1 Introduction

The topic of this MSc. dissertation is the teaching of neural networks to autonomously play games. This report serves to provide some background on the topic, as well as document the research, design, and implementation carried out to achieve the overall aim.

2 Overall Aim

The aim of this project was to create a neural network (NN) capable of autonomously playing through a skill-based video game, as well as an interface that sends game state information to the NN and converts the NN's outputs into usable control signals for the game. To complete the game, the NN would need to be capable of reacting quickly to chaotic situations with little margin for error. Such an NN will need to be competently trained, with a responsive and reliable supporting system.

3 Project Goals

In Section 3: Project Goals, a roadmap for development is presented. The details of this roadmap were decided based on literature review in Section 4: Research.

Section 3.1: Goal 1 – Software Architecture, entails creating a skeletal structure of the system as well as implementing and testing a basic game interface.

Section 3.2: Goal 2 - Neural Network details the implementation of a neural network and training algorithm, as well as a test that trains the network on a basic task.

The following sections share similar completion criteria. All criteria are restated for clarity, but information specific to that section is **bolded.**

Section 3.3: Goal 3 - Minimal Competency details the training of the NN to successfully complete **the first level.**

Section 3.4: Goal 4 - Moderate Competency details the training of the NN to successfully complete **the entire game.**

Section 3.5: Goal 5 - Advanced Competency details the training of the NN to successfully complete **the entire game at a speed comparable to world class speedrunners.**

Section 3.6: Project Timeline summarises the roadmap with all goals together.

3.1 Goal 1 - Software Architecture

Create a system of software consisting of 2 applications. The first application will be a modified version of the chosen game. The game chosen was Awaria by Vanripper (see Section 4.2: Game Selection), and the application name will be Awaria.exe. The second application will be created in Visual Studio is to eventually contain the neural network and training algorithm. This application is called MainApplication2.exe. For this goal, MainApplication2.exe will need to run an extensive test that verifies correct handling of game state and player inputs in both applications.

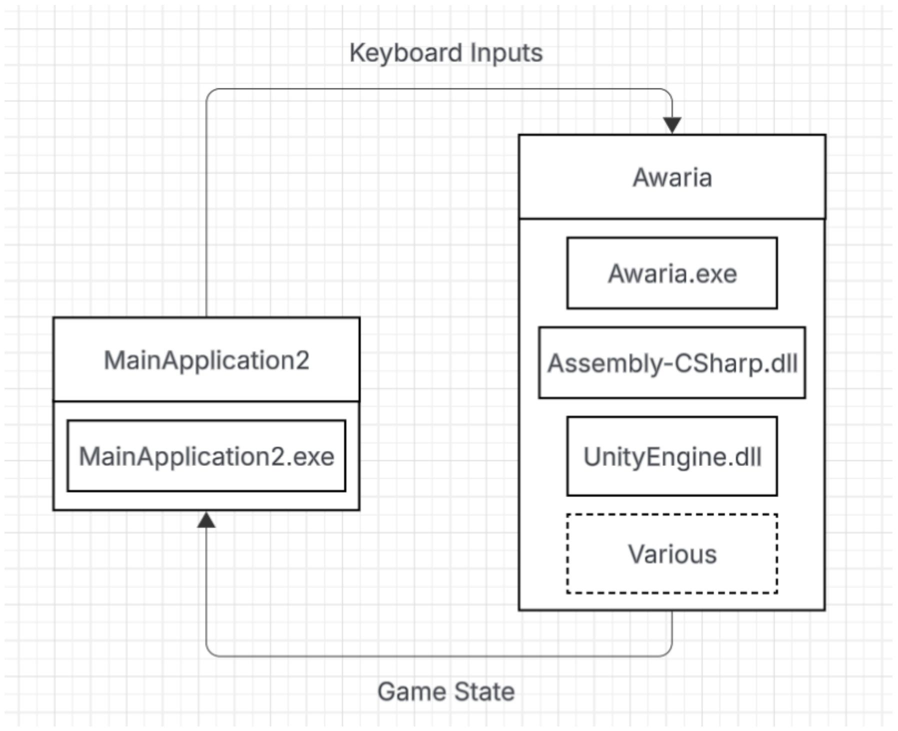


Fig. 1: A basic overview of the project software architecture. Arrows indicate information transfer.

3.1.1 Details of Awaria.exe

Exactly 2 of the files used by Awaria.exe are to be modified: Assembly-CSharp.dll, which contains the game's main logic, and AwariaInterface.dll, a new file that will serve as an interface between Awaria.exe and MainApplication2.exe.

3.1.1.1 Modification of Assembly-CSharp.dll

To minimise the risk of modifying gameplay, modifications in Assembly-CSharp.dll should primarily involve sending copies of elements of the game state to AwariaInterface.dll, and should never directly change game state data, except to replace control signals from the Unity engine with signals provided by AwariaInterface.dll. Assembly-CSharp.dll will also need to load AwariaInterface.dll early in execution.

3.1.1.2 Responsibility of AwariaInterface.dll

AwariaInterface.dll will receive game state data from Assembly-CSharp.dll and send it to MainApplication2.exe as a snapshot of the game state on that frame. It will also receive player input data from MainApplication2.exe and provide it to Assembly-CSharp.dll.

3.1.2 Details of MainApplication2.exe

This early iteration of MainApplication2.exe will not require a neural network or training algorithm, and the control signals it generates will be hard coded or generated algorithmically. MainApplication2.exe will be responsible for receiving game state snapshots from AwariaInterface.dll and sending control signals to AwariaInterface.dll. MainApplication2 will also need to, when launched with a specific command line argument, run a detailed test that verifies:

1. All sent player inputs result in a predictable change in the game state.
2. All aspects of the received game state can be manipulated in a predictable way.
3. The game state is received by MainApplication2.exe at an average frequency of at least 60 Hz.

The deadline for this goal is 3rd July.

3.2 Goal 2 - Neural Network

The NN and training algorithm are to be implemented in MainApplication2.exe. The NN chosen as well as its specifications are discussed in Sections 4.3-4.7. The training algorithm chosen is discussed in Sections 4.8-4.12.

For this goal, MainApplication2 will also need to, when launched with a specific command line argument, run a test that verifies the function of the NN and training algorithm. This test will first attempt to train the NN to always move west regardless of game state. To evaluate the success of the training, the test will then feed the NN an artificial, randomised game state and observe the output. The evaluation is to run 100 times, and the test is considered a success if the NN chooses to move west at least 95% of the time.

The deadline for this goal is 17th July.

3.3 Goal 3 - Minimal Competency

When MainApplication.exe is launched with a specific command line argument, followed by Awaria.exe being launched from Steam, the system is to play through the **first level** on hard mode **in at most 1 minute**. All player inputs that are sent from MainApplication2.exe to AwariaInterface.dll must only be dependent on the outputs of the NN, rather than being hard coded or generated based on the received game state, with the exception of menu navigation.

The deadline for this goal is 19th July.

3.4 Goal 4 - Moderate Competency

When MainApplication.exe is launched with a specific command line argument, followed by Awaria.exe being launched from Steam, the system is to play through the **full game** on hard mode **in at most 3 hours**. All player inputs that are sent from MainApplication2.exe to AwariaInterface.dll must only be dependent on the outputs of the NN, rather than being hard coded or generated based on the received game state, with the exception of menu navigation.

The deadline for this goal is 2nd August.

3.5 Goal 5 - Advanced Competency

When MainApplication.exe is launched with a specific command line argument, followed by Awaria.exe being launched from Steam, the system is to play through the **full game** on hard mode **as quickly as a world-class speedrunner.** To complete this goal, it must be statistically proven with >=95% confidence that in its fastest 5% of runs, the system can complete the game in **at most 13m00s317ms.** All player inputs that are sent from MainApplication2.exe to AwariaInterface.dll must only be dependent on the outputs of the NN, rather than being hard coded or generated based on the received game state, with the exception of menu navigation.

The deadline for this goal is 9th August.

3.6 Project Timeline

The project goals described in Section 3 can be grouped into 2 main phases: an implementation phase and a training phase. The implementation phase, consisting of Goals 1-2, entails implementing most of the features initially planned for the design. The training phase, consisting of Goals 3-5, entails training the NN using the existing system such that it can competently play Awaria. New features were expected to be added as needed during the training phase.

The Goals were intended to be completed sequentially, leading to a project timeline as follows:

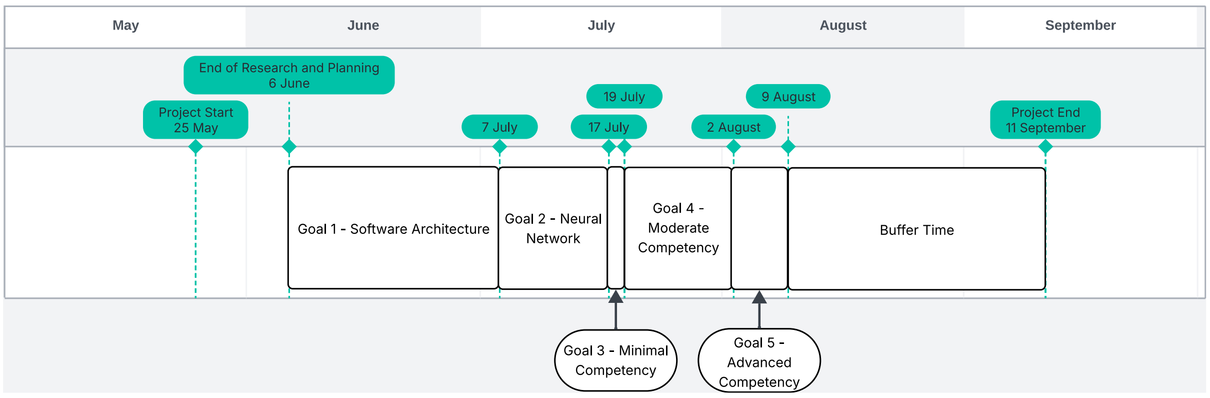


Fig. 2: A Gantt chart showing the initially planned project timeline.

4 Research

Section 4: Research contains 12 subsections, each with a specific research question. Sections 4.3-4.7 are related to the NN architecture, while Sections 4.8-4.12 are related to the training algorithm. A brief description of each subsection is as follows:

Section 4.1: Background details core concepts involved in training NNs to play games using machine learning.

Section 4.2: Game Selection details the selection of a game for the neural network to play as well as a general overview of the game selected.

Section 4.3: Network Architecture details the selection of an NN architecture.

Section 4.4: Details of Network Architecture details the structure and function of the chosen NN architecture.

Section 4.5: Hyperparameter Selection details the selection of hyperparameter values for the chosen NN architecture.

Section 4.6: Network Initialisation details the generation of initial values for the weights and biases of the NN.

Section 4.7: Activation Functions details the selection and distribution of neuron activation functions.

Section 4.8: Training Algorithm details the selection of an algorithm to train the NN to play the game.

Section 4.9: Details of Algorithm details the function of the chosen training algorithm.

Section 4.10: Algorithm Hyperparameters details the selection of hyperparameter values for the chosen training algorithm.

Section 4.11: Curriculum Learning describes a possible addition to the chosen training algorithm.

Section 4.12: Training Duration details research into the expected training time of the network.

4.1 Background

Before diving into research on how the system described in Section 2: Overall Aims would best be implemented, it was deemed prudent to conduct surface level research on how NNs play games, and why they in particular are chosen for that purpose.

Research Question 1: What are the mechanisms by which a NN can learn to play games?

A policy is a function that maps the game state to a particular action that an agent such as an AI or human is to take (Sutton, R. 2018). Given a goal (such as winning the game), the agent is not guaranteed to be able to play totally optimally towards that goal using only a policy, because a policy does not take into account previous game states or perfectly predict future random events. However, most games can be played competently using only a policy, and some can even be played optimally.

Neural networks are universal function approximators, meaning that with the correct weights and biases, a neural network can implement any continuous function (Hornik, 1989). For this reason, neural networks are often trained to play games by teaching them to approximate an optimal policy, such that the inputs to the neural network are the game state and the outputs can be interpreted as the appropriate action.

4.2 Game Selection

Selecting the appropriate game for this project was expected to be essential for success. A game that was too complex would take the NN too long to learn, while a game that was too simple risked demonstrating insufficient academic merit. Other factors such as game genre and length were also expected to greatly influence the viability of training and difficulty of implementation. Above all, the chosen game was not to have any risk of legal or ethical concerns.

Research Question 2: What is an appropriate game to use for this project?

4.2.1 Methodology

First, a set of mandatory and preferred criteria were created for potential games. A list of games was then created that met the mandatory criteria and were close to meeting the preferred criteria. Finally, the developer of each game on the list was contacted for permission to modify the game for this project.

4.2.1.1 Mandatory Criteria

- Must be **able** to be modified such that important data (such as damage taken, player position, etc.) can be provided to the neural network in real time.
- A rough section of the game must be chosen for the neural network to learn. This section should be quickly repeatable (maximum length 15 minutes) to facilitate training.
- Must be at least somewhat challenging, and primarily skill based.
- Must be single player, or able to be played in single player

4.2.1.2 Preferred Criteria

- Should be **easily** modified to fit the project.
- Should have a low amount of RNG to make it easier to learn.
- Should not be hardware intensive to give more room for suboptimal implementation and potentially reduce training time
- Should be able to be sped up during training.
- Should have clear criteria for successes and failures
  - Example of good success: gaining score
  - Example of good failure: dying
- Should use primarily or exclusively digital/discrete inputs (i.e. no mouse) to reduce complexity.
- Should be a game the author has a lot of experience in.

4.2.1.3 Selected List

The initial list compiled meeting the above criteria was as follows:

1. Audiosurf
2. Awaria
3. Bleed
4. Bleed 2
5. Bullet Heaven 2
6. Shantae and the Pirate's Curse
7. Super Meat Boy
8. They Bleed Pixels
9. UFO 50
10. Wings of Vi

Out of the options on this list, permission to modify Audiosurf, Awaria, and Bullet Heaven 2 was acquired.

4.2.1.4 Final Selection and Game Overview

Game Candidate	Easily Modifiable	Low RNG	Not hardware intensive	Able to be sped up	Clear success/failure criteria	No continuous inputs	Author's experience
Audiosurf	Unknown	Yes	Yes	No	Yes	Yes	Moderate
Awaria	Yes	Somewhat	Yes	Unknown	Yes	Yes	High
Bullet Heaven 2	Unknown	Unknown	Yes	Unknown	Yes	Yes	Low

After reviewing the 3 remaining options, Awaria was selected due to the ease of modification and the author's high level of experience with the game. Awaria is an indie game developed by Polish developer Vanripper and released in 2024. It is a challenging top-down action game in which the player navigates arena-like rooms, fixing generators using components found in the arena while avoiding hazards. The game is separated into 13 levels, each of which is completed by repairing a certain number of generators.

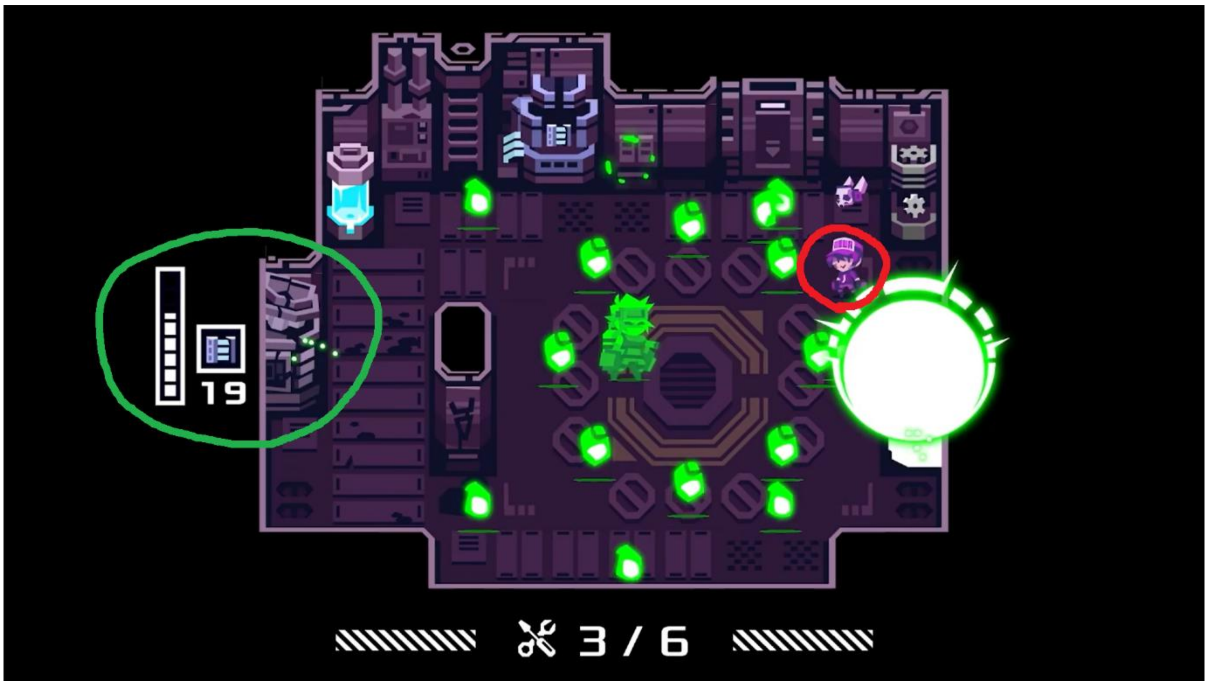


Fig. 3: A screenshot of gameplay in Awaria. The player is circled in red, and the generator is circled in green. Gameplay footage can be found at the Steam store page: https://store.steampowered.com/app/3274300/Awaria/

4.2.2 Summary

Awaria by Vanripper was chosen for this project.

4.3 Network Architecture

A NN is a set of neurons that is connected in some way, which collectively implement a function mapping a set of inputs to a set of outputs. A wide variety of NNs exist under this broad definition, some more appropriate for this project than others.

Research Question 3: What is an appropriate neural network architecture for this project?

The definition of an NN architecture as it is used here refers to the organisation of neurons, their connections, and if applicable, non-neuron structures such as convolutional layers.

4.3.1 Methodology

Creating a comprehensive list of all NN architectures was deemed impractical, but a list of plausibly useful architectures was compiled via ChatGPT and Google Scholar as follows:

- Multilayer perceptron (MLP)
- Convolutional neural network (CNN)
- Recurrent network (RNN)
- Deep dense architecture
- Bridged multilayer perceptron (BMLP)
- Fully connected cascade (FCC)

Each architecture on the list was then evaluated for viability in turn.

4.3.2 Multilayer Perceptron

Multilayer Perceptrons (MLPs) are the most simple and generalised modern NN. Almost all other modern NNs are modified versions of the MLP. (Hunter, 2012; Du, 2022).

Other network architectures are typically MLPs that are specialised in some area, such as Convolutional Neural Networks being specialised in large numbers of inputs. This always comes with a trade-off of some kind, typically increased computation time and complexity (Hunter, 2012; Yu, 2019). For this reason, the MLP architecture was chosen for the project, barring the discovery of another architecture specialised in an area relevant to the project.

4.3.3 Convolutional Neural Network

Convolutional neural networks (CNN) are perhaps the best known NN architecture used to play games due to AlexNet's (Krizhevsky, 2017) role in kickstarting the modern AI boom (Sejnowsky, 2018), as well as AlphaGo's triumph against European Go champion Fan Hui (Silver, 2016). CNNs are so called due to the presence of one or more convolutional layers between layers of neurons in the network, which serves to reduce the effective dimensions of the input data. CNNs are primarily used for applications with very large sets of input data, particularly high-resolution images (O'Shea 2015).

The reason CNNs are often used to play games is that they specialise in interpreting rendered graphics. However, as the NN inputs for this project were expected to be a minimal representation of the game state pulled from the game itself, and therefore lower dimension, CNNs were not chosen due to unnecessary complexity and computation time.

4.3.4 Recurrent Network

Recurrent Neural Networks (RNNs) are not strictly feedforward, as neurons from earlier layers may use neurons from future layers as inputs. These networks can be useful when short term memory is important, or the network can only be given a partial representation of the game state (Justesen, 2020). This was deemed unnecessary as short term memory was not considered particularly helpful in Awaria, and all relevant game state information should be able to be passed to the network in each frame.

4.3.5 Deep Dense Architecture

This architecture is an MLP with the original (game state) inputs concatenated into the inputs of each hidden layer after the first. The goal of this concatenation is to improve the performance of very deep neural networks with large numbers of inputs and actions (Sinha, 2020). As the number of inputs and actions for the project are small, and the network shallow, this design was dismissed due to unnecessary computation time and complexity.

4.3.6 Bridged Multilayer Perceptron and Fully Connected Cascade

Both Bridged Multilayer Perceptron (BMLP) and Fully Connected Cascade (FCC) are variants of the MLP that add extra connections across layers. BMLP and FCC were designed to reduce the minimum number of neurons to solve a parity-N problem, and experimentation confirmed their superior performance (Hunter, 2020). While the BMLP and FCC showed promise, the parity-N problem these networks were trained to solve is quite different from the game playing problem central to this project, and these designs lack the MLP's reputation for reliability and adaptability to different problems. Due to these concerns, the BMLP and FCC designs were put aside, to be revisited if the MLP design was deemed insufficient and time allowed.

4.3.7 Short Answer to RQ

The chosen NN architecture was multilayer perceptron (MLP).

4.4 Details of Network Architecture

To implement and train an MLP, it is necessary to fully understand the structure and operation of the architecture.

Research Question 3: what are the details of the chosen NN architecture?

An MLP consists of an input layer, an output layer, and one or more hidden layers. All layers are "fully connected", meaning each neuron in a layer is connected to every neuron in the layers directly before and after.

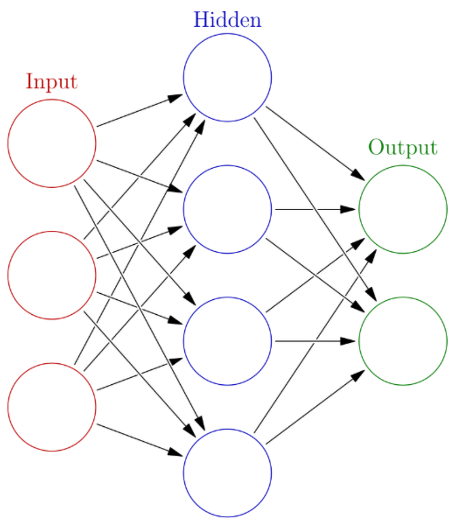


Fig. 4: An example of an MLP neural network with 1 hidden layer of 4 neurons and an output layer with 2 neurons.

The input layer represents the raw inputs to the network, while the hidden and output layers are made up of neurons. A neuron is a simple structure with the following features:

1. 1 or more input scalars
2. A weight scalar for each input
3. A bias scalar
4. An activation function

A functional neuron only needs 1 capability: calculating its output value. It does so via the following formula:

$$output = f[\sum_{i=1}^{n}(I_i * W_i) + B]$$

Where
B = bias
f(x) = activation function

I = vector of inputs
W = vector of weights corresponding to inputs

Neurons in an MLP may continuously calculate their outputs, but in practice typically perform the calculation as needed. An MLP is a type of "feed-forward network", so called because it propagates data 1 layer at a time from input to output, typically represented as left to right. Starting at the first hidden layer, all neurons in a layer calculate their outputs in any order. Then, the neurons in the layer directly to the right will calculate their outputs. Neurons of each layer beyond the first use the outputs of the previous layer's neurons as their inputs. The end result, or the outputs of the network, is the outputs of the neurons on the rightmost layer, which will need to be interpreted based on the training algorithm and the problem the network was trained to solve (Popescu, 2009).

4.5 Network Hyperparameters

In the context of neural network training, hyperparameters are parameters that are selected before training and are not updated through the learning process. Different architectures and training algorithms use different hyperparameters, and they must be selected based on the problem being solved. Poorly selected hyperparameter values can lead to increased training time or poorer performance once trained (Yang, 2020).

The hyperparameters to consider when training an MLP are number of hidden layers and number of neurons per layer. The initial values and activation functions of each neuron are arguably hyperparameters as well, but they are covered in Section 4.6: Network Initialisation and Section 4.7: Activation Function respectively.

Research Question 5: What are appropriate values for the NN hyperparameters?

4.5.1 Number of Hidden Layers

Uzair (2020) performed a survey of 10 neural networks created for different research projects with hidden layer counts ranging from 1 to 5. Uzair concluded that higher complexity problems required more hidden layers to achieve acceptable performance once fully trained, and increasing the layer count tended to provide higher accuracy overall. However, increasing the layer count also increased the required training iterations. Uzair also mentioned that too many layers or neurons can lead to overfitting (poor performance on data not in training set). Uzair found that 3 hidden layers were optimal on average but recommended changing the number as needed if there are symptoms of overfitting or underfitting.

In supervised and unsupervised learning (discussed in Section 4.8.2: Machine Learning Approaches), overfitting is a serious concern, as Awaria has virtually infinite game states, meaning that the chosen training data cannot be 100% representative of the game. However, as discussed in Section 4.8.2, reinforcement learning was chosen over these options. Reinforcement learning is much more resistant to overfitting than its competitors, as the NN is continuously evaluated using situations not in the set of training data, and significant symptoms of overfitting should be corrected just like any other suboptimal behaviour. Because reinforcement training was selected, overfitting was deemed less of a concern for this project, and a hidden layer count of 5 was tentatively selected. However, due to concerns raised by research in Section 6.11. Training Duration, the hidden layer count was later reduced to 2 to minimise training time.

4.5.2 Number of Neurons Per Layer

As discussed in Section 4.4: Details of Network Architecture, the input layer of an MLP is not composed of neurons, but the hidden and output layers are. The training algorithm selected in Section 4.8: Training Algorithm mandates that the number of neurons in the output layer is equal to the number of unique actions the agent can take. Thus, the remaining hyperparameter is the number of neurons per hidden layer.

Hunter (2020) found that while NN competence and training time improved with increasing neuron count, the computation time increased as well. Hunter also mentioned that an excessively large neuron count can leave to overfitting. Increasing the number of neurons will also change the optimal values for other hyperparameters, which may make it more challenging to find values that result in acceptable convergence (Chen, 2018).

Ultimately a neuron count of 64 per hidden layer was chosen, as it is the default for Stable Baseline (Stable Baselines 2021) and PyTorch (Strunk, 2024).

4.5.3 Short Answer to RQ

2 hidden layers was selected, each with 64 neurons.

4.6 Network Initialisation

As mentioned in Section 4.4: Details of Network Architecture, each neuron has a set of weights and a bias. While initialisation may at first seem trivial, an untrained network is typically best initialised with specific and appropriate nonzero values, as it can improve training speed and stability.

Research Question 6: What values should be used to initialise the weights and biases of the NN?

4.6.1 Initial Research

Dawson (1998) recommends initialisation of both weights and biases to random values in the range (-2/X, 2/X) for a neuron with X inputs. As described in Section 5.3.3: Obstacles Encountered During Goal 2, using this initialisation method resulted in gradient explosion, or a rapid uncontrolled increase in NN weights and biases during training. As a result, an alternative method was explored.

4.6.2 He Initialisation

An alternative initialisation method is to set each bias to 0 or a small magnitude constant, and to set each weight to a value sampled from a normal distribution with parameters (mean = 0) and (standard deviation = $\sqrt{\frac{2}{X}}$), where X is the number of inputs to the neuron (He, 2015). Switching to this initialisation method temporarily resolved problems with gradient explosion.

4.6.3 Short Answer to RQ

Initial values for an untrained network were chosen as follows: bias = 0.01, weight = random sample from $norm(0, \sqrt{\frac{2}{fan\_in}})$

4.7 Activation Function

As mentioned in Section 4.4: Details of Network Architecture, each neuron has an activation function used when calculating its output. The activation function determines in part the behaviour and performance of the NN, particularly in the output layer, where it can determine the discretisation and bounds of the output values.

Research Question 7: What activation functions should be used for each neuron in the NN?

4.7.1 Initial Research

Typically, a neural network has 2 activation functions. Each neuron in the hidden layers shares one activation function, and each neuron in the output layer shares a second activation function (Rodríguez, 2022).

For hidden layers, the ReLU function is the default. It is common, easy to use, and works for most applications. It's generally recommended for all MLPs (Rodríguez, 2022).

The three most popular output layer activation functions are linear, logistic, and softmax. The choice of output function depends on what the output values are meant to represent. For a linear regression, the linear function is best used. To classify the input data into 1 of >2 classes, softmax is used. To classify data into 1 of 2 classes, or >1 of >=2 classes, sigmoid is used (Rodríguez, 2022).

Softmax could have been reasonably chosen for this project if each output neuron represented a potential action the agent could take, but sigmoid was chosen with the intent that each output neuron represented a button that could be pressed (i.e. up, down, left, right, enter, esc, space). The sigmoid implementation would require fewer output neurons, as there are ~18 actions the agent can take, but only 7 buttons.

4.7.2 Deep-Q Activation Functions

As discussed in Section 4.8: Training Algorithm, the Deep-Q training algorithm was selected for this project. The outputs of a NN trained with the Deep-Q algorithm (also known as the DQN) each correspond to an action the agent can take and represent the cumulative rewards that the DQN expects to receive for taking that action in its current situation. These expected cumulative rewards are known as Q-values. This more closely resembles the type of problem a linear activation function is suited for, particularly because softmax and sigmoid always output values between 0 and 1 (Rodríguez, 2022), which does not allow them to properly express Q-values.

4.7.3 Short Answer to RQ

The chosen activation functions for the network were ReLU for all hidden neurons and linear for all output neurons.

4.8 Training Algorithm

Simple observation is insufficient for a NN to learn to play a game, as it does not by default have any means to modify its own weights and biases nor determine how they should be changed. Instead, NNs are trained by an algorithm that iteratively changes the weights and biases over time to produce a desired output. There are many types of training algorithm, each with their own strengths and weaknesses.

Research Question 8: What is an appropriate training algorithm to choose for this project?

4.8.1 Methodology

To select an appropriate algorithm, 3 broad categories were created, representing nearly all relevant candidates. These categories were then evaluated at a surface level, with one category being chosen as the most suitable. The category was then divided into subcategories, and the process was repeated until a manageable list of candidates remained. Finally, each candidate was evaluated in turn. Subsections 4.8.2-4.8.4 detail iterations of this process, and Subsection 4.8.5 details the evaluation of the final list.

4.8.2 Machine Learning Approaches

There are 3 families of machine learning algorithm in common modern use: supervised learning, unsupervised learning, and reinforcement learning. All 3 families train a NN to solve some sort of problem using a finite set of training data, resulting in a NN that can not only solve that problem, but through generalisation, similar problems as well. (Sutton, 2018).

4.8.2.1 Reinforcement Learning

Reinforcement learning involves allowing an untrained NN to attempt to solve the problem presented to it repeatedly. Through attempting and most likely failing to solve the problem, the network learns to take more optimal actions over time. This family is particularly useful for problems in which it is difficult to create a set of training data ahead of time that is sufficiently detailed to encode all the information required to effectively train the NN (Sutton, 2018).

4.8.2.2 Supervised Learning

Supervised learning involves training NN using a carefully preselected set of training data. Each datapoint in the set describes a situation and the correct action to take in that situation. Through supervised learning, a NN can learn the general relationships between different aspects of the situation and their effect on the correct outputs (Sutton, 2018).

4.8.2.3 Unsupervised Learning

Unsupervised learning also involves training a NN using a preselected set of training data. It differs from supervised learning in that the NN is trained to generate an approximation of its training data. Rather than providing the correct output for its situation, it will reproduce its training data regardless of the situation (Sutton, 2018).

4.8.2.4 Most Suitable Category

Reinforcement is typically used to train NNs to play games, as the state space (the set of possible game states) is large if not virtually infinite, and it is often impractical to create a premade set of training data that is sufficient to represent that state space (Sutton, 2018). For these reasons, reinforcement learning was chosen for this project.

4.8.3 Reinforcement Learning Approaches

Sutton (2018) describes 3 varieties of reinforcement learning: dynamic programming, Monte Carlo methods, and temporal-difference learning.

4.8.3.1 Dynamic Programming

A dynamic programming algorithm trains a NN with an optimal policy for a problem but requires a perfect model of the environment to do so. That is, for a given state, the probability distributions of all possible next states must be readily available. In practice, this is seldom appropriate for a network that learns through experience.

4.8.3.2 Monte Carlo Methods

Unlike dynamic programming algorithms, Monte Carlo methods require no prior knowledge of the environment, allowing the NN to learn solely from experience. A Monte Carlo method can only be used to train a NN to solve an episodic problem. That is, a repeatable task for which a terminal state is reached regardless of actions taken.

Each repetition is termed an episode, and after the completion of each episode, the NN is updated such that it more closely approximates an optimal policy.

4.8.3.3 Temporal Difference Learning

Temporal difference algorithms differ from Monte Carlo methods only in that they do not need to wait for the end of an episode to update the NN. Temporal difference algorithms can update the NN as often as every time step, and in fact do not need episodes at all. This makes temporal difference algorithms more responsive and better at learning from suboptimal actions, and they are particularly well suited to non-episodic problems or problems with long episodes (Sutton, 2018).

4.8.3.4 Most suitable category

Dynamic programming was deemed non-viable, as the number of possible game states in Awaria is virtually infinite, making a complete model of the game impractical. Both Monte Carlo methods and temporal difference algorithms are theoretically viable, and in some applications both may offer the same performance. However, Sutton (2018) specifically mentions that temporal difference algorithms tend to outperform Monte Carlo methods on stochastic (probabilistic) tasks. As Awaria has random elements and the computer running the system has unavoidable unpredictability due to interference from the operating system and other processes, temporal difference was chosen as the preferred category.

4.8.4 Temporal Difference Learning Approaches

There are two types of temporal difference approaches, tabular and approximate. Tabular methods are capable of finding the exact optimal policy for a problem but are only likely to do so if the state space (the set of all possible states) is small. Approximate algorithms only approximate the optimal policy but outperform tabular algorithms when the state space is large. The cutoff point varies based on several factors and is therefore difficult to determine, but for this project the choice was clear. Awaria has a virtually infinite state space, so an approximate algorithm would be optimal.

4.8.5 Final Selection

After narrowing down the set of potential training algorithms, two viable candidates remained: Q-learning and SARSA. Q-learning and SARSA are quite similar and only differ in that SARSA is on-policy while Q-learning is off-policy. In both cases, the NN has an ε% chance to take a random action and a (1- ε)% chance to take what it thinks is the optimal action in each situation. After taking this action and observing the result, the NN then predicts the value of its follow up action. In an on-policy algorithm, the predicted follow up action is subject to the same ε% chance of random selection as its predecessor, whereas in an off-policy algorithm, ε is set to 0 for the follow up action (Sutton, 2018).

Both SARSA and Q-learning have improved modern variants that have successfully learned various games. Named Deep-Q (Mnih, 2015) and Deep SARSA (Zhao, 2016), neither appears to be clearly superior to the other (Zhao, 2016). However, Deep-Q is much more well known (Arulkumaran, 2017) and widely used. Due to the more proven status of Deep-Q as well as the wealth of documentation available, it was chosen over SARSA.

4.8.6 Short Answer to RQ

The selected training algorithm is Deep-Q.

4.9 Details of Algorithm

The Deep-Q training algorithm is quite complex and challenging to implement, so it would be prudent to gain a thorough understanding of the process before implementing Deep-Q.

Research Question 8: What are the details of the chosen algorithm?

Deep-Q is a form of Q-learning specialised for machine learning. It was originally created to train NNs to play various Atari games (Mnih, 2015) and has since become well known (Arulkumaran, 2017), particularly for playing games and image processing (Krizhevsky, 2017). The goal of Deep-Q is to train a NN to take the optimal action when presented with any environment state.

Sections 4.9.1 through 4.9.3 detail the Deep-Q algorithm, only briefly mentioning Stochastic Gradient Descent (SGD). SGD is an extensive component of Deep-Q and is described independently in Section 4.9.4 for clarity.

4.9.1 Markov Decision Processes

As mentioned in Section 4.8.2.1: Reinforcement Learning, the goal of reinforcement learning is to allow the NN in training to interact with its environment, and through the resulting experience adjust the NN to more closely approximate an optimal policy. In Deep-Q, the agent's interactions with its environment are modelled by a Markov Decision Process (MDP).

MDPs are a formalisation of a decision-making sequence, wherein the interactions between the agent and its environment are separated into discrete time steps. A time step starting at time t0 and ending at time t1 can be fully described by an initial environment state S0, the action taken in response to that state A0, a final environment state S1, and a reward received for the transition R1 (Sutton, 2018). Note that the reward may be generated by the environment or artificially by a training algorithm.

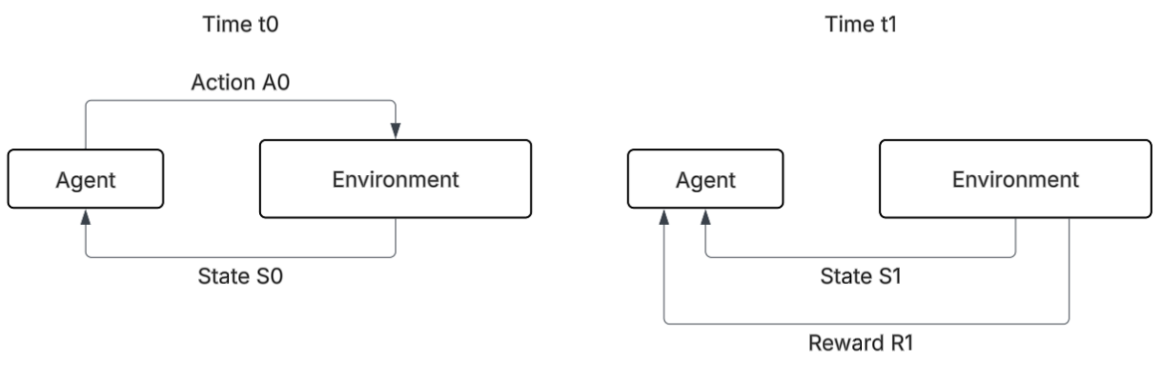


Fig. 5: A representation of a single time step in a Markov Decision Process.

As an MDP is a chain of these time steps, S1 is then used to generate A1, and so on.

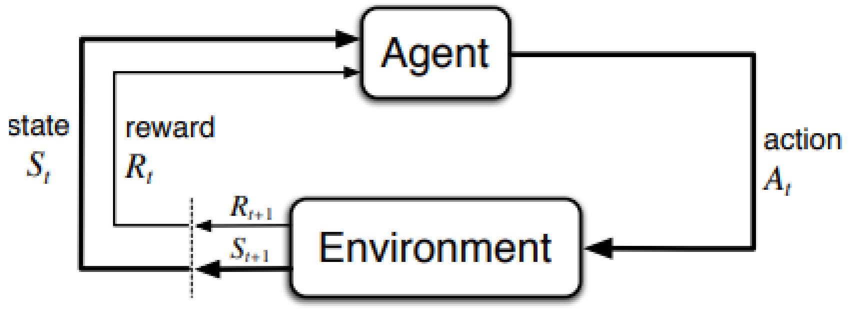


Fig. 6: A representation of a Markov Decision Process consisting of many time steps (Sutton, 2018).

4.9.2 Structures

The Deep-Q algorithm requires several data structures as follows:

4.9.2.1 Transitions and Replay Buffer

A transition is Deep-Q's representation of a single time step in the MDP. It contains S0, A0, S1, and R1, and additionally F, which indicates whether the transition is the end of an episode. When a new transition is recorded, it is added to a large FIFO buffer called the replay buffer. When training, transitions are then sampled from the replay buffer (Mnih, 2015).

4.9.2.2 Deep-Q Network (DQN)

The Deep-Q Network (DQN) is the NN being trained by the algorithm. While the architecture of the DQN may vary, Deep-Q poses certain requirements about the DQN's input and output layers. The input layer, which simply passes input data to the network, receives an environment state. The output layer provides a numerical value for each possible action the agent can take. Each output value, while garbage data at first, will be gradually trained by Deep-Q to represent the expected cumulative reward for the remainder of the episode, given that the agent takes the action corresponding to that output (Mnih, 2015). In other words, beneficial actions have higher corresponding values, while detrimental actions have lower corresponding values. This expected cumulative reward is called a Q-value, and by always taking the action with the highest Q-value, a correctly trained network will follow the optimal policy (Sutton, 2018).

4.9.2.3 Target Network

The Target Network is a periodically updated copy of the DQN, and during training, both the DQN and target network are given respective environment states. The Q-values of the target network are then used in certain formulas in place of DQN Q-values. The purpose of the target network is to stabilise learning, particularly by reducing oscillation and divergence (Mnih, 2015).

4.9.3 Algorithm

The Deep-Q algorithm assumes a reinforcement training framework in which an agent and environment may interact, forming an MDP, as well as appropriately selected hyperparameters ε, batch size, and target network update frequency, in addition to those required by stochastic gradient descent (see Section 4.10: Algorithm Hyperparameters). Once the DQN and target network are initialised (see Section 4.6: Network Initialisation), Deep-Q loops through its core algorithm until indefinitely. Each loop of the algorithm is a single time step in the MDP and thus adds a single transition to the replay buffer.

The core algorithm is as follows:

1. Observe the current environment state S0.
2. Propagate S0 through the DQN.
3. Select an action A0 for the agent to take:
    a. ε% chance: random action.
    b. (1 - ε)% chance: the action corresponding to the highest Q-value of the DQN.
4. Observe the following environment state S1.
5. Calculate or observe a corresponding reward R1.
6. Create a new transition for the current timestep consisting of S0, A0, S1, R1, F, where F = true if the state S1 is terminal.
7. Add this transition to the replay buffer.
8. Take a random sample of size Batch Size from the replay buffer.
9. Using the sample, the target network, and the DQN, perform SGD (see Section 4.9.4: Mini-batch Stochastic Gradient Descent) to alter the weights and biases of the DQN to values that more closely approximate the optimal policy.
10. If appropriate (depending on target network update frequency), set the target network to be a copy of the DQN.(Mnih, 2015).

4.9.4 Mini-batch Stochastic Gradient Descent (SGD)

Mini-batch Stochastic Gradient Descent is a maths heavy algorithm component of Deep-Q and is typically responsible for the majority of the CPU load during training. During a Deep-Q training step, mini-batch SGD is used to iteratively adjust the weights and biases of the DQN to more optimal values via 3 phases: forward propagation, backpropagation, and final adjustments.

4.9.4.1 Background

A concept central to Deep-Q and many other machine learning algorithms is gradient descent. Given a function that contains multiple coefficients defining its behaviour, gradient descent is an algorithm that can be used to iteratively change that function's coefficients to minimise its output. In machine learning, the coefficients in question are weights and biases, whereas the output being minimised is the loss function, which itself is a function of the output of the NN.

Reinforcement learning approaches typically utilise *stochastic* gradient descent. This differs from standard gradient descent in that its iterative updates are only approximately correct but can be calculated from incomplete data. This is necessary for reinforcement learning as only a small portion of the possible environment states is known at any time.

The mini-batches used in Deep-Q are not necessarily required for reinforcement learning or even Q-learning but can speed up and stabilise training. A mini-batch SGD step calculates weight and bias adjustments for each transition in a batch, then applies the average of these adjustments to the DQN. This batch of transitions is randomly sampled from the replay buffer, a FIFO array of recent agent experiences.

4.9.4.2 SGD Symbol Glossary

The following symbols are referenced by Equations 1-5 in the following sections.

a^l: 1D array; the outputs of the neurons of layer l. Includes the input layer.
b^l: 1D array; the biases of neurons of layer l
δ^l: 1D array: the error of neurons of layer l
η: scalar; hyperparameter; the learning rate or learning coefficient
$f'(x^l)$: 1D array; the derivative of the activation function shared by neurons in layer l
γ: scalar; hyperparameter; the discount
i: identifier; current element of batch/sample (δ_i would indicate the error of transition i).
l: identifier; current layer (δ^l would indicate the error of layer l).
L: identifier: l at output layer
n: scalar; hyperparameter; batch size
W^l: 2D array; the weights of connections between layers l-1 and l
y: scalar; The target Q-value
z^l: 1D array; the preactivation values of the outputs of the neurons of layer l
∘: Operator indicating Hadamard product. Inputs are any 2 equal dimension equal sized vectors; output is a vector of that same dimension and size.
Structure of an element of the replay buffer (a transition)

S0: Initial state
A0: Action taken
S1: Final state
R1: Reward given
F: “Training iteration complete” flag

4.9.4.3 Forward Propagation

Forward propagation is the act of calculating the outputs of the DQN and target network after providing them with S0 and S1 of a transition respectively, then copying various intermediate values from the calculation and storing them for use in backpropagation and final adjustments. In standard SGD, forward propagation is only performed once per iteration, while in mini-batch SGD it is performed for each transition in the batch. A more sequenced description of forward propagation is as follows:

1. Feed S0 into the DQN.
2. Propagate the data through the DQN.
3. Create a copy of the following values in the DQN.
   a. W^l for all layers
   b. z^l for all layers
   c. a^l for all layers
4. Feed S1 into the Target Network.
5. Propagate the data through the Target Network.
6. Copy the Q-value corresponding to A0 from the Target Network, which we will call TNO.

(Sutton, 2018).

4.9.4.4 Backpropagation

The backpropagation phase of SGD calculates the error δ of each neuron in the DQN for a transition. This is done using Equations 1, 2, and 3. Note that Equations 2 and 3 are not generally true for all SGD implementations. They are specific to the loss function (MSE), output activation function (linear), and hidden layer activation function (ReLU) used by Deep-Q (Mnih 2015).

In standard SGD, backpropagation is only performed once per iteration, while in mini-batch SGD it is performed for each transition in the batch. A more sequenced description of backpropagation is as follows:

1. Calculate y using the sampled transition and the following formula:
   $$y = \begin{cases} R1, F = 1 \\ R1 + \gamma TNO, F = 0 \end{cases}$$
2. Calculate ${\delta^L}_i$:
   a. Calculate an unclipped value of ${\delta^L}_i$ using Equation 2.
   b. If any values in ${\delta^L}_i$ are less than -1, set them to -1. If any are greater than 1, set them to 1 (Mnih, 2015).
3. For each layer l in the DQN, starting at $l = L - 1$, calculate ${\delta^l}_i$:
   a. Calculate an unclipped value of ${\delta^l}_i$ using Equation 1, Equation 3, and ${\delta^{l+1}}_i$.
   b. If any values in ${\delta^l}_i$ are less than -1, set them to -1. If any are greater than 1, set them to 1 (Mnih, 2015).

${\delta^l}_i$ is defined recursively as

$$\boxed{\text{Eq. 1}}\quad {\delta^l}_i = f'({z^l}_i) \circ [(W^{l+1})^T {\delta^{l+1}}_i]$$

With a base case of ${\delta^L}_i$, where ${\delta^L}_i$ = an array with size sizeof(a^L) such that

$$\boxed{\text{Eq. 2}}\quad {\delta^L}_i[k] = \begin{cases} 2({a^L}_i[k] - y_i), k = action \\ 0, k \neq action \end{cases}$$

And an expansion of the shorthand $f'({z^l}_i)$ as $f'({z^l}_i)$ = an array with size sizeof(z^l) such that

$$\boxed{\text{Eq. 3}}\quad f'({z^l}_i)[k] = \begin{cases} 1, ({z^l}_i)[k] > 0 \\ 0, ({z^l}_i)[k] \leq 0 \end{cases}$$

(Nielsen, 2015).

4.9.4.5 Final Adjustments

The final phase of mini-batch SGD modifies the weights and biases of the DQN using values from both forward propagation and backpropagation. This is done using Equations 4 and 5. Note that the averaging operation $\frac{1}{n}\sum_{i=1}^{n}[]$ in these equations serves to combine data from all transitions in the mini-batch and is therefore omitted in standard SGD.

$$\boxed{\text{Eq. 4}}\quad W^l <= W^l - \eta\frac{1}{n}\sum_{i=1}^{n}[{\delta^l}_i({a^{l-1}}_i)^T]$$

$$\boxed{\text{Eq. 5}}\quad b^l <= b^l - \eta\frac{1}{n}\sum_{i=1}^{n}{\delta^l}_i$$

(Nielsen, 2015; Mnih, 2015).

4.10 Algorithm Hyperparameters

A hyperparameter in machine learning is some parameter that is selected before training and not updated during the learning process. The Deep-Q algorithm has several hyperparameters, including those specific to SGD (batch size, learning rate η, and discount γ), Q-learning (ε), and Deep-Q itself (replay buffer size, target network update frequency).

Research Question 9: What are appropriate values for Deep-Q's hyperparameters?

4.10.1 Methodology

When selecting hyperparameters for a NN and its algorithm, it is typically best to tune each to fit the problem at hand through iterative updates and practical experimentation (Yang, 2020). However, due to time constraints, hyperparameters were instead initially chosen by reviewing multiple successful projects and copying hyperparameters from the most relevant sources, preferring reliability over performance. The aim of this method was to reduce or eliminate the time spent fine tuning hyperparameters, on the basis that the architecture and algorithm chosen had a reputation for excellent generalisation, and that slightly suboptimal hyperparameters would be unlikely to pose a significant obstacle in training.

4.10.1 Batch Size

When updating the NN's weights and biases through SGD, a specific scalar is calculated for each weight and bias and added to the current value. Each time step, when the Deep-Q core algorithm loops, a sample of size "batch size" is taken from the replay buffer and used for training. The gradients will then be computed using the combined experience of all the transitions in the sample (Mnih, 2015). Mnih's design used a batch size of 32, and Masters (2018) found that best performance with batch sizes 2-32, so 32 was chosen for the project.

4.10.2 Learning Rate

The learning rate η is a coefficient used in SGD. During a Deep-Q training loop, the last step in calculating each gradient is multiplying it by the learning rate, which is the same value for each weight and bias (Sutton, 2018). Multiplying the learning rate by 10 does not necessarily mean the NN will learn 10x faster, even in the best case, because the experience available to learn from and the granularity of the weight/bias updates are still limited.

A learning rate that is too low increases the total required training time, as it takes longer to move weights and biases to their optimal values. A learning rate that is too high reduces training stability and can cause the training to fail entirely (Yang, 2020).

The original Deep-Q implementation used a learning rate of 2.5e-4 (Mnih, 2015). However, Obando-Ceron (2023) found that with a batch size of 32, 2.5e-4 may be a bit on the high side, and as the selection priority is reliability, a learning rate of 1e-4 was ultimately chosen.

4.10.3 Discount

The discount is a coefficient used in SGD to represent the higher relative value that immediate rewards have over delayed rewards. A discount of 1 indicates that future rewards have equal value to immediate rewards, while a discount of 0.5 indicates that a reward on time step t+2 is worth half as much to the agent as the same reward on step t+1 (Sutton, 2018). Mnih (2015) used a discount of 0.99, as did Paszke (2025). As such, 0.99 was selected for the project.

4.10.4 ε

ε does not strictly fit the definition of hyperparameter as it changes during training, but initial and final values are typically preset, as is the function that changes the value of ε over time. ε is the probability that the agent will take a random action on a given time step, also known as an exploration action. The purpose of an exploration action is to explore the environment and potentially stumble upon more optimal policies, regardless of what the NN currently believes is optimal, a mechanism called ε-greedy exploration (Sutton, 2018). ε starts high and decreases over time, the reason being that without sufficiently low ε, the agent may not be able to progress far enough through the episode to gather relevant experiences, even if it knows the optimal path. When a NN is fully trained, it knows the optimal action to take in any situation, and there is no longer a reason for it to explore, thus ε is typically set to 0. Deep-Q can continue to train the NN while ε=0 however, as the action that the NN considers optimal may change over time.

While ε generally starts at 100%, the gradual decrease of ε to 0 can be implemented in many different ways, the efficacy of each depending much more on the environment than the NN architecture or other hyperparameters. Mnih (2015) linearly decreases ε from 1.0 to 0.1 over the first 2% of training, then maintains ε at 0.1 for the remaining 98%. Paszke (2025) linearly decreases ε from 0.9 to 0.01 over 2.5 thousand time steps, then maintains ε at 0.01 for the remainder of the training. The implementation that was chosen for this project was to decrease ε from 1.0 to 0.05 over 5 thousand episodes, followed by maintaining ε at 0.05 indefinitely.

4.10.5 Replay Buffer Size

The replay buffer is a FIFO array that stores the most recent experiences of the agent and is sampled from each time step to use in SGD (Mnih, 2015). If a replay buffer is too large, there is increased risk of sampling less relevant transitions. For example, if the NN already knows how to beat levels 1-10, sampling a transition from level 1 is certainly helpful and does improve the NN, but not as much as a transition from level 11. If the replay buffer is too small, recent experiences may become oversampled, increasing the risk of forgetting- or rather, overwriting- older behaviours or learning too much from a single misleading transition.

Mnih (2015) uses a replay buffer size of 1 million, Paszke (2025) uses a replay buffer size of 10 thousand, and Stable Baslines (2021) uses a replay buffer size of 50 thousand. The much larger size of Mnih’s buffer may be due to the much larger amount of training steps (50 million vs Paske’s ~150 thousand). A value of 100 thousand was chosen on the grounds that curriculum learning (discussed in Section 4.11) should counter some of the negative effects expected of a small replay buffer.

4.10.6 Target Network Update Frequency

The target network is a periodically updated copy of the DQN that is used to stabilise training (Mnih, 2015). During SGD, adjustments are calculated based on the actual difference between the actual Q-value and an approximation of the correct Q-value, also known as the target value (Sutton, 2018). In standard Q-learning, the DQN is used to calculate the goal Q-value, whereas in Deep-Q, the target network is used instead (Mnih, 2015). This effectively delays the feedback loop, which increases stability.

If the target network is updated- that is, set to be a copy of the DQN- too frequently, the training system loses some of the benefit of delaying the feedback loop. If it's updated too infrequently, the risk of using outdated target values increases. Mnih (2015) uses a target network update frequency of 1 every 10-thousand-time steps, Paszke (2025) uses 1 every 200 steps, and Stable Baselines (2021) uses 1 every 500 steps. A value of 1 every 1 thousand time steps (about 30 seconds) was chosen for this project. 1 in 1 thousand is within the range of other successful projects, and a value on the low end of the range was chosen in hopes that the algorithm would be able to more quickly adapt to new levels during curriculum learning.

4.10.7 Short Answer to RQ

The chosen training algorithm hyperparameters were as follows:
Batch size = 32.
Learning rate = 1e-4.
Discount = 0.99.
ε decreases from 1.0 to 0.05 over 5 thousand episodes, then is maintained at 0.05.
Replay buffer size = 100 thousand.
Target network update frequency = 1 in 1 thousand.

4.11 Curriculum Learning

As discussed in Section 4.9.1: Markov Decision Processes, Deep-Q operates on a repeatable MDP for which each repetition is an episode. Shorter MDPs lead to more frequent episodes which typically results in faster training. The MDP of completing Awaria is separated into 13 smaller MDPs, one for each level. It was hypothesized that due to the transferability of knowledge between levels, training the NN on individual levels rather than the whole game could lead to faster overall learning. Further investigation showed that this could be effectively implemented via curriculum learning.

Research Question 11: What is curriculum learning, and would it be beneficial for this project?

Curriculum learning in the context of machine learning is the ordering of samples of the agent's experiences to accelerate or otherwise improve the training process. In practice, this usually involves seperately training the NN to learn individual concepts or policies that are related to the final goal policy but are significantly easier to learn. For example, a Rubik's cube solving NN could be initially trained with a 2x2x2 Rubic's cube, or a chess-playing NN could be trained with only pawns. In many applications, curriculum learning has resulted in decreased training times and increased NN competence (Narvekar, 2020).

Awaria is particularly well suited to curriculum learning, as it is made up of 13 levels of gradually increasing complexity and difficulty, each of which being easily repeatable by itself. To be a perfect analogue to Naverakar's (2020) work, a training curriculum for Awaria would consider completing level 13 of Awaria in isolation to be the final goal, when in fact the goal is to complete all 13 levels in sequence. However, given that a NN proficient in level 13 would be significantly better prepared to learn the project's actual goal policy than an untrained network, a curriculum consisting of learning each level in sequence before moving on to the full game was expected to be highly beneficial.

4.12 Training Duration

Research Question 12: How long is training the NN expected to take?

As stated in Section 4.9: Details of Algorithm, the training of Deep-Q is separated into episodes, which are in turn separated into time steps. The training duration would then be (# of episodes) * (duration of episode). These numbers can vary greatly depending on the application, however. For example, Paszke's (2025) Deep-Q project had a training time of 9m1.08s over 600 episodes, indicating an average episode length of roughly 0.9018 seconds, while Mnih's (2015) Deep-Q project had a training time of 55 thousand minutes, with an average episode length of seconds to minutes.

4.12.1 Episode Length

As episodes of an MDP are independent and repeatable, the length of an episode mostly depends on the design of the game and the frequency of its time steps. In simple games such as CartPole-v1, episodes are consistently 1-500 time steps long, but the maximum frequency is so high that the deceptively high episode length of 0.9018 seconds is dominated by the time needed to perform Deep-Q's computations (Paszke, 2025). Other games have a maximum frequency on the order of 60 Hz, such as the Atari 2600 games used by Mnih (2015). The length of episodes for these games is typically the shortest repeatable experience, such as a level or the start of the game to death, win, or time limit. Without curriculum learning, an episode in the MDP representing completing Awaria would be defined as the start of the first level to the completion of the 13th level or death, whichever happens first. Tests showed that deathless playthroughs of Awaria had a duration of 10-15 minutes.

4.12.2 Episode Count

The total number of episodes required to train a NN is much more difficult to determine in advance. The number of episodes required is, at minimum, the number required to gain experience that is sufficiently detailed and being representative of the whole game. Games with more detailed game states and more content will require more episodes to meet this minimum, especially in games with rng. Additionally, more difficult games require more episodes. A game-playing NN does not experience difficulty in the same way as a human in that they will never fail to take the intended action at the intended time. In other words, they have perfect mechanics. However, games with lower tolerance for incorrect actions will still require a more thoroughly trained NN to successfully navigate.

In short, more complex and more difficult games will take more episodes to successfully train, and as these would be prohibitively difficult to quantify, the episode count for this project was roughly estimated in reference to similar successful projects. Examined projects were completed in 600 episodes for CartPole-v1 (Paszke, 2025), 732 episodes for BananaCollector (Serrano, 2020), ~100 episodes for Unity 3DBall (Unity Technologies, 2025), and ~1000 episodes for LunarLander (Gadgil, 2020). Each of these games were deemed less complicated and potentially less difficult than Awaria and as such they were used as a rough minimum. The complexity of Awaria is much more comparable to the Atari 2600 games used by Mnih (2015), but episode counts for these games were not provided. Given the limited available information, a rough episode count of 3000 was estimated for this project, leading to an estimated duration of 31.25 days at 15 minutes per episode.

4.12.3 Short Answer to RQ

The best training time estimate is 31.25 days, but with very low certainty.

5 Implementation

Section 5 presents a chronological overview of important design features, experiments, and project organisation conducted between 6th June and 1st September 2025. Following literature review detailed in Section 4: Research, a project timeline was drafted consisting of 5 goals as described in Section 3: Project Goals.

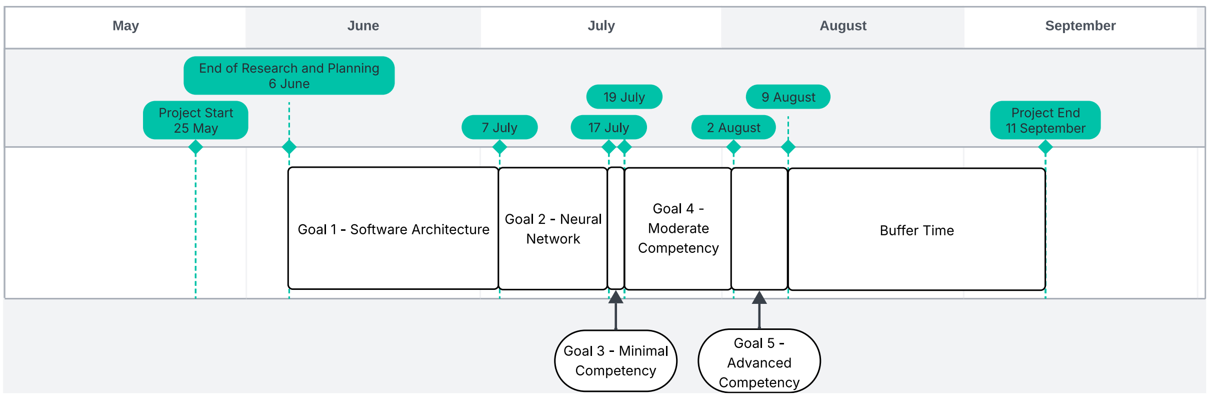


Fig. 7: A Gantt chart showing the initially planned project timeline.

Section 5.1: Compensation for Training Time Uncertainty details a plan for adapting the project timeline depending on the duration of experiments performed from Goal 3 onwards.

Section 5.2: Goal 1 describes the implementation and completion of Goal 1.

Section 5.3: Goal 2 describes the implementation and completion of Goal 2.

Sections 5.4 through 5.10 describe major subtasks performed pursing completion of Goal 3.

Section 5.4: The Gear Machine Test details a subtask of Goal 3 concerned with only completing a small portion of the first level.

Section 5.5: Input Sequence Test details a task not directly related to Goal 3 performed for diagnostics.

Section 5.6: Revisiting Update Synchronisation describes an alteration to the design that ensured accurate transitions.

Section 5.7: Automatic Recovery Feature details a feature that improves the system’s reliability.

Section 5.8: Training State Recovery Feature describes another feature that improves the system’s reliability.

Section 5.9: Adjusting Project Timeline records a reorganisation of the project roadmap in response to unexpected delays.

Section 5.10: Attempts to Complete Goal 3 describes attempts to train the DQN to complete the first level in its entirety.

5.1 Compensation for Training Time Uncertainty

Due to the uncertainty in training time discussed in Section 4.12: Training Duration, a plan was drafted to prepare for both short and long training times. Early testing showed that the simulation speed, or the rate at which events occurred in Awaria, was the bottleneck for episode duration, and that accelerating the game would have been difficult, though potentially possible. It was hypothesized that the training portion of Goal 3 (Section 3.3) would take at most 5 days, and it was decided to observe the actual training time of Goal 3 before attempting to accelerate the gameplay. Formally, acceleration would be attempted if the training duration was more than 7 hours.

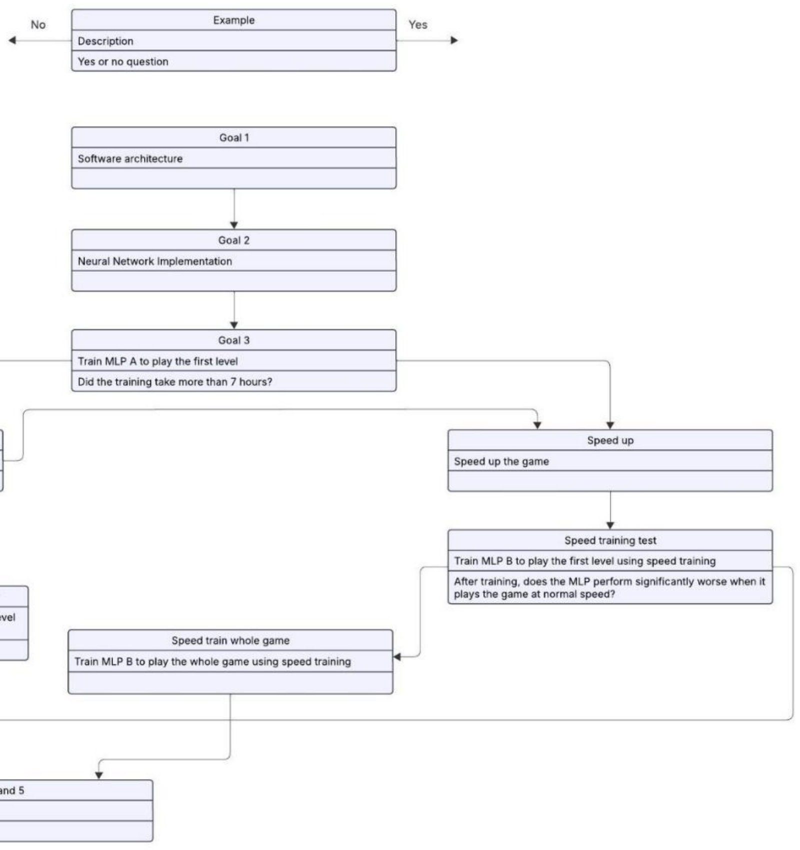

No
Example
Description
Yes or no question
Yes
Goal 1
Software architecture
Goal 2
Neural Network Implementation
Goal 3
Train MLP A to play the first level
Did the training take more than 7 hours?
Standard training
Train MLP A to play the whole game
Is training still ongoing after a week?
Speed up
Speed up the game
Speed training test
Train MLP B to play the first level using speed training
After training, does the MLP perform significantly worse when it plays the game at normal speed?
Worst case scenario
Train MLP A to play the whole game. Train MLP C to play the hardest level.
Did you run out of time?
Minimal Competency
Only learned level 1 and level 12, seperately
Speed train whole game
Train MLP B to play the whole game using speed training
Goals 4 and 5
Network is fully trained

Fig. 8: A flow chart describing the plan to compensate for potentially slow training

5.2 Goal 1

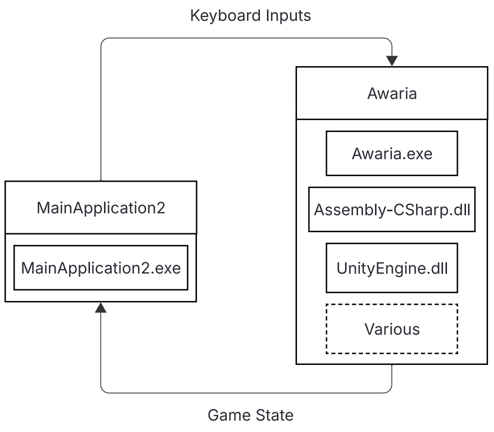


Fig. 9: A basic overview of the software architecture

Goal 1 (Section 3.1) involved creating a new application (MainApplication2) and modifying Assembly-CSharp.dll, a component of Awaria, such that Awaria can send game state information to MainApplication2 and MainApplication2 can send control signals to Awaria.

5.2.1 MainApplication2 and Communication

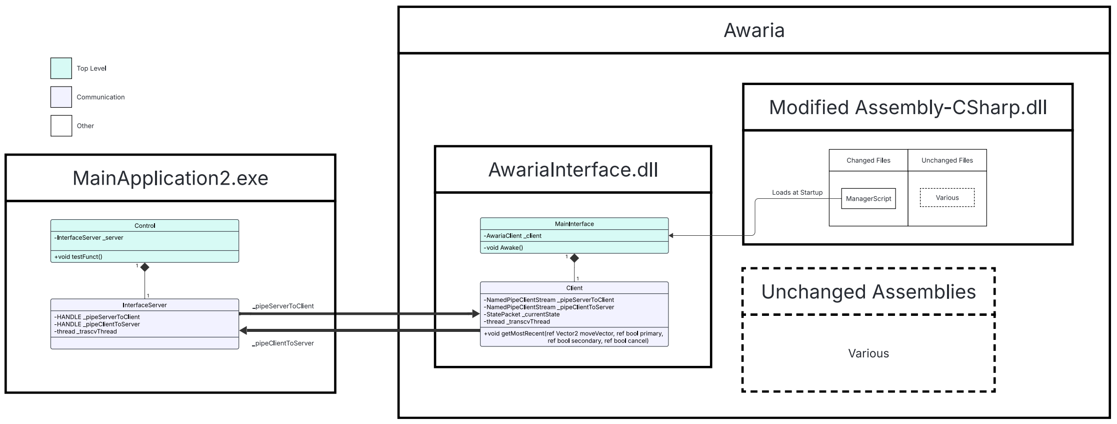


Fig. 10: A diagram of the minimal software architecture required to support communication. Bolded arrows indicate data flow and normal arrows indicate dependencies.

To minimise the risk of unintentionally altering gameplay, a .dll file was created to be injected into Awaria at runtime, with the intent of modifying Assembly-CSharp.dll as little as possible. MainApplication2 was designed using Visual Studio C++ to host 2 named pipes, a Windows mechanism for inter-process communication. AwariaInterface.dll would then connect to those pipes on startup.

5.2.1 Control Signals

After setting up communication, the next step was to determine how Awaria received control signals and replace them with artificial signals from another application. Through code and documentation review, it was determined that control signals originated in device drivers and first entered the game logic in Assembly-CSharp.InputScript, which stored the most recent signals each frame. InputScript was modified to instead store signals from the injected AwariaInterface, which in turn received them through a named pipe in 11-byte packets.

5.2.2 Game State Signals

Awaria's internal game state is not centralised but rather split among many objects in the scene. To construct a coherent representation of the state, several scripts in Assembly-CSharp were modified to send important data to AwariaInterface each frame, which would maintain a snapshot of the most recent information and send it over a named pipe each frame in the form of a 229-byte packet.

To determine what data should be included in the game state packet, the question was posed: "What is the minimum amount of information the neural network needs to reliably win any level, even if it has no access to previous game states?" It goes without saying that without enough information, the network won't be able to make educated decisions, but too much information may also impair learning by increasing the computation load and making it more difficult to discern what inputs are important in what situations. This question was answered with the following list:

- Player Position- obstacle avoidance and limited range interaction are core mechanics of the game.
- Generator information
  - Broken generator flag- Which and when generators break is random and cannot be memorised by the NN.
  - Required parts- These are also random and cannot be memorized.
- Player held objects- The NN has no short-term memory and will forget if it has picked up a required item unless reminded each frame.
- Component machine status- Like held objects, this is deterministic but will be forgotten between frames.
- Hazard locations- the NN needs to dodge hazards and has no memory of where they originated. There is no limit to the number of hazards on screen, so only the three closest hazard positions were provided to the NN.
- Current Level- In the absence of any other information, the network would need to know the current level to know what the level layout is.

After some deliberation, additional non-essential information was added in hopes that it would do more good than harm to the training:

- Generator location, component machine location- Every level has these points of interest in its layout. Adding the locations as explicit inputs was predicted to free the NN from having to memorise their locations during training and encourage generalisation between levels. It was held that if a NN learns that “input 15” is the location of “generator 2”, that knowledge would transfer to any other level.
    - Note that “Current Level” from the previous list was kept due to its high information density.
- Component machine type- Each machine type is present in more than 1 level, and providing the type, location, and status of a component machine was predicted to aid in generalisation.
- Ghost location- Ghosts are different from standard hazards in that they typically do not do contact damage, and instead regularly perform dash melee attacks or spawn projectiles. Providing the current location of all ghosts (1-3 depending on the level) was expected to greatly enhance dodging ability.
- Ghost ID- Some ghosts are present in more than 1 level, and tagging their positions with an ID was hoped to improve generalisation.
- Dash ready flag- Outside of a tutorial, dashing is not required to win the game. Nevertheless, it is quite useful and adding a dash flag required for the NN to know with certainty whether a dash action will fail.

5.2.3 Completion

A total of 20 files inside Assembly-CSharp, which can be seen in Fig. 11, were modified to send game state data to AwariaInterface. These updates were typically sent during the Unity main loop's Update() stage.

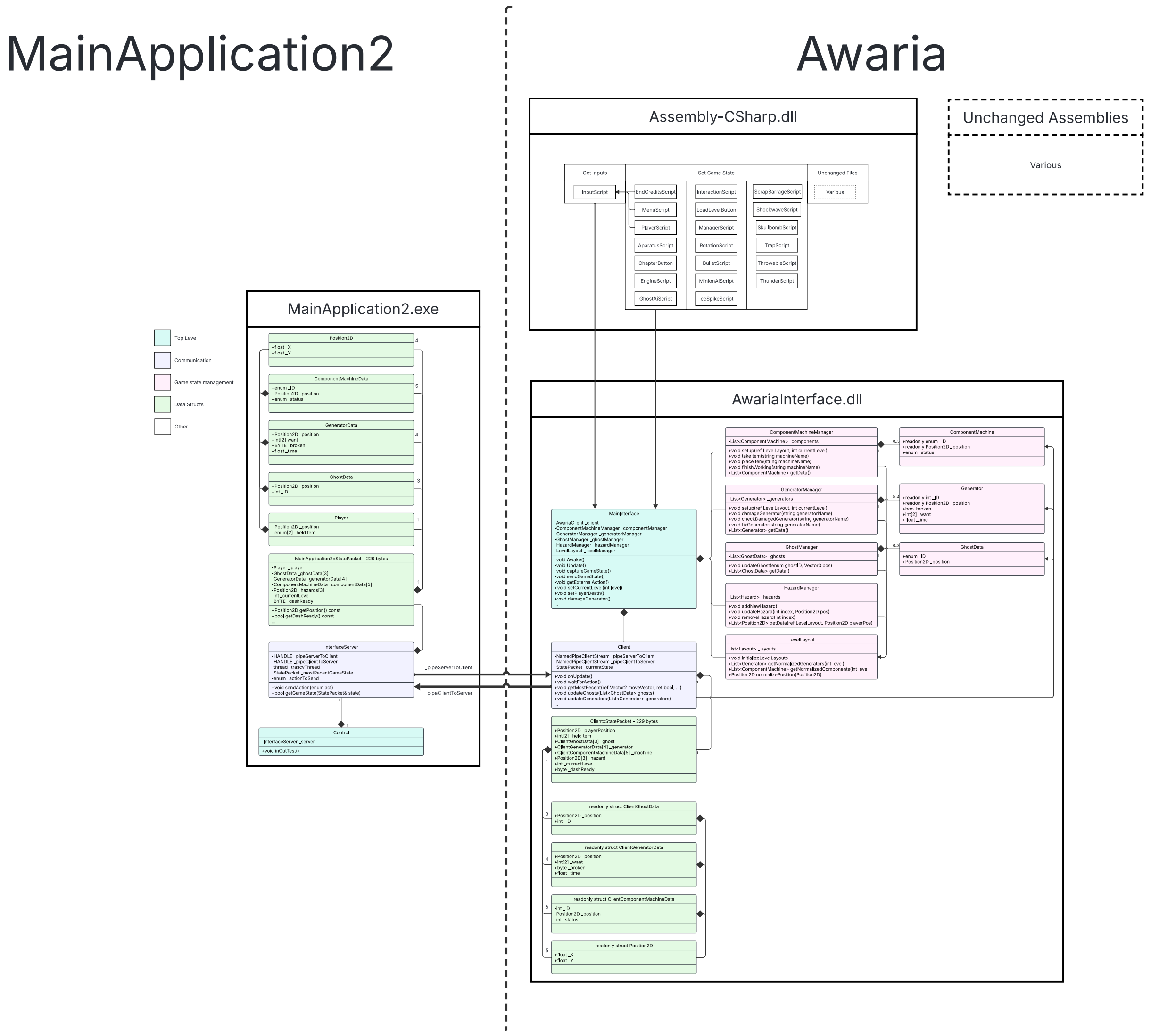


Fig. 11: An overview of the full system on completion of Goal 1

The final step in Goal 1 was to create an automated test in MainApplication2 that sent inputs to Awaria and received game state snapshots back. The inputs and game states were to be verified against expected values to ensure the communication was working correctly. The test primarily consisted of entering a level, taking a specific sequence of actions to provoke a specific change in game state, and resetting. A full 11-minute recording of a successful test can be found here: https://www.youtube.com/watch?v=KQFc1Sr7FXA. The test was first passed in full on 3rd July, and Goal 1 was considered completed.

5.3 Goal 2

Goal 2 (Section 3.2 Neural Network) was to implement the NN and Deep-Q algorithm. The culmination of Goal 2 was a test in which the DQN was trained to solve a very simple problem.

5.3.1 Multilayer Perceptron

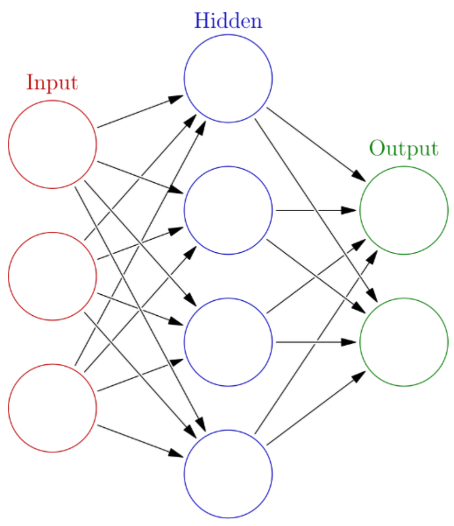
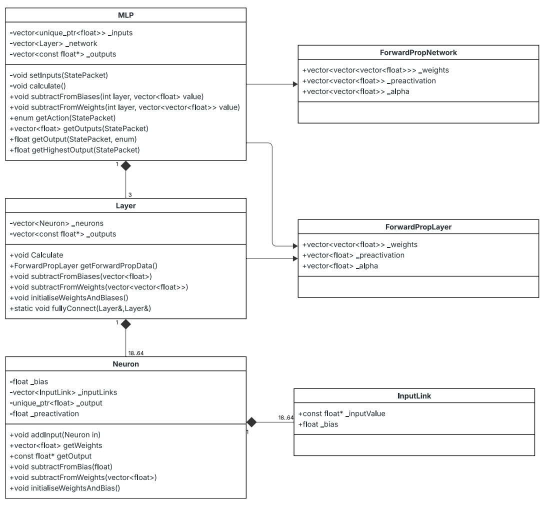


Fig. 12: An example of an MLP with 1 hidden layer

As discussed in Section 4.3: Network Architecture, the architecture selected was a fully connected Multilayer Perceptron (MLP), one of the simplest neural networks. For a full explanation of this architecture see Section 4.4: Details of Network Architecture.

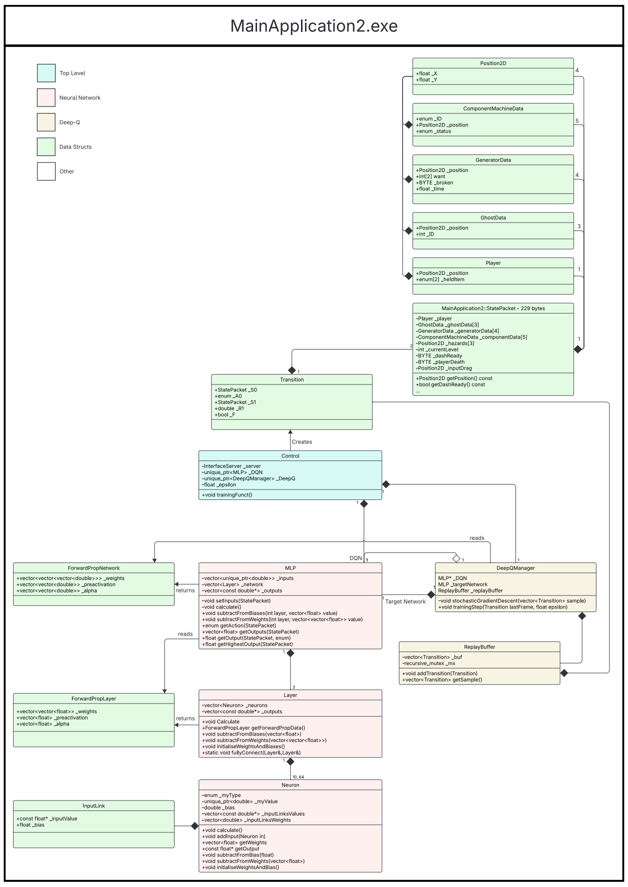


```
MLP
-vector<unique_ptr<float>> _inputs
-vector<Layer> _network
-vector<const float*> _outputs
-void setInputs(StatePacket)
-void calculate()
+void subtractFromBiases(int layer, vector<float> value)
+void subtractFromWeights(int layer, vector<vector<float>> value)
+enum getAction(StatePacket)
+vector<float> getOutputs(StatePacket)
+float getOutput(StatePacket, enum)
+float getHighestOutput(StatePacket)

ForwardPropNetwork
+vector<vector<vector<float>>> _weights
+vector<vector<float>> _preactivation
+vector<vector<float>> _alpha

Layer
-vector<Neuron> _neurons
-vector<const float*> _outputs
+void Calculate
+ForwardPropLayer getForwardPropData()
+void subtractFromBiases(vector<float>)
+void subtractFromWeights(vector<vector<float>>)
+void initialiseWeightsAndBiases()
+static void fullyConnect(Layer&,Layer&)

ForwardPropLayer
+vector<vector<float>> _weights
+vector<float> _preactivation
+vector<float> _alpha

Neuron
-float _bias
-vector<InputLink> _inputLinks
-unique_ptr<float> _output
-float _preactivation
+void addInput(Neuron in)
+vector<float> getWeights
+const float* getOutput
+void subtractFromBias(float)
+void subtractFromWeights(vector<float>)
+void initialiseWeightsAndBias()

InputLink
+const float* _inputValue
+float _bias
```

Fig. 13: An overview of the MLP implementation

As seen in Fig 13, the neural network was implemented as a container of layers, themselves being containers of neurons. Connections between neurons are represented by pointers stored within the neurons themselves which point to either input scalars owned by MLP or output scalars owned by other neurons. Weights and biases for each neuron are also stored in the neuron itself. Objects external to MLP can set the input environment state, receive copies of Q-values from the output layer, or receive detailed forward propagation data as a struct.

5.3.2 Deep-Q



MainApplication2.exe
Top Level
Neural Network
Deep-Q
Data Structs
Other
Position2D
+float _X
+float _Y
ComponentMachineData
+enum _ID
+Position2D _position
+enum _status
GeneratorData
+Position2D _position
+int[2] want
+BYTE _broken
+float _time
GhostData
+Position2D _position
+int _ID
Player
+Position2D _position
+enum[2] _heldItem
MainApplication2::StatePacket - 229 bytes
-Player _player
-GhostData _ghostData[3]
-GeneratorData _generatorData[4]
-ComponentMachineData _componentData[5]
-Position2D _hazards[3]
-int _currentLevel
-BYTE _dashReady
-BYTE _playerDeath
-Position2D _inputDrag
+Position2D getPosition() const
+bool getDashReady() const
...
Transition
+StatePacket _S0
+enum _A0
+StatePacket _S1
+double _R1
+bool _F
Creates
Control
-InterfaceServer _server
-unique_ptr<MLP> _DQN
-unique_ptr<DeepQManager> _DeepQ
-float _epsilon
+void trainingFunct()
reads
DQN
ForwardPropNetwork
+vector<vector<vector<double>>> _weights
+vector<vector<double>> _preactivation
+vector<vector<double>> _alpha
returns
MLP
-vector<unique_ptr<double>> _inputs
-vector<Layer> _network
-vector<const double*> _outputs
-void setInputs(StatePacket)
-void calculate()
+void subtractFromBiases(int layer, vector<float> value)
+void subtractFromWeights(int layer, vector<vector<float>> value)
+enum getAction(StatePacket)
+vector<float> getOutputs(StatePacket)
+float getOutput(StatePacket, enum)
+float getHighestOutput(StatePacket)
Target Network
DeepQManager
MLP* _DQN
MLP _targetNetwork
ReplayBuffer _replayBuffer
-void stochasticGradientDescent(vector<Transition> sample)
+void trainingStep(Transition lastFrame, float epsilon)
ReplayBuffer
-vector<Transition> _buf
-recursive_mutex _mx
+void addTransition(Transition)
+vector<Transition> getSample()
reads
ForwardPropLayer
+vector<vector<float>> _weights
+vector<float> _preactivation
+vector<float> _alpha
returns
Layer
-vector<Neuron> _neurons
-vector<const double*> _outputs
+void Calculate
+ForwardPropLayer getForwardPropData()
+void subtractFromBiases(vector<float>)
+void subtractFromWeights(vector<vector<float>>)
+void initialiseWeightsAndBiases()
+static void fullyConnect(Layer&,Layer&)
10..64
Neuron
-enum _myType
-unique_ptr<double> _myValue
-double _bias
-vector<const double*> _inputLinksValues
-vector<double> _inputLinksWeights
+void calculate()
+void addInput(Neuron in)
+vector<float> getWeights
+const float* getOutput
+void subtractFromBias(float)
+void subtractFromWeights(vector<float>)
+void initialiseWeightsAndBias()
InputLink
+const float* _inputValue
+float _bias

Fig. 14: A diagram of the system upon completion of Goal 2. Truncated to show only classes relevant to Goal 2.

The Deep-Q algorithm was implemented as described in Section 4.9: Details of Training Algorithm and Section 4.10: Algorithm Hyperparameters. The only change made was that the error clipping described in parts 2b and 3b of backpropagation (See Section 4.9.4: Mini-batch Stochastic Gradient Descent) was erroneously excluded. This mistake, the exclusion of error clipping would lead to sparse reward (high value, low frequency) transitions creating larger adjustments in weights and biases while increasing risk of oscillation and gradient explosion.

The Deep-Q implementation was encapsulated in a new class DeepQManager. DeepQManager contained the replay buffer and target network (an instance of MLP) as well as a reference to the DQN (another instance of MLP) owned by Control.

5.3.3 Obstacles Encountered During Goal 2

During testing, 2 major issues were encountered with the Deep-Q implementation: gradient explosion and high CPU usage.

5.3.3.1 Gradient Explosion

Gradient explosion is a result of SGD adjusting one or more weights or biases by too much during a training step. Even if the adjustment is in the correct direction, and it doesn't overshoot the optimal value, it can still lead to a large error value in a subsequent training step, which produces an even larger adjustment, creating a positive feedback loop. In the case of gradient explosion, weights and biases increase rapidly until overflow or the end of training.

The error clipping discussed in section 4.9.4: Mini-batch Stochastic Gradient Descent that was erroneously omitted from the implementation theoretically makes gradient explosion impossible. Rather than correcting the mistake, the problem was solved by using He initialisation to initialise the weights and biases of the MLP. He initialisation sets all biases to 0 or another small constant and all weights to random values sampled from normal distributions of mean 0 and standard deviation = $\sqrt{\frac{2}{fan_{in}}}$, where fan_{in} is the number of inputs the neuron has (He, 2015). Using He initialisation does not necessarily provide immunity to gradient explosion, but it can help, and in this case it was sufficient.

5.3.3.2 CPU Usage

During testing it was noted that each training step required 5e-2 seconds (20Hz) of computation in Debug mode. As each time step in Awaria is ~6.9e-3 to 1.67e-2 (60Hz-144Hz) seconds, and MainApplication2 would ideally run a training step each time step, this was considered too slow. In preparation for Goal 3, efforts were made to accelerate the algorithm by identifying hot loops with the Visual Studio profiler and optimising or parallelising code, but the resulting benefits were insufficient. Ultimately it was decided to compile MainApplication2 in Release mode, which was an effective solution (lowering the training step duration to 0.005 seconds (200Hz) without parallelisation) but increased the difficulty of future development.

5.3.4 Completion

The final step of Goal 2 was to create a new test inside Control that verified the functionality of MLP and DeepQManager. Rather than creating transitions using communication with Awaria, transitions were artificially created by setting the start state S0 and the end state S1 to contain random values within the values' respective valid ranges.

In this test, the DQN was trained with 1000 episodes, each 1-time step in length. For each episode, a new transition was created and given to the DQN, which then chose an action with ε = 0. If the chosen action was "move west", the reward R1 was set to 100.

Otherwise, R1 was set to 0. Finally, the transition was given to DeepQManager, which performed a training step on the DQN.

After training, the competence of the trained DQN was evaluated via a by providing the DQN with another randomly generated state and testing its preferred action. An action of “move west” was considered a success, while any other action was a failure. This evaluation was repeated 1000 times, achieving a success rate of 100.0%. This success rate is more than the 95% required by Goal 2 (Section 3.2), and Goal 2 was considered completed on 17th July.

5.4 The Gear Machine Test

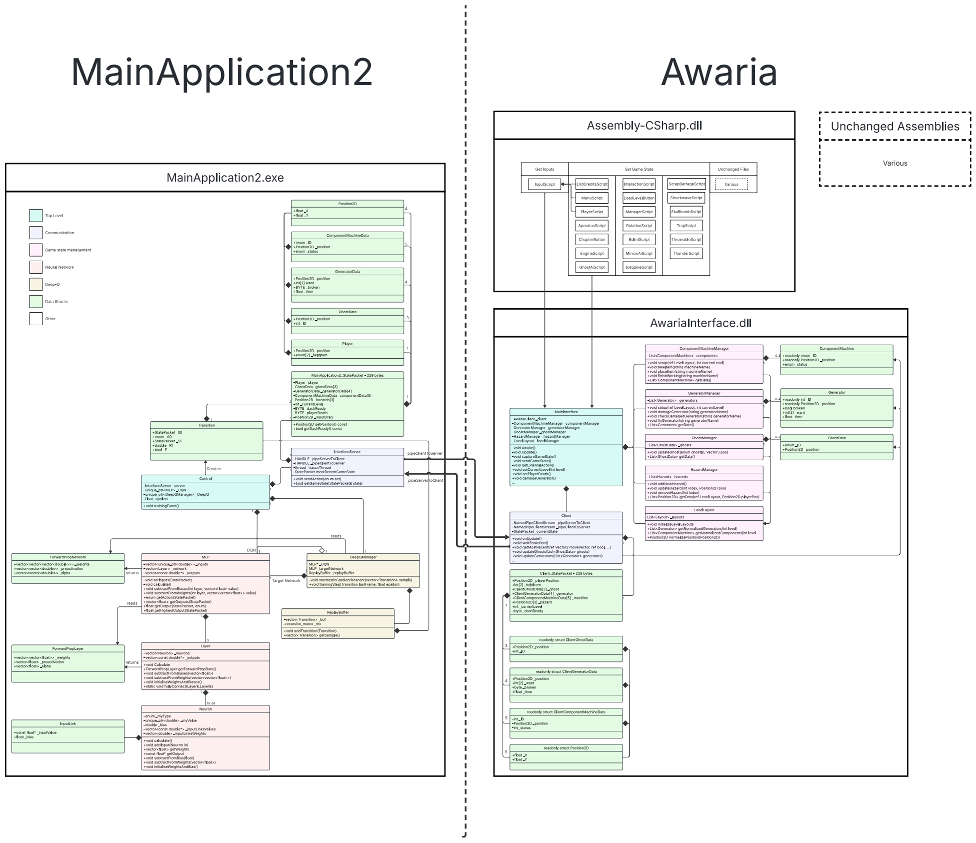


Fig. 15: An overview of the system on completion of Goal 2, not truncated as in Fig. 14. This system was expected to be able to complete Goal 3 with minimal modification.

Goal 3 (Section 3.3) involved training the DQN to complete the first level of Awaria. Fully training the DQN to complete this task was expected to take hours to days, and the actual training time was to be used as described in Section 5.1: Compensation for Training Time Uncertainty to direct future development. At the time of Goal 2’s completion on 17th July, access to the hardware necessary for multi-hour training sessions was still pending. An alternative, much simpler task titled the Gear Machine Test was given to the system to serve as a proof of concept and identify potential obstacles. It was expected that fully training the NN to solve this task would take at most 15 minutes.

Rather than attempt the entire level, the goal of the Gear Machine Test was to train the DQN to quickly move across the room upon level start and interact with a specific machine. To accomplish this, a sparse reward (infrequent, but high value) of 300 was provided for interacting with the machine, and a dense reward (every frame, but low value) with a maximum cumulative value of 100 was provided for moving closer to the machine. After training was complete, subjective observation revealed that the agent was showing few if any signs of progress. Repetition of training with 10x learning rate, 10x learning rate and 10x rewards, and 2x training duration did not produce significantly better results.

5.4.1 Ensuring Accurate Transitions

After some investigation into the failure of the system to pass the Gear Machine Test, 2 potential causes were identified: Input drag and desynchronised game updates. Both of which stem from the way that Deep-Q teaches delayed gratification. Simplifying, Deep-Q tells the network that in situation S0, the desirability of action A0 should equal $R1 + \gamma TNO$, where R1 is instant gratification and γTNO is delayed gratification. γ is constant and *TNO* is essentially the desirability of the situation in the following time step. Therefore, Deep-Q can only teach delayed gratification if there is a quantifiable difference between S0 and S1 that results from action A0, preferably with a deterministic relationship. In other words, an action can only be taught as beneficial without an immediate reward if it can be causally linked to an immediate change in environment state.

5.4.1.1 Input Drag

Input drag in Awaria is a feature in PlayerScript that prevents the agent from changing direction too quickly. When a new direction is specified via player input, the agent will start changing direction immediately but may take a few frames to reach the specified direction. Due to the player's velocity and move direction not being provided to Deep-Q, it was suspected that input drag was preventing the DQN from learning how to control its movement. However, temporarily disabling input drag entirely did not provide a better result.

5.4.1.2 Update Synchronisation

In Unity, the Update portion of the main loop is called 0 or 1 times per frame, typically once, during which the Update() method of every active script will run. This includes InputScript, which updates its stored control signals during Update() as well as PlayerScript, MenuScript, and EndCreditsScript, which copy control signals from InputScript during their respective Update(). The order in which the Update() method of each script runs can be set in the Unity Editor, but if left as default, the order may change at any time. This means that InputScript.Update() may run before PlayerScript.Update() and after MenuScript.Update(), which would result in MenuScript using player inputs that are 1 frame newer than PlayerScript's. Tests indicated that the order of execution did in fact change in some situations, which could lead to causal changes in state being delayed by 1 frame.

To address this, the design was modified such that all classes that read the inputs stored in InputScript would instead receive them directly from the injected DLL (AwariaInterface). AwariaInterface was then set to always update its stored control signals at the end of Unity's Update phase. Finally, communication between Awaria and MainApplication2 was synchronised in a ping-pong protocol such that at the end of each Update(), Awaria would send the game state and block until receiving an action, while MainApplication2 would only send an action in response to receiving a state, guaranteeing accurate transitions. These changes in the design did not lead to improved performance, but they were kept nonetheless to prevent any future issues.

5.4.2 Metrics and Increased Training Duration

Due to the failure of efforts described in Section 5.4.1: Ensuring Accurate Transitions to resolve the problems encountered when first attempting the Gear Machine Test, a transition logging feature was added to MainApplication2 to aid in debugging as well as to provide a quantitative way to evaluate competing designs in the future. This new feature periodically calculated the average loss (a number used in SGD) and reward of all transitions created since the last log entry, which was written to disk in CSV format such that loss and reward could be easily plotted over time in Excel.

The loss of a transition represents the difference between the outputs of the DQN and an approximation of the optimal NN. A smaller loss indicates that the DQN is closer to an optimal NN, so loss should ideally decrease over time. The reward of a transition is set by Deep-Q, and should ideally increase over time, indicating that the agent is showing more valued behaviour as training progresses. Both of these values are highly dependent on the current value of ε, which can provide result in deceptive metrics. Methods to reduce this dependency include using a constant ε for some or all of training as in Mnih's (2015) Deep-Q implementation or to calculate a reward and loss using the preferred action rather than the action taken.

Using this new logging, 5 training sessions of 1 hour duration were performed with varying hyperparameter values. The metrics showed that reward and loss were highly variant throughout training with no clear trend, akin to static. Additionally, none of the agents appeared to be playing competently once fully trained.

5.5 Input Sequence Test

Due to the failure of the system to pass the test described in Section 5.4: The Gear Machine Test, a new test was designed that did not require communication with Awaria. Although of similar complexity, this test would be much shorter, and the learned policy would be very different. It was hoped that the rapid repeatability and alternative policy would provide more useful diagnostic data than further runs of the Gear Machine Test.

The goal of this new test was to teach the DQN to take a specific sequence of 23 actions. A set of 24 artificial game states was created, each randomised as in the final test of Goal 2 (Section 5.3). This ordered set of states formed a finite state machine, for which each state had 1 correct action and 17 incorrect actions. A correct action resulted in a transition to the next state in the set and a reward of 50. If the next state is the final state, the episode ends. An incorrect action resulted in a transition to the first state, an episode end, and a reward of -100. The test was considered a success if the DQN entered the full sequence correctly 4 times in fewer than 150 thousand training steps, or about 3 minutes. Successful tests could be compared by the number of training steps taken, and unsuccessful tests could be compared by the maximum index reached in the sequence.

Testing showed that the current hyperparameters were very suboptimal for the policy, which was unsurprising, given that the environment was both much simpler and much less forgiving than Awaria. In particular, optimal results were found with 10x more frequent target network updates, a batch size of 80, and $\varepsilon = 1 - \sqrt[max\_index+1]{\frac{1}{2}}$. With these hyperparameters, the success rate was 100%. Passing this test showed that the system was capable of quickly learning difficult problems with high accuracy.

The 2 main differences between the Gear Machine Test and Input Sequence Test were that the former relies on communication with Awaria while the latter does not, and the former uses semi-random, meaningful game states while the latter uses completely random, meaningless game states. The fact that the system succeeded at the latter task while failing at the former implied that the issue the design was related to 1 of those differences.

5.6 Revisiting Update Synchronisation

Further investigation into the failure of the system to pass the test described in Section 5.4: The Gear Machine Test revealed that while the changes made in Section 5.4.1: Ensuring Accurate Transitions did resolve the symptoms observed at that time, game states were still not perfectly synchronised with actions.

5.6.1 Discovering the Problem

This problem was discovered through a small input responsiveness test run by MainApplication2, which sent simple inputs to Awaria and measured the latency before an expected result was received in a game state snapshot. To ensure viable transitions, the latency needed to be exactly 1 frame, preferably with 100% consistency. Testing instead showed a consistent 2-frame delay.

The newly discovered issue was found to stem from the Update and FixedUpdate phases of Unity's main loop. During a single frame, the Unity engine will call the FixedUpdate() method and the Update() method in every script 0 or 1 times each (typically once for each). The FixedUpdate phase is intended used for physics, while the Update phase is intended for game logic, though the separation is not enforced in practice. Both of these phases can cause significant changes in the game state. For example, in vanilla Awaria, the items the player is holding is set during Update, while the location of the player is set in FixedUpdate. The FixedUpdate phase always comes before the Update phase (provided a frame has an instance of both) meaning that if the behaviour of FixedUpdate depends on data created by Update, the corresponding change in game state will be delayed by 1 frame. In Awaria, this dependency is typically the velocity of objects such as ghosts, projectiles, and the player. As discussed in Section 5.4.1: Ensuring Accurate Transitions, a consistent 1 frame delay such as this was expected to lead to a total failure of training.

5.6.2 Ensuring Synchronisation

This problem was solved by moving the communication phase- send state, receive action, save action for reference- from the end of the Unity Update phase to the end of the Unity FixedUpdate phase. This effectively meant that the game state snapshot was of FixedUpdate on frame N and Update on frame N-1, ensuring synchronisation without significantly modifying the gameplay. This did result in the game being 1 frame more responsive on all player inputs, but the change was theoretically within the variance of vanilla gameplay, in which InputScript.Update() can run at any time during the Update phase.

After a training session of 15 minutes using this improvement, the DQN was able to interact with the gear machine within 30 seconds in 20 out of 20 trials, finally passing the Gear Machine Test.

5.7 Automatic Recovery Feature

At the time the system passed Gear Machine Test (Section 5.4) due to the modifications made in Section 5.6.2: Ensuring Synchronisation, access to hardware required for multi-hour tests had been secured, and it was deemed appropriate to begin the primary objective of Goal 3: to train the DQN to complete the first level. During the first multi-hour training sessions, training consistently halted roughly 55 minutes into training.

5.7.1 Diagnosing the Problem

The initial symptoms showed that while neither Awaria nor MainApplication2 had crashed, one or both had paused execution, and the computer had locked. Subsequent testing showed that execution continued indefinitely when vanilla Awaria ran by itself, or when the modified Awaria ran along with MainApplication2, but with the blocking communication between the two severed. Maintaining the connection but forcing a deadlock between the applications or putting MainApplication2 to sleep resulted in an (expected) immediate halt in execution followed by the screen locking after 15 minutes. All other tests resulted in the same halt of execution after 55 minutes.

These tests conclusively proved little other than that the problem is not present in Vanilla Awaria. All tests that did not change the communication between MainApplication2 and Awaria resulted in the 55-minute failure time, and all tests that blocked Awaria's main loop resulted in the screen locking after 15 minutes. Given the available information, the cause of the problem could not be determined, but it was hypothesised that:

1. The computer locking was a symptom of the problem, not the cause.
2. The cause of the computer locking was Awaria halting execution.
3. The most likely cause of Awaria halting execution was a deadlock in its communication with MainApplication2.

5.7.2 Attempting to Resolve the Problem

Since the communication between MainApplication2 and Awaria follows a ping pong protocol (implemented in Section 5.4.1: Ensuring Accurate Transitions), it can deadlock if both applications are trying to read or write at the same time. It was intended to be impossible for such a deadlock to occur, but no mechanism was in place to prevent a deadlock from happening or to recover from one. To resolve the problem encountered in Section 5.7.1: Diagnosing the Problem, a watchdog thread was added to MainApplication2 that would forcibly unblock the communication thread whenever it had been blocking for more than 1 second. The communication thread would then proceed to a blocking read or write, the opposite of whatever it had been deadlocked on. Additionally, the watchdog would delete the last second of changes, rounding up, from the Replay Buffer, transition buffer (a buffer used to send transitions from the communication thread to the main thread), DQN, and target network. This ensured that any invalid transitions were discarded and did not harm training.

Further testing with this feature indicated that the problem was had not been solved but rather had been altered- execution now halted after roughly 2 hours and 50 minutes rather than 55 minutes. The change to the design was nonetheless kept due to improved diagnostics and robustness for the system.

5.8 Training State Recovery Feature

As the automatic recovery feature implemented in Section 5.7 did not solve the problem in which training halted during multi-hour sessions, and further investigation failed to reveal a cause, a workaround was created that was expected to work regardless of the root cause: a global singleton named MyExcept was created with references to the ReplayBuffer, target network, DQN, and CurriculumManager (an object that tracks training progress). In the event of an error state or end of training, MyExcept would save the contents of each object to disk, such that the training state could be loaded when next launching the applications.

Testing showed that the feature was working correctly, and while it did not address the root cause of the issue, multi-hour training sessions could now be run successfully, provided they were manually restarted every 3 hours. This was considered sufficient to continue with the main objective of Goal 3, training the DQN to complete the first level.

5.9 Adjusting Project Timeline

The feature described in Section 5.8: Training State Recovery Feature was completed on 17th August, well past the Goal 3 deadline of 19th July. After considering the situation, completion of Goals 4 and 5 was deemed unlikely within the remaining time allotted to the project. It was decided to refocus on Goal 3 as the primary remaining goal of the project, and to postpone Goals 4 and 5 indefinitely. The deadline for Goal 3 was then moved to 1st September. Goal 3 had always been intended as a prerequisite for Goals 4 and 5, so the only immediate change made was to cut down the input and output layers of the NN.

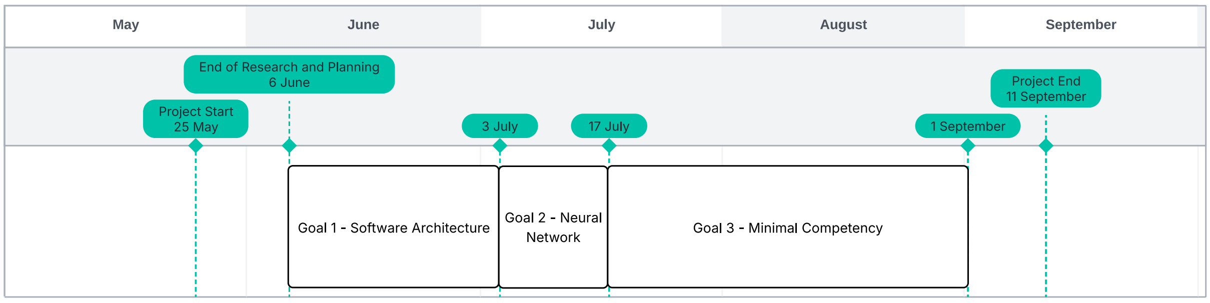


Fig. 16: An updated Gantt chart showing the changed timeline.

5.9.1 Reducing the Neural Network Size

The size of the input layer of a DQN corresponds to the size of the environment state, and a more detailed environment requires more training to interpret accurately. The size of the output layer corresponds to the number of possible actions the agent can take, and a larger action space requires more exploration to determine optimal actions. The input and output layers of the NN created in Section 5.3: Goal 2 were designed with these limitations in mind but were intended to be used for the full game. Many of the inputs and outputs of the NN are unused in the first level, such as hazards or the dash feature.

To improve training speed, the input layer was reduced from to 80 to 18 floats, representing the following environment state: player position, inventory, ghost position, 1 generator, and 2 component machines. The output layer was reduced from 18 neurons to 10, representing interact, stand still, and 8 movement directions.

5.10 Attempts to Complete Goal 3

Beginning with the system that passed the test described in Section 5.4: The Gear Machine Test and then modified as described in Section 5.9: Adjusting Project Timeline, the reward scheme was modified such that the DQN would be trained to complete the full level rather than only interacting with the gear machine. Sparse (high value, low frequency) shaping rewards (not directly corresponding to completing a core objective) were provided for actions such as picking up items, checking generators, and other useful interactions. Sparse goal rewards (corresponding to a core objective) were provided for fixing generators and completing the level. No dense rewards (low value, high frequency) were provided.

5.10.1 Initial Attempts

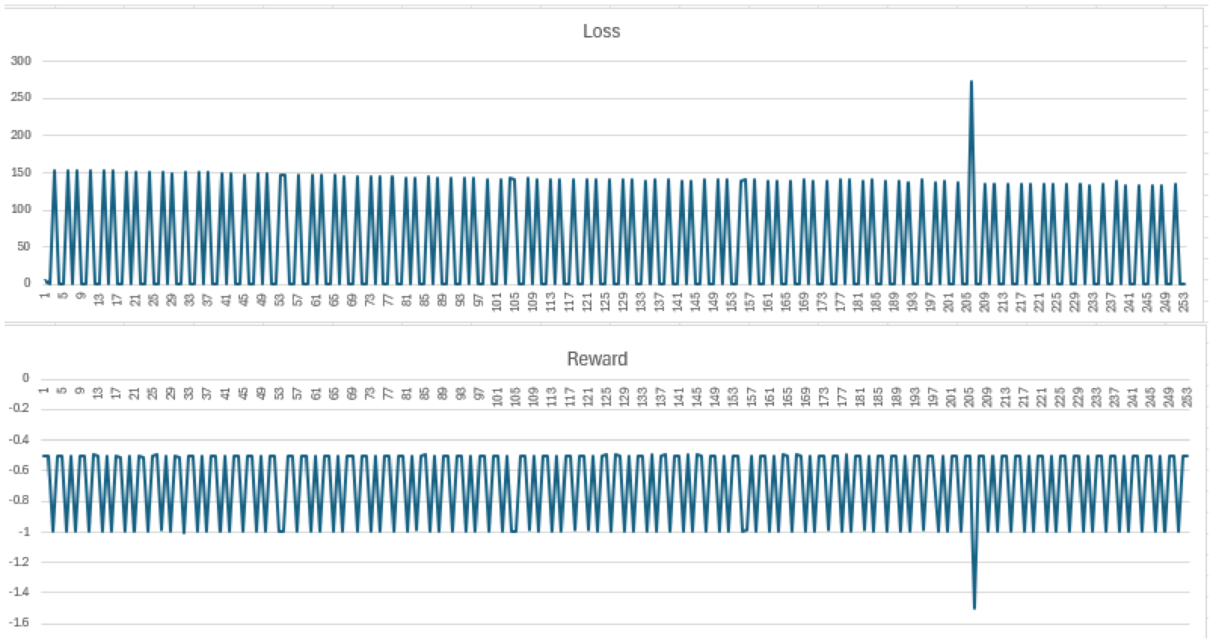


Fig. 17: The loss and reward (y axis) over time (x axis) of the first complete attempt at Goal 3 (100x Learning rate, 2.4 hours)

5 training sessions with a length of 2 hours were attempted: 1 control group, and 4 with varying hyperparameters. 3 tests, including the control group, failed due to an unknown error. 1 test failed due to gradient explosion (defined in Section 5.3.3.1: Gradient Explosion). The remaining test differed from the control group by having 100x learning rate η. Metrics showed that loss decreased slightly over time while reward stayed constant, and both appear to be oscillatory. It was hypothesized that this oscillation was due to the 100x learning rate. However, repeated tests of the control group and other variations typically ended with gradient explosion, frustrating attempts to confirm the hypothesis.

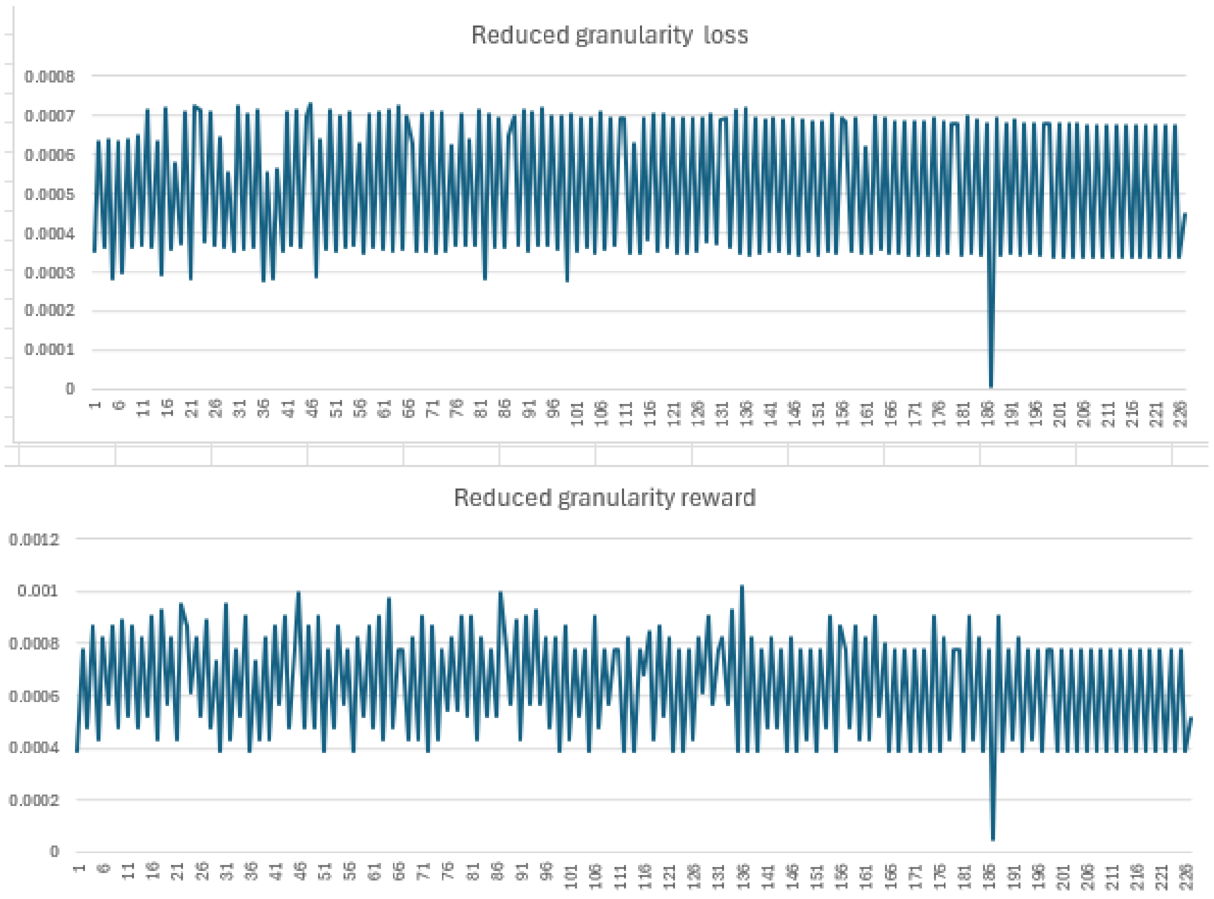


Fig. 18: The loss and reward (y axis) over time (x axis) of the second complete attempt at Goal 3 (0.02x Reward, 3.6 hours).

In an attempt to mitigate gradient explosion, all rewards were divided by ~50, such that fixing a generator provided a reward of 5. This prevented gradient explosion, but resulting metrics indicated that loss and reward were still oscillatory without trends indicating significant improvement. Due to the small negative slope of the loss in all tests, it was hypothesised that weight and bias adjustments were excessively small, leading to possibly accurate, but extremely slow learning. Previous tests had clearly shown that arbitrarily increasing the reward or learning rate could easily lead to gradient explosion, so it was decided to next focus testing entirely on determining an appropriate reward scale.

5.10.2 Adjusting Reward Scale

The limited success of training sessions from 2 to 15 hours in length in Section 5.10.1: Initial Attempts prompted an investigation into an appropriate reward scale to maximise training speed while avoiding gradient explosion.

First, a ballpark ideal reward value was calculated. This ballpark assumed that weights were initialised to 0.3 (a reasonable value with He initialisation) and would be changed to 0 over 50 episodes, or about 90 thousand training steps. Additionally, a simplified loss formula was used. The resulting ballpark reward was 1.0. As a result, rewards were rescaled such that fixing a generator provided a reward of 1.

Before testing this scale, floats in Deep-Q and MLP were replaced with doubles to minimise rounding errors. Additionally, the loss and reward logging feature was modified to additionally log bias and weight gradients. Specifically, for a given transition, only the magnitudes of the largest weight and bias gradients in the DQN are recorded. Note that while the loss and reward plots are directly calculated from incoming transitions as they are added to the replay buffer, the gradients are calculated later that frame during SGD, which may or may not use the most recent transition.

Training sessions were then run with 0.1x, 1x, 10x, and 100x reward multipliers. 10x and 100x reward resulted in gradient explosion, while 0.1x and 1x reward completed successfully. Testing moved forward with 1x reward as it was the highest reward that did not cause gradient explosion as well as being the same reward scale used by Mnih (2015) and Dabney (2018).

5.10.3 Error Clipping

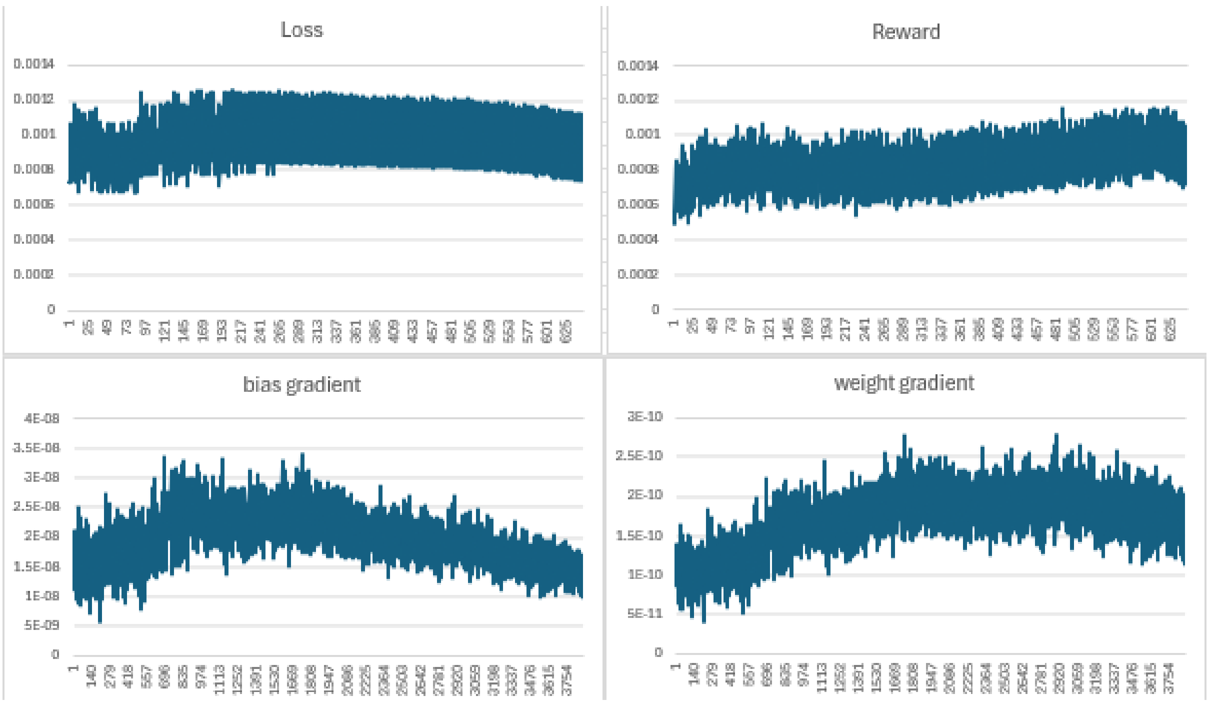
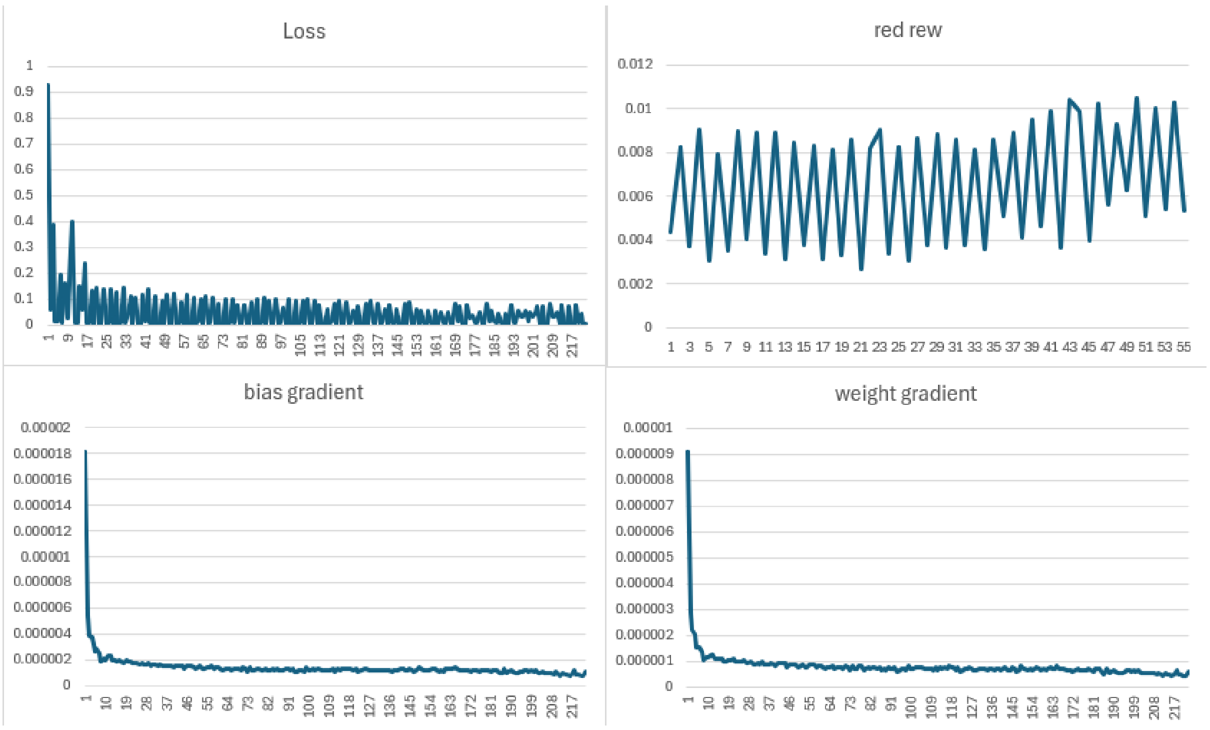


Fig. 19: Metrics (y axis) over time (x axis) of a 15-hour training session using a reward scale centred around R=1.

A 15-hour training session was conducted using the new reward scale, and the metrics showed some improvement over previous designs. Loss appeared to decrease slightly, while reward increased slightly over time. The average maximum weight gradient was 9 orders of magnitude smaller than the initialised weight values, indicating that adjustments to weight and bias were likely still too small. In later episodes, the agent exhibited some competency in picking up objects but came nowhere close to completing the level. These results were consistent with the DQN learning the optimal policy, but extremely slowly. It was hypothesized that if the training duration were longer, competency would have continued to improve.

Given that 1x reward appeared to produce excessively small adjustments and 10x reward appeared to cause gradient explosion, it was hypothesized that the reward window in which acceptably fast, stable training could take place was either small or non-existent. While reviewing literature for a way to increase the maximum stable reward, it was discovered that error clipping had been erroneously excluded from the Deep-Q implementation as mentioned in 5.3.2: Deep Q, and that it would likely solve the problem. Error clipping is a feature in Mnih's (2015) Deep-Q implementation that set the maximum and minimum value of the error term in SGD to 1 and -1 respectively. This prevents gradient explosion entirely at the potential expense of accuracy, as the maximum bias gradient magnitude becomes η and weight adjustments are clipped at a similar magnitude. This error clipping is effectively an implementation of the Huber loss function (Dabney, 2018), whereas the previously used loss function was Mean Squared Error (MSE).



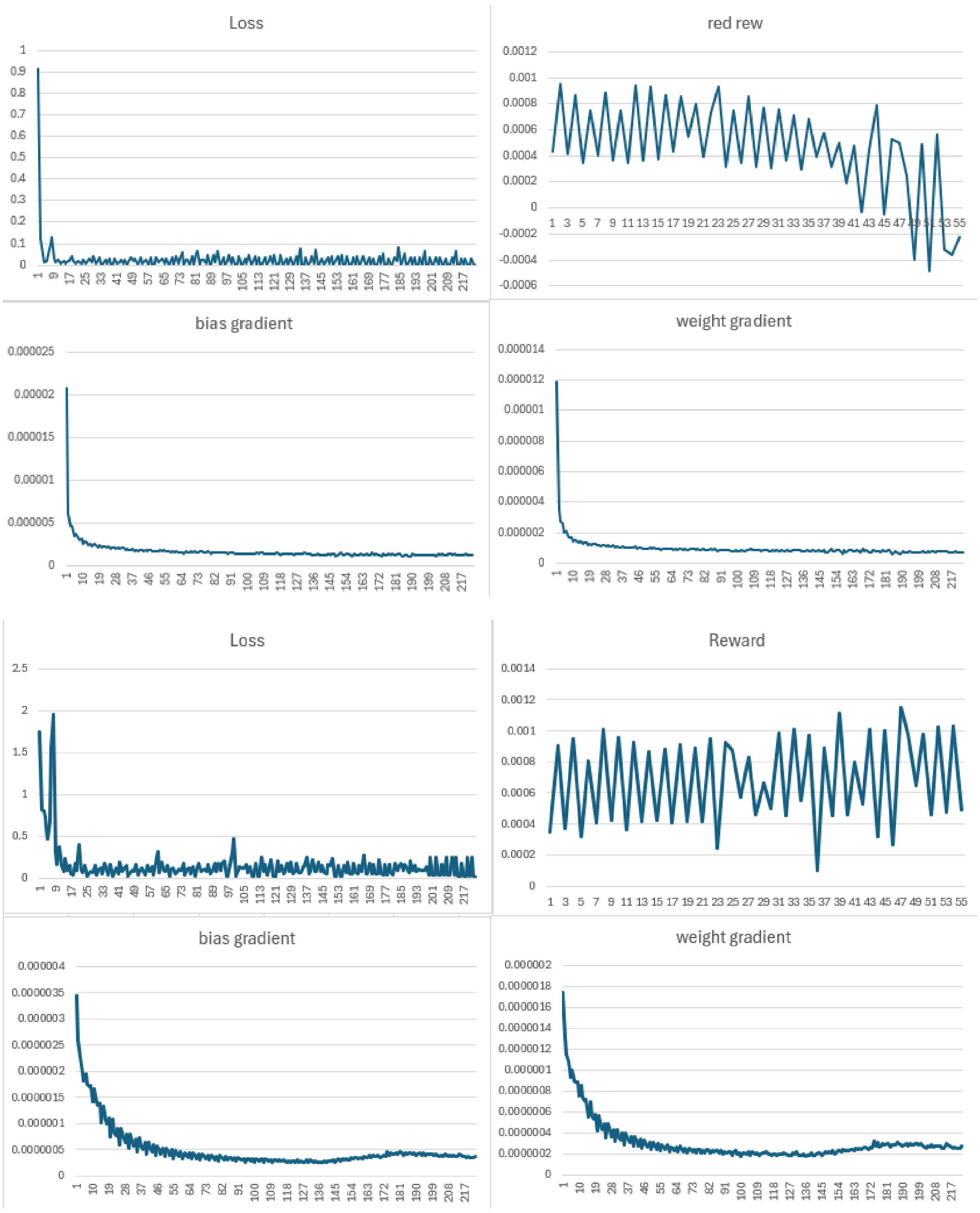


Fig. 20: Metrics (y axis) over time (x axis) of 3 1-hour training sessions with error clipping.
Top: 10x reward and 10x learning rate
Middle: 10x learning rate
Bottom: Default parameters

Combined with unrelated bugfixes, the introduction of error clipping resulted in significantly different metrics than in previous tests. All tests showed a decrease in loss, bias gradient, and weight gradient approximating exponential decay, though reward still did not significantly increase over time. Despite similar metrics, the subjective performance of the DQN trained with default parameters was substantially better than the DQNs trained with 10x learning rate or 10x reward and 10x learning rate, leading to the default hyperparameters again being chosen as the best performing variant of the system.

5.10.4 Final Experiments

The deadline for Goal 3 was moved to 1st September as detailed in Section 5.9 Adjusting Project Deadline. As the final round of testing in Section 5.10.2: Adjusting Reward Scale was completed on 28th August, a plan was drafted to maximise the chances of training a DQN capable of completing the first level by 1st September. The best performing system from the most recent round of testing continued to train for 22 hours, while 14 1-hour tests were performed on other computers using systems that varied slightly from the default parameters. The best performing 6 systems were to be run in parallel for the remainder of the time allotted.

5.10.4.1 1-Hour Tests

The 14 1-hour tests performed varied from the best performing system as follows:

- Discount = 0.97.
- Discount = 1.0.
- # of hidden layers = 1.
- # of neurons per hidden layer = 128.
- Target network update frequency = 1/200.
- Target network update frequency = 1/5000.
- Batch size = 150.
- Batch size = 250.
- Reward multiplier = 10.
- ε decays from 1.0 to 0.1 over 1/3 of training duration before remaining constant. Similar to implementation by Mnih (2015).
- Game is set to easy mode rather than hard.
- Alternative reward scheme 1: Teaches a specific sequence of actions rather than allowing freedom.
- Alternative reward scheme 2: Removes reward penalty for using the interact action pointlessly.
- Alternative reward scheme 3: All rewards are -1, 0, or 1. Identical to implementation by Mnih (2015).

5.10.4.2 22-Hour Test Results

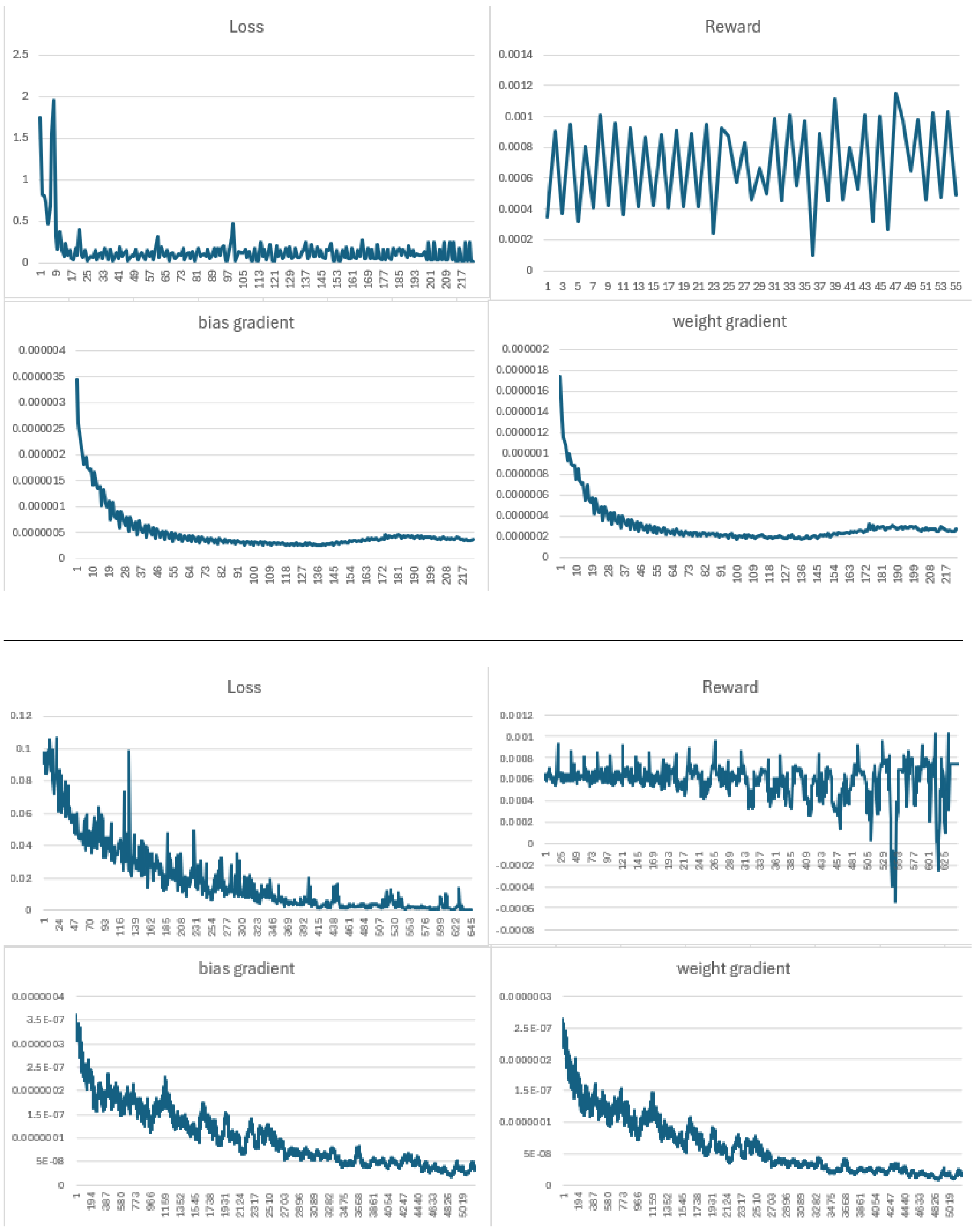


Fig. 21: The metrics (y axis) over time (x axis) of a system trained with default parameters over 1 hour (top) and 22 hours (bottom).

Metrics from the default parameters 22-hour training session showed a further decrease in loss over time, but no corresponding increase in reward. Additionally, reward appeared highly oscillatory, with magnitude of oscillation increasing over time.

The agent trained over 1 hour manages to consistently interact with the generator once, while the agent trained over 22 hours manages to consistently interact with the generator once and the pipe machine twice. This shows an improvement in competency, but no more competent than could plausibly be achieved with 3 hours of training. This data challenged the hypothesis that training was progressing accurately, but extremely slowly. Rather, training appeared to slow down over time.

5.10.5 63-Hour Tests

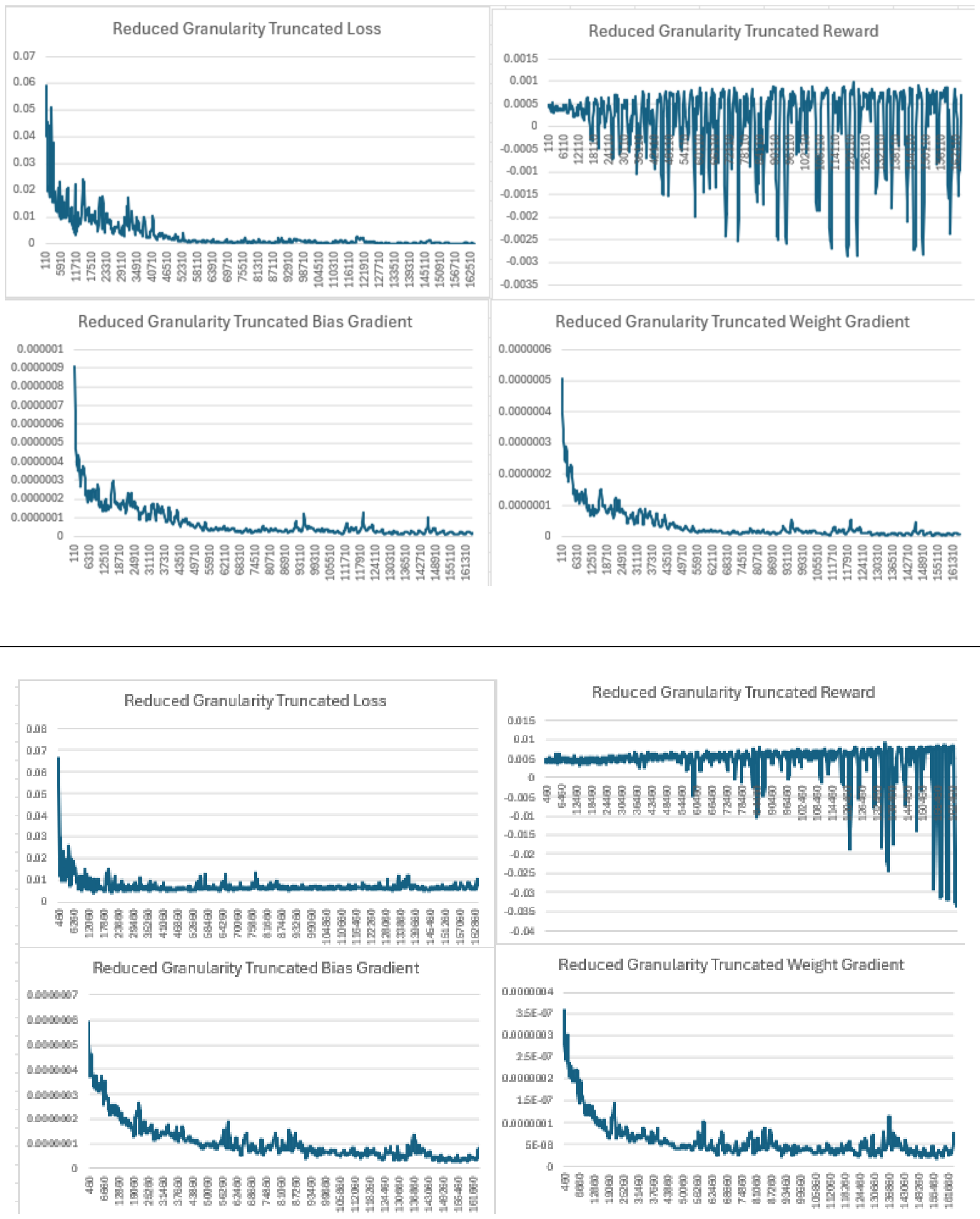


Fig. 22: Metrics (y axis) over time (x axis) of a system that linearly decreases ε from 1.0 to 0.1 over 1/3 of the training duration (above) and a system with a reward multiplier of 10 (below).

The 4 best performing variations of the default system from tests performed in Section 5.10.4.1: 1-Hour Tests were trained in parallel for 63 hours alongside 2 instances of the default system. 4 of these 6 training sessions stopped execution after roughly 20 hours due to an unknown problem. The 2 complete training sessions logged metrics as shown in Fig. 22. Compared to the 22-hour training session run by the default system the previous day, the loss of the completed 63-hour tests does not appear significantly different. The reward for both tests had a slightly increasing maximum value but showed a tendency to frequently oscillate into negative rewards. The only source of negative reward in these 2 tests was taking the interact action when not in range of an interactable, which strongly implies that the Q-value of the interact action was oscillating between high and low values.

6 Reflection

The original aim of this project was to create and train a neural network such that it can implement a policy to competently play the game Awaria, as well as a surrounding system that trains the network and provides an interface with the game application, effectively giving control of the player to the neural network.

To achieve these aims, a roadmap with 5 goals was created as described in Section 3: Project Goals. These goals served to guide development and provide objective evaluation criteria and are further discussed in Section 6.1: Evaluation of Success.

The final software system created, including the neural network, training algorithm, and interface with the game application are described in more detail in Section 6.2: Overview of Final Design.

Section 6.4: Possible Improvements discusses project decisions that could have been improved in hindsight.

Section 6.5: Future Work discusses potential directions for further development to better achieve the overall project aims.

6.1 Evaluation of Success

The primary measure of this project's success in achieving the project aims was the objective criteria described in detail in Section 3: Project Goals.

Goal 1 entailed the creation of the overall software framework and the communication interface that would later be used to give control of the agent to the neural network. This goal was considered successfully completed after performing a test that used this interface to send control signals to Awaria, comparing the received game states to hard coded values to verify functionality. A video of a successful run of the test can be found here: https://www.youtube.com/watch?v=KQFc1Sr7FXA.

Goal 2 entailed the addition of a NN and Deep-Q training algorithm to the design. This goal was considered successfully completed after performing a test that created artificial transitions with randomised environment states and using them to train the NN to solve a simple problem. A video of a successful run of the test can be found here: https://www.youtube.com/watch?v=E_BYf0Cyv_U.

Goal 3 was the beginning of the training phase of development and entailed training the NN to complete the first level of Awaria in at most one minute. This goal was partially completed, as the Gear Machine Test described in Section 5.4 demonstrated clear signs of learning, with the final evaluation showing a 100% success rate. However, all attempts at completing the first level resulted in a 0% success rate, with agents observed to repair the first generator in less than 1% of attempts. A video showing the behaviour of an agent that was trained for 63 hours can be found here: https://www.youtube.com/watch?v=VXc0E_CfOIY. Note that the behaviour of the agent is not identical between attempts. This is because ε is set to 10% in this example to exactly simulate the agent’s behaviour at the end of training.

Goals 4 and 5 were not attempted, as Goal 3 was intended to be a prerequisite goal.

6.2 Overview of Final Design

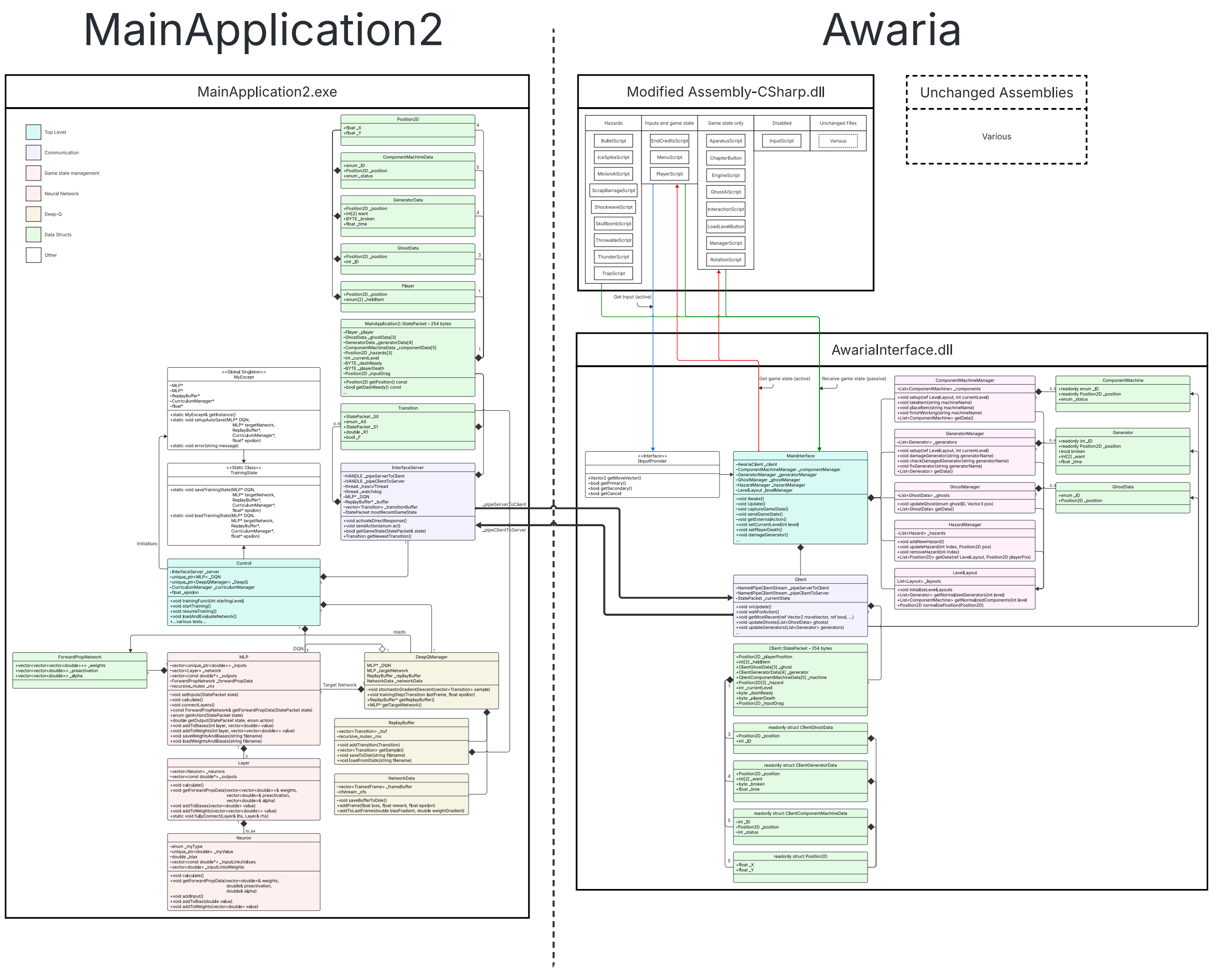


Fig. 23: A diagram of the final system architecture. Note that few changes were made after the completion of Goal 2, the system at that time being shown in Fig. 15 in Section 5.4.

The final design is similar in architecture to plans drafted following initial research. Notable additions include the MyExcept singleton, capable of saving the training state in the event of an error and the NetworkData class, which logs the loss, reward, and gradients of training over time. The only excluded feature in the system is curriculum learning. This feature was never fully implemented, as it was only relevant for Goals 4 and 5.

The system has 2 primary modes selected via command line argument. It can train a NN either from a partially trained state or from He initialisation to complete the first level, or it can load a pretrained network, enter the first level, and give control of the agent to the NN. A variety of tests can also be performed via command line argument, some of which are no longer fully compatible with the most recent version of the system.

As discussed in Section 6.1: Evaluation of Success, the system appears to work correctly, but NNs trained by the system show very low competence in practice.

6.3 Possible Improvements

Certain design choices made during and directly after initial research could have been improved in hindsight, the first of which being the game selected for training. The complexity of Awaria in comparison to games such as CartPole, LunarLander-v1, and BananaCollector was underestimated, leading to an unrealistically high expectations for training efficiency. Even the much more complex Atari 2600 games used by Mnih (2015) were shorter and provided denser rewards than Awaria. Alternatively, if Awaria was still chosen, a project roadmap with more tempered expectations and based on projects that train more comparable games, such as StarCraft or Doom would have been beneficial.

Additionally, 3 variations of the Deep-Q algorithm used by Mnih (2015) were discovered later in development that would likely have shown better results. They are the Double Q, Dueling, and Actor Critic algorithms. Rather than being specialisations of Deep-Q, these algorithms are intended as iterative improvements and showed improved performance when trained with the same Atari 2600 games as the original Deep-Q algorithm.

During implementation, particularly from the completion of Goal 2 onward, the capability of the NN to pass competency benchmarks despite major flaws in the system was repeatedly underestimated, resulting in costly setbacks. It would have been prudent to create significantly more tests of iterative difficulty and complexity to get a more precise evaluation of the current competency of the system.

6.4 Future Work

The project aims were only partially realised by the existing system. While adjustments such as formal, algorithmic hyperparameter tuning and expansion of the training algorithm to Double Q, Dueling, or Actor Critic may greatly improve performance, the fact remains that literature review indicates the existing system should be functioning substantially better than it is, even given the complexity of Awaria. Given the current state of the project, it is hypothesised that the continued failure of the system to come remotely close to completing the first level is indicative of a major flaw in the system rather than slightly suboptimal implementation. As stated in Section 6.3, the system has shown a tendency to display passable performance despite such flaws, and as such the best next step would be to create several similar tests of iterative difficulty to more easily narrow down any existing flaws and to aid future development.

7 Glossary

- BMLP – Bridged Multilayer Perceptron
- CNN – Convolutional Neural Network
- DQN – Deep-Q Network
- FCC – Fully Connected Cascade
- MDP – Markov Decision Process
- MLP – Multilayer Perceptron
- NN – Neural Network
- RNN – Recurrent Neural Network
- RQn – Research Question n
- SGD – Stochastic Gradient Descent

8 References

Arulkumaran, K., Deisenroth, M., Brundage, M. & Bharath A. (2017) Deep reinforcement learning: a brief survey. *IEEE Signal Processing Magazine,* 34(6), 26-38. https://doi.org/10.1109/MSP.2017.2743240

Chen, L., Wang, H., Zhao, J., Koutris, P. & Papailiopoulos, D. (2018) The effect of network width on the performance of large-batch training. *NIPS,* Montréal Canada, 3 December 2018. https://dl.acm.org/doi/10.5555/3327546.3327603.

Dabney, W., Rowland, M., Bellemare, M. & Munos, R. (2018) Distributional Reinforcement Learning with Quantile Regression. *The Thirty-Second Conference on Artificial Intelligence (AAAI-18),* 2-7 February 2018, New Orleans, Louisiana, USA. https://doi.org/10.48550/arXiv.1710.10044.

Dawson, C. & Wilby, R. (1998) An artificial neural network approach to rainfall-runoff modelling. *Hydrological Sciences,* 43(1), 47-66. https://doi.org/10.1080/02626669809492102.

Du, K., Leung, C., Mow, W. & Swamy M. (2022) Perceptron: learning, generalization, model selection, fault tolerance, and role in the deep learning era. *Mathematics,* 10(24), 4730. https://doi.org/10.3390/math10244730.

Gadgil, S., Xin, Y. & Xu, C. (2020) Solving the Lunar Lander problem under uncertainty using reinforcement learning. *IEEE SoutheastCon*, 28-29 March 2020, Raleigh, NC, USA. https://doi.org/10.1109/SoutheastCon44009.2020.9368267

He, K., Zhan, X., Ren, S. & Sun, J. (2015) Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. *IEEE Conference on Computer Vision (ICCV),* 7-13 December 2015, Santiago, Chile. https://doi.org/10.1109/ICCV.2015.123.

Hornik, K., Stichcombe, M. & White, H. (1989) Multilayer feedforward networks are universal approximators. *Neural Networks.* 2(5), 359-366. https://doi.org/10.1016/0893-6080(89)90020-8.

Rodríguez, A. & Buitrago, X. (2022) How to choose an activation function for deep learning. *Tekhnê,* 19(1), 23-32.

Hunter, D., Yu, H., Pukish, M., Kolbusz, J. & Wilamowski, B. (2012) Selection of proper neural network sizes and architectures – a comparative study. *IEEE Transactions on Industrial Informatics,* 8(2), 228-240. https://doi.org/10.1109/TII.2012.2187914.

Justesen, N., Bontrager, P., Togelius, J. & Risi, S. (2020) Deep learning for video game playing. *IEEE Transactions on Games,* 12(1). https://doi.org/10.1109/TG.2019.2896986.

Krizhevsky, A. (2017) ImageNet classification with deep convolutional neural networks. *Communications of the ACM.* 60(6), 84-90. https://doi.org/10.1145/3065386.

Masters, D. & Luschi, C. (2018) *Revisiting small batch training for deep neural networks.* ArXiv. https://doi.org/10.48550/arXiv.1804.07612. [Accessed 26 August, 2025].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015) *Human-level control through deep reinforcement learning.* Nature, 518, 529-533. https://doi.org/10.1038/nature14236

Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M & Stone, P. (2020) Curriculum learning for reinforcement learning domains: a framework and survey. *Journal of Machine Learning Research,* 21, 1-50. https://doi.org/10.48550/arXiv.2003.04960.

Nielsen, M. (2015) *Neural networks and deep learning.* Determination Press.

Obando-Ceron, J., Bellemare, M. & Castro, P. (2023) Small batch deep reinforcement learning. *NIPS '23: Proceedings of the 37th International Conference on Neural Information Processing Systems,* New Orleans, LA, USA, 10-16 December 2023. https://dl.acm.org/doi/10.5555/3666122.3667254.

O'Shea, K. & Nash, R. (2015) *An introduction to convolutional neural networks.* ResearchGate. https://doi.org/10.48550/arXiv.1511.08458. [Accessed 13 August 2025].

Paszke, A. (2025) *Reinforcement Learning (DQN) Tutorial.* Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.8.0+cu128 documentation

Popescu, M., Balas, V., Popescu, L. & Mastorakis, N. (2009) Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems,* 7(8), 579-588. https://dl.acm.org/doi/abs/10.5555/1639537.1639542.

Sejnowski, T. (2018) *The deep learning revolution*. The MIT Press.

Serrano, A. (2020) *Deep Q Network For Banana Collector Environment*. [Project report] GitHub. https://github.com/AntonioSerrano/Deep-Q-Network-for-Banana-Collector-environment/blob/master/report.md. [Accessed 30 August 2025].

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. (2016) Mastering the game of Go with deep neural networks and tree search. *Nature,* 529, 484-489.

Sinha, S., Bharadhwaj, H., Srinivas, A. & Garg, A. (2020) *D2RL: deep dense architectures in reinforcement learning.* arXiv. https://arxiv.org/abs/2010.09163. [Accessed 14 August 2025].

Stable Baselines (2021) *DQN*. [technical documentation]. https://stable-baselines.readthedocs.io/en/master/modules/dqn.html.

Strunk, G. (2024) *Source code for prt_rl.common.networks*. [PyTorch source code]. https://reinforcement-learning.readthedocs.io/en/latest/_modules/prt_rl/common/networks.html. [Accessed 30 August 2025].

Sutton, R. & Barto, A. (2018) *Reinforcement learning: an introduction,* 2nd edition. The MIT Press.

Unity Technologies (2025) *Running an Example Environment*. [ML Agents 4.0.0 tutorial]. https://docs.unity3d.com/Packages/com.unity.ml-agents%404.0/manual/Sample.html

Uzair, M. & Jamil, N. (2020) Effects of hidden layers on the efficiency of neural networks. *2020 IEEE 23rd International Multitopic Conference (INMIC),* The Islamia University of Bahawalpur, 5-7 November 2020. https://doi.org/10.1109/INMIC50486.2020.9318195.

Yang, L. & Shami, A. (2020) On hyperparameter optimization of machine learning algorithms: theory and practice. *Neurocomputing,* 415, 295-316. https://doi.org/10.1016/j.neucom.2020.07.061.

Yu, H., Samuels, D., Zhao, Y. & Guo Y. (2019) Architectures and accuracy of artificial neural network for disease classification from omics data. *BMC Genomics,* 20(167). https://doi.org/10.1186/s12864-019-5546-z.

Zhao, D., Wang, H., Zhu, Y. & Kun, S. (2016) Deep reinforcement learning with experience replay based on SARSA. *2016 IEEE Symposium Series on Computational Intelligence (SSCI),* Athens, Greece, 5-8 December 2023. http://dx.doi.org/10.1109/SSCI.2016.7849837