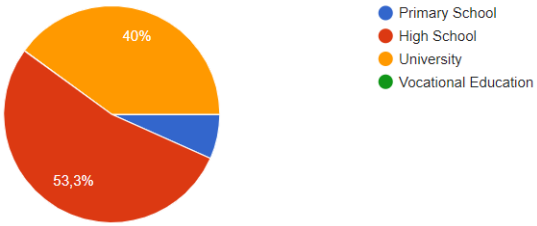


Figure 29: Education of the survey participants

Gender	Age	What is your highest completed education?
Male	18	High School
I'd rather not say	28	High School
Male	48	High School
Female	21	High School
Male	46	High School
I'd rather not say	20	High School
Male	23	University
Male	27	University
Female	29	University
Male	24	High School
Male	30	University
Male	24	High School
Male	28	University
Male	27	Primary School
Female	27	University

Figure 30: Gender, age, and education of the participants

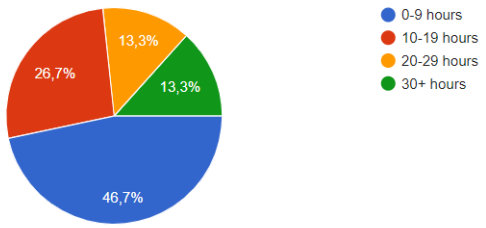


Figure 31: Spread in hours spent playing video games per week

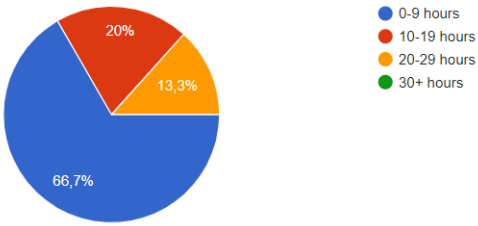


Figure 32: Spread in hours spent watching others play video games per week

Moving on to the results of AI knowledge, eight ranked themselves as having above average knowledge about AI in general (figure 33), while nine ranked themselves above average when it came to AI in video games (figure 34). Comparing the numbers, the participants ranked themselves equal or higher in knowledge from general to game AI, with only two exceptions. What made these two judge themselves lower in game AI knowledge is hard to determine. In one case, the person only had a primary school education and spent more time watching streams (10-19 hours) than actually playing (0-9 hours). In the other case, the person has a high school education, and plays video games (20-29 hours) more than watching streams (0-9 hours). Looking further, the same two also had different ideas of the inclusion of AI in games. The first rated AI's used highly (8/10), while the second rated its use low (3/10). Meanwhile, the rest majorly placed AI inclusion at average or above (figure 35). Lastly, the majority of people moved their ranking higher, when asked how many games they thought included AI (figure 36).

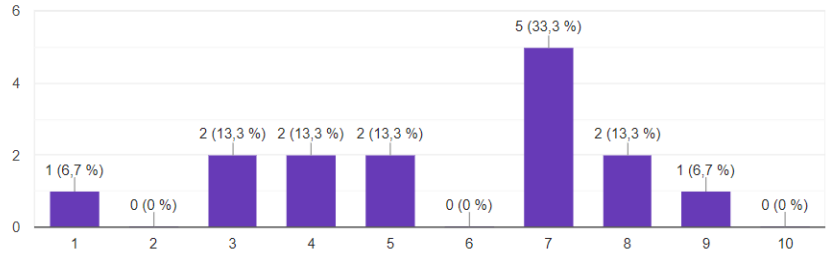


Figure 33: Self-estimated knowledge about AI in general

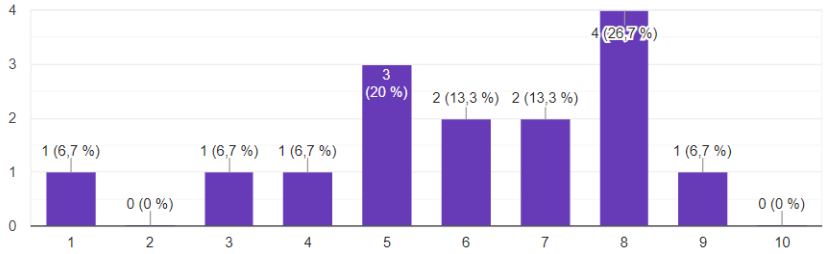


Figure 34: Self-estimated knowledge about AI in games

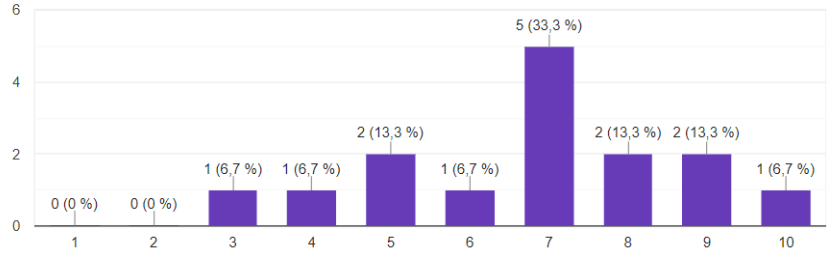


Figure 35: Thoughts about AI inclusion in games

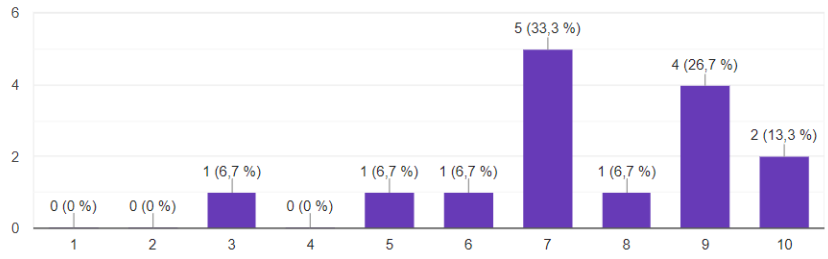
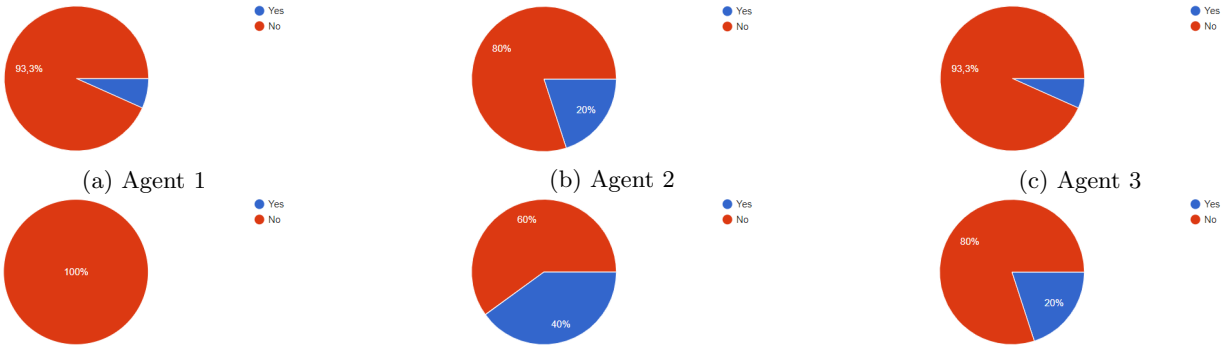


Figure 36: Assumption about the amount of games containing AI

The last section showed some interesting results, as some participants (5/15) did not find any of the recordings to be of humans at all. The video given the highest score when asked if it was a human (based on the questions asked after each recording) was Agent 5, the human player (figure 37. This was not a major surprise, however, looking at second place, it was shared between Agent 2 and Agent 6, the behavior tree and playback, respectively. Having the playback being second does make sense, as it is a playback of a human recording, but having the behavior tree there as well makes for an interesting result. To further explore the human-likeliness of the agents, the rankings can shed some light on these results (figure 38). The agent ranked highest was Agent 5, closely followed by Agent 6. Agent 2's ranking was more spread, from being the most human-like to also being the fifth-most human-like. Agent 2 was mostly placed as the fifth-most or second-most, which means it did something human-like according to some participants. Of the two participants that ranked Agent 2 highest, only one of them actually thought it was a human in the earlier question. And the participant that thought it was a person, was the second guy discussed above (the one ranking their AI knowledge lower, and playing more than watching others). The ranking also makes it clear that Agent 1 (ANN) was the least human-like, though it also managed to get a similar ranking to Agent 2 on the most human-like. Meanwhile, Agent 3 (CNN) and Agent 4 (FSM) were spread between second-most human-like to fifth-most human-like.



(a) Agent 1 (b) Agent 2 (c) Agent 3

(d) Agent 4 (e) Agent 5 (f) Agent 6

Figure 37: Graphs showing the percentage of people stating that a recording was of a human player

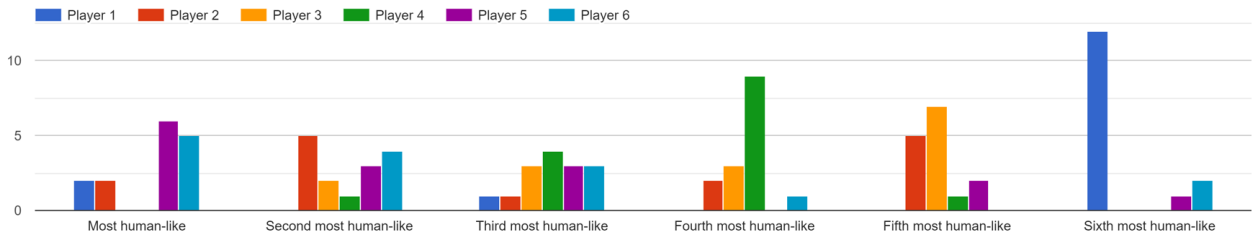


Figure 38: The ranking of the agents

Discussion of Survey Results

Different questions can be raised based on the survey results:

1. Why did the behavior tree (BT) perform so well?
2. Why did the networks in general fail?
3. Why did the finite state machine (FSM) perform worse than the behavior tree?
4. What made the human recordings seem more natural?
5. How does the demographics play into the rankings?

Most of these questions would require a new study (qualitative or mixed methods) to properly explore, but based on the available information, and the recordings used, some things could be seen as reasonable explanations.

Why did the behavior tree perform so well? It might be its implementation that made it look more realistic. Combine that with the next question about the networks, it is easy to see a difference: the networks only made the agent walk forward. Meanwhile, the behavior tree made the agent turn and choose different paths when blocked. This also leads into the third question, which can be answered in one of two ways: the BT's way of choosing an option was better, or the BT's implementation for finding its way and moving was better. The BT did get more assistance in finding its way towards the goal through a depth-first search, while the FSM was limited to randomness. Furthermore, the BT was programmed to always prefer going towards the right, whenever possible, something that could help explain question four as well, since the human players would likely attempt to go towards the goal in order to find it. But how does the demographics play into this? For this, it is important to remember that the survey only had 15 responses, so nothing can be concluded. Looking at figure 39, there does not seem to be any link between the people that ranked the actual human agents (this includes the playback agent) and their demographics. The majority were male, but considering the majority of responses were from males, it cannot be seen as valid. The only things that seem a little different are their weekly time spent playing or watching others play video games. But this could as well have been a coincidence with the small number of participants. A more in-depth research would be required to make any assumptions on this part.

Gender	Age	What is your highest c	If yc	How much do you p	How much do	How	How	How	How	Do yc	Do yo	Do yc	Do y	Do yc	Do yc	Rank the players based on how human-like you find them [Most human-like]
Male	18	High School		30+ hours	10-19 hours	4	6	7	10	No	Yes	No	No	Yes	No	Player 5
I'd rather not say	28	High School		30+ hours	10-19 hours	5	5	5	7	No	No	No	No	No	No	Player 5
Male	48	High School		20-29 hours	0-9 hours	5	4	3	3	No	Yes	No	No	No	No	Player 2
Female	21	High School		10-19 hours	20-29 hours	1	1	10	10	No	Yes	No	No	No	Yes	Player 6
Male	46	High School		0-9 hours	0-9 hours	7	8	4	8	No	No	No	No	Yes	No	Player 5
I'd rather not say	20	High School		0-9 hours	0-9 hours	9	9	9	9	No	No	No	No	Yes	Yes	Player 6
Male	23	University	Lea	0-9 hours	0-9 hours	3	5	7	7	No	No	No	No	No	No	Player 6
Male	27	University		10-19 hours	0-9 hours	7	7	7	7	No	No	No	No	No	No	Player 1
Female	29	University	Uni	0-9 hours	0-9 hours	3	3	7	7	No	No	No	No	Yes	No	Player 5
Male	24	High School		0-9 hours	0-9 hours	8	8	6	9	No	No	No	No	No	No	Player 2
Male	30	University	Lea	10-19 hours	0-9 hours	8	8	8	9	No	No	No	No	Yes	No	Player 5
Male	24	High School		0-9 hours	0-9 hours	7	7	9	9	No	No	No	No	No	Yes	Player 6
Male	28	University	Opt	20-29 hours	20-29 hours	4	6	5	5	No	No	No	No	No	No	Player 5
Male	27	Primary School		0-9 hours	10-19 hours	7	5	8	6	Yes	No	Yes	No	Yes	No	Player 1
Female	27	University	I ch	10-19 hours	0-9 hours	7	8	7	7	No	No	No	No	No	No	Player 6

Figure 39: Responses colored based on the choice for most human like

After the comparison test, small-talk with one of the participants revealed information about their decision of what agent seemed most human-like. Based on what they found to be more human-like, they chose Agent 2 (BT), because of the way it moved. Sometimes, it would turn around itself a bit, similar to a player that is indecisive of their next move. This seemed more natural to them, which is a significant detail. A question about the delay during the actual human recordings was asked, but this did not seem to have the same amount of impact, though they did say it made it feel more natural, as the flow was not continuous compared to the AI agents. Future research should explore these behaviors more, as they seem to have an impact, even if more subconsciously.

Lastly, the test only focused on the speedrunner persona. As stated, this was chosen since it was easier to work with, while the others could had been tested later. However, tests of the other personas was dropped,

as the networks did not improve (see Improvements (Post-Testing)), and making the adjustments to the FSM and BT would be too time consuming at that point. As this test shows, it is possible to simulate players decently using a BT, so in theory, and with enough time, creating an AI that simulates a player really well is possible.

Discussion

Though the project did show promising results, it was not without its fair share of complications and issues. Working with convolutional neural networks (CNN's) was an alternative way to analyze and learn to play games compared to more commonly used techniques, such artificial neural networks (ANN's), reinforcement learning (RL), and neuro-evolution (NE). These other techniques were also considered, but due to the limited time of the project, only ANN's were chosen alongside CNN's, as to compare the two. The major problems that occurred were mostly confined to the CNN, as attempts to implement more functionality to better control and wary the network architecture resulted in computational issues. Attempting to fix these issues was time consuming, time that could had been focused on other parts to gather more in-depth data about the issues with using ANN's and CNN's in this specific game. Some minor issues with the ANN implementation also occurred, which were addressed in the Improvements (Post-Testing) section. As the Agent Comparison Test recordings showed, the two networks did not perform particularly well compared to the finite state machine (FSM) and the behavior tree (BT). The problems with this could be due to lack of training, which was addressed for the ANN in ANN Re-Testing, but it could also be the nature of the game that is too restricted for the networks to properly operate. As some of the prior research presented in Artificial Intelligence does successfully utilize agents using neural networks in rouge-like games, the issues presented here are not a justification for stating that neural networks cannot act in this type of rogue-like. More likely, the agents in the prior research has had better and longer training, and since they mostly seem to use RL or NE in some way, this allows them to perform unsupervised learning in contrast to the supervised learning presented here, which required the collection of training data before training could commence. Other issues that affected the network negatively could be the choice of inputs they were fed. The networks were only given information about the map to make decisions, but could had required more information about state, general position, and specific features present in the visible part of the map. This was the base idea of the CNN; trying to identify these areas, which it did not learn to do properly. As explained in ANN Implementation, the ANN was only allowed to use discrete activation, as the game only allowed turn-based actions, where a delay was present between each performed action. Removing this constraint, and possibly choosing a different baseline genre for future work could give different results, as a network might be able to gather more precise information to make decisions.

When comparing the neural networks to the FSM and BT, the latter performed a lot better. They were able to react to the surroundings and path around the obstacles in their way. What makes these techniques easier to perform well is the hard-coded actions they are given, as they can easily change based on the inputs they are allowed to gather from their surroundings. As presented with the BT, giving it the small advantage of using a depth-first search (DFS) to locally analyze the map made it perform better than the FSM in the comparison test. Using a DFS is an easy way of pathing through a maze, but when trying to simulate humans, the developer cannot give an AI that advantage as it would become too perfect. Instead, DFS allowed the BT to take a similar approach in getting as far towards the right as possible, leading it towards the goal in a more realistic manner. Furthermore, in comparison to the networks, the FSM and BT implementations were given some randomness and forced direction, which assisted them to get further towards the goal, should they ever get stuck, whereas the networks would have to figure that out by themselves. Therefore, working with the classic AI techniques does seem to be a better way of ensuring a more realistic behavior. Further considering the time spent implementing and working with the different techniques, it does make the classic techniques seem superior, as they all-in-all took about 20 hours to implement, while implementation, experimentation, bug fixing, and training of the network took months. It is important note that the CNN was implemented from the ground up (as mentioned in ANN Implementation, the ANN implementation was made prior to the project start), and if a prior implementation had been used, the total time spent could probably had been closer to that of the classic techniques. To make the time comparison more honest, the BT was partly implemented using a Unity package that was in progress before the project started, and was completed in the first weeks of the project. This package did take close to 50 hours in itself, but implementing the

general logic of a BT would realistically only take around five hours. Using implementations that allow more control does make the work faster for realistic agents, but it also requires more time testing and adjusting to perform different play styles properly, whereas a good implementation of a self-learning agent could reduce implementation time a lot, after which the agent can train itself without human interaction.

Stating before that the agents in the prior research did perform well in rogue-like games while being controlled by neural networks left out a single important detail: simulating players. Creating an agent to perform a specific play style is possible, but does that mean that it performs like a human would do? If an agent is trained to be a treasure hunter, it will most likely take the most optimal ways to the treasures. In a game like this one, the agent will only have limited vision, so it would either require memory of its exploration or have to search randomly to find everything. Meanwhile, a human player might remember where they have been, or they might have a hard time finding something, because they do not know where they missed that something. Simulating confusion or target oriented behavior could be the link for making agents seem more human-like. As mention by one of the participants during small-talk (see Results), agents that seem to hesitate before moving or seem indecisive were regarded as more human-like. This small detail could be implemented in future agents to make them better at simulating the indecisiveness of human nature, when presented an unknown space to traverse. As the comparison test indicates, trying to simulate players is not an easy task, as people still found certain things to be more realistic with the actual human recordings. The agent had an action delay of 0.25 seconds between accepting inputs, as to make it easier to control if a player held down the button. But the data collection would also record the time from the delay running out to the player actually giving a new input. So, if the player waited for a total of 0.75 seconds from the last input before giving a new input, the 0.5 seconds would be recorded as a player-decided delay that would be used during the playback. These delays were not implemented in any of the agents, which could had played a major role in differentiating them from the human recordings. Aiming at simulating different personas, the project did fail in testing anything but the speedrunner, as the networks performed horribly, and the premise was using them to learn from humans and simulate their actions. Having tested this using the FSM or BT could had given more data for comparison, but as FSM's and BT's allow for precise behavioral programming, making the adjustments and tests seemed was regarded more as a proof-of-concept than an actual comparison test. Had the networks performed better after the re-training, another test would had been preferable, but as this was not the case, attempting to simulate the other two personas with only the FSM and BT would had been less fruitful. This would be better to expand with in future projects with ML implementations that perform better than the ANN and CNN presented here.

Conclusion

Trying to simulate players is an interesting twist to the common ways of making agents that efficiently plays games. Creating a game that strips away as many mechanics as possible, and focuses on presenting only a small part of what normal games contain as interaction, resulted in a simple game to work with artificial intelligence (AI). By having a simple agent that was only capable of moving forward or back, or turning left or right, the game had limited the overall interaction to a minimum, which also reduced the number of possible states. Having a reduced number of states also reduced the overall number of possible inputs for the different AI's. So, analyzing the visible part of the map and using that information, it allowed the classic AI techniques to traverse the map, while the machine learning (ML) techniques had a harder time performing. Using classic AI techniques for games, such as finite state machines (FSM's) and behavior trees (BT's), can complete the task of creating human-like agents, whereas the same can be hard to accomplish with ML. FSM's and BT's allow for very precise programming of their behavior, where the decisions are chosen through simple logic that is based on the information they are allowed to gather. However, they do require some adjusting to simulate different personas, as their nature makes them very situation specific. Trying to address this problem through artificial neural networks (ANN's) and convolutional neural networks (CNN's) does give a more general way of creating these agents, but they can be highly unpredictable without proper guidance towards an optimal solution for playing the game. Adjusting the topology of an ANN does seem to change it a bit, but without more in-depth testing, and probably longer training sessions, finding an optimal topology might be a major issue. Furthermore, different technical issues with such networks can be hard to track down, as their functionality is mostly hidden between the input and output. Issues during their calculations can result in erratic training sessions, which translates to bad performances when controlling the agent. Due to the compact and general implementations required for adjusting and testing the networks, the number of variables become massive, and a single calculation error in one can cause a chain-reaction throughout the network's whole calculation. Considering the test results presented, spending time on writing the specific cases for the BT agent might be better spent than experimenting with and training ANN's and CNN's for long periods of time. As the BT seemed to perform better than the other techniques, also considering its advances over the FSM, letting the developer spend more time developing specific behaviors might be the better solution to this kind of problem. However, the small number of survey participants, the issues with the implementations, and the limited training time that the networks were given could bias these results heavily. A larger, more in-depth study would probably shed more light on the details that makes an AI agent seem more human-like. Taking this into consideration, trying to simulate players using machine learning can be seen as an advanced problem, even though there is research that shows it is possible to teach networks based on human playthroughs. Looking towards other ML techniques, or letting the ANN and CNN learn through unsupervised learning, especially for longer periods of time, might give better results then what was achieved for this project. This project did not properly succeed in simulating players with ML, even with the decisions to limit the available mechanics to the bare minimum, but it did show that it is possible through classic AI techniques, as long as time is allocated for it. Simulating players is a possibility, though it does seem to require mapping and reconstructing the different behaviors of a human player, which must be performed by a developer. Addressing this problem with other techniques than presented here, could give more promising results, though a less limited game might play a major role in succeeding in this endeavor. 2D tile games, such as rogue-like, are limited in available information, as each tile is either walkable or non-walkable, based on what type the tile is and what is on top of it. Moving away from this type of game could potentially allow for more information that could prove vital for network training by allowing more freedom for performing different actions.

Reflection

As discussed throughout, a lot of things should be improved in future similar projects. If the time is constrained, it would be wise to use implementations that are complete, as they should allow to get started quicker with testing, though they can be hard to find for Unity and C# at times. However, getting an understanding of these packages can sometimes be problematic, and being able to properly adjust for the necessary use case is not always possible, so these are concerns to keep in mind when choosing techniques and packages. Similar research could utilize the available repository from this project, and either replace or fix the problems with the implementations, which could speed up similar projects. This might present an issue, as the implementations can hard to understand, and there are a lot of issues to address. But getting a proper foundation for creating and training networks would allow for more time to be spent doing studies and optimizations to the agent behavior. Performing more studies with more participants would be a major improvement to this research. The project aimed at reaching a lot of people, but only reached 15. This left the survey result pool too low to make any proper analysis of the data or find any conclusions. As discussed, there might be some connection between demography, artificial intelligence (AI) knowledge, and perception of human-likeness in agents. This might be an interesting area to explore further, as it could hint at a proper target demography for performing comparison tests. If the demographic that has the keenest eye on spotting AI's is tricked, it could potentially mean that other demographics can be tricked with the same AI agents as well. Analyzing how different groups of people react, interact, and regard the AI controlled agent could reveal the last pieces of the puzzle for creating more human-like AI. If that is addressed, moving away from the classic techniques used here and focusing more on other machine learning methods could also yield promising results. AI agents that learn from supervised learning will always require a lot of data and time to learn, while AI agents that use unsupervised learning, such as reinforcement learning, could yield better results. Exploring different machine learning techniques would likely give different, possibly positive, results in comparison with classic techniques, as the goal is teaching an AI to learn the small things that human players do (even if they do not realize it themselves), which can be hard to implement with simpler AI. If focus is kept on the classic methods, explore ways of analyzing the map in detail using different algorithms. As the BT indicated, the depth-first search algorithm gave it an advantage over the finite state machine (FSM). This could be improved in different ways, e.g., using A* instead for searching for a proper route, but also allow the search algorithm to explore the whole visible area, as long as it does not leave that area. This same concept could be expanded to the networks as well, where their inputs comes from a map analysis instead of only feeding them the visible blocks. Including more player data, such as the delay, could also improve the network's attempt at simulating players, as it might make them wait before taking action, thereby simulating the indecisiveness. The most important thing when improving the networks will be giving them new inputs in some way, as the tests showed the networks getting stuck turning if they choose the option at start. As the inputs do not change in that case, it would be important to feed the network some new inputs, which could be based on a map analysis from the current heading of the agent.

References

Literature

Agarwal, M. (2017). *Back propagation in convolutional neural networks — intuition and code*. Retrieved May 14, 2021, from https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199

Albawi, S., Mohammed, T. A., & Al-Zawi, S. (2017). Understanding of a convolutional neural network. *2017 International Conference on Engineering and Technology (ICET)*, 1–6.

Arbel, N. (2018). *Convolutional backpropagation*. Retrieved May 14, 2021, from https://medium.com/@arbelnir/convolutional-backpropagation-cfa46cf6a24d

Barriga, N. A., Stanescu, M., Besoain, F., & Buro, M. (2019). Improving rts game ai by supervised policy learning, tactical search, and deep reinforcement learning. *IEEE Computational Intelligence Magazine*, *14*(3), 8–18.

Burgun, K. (2015). *Clockwork game design*. CRC Press.

Cerny, V., & Dechterenko, F. (2015). Rogue-like games as a playground for artificial intelligence–evolutionary approach. *International Conference on Entertainment Computing*, 261–271.

Compton, C. (2018). *Beating feature creep*. Retrieved May 10, 2021, from https://www.gamasutra.com/blogs/CalebCompton/20180918/326678/Beating_Feature_Creep.php

Cook, D. (2007). *Beating feature creep*. Retrieved May 10, 2021, from https://www.gamasutra.com/view/feature/129948/the_chemistry_of_game_design.php?page=2

Creswell, J. W., & Creswell, J. D. (2018). *Research design qualitative, quantitative, and mixed methods approaches*. SAGE Publications.

Galway, L., Charles, D., & Black, M. (2008). Machine learning in digital games: A survey. *Artificial Intelligence Review*, *29*(2), 123–161.

Guerrero-Romero, C., Lucas, S. M., & Perez-Liebana, D. (2018). Using a team of general ai algorithms to assist game design and testing. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Heaton, J. (2005). *Introduction to neural networks with java*. Heaton Research, Inc.

Holmgård, C., Liapis, A., Togelius, J., & Yannakakis, G. N. (2014). Evolving personas for player decision modeling. *2014 IEEE Conference on Computational Intelligence and Games*, 1–8.

Jefkine. (2016). *Backpropagation in convolutional neural networks*. Retrieved May 14, 2021, from https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/

Justesen, N., Bontrager, P., Togelius, J., & Risi, S. (2019). Deep learning for video game playing. *IEEE Transactions on Games*, *12*(1), 1–20.

Khan, S., Rahmani, H., Shah, S. A. A., & Bennamoun, M. (2018). A guide to convolutional neural networks for computer vision. *Synthesis Lectures on Computer Vision*, *8*(1), 1–207.

Levine, J., Bates Congdon, C., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., & Thompson, T. (2013). General video game playing.

Li, B., Luo, H., Zhang, H., Tan, S., & Ji, Z. (2017). A multi-branch convolutional neural network for detecting double jpeg compression. *arXiv preprint arXiv:1710.05477*.

Liapis, A., Holmgård, C., Yannakakis, G. N., & Togelius, J. (2015). Procedural personas as critics for dungeon generation. *European Conference on the Applications of Evolutionary Computation*, 331–343.

Liapis, A., Yannakakis, G. N., & Togelius, J. (2013). Sentient sketchbook: Computer-assisted game level authoring.

Lim, M. Y., Dias, J., Aylett, R., & Paiva, A. (2012). Creating adaptive affective autonomous npcs. *Autonomous Agents and Multi-Agent Systems*, *24*(2), 287–311.

McGonigal, J. (2011). *Reality is broken: Why games make us better and how they can change the world*. Penguin.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Oppy, G., & Dowe, D. (2020). The Turing Test. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Winter 2020). Metaphysics Research Lab, Stanford University.

O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.

Perez-Liebana, D., Liu, J., Khalifa, A., Gaina, R. D., Togelius, J., & Lucas, S. M. (2019). General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, *11*(3), 195–214.

Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., & Lucas, S. (2016). General video game ai: Competition, challenges and opportunities. *Proceedings of the AAAI Conference on Artificial Intelligence*, *30*(1).

Rumelhart, D. E., Durbin, R., Golden, R., & Chauvin, Y. (1995). Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, 1–34.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

Simpson, C. (2014). *Behavior trees for ai: How they work*. Retrieved April 29, 2021, from https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Skalski, P. (2019). *Gentle dive into math behind convolutional neural networks*. Retrieved May 30, 2021, from https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9

Smith, G. (2014). The future of procedural content generation in games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *10*(1).

Stanescu, M., Barriga, N. A., Hess, A., & Buro, M. (2016). Evaluating real-time strategy game states using convolutional neural networks. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–7.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99–127.

Swink, S. (2007). *Game feel: The secret ingredient*. Retrieved April 29, 2021, from https://www.gamasutra.com/view/feature/130734/game_feel_the_secret_ingredient.php

Yadav, N., Yadav, A., Kumar, M., et al. (2015). *An introduction to neural network methods for differential equations*. Springer.

Yannakakis, G. N. (2012). Game ai revisited. *Proceedings of the 9th conference on Computing Frontiers*, 285–292.

Yannakakis, G. N., & Togelius, J. (2018). *Artificial intelligence and games* (Vol. 2). Springer.

Videos & Tools

bias2402. (2021a). Behavior-tree-library [GitHub Repository]. Retrieved May 28, 2021, from https://github.com/bias2402/Behavior-Tree-Library

bias2402. (2021b). Map-generator [GitHub Repository]. Retrieved May 28, 2021, from https://github.com/bias2402/Map-Generator

bias2402. (2021c). Neural-networks [GitHub Repository]. Retrieved May 28, 2021, from https://github.com/bias2402/neural-networks

Jensen, T. (2021a). Agent 1 [Video]. Retrieved May 27, 2021, from https://www.youtube.com/watch?v=Gx2h3D0FCvA

Jensen, T. (2021b). Agent 2 [Video]. Retrieved May 27, 2021, from https://www.youtube.com/watch?v=3FFsR6lpaJ4

Jensen, T. (2021c). Agent 3 [Video]. Retrieved May 27, 2021, from https://www.youtube.com/watch?v=C9W5xwdoAWw

Jensen, T. (2021d). Agent 4 [Video]. Retrieved May 27, 2021, from https://www.youtube.com/watch?v=T2p7bux6tvY

Jensen, T. (2021e). Agent 5 [Video]. Retrieved May 27, 2021, from https://www.youtube.com/watch?v=kbkFoUhNOSc

Jensen, T. (2021f). Agent 6 [Video]. Retrieved May 27, 2021, from https://www.youtube.com/watch?v=jU43rbSbnFY

Unity. (2021). Scriptableobject [Docs]. Retrieved May 29, 2021, from https://docs.unity3d.com/2020.1/Documentation/Manual/class-ScriptableObject.html

Appendix 1

Example of a CNN debug output. A file with the full debug is among the extra materials.

Post-AF errorMap: — 0 — 0,04718655 — 0 — 0 — 0 — 0 — 0 — -0,08861361 — 0 — 0,08980591 — 0 — 0,07663006 — 0 — 0,03805995 — 0 — 0,05286022 — 0 — 0 — 0 — 0 — 0,007537778 — 0 — -0,01968932 — 0 — 0,07181613 — 0 — -0,03778671 — 0 — 0,03413215 — 0 — 0 — -0,02567658 — 0 — 0 — 0 — 0 — 0 — 0 — 0,06808291 — 0 — 0,004723124 — 0 — 0 — 0 — -0,0351277 — 0 — 0 — 0 — 0 — 0,06015857 — 0 — 0,03764351 — 0,03692606 — 0 — 0 — 0 — 0 — 0,002110223 — 0 — 0 — -0,03183278 — 0 — 0 — 0 — 0 — 0 — 0 — -0,04873314 — 0 — 0,03398458 — 0 — -0,001841402 — 0,004832574 — 0 — 0,03052304 — 0 — 0 — 0 — 0 — 0 — 0,004216022 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — -0,1124815 — 0 — 0 — 0,001611391 — 0,02838568 — 0 — -0,007548643 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — -0,03850503 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — 0 — Pre-FilterUpdate: — -1,006701 — -0,009475272 — -1 — -1,013045 — -0,01504001 — -1 — -1 — 0 — -1 — Post-FilterUpdate: — -1,011213 — -0,01204349 — -1 — -1,022606 — -0,02275142 — -1 — -1 — 0 — -1 — ANN Backpropagation: DesiredOuputs: — 0 — 0 — 0 — 1 — ActualOutputs: — 0,736408854090299 — 0,446227525824928 — 0,284878325794583 — 0,416887138683464 — OutNeuronBackprop: — Error: -0,736408854090299 — Gradient: -0,836385121601538 — OldWeights: — -0,0610470674284953 — 0,233101264216519 — 0,212954900792313 — 0,620766637670233 — 0,463380831975202 — 0,790445696464947 — 0,637963491323387 — -0,928548190709459 — Updated-Weights: — -0,0610470674284953 — 0,233101264216519 — 0,212954900792313 — 0,620766637670233 — 0,463380831975202 — 0,790445696464947 — 0,637963491323387 — -0,928548190709459 — Bias: 0,0418192560800769 — OutNeuronBackprop: — Error: -0,446227525824928 — Gradient: -0,238767382877907 — OldWeights: — 0,658130584125468 — -0,745723646015731 — 0,508589178094915 — -0,485744788071953 — -0,609689756114823 — -0,0503223375651624 — 0,213060531398775 — -0,00107862474447051 — UpdatedWeights: — 0,658130584125468 — -0,745723646015731 — 0,508589178094915 — -0,485744788071953 — -0,609689756114823 — -0,0503223375651624 — 0,213060531398775 — -0,00107862474447051 — Bias: 0,0119383691438954 — OutNeuronBackprop: — Error: -0,284878325794583 — Gradient: -0,087741901739937 — OldWeights: — -0,463842716749684 — -0,996054892426382 — -0,453435791401861 — 0,181669368958878 — -0,644419106489243 — -0,185353753708933 — -0,813669723371821 — 0,51740281727044 — UpdatedWeights: — -0,463842716749684 — -0,996054892426382 — -0,453435791401861 — 0,181669368958878 — -0,644419106489243 — -0,185353753708933 — -0,813669723371821 — 0,51740281727044 — Bias: 0,00438709508699685 — OutNeuronBackprop: — Error: 0,583112861316536 — Gradient: 0,183998409163378 — OldWeights: — 0,499840653268546 — -0,473168080427297 — 0,990318045015595 — -0,691248290096991 — -0,393343704470128 — -0,0209921249286235 — 0,476865136752308 — -0,628140970891407 — UpdatedWeights: — 0,499840653268546 — -0,473168080427297 — 0,990318045015595 — -0,691248290096991 — -0,393343704470128 — -0,0209921249286235 — 0,476865136752308 — -0,628140970891407 — Bias: -0,00919992045816891 — Hidden1NeuronBackprop: — GradientSum: -0,978895997056004 — UpdatedWeights: — 0,588484575780334 — 0,667965436199664 — 0,987726157525427 — -0,943181415062017 — 0,474867121071959 — -0,963819468377074 — 0,0669424846148783 — 0,519024627990567 — Gradient: -0,244468284845798 — Bias: 0,00466745163971886 — -0,245210027902019 — 0,134145572378368 — 0,0322030517422609 — -0,908309169070939 — -0,257456346069209 — -0,997141091151694 — -0,314662154444802 — -0,530371743035676 — Gradient: -0,13795178310571 — Bias: 0,00263551610796646 — -0,254021370436075 — 0,752421975020516 — -0,71350899884175 — -0,989079874935132 — -0,317547182234725 — 0,858366907508283 — -0,560343767311584 — 0,976483307767885 — Gradient: -0,242811824157055 — Bias: 0,00463638445107422 — 0,342121656677742 — 0,0396305411307283 — -0,702977955203027 — 0,701122205565275 — -0,586417190537982 — 0,19083054791709 — -0,993640555065889 — 0,830541888172991 — Gradient: -0,125575689539504 — Bias: 0,00239839013505778 — -0,866695642409239 — -0,836273439152294 — -0,319946817737048 — 0,512877365347407 — -0,460532925771798 — -0,334275636511983 — -0,452460706910333 — -0,107992878699672 — Gradient: -0,243802912356771 — Bias: 0,0046554598330416 — -0,631795576602125 — -0,924350568989455 — 0,220753475660809 — 0,0878453026934738 — 0,531041363035814 — 0,782766832868926 — 0,310811961680098 — 0,46390514702718 — Gradient: -0,242941617523978 — Bias: 0,00463777800167839 — 0,0074380696785814 — 0,577564450715466 — -0,0144873815656115 — -0,870640749982857 — 0,617162352249568 — 0,498383813304074 — 0,157296471836649 — 0,724820827937136 — Gradient: -0,243426325311132 — Bias: 0,00464807854412254 — 0,0301364758192266 — 0,0536677665327059 — -0,27943723708365 — 0,841372503825171 — -0,91466861400505 — -0,0879982342422 — -0,518810992836399 — -0,447966512035563 — Gradient: -0,235032421511465 — Bias: 0,00448704320250539 — Hidden2NeuronBackprop: — GradientSum: -1,71601085835141 — UpdatedWeights: — -0,103366761982146 — -0,72647366939414 — -

0,0637016757688027 — 0,568413195930614 — 0,963309229334495 — -0,680552026108164 — -0,673917087574451 — 0,231229565679668 — -0,64025997074333 — -0,628440376198124 — 0,982836551024968 — 0,47983230719335 — 0,0533277909519745 — 0,69361653816589 — 0,781036880696675 — 0,434495336112797 — -0,548041528811698 — 0,336420097079324 — 0,320844142381495 — -0,310049239224777 — 0,973789680736973 — -0,882011935991241 — 0,0147352325798642 — -0,0942175002276048 — -0,190646092961843 — 0,118390830754485 — -0,687960149575006 — 0,581252435958596 — -0,594381049086517 — 0,831186604607472 — 0,949818703322587 — 0,522768755221166 — 0,0166213489214988 — 0,997984501532272 — 0,203075876088383 — 0,0535604539576733 — 0,0307078180046323 — -0,0563575406821247 — -0,234302896649718 — -0,79309283466688 — 0,660624446189322 — 0,568803652920203 — -0,988051733927825 — -0,899377093603544 — -0,821995755574664 — -0,888054142653967 — 0,0422840337465908 — -0,278881705030278 — 0,353717291426713 — 0,221553127850198 — 0,823684935375901 — 0,529348465394857 — 0,378340622120696 — 0,0561636681929993 — -0,40173370177007 — -0,221354825990905 — 0,258791098025996 — -0,969484175541198 — -0,240940711107543 — -0,15508160141999 — -0,992591876533158 — -0,255169523533047 — -0,174389385233815 — -0,471446575350802 — -0,578259079520711 — -0,539932204196198 — -0,536789041728149 — 0,0553432894197028 — -0,509459337922493 — -0,272523573260998 — -0,596212481891835 — 0,508316011870427 — -0,429277006270959 — 0,113936977048375 — 0,0293263145859011 — -0,59501397218323 — -0,893960202063415 — -0,0858876738165913 — -0,272221744652941 — -0,302591950307876 — -0,923893202992106 — 0,590921555455272 — -0,772464855468117 — 0,184065823063285 — -0,99249833076843 — -0,57952976207227 — -0,855571866899529 — 0,9604576807285 — 0,399718203302342 — -0,575569297920712 — 0,794769355931678 — 0,000191993545830327 — -0,815416829574582 — 0,920778704770272 — -0,800500958133722 — -0,0842060302776313 — -0,256806961845982 —

Appendix 2

Laptop Specs:

- Motherboard: ASUS GL553VW
- CPU: Intel i5-6300HQ @ 2.30GHz
- RAM: 8.00 GB, DDR4, 2133MHz
- GPU: NVIDIA GeForce GTX 960m

Desktop Specs

- Motherboard: MSI MPG X570 Gaming Plus
- CPU: AMD Ryzen 7 5800X, overclocked to 4.1 GHz
- RAM: 32.00 GB, DDR4, 3600MHz
- GPU: ASUS GeForce GTX 1080 ROG Strix Gaming

Appendix 3

The survey for the Agent Comparison Test. CSV file with answers is among the extra materials.

1. Gender:
    - Male
    - Female
    - I'd rather not say
2. Age:
    - Input field

3. What is your highest completed education?:
    - Primary School
    - High School
    - University
    - Vocational Education
4. If you answered "University" or "Vocational Education" in the previous, what did you choose?:
    - Input field
5. How much do you play video games in an average week?:
    - 0-9 hours
    - 10-19 hours
    - 20-29 hours
    - 30+ hours
6. How much do watch streamers or videos of people playing video games in an average week?:
    - 0-9 hours
    - 10-19 hours
    - 20-29 hours
    - 30+ hours
7. How much, according to yourself, do you know about AI in general?:
    - Slider 1-10
8. How much, according to yourself, do you know about AI in games?:
    - Slider 1-10
9. How much do you think AI is used in games?:
    - Slider 1-10
10. How many games do you think contain AI?:
    - Slider 1-10
11. Do you think Player 1 is a human?:
    - Yes
    - No
12. Do you think Player 2 is a human?:
    - Yes
    - No
13. Do you think Player 3 is a human?:
    - Yes
    - No
14. Do you think Player 4 is a human?:
    - Yes

- No

15. Do you think Player 5 is a human?:
    - Yes
    - No
16. Do you think Player 6 is a human?:
    - Yes
    - No
17. Rank the players based on how human-like you find them:
    - Ranking matrix