A Multi-tiered Level of Detail System for Game AI

by

Patrick O' Halloran, B.Sc. (Hons)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

September 2016

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Patrick O' Halloran

August 30, 2016

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Patrick O' Halloran

August 30, 2016

Acknowledgments

Firstly, I would like express my thanks to Dr. Mads Haahr for his immeasurable help and guidance throughout the dissertation.

I would also like to thank the course director Dr. Michael Manzke for his help throughout my time here in Trinity.

In addition, I would like to thank my friends and family for all their support. I would like to thank Antonio Nikolov for generously proofreading this dissertation, and Giovanni Campo for providing me with feedback on the project.

Last but not least, I would like to thank my mother, without whom none of this would be possible.

Patrick O' Halloran

University of Dublin, Trinity College
September 2016

A Multi-tiered Level of Detail System for Game AI

Patrick O' Halloran
University of Dublin, Trinity College, 2016

Supervisor: Mads Haahr

This dissertation explores the idea of using simulation Level of Detail (LOD) to reduce CPU and memory use associated with persistently simulating NPCs (Non-player Characters). Current games are often CPU bound so developers have severe limits on how many AI characters can be simulated concurrently. The aim is to design, implement and evaluate a system that alleviates these computational stresses by simulating characters and events at varying levels of detail. Using Machine Learning, a model is trained that can predict the outcome of events so that they do not have to be simulated fully, allowing more agents to exist in the simulation. The system aims to attain an accurate model of the full detail simulation. This makes the simulation feel consistent to the player, increasing immersion.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The aim of this research is to create a system that can decrease the cost of simulating agents when they are off-screen, with a view to simulating more agents than would otherwise be possible. The agents are simulated persistently. The technique is expected to be suitable for open world video games, where one would hope that the agents would be believable and interesting for the player. Decreasing simulation costs for off-screen agents is called Simulation LOD, or AI LOD.

Simulation LOD is similar in concept to Graphical LOD, which is used in graphics rendering to reduce the complexity of objects that are further from the viewer. In Graphical LOD, the amount of polygons used to display the model is reduced the further away it is from the player's viewpoint. Rendering a full detail model which amounts to a small amount of pixels on the player's screen would be a huge waste. Another rendering optimisation technique is frustum culling, where models outside of the player's view frustum are not rendered. Graphical LOD has reached a stage where it is mature and the application of graphical LOD has become mainstream in game programming, being included in most game engines [7][8]. However, LOD applied to logic is still in its infancy. In commercial games, in-game characters will generally not be simulated when off-screen. While this approach keeps processing costs low, it hampers the believability of the agents having lives of their own.

This dissertation proposes to use machine learning to resolve off-screen events.

Given input parameters, it can 'skip' over the details to provide a predicted outcome as its output. This can be done in a hierarchical manner. For example, it could predict the outcome of a battle between two soldiers, or at a reduced LOD a battle between two armies. The system will run the simulation at a reduced level of detail, while attempting to produce results similar to the full simulation detail. The trade-off between resources saved by the system versus the accuracy of the system at predicting outcomes will be examined.

1.1 Motivation

The motivation for doing this research stems from the fact that simulating background characters in current Open World games is severely limited by the CPU time budget available for AI Processing. Modern games allow at most roughly 5ms for AI calculations [9]. Due to the complexity of autonomous agents only a small number of them can be simulated while adhering to this budget. If these agents could be simulated more cheaply, then we could have our game worlds populated with many complex agents which would really improve game immersion.

Simulation LOD attempts to address the problem by providing different levels of simulation detail. In the academic literature, there has been various frameworks suggested, which will be reviewed in Chapter 2.

1.2 Objectives

The overall goal of this project is to create an flexible system for simulating off-screen persistent agents cheaply without degrading the simulation quality. To achieve this, there are a few sub-objectives that need to be fulfilled:

- Create a LOD system that requires minimal new game logic code.
- Evaluate optimisation of the system:

  - Evaluate the system's effectiveness in reducing the CPU costs of simulating autonomous agents.
  - Evaluate its effectiveness in reducing the memory footprint of the agents.
- Evaluate the system's accuracy in predicting outcomes at reduced LOD levels.

1.3 Dissertation Roadmap

Chapter 2 reviews the state of the art in Simulation LOD and also reviews select Machine Learning papers. Chapter 3 overviews the design of the system from a theoretical standpoint. Chapter 4 goes in to the implementation specifics of the framework and the demo application. Chapter 5 shows results of the tests performed to evaluate the framework. Finally, Chapter 6 presents a conclusion which discusses the contribution and ideas for future work.

Chapter 2

State of the Art

This chapter reviews the state of the art in the area of Simulation Level of Detail and also looks at studies that applied machine learning to predicting battle outcomes, as this is central to the approach taken for the LOD system proposed in this dissertation.

The first section looks at various approaches towards Simulation LOD in the literature and in commercial games. The second section reviews machine learning use-cases similar to the proposed system. The chapter ends with a short summary.

2.1 Simulation LOD

Simulation LOD is concerned with what occurs when the NPCs are out of frustum [1]. The LOD system described by Brom et al. [1] can support areas without an observer having a high LOD, if they are particularly important areas. It supports gradual degradation of the simulation, as opposed to an online - offline approach. It differs from other methods in that it attempts to simplify the space as well as the behaviours.

The space is represented in a hierarchical manner, with agents using a more coarse representation of the space for path finding if the simulation LOD is lower (See Figure 2.1). There are four levels of detail and they are applied to the action selection and environment representation [10]. Brom et al. also discuss the difficulty of retaining information about areas with lower simulation LOD such that they can be reconstructed

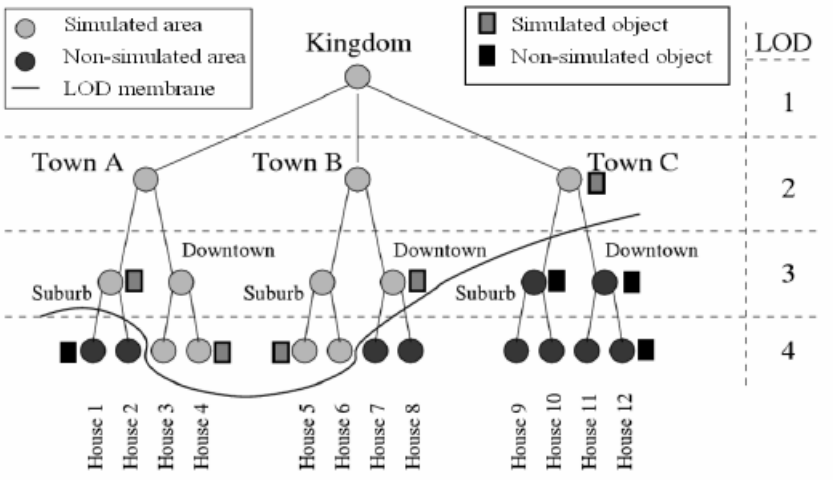


Figure 2.1: Hierarchical Representation of Places and Objects [1]. The wavy line is called the *LOD Membrane*. Nothing below the membrane is simulated.

back to full LOD, as hierarchical LOD can be likened to lossy compression. The Membrane metaphor is used to describe what is currently simulated: nothing below the membrane line is simulated in Figure 2.1.

An example scenario of a miner having a drink in a pub is shown in Figure 2.2. At each level of detail the square boxes show the agent's behaviour. If the player is far away from the pub then the miner will just be enjoying himself in the pub, with no further detail about what is happening. At *full-2* the miner is sitting at a table. At *full-3* the miner is drinking a beer. At the full detail the miner is sipping their beer. While this approach is efficient it requires a lot of work to specify the logic at different levels of detail. This dissertation aims to create a more automated approach.

The Simulation LOD system described by Wissner et al. [10] is based on both distance and visibility. There are eight levels of detail in their system. The LOD system is applied to action execution, updating movement, collision avoidance and navigation. For example, characters with lower levels of detail can walk through each other. Animations are omitted and characters only lip sync their dialogue at high levels of detail. As the detail decreases path planning is simplified. Instead of using the shortest possible route, they use any route that will get them to their target. The agent update

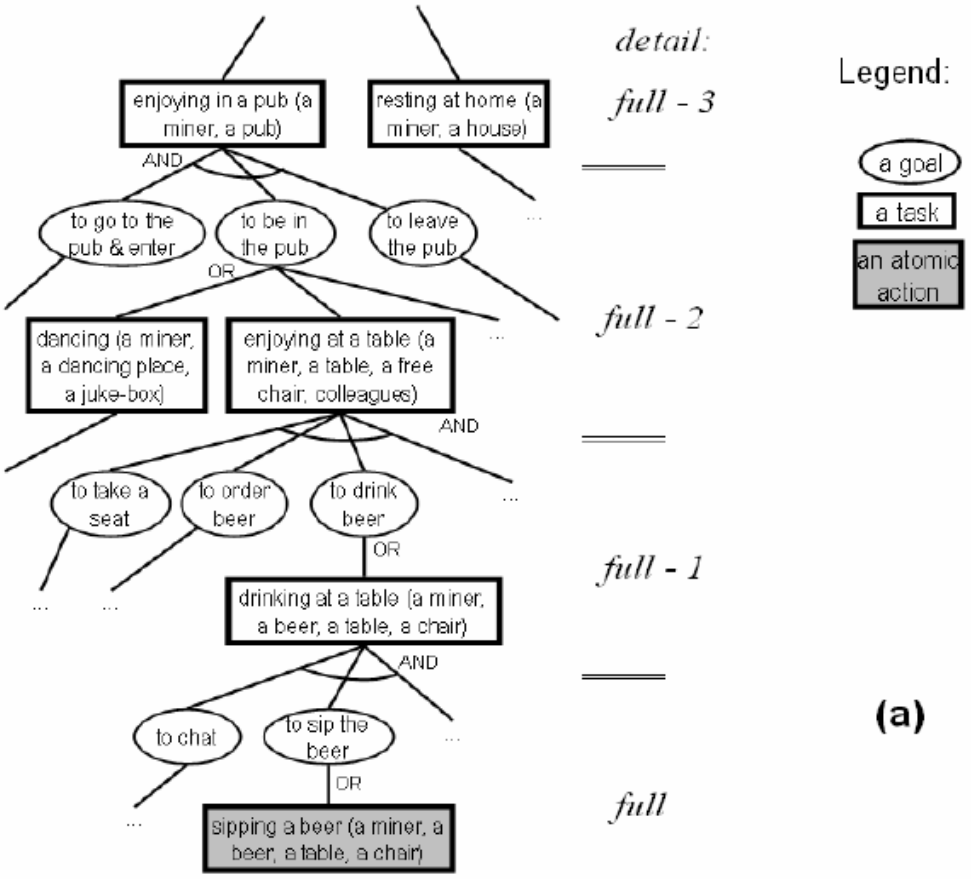


Figure 2.2: The behaviour of a miner drinking in a pub at different levels of detail. [1]

frequency is also adjusted based on their LOD level.

Chenney et al. [11] developed a traffic simulation in which the space of the environment is partitioned and a discrete event simulation is used to track when a car crosses to a new area (enters a new road or intersection). In this way, the intervening travel is not simulated. When entering a cell, a prediction is made on when it will leave the cell, based on the acceleration profile of the car and the traffic congestion. The authors define three requirements for the proxy simulation:

1. The proxy must provide a reasonable stream of view entry events.
2. The proxy must provide reasonable state for objects when they re-enter the view.
3. The proxy must be significantly cheaper to compute than the accurate simulation.

In another paper by the same authors [12] they also apply LOD to path planning. If an agent is out of view, then it only performs global path-finding and not local path-finding i.e. obstacle avoidance. They do not just ignore local interactions however, as an agent moving through a crowd would be expected to reach their destination slower than if they were unimpeded. They sample the full simulation using an Expectation Maximization (EM) algorithm to obtain a probability mixture distribution which can predict how long the delay should be when out-of-view parties interact. The goal is that the player could infer what happened off-screen. To further optimise path planning for off-screen agents, the shortest paths between every pair of vertices are computed and stored in a table.

Osborne and Dickinson [2] describe what they call Grouped Hierarchical Level of Detail. It takes a top down approach, where the group is a single entity, but groups can be composed of subgroups and the leaf nodes are individual agents. If a lower level of detail is required, the subgroups are not simulated. There are increasing levels of abstraction coinciding with decreasing levels of detail. How many nodes are expanded depends on the proximity of the player. Attributes can be maintained for nodes relative to their parents, resulting in a more persistent simulation, but requires memory for the nodes are maintained down to the leaf node. The approach is similar to Brom et al. except the abstraction is applied to agents instead of locations. The simulation is

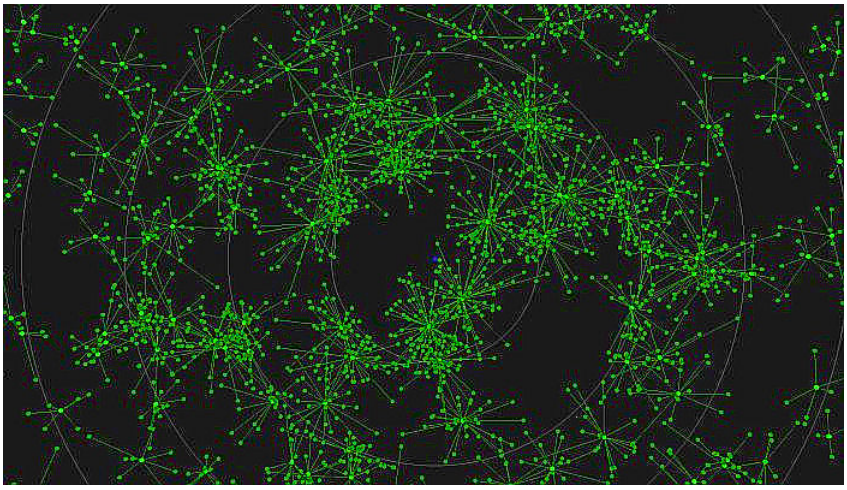


Figure 2.3: Grouped Hierarchical LOD [2]

shown in Figure 2.3. The connected points are fully simulated child nodes. Child nodes are collapsed to the parent as the groups move further from the centre of the simulation.

MacNamee and Cunningham [13] introduce the idea of Proactive Persistent Agents (PPS). To introduce the concept they contrast it to the presiding attitude in game development by quoting a code comment in the Half Life SDK: *"don't let monsters outside of the player's PVS* [Potentially Visible Set] *act up, or most of the interesting things will happen before the player gets there"*[14]. MacNamee and Cunningham go on to say that there are a range of games where this is not the case, in particular RPG games, *"in which a virtual world, populated by agents which act autonomously - irrespective of the player's location - would greatly add to the sense of immersion."*.

The system they developed had four major modules:

- Behaviour system (Reactive)
- Social system (Reactive)
- Goal based planner (Deliberative)
- Schedule

The Behaviour system is handled by a technique called Role-Passing [15] [16]. Certain roles require less computational resources than others. Every agent is a base agent object, and then can be assigned roles in the simulation, which can be added, removed and layered to change the agent's behaviour. This modular approach allows the agent to be no more complex than it needs to be, and as such can speed up AI processing. For example, if the NPC is not in view of the player, roles involved in making the agent appear visually realistic can be removed. The Schedule system is a cheap way to place agents where they should be at a given time of day according to a schedule. Games such as Skyrim [17] use a system like this to seem as though the agents are simulated off-screen. The Social system is handled by a module called μ-Sic [6] which will be discussed later.

Sunshine-Hill [18] describes a system that eschews simple distance based LOD called the LOD Trader. He argues that there is no natural threshold for AI LOD like there is for graphics LOD, and he sets out to create a better heuristic for AI LOD. The idea centres around optimal LOD selection. The LOD Trader tries to maximise realism, so a metric for realism is needed. It uses a stock trader metaphor - the trader has a set amount of resources [CPU and RAM] to spend, and must choose the stocks [LODs] that maximise return [realism].

An event in which the player notices a problem with the simulation is called a Break in Realism (BIR). The LOD trader attempts to reduce the likelihood of a BIR occurring. The LOD Trader uses logarithmic probability instead of linear probability as it makes operations such as adding and comparing different probabilities easier. The trader attempts to minimise the total probability of a BIR occurring. The Trader will not decrease detail until it becomes necessary for performance

The BIR categories are:

- **Unrealistic State** These events are immediately noticeable to the player, such as an agent walking in place or disappearing.
- **Fundamental Discontinuity** The NPCs current state is incompatible with the player's memory of the agent. These events are not unrealistic when taken out

of context, but are unrealistic because the player knows they can't be true, such as a character who is known to be dead suddenly appearing alive and well.

- **Unrealistic long-term behaviour** If the NPC is observed for an extended period, unrealistic behaviour is discovered.

Instead of using fixed transition rules, the LOD Trader tries to figure out the entire space of detail levels and finds the LOD for each agent that minimises the probability of above BIRs. It respects performance constraints and handles clustering of agents around the player gracefully. It is one of the more sophisticated LOD selection methods in the literature, however it does not describe how the actual LOD levels are implemented, which is the main focus of this dissertation.

2.1.1 Simulation/AI LOD in Commercial Games

LOD systems have been applied to commercial games, although these approaches are usually not publicly documented. Developer interviews are the main (and often only) source of information on LOD systems used in commercial games..

Assassin's Creed Unity (2014)

Assassin's Creed Unity [19] applied a LOD system [3] to simulate the large city crowds in Paris. The legacy constraints of the older Assassin's Creed games were a CPU limit of one hundred NPCs, and a limit of twenty civilians. There were no systemic crowds. For Assassin's Creed Unity they implemented three levels of detail: Autonomous, Puppet and Low Resolution. The distance threshold for each LOD can be seen in Figure 2.4. The puppet has the same appearance as autonomous but with all the components turned off. The low resolution agent uses a custom fast animation system with a much simpler animation rig. There can be 40 autonomous agents which are less than 12m from the player. There can be thousands of lower detail agents.

The low resolution and puppet agents don't use the Havok physics engine for collisions. Instead they use a 2D partition map for queries. All agents have a 'shepherd' which is placed by level designers and can be scripted. The shepherds contain all the

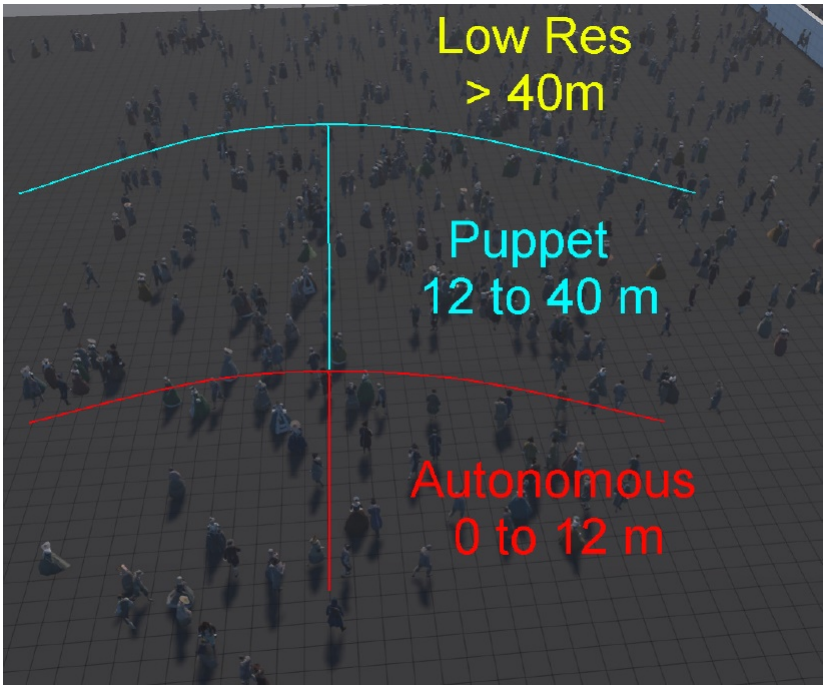


Figure 2.4: Assassin's Creed Unity LOD threshold distances [3]

AI data and control the agents. One type of shepherd is the wandering crowd, which uses randomly generated closed paths. Object pooling is used for the meshes. Object Pooling is an optimisation pattern in which a set amount of objects are loaded into memory once and are then re-used throughout the lifetime of the program, as opposed to allocating and freeing them continuously [20]. When an agent needs their LOD changed, the best matching entity in the object pool is selected, and props such as hats are spawned.

S.T.A.L.K.E.R.: Shadow of Chernobyl (2007)

S.T.A.L.K.E.R. [21] is a first person shooter survival horror game developed by GSC Game World. The game is built in GSC Game World's proprietary game engine, X-ray. A key feature of S.T.A.L.K.E.R. is the A-Life (Artificial Life) system, which is to date one of the best examples of A-Life in a commercial game. In an A-Life system the NPCs continue to exist even outside the player's field of view. Generally in open world games NPCs will be de-spawned (deleted) when the player leaves the area, to

save processor resources. The AI developers of S.T.A.L.K.E.R. decided to continuously simulate the NPCs, albeit at a lower level of detail.

In S.T.A.L.K.E.R. the NPCs have two levels of detail: Online and Offline [22]. They have very different behaviour implementations. When in offline mode, the NPC will not play animations or sounds, will not manage their inventory actively and does not use detailed, smoothed paths. It builds coarse paths, on the global navigation graph. The online mode has the full level of detail.

S.T.A.L.K.E.R. has a few separately instanced areas. All NPCs in other areas are offline. Some NPCs in the player's current area are also offline, as the areas are quite large and contain a large population of NPCs. When the NPC crosses a predefined radius from the player, it switches to full detail. The radius is typically 150m but varies in different areas.

Each area has its own detailed navigation graph, which is used by online characters. Each area's detailed graph has a less detailed version, which is connected up with the less detailed graphs from all the other areas. This global graph is used by offline characters. When switching areas, an online NPC uses the detailed graph to calculate a path to the nearest node in the global graph. The agent then uses the global graph to move to the next area. The area transition points for NPCs are further than they are for the player, somewhere "around the corner" so that they do not just vanish when transferring to the next area.

The goal that all the NPC characters shared is to uncover the mystery of the 'Zone', the world of the game. In offline mode, the NPC tries to find a goal. If it can't find one it wanders aimlessly. When it has a goal, it moves to the goal location using the global graph. While on the mission, the NPC could encounter other characters and fight or trade, run away from, or ignore each other. To resolve encounters in offline mode, a turn based system was used. It consisted of a formula and a random factor. The outcome of the events would be in the news for the player to read, if another character was there to witness them.

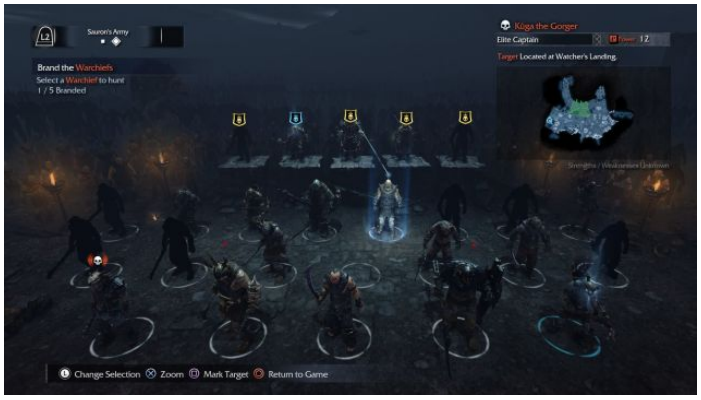


Figure 2.5: The Nemesis System in Shadow of Mordor [4]. The Orc captains are displayed according to their place in the Uruk faction hierarchy.

Shadow of Mordor (2014)

The Nemesis System in Middle Earth Shadow of Mordor [4] is a recent example of a game trying to make more interesting NPCs. The Orc captains in the game are shown in a menu (See Figure 2.5). If you have not met them before they are just a silhouette. The player can discover an Orc, revealing one of the silhouettes, by encountering that Orc in the wild, or by interrogating a regular Orc.

While the Orc characters are not continuously simulated, their names and traits are persistent. When the player goes to their location on the map the Orc is spawned. The Orcs have a procedurally generated name, appearance and personality.

The version of the Nemesis System seen in the released version of the game was significantly scaled back from the original goals [23]:

> For example at one point we had multiple Uruk Factions with separate bars for Morale and Discipline, each Captain influenced these Bars and their state determined the behavior of the Orcs in combat as well as emergent missions. At this point, their Hierarchy UI looked somewhat like a Christmas tree.

However, the Nemesis system was very well received by critics and fans, which shows that there is a strong desire for more interesting NPCs and emergent gameplay.

2.2 Machine Learning

2.2.1 Combat Outcome Prediction

There has been work done recently in trying to predict the outcome in MOBA (Multiplayer Online Battle Arena) and RTS (Real-time Strategy) games. StarCraft (1998) in particular is popular in AI research due to the Brood War API (BWAPI) [24] which allows the programming of custom StarCraft AIs. The research has been mainly to develop strategies to assist either human or AI players. Two common modelling goals are to find the best way to spend resources, or the best way to use individual units to win combats [5].

Yang et al. [5] present a data-driven approach for identifying patterns in combat that lead to success in MOBA games - specifically Dota 2 (2013). Dota 2 is a 5v5 MOBA game. Yang et al. use game logs from pro game competitions to build their model. They model combat as a sequence of graphs that represent the ten players' interactions at different times. The interactions modelled are dealing damage and healing. There are eleven nodes in a graph (See Figure 2.6), each node in the graph being a character role for each team, plus an extra one for death. A unidirectional edge is made between two agents if one is attacking or healing the other. An edge is drawn to the death node if the agent died in that stage of the game.

There are a number of graph metrics that are recorded, such as how many in and out connections a node has. These are used as features to train a decision tree. Different features are tried until a tree with a good classification accuracy can be trained. Combat rules are then generated from the tree, simply by tracing a path from the root node to the leaf nodes. However these rules are written in terms of the graph metrics used to train the tree. They are difficult to relate to real in-game actions. Therefore they use the combat rules to find subgraphs which show good combat tactics. This study shows that it is possible to predict in-game outcomes using machine learning, and also that it is possible to extract useful information about the game in the process

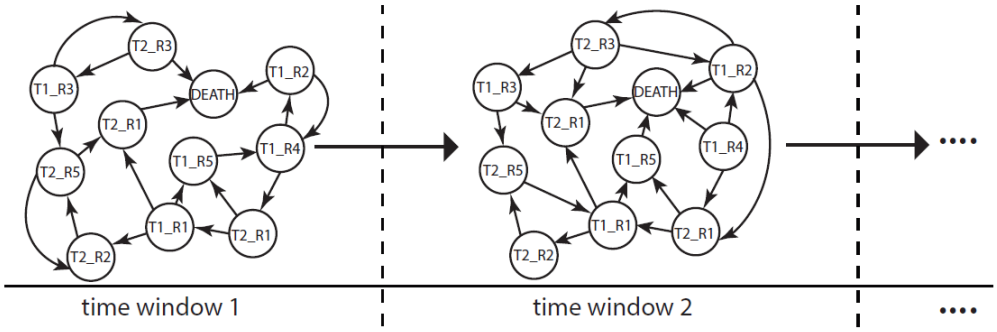


Figure 2.6: A sequence of graphs describing a game in Dota 2. The game stages are around 9 minutes apart. Tx_Ty = Team X Role Y. [5]

of doing so.

Sánchez-Ruiz [25] uses machine learning techniques to predict small battles in StarCraft. Being able to predict the outcome of a battle would allow an AI to adapt their strategy. The paper uses Linear and Quadratic Discriminate Analysis, Support Vector Machines, and k-Nearest Neighbours. The scenario used was a battle between two groups of 6 units. The author collected data from 200 games. Samples were taken six times per second. At the end of a game, all the samples were labelled either one or zero for win or lose. The features recorded were the number of units, their total amount of life and the area of the minimum axis aligned rectangle that could enclose the agents, for both teams. A seventh feature was the distance between the two teams rectangles. The average game length was $\sim$ 19 seconds.

The Support Vector Machine had a global accuracy of 76.27% while the KNN and KKNN (Weighted K-Nearest) algorithms had a very high global accuracy of 84.3% and 95.7% respectively. The accuracy of each model increases as the game evolves, but the nearest neighbours can predict who will win very early in the game, before the agents even begin fighting. The author does mention a potential reason for the high accuracy of the case based approaches (KNN and KKNN) is that the dataset is biased due to the AI playing against itself and not using enough variety of attack strategies. The author planned future work is to test with human players and also with more complex

battles, and when predicting to consider the evolution of the game rather than just a snapshot.

2.2.2 The μ-SIC System

An Artificial Neural Network (ANN) can be used to resolve character interactions and events. The μ-Sic system [6] uses an ANN (See Figure 2.7) that has been trained to take a set of values, and determines the action that a character should take. The inputs are the personality of the agent, the agent's relationship with the other agent, and the agent's mood. The ANN was trained with 300 hand-crafted example scenarios (as there were no databases for how people interact), and then used this to solve for the general space of possibilities in real-time. The output layer contains all the possible actions the character can take. The largest output is the interaction that is used.

This model was a component in the PPA architecture [13] mentioned earlier in this chapter. Neural Networks are very fast once they've been trained. This allows the μ-Sic system to generate believable interactions with minimal runtime cost. This cheap way to resolve encounters was a major influence on the direction of this dissertation. A big advantage of using an ANN for the LOD system described in this dissertation is that there is plentiful data available to train with since the samples are taken from the game itself.

2.2.3 Black and White (2001)

One of the most famous examples of Machine Learning being applied in a commercial video game is Black and White [26], a god game by Lionhead Studios. In the game, there is a creature that the player owns, and it responds to feedback from the player. The player is able to slap or pet the creature. The creature has a decision tree which attempts to predict the player's feedback to potential actions by the creature. Whenever the player gives feedback, the decision tree is updated. In this way the player is able to teach the creature how to behave.

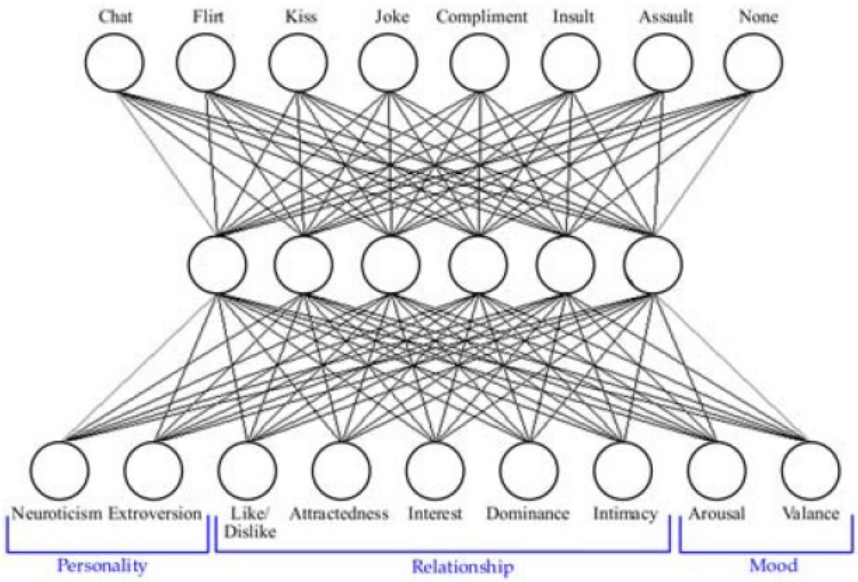


Figure 2.7: The ANN used by μ-SIC [6]

2.3 Summary

This chapter reviewed the AI/Simulation LOD approaches that have been taken in academia and in the games industry. Unlike graphical LOD no single method has emerged, likely because there is no obvious method, and it can be very domain-specific. The purpose of this review was to inform the design of the system put forth in this dissertation.

Papers about Machine Learning applied to the prediction of encounter outcomes were also examined. Firstly papers predicting combat outcomes were discussed, and then the μ-Sic system which 'predicts' an appropriate character reaction given a situation. The papers discussed show that games are a great test bed for machine learning. Due to the ease of data creation in a virtual environment, very accurate models can be built.

Chapter 3

Design

3.1 Introduction

This chapter describes the architecture of the proposed Simulation Level-of-Detail system. The aim of the system is to simulate out-of-view agents cheaply to reduce processor and memory load. This would allow for a much greater amount of agents in the game or simulation.

In Chapter 2 existing simulation LOD systems were discussed. A drawback of these systems is that they require the developer to program different logic for every level of detail. Because the logic is for out-of-view agents, the moment to moment details of what happens are not important. In this way, big savings can be made. However, there is a development cost in programming different logic for each detail level, and the LOD system may not be easily transferable between projects.

In STALKER [22] the offline LOD uses a turn-based system instead of the full combat logic. The player will never see this turn-based system play out, so it doesn't matter how it works. The only concerns for the developer are that it is cheap and provides a interesting, dynamic and believable experience for the player.

The box in Figure 3.1 contains the logic that maps input states to output states in the reduced level of detail. Since it doesn't matter how the box works, it can be

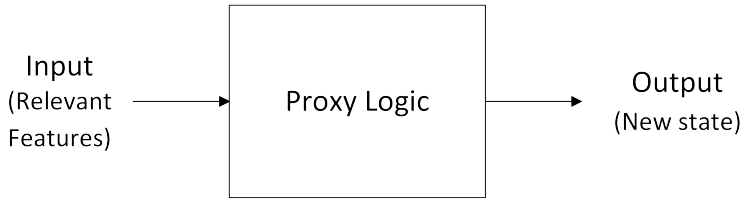


Figure 3.1: Off-screen logic

thought of as a classifier. A classifier takes inputs which can be any variable (feature) which is important in deciding the outcome, and outputs a class or category. If it outputs a continuous value it is called a regression model [1]. When developers create logic for off-screen agents they are actually hand-crafting classifiers, or regression models if the response variable is continuous.

Supervised machine learning is a good fit for this kind of problem. A complex function can be approximated by the machine learning algorithm, and its accuracy at predicting or classifying new instances can be measured. The machine learning algorithm is trained on sample data from the full simulation. When an agent is out-of-view, instead of simulating events fully, the trained model is then used to predict the outcome of the event. This approach could equally be applied to any simulation that could benefit from cheaper proxy simulations.

There are several advantages of this approach. Querying the model should be a cheap operation. The speed of predictions will be tested in Chapter 5. The models are expected to be fast, because the main computation takes place offline during development. Another advantage is the developer only needs to explicitly create one level of detail - the full LOD - so the development resources can be focused on that. The framework described here can be used for the creation of reduced levels of detail between projects, and once the developer or designer becomes familiar with the framework they can easily add simulation LOD to their game.

Using machine learning to create the LOD levels also allows the accuracy of the system to be evaluated, by the percentage of correct classifications or the average error

[1]Regression is the process of estimating the relationship between variables

when predicting continuous variables. If a high enough accuracy is achieved then the proxy simulation outcomes could be virtually indistinguishable from the full simulation. This would go a long way to creating living, breathing worlds if the system can support thousands of agents.

Finally, a virtual environment gives the large benefit of easy generation of data for the learning. Building a dataset for training the model can be done by simply collecting game samples.

The following section will describe the architecture of the system.

3.2 Architecture Overview

3.2.1 LOD system

There can be n levels of detail in the system, *Full, Full-1, Full-2 ... Full-n*, following the naming convention in [1]. Below is an example of what this would look like in a war game set in Ancient Rome:

- **Full** The agents are simulated as individual soldiers using the full simulation with no LOD reductions. All their components such as animation and audio are enabled.
- **Full-1** At the first reduced level of detail, agents are still simulated as individuals, but they now use the prediction system and most of their components are turned off. Only the components needed to move the agent and detect encounters are retained.
- **Full-2** At the second reduced level of detail, agents are no longer simulated as individuals. They now form small groups of 8 soldiers called a tent party. They move as one unit so they only need one movement controller. The same is true for all reduced LODs. The party encounter sphere is larger than a single agent. The encounter sphere is the 'sphere of influence' of an entity.

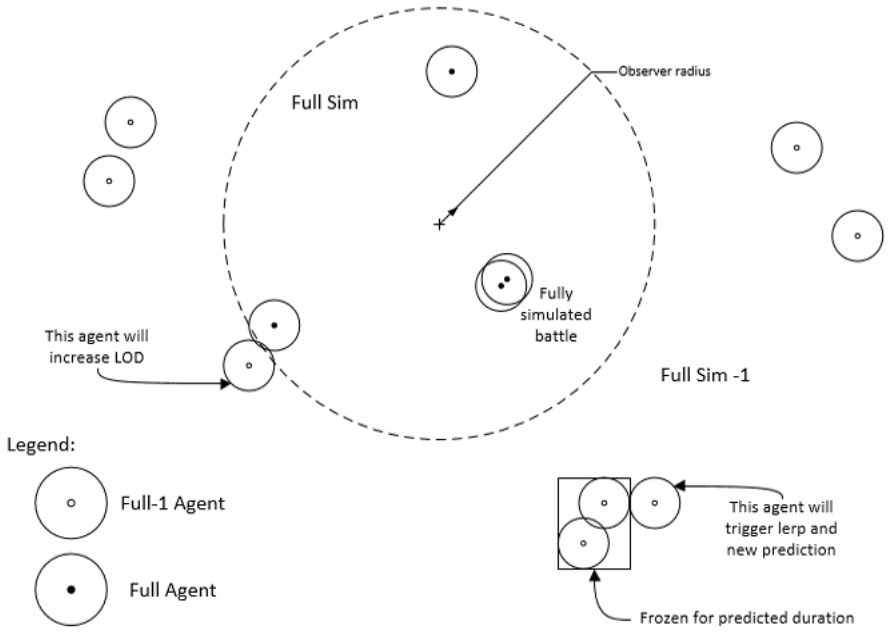


Figure 3.2: The Encounter System for two levels of detail

- **Full-3** At the third reduced LOD, 10 tent parties become a century. The encounter sphere grows larger.
- **Full-4** At the fourth reduced LOD, 6 centuries become a cohort. The encounter sphere grows larger.
- **Full-5** At the lowest level of detail, 10 cohorts become a legion, with a very large encounter sphere. Legions fight with other legions.

This would be implemented similar to the Grouped Hierarchical Level-of-detail system mentioned in chapter 2. When at a reduced level of detail, agents (or their abstractions) will still move around the world. Most of their other components will be disabled or removed when the LOD is reduced. In the Full LOD, agent sensing is used to initiate an interaction with another agent or object in the world. At the lower levels a different system is used called the encounter system.

The encounter system (See Figure 3.2) tries to answer one question: what will happen to the entity x when it encounters an entity y? It asks this question when

something of interest (y) comes within the encounter radius of the agent (x). This radius around the agent is called the Encounter Sphere. It can be viewed as a simplification of the more complex sensing typical in video games. The encounter sphere can increase in size at reduced levels of detail, as a legion would be expected to have a larger sphere of influence than a party or an individual agent.

When the encounter spheres of entities overlap, instead of targeting each other and commencing combat as they would in the Full LOD, they consult the model for a prediction. Each agent in the encounter is then frozen (their navigation and movement is turned off) for the predicted duration of the encounter. A function that will apply the predicted state is queued. When the duration time has passed, their queued prediction is applied and their movement is unfrozen if they survived the encounter. If they died they can be removed from the game and replaced by a model in a death pose.

If another unit stumbles upon the frozen units, then a new encounter is triggered. The agents that were frozen will have their stats (such as health points) interpolated based on the previous prediction to reflect the time that has passed since that prediction was made, and the queued function to apply that prediction will be cancelled. The agents are then frozen again along with the new agent, and a new prediction is made. In the case of two forces at different detail levels meeting, the force with the lower detail level will increase their detail level. The procedure can be seen in Algorithm 1.

```
begin
    highestDetail = n;
    for agent ∈ agentsInEncounter do
        if agent has apply prediction queued then
            cancel previous apply prediction;
            interpolate;
        end
        if highestDetail < LOD of agent then
            highestDetail = LOD of agent;
        end
    end
    for agent ∈ agentsInEncounter do
        SetLOD(highestDetail);
        get feature vector for agent;
        make prediction;
        queue apply prediction;
        freeze for predicted duration;
    end
end
```

Algorithm 1: Encounter resolution

Predictions are also interpolated if a player comes close enough that the agents in the encounter get moved up to full LOD. Since the predictions have already been made the system should be quick enough that a player could teleport in to the location of the battle and it would seamlessly appear as if the agents were mid battle.

LOD selection can be done in a variety of ways as seen in Chapter 2. The framework should work with any LOD selection scheme. Figure 3.2 shows how the observer works for two levels of detail using a simple distance-based heuristic for LOD selection.

Switching up and down LODs is another issue. How will the Legion become 5400 soldiers and vice versa? Maintaining state for each agent would use a lot of mem-

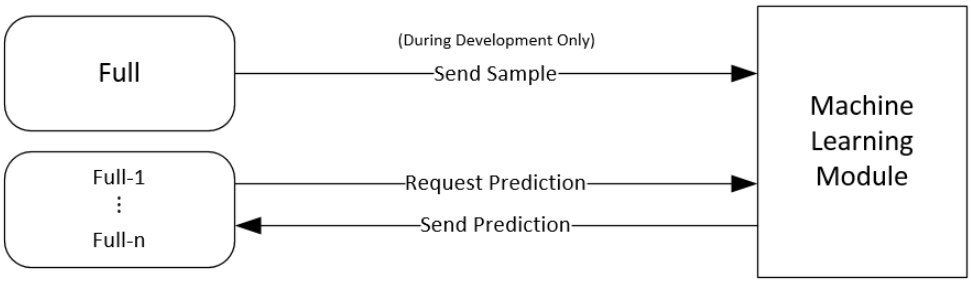


Figure 3.3: How the LOD levels interact with the Machine Learning Module

ory so distributions could instead be stored and then the individual agents could be respawned according to those distributions. It is at the discretion of the developer how much state for individual agents should be preserved when reducing LOD.

The next section will describe how the models are trained for each level of detail.

3.2.2 Machine Learning

Sampling

To train the prediction models, a sampling strategy is needed. Continuous time slices were considered, but was decided against as it would require the reduced LOD entities to continuously query the model at the chosen timestep to update their state. Also, in many samples nothing would happen creating a heavily biased set which can cause problems for machine learning algorithms.

Instead, sampling *events* of various duration was chosen. In this sampling strategy, samples are only taken when an encounter occurs. An encounter takes place when two or more agent's encounter spheres are overlapping. The encounter spheres can be implemented as sphere colliders in the physics engine.

Samples are taken for each level of detail. All sampling takes place in the full simulation as that is what the framework is trying to approximate (See Figure 3.3). Using the Roman Legion example, if two Legions are fighting in the full simulation,

then every agent in both Legions is fully simulated. In a sophisticated simulation, the agents will use battle tactics and act according to the chain of command. During development of the game, each group - Legion, Century, Cohort, Party, Solider - will have their own encounter sphere enabled for sampling.

Whenever the encounter spheres overlap a sample will begin for each entity in the encounter. Each entity has a sampler component which records a *feature vector*. The *feature vector* contains the features of the encounter that are necessary for the model to make a prediction on the outcome of the event. It is a 1 * n row vector, where n is the number of features. Whenever the encounter ends - the agents won, died, or fled the sphere of influence of the other team - then the details are recorded and the sample ends. Separate datasets are stored for each level-of-detail model being trained.

Feature Selection

Selecting the right features is important. If the feature space does not contain enough structure for the model to learn, it will not be able to produce good quality predictions.

The feature vector is a 1 * n row vector:

$$F = \begin{bmatrix} f1 & f2 & \dots & fn \end{bmatrix} \tag{3.1}$$

The features for each LOD will be different. For example, the chances of survival for an agent who is simulated as an individual will be less affected by high level tactics such as formations, than a legion's survival would. At the legion LOD in which battle durations could be very long between evenly matched legions, the high level tactics and troop formations could be essential in predicting victory and would need to be considered for the feature vector.

Target Variables

The sampler also takes care of recording the target variables so that the feature vector(s) can be labelled. For example the health loss incurred by the agent in the battle

could be recorded if the target variable to be learned is the health loss incurred.

If a new combatant enters the encounter sphere of an existing combatant in the encounter, then a new sample is started, as a new combatant invalidates the previous sample, due to the feature vector of that sample not accounting for the new entrant.

When the sample ends, the target variables (i.e. labels) are concatenated onto the feature vector. This row vector contains the features and output variables for that particular sample. The full sample vector is:

$$S = \begin{bmatrix} F & Tgt \end{bmatrix} \tag{3.2}$$

where S is a sample and Tgt is a row vector of target variables.

Preparing the data

The samples are stored in a dictionary of matrices. Each entry in the dictionary contains a dataset (many rows of samples). This way it is possible to store multiple datasets.

$$Dataset = \begin{bmatrix} S_1 \\ \vdots \\ S_n \end{bmatrix} \tag{3.3}$$

The samples are serialised so that samples can be taken over multiple play sessions. Being able to merge datasets is also useful if sampling on multiple machines.

Training multiple models at once is required for predicting multiple target variables, unless a Neural Network is used. This is due to the other algorithms only being able to predict one target variable.

Learning

The machine learning algorithms can then use the datasets to learn prediction models. These models will then be queried by the proxy LODs to resolve encounters.

Ideally a machine learning library that implements several different machine learning techniques should be used. Different algorithms have different characteristics and capabilities, which are reviewed in Chapters 4 and 5. Different problems will favour different algorithms. If none of the algorithms perform well, then more work is needed on the feature engineering [2] step, as there is not enough structure in the feature space for the algorithms to learn.

3.3 Summary

The chapter gave a high level overview of the idea behind this dissertation. With the proposed framework, a proxy simulation with multiple levels of detail corresponding to different agent abstractions can be created. In order for the proxy simulation to be built, the game-specific things that need to be done are:

- Features need to selected for each level of detail.
- Samples need to be collected by running the game or simulation.
- Encounter spheres need to configured for each agent abstraction.
- LOD switching needs to be implemented.
- The user of the framework will need to evaluate which model is performing the best (although this could be automatic).

The next chapter will discuss the implementation of the proposed architecture.

[2] Feature engineering is the task of creating features for the machine learning algorithm to use. It requires intuition of the domain to decide what features are important to the outcome.

Chapter 4

Implementation

4.1 Platform Selection

Various game engines were considered to implement the framework. It was necessary to get a simulation up and running quickly so that work could begin on the LOD system.

Firstly, game engines for open-world games were considered. Of these, OpenMW [**?**], a re-implementation of the Elder Scrolls III: Morrowind (2002), and OpenXRay [**?**], an unofficial open source version of the engine used to develop S.T.A.L.K.E.R. were of the most interest. However, the problem with these engines is they were designed for one game and are quite inflexible. Previous IET students have grappled with this problem using the Elder Scrolls V: Skyrim modding tools [27][28]. There would be a high learning curve, and it would be likely to run in to some unforeseen limitations using these engines.

For this reason Unity3d was used, as it is very flexible and ideal for prototyping. There are a number of AI assets on the Unity Asset Store that could be used to build a simulation. It was decided that a simulation with combat would be the best way to demonstrate the LOD system, so an asset called Modular RPG Combat Kit[29] was used. This asset was chosen because:

- Full C# source is provided.
- It contains different character classes - Melee, Ranged, Magic.

Figure 4.1: Demo scene from Modular RPG Combat Kit

- It has a levelling system.
- It comes with some example scenes showing faction combat such as that seen in Figure 4.1.
- It integrates with Unity's navigation and animation systems.
- It is modular and quite simple. There is no superfluous code.

Originally the plan was to implement a decision tree algorithm like the ID3 (Iterative Dichotomiser 3)[30] algorithm. The ID3 algorithm is a classification tree, meaning that it can only predict outcomes that fall into discrete classes. This would only be useful for a small subset of problems in a game or simulation. CART (Classification and Regression Tree)[31] supports both classification and regression. The OpenCV [32] library's Machine Learning module implements the CART algorithm. Therefore, Emgu 2.4.9 [33], a C# .NET wrapper for OpenCV, was used. Wagner [34] provides a good introductory article to using machine learning in OpenCV. Using OpenCV came with the benefit of having a suite of machine learning algorithms to test with. In Chapter 5, there is an analysis of which machine learning algorithms performed best for the LOD system.

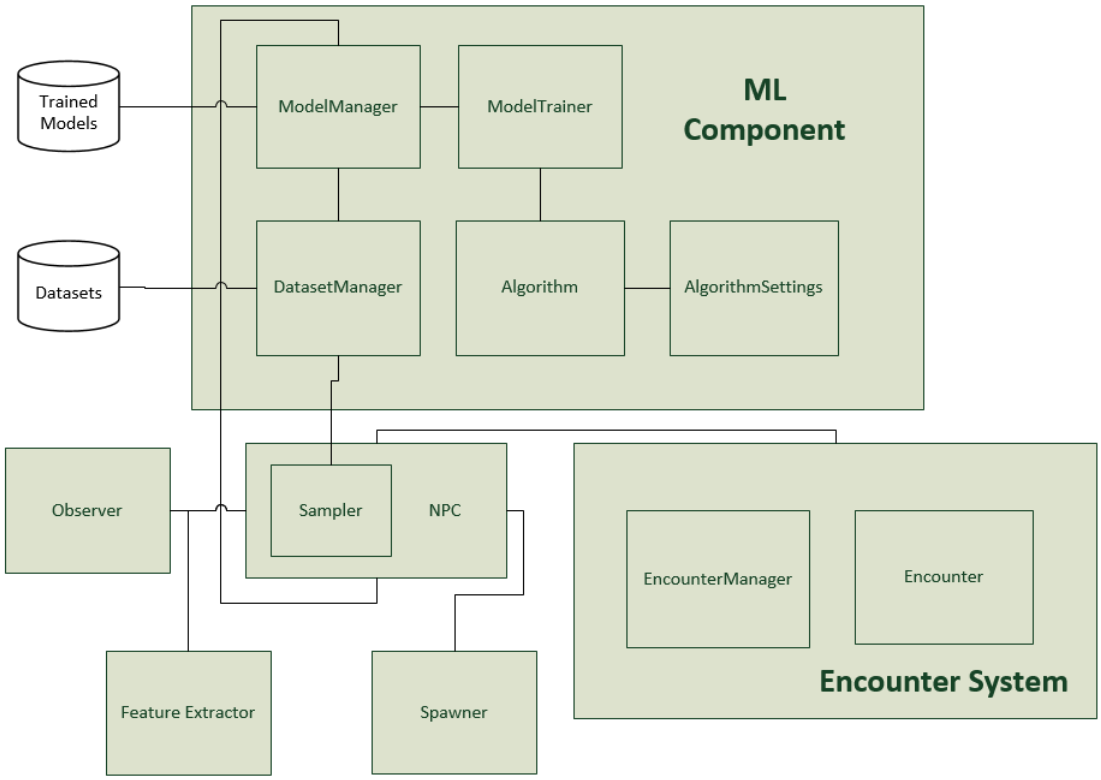


Figure 4.2: System Architecture

The models trained by the system can be used to predict individual behaviour, group behaviour or even army or civilisation behaviour. Due to the relative lack of complexity in the agents used to demonstrate the system, there are only two levels of detail: the *full* level of detail and the *full-1* level, where agents are still individuals but are using the trained model to predict encounter outcomes. The levels of detail lower than *full-1* would use more abstract agent representations.

4.2 Implementing the System

4.2.1 Overview

The full implementation architecture can be seen in Figure 4.2. The following sections discuss the components of the framework.

4.2.2 Spawning the agents

To automate sampling, the spawner is able to continuously spawn agents, deleting the old ones once a faction has emerged victorious, or an inactivity timer has exceeded the time limit. The spawner component allows the user to configure the amount of agents which can spawn, the population density of the agents and the max level an agent can spawn at. When spawning an agent they are given a random character class (Fighter, Gunner, Mage or Archer) based on probability frequencies also decided by the user. They are given a random level, which decides all their other stats such as HP and how large the agent is visually.

The agent is placed on the terrain according to the user-specified population density. This option was added to see how the system scaled when NPCs are clustered together. The terrain used is $4km^2$ or $4,000,000m^2$. This can accommodate 13333 agents at the default density used which is $300m^2$ per agent. The spawner calls `SetLOD()` on the new NPC, which performs the rest of the initialisation. In the demo there are two factions. Agents freely roam about the terrain, using a navigation graph, when they are not in an encounter.

4.2.3 Building the Feature Vector

The static `FeatureExtractor` class contains the function for building the feature vector, `GetFeatureVector()`, and a dictionary of feature vector presets. A feature vector preset is a list of `Features`, which is an enumerated type. It is passed to the `GetFeatureVector()` to specify which features are to be used to build the vector. An example preset can be seen in Listing 4.1. This allows a lot of flexibility and fast iteration when choosing which features to extract. The `BuildFeatureVector()` function

```
featureVectorPresets["fvectorv1"]
    = new List<Feature>()
    {
        Feature.characterClassLegacy,
        Feature.unitLevel,
        Feature.unitHp,
        Feature.alliesNumberOfAgents,
        Feature.alliesAverageHealth,
        Feature.alliesAverageLevel,
        Feature.enemiesNumberOfAgents,
        Feature.enemiesAverageHealth,
        Feature.enemiesAverageLevel
    };
```

Listing 4.1: Feature Preset

will build the vector with the same ordering of variables specified in the list, which is vital for the machine learning algorithm. The `BuildFeatureVector()` function takes in the `NPC`, the `featuresToUse` list and a reference to the vector it will fill. The function is called at the beginning of an encounter.

4.2.4 Encounter System

Each NPC uses a SphereCollider trigger, which acts as the Encounter Sphere discussed in Chapter 3. The encounter spheres of an agent from the red faction and an agent from the blue faction are visualised in Figure 4.3. A callback function is called whenever two agents' SphereColliders overlap, which registers the trigger event with the `EncounterManager`. When the event is registered, the `EncounterManager` checks if either of the agents are already in an encounter. If they are, then the combatants in that encounter are added to the combatants list for the new encounter, and their old encounter is removed from the `EncounterManager`'s list of active encounters.

A new encounter is created with the list of combatants. The `Encounter` Constructor performs a number of operations to get things set up. Firstly, if any of the combatants are fully simulated, the new encounter is set to full LOD and `SetLOD(LOD.Full)` is called on each agent in the encounter. Otherwise all of the agents are simulated at the Proxy LOD. In this way, an encounter cannot contain agents at different LOD levels.

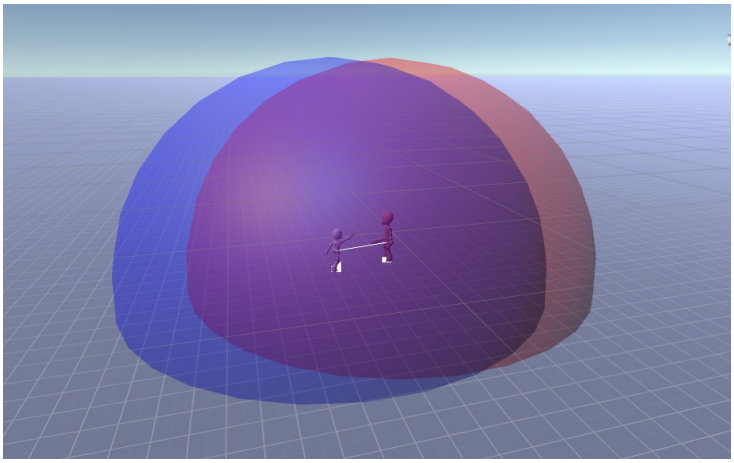


Figure 4.3: Visualising the encounter between two agents.

Next, details are computed for each faction. These are needed to build the feature vectors, which are used for prediction (and sampling during development). When building a feature vector for an individual agent it can just look up the faction details to avoid redundant computation.

The `BuildFeatureVector()` function is then called for each agent. If the NPC LOD is full and the `DatasetManager` is sampling then `BeginSample()` is also called here. The fully simulated agent will continue on to fight the encounter, until it wins or loses. It will subsequently call `EndSample()` to record the target variables in order to label the feature vector and save it to the dataset. Sampling is performed by the sampler script which is attached to the NPC GameObject.

Unity3d uses the composition programming paradigm. Game objects are made of components, which are classes inheriting from `MonoBehaviour`. `MonoBehaviours` have callbacks such as `Start()` and `Update()` which are called by the Unity Game Engine. The `Sampler` class was created as a component that attaches to the NPC so that it could be easily removed for the production game.

The feature vector is then passed to the trained model to receive a prediction for the agent. If the NPC LOD is Proxy, the agent's movement controller and navigation are turned off. `MonoBehaviour.Invoke()` is called, which takes the name of the

function to be invoked and the time in seconds that should pass before it is called. The function invoked depends on whether the agent is predicted to win or lose. If the agent is predicted to win, then `InvokeLive()` is used, otherwise `InvokeDie()` is used. `InvokeDie()` simply kills off the agent. `InvokeLive()` sets the agent's health to be the agent's current health minus the predicted `hpLoss`. It also adds the predicted `xpGain` to the agent's xp and then performs the appropriate amount of level up operations. Finally, it re-enables the agent's movement. Using the Invoke functions along with the predicted duration gives the illusion that the agent really fought for that length of time, and then continued moving.

However if a new encounter begins for that agent, those invocations are cancelled. This happens in two scenarios:

- A new agent enters the Encounter Sphere of any agent in the encounter.
- The agent becomes observed and set to full detail.

When an agent's invocation is cancelled, the `InterpolatePrediction()` function is called which interpolates the agent's health to apply the prediction partially (See Listing 4.2).

```
statController.Health = (int)Mathf.Max(Mathf.Lerp(statController.
    Health, statController.Health - prediction.hpLoss, prediction.T)
    , 1);
```

Listing 4.2: The agent's health is interpolated on the cancellation of an invocation, where prediction.T is given by $(t - t0)/predictedDuration$.

As mentioned, an encounter can only contain agents of the same LOD type. Every agent in an encounter will be switched up to Full LOD if any agent is observed i.e. at full detail. If the amount of agents observed falls to 0, then the agents will all switch to Proxy LOD and the `Encounter` will be restarted so that predictions can be obtained and new invocations made.

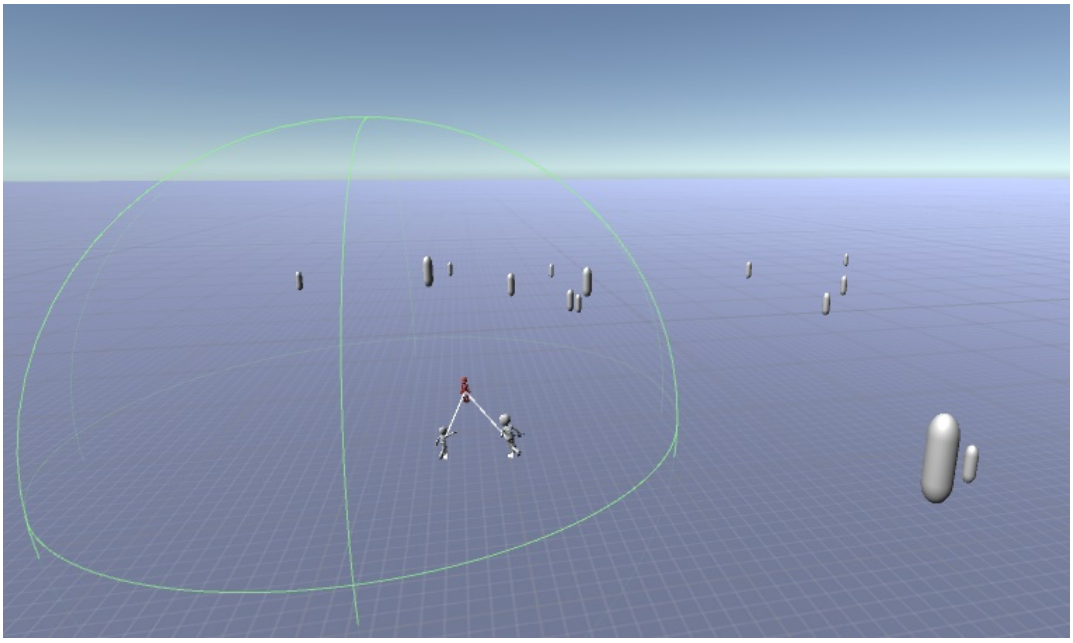
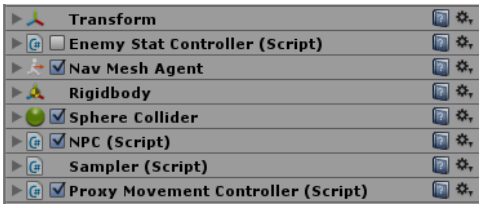


Figure 4.4: Observer radius visualised as a wireframe sphere.

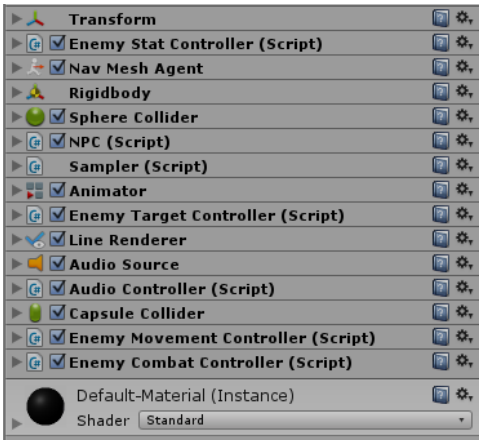
4.2.5 LOD Switching

The observer component is attached to the camera. An agent is observed if it is within a distance threshold of the observer. The observer simply loops through the NPCs doing distance checks to keep the NPC's observed property up to date. If the observed property changes then the agent's LOD is updated by calling `npc.SetLOD()`. To avoid needless switching along the perimeter, the radius to become unobserved is slightly larger. Figure 4.4 shows the observer radius and agents at both levels of detail. Proxy agents are visualised as capsules, however normally these agents would not be visible.

NPCs at different levels of detail use different components. Some components are in both Full and Proxy detail but use different settings. An example of this is the NavMeshAgent which does not do collision avoidance at the Proxy LOD. The components used by each LOD can be seen in Figure 4.5. An important consideration when deciding how the LOD switching would work, was how the NPC GameObject is updated. At first, having different Unity prefab objects for each LOD was considered. This would involve destroying the NPC and creating a new NPC using a prefab and assigning it values from the old NPC. The second option was to keep the NPC object



(a) NPC at Proxy LOD



(b) NPC at Full LOD

Figure 4.5: NPC GameObject Components. The Sampler component would not be in the shipped game.

and instead add and destroy components. The second option was selected because it much more flexible. The `SetLOD()` function contains all the necessary code for moving an agent up and down in detail. This more procedural approach allows the composition of components to be changed with a couple of lines of code rather than having to create or update a prefab in the Unity Editor. The `SetLOD` function can easily be updated to provide several levels of detail.

4.2.6 Machine Learning

The `ModelManager` class is used to configure which models are to be trained. Trained models are stored in XML files. The user can then decide which one will be loaded in for prediction in the Proxy LOD. The `ModelManager` has a dictionary of trained models which is useful if multiple models are required by the LOD system. For convenience models can be trained using just parts of the samples, which is useful when feature engineering to try different combinations of data from the same sample. It is also possible to merge datasets using the `AppendDataset()` function, provided they have the same row size. Datasets are also stored in XML files.

The ModelTrainer class prepares the data for the algorithms by splitting the data set

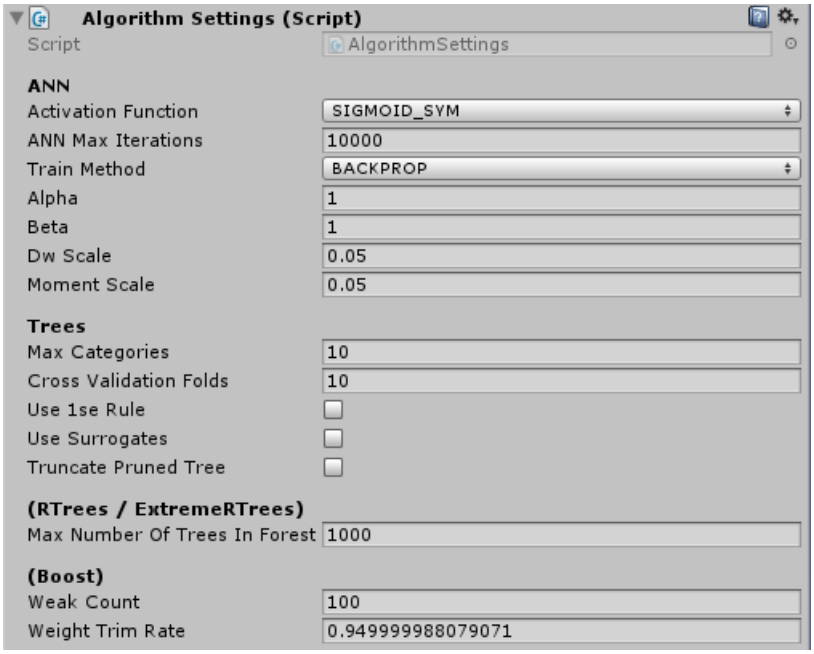


Figure 4.6: Algorithm Settings Component

up in to four matrices, `trainingData`, `trainingClasses`, `testData` and `testClasses`. These matrices are then passed to the various algorithms in the `Algorithm` class, which are discussed below. The `AlgorithmSetting` class allows the user to tune the algorithms in the Unity Inspector (See Figure 4.6).

Pre-computation is common in game engines. For example, Unity3d can pre-compute lighting and navigation meshes. These are non-blocking background processes in the editor. To make this system work in a similar fashion, the model training takes place on a separate thread using the C# .NET multi-threading functionality [35] [36]. This allows the user to use the editor freely while the pre-computation takes place.

Artificial Neural Network

An Artificial Neural Network, or ANN, is a model based on biological neural networks. An ANN is the only algorithm in OpenCV used that allows for multiple target variables, meaning it is possible to train a single model which can predict all the target variables of interest in the simulation. Whilst powerful, neural networks are difficult to interpret,

so they may not be a suitable choice in cases where the model needs to be understood or debugged. Tunable parameters for the ANN algorithm:

- **Hidden Layers** The algorithm takes an array of integers which specify how many neurons are in each layer. The length of the array is the total number of layers. A rule of thumb for choosing the number of neurons in the hidden layer is that it should be between the size of the input layer and the size of output layer [37, pp. 48].
- **Activation Function** The activation function transforms a neuron's inputs to an output.
- **Max Iterations** The amount of iterations the training is allowed
- **Weight Gradient Strength** The learning rate, or rate of convergence. Setting it too high could cause the algorithm to oscillate.
- **Momentum** How much momentum previous deltas add, to smooth random fluctuations of the weights [38].

K-Nearest Neighbours

KNN works by classifying based on the closest training examples. It is a type of lazy learning, as there is no computation performed prior to classification. KNN can be used for classification or regression. A disadvantage of KNN for this application is the computation is done at runtime rather than offline. The tunable parameter for KNN is k, which is the number of used nearest neighbours. A good value for k is $n^{0.5}$ where n is the number of features[39].

Naive Bayes Classifier

The Naive Bayes classifier is a simple technique that uses Baye's theorem to guess the class given the features. It is called naive because it considers each feature completely independently. It can only do categorical classifications i.e. what category is the most likely. The Bayes algorithm doesn't have any tunable parameters.

Decision Tree

Certain implementations of decision trees, such as the CART algorithm used in OpenCV, support both classification and regression. A decision tree has decision nodes which have two or more branches, and leaf nodes witch are the decisions or classification. The higher a decision node is in the tree, the more important it is in predicting the outcome. An advantage of this is that you can learn how important certain features are. This functionality will be used to evaluate the feature vector in Chapter 5.

The tunable parameters of the decision tree algorithm in OpenCV are:

- **Max depth** The maximum depth of the tree.
- **Minimum Sample Count** The minimum samples required at a leaf node for it to be split. According to the OpenCV documentation a good value is 1% of the total data [40].
- **Cross Validation Folds** Cross validation allows the algorithm to use all the samples for validation, by dividing the sampleset in to a number of subsamples called folds. This assists in pruning the tree.
- **Use 1se Rule** This rule more harshly prunes the tree, decreasing accuracy but making it more resistant to overfitting. In a statistical model overfitting occurs when the model is too complex, so that it does not generalise well. It is said to have 'overfit' the training data.

Boosting

Boosting is a machine learning technique that trains an ensemble of weak classifiers to create a strong classifier. The weak classifiers are usually trees, and just need to be better than random. The OpenCV Boost algorithm implements the two-class Discrete AdaBoost algorithm. This actually makes its usage quite limited, but it can be used for the `didSurvive` prediction. Tunable parameters are:

- **Weak Count** The amount of weak classifiers.

There is another Boosting algorithm implemented in OpenCV called Gradient Boosted Trees which can deal with multiclass classification and regression.

Random Trees

Random Tree, or Random Forests, is similar to the Boost algorithm. It builds an ensemble of trees, which are all trained on different training sets. There is a variant of the algorithm called Extremely Random Trees which, when building the trees, picks a node split randomly as opposed to the Random Forest which finds the best split. The user can specify the max number of trees in the forest.

The performance of the various algorithms will be examined in the next section.

Chapter 5

Evaluation

This section will evaluate the LOD system under three criteria: accuracy, processing and memory.

5.1 Model Accuracy

Since the goal of the system is to simulate agents realistically when out of view, a measurement for proxy simulation accuracy would be useful. In most cases simulation LOD is only evaluated on the criteria of runtime performance, and sometimes a survey with real users to evaluate believability [10][18]. With the approach that has been taken, accuracy of the lower level of detail can be measured precisely. This is because we have a test set with labelled feature vectors to test the predictions against.

The dataset that has been collected by the agent sampler component is split into the training and test sets, to train and evaluate the model. Most of the algorithms can only produce models which predict one target variable. Neural Networks are the exception in that they can produce a vector output. When collecting samples, the Spawner was set to spawn between 2 and 20 agents. The Spawner created a new instance whenever a team had won or there had been a period of no encounters.

The naming convention for models is `_featureVectorVersion_(binary flags for won,duration,xpGain,hpLoss)`. For example, the model which uses version 2 of the

Table 5.1: Model _2_1000. Training set size: 5000. Test set size: 5000. This model predicts whether the agent will survive the encounter.

Model	Accuracy	Avg. predict time(ms)	Model parameters	Training time (ms)
Base	0.551			N/A
ANN	0.8414	0.0031	Activation: Sigmoid	1471
Bayes	0.7846	0.0026		3
Boost	0.8446	0.0272		2377
DTree	0.8282	0.0062		24
ERTree	0.4324	0.0015		117
GBTree	0.4308	0.0012		4
KNN	0.7366	0.221		0
RTrees	0.8444	0.0032	10 Trees	246
RTrees	0.8532	0.0254	100 Trees	2517
SVM	0.6272	0.1188		919
Ensemble	0.8774	0.1135		N/A

feature vector and outputs a predicted duration is named `_2_0100.`

The four target variables of interest are `didSurvive`, `duration`, `xpGain` and `hpLoss`. `didSurvive` is a binary classification problem. `duration`, `xpGain` and `hpLoss` are continuous variable regression problems. The Bayes and Boost algorithms can only do classification problems, so they are only used for the `didSurvive` model. In Tables 5.1 and 5.2 results are shown for models predicting `didSurvive` and `duration`. In Table 5.3 results are shown for a Neural Network predicting all four target variables.

In Tables 5.1 and 5.2, the models shown are trained to predict one target variable. For continuous target variables, the Root Mean Squared Error (RMSE) and Mean Average Error(MAE) are calculated. Both measure the average error in the same scale as the target variable being predicted, with RMSE lending extra weight to larger errors. For discrete target variables, or classes, the accuracy is simply a percentage of correct classifications. The classification results are compared against a base classifier which just chooses the class that is most frequent.

In Table 5.1, the results for the `_2_1000` model are shown. The main considerations when choosing an algorithm are the accuracy and the prediction time. The prediction time must be quick so the model can be consulted numerous times within in a single

Table 5.2: Model _2_0100 - Training set=5000 Test set=5000. This model predicts the duration (in seconds) of a battle

Model	RMSE	MAE	Avg. Predict Time (ms)	Parameters	Training Time
ANN	5.6539	4.0904	0.00287		519
DTrees	5.4629	3.5009	0.00231		34
ERTrees	5.6910	4.1188	0.23790	500 Trees	25918
KNN	8.7234	6.3291	0.22489		0
SVM	6.9322	5.2214	0.05935		523
RTrees	5.2001	3.4992	0.00327	10 Trees	386
Rtrees	5.1344	3.4684	0.39396	500 Trees	17477
Ensemble	5.9819	3.9778	0.01522	Mean	N/A
Ensemble	5.1373	3.4237	0.03176	Median	N/A

frame without slowing down the game. The speed of the Random Tree and Extremely Random Tree algorithms only become acceptable when using a small number of trees. The accuracy does decrease somewhat when the number of trees is reduced for the Random Tree algorithm, but it still has very good accuracy. The Neural Net using the Sigmoid activation function also performs very well. Table 5.2 shows the results for the duration model. Neural Networks and Random Trees again perform well. Algorithms that performed poorly for classification, such as the Extremely Random Tree algorithm, perform well here.

The accuracy and prediction time columns from Table 5.1 are plotted in Figure 5.1. Algorithms in the placed on the left of the plot are the most suitable for a real-time system due to their high prediction speed. Algorithms close to the top have the highest accuracy. Neural Network and Random Trees both perform very well, placing in the top left of the plot. Since K-Nearest Neighbours is lazily evaluated, all the computation takes place at run-time, making it quite slow. This can be seen in Figures 4 and 8 included in the appendix which plot prediction time for each algorithm against number of training samples. KNN prediction time increases linearly with training samples as opposed to the other algorithms which remain constant. Since speed is a major consideration for the LOD system - this rules out the use of KNN. The speed of the Random Trees algorithm increases linearly with the amount of trees, so there is a trade-off between accuracy and prediction speed.

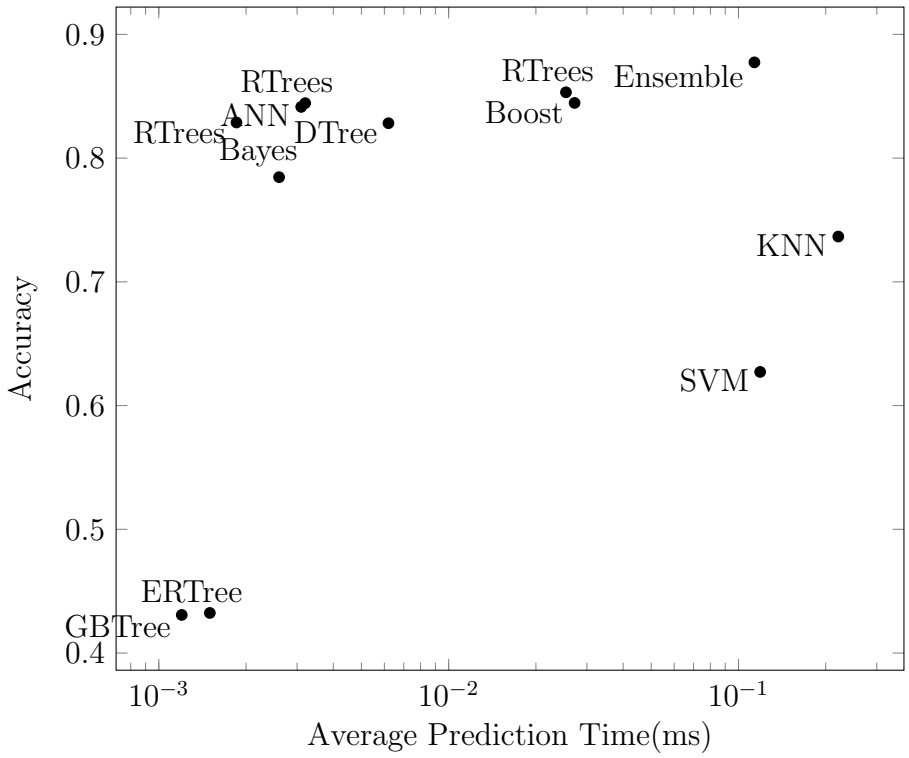


Figure 5.1: The accuracy and prediction time of the various algorithms. Multiple RTrees are shown as different forest sizes produce different results. Using a small forest will result in a speedier, but less accurate prediction, as shown with the furthest left RTrees.

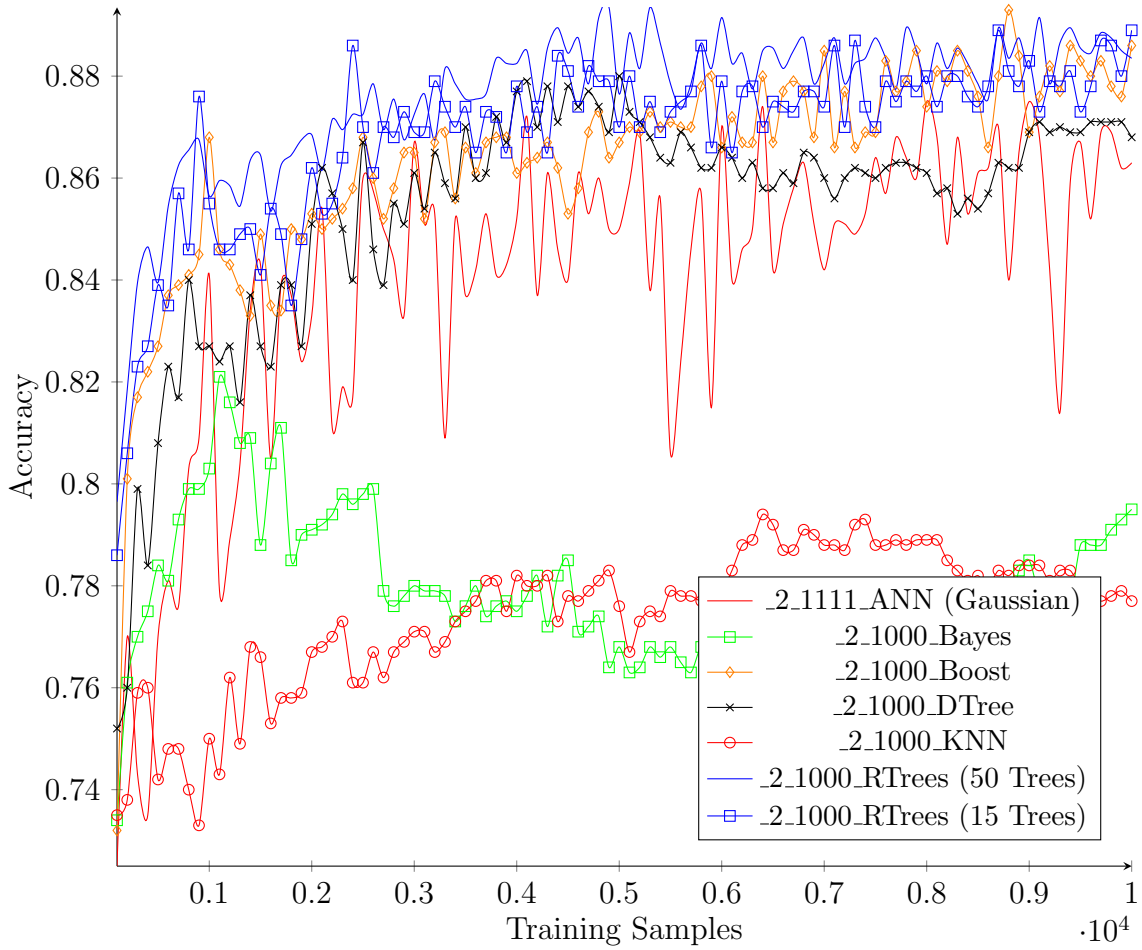


Figure 5.2: The convergence of the various algorithms predicting `didSurvive`. A test set size of 1000 was used.

An advantage of having a few different algorithms at hand is that different problems may be more suited to varying approaches. This allows the best approach to be selected for solving the problem. A potential feature of the framework is the automatic selection of the best performing model for use in the LOD system. Another application of using multiple algorithms is they can be combined into an ensemble. The ensemble meta-algorithm implemented uses a majority vote for classification problems, or the mean or median for regression problems. While the ensemble produces the highest accuracy, it comes at the cost of prediction speed at runtime.

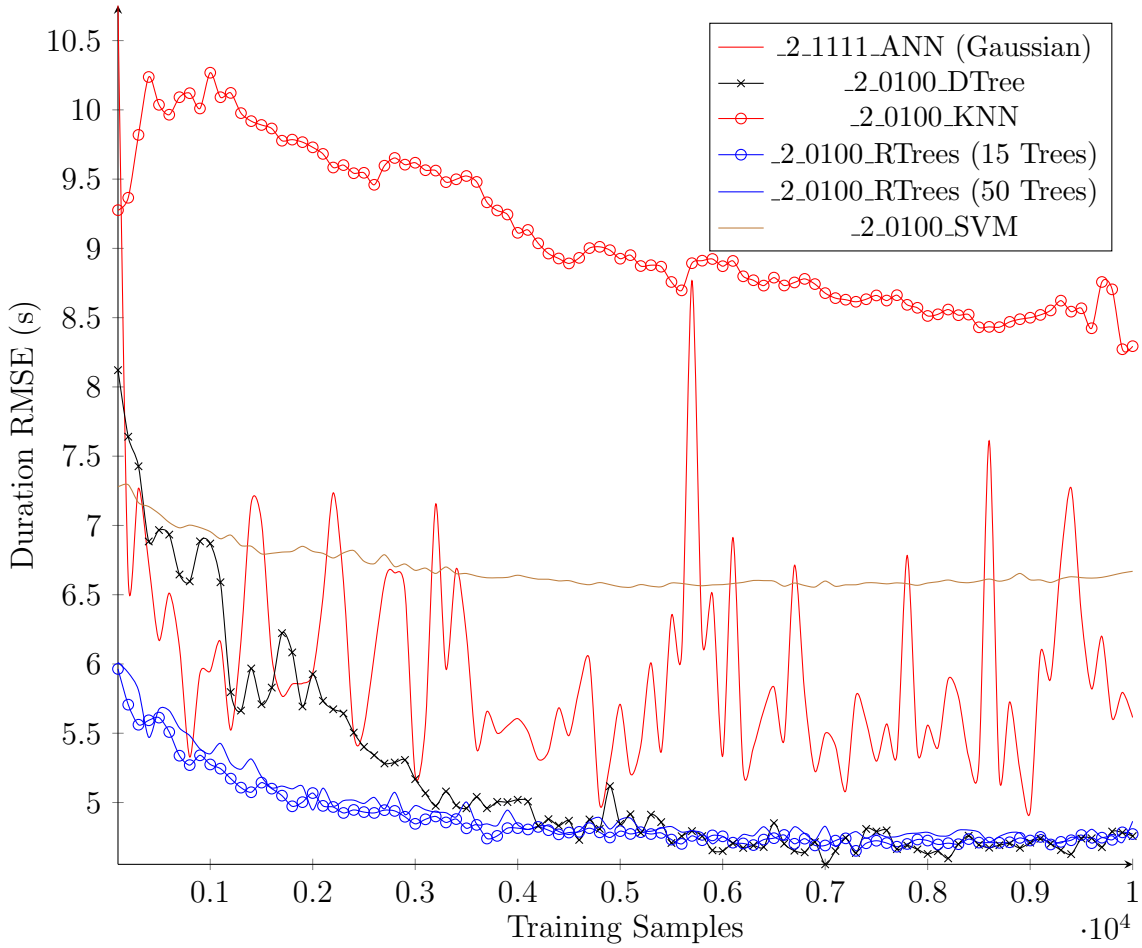


Figure 5.3: The convergence of the various algorithms predicting `duration`. A test set size of 1000 was used.

Table 5.3: Training results for Neural Nets using the Gaussian activation function trained with feature vectors version 1 (_1_1111) and version 2 (_2_1111). Training Set: 7000. Test Set: 1000.

	_1_1111	_2_1111
Avg. Prediction Time	0.002326	0.002301
didSurvive (%)	80.44	85.2
duration (RMSE)	6.6738	6.2054
duration (MAE)	4.6487	4.2610
xpGain (RMSE)	32.0619	31.3898
xpGain (MAE)	24.6001	24.0823
hpLoss (RMSE)	63.3510	60.6519
hpLoss (MAE)	50.6910	44.9081

Figure 5.2 shows how the accuracy changes over various training set sizes. The same test set is used for all of these tests, which comes from the front of the dataset. The general trend seen in Figure 5.2 is upward the more training samples that were used. However good results can be achieved with quite a small sample set. Sometimes the Neural Network Backpropagation algorithm does not converge which accounts for the occasional dip in accuracy for `_2_1111_ANN`. In these cases a very long training time was observed. Random Trees tended to be more accurate than the Neural Networks. Figure 5.3 shows a convergence plot for the prediction of duration. Further convergence plots are included in the appendix.

The results in Tables 5.1 and 5.2 are for a Neural Network with a scalar output, but similarly good results are obtained with a Neural Network outputting a vector response, as can be seen in Table 5.3, which shows results of training a neural network with $fvectorv1$ and $fvectorv2$. The second feature vector improves the prediction quality, without the extra features having any negative impact on the prediction time. The Gaussian activation function worked better than Sigmoid for Neural Networks with a vector output.

The two feature vectors used are shown below:

$$fvectorv1^{\mathsf{T}} = \begin{pmatrix} characterClass \\ unitLevel \\ unitHp \\ alliesNumberOfAgents \\ alliesAverageHealth \\ alliesAverageLevel \\ enemuesNumberOfAgents \\ enemiesAverageHealth \\ enemiesAverageLevel \end{pmatrix} fvectorv2^{\mathsf{T}} = \begin{pmatrix} characterClass \\ unitLevel \\ unitHp \\ premptiveStrike \\ alliesNumberOfAgents \\ alliesAverageHealth \\ allieTeamComposition \\ alliesCritical \\ enemiesNumberOfAgents \\ enemiesAverageHealth \\ enemiesAverageLevel \\ enemiesTeamComposition \\ enemiesCritical \end{pmatrix} \quad (5.1)$$

where $alliesTeamComposition$ and $enemiesTeamComposition$ are the vector

$$\begin{bmatrix} fractionFighters & fractionGunners & fractionArchers & fractionMages \end{bmatrix} \quad (5.2)$$

The feature vectors need to expose the underlying structure of the problem to the algorithms so they can make good predictions. Anything in the simulation that has an impact on the resulting target variable in question should be considered when selecting features. As the features are domain specific, anyone using this framework will need to do feature engineering. Choosing too many features could lead to overfitting.

The first feature vector built, $fvectorv1$, had nine features. It produced good results of over 80% accuracy, but it was missing some important information, such as the character classes of a unit's allies and enemies. For $fvectorv2$ the team composition for both sides is taken in to account. This adds eight features, as there are four character classes. The boolean *preemtiveStrike* was also added. This is true if the agent is not in combat at the time of the prediction. This means they are getting the 'jump' on their opponent in fights involving more than 2 agents. In $fvectorv1$, since the average health for the allies and enemies was just an average, it could obscure the fact that there may be some agents in critical condition, so the fraction of agents in critical condition on the allied and enemy teams are used as features.

Another benefit of having a suite of algorithms is that some of them have extra

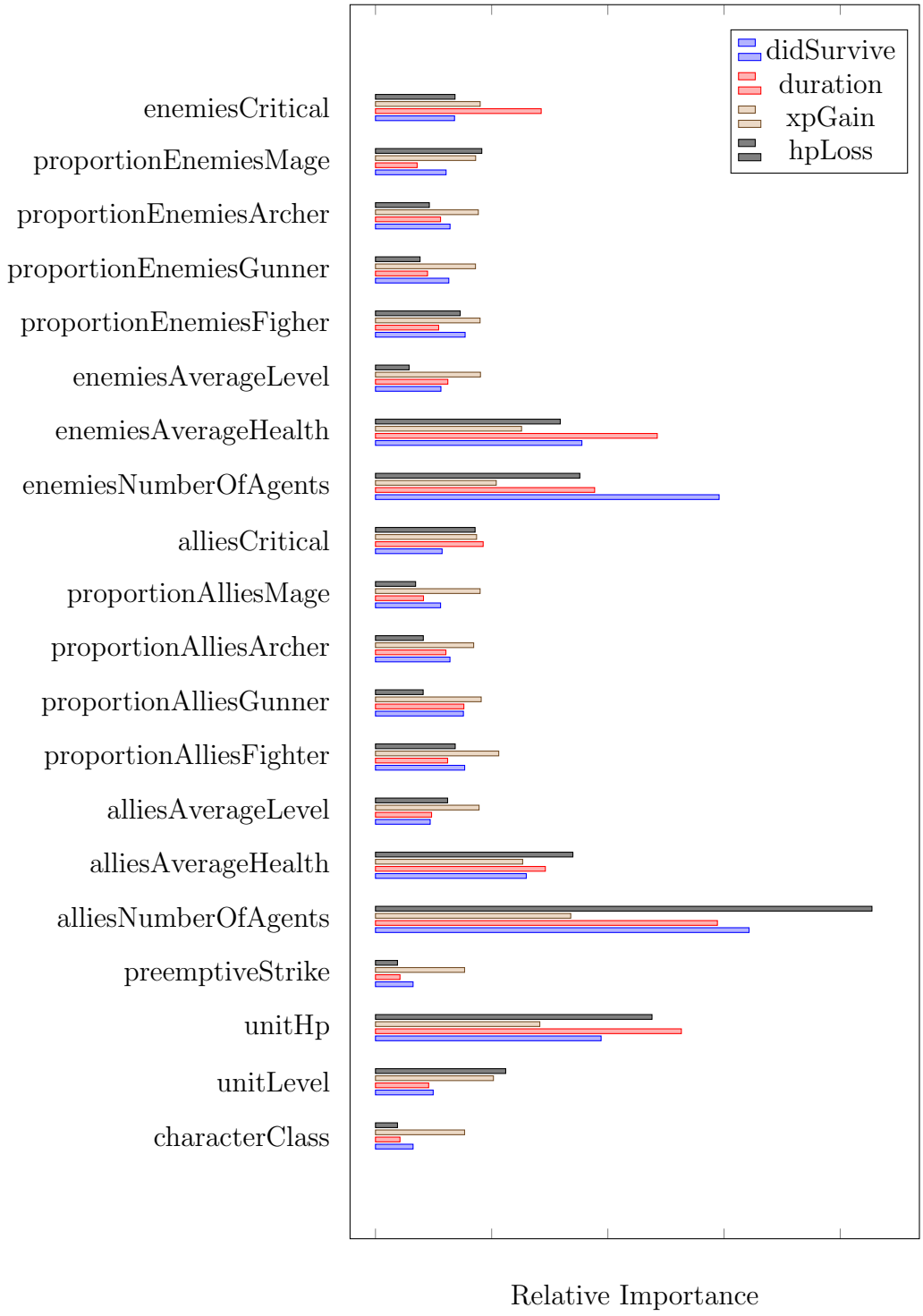


Figure 5.4: The relative importance of each variable in making a prediction for the four target variables. This data was obtained from the Random Trees algorithm.

uses outside of prediction. Whilst Neural Networks are very powerful, they are a black box, and can't expose much about how they arrived at their prediction. This is not the case with tree algorithms, which can tell you which features best split the data. The Random Forest implementation provides a function which returns a vector of the same dimensions as the feature vector containing scores for the relative importance of each feature. How important each variable is in predicting the four target variables is shown in Figure 5.4.

The best models trained can predict with $\sim 85\%$ accuracy if an agent will survive an encounter, and to within ~ 4 seconds the duration of the encounter. From visual inspection, the misclassifications usually occur in cases where the two sides are very evenly matched, and it comes down to chance, such as whose attacks are registered first or who is targeting who in larger fights. In these cases where it could go either way, a misclassification will have no discernible impact on the believability of the simulation.

5.2 Runtime Performance

The goal of the project is to simulate off-screen agents in as cheap a manner as possible while still maintaining the integrity of the simulation. The previous section has shown how the approach can produce accurate predictions when given an input vector. The other important consideration is how efficient the system is. Modern games can only allocate a very small window of the frame time processing to off-screen agents. Another important consideration is how much memory is being used by the approach, as an aim of the system is to simulate a large number of agents.

5.2.1 Processing

Unity 5 provides a CPU and memory profiler, however it doesn't export usable data so it is only really useful for visual inspection and to aid optimisation. The results in Figure 5.5 were acquired by averaging the frame update time over many frames. All tests were performed on a Windows 10 PC with a AMD FX-4170 4.20GHz Quad Core processor, 8 GB of DDR3 RAM and the GTX 570 graphics card. Version 5.3.4f1 of

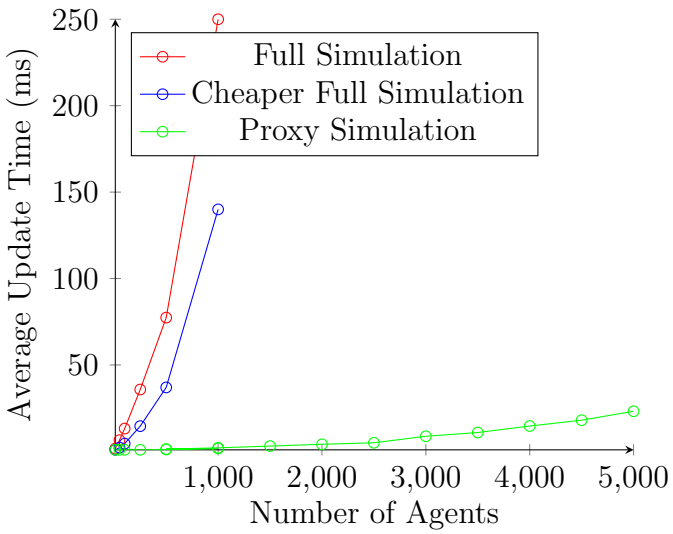


Figure 5.5: The average update time taken with different amounts of agents in the scene. The profiling was done in an optimised build of the game.

Unity3d was used.

The system was able to simulate 1000 agents at 588 frames / second on average, or 1.7 ms / frame for the full update. The budget for a frame is typically 16ms to meet the need for 60 frames per second, so this leaves plenty of time left over. Also, not all of the 1.7ms was attributable to the LOD system, as it was the full game update being measured. The 'Cheaper Full Simulation' in Figure 5.5 has the full game logic but with no audioControllers, rendering turned off, and the navmesh agents collision avoidance turned off. This allowed a more fair assessment of the savings made by the LOD system.

Some sections of code vital to the LOD system were also benchmarked. These were `SetLOD()` which performs the level of detail switch. This function is quite involved as it has to add or remove a number of components from the object based on which level of detail the agent is to be simulated at. `SetLOD()` averaged 0.27ms per call over 1314 iterations. This could be optimising using Object Pooling, which will be discussed in the Memory section of this chapter.

When an encounter begins, the details for each faction in the encounter are com-

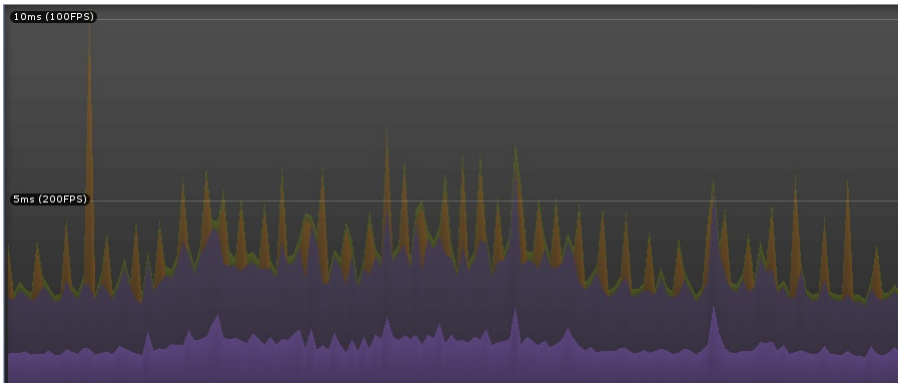


Figure 5.6: CPU utilisation in the Unity Profiler. Orange = Physics Engine. Light purple = NavMeshManager. Green = Rendering. Dark purple = MonoBehaviours i.e. scripts

puted. This computation is shared for each NPC's feature vector, so the agents only have to compute parts of the feature vector specific to them. The `Predict()` function calls `GetFeatureVector()` and then consults the model (Neural Network tested). `GetFeatureVector()` computes the feature vector for a particular agent when it needs a prediction. `GetFeatureVector()` averaged 0.005768ms over 31286 iterations. Building the feature vector is slightly more expensive than consulting the neural network itself. The full prediction function which builds the feature vector and consults the model averaged 0.013602 ms per call. If we were to budget 1ms / frame for predictions, we could perform 73 predictions.

Figure 5.6 shows the CPU utilisation in the Unity Engine when 1000 agents are being simulated at the proxy level of detail. A big chunk of the processing is used by the physics engine and the NavMeshManager. The physics engine is managing all the trigger colliders which are used to trigger encounters. Spheres are very cheap collision shapes, and with broadphase collision detection optimisations in the PhysX engine in use by Unity, thousands of sphere colliders can be easily handled. However, there is definitely scope to find an even cheaper way to track off-screen agent encounters.

An even bigger chunk of the processing is used by the NavMeshManager, even without the collision avoidance and with a relatively coarse navigation mesh used. The Unity Engine does not have any LOD features for navigation meshes. There is scope

Table 5.4: The size of the models in memory

Model	ANN	SVM	KNN	Bayes	Decision Tree	Random Trees
Size (Bytes)	176	128	28	52	24	48

to improve this aspect of the project, and work has been done on level of detail for navigation [1]. The spike near the beginning of 5.6 sometimes occurs when a very large encounter is initialised, requiring many new predictions. It is registered as physics under the profiler because it is called from a physics callback function. Spreading prediction queries over several frames is a possible solution.

5.2.2 Memory

The other runtime performance consideration is how much memory is used by the system. As discussed in Chapter 4, the NPC object expands and contracts based on what level of detail it is at. This is done to save memory resources when an agent is off-screen, as there is a lot of state at the full LOD which is no longer necessary at the proxy LOD. The trade-off is that the `SetLOD()` function is a bit more intensive as it has to add and destroy components rather than just enabling and disabling them. The observer radius for switching to back to proxy LOD has a small spillover to avoid unnecessary extra calls to the `SetLOD()` function.

There are allocations and deallocations occurring as components are added and removed from the application. Instantiating and destroying objects on a regular basis is inefficient. Using Object Pooling, instead of objects being destroyed, they are placed in a buffer and de-activated. They are then taken from the buffer and re-activated when needed. There is just one initial memory allocation at the beginning of the program. This technique was used for the large crowds in Assassin's Creed Unity[ref] as mentioned in Chapter 2. However, this would have a larger memory footprint than the current system, so it is a trade-off between memory and CPU usage. It would still be cheaper in memory than maintaining full state for every agent, as the object pool only needs to have enough components to assign to the number of NPCs that are expected to be close to the observer.

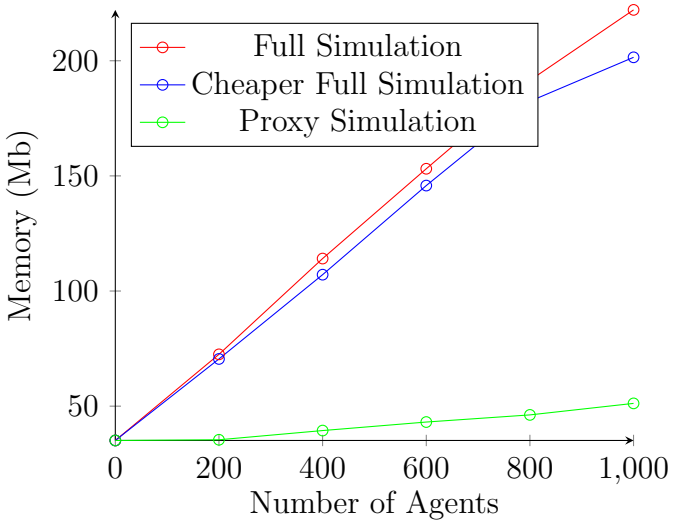


Figure 5.7: Memory usage of the application when simulating different numbers of agents. The results are from snapshots of the memory usage taken from the Unity Profiler.

Figure 5.7 shows the memory usage of the application at the different levels of detail. As can be seen, the amount of state needed for the full agents is considerably larger. A breakdown of each component's memory usage as reported by the Unity Memory Profiler can be found in the appendix.

Another consideration is the memory footprint of the model itself. To test how much space each model takes up in memory, the C++ version of OpenCV was used. This is because the unmanaged memory usage could not be assessed in C#. The results are shown in Table 5.4. Other objects in memory attributable to the LOD system are the encounter spheres and *EncounterManager* which use 136 bytes (each) and 168 bytes respectively.

5.3 Summary

Overall the system made significant gains in performance. There was a 68x speed-up between *Proxy* and *Cheaper Full* when simulating 1000 agents. There is reason to believe even more savings could be made which will be discussed in Future Work. There was a 3.94x reduction in memory usage by the *Proxy* simulation vs. the *Cheaper*

Full simulation when simulating 1000 agents.

Chapter 6

Conclusion

This chapter concludes the dissertation, discussing the main contributions, some ideas for future work and ends with some final thoughts.

6.1 Main Contributions

A design objective highlighted in Chapter 1 was to design the system in a flexible way so that it could be used by developers to add Simulation LOD to their game. The system was able to provide a lower level of logic detail without writing any new game logic code. This makes it easier to implement than existing approaches. The system lacks much domain specific code, so it shows promise as a potential middleware for implementing simulation LOD.

Another objective was that the system would be able to simulate many persistent off-screen agents cheaply. As shown in Chapter 5, the system provides a major speed-up over the full simulation, and a significant reduction in memory usage. Figure 6.1 shows the system simulating 2500 agents at real-time rates.

It was also an aim that the system provide a reduced LOD that did not diminish the quality of the simulation. In Chapter 5, the prediction system was evaluated. It could correctly decide which team would win in a battle $\sim 85\%$ of the time, with the misclassifications being in cases where the teams were very evenly matched. While the

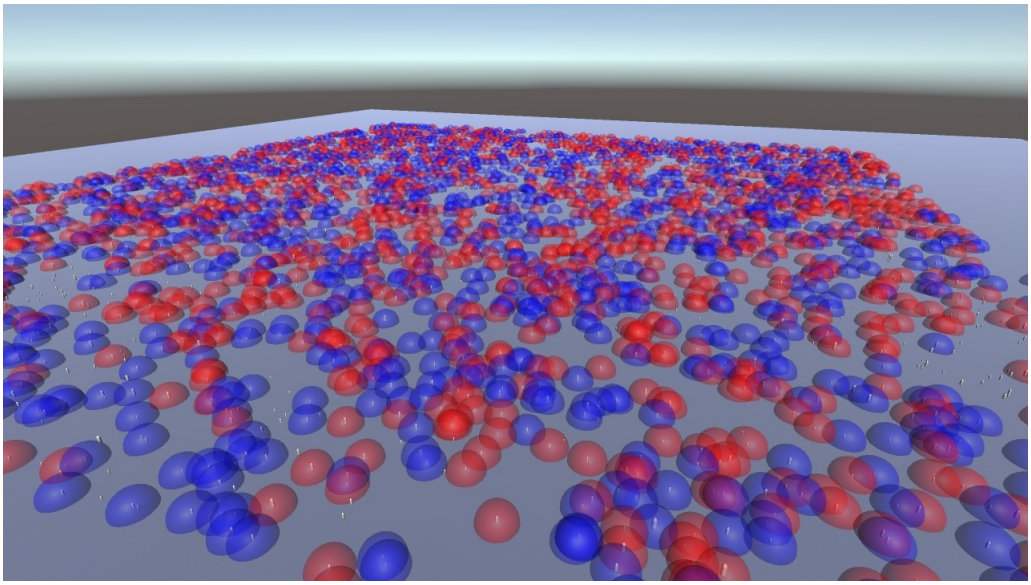


Figure 6.1: The system simulating 2500 agents at the proxy LOD, with visualisation.

behaviour of the proxy was not exactly the same as what the full simulation would have produced, there is no reduction in the complexity or believability of the behaviour.

6.2 Future Work

While the system developed should be flexible enough to be applied to other games, improvements could be made in the usability. Testing it with other games and genres would likely give insights in how to improve the workflow and generality of the system. Some examples of how the system could be streamlined are the following:

- Allow the user to easily expose features to the machine learning module. Unity3d has a class called `PropertyAttribute` that allows some metadata to be associated with a variable. These attributes appear in square brackets above the variable field. An attribute called `PotentialFeature` could be used to mark a variable to be exposed to the machine learning module.
- See the progress of the learning in a progress bar similar to those found in Unity3d for lightmapping or navigation mesh baking.

- Train and evaluate several models behind the scenes, automatically choosing the best one, where being the 'best' depends on whether prediction speed or prediction accuracy is prioritised.

In Chapter 3 the system's hierarchical approach to training the models was outlined. However the implementation only has one reduced level of detail, as no complex group behaviour was present in the full simulation used for the demo implementation. Future work could increase the complexity of the simulation to include group behaviour. Then models could be trained to predict what happens when groups of agents meet, or even what happens when civilisations meet. The system's capability to model non-combat situations could be tested. Examples of these could be negotiations, trade, personal relationships etc.

The LOD selection heuristic in the demo application is a simple distance-based one. This has some major drawbacks. With this system, agents that are behind the observer are fully simulated. This is wasteful, as the system has the capability to quickly switch LOD levels should the observer turn around. A visibility based LOD switching scheme would solve this problem. Another limitation of the system is that agents outside of the observer's radius are not visualised, even if they are within the observer's view frustum. Agents 'pop' in and out as their visualisation is enabled and disabled. Future work could be to decouple logic and visualisation so that the characters using the reduced level of detail could be visualised if they are within the observer's view frustum.

As mentioned in Chapter 5, implementing an object pooling system similar to Assassin's Creed Unity for agent components would improve the CPU performance of LOD switching. Also mentioned in Chapter 5 was the high cost of the navigation and physics in the proxy simulation. Future work could be to implemented cheaper ways to move the agents around the world, and check when they cross paths.

Finally, a perceptual survey could be designed, where participants would be asked to rate the believability of the simulation.

6.3 Final Thoughts

Shadow of Mordor has sold nearly 3 million copies copies [41], and the much touted Nemesis system was certainly a reason for its success [42]:

> What's most impressive about the nemesis system is how self-sufficient it is. This is a game about you coming into the picture and messing up the natural structure of Orc society, but with or without your input, that society will shift. An Orc veteran on the rise will make a power play to move up in the ranks, or a weaker captain will get himself eaten by a Warg, leaving an empty leadership position for some previously unknown soldier to fill.

This shows there is a desire for complex, persistent NPCs in open world games. The LOD system described in this dissertation could be used today to facilitate an even more complex system than that seen in Shadow of Mordor.

The encounter system works well for an open world game, in which agents move around a large environment. However, it would be interesting to see how the architecture would need to change when considering other genres. Perhaps in a sports game, the outcome of other games in the league could be predicted in a similar fashion. In this case an encounter system would not be needed, as the teams would naturally 'encounter' each other when its their turn to play in the season.

Applying machine learning techniques to simulation LOD was the main idea of this dissertation. Using this approach a cheaper simulation can be built for off-screen NPCs, but with the same behavioural complexity. It becomes possible to simulate many more agents with complex behaviour than if they were simulated without using Level of Detail. This allows interesting, emergent behaviour to arise from background NPC characters, resulting in a more lively, dynamic game environment.

Appendix

The DVD attached to the back of this dissertation contains the source code and experiment data for the project.

Table 1: Component Memory Usage in Bytes

	Proxy	Full	Cheaper Full
NavMeshAgent	136	136	136
RigidBody	160	160	160
SkinnedMeshRenderer	800	800	800
NPC GameObject	360	600	600
NPC Transform	288	288	288
Model GameObject	0	152	152
Model Transform	0	152	152
Armature GameObjects	0	3800	3800
Armature Transforms	0	6600	6600
MrpgChibit GameObject	0	168	168
MrpgChibit Transform	0	256	256
SphereCollider	136	136	136
NPC MonoBehaviour	288	288	288
EnemyStatController	288	288	288
MrpgEnemyChibit Material	0	2400	2400
MrpgcChibitColour Material	0	2400	2400
AudioSource	0	800	0
AudioController	0	288	0
CombatController	0	288	288
TargetController	0	288	288
EnemyMovementController	0	288	288
ProxyMovementController	288	0	0
CapsuleCollider	0	144	144
Animator	0	90500	90500
Weapon Mesh	0	452	452
Weapon Transform	0	256	256
Weapon GameObject	0	184	184
Weapon Mesh Filter	0	64	64
TOTAL	2744	112176	111088

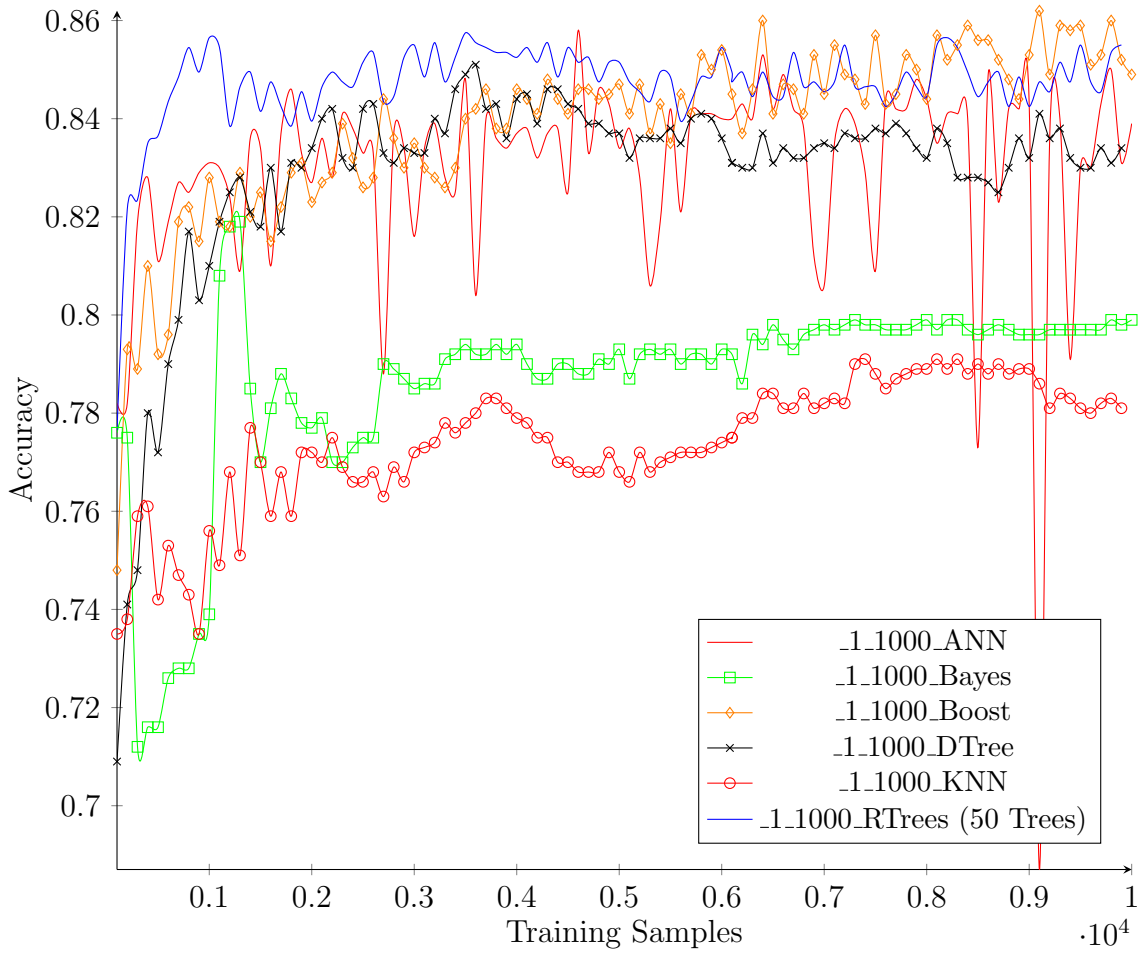


Figure 2: The convergence of the various algorithms predicting `didSurvive`, using `fvector_v1`. A test set size of 1000 was used.

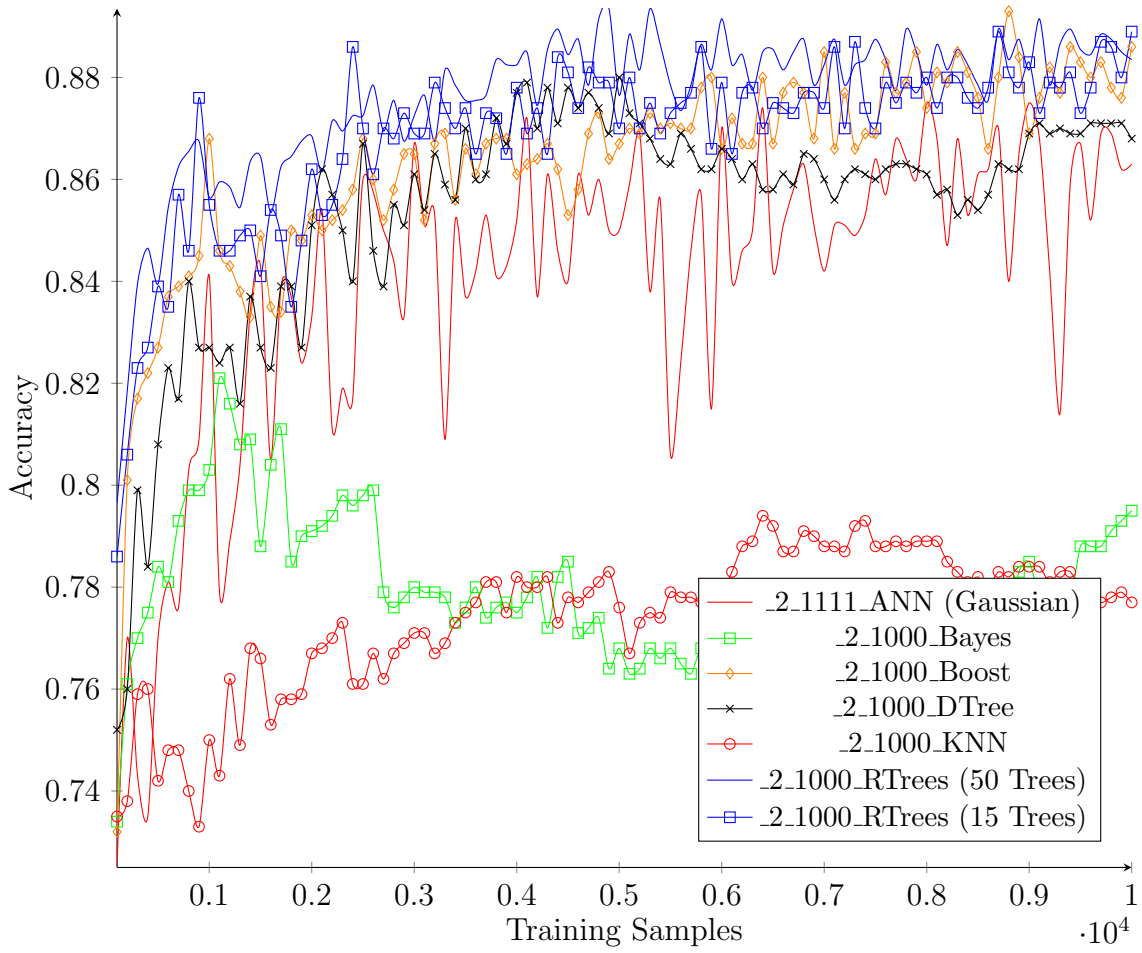


Figure 3: The convergence of the various algorithms predicting `didSurvive`, using `fvector_v2`. A test set size of 1000 was used.

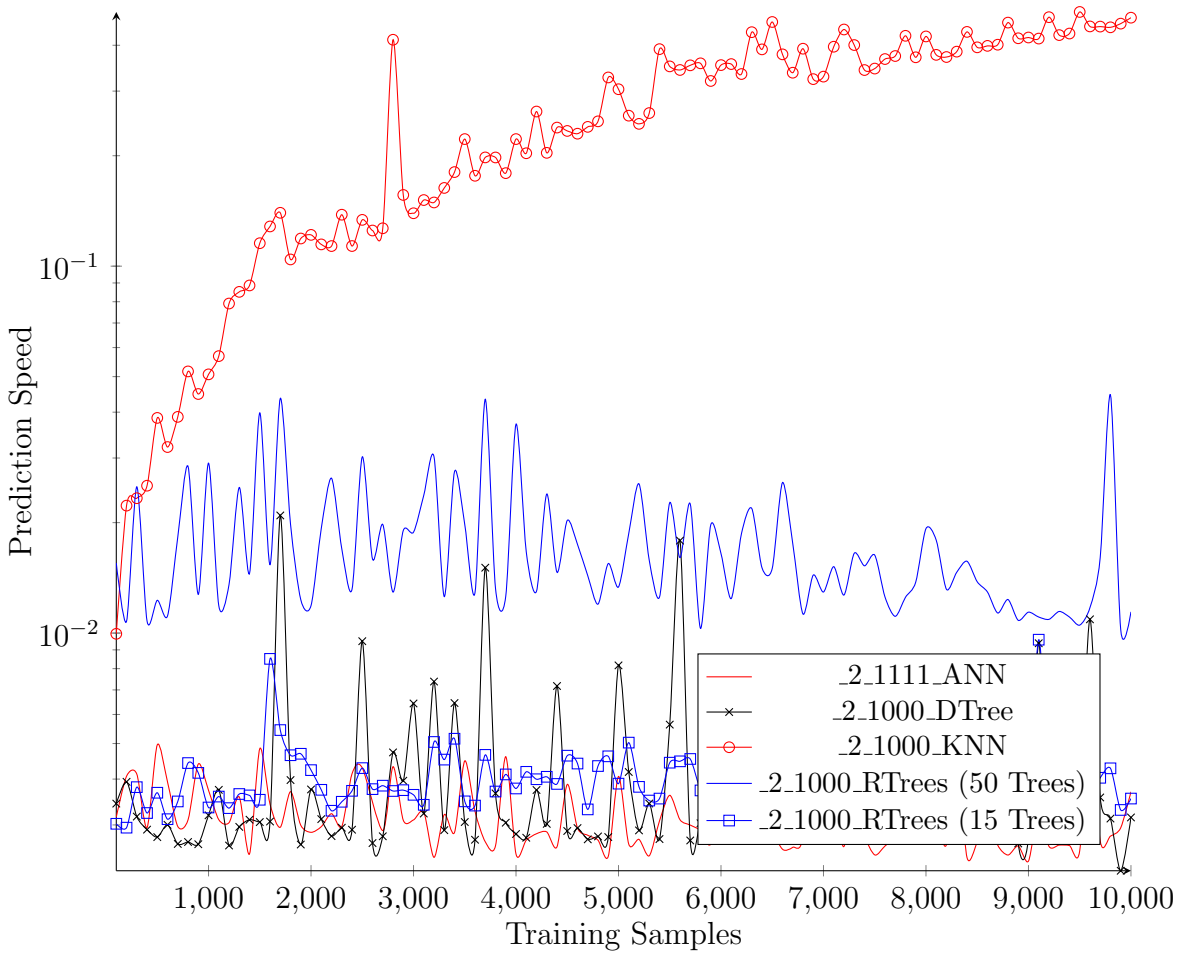


Figure 4: The prediction speed of the various algorithms predicting `didSurvive`, using `fvector_v2`. A test set size of 1000 was used.

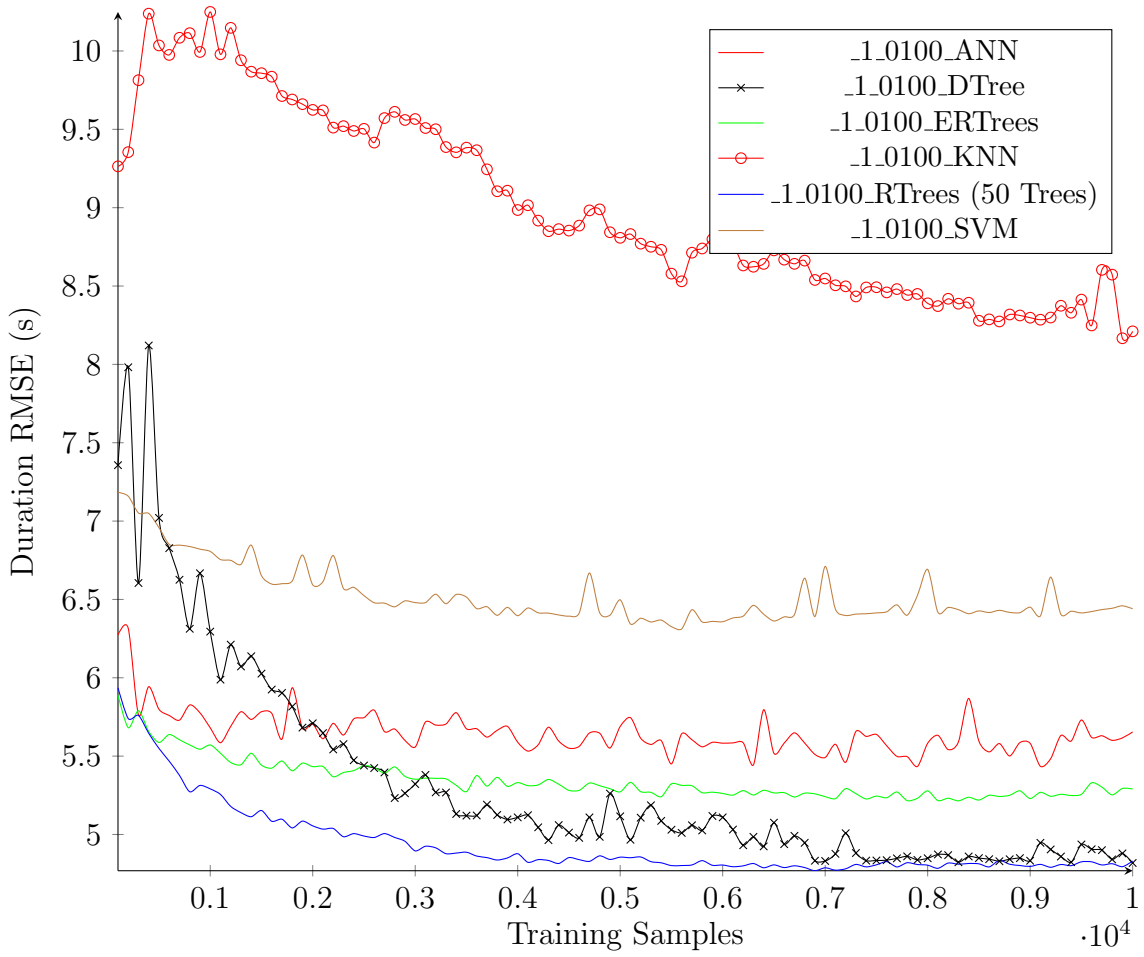


Figure 5: The convergence of the various algorithms predicting `duration`, using `fvector_v1`. A test set size of 1000 was used.

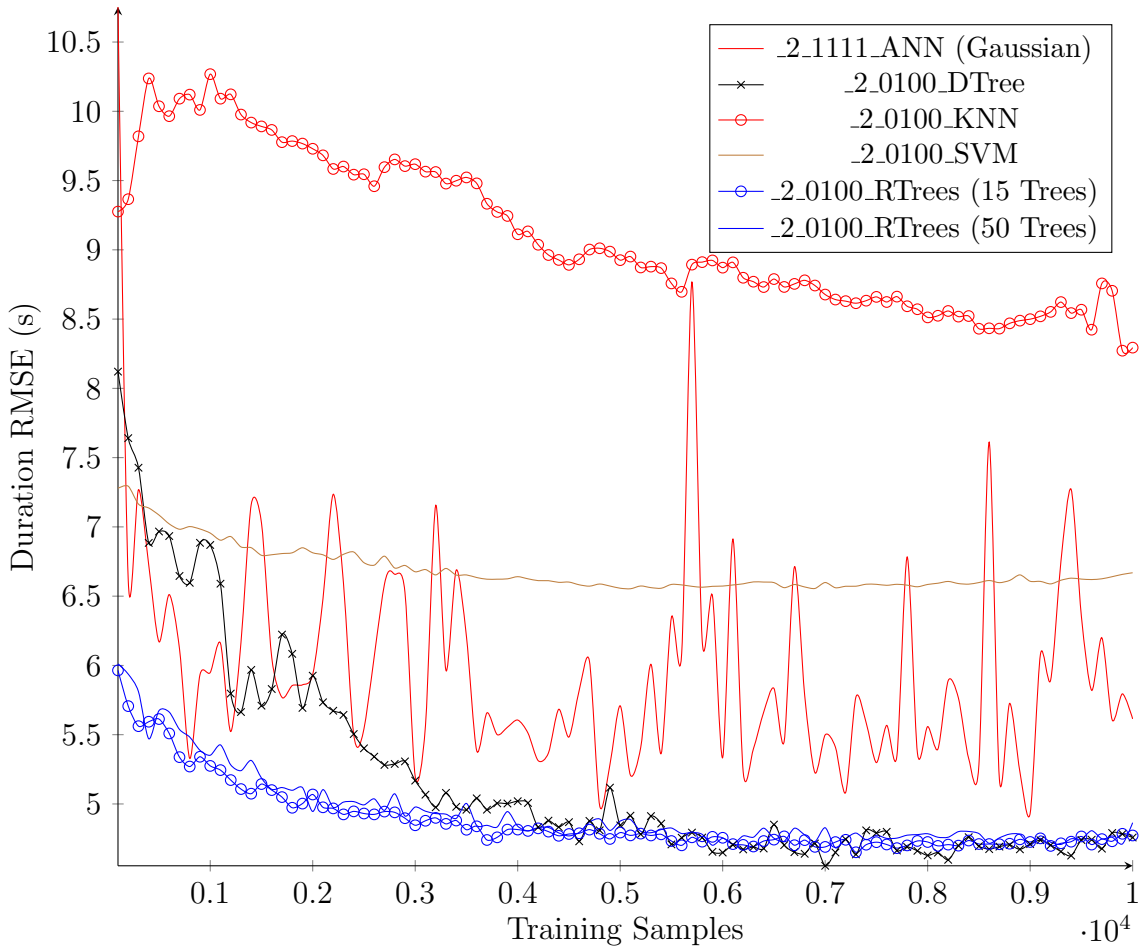


Figure 6

Figure 7: The convergence of the various algorithms predicting `duration`, using `fvector_v2`. A test set size of 1000 was used.

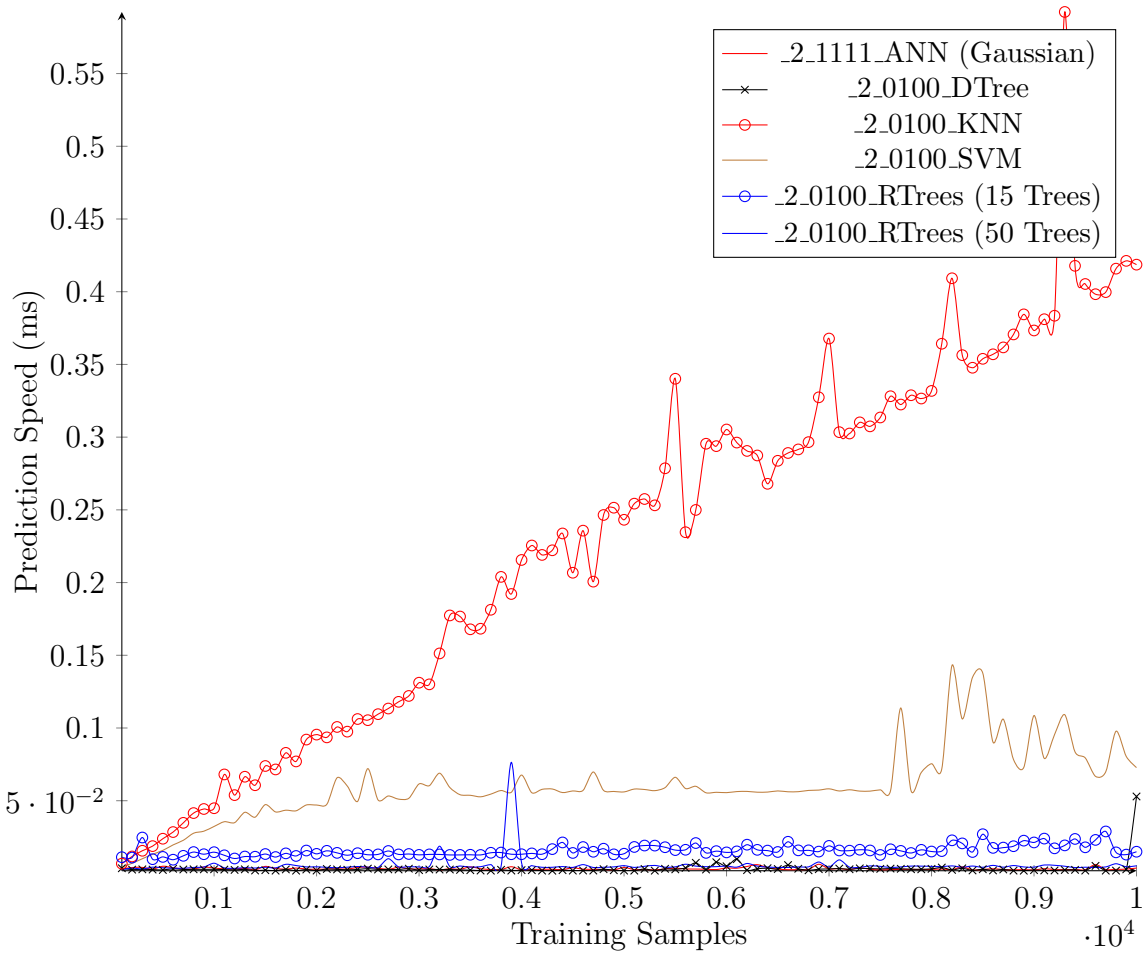


Figure 8: The prediction speed of the various algorithms predicting `duration`, using `fvector_v2`. A test set size of 1000 was used.

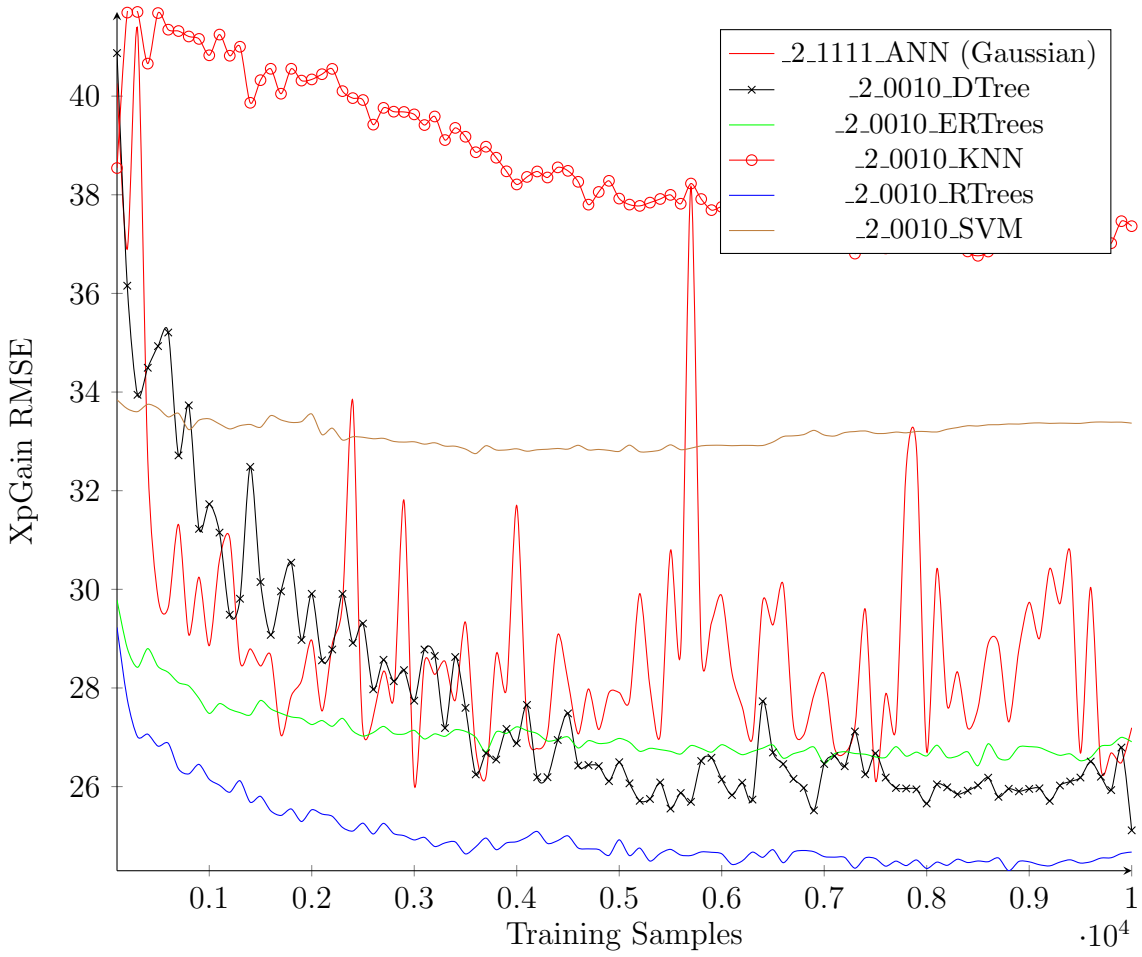


Figure 9: The convergence of the various algorithms predicting `xpGain`, using `fvector_v2`. A test set size of 1000 was used.

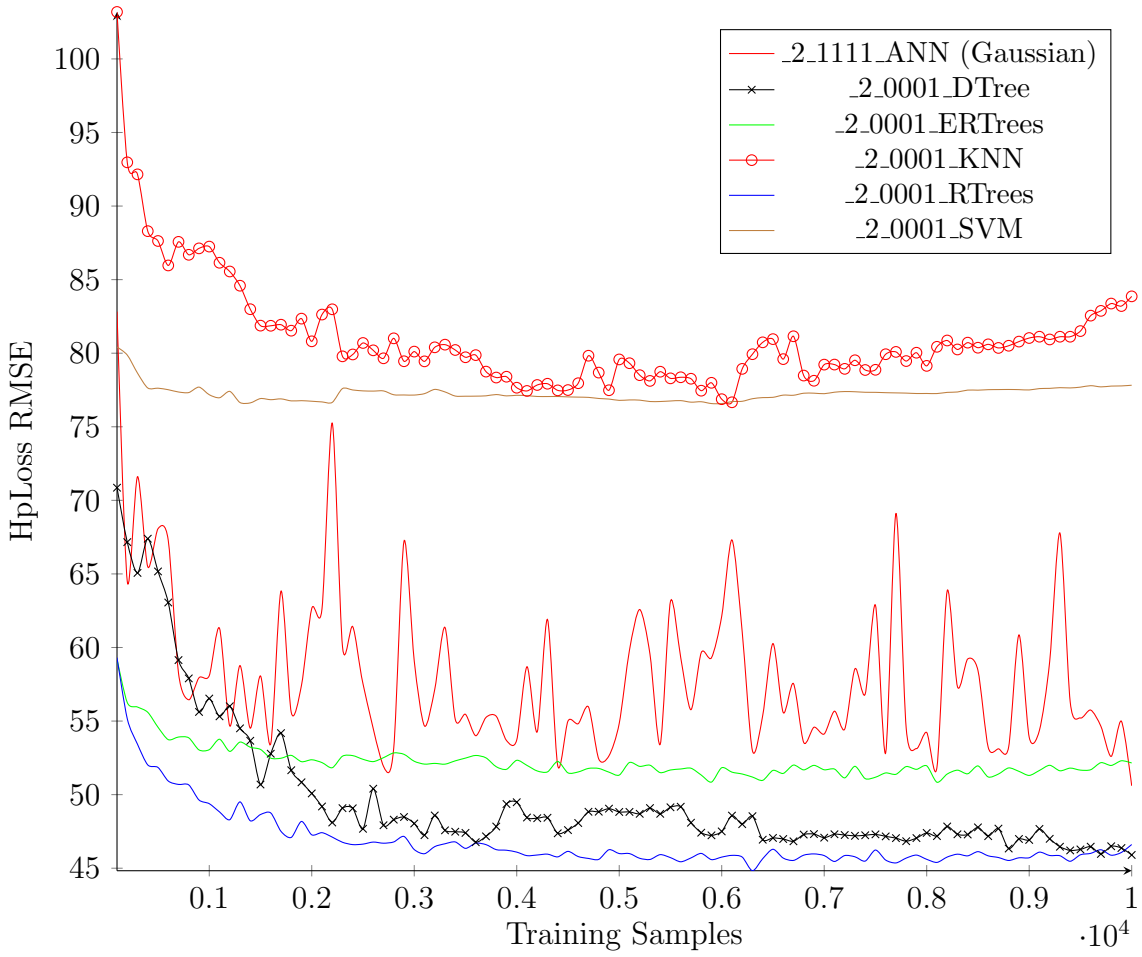


Figure 10: The convergence of the various algorithms predicting `hpLoss`, using `fvector_v2`. A test set size of 1000 was used.

Bibliography

[1] C. Brom, O. Šerỳ, and T. Poch, "Simulation level of detail for virtual humans," in *International Workshop on Intelligent Virtual Agents*, pp. 1–14, Springer, 2007.

[2] D. Osborne, P. Dickinson, *et al.*, "Improving games ai performance using grouped hierarchical level of detail," 2010.

[3] "Gdc vault - massive crowd on assassin's creed unity: Ai recycling." `http://www.gdcvault.com/play/1022411/Massive-Crowd-on-Assassin-s`. (Accessed on 21/08/2016).

[4] "Middle-earth: Shadow of mordor - official site." `https://www.shadowofmordor.com/`. (Accessed on 08/30/2016).

[5] P. Yang, B. Harrison, and D. L. Roberts, "Identifying patterns in combat that are predictive of success in moba games,"

[6] B. MacNamee and P. Cunningham, "Creating socially interactive no-player characters: The μ-siv system," *Int. J. Intell. Games & Simulation*, vol. 2, no. 1, pp. 28–35, 2003.

[7] "Unity - manual: Level of detail." `https://docs.unity3d.com/Manual/LevelOfDetail.html`. (Accessed on 08/30/2016).

[8] "Creating and using lods — unreal engine." `https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/HowTo/LODs/`. (Accessed on 08/30/2016).

[9] T. Plch, M. Marko, and P. Ondrácek, "An ai system for large open virtual world.," 2014.

[10] M. Wißner, F. Kistler, and E. André, "Level of detail ai for virtual characters in games and simulation," in *International Conference on Motion in Games*, pp. 206–217, Springer, 2010.

[11] S. Chenney, O. Arikan, and D. Forsyth, "Proxy simulations for efficient dynamics,"

[12] O. Arikan, S. Chenney, and D. A. Forsyth, "Efficient multi-agent path planning," in *Computer Animation and Simulation 2001*, pp. 151–162, Springer, 2001.

[13] B. Mac Namee and P. Cunningham, "Proposal for an agent architecture for proactive persistent non player characters,"

[14] "halflife/roach.cpp at master valvesoftware/halflife." `https://github.com/ValveSoftware/halflife/blob/master/dlls/roach.cpp`. (Accessed on 21/08/2016).

[15] C. O'Sullivan, J. Cassell, H. Vilhjalmsson, J. Dingliana, S. Dobbyn, B. Mcnamee, C. Peters, and T. Giang, "Levels of detail for crowds and groups," in *Computer Graphics Forum*, vol. 21, pp. 733–741, Wiley Online Library, 2002.

[16] B. MacNamee, S. Dobbyn, P. Cunningham, and C. OSullivan, "Men behaving appropriately: Integrating the role passing technique into the aloha system," in *Proceedings of the AISB02 symposium: Animating Expressive Characters for Social Interactions (short paper)*, pp. 59–62, 2002.

[17] "The elder scrolls official site." `http://www.elderscrolls.com/skyrim/`. (Accessed on 08/30/2016).

[18] B. Sunshine-Hill, "Managing simulation level-of-detail with the lod trader," in *Proceedings of Motion on Games*, pp. 13–18, ACM, 2013.

[19] "Ubisoft - assassin's creed unity." `https://www.ubisoft.com/en-US/game/assassins-creed-unity/`. (Accessed on 08/30/2016).

[20] "Object pool optimization patterns game programming patterns." `http://gameprogrammingpatterns.com/object-pool.html`. (Accessed on 21/08/2016).

[21] "S.t.a.l.k.e.r.." `http://www.stalker-game.com/`. (Accessed on 08/30/2016).

[22] "A-life, emergent ai and s.t.a.l.k.e.r.: An interview with dmitriy iassenev — aigamedev.com." http://aigamedev.com/open/interviews/stalker-alife/. (Accessed on 21/08/2016).

[23] "Gamasutra - postmortem: Monolith productions' middle-earth: Shadow of mordor." http://www.gamasutra.com/view/news/234421/Postmortem_Monolith_Productions_Middleearth_Shadow_of_Mordor.php. (Accessed on 21/08/2016).

[24] "bwapi/bwapi: Brood war api." https://github.com/bwapi/bwapi. (Accessed on 29/08/2016).

[25] A. A. Sánchez-Ruiz, "Predicting the outcome of small battles in starcraft," in *ICCBR, Workshop Proceedings*, 2015.

[26] L. Studios, "Black & white," 2001.

[27] N. Mullally, "An emotional model for background characters in open world games," 2014.

[28] T. McNulty, "Residual memory for background characters in complex environments," 2014.

[29] "Modular rpg combat - asset store." https://www.assetstore.unity3d.com/en/#!/content/12879. (Accessed on 21/08/2016).

[30] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[31] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[32] "Opencv — opencv." http://opencv.org/. (Accessed on 21/08/2016).

[33] "Emgu cv - browse /emgucv/2.4.9-alpha at sourceforge.net." https://sourceforge.net/projects/emgucv/files/emgucv/2.4.9-alpha/. (Accessed on 21/08/2016).

[34] "Machine learning with opencv." http://bytefish.de/blog/machine_learning_opencv/. (Accessed on 28/08/2016).

[35] "Thread ninja - multithread coroutine - asset store." https://www.assetstore.unity3d.com/en/#!/content/15717. (Accessed on 29/08/2016).

[36] "System.threading namespace." https://msdn.microsoft.com/en-us/library/system.threading(v=vs.110).aspx. (Accessed on 29/08/2016).

[37] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[38] "Neural networks opencv 2.4.13.0 documentation." http://docs.opencv.org/2.4/modules/ml/doc/neural_networks.html. (Accessed on 21/08/2016).

[39] "A detailed introduction to k-nearest neighbor (knn) algorithm." https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/. (Accessed on 21/08/2016).

[40] "Random trees opencv 2.4.13.0 documentation." http://docs.opencv.org/2.4/modules/ml/doc/random_trees.html. (Accessed on 30/08/2016).

[41] "Middle-earth: Shadow of mordor (playstation 4) - sales, wiki, cheats, walkthrough, release date, gameplay, rom on vgchartz." http://www.vgchartz.com/game/77129/middle-earth-shadow-of-mordor/. (Accessed on 29/08/2016).

[42] "Middle-earth: Shadow of mordor review: all those who wander — polygon." http://www.polygon.com/2014/9/26/6254177/middle-earth-shadow-of-mordor-review-lord-of-the-rings-ps4-xbox-one. (Accessed on 29/08/2016).