Technische Universität Berlin

Faculty IV – Electrical Engineering and Computer Science

Institute of Software Engineering and Theoretical Computer Science

Chair for Neurotechnology

Marchstr. 23, 10587 Berlin - https://www.tu.berlin/neuro

Bachelor's Thesis

Imitating Human Video Game Navigation with Neural Networks using Daggerfall Unity

Author: Suyog Srivatsa

Matriculation No.: 451626

Date of Submission: 30 May 2026

Reviewers: Prof. Dr. Benjamin Blankertz

Dr.-Ing. Stefan Fricke

Supervisor: Noah Schlegel

To my parents, in loving memory. I owe everything to their love, sacrifices, and the life they made possible for me.
To my brother, for his constant support, strength and encouragement.
To my friends, for always being there for me.

With sincere thanks to my supervisor, Noah, for his kindness, patience, empathy, guidance, professionalism, thoughtful mentorship, and unwavering support throughout this period. Without his help, this thesis could not have been brought to completion.

Abstract

This thesis intends to demonstrate how neural networks can be used to play games based on Unity and its efficacy. A unity-based open source game was used to create custom levels, and behaviour cloning was used to train two neural network policies. A pipeline for collecting, training, and evaluating human-like gameplay in Unity is proposed. The potential and limitations of using behaviour cloning when the dataset is small are brought to light.

Zusammenfassung

Diese Arbeit soll aufzeigen, wie neuronale Netze zum Spielen von Spielen auf Basis von Unity eingesetzt werden können und wie effektiv dies ist. Anhand eines auf Unity basierenden Open-Source-Spiels wurden benutzerdefinierte Levels erstellt, und mithilfe von Behavior Cloning wurden zwei neuronale Netz-Strategien trainiert. Es wird eine Pipeline zur Erfassung, zum Training und zur Bewertung menschenähnlichen Spielverhaltens in Unity vorgeschlagen. Dabei werden das Potenzial und die Grenzen des Einsatzes von Behavior Cloning bei kleinen Datensätzen aufgezeigt.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The application of algorithms and artificial intelligence (AI) to digital games has attracted significant research interest in recent years. Notable advances have been achieved in the development and optimisation of algorithms capable of solving or mastering games such as Sudoku, Chess, and Go, among others. These successes have established games as controlled yet expressive environments for studying learning, decision-making, and adaptive behaviour in artificial agents.

This thesis investigates how efficiently a neural network can play a game developed using the Unity Engine. While the integration of neural networks with Unity-based games is considered as part of the experimental setup, it is not the primary focus of this work.

In particular, this thesis examines how the architecture of a neural network influences gameplay efficiency and learning behaviour. Furthermore, it explores whether a network trained on human gameplay data is able to generalise its learned behaviour to a new, slightly more advanced level within a similar environment. Differences between human and artificial gameplay strategies are analysed, as well as the extent to which the network's performance improves over time in comparison to human players.

By addressing these questions, this thesis aims to contribute insights at the intersection of cognitive science, machine learning, and psychology.

1.2 Objective

The objective of this thesis is to evaluate whether a neural network can learn effective gameplay behaviour in a Unity-based game environment solely from human input data. To achieve this, the project was structured into *three* main components: game development, neural network optimisation, and training using recorded human gameplay.

An open-source game framework, Daggerfall Unity, was extended through the design and implementation of three custom levels. In addition, a telemetry recording system was integrated to capture detailed gameplay data, including keyboard input, mouse movement, and the spatial relationships between the player and objects within the game world.

The recorded human gameplay data was subsequently used to train a neural network. The central aim is to assess whether the network can learn to navigate and interact with the game environment based on this data, and to analyse its performance, generalisation capabilities, and behavioural characteristics in comparison to human players.

1.3 Scope

The focus of this thesis is the training of neural networks using human gameplay data through a sub-branch of imitation learning called behaviour cloning. The work is limited to learning from recorded human input and does not investigate other approaches, such as reinforcement learning, nor does it provide a comparative evaluation between imitation learning and reinforcement learning for game-playing tasks.

This thesis is also confined to a Unity-based game environment and primarily examines navigation and interaction behaviour within custom-designed levels. It does not aim to develop novel learning algorithms or to make general claims about optimal training strategies for games beyond the scope of the presented experimental setup.

2 Background

2.1 Human Gameplay Behaviour

The study of human players is a multidisciplinary field that draws on insights from Psychology, Cognitive Science, and human-computer interaction, among others. The combination of these areas allows us to analyse human behaviour, cognition, and the perception of emotions. This is particularly relevant in contexts where there is a need to model or simulate human behaviour or expression to enhance user experience or develop better, more immersive games [1].

An essential aspect of creating user-centric products is understanding how humans behave before modelling them. Computer Games provide us with an ideal test bed for this because they generate dynamic and complex player experiences, which are not easily captured by standard methods in the aforementioned fields [1]. Furthermore, games place the player in a constant interactive mode, which evokes complex cognitive and behavioural responses.

2.2 Imitating Human Behaviour with Artificial Intelligence

Several attempts have been made to develop agents that imitate human actions to varying degrees of success. Togelius et al. [2] divide these attempts into two categories: *direct* and *indirect* modelling or behaviour imitation.

In direct modelling or imitation, the most straightforward and frequently used approach [2, 3], some form of supervised learning is used to train the controller to mimic the actions of a human player. Here, the state of the controller is associated with what actions a human would take in the same state and is trained to output the same result as the player. Human gameplay data are used as training sets, with input parameters typically being descriptions of the game state, such as the game environment, and the target being the player's action.

In indirect modelling or behaviour, certain aspects of a human player's behaviour are used to model and optimise a reward or fitness function until the behaviour of the controller agrees with that of the human player. This method uses reinforcement learning to this end.

The main problem with the direct modelling approach has been generalisation. Since the agent is not rewarded for playing well when trained with supervised learning, when the agent is faced with a scenario that is unfamiliar to it, it is likely to take actions that are very unlike those of a human player in the same situation. The output is dependent on the quality of the training data; if the human player plays only moderately well, the results are likely to be much worse than the behaviour on which the agent was trained. There have, however, been significant strides in using imitation learning to train neural networks, which will be discussed further in 2.3.5.

2.3 Technologies Used in the Project

2.3.1 The Game Development Engine

2.3.1.1 Unity

Unity [4] is a real-time development platform primarily used for creating three-dimensional interactive applications, including video games. The engine integrates rendering and physics simulation capabilities and is accompanied by a graphical development environment known as the Unity Editor. Unity is designed as a general-purpose engine and supports a wide range of target platforms and application types, which makes it well-suited as a simulation environment for artificial intelligence research. Furthermore, the Unity Editor enables rapid prototyping and iterative development of interactive systems.

A Unity project consists of a collection of *Assets*, which may include textures, three-dimensional models, audio files, and scripts. A *Scene* is a special type of asset that defines the environment or level of a project. Scenes are composed of a hierarchical structure of *GameObjects*, which represent entities within the game environment. The behaviour of each GameObject is defined by the *Components* attached to it, allowing functionality to be composed in a modular manner.

2.3.1.2 The Unity Editor

The Unity Editor [5] is a graphical user interface used to design, configure, and test two-dimensional, three-dimensional, and virtual reality environments. Beyond its role in game development, the editor provides functionality that is particularly useful for AI research, such as the creation of custom scenes and controlled environments. It also allows the recording of

[1] Image source: Unity Technologies, https://learn.unity.com/tutorial/explore-the-unity-editor-1

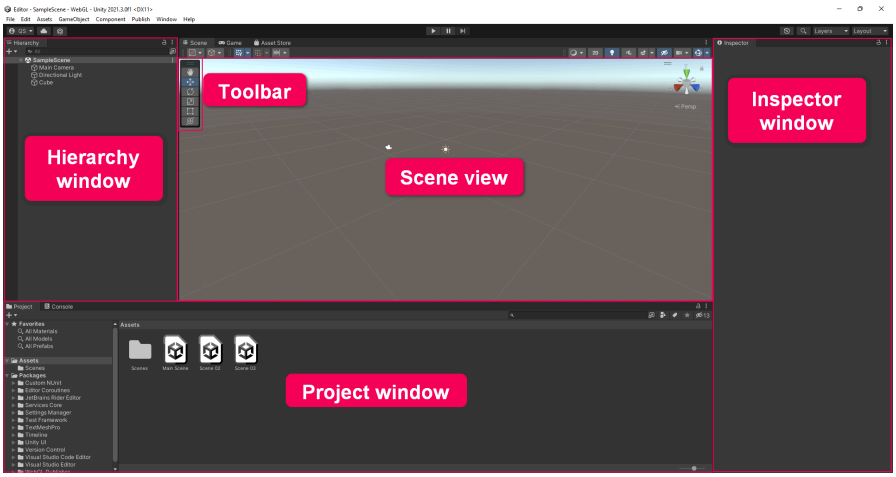


Figure 2.1: The Unity Editor [1]

expert demonstrations through user input devices such as the keyboard and mouse, which can be used as training data for learning-based agents.

2.3.2 Daggerfall Unity

The Elder Scrolls is a popular series of action role-playing video games created by Bethesda Game Studios. It is known for its rich lore, expansive open worlds, and immersive gameplay experiences. One of the most notable entries in the series is Daggerfall, released in 1994, which is set in the provinces of High Rock and Hammerfell in the game's universe.

Daggerfall Unity (DFU) [6] is a modern, open-source reimplementation of Daggerfall using the Unity game engine. It enhances the original game's graphics, improves user interface elements, and provides modding support while retaining the core gameplay elements. This project will not make use of any original DFU gameplay, but will utilise custom levels created using pre-existing DFU assets. The methodology will be discussed in detail in 3

2.3.3 Neural Networks

Definition and Biological Background

A neural network is a type of artificial intelligence that seeks to emulate the workings of the human brain. Gurney [7] defines a neural network as "an interconnected assembly of simple processing elements, called units or *nodes*, whose functionality is based on the structure of an animal neuron. The processing ability of the network is dependent on the inter-node connection strength or *weights* obtained by a process of adaptation to, or learning from, a set of

training patterns."

Biological neurons communicate via short-lived electrical signals or *spikes* in the voltage of the cell membrane. Inter-neuron connections are controlled by junctions or *synapses*, which are located on the branches of the neurons called *dendrites*. Each neuron receives input from a multitude of other neurons, and when the spike in voltage exceeds a certain limit, called the *threshold*, the neuron generates a voltage in response. This is then transmitted to other neurons via the *axon* [7].

Artificial Neural Networks abstract this biological process into a computational model. The nodes form the artificial equivalent of biological neurons, while weighted connections model the synapses. Each input to a node is multiplied by its corresponding weight, and the resulting values are summed to produce a weighted input. This value is then passed through an *activation function*, which determines the output of the node.

Components

A basic neural network consists of the following components [8]:

1. **Set of Synapses**: Weighted connections between nodes that transmit input signals.
2. **Integrator**: A mechanism that computes the weighted sum of the inputs.
3. **Activation Function**: A function applied to the integrated input to determine the node output. This function is non-linear, monotonically increasing, and bounded.
4. **Bias**: An external parameter that increases or decreases the net input of the activation function.

Mathematically, the output y of a single artificial neuron can be expressed as:

$$y = \varphi\left(\sum_{i=0}^{n} w_i x_i + b\right)$$

where $x_1, x_2, \ldots, x_n$ are the input signals, $w_1, w_2, \ldots, w_n$ are the synaptic weights, b is the bias, and $\varphi(z)$ is the activation function. The term $\sum_{i=0}^{n} w_i x_i + b$ is the operation of the integrator.

Different activation functions, such as the *Sigmoid function*, *tanh*, *ReLU* are used depending on the requirements. The Sigmoid Function is commonly used as the function, and its derivative are both continuous, and is defined as:

$$y = \frac{1}{1 + e^{-z}}$$

Architecture

Multiple artificial neurons are connected to form an Artificial Neural Network (ANN). An ANN generally consists of an input, hidden, and output layer. The hidden layer may consist of zero to several layers. The architecture of the neural network depends on the type of problem that needs to be solved. During the learning process, the number of layers, the number of nodes in each layer, the synapses, and the weights associated with them are adjusted. The difference between the expected and true value is calculated using the *loss function*, depending on which the weights are adjusted.

Training

Three primary training paradigms are used in machine learning: supervised learning, unsupervised learning, and reinforcement learning [8]. Supervised Learning uses a set of training data where the correct output is known, and the goal is to train the agent on the relationship between the input and output data. Unsupervised learning, on the other hand, uses training data where the results are unknown, and the goal is to find new patterns in the data. Reinforcement learning does not require a training data set, and the agent is trained to behave in an environment through trial-and-error interactions and an output with rewards and punishments is provided.

The backpropagation algorithm is most commonly used to train neural networks. The algorithm requires a set of parameters which influence the generalisation capabilities of the network.

Neural Networks are powerful computational tools, and are used in a panoply of fields, ranging from information processing to medicine and economics. They have become particularly popular recently due to their use in Large Language Models (LLM).

2.3.4 Neural Networks in Games

Deep Neural Networks (DNN) have been used to train and create games. While algorithms and heuristics exist that can play games such as Chess and Sudoku, neural networks add a layer to this because they can learn from pre-existing data and can also handle high-dimensional input (pixels). Using a combination of algorithms and DNNs has led to acquiring better results. D. Silver et al. [9] used a combination of supervised learning and reinforcement learning, along with MCTS, to train a DNN to play AlphaGo, achieving strong results against human expert players and other computer programs as well.

V. Mnih et al. [10] used a novel neural network termed a deep-Q network on Atari 2600

games, where the network, receiving only pixels and game score as inputs, was able to surpass all previously used algorithms and achieve a score comparable to that of professional human players.

2.3.5 Imitation Learning

Imitation learning is a machine learning paradigm in which an agent learns a behaviour policy from demonstrations rather than an explicitly defined reward function. Here, an expert provides examples of how a task should be performed, and the learning system attempts to reproduce the expert's behaviour when placed in similar states. This is also referred to as learning from demonstration [11].

A common form of imitation learning is behaviour cloning, where the problem is formalised as supervised learning. Given a dataset of state-action pairs (s, a), the model learns a policy $\pi(a|s)$ that predicts the expert action a from the observed state s. It is highly dependent on the quality and coverage of the demonstration data.

A known limitation of behaviour cloning is the covariate shift. During training, the model only observes states visited by the demonstrators, but during deployment, its own small mistakes can lead it into unfamiliar states. In these states, the model may produce poor actions. Ross et al. [12] introduced Dataset Aggregation (DAgger), an iterative algorithm that improves performance by collecting corrective expert data in states visited by the learned policy.

3 Methodology

3.1 Overview

This chapter discusses the experimental setup used in this thesis. The methodology consisted of selecting a suitable game environment, designing custom controllable levels within that environment, recording player gameplay demonstrations, converting the recorded gameplay into structured training data, and training neural network policies to be executed in Unity at runtime.

The objective was to determine whether behaviour cloning could be used to train an agent to imitate human navigation behaviour in a first-person three-dimensional game environment. This required the agent to move through custom dungeon levels, avoid or overcome obstacles, turn appropriately at junctions and walls, and reach a goal object. The environment, therefore, needed to be sufficiently controllable for experimental purposes. The data collection and inference pipeline also had to be integrated directly into the game engine to ensure the trained policy was evaluated under the same conditions as the demonstrations.

3.2 Requirements-Driven Environment Selection

The first stage of the project involved selecting a game environment suitable for behaviour cloning. The game had to support first-person movement, keyboard-and-mouse input, three-dimensional navigation, and custom level design. It also had to allow access to the internal game state so that player position, camera rotation, raycast observations, and action labels could be recorded during gameplay.

Daggerfall Unity (DFU) was selected because it is implemented in the widely used Unity game engine. This made it possible to create custom levels, enable or disable certain in-game features, and obtain accurate gameplay data during runtime. The use of Unity also allowed *Open Neural Network Exchange* (ONNX)-based model inference using Unity's *Barracuda* after neural network training for visual confirmation of the agent's ability. Building on a pre-existing game also meant that basic features such as movement and action, as well as art assets required for level creation, were already in existence and did not have to be implemented from scratch.

3.3 Level Building with Daggerfall Unity

To cater to the specific needs of this project, many pre-existing DFU features had to be disabled, with new ones implemented where necessary. Each dungeon has a clearly defined start, obstacle arrangement, and goal. Every expert demonstration corresponds to a known navigation route, thereby making it a supervised imitation learning problem.

3.3.1 World Data Editor

DFU comes equipped with a *World Data Editor* (WDE), which enables game designers to modify the game world without altering existing game data. Dungeons in DFU are assembled at runtime from reusable dungeon blocks, whereby each block contains a prefab arrangement of walls, floors, doors, lighting, etc. The WDE exposes these blocks as editable assets and provides a graphical interface to view and edit separate blocks to create a dungeon. These can then be saved as separate files for later use.

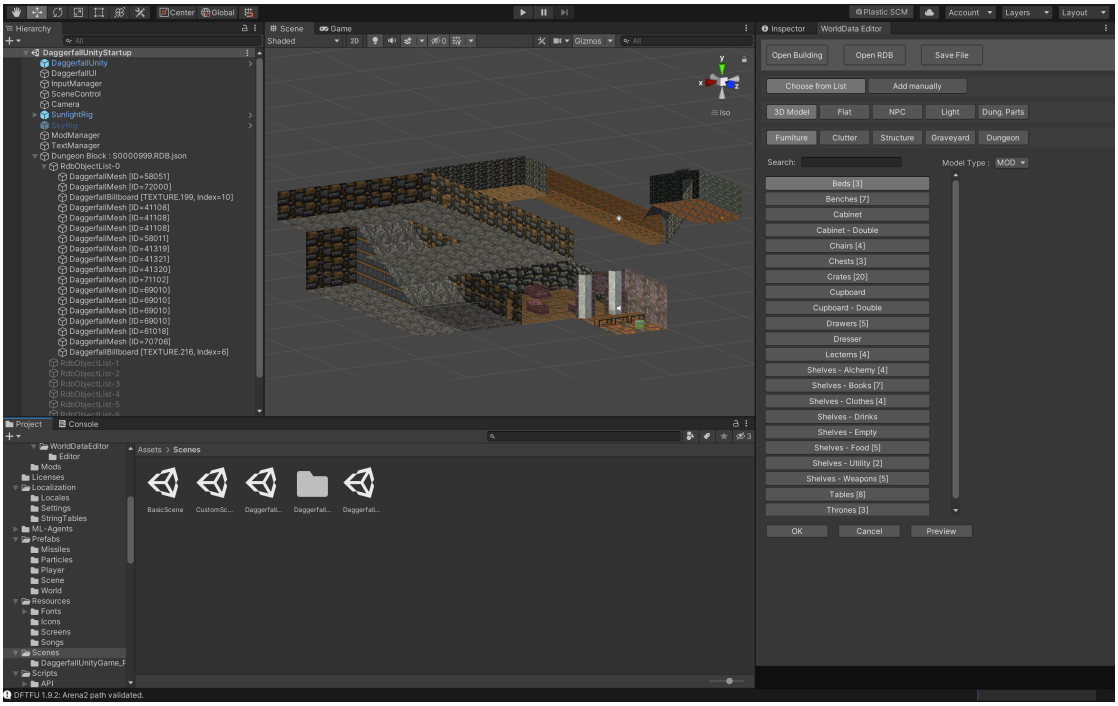


Figure 3.1: Modifying a Dungeon in DFU using the World Data Editor

The WDE is built on top of Daggerfall Unity's world data override system [13]. In the original game, world data is stored in binary data files and defines locations such as towns, villages and dungeons. DFU streams this data at runtime and then constructs the corresponding game

world. DFU distinguishes between exterior locations, called *Record Map Blocks* (RMB), and interior blocks called *Record Dungeon Blocks* (RDB). These override files are stored in JSON format, which makes them easier to inspect and modify than the original binary records.

```
{
    "Name": "S0000999.RDB",
    "Type": "Rdb",

    "RdbBlock": {
        "ModelReferenceList": [
            {
                "ModelId": "61028",
                "ModelIdNum": 61028,
                "Description": "LOZ"
            },
            ...
        ],

        "ObjectRootList": [
            {
                "RdbObjects": [
                    {
                        "Type": "Model",
                        "XPos": 0,
                        "YPos": 0,
                        "ZPos": 2,
                        "Resources": {
                            "ModelResource": {
                                "ModelIndex": 150,
                                "XRotation": 0,
                                "YRotation": 0,
                                "ZRotation": 0
                            }
                        }
                    }
                ]
            }
        ]
    }
}
```

Figure 3.2: Abbreviated structure of an exported `RDB.json` file.

A separate location override file specifies how these blocks are linked to each other by listing their block names and positions. This file also specifies where the player is spawned when the game is started. Figure 3.2 shows the structure of an exported dungeon block, and figure 3.3 shows how the location file links the RDB blocks into one dungeon.

```
{
    "Loaded": true,
    "Name": "Privateer's Hold",
    "RegionName": "Daggerfall",
    "HasDungeon": true,

    "MapTableData": {
        "LocationType": "DungeonLabyrinth",
        "DungeonType": "HumanStronghold",
        ...
    },

    "Dungeon": {
        "Header": { "BlockCount": 3 },
        ...
        "Blocks": [
            {
                "X": 0,
                "Z": -1,
                "IsStartingBlock": true,
                "BlockName": "S0000999.RDB",
                ...
            },
            {
                "X": -1,
                "Z": 0,
                "IsStartingBlock": false,
                "BlockName": "S0000998.RDB",
                ...
            },
            {
                "X": 0,
                "Z": 1,
                "IsStartingBlock": false,
                "BlockName": "S0000997.RDB",
                ...
            }
        ]
    }
}
```

Figure 3.3: Abbreviated structure of the location override file that links the custom RDB blocks.

For this thesis, three custom RDB dungeon blocks were designed using the WDE, namely `S0000997.RDB`, `S0000998.RDB`, and `S0000999.RDB`, where each block represents a separate level. The three blocks are linked to a single contiguous location by the override file, with `S0000999.RDB` marked as the starting block via the `IsStartingBlock` flag. This flag ensures the player character spawns at a deterministic position at the start of every episode, which is required for consistent expert demonstrations and reproducible policy evaluation. Along with modifying the `RDB.json`, the default DFU game flow, which routes the player through character creation, an introductory video, and a starting town, before any dungeon becomes accessible, was bypassed so that every session began directly inside the chosen dungeon.

Obstacles and goal objects were placed within each block by referencing existing DFU model IDs (see the `ModelReferenceList` in Figure 3.2). This reuse of existing model IDs enables the runtime binder pattern described in Section 3.3.4, since the model IDs are stable across runs and can be matched by patterns at scene-load time. Each level was play-tested

manually before recording demonstrations to ensure that the goal object was reachable and that the player could not exit the testing environment.

In addition to the dungeon layouts, the WDE (along with default code changes) was used to remove systems, originally present in the DFU game world, which were not relevant to the thesis. This included combat encounters, inventory interactions, the day-night cycle, save-and-load prompts, quest journal events, and NPC dialogue triggers. This ensured that the human player was not confronted with scenarios which the agent would not encounter at inference, and that the expert's action remained constrained to button-induced movement and look orientation.

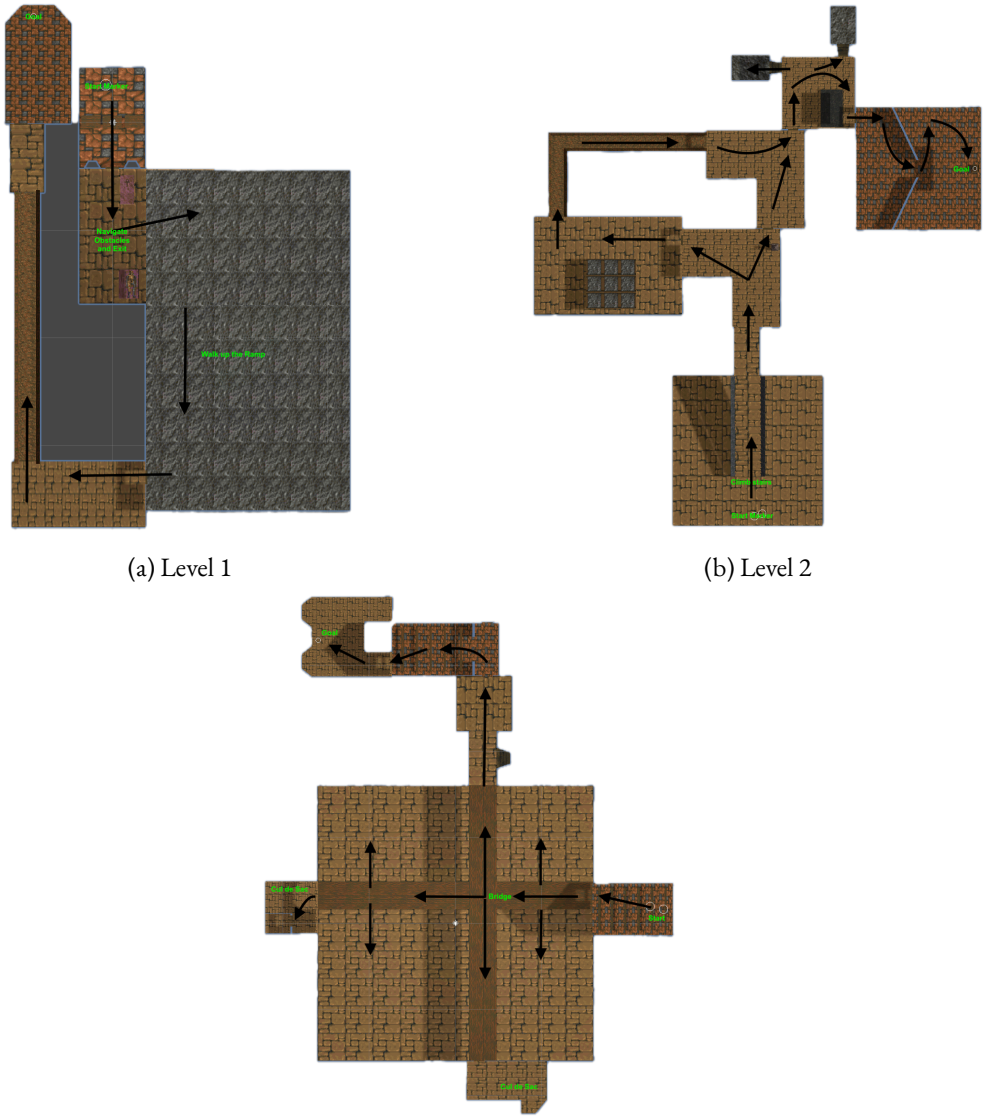
3.3.2 Level Design and Progression

Three custom levels with increasing difficulty were designed to permit a graded evaluation of the trained policy. Difficulty was determined by: (i) the number of obstacles the agent must clear by jumping, crouching, or walking around them, (ii) the number of decision points at which the agent must turn as opposed to continuing straight, and (iii) the length and complexity of the trajectory. The layouts of the three levels are shown in Figure 3.4.

Level 1 (block `S0000999.RDB`) is the simplest configuration. It consists of a starting chamber, where the player is spawned at the beginning of the game. This is determined by the presence of a Start Marker, added with the WDE, and the configuration of the location override as discussed in Section 3.3.1. A single jumpable obstacle is placed in the starting chamber, which must be cleared before entering the subsequent rooms. Each obstacle in these rooms can be overcome either by jumping multiple times or walking around it. The obstacles are existing DFU model assets — primarily furniture and rubble props — placed within each block by reference to their model IDs in the RDB override files. The third room has a ramp, which requires that a rectilinear trajectory be maintained or risk falling off the edge. This will require the player to re-traverse the room to reach the ramp again.

The goal object, used identically across all three levels, is a crown asset from the DFU model library. The crown is tagged at runtime by the `CrownGoalBinder` component (Section 3.3.4) so that observation rays striking it are classified as the goal category. When the player makes contact with the goal object of a level, the episode is marked as complete by the level-flow controller (Section 3.3.3) and the player character is teleported to the starting position of the next level. The same level structure is used for both expert demonstration recording and trained policy evaluation.

This level requires forward locomotion, obstacle clearance (by jumping, crouching, or walking around), and goal acquisition. Each room has precisely one entry and exit point and contains no junctions that demand a trajectory decision.



(a) Level 1

(b) Level 2

(c) Level 3

Figure 3.4: Top-down layouts of the three custom levels used in this thesis. In each level, the starting position and goal are indicated, with arrows tracing the player's trajectory from start to goal.

Level 2 (block `S0000998.RDB`) builds on Level 1 while incorporating active pathfinding. After spawning, the player must find and climb a flight of stairs and walk through a corridor to reach a room with two exits. One exit (the one to the right, see Figure 3.4b) has a shorter and more direct path to the goal, whereas the other is more convoluted. Taking the right exit, the player encounters a second decision point: one leading away from the goal and the other closer to it. Upon reaching the penultimate room, the player must navigate an object to reach the exit, which leads to the goal room, and the player must also navigate to an opening where the goal object is placed.

This level was designed so that reaching the goal is non-trivial, and a first-time player must use trial-and-error to navigate to it. The objective was to determine whether the trained agent could imitate route-selection behaviour in an environment where multiple paths were available, including one shorter trajectory toward the goal.

Level 3 (block `S0000997.RDB`) was used as the validation level and combines features from both Levels 1 and 2. The player is confronted with a bridge structure in the beginning, with multiple exits, only one of which leads to the goal. If the player falls off the bridge, there is no way to respawn, analogous to the ramp in Level 1, but with no possibility of recovery.

Level 3 serves as a data source against which validation loss is computed when training the neural network. It is also used as a generalisation probe during evaluation to determine the behaviour of the agent when confronted with a previously unseen geometry.

3.3.3 Teleportation and Episode Boundaries

While the three levels are construed as separate dungeon blocks within a single location, they are treated as three distinct episodes during data collection and training. The transition between levels, and the corresponding boundary recorded in the telemetry JSON is managed by a level-flow controller component that coordinates with the telemetry recorder.

DFU's existing level-transition systems, which are designed more for free-form exploration rather than linear sequences, were disabled in the beginning to constrain the agent's action space. This removed completion tracking entirely, and there was no other hook to notify the telemetry recorder of episode completion at the precise frame where the goal occurs. A custom level-flow controller and teleportation system were therefore implemented to enforce the linear progression required for accurate data capture. A teleportation system was also necessary to provide players with an authentic gameplay experience.

The level-flow controller is implemented as a singleton MonoBehaviour that persists after the scene loads. It holds an ordered array of `LevelSpec` entries. Each specification holds the name of the Start Marker GameObject's name (used to locate the next level's spawn point at runtime) and a fallback position with rotation in case the marker cannot be found. The

controller also tracks the index of the level currently being played.

When the player makes contact with the goal object of a level, the `GoalTrigger` component attached to the crown invokes `LevelFlowController.OnLevelCompleted`. This method then locates the active telemetry recorder via `FindObjectOfType<TelemetryRecorder>` and invokes `MarkEpisodeEnd(success: true)`. If a next level exists, it increments the level index and starts a coroutine that teleports the player to the next level's start marker.

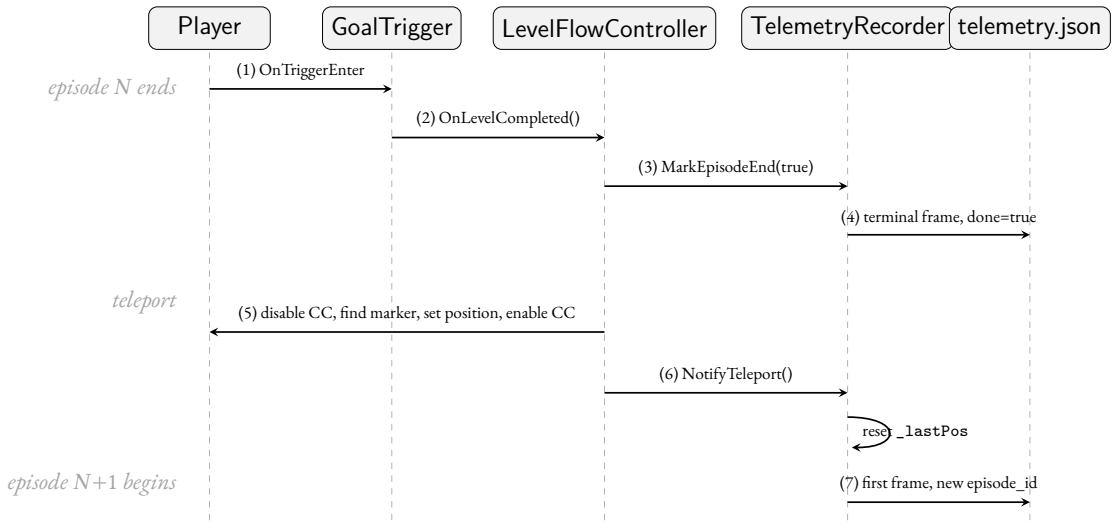


Figure 3.5: Level transition sequence between Unity components. Sequence of method calls and JSON writes during a level transition. The terminal frame of episode N carries `done=true` and `success=true`. After the teleport, the recorder's velocity baseline is reset by `NotifyTeleport` so that the first frame of episode $N+1$ has a zero velocity estimate rather than the spurious value that would otherwise be computed across the discontinuous position change.

Teleportation Procedure: The teleport coroutine performs the displacement in three steps. First, it disables the player's `CharacterController` component to prevent physics interactions during the discontinuous position change, giving rise to spurious velocity values which would taint data. Second, it searches the active scene for a GameObject whose name contains the configured Start Marker substring; if found, the player is placed at the GameObject's position and rotation; otherwise, the controller falls back to the hard-coded position and rotation. Third, the `CharacterController` is re-enabled. Once the player is in place, the coroutine notifies the telemetry recorder of the teleport by invoking `TelemetryRecorder.NotifyTeleport`.

Encoding Episode Boundaries: The end of an episode or level is demarcated in the telemetry recording by two fields: `done: true` and `success: true`. Frames recorded after this are marked with new episode IDs, thereby achieving the level separation required to train the

agent and support the training and validation split, where the first episode corresponds to Level 1 and so on.

3.3.4 Runtime Object Binding for Goals and Obstacles

To enable teleportation at the right moment, i.e., when the player makes contact with the goal object, and to identify and differentiate goal objects and obstacles from walls, floors, and ceilings, the goal and obstacle components require marker objects to be attached to the relevant GameObjects. This was essential for categorising raycast observations later when training the agent.

In DFU, dungeon geometry is not present in the scene when editing. DFU constructs each dungeon procedurally when the scene loads by combining prefab blocks. The resulting scene consists of GameObjects that cannot be accessed via the Unity editor. This is addressed by a runtime binder implemented as two parallel components, `CrownGoalBinder` and `ObstacleBinder`, attached to the `PlayerAdvanced` GameObject and run after the scene loads. Each binder has a list of name strings (for example, `Models [TEXTURE.216, Index=6]`) corresponding to GameObjects in the dungeon that should be treated as goals or obstacles, respectively. The binder enumerates and attaches a marker component to each object, which is then retrieved by the `GetComponentInParent<T>` query.

Both binders are functionally identical and differ only in which strings they match and which binders they attach. The runtime binder allows ray classification in environments where geometry is procedurally generated without modifying existing systems.

3.4 Data Collection Methodologies

3.4.1 Telemetry Recorder

The Telemetry Recorder is implemented as a Unity `MonoBehaviour` to the Player GameObject. It runs per frame during gameplay and writes a JSON file and a folder of per-frame images to a session directory. The recording is toggled on and off with the F8 key, and only frames captured when it is enabled are written.

On each frame in which the recorder is active, four types of data are captured from the game: (i) player kinematics, (ii) raycast observations, (iii) input events (`keyDown`, `keyUp`, `mouseDown`, `mouseUp`), and key hold duration, and (iv) image capture. Each frame record contains the fields shown in Figure 3.6.

```
{
    "v": 1,
    "frame": 142,
    "unixTimeMs": 1747318291534,
    "image": "frame_000142.jpg",
    "camera": {
        "position": [x, y, z],
        "rotationEuler": [x, y, z],
        "fov": 65.0,
        "near": 0.3,
        "far": 1000.0,
        "pixelWidth": 640,
        "pixelHeight": 360
    },
    "player": {
        "position": [x, y, z],
        "rotationEuler": [x, y, z]
    },
    "entities": [
        {
            "name": "...",
            "entityType": "...",
            "pose": {
                "position": [x, y, z],
                "rotationEuler": [x, y, z]
            },
            "screenBBox": [x, y, width, height]
        }
    ],
    "action": {
        "move": [mx, mz],
        "look_delta": [lx, ly],
        "jump": false,
        "sprint": false,
        "crouch": false,
        ...
    },
    "kin": {
        "vel_local": [vx, vz],
        "grounded": true
    },
    "rays": {
        "dist": [...14 values...],
        "cat":  [...14 values...]
    },
    "episode_id": "run_20260512_205136",
    "done": false,
    "success": false,
    "input": [
        { "type": "keyDown", "code": "W" }
    ],
    "keyHold": [ ... ],
    "mouseHold": [ ... ]
}
```

Figure 3.6: Schema of a single frame record in `telemetry.json`. Each session produces a JSON array of such records.

3.4.2 Raycast Observations

In each frame, the recorder casts 14 rays from the player's perspective and records, for each ray, a normalised distance and a category. The directions of the rays are specified in the camera objects as (yaw, pitch) pairs in degrees and are constant across sessions. Eleven rays from a horizontal fan spanning -90° to $+90^{\circ}$ in 18° steps. Two rays are angled forward and downward at 30° and 60° below the horizontal, giving the agent visibility of the floor. One ray is directed upwards. The fan of rays provides obstacle awareness; the downward rays help detect ramps and ledges, and the upward rays detect ceilings.

When a ray hits an object, it is classified into one of five categories: wall, floor, goal, jumpable obstacle, or nothing. The goal and obstacles are detected by their respective binders, as discussed in Section 3.3.4. The floor is inferred from the surface normal, where when a hit whose normal is within 45° is classified as floor, regardless of which ray strikes it. This handles ramps and stairs as well. A surface that does not match any of the above is classified as a wall. When no hit occurs within range, it is classified as nothing.

3.4.3 Action Labels

Each frame consists of nine action labels associated with it: two movement axes, two look-delta axes, and five button labels. The two movement axes are drawn from Unity's input system, and their values correspond to the A/D keys for horizontal movement and the W/S keys for vertical movement. The two look delta axes refer to mouse movement in the current frame. The five buttons correspond to the standard first-person-shooter actions: jump (Space), sprint (Left Shift), crouch (Left Control), attack (left mouse button), and use (E). Each is set to `true` on frames during which the corresponding key or button is held.

3.5 Dataset Construction

Demonstrations were collected from a human player who navigated the levels, as the telemetry recorder captured information on kinematics, button presses, and action. Each recording session covers all three levels in sequence and is then later split into training and validation levels. After the telemetry data was recorded, the gameplay sessions were converted to neural network training datasets to facilitate imitation learning.

3.5.1 Train and Validation Split

The dataset is partitioned along level boundaries, with Levels 1 and 2 contributing to the training dataset and Level 3 being the validation level, used to determine whether the agent

can generalise when presented with an unseen trajectory. This is implemented by reading the telemetry file and splitting levels based on episode numbering. The first episode is assigned to Level 1, the second to Level 2 and so on. The `--train-levels` and `--val-levels` command-line flags of the training script permit alternative partitions, if necessary.

3.5.2 Iterative Correction Data

When the agent was tested in Unity on the initial data collected, it displayed a systematic issue: when the agent encountered a wall, it neither looked away nor backed up; instead, it continued to push forward against the obstacle until the run was manually stopped. Ross et. al. describe this as a covariate shift. This happens when the model is trained on experts, and even though certain actions are recorded in the telemetry stream, the expert never uses them enough for the model to be trained on those actions. The agent, therefore, does not learn what to do when faced with a wall.

To mitigate this, a technique proposed by Ross et. al. called *Dataset Aggregation* (DAgger) was used, where the expert intervenes when the model is faced with an unseen scenario and deliberately adds corrective training data by using key presses and movements that would otherwise not have been used. This form of correction is not equivalent to the full DAgger described by Ross et. al., but it brings to light a central problem faced when training a model with Behaviour Cloning.

3.6 Neural Network Architecture

3.6.1 Input Feature Vector

The feature vector used by every policy is assembled per-frame from the telemetry record. It has a dimension of 92 in the default configuration, and the dimension 95 when the optional pose features are included. Its contents and the way each component is encoded are summarised in Table 3.1.

Camera angles are recorded as $(\sin\theta, \cos\theta)$ pairs rather than raw degrees or radians to prevent the $0^\circ \rightarrow 360^\circ$ wrap-around discontinuity, which would represent two identical orientations as numerically distinct inputs.

3.6.2 Output Action Vector

Both MLP and LSTM policies emit a nine-dimensional output vector per frame with a fixed layout. The first four components correspond to the two movement axes and the two look

Component	Dim.	Encoding
Raycast distances	14	clamped to $[0, 1]$
Raycast category one-hots	70	14×5 one-hot of {wall, floor, goal, obstacle, nothing}
Local planar velocity	2	$\mathbf{v}_{\text{local}}^{xz}/10\ \text{m s}^{-1}$, clamped to $[-1, +1]$
Grounded flag	1	$\{0, 1\}$ from `CharacterController.isGrounded`
Time delta	1	$\Delta t/0.25$ s, clamped to $[0, 1]$
Camera pitch	2	$(\sin\theta_{\text{pitch}}, \cos\theta_{\text{pitch}})$
Camera yaw	2	$(\sin\theta_{\text{yaw}}, \cos\theta_{\text{yaw}})$
Player position (optional)	3	$\mathbf{p}/250$ m, clamped to $[-1, +1]$
Total	**92 or 95**	

Table 3.1: Layout of the input feature vector used by the BC policies. Pose features are included when the `--include-pose` training flag is set.

axes. The remaining five components are binary-button logits corresponding to jump, sprint, crouch, attack, and use, in that order. This is shown in Figure 3.7.

At inference time, the movement components are passed through *tanh* to map them into $[-1, +1]$, the look components are passed through unchanged, and the button logits are passed through the sigmoid function and thresholded at 0.5 to produce binary actions. These transformations are applied identically to the MLP and LSTM Unity controllers so that the in-game behaviours of both architectures can be compared.

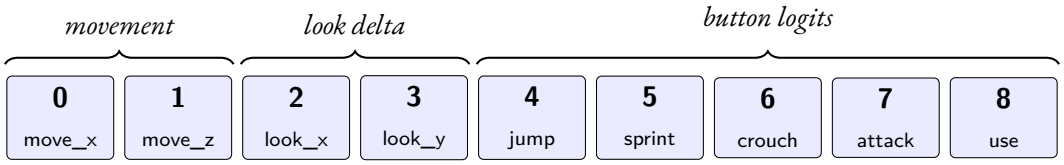


Figure 3.7: Layout of the nine-dimensional output action vector emitted by both the MLP and LSTM policies. Indices 0–1 represent movement, indices 2–3 represent look-delta predictions, and indices 4–8 represent button logits.

3.6.3 MLP Policy

The feed-forward policy is a multi-layer perceptron, whose hidden layers each consist of a linear projection, `ReLU` activation, and a dropout. The number of hidden layers and their widths are configurable, but the baseline uses three hidden layers of width 256 and a dropout of 0.10. The architecture is summarised in Figure 3.8.

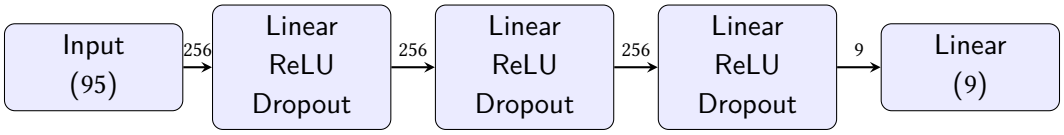


Figure 3.8: Architecture of the MLP policy. Each hidden block applies a linear projection, a ReLU activation, and a dropout layer with rate $p = 0.10$. The output head is a single linear layer with no non-linearity; its nine outputs correspond to two movement axes, two look axes, and five button logits.

The MLP is a memoryless function, and the predicted action on a frame depends only on the feature vector with no dependence on the previous frame. This can be used to test whether the MLP and LSTM achieve comparable results when trained with the same data.

3.6.4 LSTM Policy

The recurrent policy follows a per-frame, linear encoder pattern. The encoder is a single linear layer with ReLU activation and dropout that lifts each frame's feature vector into a latent representation of dimension H_{enc}. The LSTM operates on the resulting sequence of latents, maintaining a hidden state $(\mathbf{h}_t, \mathbf{c}_t)$ across time. The final hidden output at each layer is passed through a linear head to the nine-dimensional action vector. The architecture is illustrated in Figure 3.9.

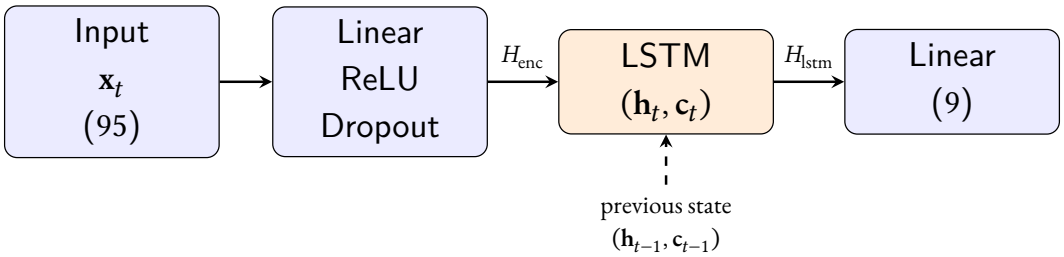


Figure 3.9: Architecture of the LSTM policy. A linear layer with ReLU activation and dropout first encodes each frame's feature vector. The encoded feature is then processed by the LSTM, which maintains hidden and cell states across time. A final linear layer maps the recurrent output to the nine-dimensional action vector.

The LSTM configuration uses $H_{\text{enc}} = 128$, $H_{\text{lstm}} = 128$, and a single LSTM layer, with dropout $p = 0.15$ applied in the encoder. This was altered during the training process to reduce overfitting and for better agent behaviour.

3.6.5 Loss Function

The loss is a weighted sum of three component losses, one per output type. Continuous movement loss is calculated using a mean-squared error on the *tanh*-transformed prediction, since movement targets are constrained to $[-1, +1]$:

$$\mathcal{L}_{\text{move}} = \frac{1}{2N} \sum_{i=1}^{N} \sum_{j \in \{x,z\}} \left(\tanh(\hat{a}_{ij}) - a_{ij}\right)^2.$$

Continuous look delta is calculated using a mean-squared error on the raw prediction, since look targets are clipped to $[-5^\circ, +5^\circ]$:

$$\mathcal{L}_{\text{look}} = \frac{1}{2N} \sum_{i=1}^{N} \sum_{j \in \{x,y\}} \left(\hat{a}_{ij} - a_{ij}\right)^2.$$

Binary buttons are calculated by a class-weighted binary cross-entropy on the raw logits, where the per-class positive weight w_k accounts for the strong imbalance between pressed and unpressed frames:

$$\mathcal{L}_{\text{button}} = \frac{1}{5N} \sum_{i=1}^{N} \sum_{k=1}^{5} \text{BCE}(\hat{a}_{ik}, a_{ik}; w_k).$$

The per-class weight is computed from the training set as $w_k = \min(25, n_k^- / n_k^+)$, where n_k^- and n_k^+ are the counts of negative and positive labels for button k.

The total loss is a weighted sum:

$$\mathcal{L} = \mathcal{L}_{\text{move}} + \lambda_{\text{look}} \mathcal{L}_{\text{look}} + \lambda_{\text{button}} \mathcal{L}_{\text{button}}.$$

3.6.6 Training Procedures

Both policies are trained with the AdamW optimiser [14] at a learning rate of $\eta = 2 \times 10^{-4}$ and weight decay 10^{-4}, with gradient norm clipped to unity before each optimiser step. The MLP operates on shuffled per-frame mini-batches of 256 independent (feature, target) pairs, while the LSTM operates on mini-batches of 32 sequence windows of length $L = 64$ and stride $S = 16$ assembled from the training episodes, with the recurrent hidden state zero-initialised at the start of every window (truncated backpropagation through time, no inter-window statefulness). For both architectures, the validation loss is computed on Level 3 after each epoch and the lowest-loss checkpoint is retained as the final model.

3.7 Unity Inference Setup

3.7.1 ONNX Export

After training terminates, the policy learned from PyTorch is exported to the Open Neural Network Exchange (ONNX) format, which can be utilised by Unity's Barracuda inference at runtime. The Unity-side controller reconstructs the same feature vector used during training, normalises it using the statistics saved in `metadata.json`, applies the trained policy, post-processes the output action vector, and injects the resulting actions into Daggerfall Unity's input system.

The MLP policy exports as a single-input, single-output ONNX graph. The input is a single normalised feature vector of shape $(1, d)$, where d is the feature dimension recorded in the metadata file. The output is the nine-dimensional action vector described in Section 3.6.2.

The LSTM policy, on the other hand, takes three inputs: `features`, `hidden_h`, and `hidden_c`. The outputs are: `actions`, `next_hidden_h`, `next_hidden_c`. It returns the current action prediction and the updated hidden and cell states. This allows Unity to execute the recurrent policy one frame at a time while maintaining LSTM memory outside the ONNX graph. For both architectures, the export process also writes a `metadata.json` file containing the feature schema, normalisation statistics, target names, training split information, and, for the LSTM, the recurrent architecture parameters.

3.7.2 MLP and LSTM Telemetry Policy Controller

The MLP and LSTM inference controllers are Unity `Monobehaviours` attached to the PlayerAdvanced GameObject. When enabled, they load the ONNX model, parse the accompanying `metadata.json` to recover the input feature vector, the ray count, and the category count, among others. On each frame where it is enabled, the controller performs the same actions as the recorded telemetry and casts fourteen rays from the player, classifying objects in the scene with the binder, adds velocity and camera pitch and yaw. The output vector is then converted into game action, which is then injected into Unity's input system.

The LSTM controller additionally maintains the hidden state across frames. The hidden and cell states are stored as plain C# float arrays of length `lstm_layers` × `lstm_hidden` and are re-packed into Barracuda tensors on each inference call. The controller reads the number of LSTM layers and hidden-state size from the `metadata.json` file, allocates hidden and cell state buffers, and passes these buffers into the ONNX step model in every frame. After each inference step, the controller copies the returned hidden and cell states back into its internal buffers. These updated states are then used as inputs on the next frame.

3.7.3 Inference-Time Post-Processing

The nine-dimensional output tensor produced by the Barracuda worker is converted into a game action by applying the same transformations used in the training loss function as discussed in Section 3.6.5.

3.7.4 Input Injection into Daggerfall Unity

To ensure that the trained policy plays the game like a human player, the policy does not bypass DFU's input system. This enables a valid comparison between the trained policy and the human player. An `InjectPolicyInput` method was added to DFU's `InputManager` to facilitate this. From DFU's perspective, this makes the policy input indistinguishable from human interaction.

4 Results

4.1 Overview

This chapter presents the empirical results obtained from training and evaluating the imitation-learning policies described in Chapter 3. Data used to train the policies and recorded from human players are presented in Section 4.2, including the number of demonstrations collected, the frequency of each action type, and the coverage of raycast observations that constitute the agent's world-view. Section 4.3 reports the offline-training behaviour of the Multi-Layer Perceptron (MLP) and the Long Short-Term Memory (LSTM) policies. Section 4.4 discusses the in-game behaviour of the trained policies when deployed in Daggerfall Unity (DFU) and examines the failure modes that remained after the final training, relating them to the demonstration data and the limitations of behavioural cloning.

Throughout this chapter, results are derived directly from the telemetry recorder, the training script's per-epoch metadata, and in-game evaluation runs. Performance results are based on gameplay observations.

4.2 Dataset Summary

The behaviour of an agent trained using behavioural cloning is based on and interpreted with the demonstrations used to train it. Two demonstration corpora are reported. The *Expert* corpus consists of unaided traversals of each level by a single human demonstrator (the author), playing each level from start to the goal. The *Correction* corpus consists of short recordings collected in the style of DAgger, in which the demonstrator was placed in states that the trained policy had been observed to fail. These typically occurred when the agent hit a wall or encountered an obstacle. The Correction corpus was added later in response to systematic failures observed during early testing in turning.

The *Novice* and *Expert New Player* corpora were collected from human players who had not previously seen the game. They were differentiated into these categories based on the amount of previous experience in playing digital games, with Novice players having little to no experience, and Expert New Players being regular gamers. Both sets of people were given

the same instructions before they played the game regarding movement, jumping, looking around and finishing the game.

4.2.1 Frame and Episode Counts

Table 4.1 shows the size of each corpus, including the number of files, the number of frames in all files combined, the number of episodes in all files, and the levels covered. Each Expert telemetry recording contains a session in which the demonstrator completed Levels 1, 2, and 3 in sequence, aided by the teleportation process. The telemetry recorder marks the level boundary at the frame at which the goal was attained so that the data can then be split into per-level episodes.

Correction recordings are short, focused demonstrations, each covering a single or a few corrective behaviours at one location, in one level. The novice and expert recordings are collected from human players who had previously not seen the game.

Dataset	Files	Frames	Episodes	Levels
Expert	17	76,849	51	L1, L2, L3
Correction	13	12,153	13	L1
Novice (external)	3	45,282	9	L1, L2, L3
Expert (external)	2	15,941	6	L1, L2, L3
Combined	35	150,225	79	L1, L2, L3

Table 4.1: Summary of the telemetry datasets used for training and evaluation.

4.2.2 Action Distribution

Table 4.2 reports the per-frame frequency of each action label across the corpora. The dominant action in all of them is forward movement. Look movement is the second most common active label. The button events (jump and crouch) are imbalanced in comparison. This led the model to learn forward movement very well, but the other actions still had to be scaled during gameplay to ensure that they were recognised.

4.2.3 Raycast Observation Coverage

The tables below summarise the per-category ray hit frequency with other frame-level statistics that characterise typical observation patterns. Walls account for the largest share of ray samples in the Expert corpus, which is consistent with the geometry of the dungeons used in

Action Type	Expert	Correction	Novice (external)	Expert (external)
Movement (any)	70.28%	40.54%	60.52%	91.83%
Forward movement	68.38%	36.30%	59.06%	86.73%
Look movement	23.62%	23.15%	33.04%	34.07%
Jump	2.00%	1.12%	1.06%	1.14%
Sprint	2.84%	0.00%	0.00%	3.20%
Crouch	0.00%	0.00%	0.00%	0.04%
Attack	0.04%	0.00%	0.16%	4.89%
Use	0.00%	0.00%	0.06%	0.00%
No-op	19.85%	44.54%	25.45%	6.88%

Table 4.2: Per-frame frequency of each action label across the demonstration corpora.

the project: most rays cast from eye-level strike a front wall. Goals are rare because only three exist.

Statistic	Value
Ray hits classified as wall	76.79%
Ray hits classified as floor	14.95%
Ray hits classified as goal	0.87%
Ray hits classified as obstacle	2.39%
Ray hits classified as nothing	5.01%
Frame with any goal-category ray visible	6.37%
Frame with front ray on wall, distance $\leq 20\%$	22.86%
Frame with front ray on obstacle, distance $\leq 20\%$	0.23%
Frame with near-front rays on wall (any), distance $\leq 20\%$	58.49%
Frame with near-front rays on obstacle (any), distance $\leq 20\%$	3.73%

Table 4.3: Raycast observation coverage in the Expert telemetry. Per-category percentages are over all ray samples; the remaining rows are over frames in which at least one ray fired.

Statistic	Value
Ray hits classified as wall	75.90%
Ray hits classified as floor	13.93%
Ray hits classified as goal	0.00%
Ray hits classified as obstacle	8.99%
Ray hits classified as nothing	1.17%
Frame with any goal-category ray visible	0.00%
Frame with front ray on wall, distance $\leq$ 20%	36.42%
Frame with front ray on obstacle, distance $\leq$ 20%	6.80%
Frame with near-front rays on wall (any), distance $\leq$ 20%	62.36%
Frame with near-front rays on obstacle (any), distance $\leq$ 20%	17.29%

Table 4.4: Raycast observation coverage in the Correction telemetry. Per-category percentages are over all ray samples; the remaining rows are over frames in which at least one ray fired.

Statistic	Value
Ray hits classified as wall	77.22%
Ray hits classified as floor	14.89%
Ray hits classified as goal	0.69%
Ray hits classified as obstacle	3.29%
Ray hits classified as nothing	3.91%
Frame with any goal-category ray visible	4.09%
Frame with front ray on wall, distance $\leq$ 20%	16.34%
Frame with front ray on obstacle, distance $\leq$ 20%	2.39%
Frame with near-front rays on wall (any), distance $\leq$ 20%	52.42%
Frame with near-front rays on obstacle (any), distance $\leq$ 20%	6.58%

Table 4.5: Raycast observation coverage in the Novice (external) telemetry. Per-category percentages are over all ray samples; the remaining rows are over frames in which at least one ray fired.

Statistic	Value
Ray hits classified as wall	76.23%
Ray hits classified as floor	15.19%
Ray hits classified as goal	0.62%
Ray hits classified as obstacle	2.37%
Ray hits classified as nothing	5.59%
Frame with any goal-category ray visible	4.60%
Frame with front ray on wall, distance $\leq 20\%$	16.19%
Frame with front ray on obstacle, distance $\leq 20\%$	1.98%
Frame with near-front rays on wall (any), distance $\leq 20\%$	58.39%
Frame with near-front rays on obstacle (any), distance $\leq 20\%$	6.08%

Table 4.6: Raycast observation coverage in the Expert (external) telemetry. Per-category percentages are over all ray samples; the remaining rows are over frames in which at least one ray fired.

4.3 Offline Training Results

Both the MLP and the LSTM were trained on a combination of the Expert, Correction, Novice, and Expert New Player data. Training loss and validation curves were generated after each run. Since the LSTM was observed to consistently outperform the MLP, training on the MLP was stopped after a few runs. Figure 4.1 shows the training and validation loss curves of the LSTM after overfitting was corrected, and other training parameter changes were made. The image on the right also shows the per-head losses. It is important to note that move losses are consistently low. The look loss on this run stayed constant while button losses were very high in the beginning and came down, but are still noticeably higher than either the move or the look loss.

4.4 In-Game Evaluation Results

The trained policies were deployed in the same levels used to collect the demonstrations. Each policy was evaluated based on how far the agent progressed in each level. The metrics used for this were the number of distinct obstacles the agent was able to clear through a successful jump, the ability to recover from a stuck position (turning around when faced against a wall or recognising and walking around an obstacle), and reaching the goal.

The MLP was rarely able to move from the starting position and reach the first obstacle. It frequently entered tight rotation patterns. It is to be noted, however, that when trained

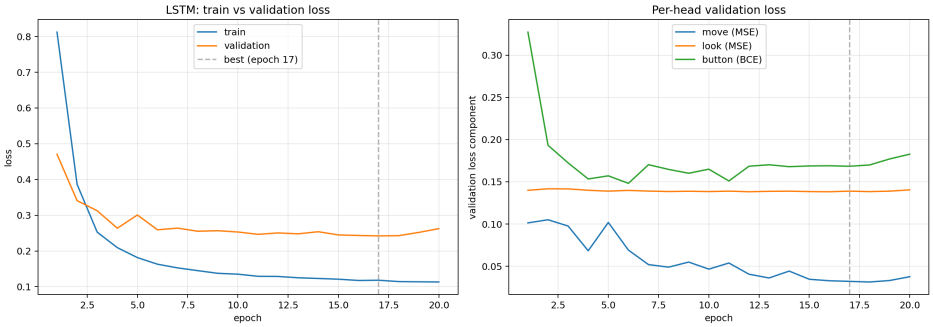


Figure 4.1: Training and validation loss during LSTM training. The training loss is computed on the demonstration sequences used for optimisation, while the validation loss is computed on held-out data and is used to estimate how well the model generalises beyond the training set.

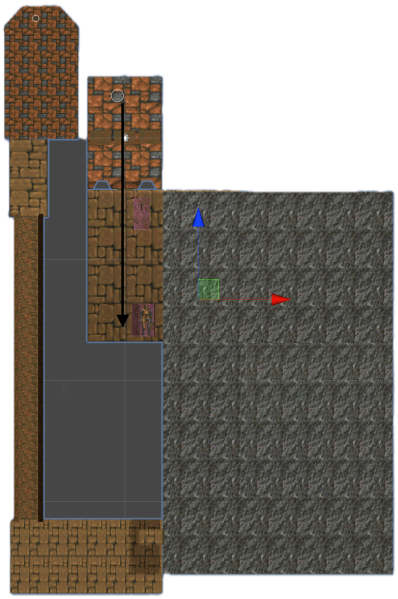


Figure 4.2: The LSTM agent was able to start the game, clear the first obstacle and enter the second room, where it became stuck. The MLP agent was unable to do this and was stuck at the starting point.

on only one data file, the MLP was able to clear the first obstacle before getting stuck at a wall; it was never able to clear the first room. The LSTM performed better. As illustrated in Figure 4.2, it was able to clear the first obstacle by jumping, clear the first room, and go into the second room. When encountering obstacles in the second room, it was able to walk around them. When faced against a wall, the LSTM, after parameter fixes, was able to turn around and move forward. This was when the LSTM was trained only on the Expert and Corrections corpora.

To assess whether the results depended on the demonstrator, three LSTM policies were trained: Expert + Corrections, Novice, and Expert New Player. The Expert + Corrections showed the best results. The policy trained on Novice and Expert New Player data exhibited the worst behaviour, with the agent being stuck in rotation without forward movement at the starting point. This could be a result of more training data available from the expert, whereas only 3 and 2 novice and expert new player demonstrations existed respectively.

Consistent problems were noted during the evaluation. The first was being stuck at an obstacle or a wall, with the agent unable to turn after encountering either. Certain actions were also under-represented in the data, which led to the agent not being able to learn them as well as others: the agent moved very well, whereas jumping and turning around were unreliable. The third failure was the inability of the agent to find exits after the first room or to fully comprehend the layout of each level. Pure behavioural cloning has no mechanism for the agent to recognise a dead-end and revise its trajectory. These failures can also be attributed to the lack of sufficient training data covering recovery from off-trajectory states.

4.5 Summary of Findings

This pipeline produced a policy that was able to start the game, clear obstacles by jumping and turn around at walls. The MLP was not able to do this, except in one edge case in which it was trained on a single Expert telemetry file. The agent was also unable to learn the layout of the levels and find exits accurately. This can be attributed to the lack of sufficient training data and brings forth a shortcoming of pure behavioural cloning.

5 Discussion

The empirical results in Chapter 4 show three observations. First, the MLP was unable to integrate the context required for jump timing and accurate turning, and its in-game behaviour reflected this directly. It either remained stationary or entered tight rotations without moving forward. The LSTM, given the same input, produced obstacle clearance and wall recovery. It was able to generalise better than the MLP. This is consistent with existing work in Behaviour Cloning.

Second, the composition of the demonstration data mattered as much as the architectural choice. The wall-turning failure faced by the LSTM was not resolved by changing the hyperparameters alone. It also required additional corrective training recorded specifically at the spots where the policy had been observed to fail, and was added to the training set. This pattern, where a behaviour remained absent until specifically trained, is an example of covariate shift that motivates DAgger. This shows in the Novice and Expert New Player corpora as well because action distribution remained consistent across demonstrators.

Third, the failures that remain are systematic. Exit-finding requires an understanding of the layout of the game, which cannot be achieved with the limited data on which the models were trained. This also brings to light a limitation of pure Behaviour Cloning, which only demonstrates successful trajectories and contains no information on how to recover when faced with a scenario that human players might not have encountered.

5.1 Conclusion

This thesis intended to determine whether a behaviour cloning policy trained on a modest corpus of human players can autonomously play custom-built Daggerfall Unity levels. A policy with recurrent memory trained on a corpus and supplemented with correction data and expert players can perform certain similar actions in the same environment. Since the data was sparse, the agent was unable to completely determine the layout of the levels and fully reach the goal.

5.2 Future Work

Behaviour cloning can be augmented when combined with reinforcement learning, which can help the agent understand failures better and learn from them. Reinforcement learning provides the optimisation signal that behaviour cloning does not. This can help the agent understand the layout and obstacle path better and eliminate failure modes. This would even work with the current dataset.

Tools and Additional Sources of Help

During the preparation of this thesis, generative AI and language-support tools were used as auxiliary aids. ChatGPT[1] and Claude Code[2] were used to assist with the proofreading of text and the creation of functions used during development. Grammarly[3] was used for spelling, grammar, and style suggestions. DeepL[4] was used for translation support. All suggested content was reviewed, edited, and incorporated under my own responsibility.

[1] https://chatgpt.com/
[2] https://www.anthropic.com/claude-code
[3] https://www.grammarly.com/
[4] https://www.deepl.com/

Bibliography

[1] G. N. Yannakakis and J. Togelius, *Player Modeling*. Cham: Springer Nature Switzerland, 2025, pp. 315–335. [Online]. Available: https://doi.org/10.1007/978-3-031-83347-2_10 2.1

[2] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *2007 IEEE Symposium on Computational Intelligence and Games*, 2007, pp. 252–259. 2.2

[3] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis, "Imitating human playing styles in super mario bros," *Entertainment Computing*, vol. 4, no. 2, pp. 93–104, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1875952112000183 2.2

[4] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," 2020. [Online]. Available: https://arxiv.org/abs/1809.02627 2.3.1.1

[5] Unity Technologies, "Unity editor," 2025, accessed: 2026-01-04. [Online]. Available: https://docs.unity3d.com/Manual/unity-editor.html 2.3.1.2

[6] Daggerfall Workshop, "Daggerfall unity," 2025, accessed: 2026-01-04. [Online]. Available: https://www.dfworkshop.net/ 2.3.2

[7] K. Gurney, *An introduction to neural networks*. CRC press, 2018. 2.3.3

[8] D. de Almeida Rocha and J. Cesar Duarte, "Simulating human behaviour in games using machine learning," in *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, 2019, pp. 163–172. 2.3.3, 2.3.3

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016. 2.3.4

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015. 2.3.4

[11] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009. 2.3.5

[12] S. Ross, G. J. Gordon, and J. A. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 627–635. 2.3.5

[13] Daggerfall Workshop, "Modding tutorials: World data overrides," 2019, accessed: 2026-05-18. [Online]. Available: https://forums.dfworkshop.net/viewtopic.php?t=2857 3.3.1

[14] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=Bkg6RiCqY7 3.6.6