DOCTORAL THESIS

Learning to Identify Bugs in Video Games

Benedict Wilkins

Supervisor: Kostas Stathis

Secondary Supervisor: Chris Watkins

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

Department of Computer Science
Royal Holloway, University of London, United Kingdom

March 2023

Declaration of Authorship

I Benedict Wilkins hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

01/03/2023

Acknowledgements

First and foremost, I would like to convey my deepest appreciation to my primary supervisor Prof. Kostas Stathis. Who after meeting in my undergraduate days set me on the path I now find myself on. His unwavering support and generous gifts of time throughout my academic journey have made the process that much more enjoyable. I have no doubt that our conservations over the years have sown the seeds for a lifetime of study in what are for myself some of the most profound questions one can ask. I am sure the answers will surprise us both.

The lab that Kostas fosters is one of freedom. Each of my colleagues in their diverse study of intelligent agents brought new collective insight. I feel fortunate to have been part of it and so thank you, Emanuele, Joel, Nausheen, and Pallavi. I would also like to thank my secondary supervisor Prof. Chris Watkins for his valuable feedback and discussion at critical moments that ended up shaping the direction of my research. As well as those at SIE whose efforts helped launch this project and ground my research.

Outside the academic environment, I would like to thank my mother, father and sister Zara for engaging with my ceaseless barrage of facts, quotes, theories pertaining to the nature of intelligence, learning, action, novelty, curiosity, agency, and all my other ramblings. But more importantly, for their love, emotional support and encouragement, all of which have been consistent throughout my life. It seemed to me that it did not matter to them what I pursued, as long as I was fulfilled. This encouragement unspoiled by expectation is the greatest gift a parent can give, and I am happy, so thank you.

Peering out of a crack in my existential box with a beetle I thought was mine alone, I received a note. It was an image of two beetles in a sea of vibrant blue. And the crack widened. Achala, your reminders that *humaning* is really the right way to go, your support in the everyday running of our lives and your love which you so freely express has brought me the deepest joy. Like the trees, the streams and the bugs, thank you for occurring so beautifully.

Finally, I would like to thank myself. After all, I had as much to do with this as anyone. Except perhaps

my laptop, who with my guidance learned many things, and I many things from it. To paraphrase Einstein,

> *My laptop and I are more clever than I.*

Abstract

The use of intelligent software agents promises to revolutionise video game testing. While agents automate the time-consuming task of repeatedly playing a game in search of issues, humans can spend their time on the more creative aspects of game development. Despite the substantial advancements in game-playing that have made this possible, agents are reliant on humans, or hand-crafted guards, to determine whether there are issues with the game's design or functioning.

This thesis aimed to develop testing agents that can identify issues with a game's function or *bugs* with minimal human involvement by learning from their prior experiences. The problem is framed as one of anomaly detection, where bugs correspond to abnormality or novelty in an agent's experience. A series of approaches based on Self-Supervised Learning (SSL) and Causal Inference (CI) have been developed to enable an agent to measure abnormality or otherwise model the game to subsequently identify bugs. The focus was on laying the foundations for testing agents that operate over the same input/output modalities as human testers. The approaches were evaluated by testing a diverse collection of purpose-built video games, where they successfully identified bugs from a broad class.

This thesis is among the first work to investigate the use of machine learning in the context of video game bug identification. It presents an exposition of the problem of learning *intended behaviour*, and then endeavours to develop solutions that demonstrate the benefits of using agents with learning capabilities for testing. Namely, ease of reuse across projects (reusability) and in identifying bugs that would otherwise require human involvement to be found (capability). The use of agents equipped with sophisticated game-playing algorithms and the identification tools outlined in this thesis offers a new framework for video game testing.

Contents

Chapter 1

Introduction

The video games industry now dominates the global entertainment market. The development of video games is a serious business, and a critical aspect of this is testing. A well-tested game will lead to a higher-quality product and more enjoyable experience for the player, which translates directly to value for the business. Spurred on by the increasing availability of computing power and industry success, games now boast massive realistic virtual worlds and swathes of content for players to enjoy. With more to test there is an increased pressure to find bugs before a game's release, the presence of which will drastically diminish the players experience if found during play. Development companies already take costly steps to avoid this, from proper use of software development and testing strategies early on, to heavy investment in manual playtesting and player-driven acceptance testing (Washburn et al. 2016; Santos et al. 2018). Even with these steps, it is common for bugs to slip through with many modern titles containing more than their fair share.

The increasing burden on testers and developers to find and fix bugs in ever larger virtual worlds has led those that are ahead of the curve to turn towards intelligent automation solutions. While automation has always been a key part of testing, for example in organizing, managing, running tests and reporting their results, these automation solutions still require a substantial amount of manual work, especially in writing the tests themselves. In addition, they cover only a portion of the testing life-cycle, usually around unit and integration testing, but do not help with playtesting, which tends to be assigned greater importance (Kasurinen et al. 2014) and is the more time-consuming. In playtesting, a tester's goal is two-fold: First, to evaluate important design requirements, such as fun-factor, playability and balance. Second, to uncover functional problems, such as those that result in a crash. Automating the former is difficult as design requirements tend to be *human-centric* and are not easily quantified. Nevertheless, software agents driven by Artificial Intelligence (AI) are beginning to be used to assist in this evaluation. For example, in tuning difficulty settings (game balancing) (García-Sánchez et al. 2018) or determining the paths players might take through a level (Holmgård et al. 2018). Being human-centric, testers are still relied upon to interpret the results that these agents produce. Functional problems on the other hand, tend to require less human involvement as simply by playing and exploring the game an agent might trigger a crash or exception, alerting us to the presence of a bug (Zheng et al. 2019). Other functional problems might be identified at a higher level, for example, in reachability analysis, which asks whether it is possible to reach a goal or whether the player can get stuck (Gordillo et al. 2021).

Software agents have some history of use in video game testing, but it is only recently with the continued advancements in areas such as Reinforcement Learning (RL) (Mnih et al. 2013; Silver et al. 2016; Vinyals et al. 2019) that it has become possible to use them in such a diverse manner. The key addition is their learning capabilities, which removes much of the need for manually engineering complex game-specific behaviours and enables them to reach the required level of proficiency in playing. Their use promises to save an enormous amount of time that would otherwise be spent on game playing, leaving humans to the more creative aspects of development. There is a substantial amount of research into game playing in the broader AI community, although much of it is focused on reaching super-human performance (Vinyals et al. 2019). This has its uses in game testing (e.g. in game balancing), but the task of a testing agent is often more nuanced. Efficient exploration with the goal of maximizing environment coverage, or exploration directed towards certain kinds of problems may be preferable.

The use of these testing agents has great potential for improving video game quality, but in functional terms, they are only as good as the tests they can perform. Many of the functional problems that video games exhibit are difficult to quantify and many do not result in a crash. Consider for example, problems that manifest graphically, or that are part of a complex interaction in the game's physics engine. It is not easy to write guards that check for these problems, and unlike a human tester, an ill-equipped agent will miss them altogether. This is a serious automation bottleneck as it means that humans will inevitably have to play the game and explore sufficiently to test for these problems themselves.

The problem of distinguishing between what is intended and what is not (bug or not a bug) is known broadly in software testing as the *test oracle problem*. (Barr et al. 2015). In essence, the problem is about finding ways of automatically drawing this distinction for software where it may be difficult to more formally specify what is intended. Software in this class, which includes compilers, Machine Learning (ML) systems, and video games, among others, is referred to as being *untestable* (Segura et al. 2020). There are various existing techniques that aim to address the problem. A popular one being metamorphic testing (Chen et al. 2020c), which uses the software as its own test oracle by comparing sets of input/output pairs using predefined relations or constraints. Along similar lines, a compiler for example, might be tested by comparing the generated machine code against that generated by an existing compiler implementation.

Tangential to these more traditional approaches, this thesis advocates and explores *learning* as a means to obtain the relevant information on intended behaviour. The insight is that video games, although untestable in the traditional sense, are actually highly structured. They follow common patterns in design and functioning which may be learned and exploited to resolve ambiguity around what is intended. At a high level, one part of a game's functioning may tell an agent about how other parts are supposed to function. There are many persistent quantities, such as the properties of simulated objects, or the effects that a player's actions have on the game state, over which an agent with learning capabilities could measure statistics. The claim is that by measuring the right statistics over abstract representations of its experience playing the Game Under Test (GUT), or other similar games, a testing agent might directly, or indirectly, make statements about intended behaviour. This would enable the agent to catch bugs as they happen without relying on guards or models that may be difficult (or impossible) to formally specify.

There is already a substantial body of work in ML that aims to solve a very similar problem (Chalapathy et al. 2019). In the field of anomaly detection, researchers are attempting to derive general mechanisms for identifying *abnormal* situations given experience or *data*. In video surveillance for example, we might be interested in identifying fires, dangerous situations, or suspicious behaviour from CCTV footage (Ramachandra et al. 2020). The techniques developed in these areas might be applied to the problem of identifying bugs in video games. Despite the obvious overlap, there are historically very few works in the automated game testing literature that aim to do this (Nantes et al. 2013)[1]. Much of the work has instead been on testing video game design and on developing game playing agents. There are a number of reasons for this: lack of suitable data, the challenges involved, and perhaps as result of some disconnect between the software engineering/game development community and the machine learning/statistics community as they are apparently solving different problems. Only very recently have a handful of works started to demonstrate the potential for learning in identifying video game bugs (Taesiri et al. 2022; Liu et al. 2020; Wilkins et al. 2020). The work presented in this thesis being some of the earliest.

Putting aside the primary motivation, which is to make existing testing agents more *capable* by enabling them to identify issues that would otherwise be missed, there are some other potential advantages that

---

[1] (Nantes et al. 2013) being one of the seminal works that did not have the benefit of the more recent developments in machine learning. In some ways it was a few years too early, the results are impressive nonetheless.

learning-based approaches to testing may have to offer. The first is in line with a goal of traditional testing: test *reusability*. If tests can be reused across projects then development time is saved. A good example of this is in the use of general game playing agents with learning capabilities. In theory, all that needs to be specified is a high-level goal or objective that the agent aims to achieve, game specific behaviours can then be learned rather than specified manually. The hope is that the same may be said for testing agents that come to know intended behaviour through their experiences. At the very least, one can see that if an agent operates over the same input/output space as human testers (i.e. images of the fully rendered screen), much of the engineering effort that would be spent interfacing tests with the game's internal implementation might be saved.

1.1 Aims & Research Objectives

The overarching aim of this thesis is to develop agents that are able to identify bugs in video games by learning from their experience. More specifically, by solving for, or optimizing a high-level learning objective that *implicitly* specifies a video games intended behaviour with respect to its experience, an agent should be able to identify bugs that would otherwise be challenging using more traditional methods. It should be able to do this in a largely *game independent* manner by obtaining the required domain specific knowledge from experience playing the GUT and by being *decoupled* from the games underlying implementation as much as possible. In short, such an agent should be capable and reusable.

Developing such an agent rests on our ability to specify suitable learning objectives. Developing these objectives is effectively the goal of any anomaly detection practitioner in a new setting of interest. How this can be done for bug identification is a central question that this thesis aims to address. This brings us to the first concrete research objective:

Objective 1: To develop learning objectives that will allow agents to identify bugs in video games, especially those that would otherwise require human involvement to be identified.

Whether an agent is able to identify a particular kind of bug will depend on the inherent biases the agent has, but also on what kind of supervision is available. In the setting that is closest to real game development, the agent is unlikely to have access to a comprehensive labelling of the different bugs that might manifest in the GUT. This effectively rules out straightforward supervised learning as a possible paradigm. Instead, the agent at best has access to *weak* supervision, which may come from a labelling of bugs in other games, from earlier versions of the GUT, or even from outside of video gaming altogether. Transferring knowledge of bugs between games, versions or otherwise is a very interesting direction, but one that is known to be extremely challenging. There has been limited progress on transfer learning in the wider anomaly detection literature (and in AI and ML more broadly). Instead, weak supervision might refer to the situation where an agent can assume everything it experiences during training is intended, or *normal*. In anomaly detection this is known as *novelty detection*. This setup happens to have a direct correspondence to the problem of regression testing where the aim is to find unwanted changes to a specific part of a video game after an update to the wider project. Novelty detection has been studied in a variety of problem settings and there is a lot of yet unexplored potential in its application to regression testing. The second research objective is as follows:

Objective 2: To develop capable and reusable agents that can make use of *weak* supervision to perform regression testing.

Regression testing might be considered a step-down from full-blown playtesting where transfer learning might ultimately be more applicable. But the two can be brought closer together by requiring that a regression testing agent play specific portions of the game to uncover bugs. Regression testing also shares many of the inherent challenges that are faced in playtesting. To give some examples, the fact that a game will likely not be explored completely and not all bugs may be seen during training, this means an agent must learn something interesting so as to generalize to unseen situations and not confuse novel but intended experiences with bugs. Other significant challenges, such as the fact that the game is under development and subject to continual arbitrary change are also shared. For the moment, the hope is that progress on the simpler regression testing setting will lead to insights in the more general setting where we may not be able to make the same assumptions.

In the long term, to fully solve the automated testing problem and reach a level of proficiency close to that of a human tester, fairly general intelligent agents will be required. Video games already have a long history of being used to test new developments in AI. The focus has previously been on proving that the latest system can play specific games, for example, Chess (Campbell et al. 2002). It was then on developing game playing algorithms that can be used to play a class of games, only requiring training on each (Mnih et al. 2013). This is the kind of reusability that we might realistically hope for in a testing agent at present. In the future, we might conceive of agents that can play novel games with a shallow learning curve like that of an experienced human game tester. The same might be said of the problem of learning intended behaviour, although for this we are really only on the cusp of what is now possible for game playing. Nevertheless, part of the aim of this thesis is to develop more general testing agents, the focus being on how to bring about reusability. For this, we might look at *our* built-in biases and capabilities that enable us to be proficient testers. This brings us to the third research objective:

Objective 3: To investigate and pinpoint one or more of the capabilities that enable humans to be proficient testers, and attempt to operationalize them with the goal of developing more general testing agents.

This is a rather lofty objective which requires some care when making claims or drawing any conclusions. The key will be in the careful operationalization of any such capability, and whether there can be any substantive demonstration of its role in video game testing.

In order to achieve the first three objectives there is a major practical hurdle which needs overcoming. This is the distinct lack of available data with which to train and evaluate any testing agent that might be developed. Video games and their associated assets (code, art, etc.) are often closely guarded intellectual property, especially during development. Even in considering open-source, it would be very costly to obtain usable data and associated labels from bug reports. This leads to the fourth and final research objective:

Objective 4: Curate and make available the data required to train and evaluate video game testing agents. The data must be labelled and include diverse examples of realistic video game bugs.

1.2 Contributions

The following is a list of the primary contributions made in pursuit of the four main thesis objectives:

Chapter 3

- We develop a formal grounding for the bug identification problem in video games as untestable software, specifically in relation to prevalent ideas in anomaly detection and machine learning.
- We outline the key research challenges in this area and a vision of the future of game of testing automation, including some of the most promising research directions.

Chapter 4

- We implement an experimental platform that openly supports research in automated bug detection by providing a means to train and evaluate new approaches to the problem.
- We make available multiple datasets containing realistic video game bugs.

Chapter 5

- We develop State-State Siamese Networks (S3N) as an approach to identifying bugs that relate to the dynamics of a video game environment in the context of novelty detection and regression testing.

- We demonstrate the use of Self-Supervised Learning (SSL) approaches (including S3N) as a means to regression test 2D and 3D video games, highlighting the key benefits of doing so (as outlined above) over more traditional approaches.

Chapter 6

- Inspired by the capabilities of biological agents, we develop a causal framework for disentangling self-caused and externally-caused sensory effects.

- We design an algorithm that, under certain assumptions, allows an agent to disentangle these effects.

- We develop the notion of *metamorphic action relations*, which along with the causal framework and algorithm (above) maybe be used to identify bugs with a player's interaction with a video game (i.e. issues relating to *action*, such as unresponsiveness).

1.3 Thesis Structure

Chapter 2

Chapter 2 introduces key concepts and reviews the broader video game testing automation literature. The initial focus is on outlining the progress that has been possible thanks to developments in machine learning, such as in automated debugging. The focus then shifts to the problem of Automated Bug Detection (ABD), and the progress that has been made in this area in recent years, especially in relation to the *search* problem - having software agents play a game with the aim of uncovering problems with design or functionality. In part, this exploration makes painfully clear the lack of progress on the equally important identification problem - the ability to identify bugs in one's experience. With approaches largely relying on crashes, or handwritten guards to identify the issues encountered by an agent. The latter part of the chapter introduces the identification problem at a high-level and reviews the few existing works that aim to address it by going beyond handwritten guards, all while motivating learning as a potential testing paradigm to follow.

Chapter 3

Chapter 3 presents an exposition of learning as an approach to the bug identification problem. This includes developing a formalization of bug identification as one of anomaly (or novelty) detection, and an analysis of the suitability of different notions of normality and of the various learning paradigms. The subsequent discussion is centred around how one might address the bug identification problem using general notions of normality and expected behaviour, as well as the problems that learning-based approaches entail. This chapter can be viewed both as an exposition and as a vision of the future of automated bug identification in video games.

Chapter 4

Chapter 4 presents the World of Bugs (WOB) platform (Wilkins et al. 2022). The platform aims to support automated bug detection research by making available video games that contain bugs. The platform aims to resolve the issue of obtaining data for training and evaluating approaches to the Automated Bug Detection (ABD) problem, both in search and identification. The games and bugs available in the platform are used in later chapters to evaluate the approaches that are developed. In the later part of the chapter, in preliminary experiments, the functionality of the platform itself is regression tested using a simple learning-based approach.

Chapter 5

Chapter 5 makes a first serious attempt at identifying bugs in video games in the context of regression testing using machine learning. State-State Siamese Networks (S3N) (Wilkins et al. 2020) is developed as an approach that aims to identify bugs from visual observations (the full rendering of the game as would be seen by a human player). Extensive experiments with S3N (and other Self-Supervised Learning (SSL) approaches) are performed on a range of substantially different video games, including some simple 2D maze-like environments, Atari 2600 games and those 3D video games made available by the WOB platform. The bugs that are identified are relatively diverse and include: player out of bounds; geometry clipping; unintended object; unintended shortcut; freeze; texture corruption; geometry corruption; high force; among others. While these represent only a small portion of the many bugs that video games exhibit, they are among the most common problems and are seen in many games. The work is to my knowledge the first that performs experiments that so clearly demonstrate the benefits of using learning for the purposes of bug identification.

Chapter 6

Chapter 6 attempts to operationalize the following capability: the ability to distinguish between sensory effects (changes in one's observation) that are *self-caused* and those that are *externally-caused*. Self-caused effects are those changes that are due to the agents own action, externally-caused effects are due to the action of other agents or environmental processes. This ability is thought to be central to a variety of functions in biological agents, both in their physiological responses and in higher-level cognitive processing. Investigating this capability is motivated by the observation that, to make statements about the intended effects of their actions an agent first needs to know these effects. Rather than specify them manually, to be reusable the agent should learn them from experience. To better frame the investigation, the notion of *metamorphic action relations* as an instance of metamorphic testing is developed as a means to show how an agent with this capability might be used in practice. A number of experiments demonstrating its use in bug identification are presented and discussed.

Chapter 7

Chapter 7 concludes the thesis, summarizing findings and discusses future work.

Appendix

The appendix starts with a glossary. The technical appendix then presents details on the experiments performed, additional experiments and discussion, reproducibility information, and details of the video games used in experiments. Numerical results are presented at the very end of the technical appendix.

1.4 Reader Assumptions

The primary topics of this thesis are AI, ML, anomaly detection, software testing and video games. Although every effort has been made to introduce key concepts in the relevant portions of each chapter, knowledge of the fundamentals of these areas is generally assumed, particularly on the machine learning side. For ML and AI practitioners, the key testing and video game concepts are presented in the glossary and detailed primarily in chapter 2. As part of the glossary there is a list of the video game bugs that are explored at different points in this thesis. Chapter 6 requires the reader be familiar with the fundamentals of Causal

Inference (CI), which is a requirement distinct from the rest of the thesis.

1.5 Publications

- Benedict Wilkins and Kostas Stathis (2023). “Disentangling Reafferent Effects by Doing Nothing”. In: *The Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023.* AAAI Press

  Status : Published, see chapter 6.

- Benedict Wilkins and Kostas Stathis (2022). “World of Bugs: A Platform for Automated Bug Detection in 3D Video Games”. In: *2022 IEEE Conference on Games (CoG)*, pp. 520–523. DOI: `10.1109/CoG51982.2022.9893616`

  Status : Published, see chapter 4.

- Benedict Wilkins, Chris Watkins, and Kostas Stathis (2020). “A Metric Learning Approach to Anomaly Detection in Video Games”. In: *2020 IEEE Conference on Games (CoG)*, pp. 604–607. ISBN: 9781728145334. DOI: `10.1109/CoG47356.2020.9231700`. arXiv: `2005.10211`

  Status : Published, see chapter 5

Chapter 2

Video Game Testing Automation

In this chapter, this thesis is placed in the broader context of video game testing automation. We review associated literature, define terms and take initial steps to outline the stance taken on the problem of Automated Bug Detection (ABD) in the remaining chapters. Integral in this, the two central problems of ABD, *search* and *identification* are highlighted. Broadly, search means to play and explore a video game with the aim of uncovering problems with its design or functionality. Identification means to recognize that one has encountered such a problem, or *bug* during this search.

Although this thesis is focused primarily on the latter problem, before going forward it is important to appreciate that the problems inform each other. As such, this chapter has greater focus on search and video game testing automation more broadly. The intention is to highlight bug identification as an essential missing piece in video game testing automation. After contextualizing the problem, identification is discussed at a high-level and related literature is reviewed. This motivates and leads up to chapter 3, where a thorough investigation of identification centred around *intelligent* automation with machine learning is given.

The chapter is structured as follows. In section 2.1 key concepts, processes and definitions that form the basis of video game testing are presented. Then in section 2.2 the focus is shifted toward testing automation where an overview of the four steps of testing, bug detection, reporting, analysis, and verification with respect to automation is given. Scope is then narrowed to the problem of ABD. Section 2.3 gives some background on intelligent agents in the context of testing, followed by an extensive review of the automated bug detection literature and agent-oriented search. Section 2.4 introduces the problem of bug identification at a high level, further reviews relevant literature and motivates machine learning as an automation paradigm in preparation for the next chapter. Finally, section 2.5 gives an overview of the problems discussed.

2.1 Video Game Testing

Like all software, video games need testing. Testing is part of the Quality Assurance (QA) process, its purpose is to control *quality* and ensure that the software does what it is supposed to. In other words, that it meets the relevant requirements[1] or the goals set out for the project.

Video game requirements tend to centre around player experience or playability - is the game fun? Playability requirements are often difficult to pin down, but require testing nevertheless. Ensuring that a game is functional is just as important in this regard. Bugs can be jarring or frustrating for a player. If their avatar doesn't do what they tell it at the right moment or if they inexplicably fall through the floor, this is going to detract massively from any immersive experience the game might be trying to create. Ultimately, a game that provides a good player experience will have the best chance of success on release, and testing is integral in this.

2.1.1 Development & Testing Roles

As large and complex pieces of software, video games are typically developed by teams of people with roles that determine their responsibilities. In more modern agile development the lines between roles are blurred (Crispin et al. 2009) and terms like *developer* or *tester* which are meant to distinguish these roles are used more for convenience. All roles ultimately have the same aim - to deliver working and complete software that provides value to the business and its customers. The list of roles is long: programmers, designers, artists, testers, domain experts, managers, with numerous specializations and crossover (Schultz et al. 2005). The roles we are interested in are what we might broadly call testers and developers. A tester is anyone who is verifying that the software works as intended. That is, they are checking that the software meets the project requirements. A developer is anyone who is bringing the software closer to the requirements, by developing code, 3D models, artwork or other assets, writing narratives, designing characters etc.

2.1.2 Types of Testing

Testing is a complicated business. There are many kinds of testing, each aims to address particular problems faced both during and after the development of software. The four phases of testing are given below:

- Unit testing - A testing phase where the smallest parts or *units* of an application are independently tested.

---

[1] When referring to requirements, I am discounting what might be called stakeholder requirements, such as those relating to finance or legality. These are part of broader quality assurance but are generally not part of testing as defined here.

- Integration testing - A testing phase where units, modules or components of an application are tested together to ensure they work together as intended.
- System testing - A testing phase in which all of the components in an application are tested together to ensure the system as a whole works as intended.
- Acceptance testing - A testing phase where the whole application against the requirements. This may involve testing *in the wild* with real users.

Within these phases there are many kinds of requirements that need to be tested, each demanding its own specialized tests. Tests can further be separated into those that address either functional or non-functional requirements. Functional requirements define what the system is supposed to *do*, for example, *a player should be able to do action X*. Non-functional requirements define how the system is supposed to *be*, for example, *the game should be fun*. Non-functional requirements also include accessibility, scalability, extensibility, security, and playability, among others.

There are types of testing that aim to stabilize development and maintain production software, these include regression testing and smoke testing. Regression testing is a kind of testing that ensures that already existing software components don't break after an update. Smoke testing aims to cover the most critical aspects of the software, usually relating to stability (i.e. does it crash with expected input?) or basic functionality. Part of acceptance testing includes testing with real users, so-called alpha and beta testing. Alpha and beta testing is a kind of black box testing - the testers don't have knowledge of the underlying code. White box testing on the other hand is where testers do have knowledge of the underlying code, this is the case if for example the tester is also a developer. There are many other types of testing, they usually have a focus on a specific requirement, such as performance testing, security testing or stress testing. In video games there is also playtesting, which simply means to test by actually playing. Playtesting is usually associated with acceptance testing, but may also be done during integration and system testing to test specific interactions.

2.1.3 Errors, Faults, Failures and Bugs

The terms fault, error and failure are not used consistently ("IEEE Standard Classification for Software Anomalies" 2010; Avizienis et al. 2004). The definitions provided by the IEEE standard ("IEEE Standard Classification for Software Anomalies" 2010) are given below:

- Error - A mistake or conceptual misunderstanding of a human.

- Fault - A manifestation of an error in software.
- Failure - Observable incorrect behaviour of a system.

Put into words, the developer *erroneously* writes *faulty* code that leads the system to *fail*. Perhaps the developer neglects to write code that allows the player to perform a certain action. This error on the part of the developer leads to a fault - the absence of the required code. The absence of code leads to the observed failure - the player cannot perform the action when they try.

There are a variety of other terms which are commonly used in place of these terms, such as *anomaly*, *flaw* or *defect*. For the sake of clarity we opt not to use these terms without providing an explicit definition beforehand. The term *bug* might refer to any of the above terms (*Software bug Definition* n.d.) and is broadly defined as *a mistake or problem in a computer program* (*bug* n.d.). Going forward the term *bug* will be used as synonymous with *failure* as defined above. This definition will be made formal and precise in later chapters. For the moment, note that *incorrect behaviour* is defined with reference to intended behaviour, which is outlined by the requirements. An incorrect behaviour is any mismatch between the intended behaviour and observed behaviour.

We might be inclined in some instances to use terms such as *enhancement* or *improvement* to refer to features that are yet to be added. Any missing feature may also be called a bug as the fact that it is missing constitutes incorrect behaviour. Other more general terms including *issue* and *problem*, are typically used to refer to bugs or faults.

2.1.4 Testing Process

Testing is deeply integrated in the development of video games, this means that both testers and developers are part of the process. Testing is a central driving force in many development methodologies, particularly agile methodologies (Crispin et al. 2009). Although each methodology has its own peculiarities when it comes to testing, there is some commonality. Regardless of methodology, virtually all types of testing proceed via the following steps:

- Bug detection - The process of searching for and identifying a bug in the game. Search typically means actually playing some portion of the game. Identification means to recognize that there is mismatch between observed behaviour and intended behaviour.
- Reporting - The process of documenting a bug and informing the relevant programmer, artist or other development team member of the bug's existence and requesting a fix.

- Analysis - The process of recreating, checking and fixing the reported bug, also called *debugging*.

- Verification[2] - The process of checking that a particular bug has indeed been fixed.

Generally it is a tester's responsibility to perform all steps bar analysis, which is the responsibility of a developer.

2.2 Automated Video Game Testing

Video game testing is notably different when compared with other software testing (Santos et al. 2018; Politowski et al. 2021a). There is a strong emphasis on using dedicated testing teams or end users (players) to find problems (Pascarella et al. 2018) and a distinct lack of testing automation. Some of this difference is explained by unique requirements, such as *fun-factor*. Video games are designed to be engaging and to be an *experience* for the user, and a large part of testing is centred around this (Kasurinen et al. 2014). An extensive survey of the grey literature (Politowski et al. 2021b; Politowski et al. 2020) found that problems with design (relating to player experience) are the number one problem faced during development. It is not just that measuring the satisfaction of design requirements is difficult, but the way they impact the development process itself. Video game development is a dynamic process of iterating on and implementing ideas, and variations of these ideas until the game *feels* right and plays well.

This has led to a development culture that *doesn't waste time on testing automation*. This is unlike other software development where testing automation is an essential part of the process. The culture forfeits the benefits of automation, namely reusability, repeatability, increased coverage, and for video games, time saved actually playing. The simple reason for this is that it is currently too difficult to implement effectively. The unfortunate irony is that the presence of bugs, which is much more likely in poorly tested games, leads to a far worse user experience.

In the last few years there has been a surge in attempts to automate some aspects of testing in video games, particularly using methods in AI and machine learning. However, it is still widely recognized that there is a long way to go (Politowski et al. 2022). The current state of testing automation in video games, and software more broadly is discussed in the sections to follow.

---

[2] not to be confused with verification in the project management sense (Verification & Validation) which refers to the process of checking that the software meets its specification (i.e. *testing* as presented here).

2.2.1 Automated Testing Frameworks

The kind of testing automation familiar to most involves software frameworks that automatically run suites of tests. This kind of automation bridges the gaps between steps in the testing process outlined in section 2.1.4.

Unit testing frameworks are a good example of this (Runeson 2006), their primary purpose is to make it easier for developers to organize, run and summarize results of collections of unit tests. The unit tests themselves still need to be provided by the development team, but once implemented no further input is required to run them. In this sense, detection is automated at the lowest level. By extension, verification is also automated at this level since re-running a test is enough to verify that an issue has been fixed. If a test is designed well, reporting can also be automated by stack tracing tools with pre-written error messages e.g. `"Unit test X failed at LINE NO : ERROR MESSAGE : STACK TRACE"`.

Unit testing is simple but very effective for catching simpler issues, sanity checking and may even be used to guide development (e.g. Test Driven Development (TDD)). It fundamentally relies on a developer's ability to write tests that capture intended behaviour. When moving to more complex types of testing, those that consider larger sub-systems and their interactions, writing tests is a significant challenge. At this level, automation becomes more about producing tests rather than running them. While there is value in testing frameworks, they are of limited use if the individual steps of the testing process can't themselves be automated. In the sections to follow, some of the work that aims to automate these steps is presented and discussed.

2.2.2 Automated Reporting

Automated reporting is meant to assist in describing an issue, or bug, and communicating it to the relevant parties. It is important to include as much useful information about an issue as possible as this will help developers locate, reproduce and ultimately fix the problem. At the bare minimum a report needs to include: the platform or environment used, a summary or description, categorization, stack-trace or error message if applicable, and a priority or severity indicator.

Issue trackers serve as the foundation for report automation (Bissyandé et al. 2013), their purpose is to organize and keep records of issues. Issue trackers are a crucial management tool for keeping track of progress, software stability, and organizing quality assurance. These tools allow users or team members to submit different kinds of issues, bug reports, general queries, feature requests etc. Large projects can often have hundreds of issues that need to be sorted, summarized and prioritized. Doing this automatically is an active area of research. Tools have been developed that automatically categorize and prioritize issues (Alenezi et al. 2013; Kallis et al. 2019; Dhasade et al. 2020; Izadi et al. 2022), or otherwise help to organise issues (Song et al. 2020). Others guide report writers by analysing the content of a report. For example, (Imran et al. 2021) automatically chooses follow-up questions if it determines crucial information is missing.

Beyond issue tracking, there are tools that try to automate report writing itself. For example in (Shi et al. 2022) where issue reports are generated automatically from team chats, discussion or message logs. It is important that the reports contain information that allows developers to perform the analysis step. Stack-traces and error messages are an important part of this, (Feng et al. 2022) develops tool that provides additional stack-trace context in GUI applications with video recordings of the user interaction.

The scope for automated reporting is broad, ranging from smart organization to automated report writing. Many of the automation solutions rely on machine learning to analyse natural text. The need for intelligent automation is a common theme in automated video game testing as will become more apparent in subsequent sections.

2.2.3 Automated Analysis

Automating analysis, or *debugging* is an extremely difficult problem, and is for the most part out of reach for video games and most other software. Beizer in his text on software testing techniques (*Software testing techniques* 1990) notes the considerable difference between detection and analysis.

> *Testing [Detection], as executed, should strive to be predictable, dull, constrained, rigid, and inhuman. Debugging [Analysis] demands intuitive leaps, conjectures, experimentation, and freedom.*[3]

Beizer additionally notes that *"automation [of analysis] is still a dream"*. In the thirty odd years since, there has been meaningful progress on the problem, see (Wong et al. 2016) for an extensive review. A particularly ingenious approach is presented in (Zheng et al. 2006), where programs are converted to sets of boolean statements representing branching in the program execution. The program is executed many times with different inputs, the values of the boolean statements are recorded and a test oracle is used to determine whether the output is correct or not. The problem is then one of finding those statements that are predictors of a failure, which the work attempts to do via statistical modelling.

In more recent years, machine learning techniques are being used to model source code directly in a fashion similar to the modelling of natural language (Allamanis et al. 2017). This has lead to a number

---

[3] Beizer uses *testing* and *debugging* to refer to detection and analysis respectively.

of breakthroughs, for example, in code summarization (Zhu et al. 2019). In large or long-running projects people other than those who originally wrote the code will be responsible for debugging and maintaining the software. Code summarization may help them get a better understanding of the different system components or source files and potentially help direct them to the fault. There are experimental tools that perform well in code summarization tasks that might be helpful for this, see (Zhu et al. 2019) for a review.

Very recently Large Language Models (LLMs) have shown promise in directly debugging programs from source. Coding assistants, such as the one developed recently by OpenAI (Chen et al. 2021b) are able to generate working code and assist in development. While there was no explicit focus on training the model for debugging, ChatGPT, another LLM, has demonstrated some debugging capability[4]. See Appendix C for an example conversation with ChatGPT and simple demonstration of this ability.

Other applications of machine learning, or statistical methods are used to (partially automate) analysis include: using source code features or metrics (e.g. cyclomatic complexity, file size, etc.) to estimate the chance of failure (Li et al. 2020); a more sophisticated version of this (Ibrahimzada et al. 2022) creates a joint embedding of code and their associated test cases, if they are misaligned then this indicates a fault; test case prioritization aims to choose a subset of test cases to run such that the number of faults detected is maximized (Mirarab et al. 2007); Bayesian reliability testing uses existing test suites and priors to determine the probability of a fault being present *after* testing (Wooff et al. 2002). See (Khaliq et al. 2022) for a more in-depth review.

The automation of both reporting and analysis are important problems, analysis still being somewhat out of reach. In both cases, framing the problem as a learning problem has led to meaningful progress. So far the discussion of automation has been rather general, the techniques for automating reporting and analysis are applicable to video games and most other software. The remaining two steps, detection and verification are different. Video games present a series of unique automation challenges in these steps. Their automation is just as important, they are at least as time-consuming and are where testers expend most of their effort.

2.2.4 Automated Bug Detection

In software testing more broadly, one might still think of automating detection in terms of testing frameworks. The developer defines a series of test cases which should capture the intended behaviour of the program or System Under Test (SUT). The test cases consist of a set of inputs, and guards or rules that evaluate the correctness of the output. The automation here only comes from the automatic execution of these predefined

---

[4] there is debate over the extent of this ability, it appears to be limited in its reasoning, but is impressive nonetheless.

test cases, this is a kind of shallow automation, since everything has to be meticulously specified in advance.

To properly automate detection, there are two key problems that need addressing. The first is about input - *which inputs are likely to uncover issues?* This problem is widely known as the *input generation problem*. The second is about output - *which outputs are correct, and which are not?*. This problem is widely known as the *test oracle problem*.

In some software, the input generation problem might be considered the simpler of the two problems. This is because it might be realistic to explore a substantial portion of the possible execution paths. The problem is then just one of producing inputs to maximise code coverage. Methods of varying sophistication exist for this purpose. Concolic testing (Sen et al. 2005) for example, uses a constraint solver, treating program variables as symbolic to derive inputs that maximize code coverage. Another, crude but effective example is fuzz testing (or fuzzing) (Miller et al. 1990), where random inputs are used. Unfortunately both of these approaches fall short for all but the simplest of video games. Concolic testing suffers from a few limitations, including dealing with non-deterministic behaviour (which is common in video games) and large execution path trees (which is also common). Random inputs are not sufficient as video games often contain puzzles that require very specific input sequences. Although we will see in later chapters that random input generators do provide a convenient way to explore *some* parts of a video game (e.g. in simple navigation tasks).

Automated identification is an exceptionally difficult problem primarily because of the test oracle problem (see (Barr et al. 2015) for a review). In many cases, automation is simply not possible, and developers are required to write test cases themselves. This is fine for software where writing test cases is possible, but there is a class of software, so-called *untestable* software (Segura et al. 2020), where even this is difficult or impossible. Video games happen to be in this class of software, and it is for this reason that in practice they are tested largely by humans.

In the sections to follow, a thorough review of the attempts to automate bug detection in video games is presented. We start with input generation or *search* as it has received most attention.

2.3 Automated Bug Detection: Search

Human testers will spend much of their time playing a game in search of issues. Even at early stages of development, testers will try out new mechanics or attempt to catch early integration issues. This is done not at the level of code, or input/output pairs of some subsystem, but with some graphical output and input from a peripheral device or controller. In doing this, they may be testing only some specific part of the overall system. Often a sustained interaction is required, the tester needs to continuously supply the game with input. This is the same for many other kinds of software such as word processors, browsers, and other GUI based software. The result of each subsequent input depends upon the previous; for video games this is often true to the most extreme degree. With sustained interaction, the problem of input generation is closer to what might traditionally be called a *search* problem, hence our decision to refer to it as such.

2.3.1 Agents & Environments

Outside of testing, interacting with software in the manner outlined is done by software agents. The agent-environment abstraction is a convenient one that allows us to discuss how this interaction might go at a high level. One can think of human testers and users also under this umbrella, allowing us to draw direct inspiration for the development of software agents for the purposes of Automated Bug Detection (ABD).

Agents are essentially sophisticated input generators, where inputs are instead referred to as *actions*. For human testers, the act of pressing a button on a controller provides input to the game. A software agent has a similar virtual controller which links it directly to the game environment, this virtual controller is referred to as an actuator or effector.

Similarly, a human tester will perceive the state of the environment as an audiovisual observation. A software agent has a virtual sensor, which at one extreme might also receive audiovisual input. At the other extreme an agent may observe the result of every program statement akin to Concolic testing; of course there is a trade-off to be made. Observations made at a higher level are coarse grained, one cannot make statements of the kind: `there is a fault at line X`, the benefit however is tractability.

Agents are frequently found in video games, they are often referred to as Non-Player Characters (NPCs); typically small programs that interact with the player in some way. These agents *reside* in the video game environment, this only means that they observe and act within it.

Traditionally, it is the environment (the video game) that mediates the execution and running of an agent. For an NPC this is certainly true, the code that comprises the agent will be run as part of the game loop. The code below is an example of a game loop that mediates the interaction with a human player. The loop is run many times per second, constantly looking for new input, updating the environment's state and displaying it to the player.

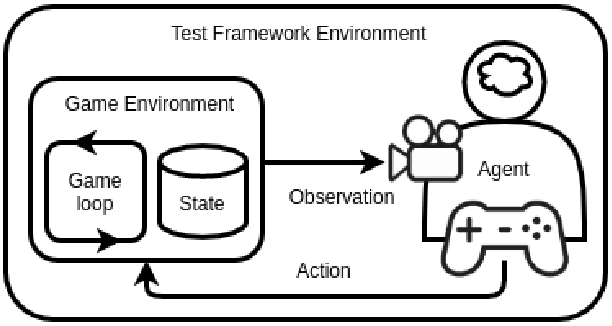


Figure 2.1: Interaction of a testing agent with a video game. The agent resides in the test framework, and takes actions to change the state of the game environment as a human player would. In some scenarios the agent may have many more actions available to it than a normal human player would have, perhaps akin to the actions available to a tester (e.g. with *cheating* enabled). The agent may observe some or all of the game state directly (e.g. positions, object properties, audiovisuals, etc.).

```
while(not action.exit):
    action = input()
    state.update(action)
    state.render()
```

A loop with a similar structure is used to mediate the agent-environment interaction. Testing agents might not be mediated by the video game environment, instead this may be done by a testing framework. This allows the agent to interface with the game in a manner similar to human testers or players. There are some advantages of doing so, one of which is to decouple the agent from the running of the game. Otherwise, if the game crashes, the agent will also crash. The environment therefore consists of both the video game and the testing framework, see Fig. 2.1. However, unless the agent is specifically allowed to take action, or observe parts of this testing framework, we will still say that the agent resides in the video game.

Agents tend to have goals and act to achieve them. A chess playing agent's goal might be to win the games it plays. A testing agent's goal might be to find a particular kind of bug, or simply to explore the game environment as fully as possible, maximizing environment coverage and therefore code coverage. The decisions that an agent makes in pursuit of its goal are produced by a policy. The policy is informed by the observations an agent makes (i.e. its experience), and any prior beliefs it may hold.

To achieve their goals, testing agents must be quite sophisticated in their behaviour. Video games contain puzzles that even human players may struggle with, many of which will requiring high-level reasoning. While the agent-environment abstraction doesn't present a solution directly, it does provide a principled way of thinking about the problem. The input generation problem for software with sustained interaction, like video games, is better framed as an AI problem. As we will see, this is the view taken in the vast majority of works that aim to tackle it.

2.3.2 Explorative Agents

Playtesting with software agents is not a new idea. Traditional approaches use hand-crafted agents to play games (Buhl et al. 2012). These agents are usually fairly simple since crafting complex behaviours is a significant undertaking that may undermine any gains made via automation. Often the focus is on navigation as one of the simpler explorative tasks. Both (Prasetya et al. 2020) and (Stahlke et al. 2020) used path finding agents to reach certain goal states, or locations in 3D environments. In (Shirzadehhajimahmood et al. 2021) more sophisticated planning agents with simple goals specified logically were used to test player-object interactions and level solvability. In these works puzzles were very simple and did not require a sophisticated solver, more difficult puzzles present a major problem for these simpler agents. In an attempt to avoid the issue altogether (Chang et al. 2019) augments human play with agents that play from where the human left off. The game is periodically saved and a policy based on the Rapidy-exploring Random Trees (RRT) algorithm (Lavalle 1998) explores locally around the save point (akin to fuzzing). This simple but effective approach drastically increases the number of states visited during a test, while off-loading the more complex puzzles to the human.

Some approaches try to tackle the problem head on with more sophisticated testing agents. ICARUS (Pfau et al. 2017) is a solver for the adventure games genre. The solver uses a kind of Reinforcement Learning (RL) that remembers which sequences of actions lead to in-game progression. Even with a specific focus on puzzle solving, in some of the more challenging cases humans hints are still required. This is a common theme among approaches, automation is possible as long as sufficient prior knowledge (hints, tactics (Zhao et al. 2021), motifs (Mao et al. 2016)) is provided for the more difficult cases.

Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning that focuses on getting agents to learn behaviours by trial and error. An RL agent aims to maximize its reward (utility) by taking actions in the environment. RL has proven very successful in producing agents that are able to play difficult video games (Silver et al. 2016; Silver et al. 2017; Fuchs et al. 2020; Vinyals et al. 2019). Since the conception of Deep Reinforcement Learning (Mnih et al. 2013), RL agents have been able to tackle increasingly complex games and are now being applied to the search problem. In (Bergdahl et al. 2020) deep RL agents explored 3D environments, their objective was to reach particular goal states in the shortest time. The agents were able to discover unintended routes to pre-selected locations and areas that players will get stuck.

The objective in RL is commonly to produce expert game players e.g. by maximising the game score. In some instances, we can take advantage of the tendency to heavily exploit the environment, as in (Bergdahl et al. 2020). However, the focus of an agent should generally be to explore the game as fully as possible, maximizing environment coverage. To do so an agent must balance solving puzzles, or *exploiting* its knowledge, with exploring the environment. Without explicit motivation for exploration, RL agents can become very narrow in their behaviour as they converge to an optimal exploitative solution. The in-game score does not fully capture the task at hand, agents need to remain curious. With this in mind, works try to incentivize exploration.

One approach taken in (Zheng et al. 2019) aims to do this at a population level by mutating RL agents trained to maximise in-game score and selecting via an explorative objective. Agents are selected based on the average number of states they visit i.e. on their ability to cover the environment.

Intrinsic Motivation

Intrinsic motivation is defined broadly as *the doing of an activity for its inherent satisfaction rather than for some separable consequence* (Oudeyer et al. 2009). Acting out of *curiosity* to explore the environment might be thought of as intrinsic motivation (Singh et al. 2005). Curiosity has been formalized computationally in various ways (Burda et al. 2019; Aubret et al. 2019). It usually involves quantifying uncertainty or novelty of a situation (see (Pathak et al. 2017) for a seminal example in deep RL), although count-based methods are also popular (see next section). In RL, intrinsic motivation is primarily used to address problems with sparse reward, or to allow agents to explore in a principled manner, see Fig. 2.2.

In (Gordillo et al. 2021) intrinsically motivated deep RL agents learn to navigate a 3D environment to uncover progression related issues similar to (Bergdahl et al. 2020). Guided by a count-based objective the agents are able to extensively explore the environment. There is a substantial amount of work on intrinsic motivation for game playing agents which has yet to be applied to search (Roohi et al. 2018).

Principled Exploration

Count-based exploration has a long history in the development of explorative agents, the idea is simply to keep a running count of states that have been visited, and visit those that have been visited less often. The RRT algorithm (Lavalle 1998) mentioned in earlier sections is an example of an efficient count-based method for exploration. It is known that count-based exploration is close to optimal in tabular reinforcement learning

(a) Random exploration (b) Principled exploration

Figure 2.2: Random vs. principled exploration in a simple 2D environment. Source (Aubret et al. 2019).

(Tang et al. 2017), that is, it leads the agent to discover the relevant factors for reward maximization as fast as possible. Many of the methods for principled exploration, such as those based on novelty, information gain or uncertainty can be seen as forms of count-based exploration (Bellemare et al. 2016). Other sophisticated instantiations of count-based exploration such as those based on information entropy (Hazan et al. 2018) might also be used for developing testing agents, but have not yet been explored. These methods are often used to improve learning efficiency when generating agents that act optimally according to some other objective (e.g. maximising game-score). However, they can also be used to generate agents that are *purely* explorative. These agents are of limited use generally, but may be useful in, for example, exploring bounded 3D worlds to search for bugs such as terrain holes.

2.3.3 Human-like Agents

So far the focus has been on agents that play and explore games without much thought for the kinds of problems that a tester might be looking for. A large part of testing video games is concerned with their design. Since design is about creating an experience for the player (Schell 2008), it is important to ask questions such as *how will the player play?* or *how will the player feel when they play?* The more traditional approach to answering these questions is to ask players or testers directly as part of a survey or questionnaire. In (Yee 2006) questions about how players would behaviour in different scenarios were answered as part of a large online survey. This gave some insight into the preferences of players which might inform future design of games in the genre. Of course, the same can be done during development to inform the design of a particular game.

As far as automation is concerned, a questionnaire is a first step but it is not enough. We cannot possibly ask players how their feelings or preferences for every design we might come up with. Instead, we might try to model players and then query the model rather than the players themselves. This is known broadly as player

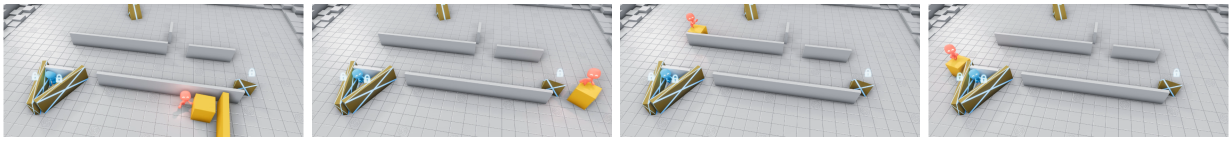


Figure 2.3: Box surfing in Hide and Seek. Source: (Baker et al. 2020).

modelling (Georgios N. Yannakakis et al. 2018). In many cases modelling the player means developing an agent that behaviours in a similar fashion (has the same motivation) as the human. These agents may then be used to answer a range of questions about design, and potentially play and test video games themselves.

There is a relationship between affect (how players feel) and their motivation, and similarly between their motivation and how they play or their *behaviour*. Motivation may be estimated from affect or from behaviour, (Melhart et al. 2019) does the latter. High-level game play features (speed of progress, play style, etc.) are used to estimate motivation factors (competence, autonomy, relatedness, presence (Ryan et al. 2006)) that have been obtained through a UPEQ questionnaire (Azadvar et al. 2018). More recently (Makantasis et al. 2023) takes this a step further and develops an approach to estimating afferent signals from audio-visual footage of in-game player behaviour. The idea is similar to approaches that model affect via audio-visual recordings that show facial expression, tone of voice, gestures, or other verbal or non-verbal queues (see e.g. (Pini et al. 2017)). While the approach doesn't produce human-like agents directly, the trained model may be useful downstream for determining player motivations (as in (Barthet et al. 2021)).

Affect modelling is a powerful way to determine player motivation, which in turn may lead to interesting human-like behaviour. Since the focus of this thesis is primarily in the identification of bugs affect modelling is less of a consideration. As will become apparent, there are many other way to model player motivation using in-game factors alone.

Automated Game Design

Even without considering the players emotional state, there are still numerous functional design questions that would benefit from automation. These range from game balancing (García-Sánchez et al. 2018) to procedural content generation (Smith et al. 2011). In these areas human-like agents may help generate new designs (Isaksen et al. 2018; Delaurentis et al. 2021), playtest new content, or otherwise inform the design process.

In (Stahlke et al. 2020) agents are used to evaluate how players with different motivation profiles might navigate a level. Their visualisations allow the testing team to see which points of interest (collectables, hazards, etc.) are visited, and ultimately whether the design needs to be iterated on. The motivation profiles attempt to capture the variation in human play, this is important for a thorough evaluation of the design as games should cater to the broadest possible audience, both in terms of skill and play style. A number of other works have looked at designing agents with different goals (Stahlke et al. 2020), play styles (Keehl et al. 2018) or *personas* (Holmgård et al. 2018). In these works the problem of designing human-like agents is reduced to determining heuristics, in the form of utility functions that seem to capture human play styles. The goal of these works is less about getting realistic human behaviour and more about getting a broad view of how a game may be played while keeping the key design features in mind. In (Mugrai et al. 2019) for example, Monte-Carlo Tree Search (MCTS) and a genetic algorithm were used to explore the design of levels in matching-tile games.

Another problem in testing design is to find those mechanics that can be used (or abused) in unexpected ways, usually for some unintended advantage. Algorithms such as those used in RL are known to exploit such mechanics. In (Baker et al. 2020) agents play hide-and-seek, they learn to move boxes and ramps, locking them in place to create shelters from the opposing team. After a long training period, the opposing team finds the *box surfing* strategy, where they can simultaneously stand on and move a box, see Fig. 2.3. This allows them to access otherwise impenetrable shelters from above. The mechanic was an unintentional side effect of the physics implementation and was unknown to the researchers before being highlighted by the agents. There are many other examples in the literature (see (Lehman et al. 2019) for a review).

Although this is clearly a powerful approach to finding exploitable game mechanics, it is important to note that, as with the majority of methods to search outlined thus far, the agents themselves do not realize (unlike a human tester) that they are abusing the mechanics. To them, this is just another part of the task at hand.

Automated Game Balancing

Game balancing is a facet of game design that focuses primarily on designing games that are *fair*, or have the right level of difficultly (Becker et al. 2020). Automated game balancing attempts to address the problem of finding those game mechanics that lead to disproportionately strong (or weak) player strategies. For single player games this is often about difficulty, a game needs to strike a balance between guiding a player and not restricting them or treating them like an idiot (Bainbridge et al. 2007), the game should be challenge but not be too challenging. For multiplayer games it tends to be more about how players will interact with one-another through the game mechanics.

In (García-Sánchez et al. 2018) an evolutionary algorithm was used to create decks for the popular online competitive card-game *Hearthstone*. The algorithm was able to identify specific cards that lead to a win-rate imbalance by monitoring card usage statistics. An expert human was required to evaluate the cards found as it was observed that the game playing agents seemed to prefer using some cards over others even where there was no imbalance (according to the expert). Others take approaches based on learning (Pfau et al. 2020; Chen et al. 2020a), take a game-theoretic approach (Volz et al. 2016), explore game variants by perturbing parameters (Isaksen et al. 2018), or otherwise explore balancing in single player games (Silva et al. 2018; Shin et al. 2020).

Learning from Demonstrations

Learning from demonstrations, or by *imitation* is one class of approaches for developing human-like agents. Agents learn from *expert* demonstrations, giving rise to behaviours that are similar to the expert's (Schaal 1997). A demonstration is a collection of games states or observations (e.g. the audiovisual presentation) and associated actions.

Behavioural cloning is imitation learning in its simplest form. It is a supervised learning approach that aims to copy the actions of the expert, relying on the generalization capability of the underlying model for situations that were not given in demonstrations. Modern behavioural cloning happens via maximum likelihood estimation, essentially minimizing the difference of action probabilities of the agent and expert. Behaviour cloning has been used to bootstrap agents for playtesting (Gudmundsson et al. 2018) and exploration (Zuo et al. 2022). The inherent explorative capability of behavioural cloning is generally quite poor since it just copies expert actions. Performance can also suffer if during play the agent encounters states that diverge from what was seen in the demonstrations. Naive imitation learning is therefore only useful in some circumstances (e.g. for difficulty testing), or when used in combination with other approaches. Improvements over behaviour cloning have been developed, and some have been applied to video game testing. In (Sestini et al. 2022) for example, the DAgger algorithm (Ross et al. 2011) is used to validate design changes in 3D game environments.

Instead of attempting to directly clone the expert's policy, Inverse Reinforcement Learning (IRL) aims to model the expert's reward function, trying essentially to capture the motivations of the expert. Recently (Sinan et al. 2019; Ariyurek et al. 2020) made the first attempts at applying IRL to playtesting. They were successful in finding bugs in some simple grid-world environments by combining IRL and MCTS. IRL is a notoriously hard problem and computational resources are quickly exhausted as environments become more complex. Still, there have been many advancements in this area which have yet to be applied to automated search (Ho et al. 2016; Finn et al. 2016; Ziebart et al. 2008).

More recently there have been efforts to combine affect modelling with reinforcement learning to produce more human-like agents. In Barthet et al. 2021, Go-Blend integrates affect modelling with RL. Rather than learning from demonstrations as collections of states and actions, a Go-blend agent will try to model the players motivation using afferent signals like those discussed earlier (survey results, facial expressions or otherwise). Motivation comes in the form of an axillary reward derived from the affect signal which is combined with the usual RL reward signal. Go-Blend lies somewhere in the intersection of IRL and RL. It has the flexibility of RL and avoids the key problem that IRL aims to address (learning motivation directly from behaviour) by leverage affect signals.

One of the biggest drawbacks of approaches that require demonstrations was raised as a concern in a survey given to game developers (Sestini et al. 2022). It is that these demonstrations need to be generated by a human, clearly this is also true of approaches that requite affect signals[5]. While producing demonstrations is not as time-consuming as full-blown playtesting, it does put a load on testers. As always, there is a trade-off to be made, we must be able to specify the desired behaviour and providing demonstrations requires far less expertise than designing a reward function, or hand crafting a behaviour. Quite often imitation learning and reinforcement learning are combined, doing so here could drastically reduce the number of demonstrations required (see for example (Judah et al. 2014)), but at the cost of requiring a suitable reward function, which may not be easy to specify.

Playing like a Tester

An agent that plays like a human may allow us to find issues with a game's design, but the kinds of bugs that testers are interested in extend beyond questions of how humans might play. Instead, at the other side of testing, the goal is to ensure that software is functional. That it doesn't crash, that there are no obvious graphical artefacts, actions can be taken when possible, in-game interactions work as intended, the game is *solvable* (i.e. the goal can be reached), and many more. Of course, these requirements are inseparable from the design questions we have already looked at, but they don't *require* human-like agents in order to be tested, a principled explorative agent may work just as well.

That isn't to say that human-like agents wouldn't be useful here. Agents that behave not just like players, but like *testers*, would actively search for bugs while playing instead of just happening upon them

---

[5] this is unless a very general affect model can be produced, which has its own substantial challenges.

during normal play. The play-style of an experienced tester will likely be heavily dependent on the task at hand, for example, when looking for collision issues they might look for ways to increase their velocity and smash into unusual geometry at increasingly high speeds. This kind of testing is known as *experience-based testing* (Marselis et al. 2014), as distinct from *coverage-based testing* which tends to better describe explorative software agents. Whether the tester is white-box or black-box will make a big difference to their behaviour for certain bugs. In terms of automation, experiential white-box testing is currently out of reach. Experiential black-box testing on the other hand may be a more realistic goal. Note that an agent does not necessarily need to know about implementation in order to direct their behaviour as long as there are patterns in the way that certain issues manifest that can be exploited.

Developing agents that behave like human testers has not been explored in the literature beyond simple hand-crafted behaviours, or those that mimic normal human play in the ways outlined previously. Although clearly very challenging, it is a direction that deserves more attention. Any meaningful attempt will need to address the test oracle problem to some extent. An agent will require some knowledge of the kinds of bugs that it may encounter, and crucially, how to manifest them. It can only do this if it has some means to distinguish between normal observations, and buggy ones.

In the vast majority of the works we have encountered thus far, addressing the test oracle problem has not been a consideration. Instead, any issues are either identified subsequently by a human in their analysis (especially in testing design), or by guards that have been written as part of the code, i.e. issues that manifest as crashes or are indicated by exceptions. As mentioned earlier, this is a problem for those bugs which are difficult to write test cases for, examples of which we will see shortly.

2.4 Automated Bug Detection: Identification

The problem of identification can be boiled down to the following question:

> *When a tester encounters a bug, how do they know?*

Previously it was stated that it is just a matter of comparing the intended program behaviour to the observed program behaviour and that if there is a mismatch then there is a bug. But how does the tester know what is intended behaviour and what isn't? An immediate answer might be along the lines off: *well, from the requirements*, or some specification that informally (or formally) outlines them. Even putting aside the issue of creating machine interpretable specifications, this answer is less than satisfactory. A specification cannot possibly outline the intended behaviour in its full glory. If it did, there would be no problem in the first

Figure 2.4: A horse doing a handstand in The Witcher 3, bug or feature?

place, since the specification would be essentially equivalent to a perfect (bug-free) and finished product. In practice, human testers work from informal guidelines and need to fill in the gaps using common sense, or by drawing on their experience. Consider the bug presented in Fig. 2.4. It is unlikely that testers were told explicitly *horses don't do handstands*, yet it is clear to the tester that this is a bug as, in context, it just doesn't make sense.

To begin addressing, or at least get a handle on the problem, we might try to fall back on the closed world assumption - what is not currently known to be true, is false. Applied to the hand-standing horse, if the agent doesn't *know* whether the horse can handstand, it had better assume it can't. With the closed-world assumption, the agent would need to know *everything* that a horse can do. Despite this being quite obviously impractical in some instances, it is ubiquitous in developing tests for software. That is, to write a series of guards that attempt to explicitly capture the intended behaviour. For this reason, it is worth reviewing the simpler guard-based, and subsequently the more sophisticated model-based approaches, which as we will see suffer from similar problems, as both can otherwise be very effective.

2.4.1 Model-Based Testing

Crashes are the result of constraint violations, or guards failing, meaning that a developer expected the issue and wrote code to catch it. Writing guards in code is the most primitive form of automated identification. In fact, some might be inclined not to call it automation at all. Perhaps when they think of automated identification they instead think of systems that can *produce* guards (i.e. solve the test oracle problem), rather than merely evaluate them. But to push back, handwritten guards are made to be repeatable (and hopefully reusable) and are evaluated without human involvement, thereby reducing the need for human input - the essential principle of automation.

In model-based testing or identification, a model of the software (or video game) is developed using some (semi-)formal language. A wide range of languages have been used, from UML (Offutt et al. 1999) and XML to Video Game Description Languages (VGDL) (e.g. (Schaul 2013; Quinones et al. 2020)). Generally, the aim is to make writing guards and test cases easier by providing the means to write a specification (the model) at a higher level, for example, at the level of objects, events, processes or agents. Then, rather than using the model directly as a set of guards, or constraints on the system, models are used to produce guards through a process of constraint solving or otherwise (Rushby 2008). This takes something specified at a high level and makes it applicable to testing lower-level code execution.

To give some examples of model-based testing in video games: in (Smith et al. 2010) games are represented and implemented using event calculus, a logical formalism that tracks the truth value of predicates through time. Logical constraints are specified on these truth values and violations are found in the process of solving, in doing so the player behaviour must also be modelled in logic. While good for testing the implications of different mechanics one might want to include in a game, the system doesn't actually test a video game that has been implemented independently of the logical solver.

In (Varvaressos et al. 2014) a run-time framework that monitors the game state as it is played is developed. In-game events are represented in XML, and logical constraints are evaluated on incoming streams of these events. In (Hernández Bécares et al. 2017), beta test runs performed by humans (or in principle by agents) are recorded, replayed and verified using Petri nets as the modelling language of choice. In (Radomski et al. 2015) game interactions are modelled as a state-machine and Linear Temporal Logic (LTL) is used to formally verify the interactions. (Ferdous et al. 2021) uses a model-based approach to generate action sequences (i.e. addresses the search problem). (Iftikhar et al. 2015) uses UML as the language of choice to test simple platform games, including their GUIs.

Metamorphic Testing & Untestable Software

The model-based testing literature for software generally is vast, but the portion concerning video games is comparatively small. Model-based testing is limited in its applicability here because, along with software like search-engines, compilers, or machine learning systems, video games are examples of *untestable* programs (Segura et al. 2020). This is highlighted by the non-trivial nature of identifying the hand-standing horse as an issue, there simply isn't a concise set of guards that describes the intended behaviour of a horse. The reason the approaches mentioned are reported to work is that they do not operate in the full domain, ignoring for example, the audiovisual aspect of video games. That isn't to say that model-based testing has no place in video game testing automation, it just isn't the final solution.

Metamorphic testing (Chen et al. 2020c) was developed to deal with instances where it is difficult to explicitly specify a test oracle, or a model of input-output relations. The idea is to use the program as its own test oracle by comparing multiple input/output pairs. To give a simple and helpful example (see (Segura et al. 2020)), consider the function `merge(L1,L2)` that takes two lists of integers and merges them into one ordered list. Since the output list is ordered, the following metamorphic relation holds: `merge(L1,L2) == merge(L2,L1)`. If this relation fails for some pair of input lists, then we know the function has an incorrect implementation. As in model-based testing, test cases can be generated from metamorphic relations. One could potentially derive similar relations even for input/output pairs in video games (e.g. between an agent's actions and resulting observations). Metamorphic testing has been used to verify agent behaviours, for example in autonomous driving (Tian et al. 2018), but appears yet to be explored in video game testing. It is a very powerful testing technique if these relations can be specified, which like for model-based testing may be challenging. We will revisit this idea again in chapter 6.

Model-based testing reduces time spent writing tests and error-proneness, but it does not address the issue of untestable programs, where collections of guards are hard to determine. Guards that are specified in a logical language (e.g. as collections of boolean statements), whether generated automatically or not, are too brittle to be applied to more complex cases such as the horse, where it is not totally clear what constitutes intended behaviour. What is needed is an approach that can represent intended behaviour in a more flexible fashion, one that is more practical to specify and that can to some extent resolve ambiguity as a human tester does.

2.4.2 Intelligent Bug Identification

For a system to resolve ambiguity around what constitutes intended behaviour *seems* impossible, as presumably it would be required to already know the intended behaviour without us having specified it. But this is only if we take an extreme position, after all the quintessential test oracle - the human tester, does not have a complete picture of the intended behaviour. Some *gap-filling* from common sense or otherwise is required. A softer view on the problem, where we trade off certainty for flexibility is the one we should take. To be clear, I am advocating the use of machine learning and its aptitude for soft decision-making, and leveraging *data* or *experience* as a source of additional information beyond what has been explicitly specified. To better ground what is being said, a parallel can be drawn with the problem of specifying agent behaviours.

To create an intelligent game-playing agent, we might try to exhaustively specify decision rules. Anyone who has attempted to do this in any setting that exhibits even remote complexity will testify to this being a fool's errand. Instead, we can and should give an agent a high-level goal and rely on a sophisticated algorithm to derive these decision rules for us. Initially, we might draw a parallel between model-based testing and planning. Planning takes a goal or utility function of some kind, and a specification (a model) e.g. of how the agent affects its environment, and produces a collection of decision rules (in whatever format) that enables the agent to achieve the goal. What I am advocating is something akin to reinforcement learning, where an agent effectively produces a model of its own accord, through experience guided by a high-level objective or reward function.

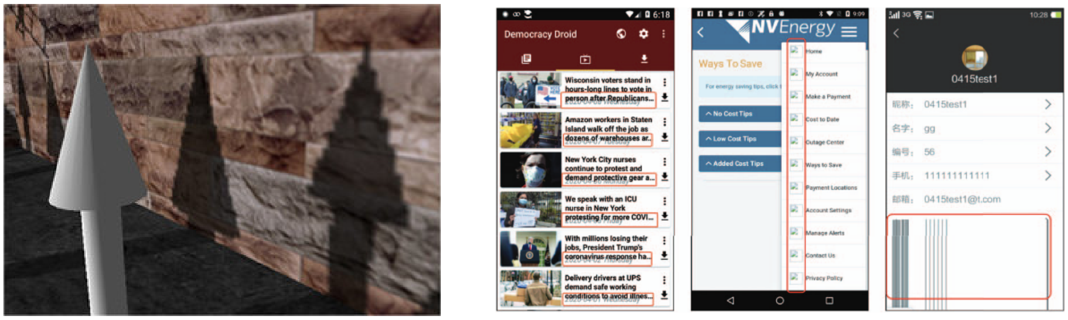
In either case, it's not as if we have avoided the problem of specification altogether. For this to work, we had better be sure that the combination of the agent's utility function and decision-learning algorithm reflect what we would otherwise be trying to achieve by writing the decision rules ourselves. In the language of machine learning, this means to give the agent the right *inductive bias*, and to *align* the agent with our goals. Doing so is far from straightforward, but if done correctly can produce extremely sophisticated behaviours; recall the hide-and-seek agents in section 2.3.3. The hope is that there is some analogous process for bug identification. For the purposes of this thesis, this kind of automation is referred to as *intelligent automation* or just *automation*. We refer to systems that perform this kind of automation as testing agents, or simply agents[6].

Identifying Simple Graphical Bugs

A class of issues where learning has already been applied in this way is in testing graphical user interfaces (GUI). Rather than examining code, approaches in this class consider the screen as it is presented to the user. In one of the first examples (Liu et al. 2020) of a serious attempt at using supervised learning to identify GUI related bugs such as overlapping text, UI component occlusions and missing images (see Fig. 2.5.a). An agent is trained to distinguish bugged UIs from normal UIs; UIs are taken from a large collection of software projects. When presented with a novel UI, it is able to identify the kinds of bugs that were seen in the training data. The caveat being that the training data must be extensive enough to *cover* new interfaces. If a UI is novel or innovative the approach may fail. Still, the learner has effectively produced guards specific to these bugs which are cross-project without being *explicitly* told how. This is a drastic improvement over previous work that used hand-crafted machine-vision approaches e.g. (Mozgovoy et al. 2018) which are akin to the usual guard-based approach.

---

[6] In related work these systems may not in-fact be agents (in the usual sense), we still refer to them as such to avoid complicating terminology.



(a) (b)

Figure 2.5: (a) An example of perspective aliasing, a kind of shadow artefact. Source: (Nantes et al. 2008). (b) Examples of different simple UI bugs on mobile devices. Source: (Liu et al. 2020)

Identifying Rich Graphical Bugs

While the work on identifying bugs in simple user interfaces is encouraging, in terms of automation solutions, there is a chasm separating simple UI issues from the graphical issues that are encountered in video games. The difference in richness of the presentation, variety of potential issues and differences among the games themselves are substantial problems. Without vast amounts of data, we could not expect a learner to generalize well across projects or to new games.

Nevertheless, with some concessions in generality there have been some attempts that aim to identify richer graphical problems in video games. In (Nantes et al. 2008) the authors attempt to identify the perspective aliasing bug, see Fig. 2.5.a, using a traditional vision based method. The method doesn't generalize to other shadow related artefacts, but it is somewhat reusable across projects.

In (Nantes et al. 2013) a learning-based approach to identifying a variety of graphical bugs, including geometry and texture corruption is proposed. The approach is to learn object descriptors, or summaries of colour and geometry information that are extracted from the graphics pipeline through various means. They are able to identify the bugs they consider by comparing what is actually rendered to the model's prediction of what should be rendered. If there is a substantial difference then the render is flagged as having a potential issue. There is limited discussion around obtaining training data, other than to say that it can be generated directly from the game by modifying various properties. This in essence means they have access to a test oracle for the game in question, which makes the approach somewhat impractical (except in regression testing for example).

Since exploring the ideas presented in this thesis, and at the time of writing, (Taesiri et al. 2022) emerged as work that does leverage vast amounts of data for this problem. The authors take advantage of the recent advancements in large language-image models. These models relate natural language captions with images and are trained on very large datasets. The work makes use of CLIP (Radford et al. 2021), a model trained on approx. 400 million image-text pairs gathered from the web. They use the model to do zero-shot bug identification by embedding images of bugs taken from real games and comparing this embedding with one of text describing a kind of bug (e.g. *a car flying in the air*). Despite never being trained on data that comes from video games[7] the model was reasonably successful in identifying bugs in the games that the authors tested. This work could be seen as a very sophisticated version of model-based testing, where rather than writing guards the traditional way, they are written in natural language. This doesn't solve the traditional test oracle problem as the method cannot identify issues that have not been given in the form of a textual prompt, but it is clearly a step in the right direction.

To my knowledge the only work to date that makes use of learning in an attempt to identify graphical bugs, other than those presented in the later chapters of this thesis, is (Chen et al. 2021a). However, upon further inspection there are some concerns about their dataset and training procedure, which so far the authors have not responded to.

The challenges surrounding data, training, and how this all relates to the test oracle problem are discussed in-depth in the next chapter. Before this, we return to the last step of testing - verification, and its automation.

2.4.3 Automated Verification

Verification can be thought of as a version of the identification problem. After a bug has been fixed, a tester simply needs to verify that the claim is true. To do so, the tester requires some knowledge of the intended behaviour, and of the issue that was supposedly fixed. Given that the bug was identified previously, presumably both of these are known. This is unless the bug was found by a human, and verification is done in some other way (via automation).

These requirements are the bare minimum for verification and may not always be enough. Changes that fixed the bug may influence other parts of the game. In such a case, it may not be as simple as repeating whatever was done to detect the bug in the first place. An agent who is to verify a fix must not get confused by any new (or absent) experience even if this experience violates its model of intended behaviour; it must have a way of distinguishing between the bug of interest, and any other bug that may have been introduced. For the most part, the problems faced in automating verification are also the ones faced in identification.

---

[7] there may have been some examples in the training set, but not enough that it would be meaningful to say so.

And so further discussion is deferred to the next chapter.

2.5 Summary

This chapter has explored the broader context of automated video game testing, presented a thorough review of the literature, and outlined at a high-level the central problems in automated bug detection. In journeying through the various kinds of testing automation that has been done to date in video games, the following observations were made:

1. Software agents can and have been used to automate one of the most time-consuming problems in video game testing - game-playing, and have benefited greatly from the recent advancements in AI and machine learning.
2. These agents are largely unable to identify bugs themselves, relying on humans to analyse the resulting data, or on guards implemented in the game to flag issues when they arise.
3. Automated testing of video game *design*, especially concerning playability will likely continue to require human post-analysis for the foreseeable future.
4. Current testing agents do not leverage knowledge of the bugs they may encounter to direct their behaviour as human testers do, this makes them inefficient, and at worst, ineffective testers.
5. In testing functionality, traditional guards are too brittle or otherwise insufficient for many of the bugs exhibited by video games.

These observations have led to the following conclusion: progress on the *test oracle problem*, or equivalently, *bug identification* as it has been presented, is needed if further progress is to be made. And, for the most straightforward gains, functional problems are where attention should be directed first. It seems plausible that the reason we have not yet seen agents that can learn to direct their behaviour as human testers do is because there has not been sufficient progress in automating bug identification.

Taking steps towards the automation of identification is the focus of subsequent chapters. In the next chapter an exposition of the bug identification problem from both a practical and theoretical perspective is given. Machine learning, and specifically *anomaly detection* is the lens through which the problem is viewed. That is, identification as the recognition of unusual or abnormal events in an agent's experience, where abnormality is specified at a relatively high-level.

Chapter 3

Learning and Bug Identification

In this chapter we will greatly expand upon the initial presentation of the bug identification problem in the previous chapter. The focus is on the central research problems that surround learning models of intended behaviour from experience, and motivating this as a potential approach to addressing the test oracle problem. Our stance is that *anomaly detection* as a field of study suitably encapsulates the problem, and that many of the ideas developed are directly applicable to bug identification. The most important of these is statistical normality. A large part of this chapter is dedicated to determining whether the identification of bugs can be framed as a problem of identifying statistical anomalies, and if not, what else is required. During our exploration of these ideas, the possible learning and testing paradigms that may be followed are discussed. Among them is the identification of novelty for regression testing, which is the paradigm followed in later chapters.

This chapter is structured as follows. In section 3.1 we try to address concerns that experienced game testers may have about the use of learning in testing. Then in section 3.2 a formalism of video games and bug identification is given, which includes relevant concepts and notation. Bug identification is then explored under the assumption that intended behaviour is known to the testing agent. This gives a picture of the ideal scenario and permits a precise definition of the term *bug*. In section 3.3 the fundamental difference between expected behaviour and intended behaviour is highlighted. This leads to a discussion of *experience* in the context of various learning paradigms and testing methodologies. Our discussion then turns to *normality* in section 3.4. More specifically, how we might align expected and intended behaviour given notions of normality that are derived from the statistical properties of an agent's experience. The problems inherent in this are discussed at length, including what we term *systemic faults* (faults that commonly manifest in an agent's experience), unlikely intended behaviour, and the biases that are introduced during the collection of training data. The chapter is summarised in section 3.5.

3.1 Principles of Testing

Aside from the obvious question of how the test oracle problem might actually be addressed by learning from experience (which is the primary focus of this chapter), the reader who is experienced in software testing may have some concerns about the implications of using learning for testing. These concerns are likely related to one or more of the following:

- **Correctness** - whether a test gives the correct or *intended* result.
- **Reliability** - whether a test always gives the same result.
- **Maintainability** - whether tests can be easily be updated when required.
- **Efficiency** - the resources required to run or generate a test.

Some of these concerns are warranted, but others, those regarding correctness for example, might come from a picture of testing as a process that is meant to provide *guarantees* about whether the software behaves as intended. We should remember that the motivation for using learning lies in identifying bugs which are difficult to pin down, and that its primary use-case is playtesting which is generally explorative. With that said, the following sections aim to address any concerns that may remain.

Test Correctness

We do not want to be in a situation where a test fails when code is working (a false positive) as this may lead to wasted time looking for an issue that doesn't exist. Or worse, to pass faulty code (a false negative) as this could have serious downstream consequences. This applies in both playtesting and other forms of testing (e.g. regression testing).

One of the major concerns regarding correctness is that we cannot always be sure that an agent's decision boundary corresponds exactly with the test goal. This is unlike in traditional testing (e.g. with guards) where it tends to be relatively straightforward to check whether a particular guard (or test case) is aligned with the test goal; it might be done just by inspection. This cannot be said of a black box (e.g. a neural network) where the test result ultimately depends upon factors that are difficult to control for (e.g. the content of the training data or the training process itself). We could go down the route of *testing our tests*. Testing machine learning systems is a very active research area (for alignment (Gabriel 2020), robustness (Sehwag et al. 2019), safety and security (Goodfellow et al. 2015)). But bear in mind that learning systems are also examples of untestable software and so we are subject to the same problems. The reason we are using such a system in the first place is that it was difficult to formally specify what is intended, so we may end up in a regress.

Another problem regarding correctness is that, unlike guards which tend to give a definite pass/fail, a agent may give a *soft* outcome: *probably pass* or *probably fail*. These outputs will ideally reflect the agent's confidence or uncertainty in the result. In terms of traditional testing this presents a problem: what are we to do with cases where the model is unsure about whether there is a problem say if we run a suite of tests? This is an open question, but work in areas such as uncertainty quantification (Brando et al. 2018) may yield some insight into whether such test results should be ignored or not (e.g. by quantifying different kinds of uncertainty).

If we are really concerned about correctness, an agent might be better used for playtesting rather than non-explorative testing, where we may not be as distressed if a problem is missed. In explorative testing finding any bug at all is a win. Regarding uncertainty, we may just take time to inspect the results for which an agent is most confident. Still, the overarching question is one of trade-offs, what do we get in exchange for accepting uncertainty both in alignment and in test results? If in an automated fashion, a test is able to identify issues that would otherwise be missed because we could not specify the required guards or model, thereby increasing overall system correctness, then perhaps we should be willing to accept some uncertainty. Whether there are an unmanageable number of false positives or false negatives can only be determined in practice. And whether learning-based approaches will offer any improvement in overall correctness remains to be seen.

Test Reliability

A test is reliable if we trust that it is going to give the same result every time. That is, if we run the same test on the same input, it should give the same test result (pass or fail). In the context of non-explorative testing, if the agent is retrained, which may need to happen when the game is updated, then it may be that it produces a different answer for test cases which are concerned with unchanged parts of the game. Problems such as *catastrophic forgetting* will need to be dealt with for example. Again, the problem is less of a concern in playtesting. The hope is that after retraining the agent will be better at finding issues in the

Figure 3.1: Collection of horse related bugs in the Witcher 3. A sufficiently capable agent might be able to identify all of these issues if it has learned what is *normal* for a horse.

updated environment.

Test Maintainability

As software is updated, tests may need to be updated. A general principle followed when writing tests is to keep *scope* to a minimum, a test should be for some specific functionality. Tests that are self-contained (have no external dependencies), are modular and have a narrow and well-defined test goal are easier to maintain (update when necessary), tend to be less error-prone and can be more easily verified for correctness.

Testing agents and learning-based bug identifiers are the antithesis of these principles. They are going to be complex, have external dependencies, potentially have broad test goals and are not modular in the same sense (by design). Regarding test scope, one of the reasons learning might be useful is for creating general-purpose tests that can identify multiple bugs, even bugs that were not anticipated. Consider Fig. 3.1, it is conceivable that a single test (or agent) could identify all of these issues, for example by training on normal horse behaviour. This is a fundamentally different kind of testing, and it should be treated as such.

As in traditional testing, an agent may still need updating after changes to a game, perhaps in a new version flying horses are allowed. A good principle for developing learning-based tests is to leave as much as possible up to experience. This is because it is generally much easier to retrain an agent than it is to completely redesign it by changing built-in biases. This principle also enables and encourages reusability, one of the major potential benefits of using learning for testing.

Efficiency & Cost

Regarding efficiency there are at least three significant considerations. The resources required to: (1) design a testing agent with the required learning capabilities; (2) collect data and train the agent; (3) run a test. The hope is that, at least for certain classes of issues, general purpose agents can be designed and applied across projects. The initial cost of designing these agents is going to be high, especially as much of the required research is yet to be done.

Regarding training, there are (at least) two possible scenarios as we shall see in later sections. (a) Agents are pre-trained on large datasets that are representative of issues we wish to identify. This is an investment into future use, with a very high initial cost but with potentially broad applicability and lesser requirement for data from the particular Game Under Test (GUT), fine-tuning might be required but this will be less resource intensive. (b) Agents are mostly trained on the GUT. This is the more expensive long term option, but unless there are substantial advancements in developing game playing agents, there is likely going to be a long training period anyway. The data collected during the training of game playing agents might also be used to train a bug identifier. The high resource requirements for training agents for game playing is one of the major hurdles of automating testing with learning. For reference, in training the AlphaStar (Vinyals et al. 2019) agent to play StarCraft II (which is about as complex as it gets), it was estimated that 200 years of game play was needed, which equated to 44 days of training on high-end processors with costs running into millions of dollars, not including the cost of the research itself. To put this in perspective, a talented and dedicated full-time human player will take at least a year to reach master level.

The setup cost of learning-based methods for bug identification (and search) is high, but assuming this is outsourced and these agents are provided as a service to game development studios, how expensive would it be to run tests? Again the answer is likely more than if the tests were handwritten guards. How much more expensive depends on what is being tested, but the difference is unlikely to be prohibitive. Cost should be a concern, but the potential benefits to overall software quality and ultimately the player experience makes research into this kind of testing worth pursuing.

Summary

The use of learning in testing may require a shift in attitudes. Test results (pass/fail) may become softer, testers and developers will need to be aware of the uncertainty associated with test results. Although learning-based tests are reliable in the traditional sense, developers must be aware that retraining after an update can alter the decisions of an agent in potentially unexpected ways if the proper considerations are not made. To encourage reusability and get the most out of learning-based approaches to testing, developers should leave as much up to experience as possible, or equivalently, ensure the agent is as general as possible. This goes against some long-standing testing principles (e.g. modularity and narrow test goals), but for good reason. A cost-benefit analysis of learning-based approaches is required, but not yet possible. It remains to be seen whether the benefits discussed are worth the additional cost.

With these secondary concerns out of the way for the moment, we return to the central theme of the thesis. We begin in the next section with a formal presentation of bug identification under the assumption that intended behaviour is known.

3.2 Formalising Bug Identification

In this section the formalisms that are followed in the remainder of the thesis are presented. The video Game Under Test (GUT), and the agent-environment interaction are formalised as a Markov Decision Process (MDP). The term *bug* is made precise by defining faults as MDP non-equivalences. The fundamental limitations of the ideal testing agent with full knowledge of intended behaviour are also highlighted.

3.2.1 Video Games as Markov Decision Processes

The agent-environment pseudo-formalism presented thus far is a convenient explanatory framework, but it does not provide a means for deriving any solutions to the Automated Bug Detection (ABD) problem. For this, a more rigorous mathematical framework is required. Perhaps the most famous and widely used formalisation of games, at least in the domain of machine learning, is the Markov Decision Process (MDP). The term was coined by Bellman in the 1950's (Bellman 1954; Puterman 1994) while studying decision-making problems. They were first studied in the context of stochastic games around the same time (Shapley 1953). Since then, MDPs have laid the foundation for many theoretical and engineering successes, including the successes in developing sophisticated game playing agents (Mnih et al. 2013; Silver et al. 2016; Vinyals et al. 2019; Berner et al. 2019).

As the discussion is centred around identification, reward, along with other theoretical devices such as discount-factors and belief states are omitted. This perhaps brings the formalism closer to (non-deterministic) Finite State Machines (FSM), which are also a popular formal model of software (Andrews et al. 2005).

Each playthrough, also called an *episode* or *trajectory* τ is a sequence of alternating observations and actions $x_0, a_0, x_1, a_1, \cdots, x_{T-1}, a_{T-1}, x_T$. An observation $x \in \mathcal{X}$ is some typically lossy and possibly stochastic transformation $\mathcal{O} : \mathcal{S} \to \mathcal{X}$ of the full environment state $s \in \mathcal{S}$. The evolution of an environment is governed by a stochastic transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. At each time-step, the agent takes an action $a_t \in \mathcal{A}$ after observing x_t to produce its next observation x_{t+1}. For stochastic Markovian environments, states are distributed according to the following conditional probability $Pr(S_{t+1}|A_t = a_t, S_t = s_t)$. Despite most software being inherently deterministic, the inclusion of stochasticity in the formalism is a necessary convenience that permits the modelling of pseudo-random processes without needing to explicitly consider their state. In our discussion the term *experience* is often used, it refers to an observation, or part of it, or any collection of observations with or without the associated actions. Experiences may be abstractly represented by an agent i.e. they are not restricted to *raw* sensory input.

3.2.2 Formal Verification: A Perspective

In chapter 2, we briefly reviewed model-based testing in the context of video games. Model-based testing derives from formal verification[1], which is the process of rigorously and mathematically (i.e. with guarantees) testing algorithms or software (Radomski et al. 2015). In order to formally verify a program, a specification of intended behaviour is required (or at the very least, a list of formal constraints that must be satisfied). As examples of untestable software, this is not generally possible for video games in practice. Nevertheless, by framing testing as a problem of formal verification, subsequent analysis will highlight the theoretical limits of approaches to identification, including the limits of human (black-box) testers. Rather than focusing on how intended behaviour is specified, for the moment we will assume that it is given and see how far an agent might get in identifying bugs. Our goal is not to solve the formal verification problem, the analysis is purely theoretical as an in-principle perspective to inform later discussion.

The formalization of a video game as an Markov Decision Process (MDP) (without reward) gives us a starting point. To formally verify that a video game works as intended, the MDP that corresponds to its most up-to-date implementation G is checked for equivalence with the MDP that corresponds to the intended behaviour G^*. The entire software development process can be thought of as iteratively bringing G *closer to* G^*. Assuming the same state space and action space (i.e. same transition function domain and codomain), non-equivalence of the MDPs will come from non-equivalence of their transition functions as outlined in Fig. 3.2. For a particular state and action, any such non-equivalences may be called faults. Faults are categorised into: (A) *distribution mismatch*, (B) *unintended state* and (C) *missing intended state*. (B) and (C) are just special cases of (A), but indicate that the image (or support) of the transition function for a state differs

---

[1] not to be confused with verification in the four steps of testing (i.e. detection, reporting, analysis and verification).

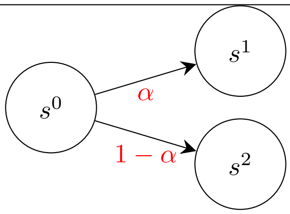
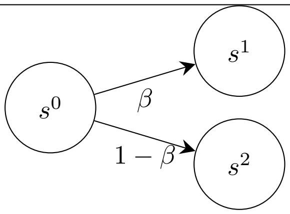
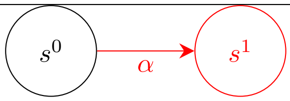
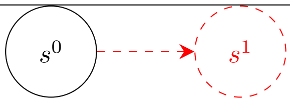
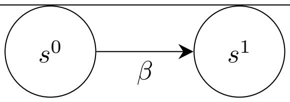
Type	G	G^*
(A) Distribution mismatch $\alpha \neq \beta$		
(B) Unintended state $\alpha > 0$		
(C) Missing intended state $\beta > 0$		

Figure 3.2: Types of transition faults. Each graph shows the possible transitions from some initial state s^0. Each edge has an associated probability. Red indicates the fault, dotted lines indicate missing states or transitions. Both (B) and (C) are special cases of (A). Actions are not shown for simplicity of presentation, but we should assume that the agent's policy is fixed across G and G^* and that all actions have a non-zero probability of being taken.

in G and G^*. Checking for equivalence between G and G^* is in general not straight forward[2] but serves to guarantee the correctness of G.

An agent that has access to G^*, but only to G through Monte Carlo sampling of the game environment (i.e. actually playing the game over and over), is akin to a human tester who has an understanding of the intended behaviour. Such an agent is limited in a number of ways as outlined below.

Sampling Limitations

The first and most obvious limitation such an agent faces is the potentially large number of samples it may require to determine the equivalence of the transition distribution for each state. Fortunately, in video games the vast majority of state transitions are *predictable*. One can generally make assumptions about these distributions, for example, few modes, low entropy or variance, and a lower bound on the transition probabilities. This reflects the typical design of a game, the player must be able to create their own relatively accurate internal model of how the game behaves. Games are generally not fun if there is too much randomness, the player must be able to make predictions and act accordingly. In addition, transitions that occur in one out of a thousand trajectories are probably at the limit of what is acceptable, since otherwise players will never encounter them. [3]

---

[2] See for example equivalence of Non-deterministic Finite Automata (NFA) (Fu et al. 2017)).

[3] There are of course exceptions, e.g. obtaining rare items, which rather than being part of core mechanics are typically a reason to brag to fellow players.

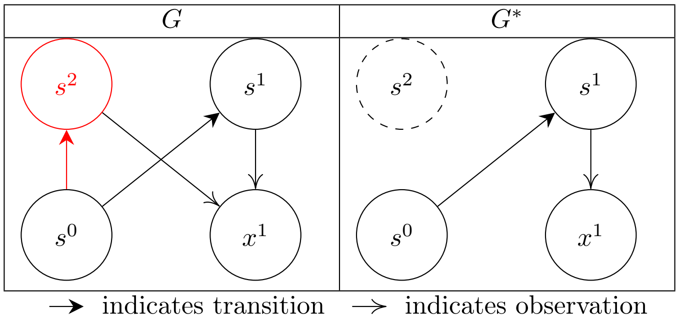


Figure 3.3: Examples of pathological faults in partially observable environments. The first example shows an unintended state fault that is hidden behind a valid observation.

A trick that human testers employ in order to address this limitation is to focus on replaying certain portions of the game for which transitions are more complex or highly variable in nature. A sophisticated testing agent might similarly choose which portions to replay, although to the best of my knowledge there are no works that have attempted this. Another trick is force the game into determinism, for example by setting the random seed, or otherwise *cheating* by explicitly choosing a particular outcome in a way not usually available to the player.

Hidden Faults

Hidden faults are a well known concept in software testing, in fact they are baked into the definition. The problem is simply that faults do not always give rise to failures. This can be made more precise: if the agent does not observe the full environment state or full state of the program (i.e. the environment is partially observable), then there are certain pathological cases that will prevent the agent from finding a fault. An example is given in Fig. 3.3. In some ways faults in this class are less of a concern since the resulting failure will never be experienced by a real player. However, they tend to be a symptom of a deeper problem and may manifest at a later date after some changes to the code. The only real solution to this is to provide more information to the agent in its observation.

At this point it is worth clarifying what is meant by an observation, and why faults have been defined only in terms of the transition function $\mathcal{T}$ and not the observation function $\mathcal{O}$. In video games, and software more generally, the program output is at some point stored as part of the program state. In video games for example, the final rendered image shown to the player is stored in memory as an array of pixel values. The transition function operates on this array like any other part of the state. Often an observation will correspond directly with the program output. In the simplest case, the observation function can be seen as a kind of indexing or *inspection* operator. It simply chooses a subset of state variables as the observation. In some instances it might transform these variables into a more useable form. Either way, it is assumed that the transformation contains no more information than is present in the state, and is therefore in a sense *bug free*. This is akin to saying, a human tester's senses are working correctly, or that there is no outside interference - there is nothing blocking the screen in front of them, they aren't under the influence, etc.

Although faults have been defined in terms of the transition function, bugs are defined in terms of *observables* (failures are always *observed*). The hidden faults problem is a serious one if an agent's observation does not accurately reflect the actual state of the environment. If for example, the agent only observes its position, by definition it cannot identify issues with its rotation. Fortunately, there is a kind of observation that captures a large portion of the state, and that is the audiovisual output of the game. Games are designed with playability in mind, and a large part of this is conveying the right information to players. This tends to mean that the audiovisual observation reflects the quantities that matter for gameplay (and therefore for bug identification). Game designers even go so far as to present information that tells the player about the future to allow them to prepare and anticipate what will happen. By doing so, they are reducing epistemic uncertainty, or in other words making the game more deterministic from the player perspective.

Contextual Bugs

Partial observability may mean that an agent needs to consider the *context* in which a bug appears in order to identify it. In other words, some hidden faults can be revealed by leveraging context, which here usually means to make use of past or future experiences. Bugs that do not require context are known as point bugs, those that do are known as contextual bugs. See Fig. 3.4 for an illustration.

Summary

Sampling limitations present a practical problem that requires some consideration when developing testing agents, this is revisited in section 3.4.3. The only way to address the hidden faults problem is to provide a testing agent with richer observations and to ensure contextual information is used where necessary.

In the next section we will move to the more practical setting where the intended behaviour G^* is not known *and* G requires sampling. This introduces a slew of new problems, the primary one being how to address the test oracle problem and gain at least some knowledge of G^* without needing to specify it explicitly.

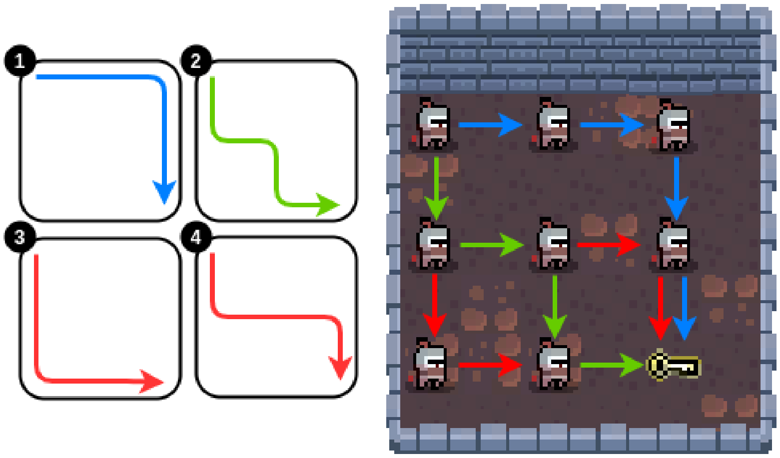


Figure 3.4: In order to distinguish between point bugs and contextual bugs, a testing agent may be required to use past information. Simply working from the current observation may not provide the relevant context to identify a problem, especially in partially observable settings (for example when looking only at the screen). To illustrate this point, an Non-Player Character (NPC) (the knight) is programmed to move from its starting position along one of two specific paths to reach the key object. A testing agent who already has a model of intended behaviour observes the NPC's trajectories as illustrated. Four distinct trajectories are observed, but only two are intended (1 blue and 2 green), the others are unintentional (3 red and 4 red). The point bug appears in trajectory 3, when the agent is at the bottom left position. This position is invalid in all intended behaviours, knowing this the agent does not need any of the additional information to determine that there is an problem. The contextual bug appears in trajectory 4 at the centre right NPC position, which is valid in trajectory 1 but not in trajectory 2. The agent must therefore look to context (i.e. the previous NPC position) for information about which path was taken. The testing agent must therefore have *memory* in order to identify the broadest range of bugs. If the agent treats entire trajectories as observations then both 3 and 4 can be considered point bugs, the of course comes with an increased cost of compute.

3.3 Learning to Identify

The stance we take on addressing the test oracle problem is one in which *experience* or *data* plays a critical role. Like in model-based testing, we wish to create a high-level specification (a model) of the intended behaviour, but unlike model-based testing, this specification is in part *learned* by an agent from experience. This is what is meant by an *implicit* specification[4], rather than an *explicit* one which in contrast gains nothing from experience.

There is already a well established field of study that effectively has this aim, namely, anomaly detection. The field has its roots in statistics and machine learning and is concerned primarily with the identification of *anomalies* in data. Anomalies are experiences, or patterns in experiences that do not conform to a well-defined notion of normality (Chandola et al. 2009)[5]. The ideal notion of normality in our setting is one that has a direct correspondence with intended behaviour, an anomaly is then any observed violation of the intended behaviour i.e. a bug. Viewing bug identification through the lens of anomaly detection gives us the relevant language, concepts and formal frameworks with which to specify high-level objectives and notions of normality (e.g. statistical normality).

3.3.1 Intended Behaviour & Expected Behaviour

Imagine an agent situated in an unknown environment G, the agent is embodied, can observe the world around it, can take action to change it, and has some capacity for learning. What can this agent say about whether the environment is functioning as it should? The agent might develop some idea of how *it* thinks the environment should function, but this would only be perspectival. Perhaps it often experiences objects through which it cannot pass and develops a notion of *solidness*, then at some later time finds an object that is the exception to the rule. It would be quite strange if the agent were to conclude that reality was somehow mistaken and insist that all objects are in fact solid or at least *should be* solid, rather than that it had not yet formed a complete model of the world[6].

The story above points to the fundamental difference between intended behaviour and what we will term expected behaviour. The agent's *expectations* are derived from experience of how the world is (or was) are what allows it to identify abnormality or *novelty*. But intended behaviour is in some sense *outside* the agent's experience, at some fundamental level whatever it experiences in G tells it nothing about G^*. There may

---

[4] in (Barr et al. 2015) an implicit solution to the test oracle problem is one that does not require domain knowledge or a formal specification, and whose purpose is to identify problems that are *always problems* (e.g. segmentation faults).

[5] other terms commonly used in place of *experiences* are: events, outcomes, observations, examples or samples.

[6] of course, an agent with enough sophistication might conclude that it was hallucinating. We will not consider this possibility as it complicates matters.

even be some agreement between what the agent considers abnormal or novel and what is abnormal in G according to G^*, perhaps the phantom object really was an oversight made by the *creator* of the environment the agent inhabits. But the agent whose experience is bound to this environment could never know for sure. What this story really points to is that to be useful, in the sense that it is able to make statements about intended behaviour, the agent will require information that comes from outside G. This may come in the form of biases instilled into the agent, or may mean providing explicit supervision which the agent must treat as axiomatic. The agent might otherwise be placed in a domain broader than G so that it might learn more general concepts. The broader domain, which may for example be composed of many video games, might contain information about G^* which the agent can leverage. In each case, this information will bring its model (expected behaviour) closer to intended behaviour. Where this is a consideration in the design of an agent, we say that expected behaviour is an approximation (or model) of intended behaviour.

Just being in a broader domain is not necessarily enough to make direct statements about intended behaviour. For this, the agent would need to be in a kind of *meta-domain* akin to a human tester, one in which the agent could conceive of *sub-domains* which are intentionally constructed and have some grasp of this intention. Nevertheless, even without this it may be easier to align expected behaviour and intended behaviour if the agent can form a more general model in the broader domain. These ideas are made more concrete in the next section.

3.3.2 Sources of Knowledge

A human tester comes to know the intended behaviour by drawing on knowledge from various places. It may be from their experience playing the game in question, playing other games, outside of game playing in their everyday life, from written requirements or specifications, or through interactions with the rest of the development team. An automated system could gain knowledge of the intended behaviour via analogous routes.

In AI broadly, there is a distinction made between two kinds of knowledge - *experiential* knowledge gained from data, and *prior* knowledge that is built into an agent. Experiential knowledge may be obtained by playing the game in question, other games, or through supervision from the testers or developers. In more traditional AI, prior knowledge often comes in the form of rules, or collections of facts about the world typically specified in a logical form. One can think of guards and models as in model-based testing as being all prior knowledge, there is nothing gained from experience.

In machine learning prior knowledge comes from the learning objective, optimization procedure, model

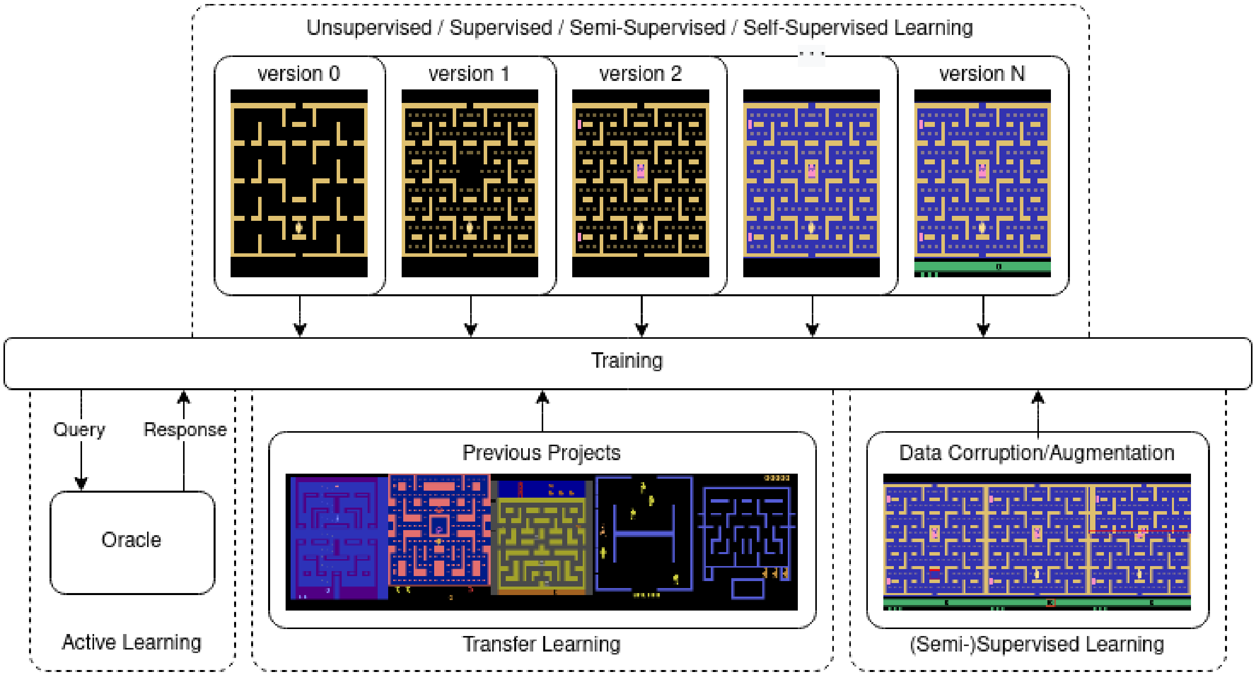


Figure 3.5: Overview of the sources of experiential knowledge available to an agent. See text for details.

class/architecture, and any other built-in assumptions e.g. prior probabilities in Bayesian learning. This kind of prior knowledge is sometimes called *inductive bias* (Mitchell 1997). The inductive biases that an agent possesses will determine what kind of things can be learned (the hypothesis class), and how experiential knowledge is represented and used. While inductive bias plays a critical role, especially in developing notions of normality as we will see in section 3.4, for the moment our focus is on the kinds of experiences an agent can have, as this is just as important.

We have already seen an analogue testing agent that only has access to the current game implementation and concluded that it could not say anything about intended behaviour without outside help. This may take many forms, for example, an agent may receive supervision or labelled examples of bugs, or know for certain periods of its experience there are no bugs present. Or as we said before, it may exist in a broader domain, having access to other games (bug-free or not), or data seemingly unrelated to video games. Each case both opens up opportunity and places constraints on the methods that might in principle be used to address the test oracle problem. By no accident, the various paradigms in machine learning are suited to addressing these alternatives.

Supervised Learning

In supervised learning, each observation x has an associated binary label y which, for the purposes of anomaly detection, typically indicates whether the observation is normal (0) or abnormal (1). Practically, since the game we are testing is currently under development, we are unlikely to have such a labelling, at least not for the game in question. This would involve a human tester going in and pre-emptively identifying bugs, which is the task we are trying to automate in the first place. Assuming we don't rely on data from other similar projects, which would take us deep into the weeds of transfer learning (see 3.3.2), what might supervision look like?

One straightforward approach may be to use a human tester initially, obtain some initial labelling and train the agent on this data in the hopes it can take over later. This might work well for bugs that reappear at a later stage. But the problem is that the labelling will not account for novel issues that are sure to appear later in development. The agent will be restricted to identifying only the issues present during the initial testing by the human. And will only be able to do so if it is able to *transfer* its knowledge of the bug to later versions of the game, see section 3.3.2.

In (Ling et al. 2020) and later Tamm et al. 2022, a different approach was taken when in search of simple graphical bugs such as texture missing. They used publicly available game assets (3D models/textures) to construct a training set. The models they tried were able to transfer well across games with different assets. Without massive amounts of pre-training (for example as in (Radford et al. 2021)), this approach is currently only realistic for the simplest of bugs, but nevertheless promises some gains.

In the vast majority of domains present in the wider anomaly detection literature it is similarly difficult to obtain labels (Chandola et al. 2009). Most work is therefore in the development of unsupervised, or semi-supervised approaches.

Unsupervised Learning

Unsupervised learning tries to model the underlying or inherent structure of data and does so without supervision in the form of labels. There are many approaches: clustering, density estimation, dimensionality reduction, latent variable models, among others. The situation video game developers will find themselves in, with a partially implemented game that likely already contains bugs, is precisely where unsupervised learning may offer solutions. Of course, to work properly, the right inductive biases must be provided and this is far from trivial. Taking an unsupervised approach will often require making strong domain-dependent assumptions. Section 3.4.1 presents a simple unsupervised approach that relies on an assumption about the statistical properties of bugs, namely that they are unlikely. Unsupervised learning methods are also used to learn *representations* that might be used downstream for the purposes of anomaly detection. This is discussed in more depth in later sections.

Semi-Supervised Learning

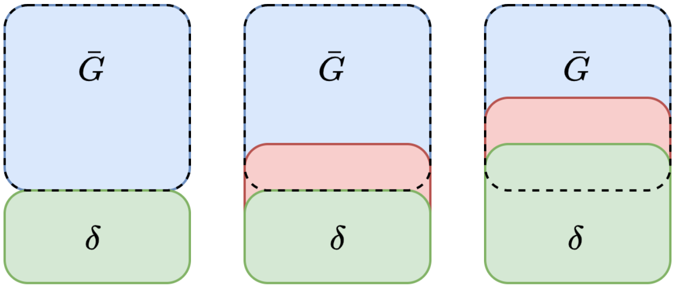
There is a spectrum between supervised and unsupervised learning. Along this spectrum lie methods that require partial or *weak* supervision. This can come in many forms, usually where labels are noisy, imprecise or otherwise limited.

One example of weak supervision in anomaly detection involves the use of in-distribution and Out-Of-Distribution (OOD) examples (Hojjati et al. 2022). In-distribution examples may be normal in-game observations, OOD examples can be anything else, images from the web, observations from other games, bugged observations or, systematically corrupted or augmented observations.

In chapter 2 we saw this approach for GUI related bugs (Liu et al. 2020), where it was relatively easy to introduce corruptions such as overlapping text. In some restricted settings introducing corruption may also work well in video games, for example in identifying corrupted geometry or textures, but would not work for bugs where it is not clear what kind of corruption process would suffice. The same applies to other kinds of OOD observations, although under some training schemes not all OOD observations need to correspond to any particular kind of bug. For example in Self-Supervised Learning (SSL) as will be seen in chapter 5.

Otherwise, weak supervision may refer to the situation where only normal examples are available and abnormality is only encountered at test time. Abnormality in this context is referred to as *novelty* (Markou et al. 2003). Novel observations are those that don't fit the patterns seen in the training data (i.e. don't fit the agent's expectations given what it has already experienced). The identification of novelty is exactly the problem faced when doing regression testing, where novelty corresponds to a regression.

The learning-based approach to regression testing follows the usual process: (1) Choose an independent submodule $\bar{G}$ (e.g. a level, puzzle, shader program, etc.), assume it is normal and should remain constant in its behaviour even after an update to the wider project. (2) Write test cases, or train an agent to model $\bar{G}$ (part of its environment). (3) Run the agent in *test mode* after each update to the wider project, changes in $\bar{G}$ are novel, and we can assume they are bugs. Treating bugs as novelty resolves the test oracle problem as the program itself represents all that the agent needs to know to perform future tests. This simplification hinges on the assumption that changes to $\bar{G}$ are indeed bugs. In practice this may not be true, the change may be a feature (see Fig. 3.6). In such a case the agent requires retraining, or similarly, test cases need to



(a) (b) (c)

Figure 3.6: Illustration of regression testing. The submodule $\bar{G}$ should be independent of the update δ as in (a). If after an update $\bar{G}$ changes, as in (b), then this constitutes a bug. In some unfortunate circumstance (c) the update will introduce new features into $\bar{G}$, in which case test cases need to be re-written, or agents retrained.

be re-written. Generally the agent need not be retrained from scratch. The developer could repeatedly run the agent until it flags only features, fixing bugs along the way, at which point it would be retrained on the updated $\bar{G}$.

The identification of novelty in regression testing is perhaps the simplest intelligent testing automation paradigm as by a blanket assumption it avoids the test oracle problem. This is not to say that the problem is easy, identifying novelty is challenging, especially in settings similar to video games where experiences may be high-dimensional. This is the testing paradigm that we develop further in chapter 5. Since novelty and abnormality are closely related, some of the ideas we develop are also applicable to the other testing paradigms.

Active Learning

Another kind of weak supervision may come from querying an oracle. Active learning tries to elicit information (labels or otherwise) by putting questions to the oracle, who in this case may be a member of the development or testing team. Active learning is often framed as a problem of asking *optimal* questions, those for which answers will give the most information possible, thereby minimizing the number of queries needed (Settles 1995). The hope is then that humans need to expend less effort in answering these queries than they otherwise would if identifying the bugs themselves.

Active learning may make bug identification possible at all phases of testing, not just when particular submodules are finished, which typically occurs at the later phases. To outline how this might work, an agent is trained on some initial version of the video game G to identify novelty, or otherwise create a model. The agent is then tested on an updated version G' where it identifies instances of novelty, and queries the oracle as to which are bugs and which are features. The agent incorporates this supervisory information and continues the process in subsequent updates. Again, there are many open questions, and some more and less obvious caveats here. Among the most glaring is: *how can queries and responses be phrased such that an agent can distinguish between bugs and features that appear in the same observation?* Many others are explored in the active learning literature (see e.g. (Ren et al. 2021; Settles 1995)).

In a fully-fledged intelligent automated testing system it is likely that this kind of human-in-the-loop learning will play a central role. However, as it is fraught with difficulties, perhaps even more so than the other paradigms, beyond the brief discussion here it is left as a broad direction for future work.

Transfer Learning

Transferring knowledge that comes from other similar games or previous projects is a promising but challenging direction. If the same kind of bug manifests in numerous environments it may be possible to learn a relatively general model that might be used across projects.

We saw in the last chapter an example of transfer learning with CLIP (Taesiri et al. 2022), which with vast amounts of data has learned useful representations which are transferable across (some) games. In the wider anomaly detection literature there are a number of recent works that look to transfer knowledge of anomalies (e.g. (Maschler et al. 2021)). That is, to learn about a particular kind of bug in a largely domain-independent fashion.

To give a more concrete example, in a class of games that require navigation around solid obstacles in 2D space, an agent might learn that it cannot pass through walls. Upon encountering a phantom wall, it may conclude that this is a bug, but only because we have made use of the agent in *this* class of games.

Along similar lines, this approach might work well for so-called *always bugs*, which include bugs such as texture missing, geometry corruption and geometry clipping. These are bugs no matter what video game they appear in. Where transfer learning may again struggle is for bugs that may in some games be considered features. To give an example, the sequence of observations resulting from a high force bug is very similar to that generated by the common *jump pad* mechanic. If the player walks on to a *jump pad* they will similarly be flung into the sky. This highlights the importance of context in determining whether a bug is a feature or not. A bug can always be contextualized at some level as human testers must be able to perform the job. But developing agents for this purpose might prove very challenging, especially if the context required is beyond what the agent has access to (e.g. in a natural language document). It is clear that transfer learning will play a central role in developing solutions to bug identification (and automated testing more broadly), but it is challenging to realize in practice.

3.3.3 Continual Learning

In our discussion thus far we have alluded to the fact that the development process means that the game is non-stationary. That is, it changes with time as new features (and bugs) are added. This makes bug identification all the more challenging as an agent must again be able to transfer its knowledge.

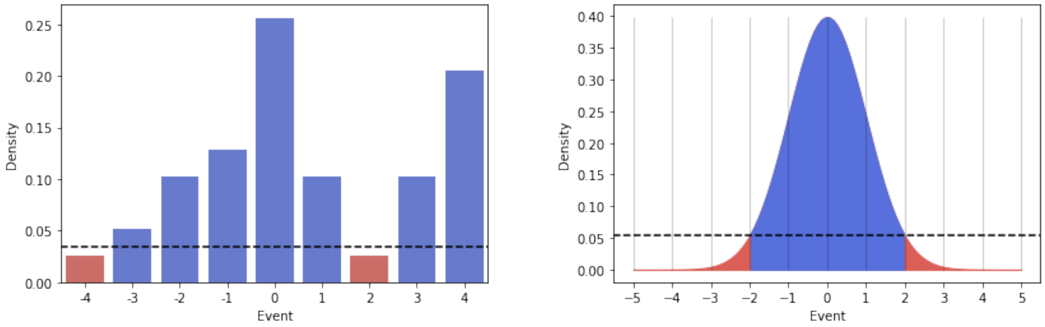
In the broader machine learning literature, this setting, where an agent must learn about a new task (identifying new bugs) without forgetting about the old tasks (identifying the bugs already seen), is referred to as continual learning. To give a concrete example, consider again the supervised setting where a developer provides a sufficient labelling of bugs in an initial version. In the next version, the developer might again provide a labelling for *new* bugs, but neglect to re-label those that were seen in the first version since this would be a waste of their time. The agent must not only remember the old bugs (avoid catastrophic forgetting), but also transfer its knowledge of them into the new version where there may be distributional shift[7]. Other settings, such as regression testing can be similarly framed (e.g. as a problem of continual novelty detection (Aljundi et al. 2022)). In general when developing testing agents non-stationarity must be dealt with. Continual learning is widely studied in machine learning (De Lange et al. 2021) and there has been some work on anomaly detection for non-stationary data (e.g. in low-dimensional time series (Salehi et al. 2018)) but the problem is still largely an open one.

Summary

Three ways of addressing the test oracle problem that leverage experience have been presented: (1) to assume it away by asserting that a certain submodule is complete, bug free and can be used as an oracle. The problem is then one of novelty detection (expected behaviour is equivalent to intended behaviour). In this case only regression testing can be performed. (2) To engineer the agent and ensure it has the correct inductive biases specific to identifying certain issues or for transferring knowledge, and/or make strong normality assumptions as in unsupervised learning. (3) Use an oracle as in active learning, or through a labelling as in supervised learning. The three approaches are not mutually exclusive. (2) and (3) are much more general and might be used at any stage of testing. In later chapters our focus is principally on (1) and (2), but we also briefly touch on (3) (full supervision) in chapter 4. We also examined a number of challenges that are faced in bug identification, including the possible lack of explicit supervision for the GUT and non-stationarity, both of

---

[7] in the new version perhaps a new shader is being used, then with visual observations this slightly changes how the agent views the environment. The agent must be *invariant* to these changes.



(a) Discrete (b) Continuous

Figure 3.7: Defining normality for discrete (a) and continuous (b) variables. (a) shows a threshold of 0.035 which given the distribution means outcomes -4 and 2 are considered anomalous. (b) shows a chosen reference interval (2 standard deviations), meaning any outcome $x > 2$ or $x < -2$ is considered anomalous.

which require an agent to transfer knowledge in some way. We also touched on the problem of alignment (bringing together expected and intended behaviour), arguing that the agent requires *outside* knowledge to be able to make statements about what is intended. In the next section this will be explored in more depth as we bring concepts from anomaly detection into the fold.

3.4 Normality

After determining the kinds of experiences that are available to an agent, a suitable notion of normality needs to be defined. A notion of normality can be thought of as the means by which an agent may come to know what is intended. Perhaps the easiest notion to consider is that provided by explicit supervision in the form of a labelling. If the agent is trained to classify experiences (normal/abnormal), then in some sense the supervision serves as the notion. This can be made much more precise by considering classical learning theory i.e. under infinite data for some model class and learning objective.

Without explicit supervision in the form of a labelling, which is more realistic in the context of testing, we must derive some other notion of normality[8]. The hope is that what ever notion we come up with will correspond to that which captures intended behaviour (as the supervised notion does). The most important and widely used notion is that of statistical normality.

3.4.1 Statistical Normality

Statistical normality is usually defined (Chandola et al. 2009) as follows: given a random variable X, a statistical anomaly is an outcome $x \sim X$ that has a low density according the Probability Density Function (PDF) associated with X. Normality is determined by an arbitrary choice of threshold, or reference interval κ. For discrete random variables, the PDF (also called the Probability Mass Function) measures the probability of observing a particular outcome. An outcome x is anomalous or *abnormal* if $Pr(X = x) < \kappa$. For continuous random variables the probability of observing a particular outcome x is zero. Instead, density measures the relative likelihood of an outcome, or how likely it is to be close to a chosen value. A threshold can similarly be defined for the density, see Fig. 3.7.

In practice we only have access to the underlying distribution through sampling (i.e. playing the game). We might try to model or estimate the PDF from data. To illustrate, a Kernel Density Estimator (KDE) is used to model the distribution of 2D player positions, with the aim of identifying the so-called player out of bounds bug. To do so, a threshold can be set on the estimated density. The model is trained on data obtained by playing the game, see Fig. 3.8. The model is able to identify the bug, but also gives a low density to the top left corner of the room since the agent neglected to visit this region. The distribution that is being modelled here is the distribution of observations that is induced by the agent's policy, this is defined concretely later in section 3.4.3. But we could just as well model a different distribution, for example the (observational) transition distribution $Pr(X_{t+1} | X_t = x, A_t = a)$ by repeatedly visiting the same observation and taking the same action.[9] The broader question in each case is whether a density below the threshold actually corresponds to unintended behaviour.

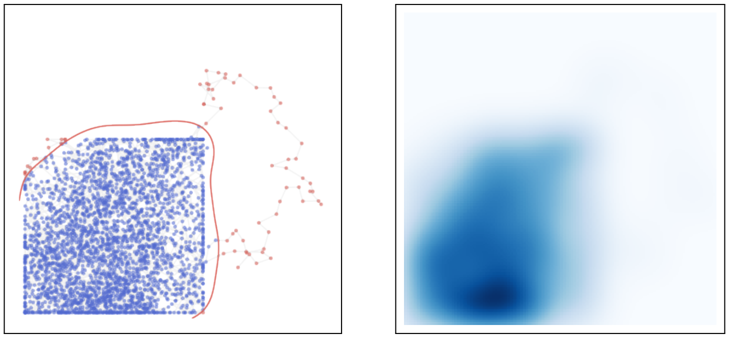
Endemic Faults & Systemic Faults

Consider again the situation above where the player is able to escape the bounds of the level. The fault that led to this bug may have been the following: the collision box of the wall was erroneously disabled. If it were disabled for the full duration of the agent's experience and the agent were modelling the transition distribution, even assuming full observability the problem would be missed. This is unless the agent is able to use other semantically similar examples i.e. of not being able to pass through other walls. Then even in the most difficult setting (unsupervised with only experience of G) normality might still be framed as

---

[8] If we are attempting transfer learning then we may have supervision, but this in itself is not enough we must still decide how to deal with distributional shift. Our subsequent discussion should also shed some light on this front.

[9] There are some complications with this. The environment may not be in the same state when the observation is revisited, and the density estimate might be systematically biased by the revisitation strategy.



(a) Random Walk (b) Density (KDE)

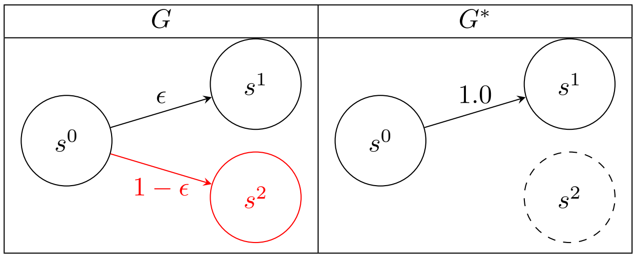
Figure 3.8: The player is in a 2D room with a boundary that they are not supposed to be able to pass through. They take a random walk and at some point are able to escape the room, shown in (a). A Gaussian Kernel Density Estimator models the density of the player positions, shown in (b). A threshold is set (red line in (a)) that determines normality. In the top left corner of the room, the estimator erroneously estimates the boundary as being inside the room as the player rarely visits this region.

statistical. In practice, the focus would become much more on learning good abstractions or *representations* of the data in order to make these kinds of semantic comparisons. We refer to the kind of fault that shifts probability mass away from intended states making unintended states much more likely as endemic (see Fig. 3.9).

Now imagine that collision detection has been completely disabled. The agent can now pass through all walls. This kind of fault, which we refer to as systemic, has far-reaching consequences. If the agent only ever experiences these ghostly walls, it can't possibly identify the resulting bugs without knowledge of the contrary which must be obtained elsewhere as was highlighted in section 3.3.1.

Many of the faults that occur in practice are either endemic or systemic. Whether this is problematic will depend on how statistical normality is being specified in practice (see next section), what biases the agent has, how it models its experiences, and what data and supervision is available. If the domain of the agent's experience is broadened, then statistical normality is still a relevant notion as systemic bugs may become endemic in the broader domain. In addition, as the agent is always measuring statistics over its experiences rather than the full state, there will be cases where the endemic nature of the problem is reshaped by other factors (for example, the statistics of the agent's policy).

But what about unlikely intended behaviour? There are situations in video games where unlikely occurrences are part of intended behaviour, perhaps the player discovers a secret area by taking a very specific sequence of actions, or a rare item is obtained through pure chance. Without access to outside information, there is no straightforward way to deal with this. It may be that the probability of these occurrences needs to



(a) Endemic Transition Fault

Figure 3.9: Example of an endemic transition fault. s^2 is missing from the intended behaviour G^*, but present with high probability (where ϵ is small or zero) in the current implementation G. The resulting endemic bug will be frequently encountered in the agent's experience of this transition. This can be problematic for a statistical notion of normality where it is assumed that bugs are unlikely to occur.

be explicitly inflated e.g. by systematic exploration or using the tricks outlined in section 3.2.2. Otherwise, again it will be a case of choosing the right biases for an agent.

3.4.2 Representations & Normality

While statistical normality is a powerful idea, it is very much reliant on the random variable X that is chosen to be representative of experience. This is nicely highlighted by the following real-world example:

Every pebble on a beach is unique in its shape, colour and pattern, and so in some sense equally unlikely. Yet only some attract one's attention while out on a stroll. For myself, it is those that are particularly symmetric or regular, that have holes, or glint in sunlight. Importantly, I have decided that for example, a pebble's topology is an important feature, but I could have just as well have chosen some other feature or combination of features. The measure by which I am comparing pebbles defines the relative likelihood - I am measuring the likelihood of a pebble matching an abstract representation. Or in simpler terms, the likelihood of observing particular features. Along the same lines, I have taken an interest in pebbles specifically, although they form only part of my current experience and have done so in part because of my bias towards objectness.

The pebbles are analogous to walls in the player out of bounds example above, especially in the endemic case. In order to identify the bug, the agent should consider the properties of each wall, namely, solidness, independently of other aspects of its experience. In the population of walls, perhaps one out of a hundred is a phantom, making this wall a statistical anomaly.

An agent's inductive bias plays an important role in determining what kind of representations[10] it is

---

[10] it can be helpful to think of representations as *properties* of certain experiences or objects (e.g. position, or solidness), but the notion is more general than this. A representation might not correspond to properties that we are familiar with or can express neatly.

measuring statistics over. In other words, the way in which an agent internally represents its environment matters greatly for identifying bugs. By fiddling with inductive biases or with the data available to the agent, as we do in later chapters, one can shape the representations to better align expected behaviour and intended behaviour.

In the player out of bounds example, the player's position was selected as the feature (or representation) that is most important for the task, but more generally it may not be clear what underlying features are relevant. Even in this simple example, not including information about walls was probably a mistake.

For more complex problems, it is often better to give an agent a richer experience[11]. Ideally, an agent would observe the full audiovisual rendering of the game as a human would see it. The problem becomes is again about introducing the right biases so that the most relevant features are used. With this kind of observation the problem is made more challenging, since the agent must first extract the relevant underlying features from the high-dimensional pixel (and waveform) data. But it comes with some major benefits, the most notable is reusability through decoupling. There is a substantial amount of work in anomaly detection that aim to solve a similar problem for visual data (see surveys (Bogdoll et al. 2022; Yang et al. 2021; Kiran et al. 2018)). In video surveillance for example we are interested in identifying fires, dangerous situations, or suspicious behaviour from CCTV footage (Patil et al. 2017; Ramachandra et al. 2020). The techniques developed in these areas might be applied directly to bug identification.

Measuring Normality with Model Error

A class of approaches that attempt to learn *good* representations and make approximate use of statistical normality are based on the following observation: a learner will tend to focus on reaching good performance for the more frequent observations and their shared underlying features. Because they arise more frequently, they carry more weight in the learning objective that is being optimized. Consider the case of Least Squares Linear Regression. Examples that lie further from the best-fit curve will have higher error and so be considered more abnormal. Under certain assumptions[12], these examples are also less likely. Under the same assumptions model error can be seen as an unnormalised density.

This idea extends to more complex models, such as Auto-Encoders (AEs) (Kingma et al. 2014; Bengio et al. 2013; Makhzani et al. 2016) or Generative Adversarial Networks (GANs) (Goodfellow et al. 2014). Deep learning approaches have been used extensively for anomaly detection in domains that deal with images, video and other temporal/high-dimensional data (Chalapathy et al. 2019; Chalapathy et al. 2019; Pang et al.

---

[11] For some simpler kinds of bugs, it may be easier to engineer the features to ensure statistical normality is applicable.

[12] where the noise term is Gaussian.

2022; Di Mattia et al. 2019).

Approaches that make use of AEs tend to use reconstruction error as a normality score. That is, they encode an input as a low-dimensional *latent*[13] vector representation, then attempt to reconstruct the input from the representation and compare the reconstruction to the original, reconstruction error is the difference in this comparison. The low dimensional bottleneck ensures that the network captures the most frequently observed, most important, or most easily represented features in the learned representations. Other analogous approaches for anomaly detection in time series attempt to predict future (or past) observations and use prediction error as a normality score e.g. (Medel et al. 2016).

Approaches that make use of GANs tend to use either the generator loss, which is analogous to reconstruction error, or the discriminator loss, which is a kind of likelihood estimate, or a combination of the two. In (Lee et al. 2018) for example, the approach is to use temporal information from a Long-Short Term Memory network (LSTM) to generate a fake frame which replaces a frame in a real video clip. A discriminator is trained to distinguish real and fake video frames. The normality score is a combination of the generator and discriminator loss.

One Class Classification with Representations

The deep generative models used in the above produce latent representations of the input which can be used in other ways. One might for example train a simple classifier (e.g. One-Class Support Vector Machines), perform clustering, density estimation or otherwise, using the representations as input to identify abnormality. This approach has been taken in a number of works, see (Chalapathy et al. 2019) for a review. Other approaches include those based on Self-Supervised Learning (SSL), which has emerged more recently as a method for learning representations for use in down-stream task such as anomaly detection, see (Hojjati et al. 2022) for a review. A more in-depth review of SSL for anomaly detection is given in chapter 5 where a method for bug identification is also developed. Other approaches for learning representations exist, for example: disentangled representation learning (Locatello et al. 2019) but are not discussed.

Normality, Novelty & Surprise

In our discussion of normality and means of measuring it from the perspective of a testing agent, novelty has been presented as a fundamentally experiential notion. In related domains, such as Evolutionary Computing (EC) similar concepts exist and have been used to design selection criteria. In EC novelty is defined as *the*

---

[13] latent as in hidden, underlying or not directly observed.

degree to which a solution behaviourally different from those prior (Lehman et al. 2011). A related notion is that of *surprise*, which perhaps is closer to the notion of novelty that has been presented in this chapter thus far. Surprise is defined as *the degree to which a solution is behaviourally different from the expected solution*, where *expected* effectively means what is *predicted* given prior solutions (Gravina et al. 2016). Solutions in the context of search are agent behaviours, and in the context of bug identification are the experiences of an agent. Of course in the case of bug identification, rather than directing search we are interested simply in the question of whether an experience is normal (novel, or surprising). This is exactly the question that is being asked in EC (and related fields) when designings search algorithms. It serves to further highlight the deep connection between bug identification as it is presented in this chapter and search.

Unlike in EC, where we have a global governing selection process, any notion of *difference* as in novelty, for an individual agent, must be derived from its previous experiences[14]. As such, in this thesis we see surprise as an instance of novelty. Novelty being an crucial ingredient for self-imposed normality as is the goal in the development of general testing agents.

Connecting Normality with Curiosity & Intrinsic Motivation

In chapter 2 we looked briefly at intrinsic motivation in discussion of the search problem. It was noted that count-based approaches aim to encourage exploration by keeping a running total of the number of times each state (or observation) is visited, and using this count to direct exploration towards regions that are less frequently visited (Bellemare et al. 2016).

This count is known as the *visitation frequency.* The visitation frequency for a particular state is an estimand that when taken collectively over all states corresponds to the density of the state distribution that is induced by a particular policy (or mixture of policies). The kernel density estimator exemplified in section 3.4.1 is estimating the visitation frequency for each player position and interpolates for unseen positions. In practice counting states is very inefficient and fails for higher dimensionality because of the infamous curse of dimensionality. This also applies to kernel density estimation.

Perhaps unsurprisingly, the methods that aim to improve count-based methods strongly resemble those presented above for anomaly detection in high-dimensional temporal settings. They are also fundamentally connected to the notion of surprise as discussed above. To give an example, in (Stadie et al. 2015) an Auto Encoder(AE) learns to represent the agent's observation $\phi(x_t)$. A second neural network, the *forward model*, is trained to estimate $\phi(x_{t+1})$ from $\phi(x_t)$ and the action a_t that gave rise to the transition. This estimate

---

[14] there contrary cases that would not in the spirit of our discussion of general testing agents, such as simple rules written by a developer that are experience independent.

is compared with the actual $\phi(x_{t+1})$ (i.e. the encoding produced from the observed x_{t+1}) and the error is used as a measure of abnormality, or in this case, the novelty of the experience. As with other approaches to anomaly detection that make use of error as a measure, the intuition is that if the model makes bad predictions then it probably hasn't seen the example often enough during training. Similar approaches have been taken with different ways of modelling dynamics and/or representing observations (e.g. (Pathak et al. 2017)).

In both count-based and predictive approaches, the agent is attempting to model the distribution of observations induced by its policy thereby directing its behaviour to regions that have a low visitation rate. This is useful for exploration, but it presents a problem for bug identification which we outline in the next section.

3.4.3 Policy Dependence

As a machine learning practitioner, imagine you are tasked with building a novelty detection model that will identify rare birds in a collection of pictures taken by a professional wildlife photographer. You develop a model that learns a *good* representation of the birds, and estimates novelty using their likelihood. To your surprise, the most exotic bird according to the model is one you are familiar with, a domestic chicken. After puzzling for a moment, you suspect that the photographer may have a preference for photographing rare birds. This would effectively invert the probability of observing those that might otherwise be considered rare. This is an example of what is commonly referred to as *sampling bias.* The data has been preferentially sampled and so does not reflect the underlying distribution that is to be captured.

In a slightly less contrived example, now imagine that the photographer does not have a preference over which birds to photograph, and always photographs a bird when seen. The photographer still has to travel around taking photos. On their journey, they frequently visit cities, airports and stay in hotels. This increases the chances that city birds will be photographed, and so rare city birds suddenly become less rare. What's more, there may be places that the photographer never visits the depths of the jungle, war-torn regions, etc. The point here is that the model of normality is tied to the photographer's decisions and the routes that they are forced to take.

One might reasonably ask at this point *well, what distribution are we actually interested in?* Perhaps it *is* the one approximated by the travelling photographer, since this is how most people are going to experience exotic birds anyway. They will also stay in hotels, travel on aeroplanes etc. On the other hand, perhaps it should be the *true* distribution of birds i.e. the count of individuals in each species. The problem with the latter view is that in practice there is always a process of sampling that needs to happen, *someone* has to go and collect the data.

In training a model for the purposes of bug identification where data is collected using a game playing agent, sampling bias must be a key consideration. The notion of normality, especially statistical normality, is deeply tied to the agent's policy since just as with the photographer, it will determine the data that will be used for training. Choosing a suitable policy for the purposes of bug identification and dealing with sampling bias is an open problem. Regardless, for the moment our agents need a policy. An approach to exploration that has some desirable properties is presented below.

Entropy Maximisation

The importance of getting good environment coverage (exploring the environment to its fullest) was highlighted the example of identifying player out of bounds in section 3.4.1. On its random walk, the agent neglects to visit the top left corner of the room. This led to a low density estimate by the KDE in this area and subsequent misidentification. The case was similar for the photographer who neglected to visit war-torn regions.

If the aim is coverage, there are many policies that can *in principle* cover an environment, including a policy that selects actions uniformly at random. But for these policies, the data will tend to be extremely imbalanced. An improvement would be to follow a policy that balances the experiences of an agent. A set of policies that achieve this maximise the entropy of the policy induced state distribution, as outlined below.

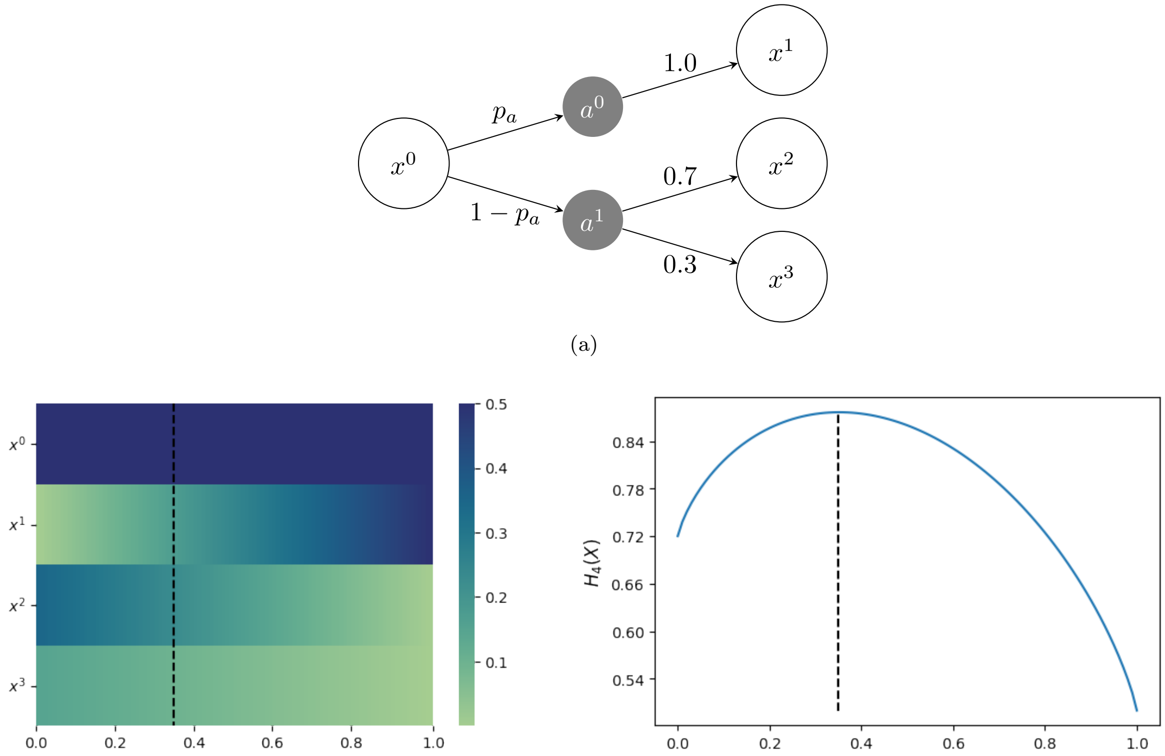
Policy Induced Stationary Distribution

An MDP (without reward) can be reduced to a Markov chain by combining action and transition probabilities. With a known and fixed policy, the stationary distribution of this Markov chain tells us about the stochasticity of an agent's overall experience for a given policy (assuming full observability). It tells us the probability that a particular state will be visited (in infinite playthroughs) and therefore be part of the training dataset. An irreducible Markov chain has a unique stationary distribution if all states are positive recurrent. As video games are finite closed loops that can be played repeatedly, they always satisfy this property. Intuitively, any game that can be restarted (e.g. via an in-game menu) satisfies this property.[15]

A policy that maximises entropy has action probabilities such that every observation is, as much as

---

[15] if the game crashes completely and the program exits, or the agent becomes "stuck" and cannot restart for whatever reason, the agent had better be able to restart the program.



(a)

(b)

(c)

Figure 3.10: The agent chooses its action probabilities so that the stationary distribution has maximum entropy, this means the agents observations are as balanced as possible in a dataset. The complete decision process is shown in (a), edges show the associated transitions probabilities. Each trajectory of states consists of a pair $(x^0, x^i), i \in \{1, 2, 3\}$, the probability of observing x_0 in the induced distribution is independent of the policy, and is therefore $P(x_0) = 0.5$. (b) shows how the stationary distribution changes with values of p_a, the colour bar represents the proportion of the given state present in the agents total experience (over n runs as $n \to \infty$). (c) shows the how entropy varies with p_a, in this instance it is maximized when $p_a = 0.35$. In general choosing a policy that maximizes entropy is highly non-trivial. See text for further details.

possible, visited equally often[16]. Consider a Markov chain with unknowns in the transition matrix that correspond to action probabilities. The problem of coverage can be cast as maximizing the entropy of its unique stationary distribution. For small MDPs this amounts to solving a system of linear equations to get the probability of each outcome in terms of the action probabilities, and then optimizing for entropy. The problem is illustrated in Fig. 3.10 with a simple example. The corresponding system of equations is easy to solve. Optimizing for entropy gives $p_a^* = \text{argmax}_{p_a} H(x) \approx 0.35$. This problem is much harder when the MDP is not known, and is intractable for larger MDPs. Nevertheless, there has been progress in this area (see e.g. (Hazan et al. 2018; Mutti et al. 2021)).

Training on a dataset generated by a maximum entropy policy should enable the agent to construct a good model that covers the environment, but will of course mean that inherently unlikely states will be greatly inflated in the training dataset. This is actually desirable if we are performing regression testing, since we will not confuse unlikely states seen in training for novel states at test time. But in an unsupervised setting where bugs are present during training it presents a problem. The sampling bias must be dealt with somehow, and as stated earlier this is an open problem. Tackling the problem is out of the scope for the rest of this thesis, but its importance should not be understated.

In later chapters, we tend to take the view that the distribution to be modelled is the one induced by the agent's policy. But, for the most part this is because it is difficult to remove policy dependence. Where the dependency can be removed, it is. In simpler environments this may be done by ensuring total environment coverage and then forcibly balancing the dataset by removing duplicate experiences. In more complex environments where this is not possible, we accept the bias introduced by the policy and either hope that its effect does not greatly interfere with the results, or consider bugs which are in a certain sense policy independent.

3.5 Summary

This chapter presented an exposition on learning as a potential paradigm for addressing the test oracle problem. The relationship between expected behaviour (statistical normality) and intended behaviour was explored. To be most broadly applicable and to avoid issues with systemic bugs, approaches based on statistical normality must be trained in a domain broader than the Game Under Test (GUT) (e.g. on other similar games). In this broader domain, the issues that make statistical normality appear incompatible with

---

[16] there may be more than one such policy.

testing software (e.g. systemic bugs, and unlikely intended behaviour) can, at least in principle, be resolved. By instilling the right inductive biases and by giving the agent access to the right kinds of experiences, one can shape the representations that are learned and better align expected behaviour with intended behaviour. In its essence, this is exactly the problem faced by practitioners of anomaly detection in any given domain. The problem of specifying intended behaviour is in a crucial sense the *same* as the problem of developing methods for identifying anomalies, which is by no means trivial.

In our discussion, the major challenges that are faced in developing testing agents have been framed as existing AI and ML problems. These include: Transferring knowledge to deal with lack of supervision, and with the inherent non-stationarity that is brought about by the development process itself. Dealing with sampling biases brought about by the policy used to collect training data and ensuring that, at the very least, there is good coverage of the environment. Dealing with high-dimensional *visual* data to decouple the agent from a particular game. And, learning abstractions or representations such that general notions of normality (such as statistical normality) might be applied.

To a certain extent, all of these problems need addressing before testing agents might become general purpose. While they inform the work in later chapters, dealing with them all at once is not something that would lead to meaningful progress and so scope must be narrowed. In later chapters our focus is primarily on the semi-supervised problem setting, specifically in the context of regression testing and novelty detection. This is setting serves as a stepping stone over which more sophisticated approaches might be developed. The aim is to provide proof of concept of the benefit claims made earlier in this chapter, specifically around reusability and capability as this has not yet been demonstrated in the literature. In order to do this, a significant practical problem needs to be overcome. This is the distinct lack of available data with which to train and evaluate our approaches. In the next chapter we present a platform that makes this data available.

Chapter 4

A Platform for Automated Bug Detection in Video Games

This chapter presents World of Bugs (WOB), and is an extended version of the paper *World of Bugs: A Platform for automated Bug Detection in 3D Video Games* (Wilkins et al. 2022). WOB has been developed with the aim of reducing a significant barrier to entry in Automated Bug Detection (ABD) research (both search and identification), and that is the lack of available data. This data is essential for training, evaluating, and comparing approaches to the problem. The lack of publicly available data has led to a fractured literature with works devising their own game environments (Zheng et al. 2019; Bergdahl et al. 2020; Shirzadehhajimahmood et al. 2021; Gordillo et al. 2021) or using real games (Nantes et al. 2013; Gudmundsson et al. 2018).

The limited use of real games for ABD research is an unfortunate side effect of the general attitude towards intellectual property in the industry. Unless one ventures into the open-source community, development versions of games are a closely guarded secret. Open source games might have presented an avenue for researchers, if not for the effort required to obtain relevant data from bug reports, and to reproduce and record each issue. An additional problem with this route is that, where controlled experiments are concerned, it is preferable to have control over, when, where and what kind of bugs manifest, as well as the data gathering process (game playing) itself.

Of the works that have developed purpose-built environments to test and train agents, only (Nantes et al. 2013) has implemented bugs more complex than those that can be identified with simple rules. The bugs are primarily visual and include geometry and texture/UV corruption. The bugs were implemented by modifying the game's source code. The other works are motivated primarily by the exploration problem and attempt to identify simple bugs, such as those that result in a crash, terrain hole, player out of bounds, or getting stuck[1]. These bugs can be implemented and identified fairly easily. Introducing more complex bugs, like those implemented in (Nantes et al. 2013), is a significant practical undertaking and requires an in-depth understanding of the many software components that compose video games and their development. Ironically, as bugs are hard to come by, the experimental setup cost is high and this has stunted research progress in important areas of ABD, such as identifying some of the more interesting bugs and exploring the relationship between the search and identification problems.

The primary purpose of the WOB platform is to create opportunities for AI researchers to begin tackling these problems. Concretely, the platform aims to address the following: (1) a lack of data that covers the more interesting bugs video games exhibit; (2) control over when, where and how these bugs manifest; (3) the lack of reproducibility and benchmarking in ABD research; and (4) to support the further development of new and interesting bugs. Satisfying these aims is an ongoing effort as video games and the bugs they exhibit are extremely diverse, so it would be difficult to capture all of this diversity right away. This chapter presents the progress thus far.

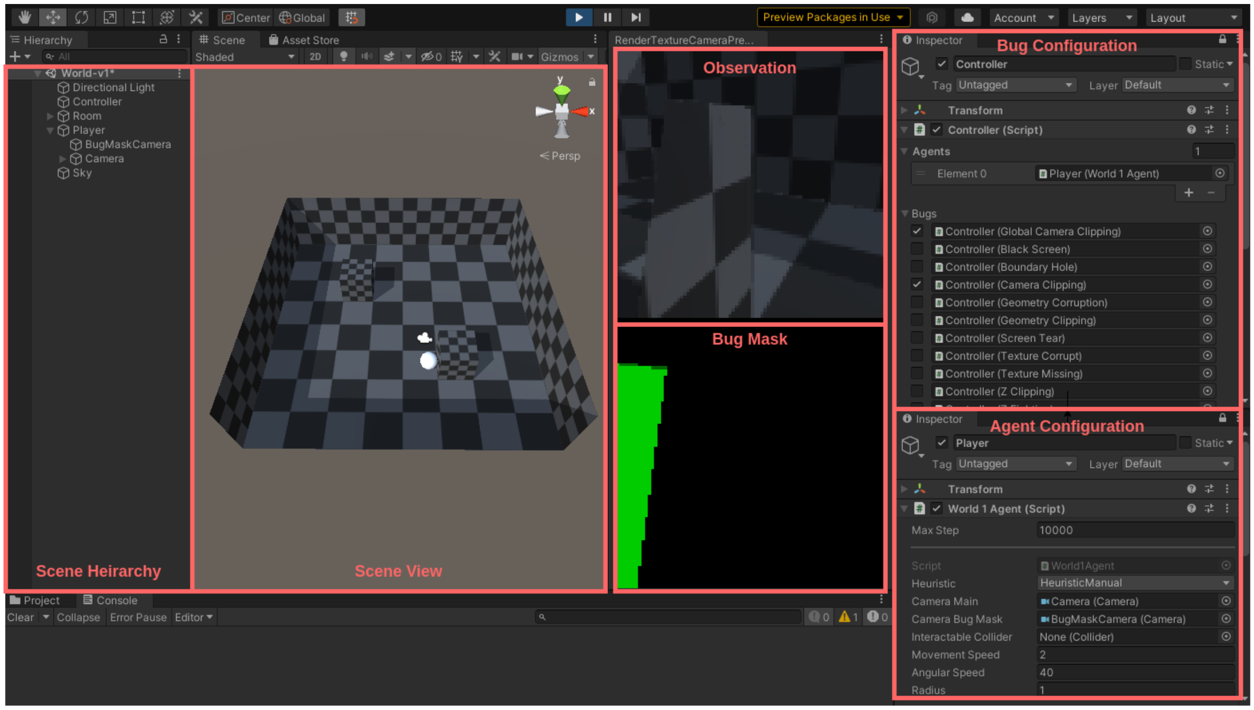
The chapter is structured as follows. Section 4.1 forms the main body of the chapter and is where details of the platform itself are presented. These include details of the environments and bugs it implements, the various architectural and design decisions that were made and why, and a brief guide on using the platform and its API. In section 4.2 we make an initial attempt at applying the ideas presented in chapter 3 to perform regression testing of the WOB platform itself. Specifically, we test one of the WOB benchmark environments with a simple learning-based identification method. Finally, in section 4.3 the chapter is summarized, and limitations and future directions are discussed.

4.1 Platform Overview

This section presents an overview of the WOB platform. WOB is an open experimental platform built with the Unity engine and the ML-Agents package (Juliani et al. 2018). Unity is a popular game engine that is widely used in industry (Wikipedia contributors 2022), it has a rich ecosystem and an intuitive and extensible interface that WOB takes full advantage of (see Fig. 4.1). ML-Agents (Juliani et al. 2018) is a Unity package

---

[1] Some progression related bugs (or stuck bug) are very hard to identify, but works tend to trivialize this bug by artificially locking the agent in place. This is discussed further in section 4.1.2.



Controller — Static ☐
Tag Untagged — Layer Default

Controller (Script)

Agents 1
Element 0 Player (World 1 Agent)

Bugs
☑ Controller (Global Camera Clipping)
☐ Controller (Black Screen)
☐ Controller (Boundary Hole)
☑ Controller (Camera Clipping)
☐ Controller (Geometry Corruption)
☐ Controller (Geometry Clipping)
☐ Controller (Screen Tear)
☐ Controller (Texture Corrupt)
☐ Controller (Texture Missing)
☐ Controller (Z Clipping)

Player — Static ☐
Tag Untagged — Layer Default

World 1 Agent (Script)

Max Step 10000
Script World1Agent
Heuristic HeuristicManual
Camera Main Camera (Camera)
Camera Bug Mask BugMaskCamera (Camera)
Interactable Collider None (Collider)
Movement Speed 2
Angular Speed 40
Radius 1

Figure 4.1: Unity interface showing the agent's observations and the configuration for bugs and an agent. In this example the camera clipping bug is enabled allowing the agent to see inside some geometry, the bug is rendered in the bug mask (described in section 4.1.1).

that supports training agents to play games, it comes with its own built-in environments, but we opted to design our own to better support some of the more complex kinds of bugs. ML-Agents primarily supports the exploration side of WOB and opens it up to the Python machine learning ecosystem. It contains many of the useful abstractions that ease the development of software agents such as sensors and actuators. WOB adds some additional functionality to the implementation of these abstractions including Unity components for heuristic behaviours and environment configuration options, among others. Details can be found in the platform documentation.

The current focus is on 3D first person games as a popular genre in the industry, and one that will provide a substantial challenge to modern machine learning and anomaly detection methods. The platform's core contribution is the implementation of growing collection of common video game bugs (Levy et al. 2010; Lewis et al. 2010). These include many of the simpler bugs found in works we have discussed in earlier chapters, including freezing, getting stuck and player out of bounds. But also some of the more interesting and complex bugs such as Z-fighting, camera clipping, geometry clipping, and geometry corruption. The platform also provides a collection of pre-built video game environments, agents and datasets that researchers may use to conduct experiments and compare approaches. All of these features are used in later chapters to

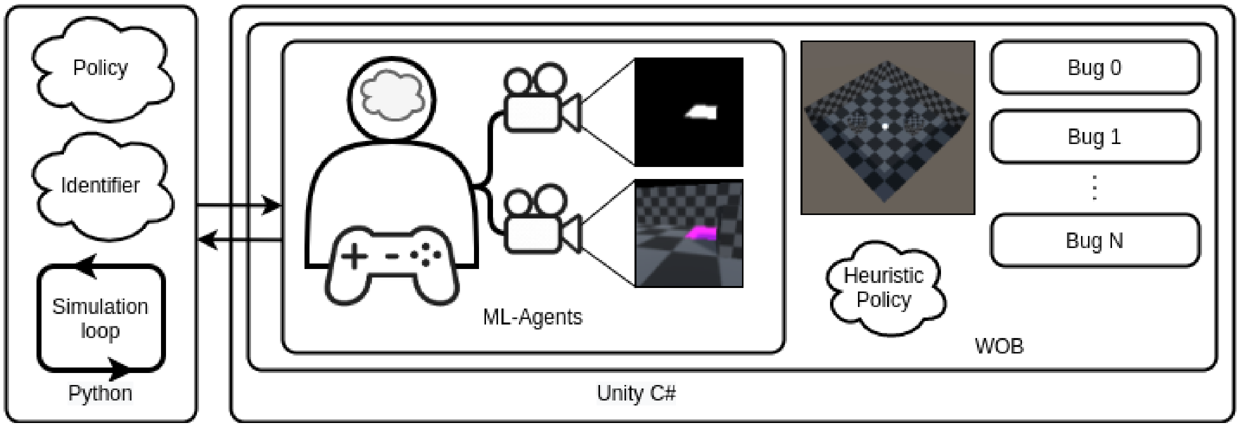


Figure 4.2: Architecture overview.

train and evaluate testing agents. Details on each of these features can be found in the sections to follow, further examples and details can be found in the platform documentation.

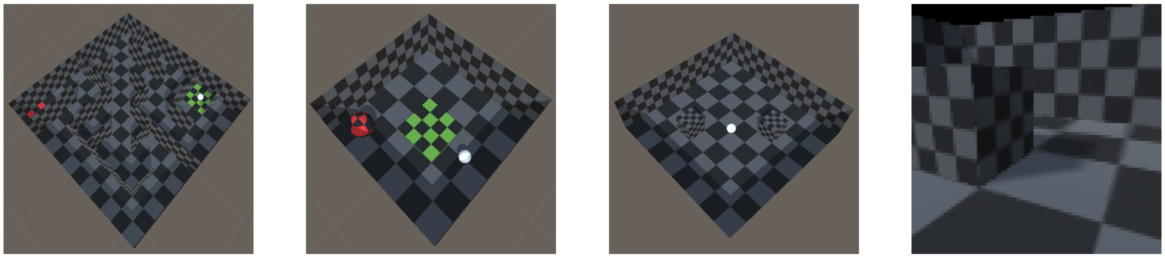
The platform is available as a Unity package on the open source Unity package registry OpenUPM, on the Python package index PyPi and on GitHub. Further links can be found here (Benedict Wilkins n.d.).

4.1.1 Agents

The platform currently implements first-person agents by default. These agents have three sensors: a main camera, which renders a view of the scene as a human player would see it; a bug mask camera, which renders a mask over the scene showing in which regions there is a bug present; and a sensor which records various environment/agent properties such as the agent's position and rotation. The camera views are shown in Fig. 4.1. The bug mask acts as a label for the agent's observation and is instrumental in enabling machine learning to be applied to the bug identification problem. It is discussed in more detail in section 4.1.3. Providing both visual observations and state information gives flexibility in the methods that might be applied. Using state information rather than visual observations makes the problem of search easier in cases where researchers wish to focus on identification.

In an attempt to standardize the comparison of explorative agents, WOB also implements a set of common actions including movement, jumping, view changes and simple event-based interaction, which can be configured during setup. Agent behaviours, goals and bug identification models may be specified in Python or in C# (the default platform language) with the support of ML-Agents and associated Python packages.

As a simple baseline, and to support work in later chapters a simple navigational agent behaviour has been



(a) Maze-v0 (b) GettingStuck-v0 (c) World-v0 (d) Observation

Figure 4.3: Example environments implemented in the platform. The agent appears as a white sphere and views the environment in first person as seen in (d). The agent can navigate these environments by moving and rotating its perspective. Only (a) `Maze-v0` has a defined goal for the agent, to navigate the maze and reach the goal. The other environments are open-ended, the goal is simply to explore and learn.

developed. The behaviour is implemented in C# as part of the platform and uses Unity's built-in navigation system inspired by (Prasetya et al. 2020). The behaviour is simply to move and look around, picking random points in the scene to navigate toward. Specifically, it selects a random point within a pre-defined front cone, refines this point to ensure it is reachable, computes the shortest path and moves to the closest point on this path. At each iteration of the simulation, the agent will take one of the available actions `forward, turn_left, turn_right, noop`[2], or take a random action (including `noop`) with some probability. The agent can be used in simpler environments where extensive exploration, or puzzle solving is not necessary to ease experimentation with approaches to bug identification. In addition, WOB provides agents that can be controlled manually and that may be used for debugging new environments or bug implementations, or for generating training data. Our hope is that with time subsequent research in this area will make use of WOB and expand the collection of agents, bugs and environments.

4.1.2 Environments

At this time WOB has a number of built-in environments, some examples are shown in Fig. 4.3. They provide a stable testing ground for ABD approaches and have been designed either to exhibit particular bugs, or to simplify certain aspects of the ABD problem. For example, the `World-v0` environment consists of a single static room in which the agent is able to move and look around. This environment should be used to test approaches to identifying visual bugs as exploration is relatively straightforward. In addition to the built-in environments currently available, new game environments can be built for testing specific kinds of behaviours or to support new kinds of bugs. To make this process easier, the bugs currently implemented in

---

[2] `noop` stands for no-operation, or *do nothing*.

the platform have been designed to be as generic as possible (relying only on common abstractions provided by Unity) and can be added with minimal setup to new environments. The platform's current focus is on 3D first-person games and many of the bugs reflect this, any new game environments that are not of this genre are not explicitly supported. Three of the built-in environments are described briefly below, other environments and further details can be found in the platform documentation and in Appendix. B.3.

Environment: World-v0

The simplest of the built-in environments, it consists of a single room with two obstacles. The agent is free to move and look around the room. There are a number of bugs implemented in this environment, including camera clipping, screen tearing, texture corruption, terrain hole and geometry clipping, see Fig. 4.6 and Fig. 4.3 (a). As exploration is trivial in this environment, the focus should be on bug identification. This environment is used in section 4.2 as part of an initial exploration of learning for regression testing.

Environment: Maze-v0

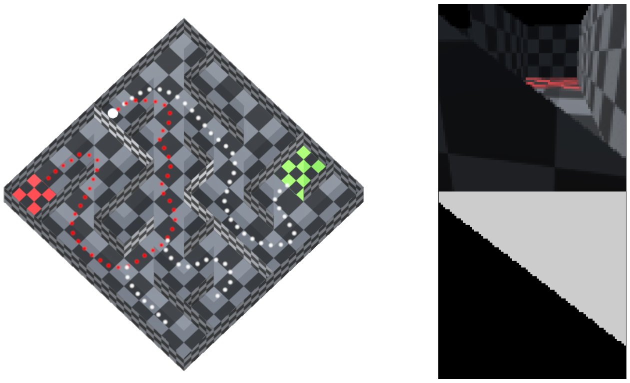
The agent is tasked with navigating a maze and is free to move and look around. There are three kinds of bugs present: (1) invalid information access - the agent can see through walls by clipping its view through the wall geometry, this can reveal the maze structure in advance and allow the agent to reach the goal more quickly, see Fig. 4.4; (2) geometry clipping: some walls are not part of the collision system, the agent can freely pass through them constituting a shortcut to the goal; and (3) player out of bounds, the agent can escape the maze by again passing through missing wall geometry. A smaller version of this environment (named `Maze-v1`[3]) that implements many more bugs is used in the experiments presented in chapter 5.

Environment: GettingStuck-v0

In this open-ended environment there are two objects, a ball which can be pushed around by the agent, and an elevator which the agent may stand on. The agent may get stuck in one of two ways, outlined in Fig. 4.5. Each bug requires specific sequences of actions to manifest and permanently prevent the agent from returning to floor level. The kind of stuck bug implemented here is more sophisticated than those investigated in related works. For example, in (Gordillo et al. 2021) surfaces were introduced at certain locations that *froze* the agent in place. Although this kind of stuck bug could happen in a real game, the stuck bugs implemented in `GettingStuck-v0` are much more realistic. In this environment the agent is not

---

[3] see Appendix. B.3.2



(a) Maze-v0 Invalid Information Access (IIA) bug

(b) Observation (top) & Mask (bottom)

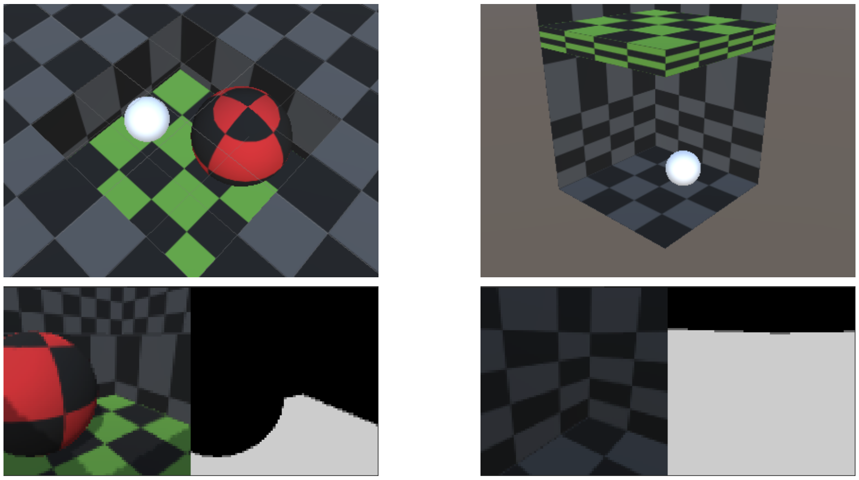
Figure 4.4: (a) shows a birds eye view of the `Maze-v0` environment with some possible paths to the goal. The agent reaches a state where it is able to see the goal pre-emptively by clipping its view through a wall in the maze. The agent's observation and the associated mask at this critical state is shown in (b). The agent can use this additional information to skip subsequent exploration of the maze and in this case head directly to the goal.

stuck in a specific location and can still move around in some restricted part of the play area, this makes identification vastly more challenging.

4.1.3 Bugs

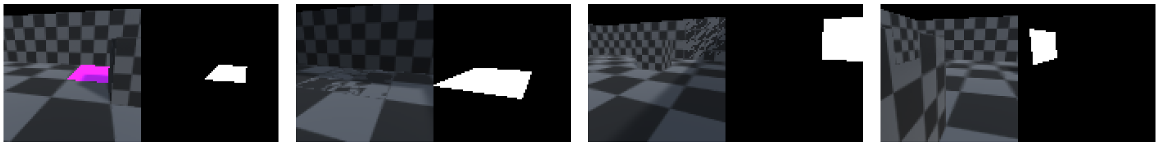
Bugs are implemented as collections of C# scripts and shaders, and are designed to be as generic as possible by relying on the abstractions provided by Unity (component system, game objects, etc.). It is up to an environment which bugs it implements. The bugs that are implemented thus far are primarily visual, specifically issues with graphics, such as texture or geometry corruption. Graphical issues tend to be challenging to identify with simple rule-based approaches but lend themselves nicely to learning and vision. Some more complex logical issues, such as getting stuck and invalid information access have also been implemented in specific environments (see Fig. 4.5).

In order to support learning-based identification, and to ensure that approaches can be properly evaluated and compared, bugs need to be labelled as manifest or not during play. There are therefore two considerations when implementing a bug: (1) how to manifest the bug, and (2) how to label this manifestation. With the wide range of bugs that are possible in video games labelling is a tricky business and there is no single universally applicable solution. Nevertheless, we give some simple guidelines that should be considered when implementing a new bug:



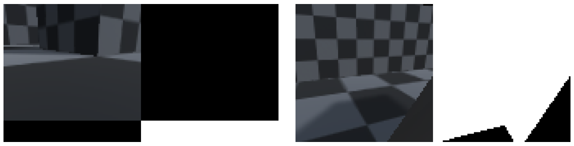
(a) Getting Stuck Example 1 (b) Getting Stuck Example 2

Figure 4.5: The two bugs that are present in the `GettingStuck-v0` environment. (a) is a complex bug stuck bug that occurs when the red chequered ball is pushed by the agent onto the moving green chequered elevator. The elevator stops moving when this happens. If the agent subsequently drops onto the elevator when the platform has stopped far enough below the floor, then it has no way of escaping. (b) is similarly a stuck bug that occurs if the agent moves underneath the elevator, the agent will fall down the shaft and not be able to return. The bugs are labelled by rendering relevant game objects in the bug mask. For (a) this happens to be the elevator, but could also be the ball or surrounding walls as in (b). These bugs represent typical scenarios where a player can get stuck due to game mechanics. The player would have to reload the level/game to escape. Identifying these kinds of progression issues is challenging and to the best of our knowledge is not something that has been attempted in the literature.



(a) Texture missing (b) Texture corruption (c) Z-fighting (d) Z-clipping

(e) Geometry corruption (f) Screen tearing (g) Black screen (h) Camera clipping



(i) Terrain hole (j) Geometry clipping

Figure 4.6: Examples of bugs implemented in `World-v0`, each image shows an observation (left) with its associated mask (right). Each bug is described in more detail in the platform documentation and bug glossary.

- (G1) A bug should be labelled as manifest if and only if it is present in the agent's observation.
- (G2) A bug should be labelled as manifest for its full duration if temporally extended.
- (G3) It is possible that multiple bugs manifest simultaneously, any label (scalar or otherwise) should retain this information.
- (G4) A labelling should convey as much information about the underlying issue as possible.

Following these guidelines, and with the focus on 3D first-person games, bugs manifest in the agent's main camera and are labelled as mask over this visual observation, in the bug mask camera. The choice to label as part of a mask over a visual observation has been made with generality (G4) in mind. There is a trade-off to be made with how much information should be included in a label, incorporating more information tends to make implementing new bugs more difficult, and more complex label formats (e.g. hierarchical) may might make it difficult to work with. Using a mask over the image observation strikes a balance, it is generally straightforward to implement using the graphics pipeline, provides more information than a scalar label and is fairly easy to work with downstream.

In many cases it is not obvious how to label a bug in such a mask, ultimately it is a design choice. Consider for example, a misbehaving NPC, this complex temporally extended issue can be reasonably labelled by masking the NPC's body, with the labelling being updated over time (G2). Consider instead, a cut-scene that fails to show. Things are less clear, perhaps the whole observation should be masked. For bugs such as invalid information access, the source (or a proxy indicator) of the information can be masked. An example of this can be seen in the `Maze-v0` environment, see Fig. 4.4.

To aid in the implementation of new bugs, WOB provides some useful tools for labelling in the mask. The first of these is a tagging system. Every bug has a tag, a unique identifier which determines its colour in the bug mask. Bug tags can be used to label bugs that are associated with particular objects in the environment, such as missing textures or corrupted geometries. A game object can be tagged, which amounts to notifying the shader associated with the mask renderer that the object should be rendered. Tagging is perhaps the most widely applicable method of labelling in WOB. The mask renderer will also automatically render back-side geometry. If and when the agent can see inside a geometry, the inside faces are rendered in the mask.

Bugs such as screen tearing, flickering, and freezing, are slightly more difficult to label but will typically involve a post-processing step in the rendering pipeline to directly modify the final bug mask. WOB currently uses Unity's built-in rendering pipeline to achieve this, see for example, Fig. 4.6 (f). The mask renderer can optionally render the sky-box below a certain height and may be used to label terrain hole or player out of bounds bugs, see Fig. 4.6 (i).

The label mask can be used in a number of ways, for example, to test different inductive biases for learning models, for segmentation or classification, and in model evaluation and comparison. A full list of bugs can be found in the platform documentation and in the bug glossary.

4.1.4 Interface & Python API

To better manage experiments and open the platform up to the Python machine learning ecosystem, WOB exposes a Python API that follows the OpenAI Gym standard (Brockman et al. 2016) with some additional features, including the ability to enable/disable bugs, set agent behaviours and configure the Unity environments at run-time. While learning agents (specifically those that make use of deep learning) are supported by ML-Agents in C# to some extent, there is no explicit support for identification and many useful libraries are missing. With its rich ecosystem of machine learning libraries, Python is the recommended language for developing agents for testing within the WOB platform.

To test an agent implemented in Python, an environment is created in the usual way with the Gym interface (`gym.make`). The platform can be used in two modes courtesy of ML-Agents: in or out of the Unity editor. Providing an environment name and version e.g `"World-v0"` will create the environment from a pre-built binary. These binaries are made available in the platform releases. Example code that configures and runs the `World-v0` environment via the Python API is shown below.

```python
import gym
env = gym.make("WOB/World-v0")                            # initialise environment
env.set_agent_behaviour("Python")                         # use Python agent
env.enable_bug("TextureCorruption")                       # enable bug
agent = ABDAgent(env.observation_space, env.action_space) # initialise agent
observation, info = env.reset()                           # next episode
while not done:                                           # simulation loop
    agent.classify(observation, info['Mask'])             # is there a bug?
    action = agent.policy(observation)                    # next action
    observation, reward, done, info = env.step(action)    # next state
```

In the second mode, where no environment ID is given to `gym.make`, assuming the Unity editor is open, the current scene will be used as the environment. While the WOB API provides some control, in this mode more fine-grained control over the environment via the various editor menus is now possible (see Fig. 4.1) and is recommended for developing new environments. Both modes are provided as part of ML-Agents with

Platforms			
Name	Ref	I	S
SC2LE	(Vinyals et al. 2017)	✗	✓
Dopamine	(Castro et al. 2018)	✗	✓
DeepMind Lab	(Beattie et al. 2016)	✗	✓
DeepMind CS	(Tassa et al. 2018)	✗	✓
Minihack	(Samvelyan et al. 2021)	✗	✓
Mujoco	(Todorov et al. 2012)	✗	✓
Petting Zoo	(Terry et al. 2020)	✗	✓
Pygame LE	(Tasfi 2016)	✗	✓
ALE	(Bellemare et al. 2013)	✗	✓
ML-Agents	(Juliani et al. 2018)	✗	✓
OpenAI Gym	(Brockman et al. 2016)	✗	✓
WOB	(Wilkins et al. 2022)	✓	✓

Figure 4.7: Available platforms, libraries or environments, and how they might support ABD research. I = support for Identification i.e. environments that contain bugs explicitly. S = support for Search. References are only given if the platform is public and freely available.

a custom configuration API and editor menus provided by the WOB platform.

4.1.5 Existing Platforms & Datasets

Since the explosion of research interest in fields such as Reinforcement Learning (RL), a number of datasets, platforms and libraries have appeared, see Fig 4.7. Many of these contain examples of video games as environments for training and testing game playing agents. Perhaps the most notable is OpenAI's Gym (Brockman et al. 2016) which forms part of WOB. Gym exposes a common API which has become a standard in the field and interfaces with a wide variety of environments. The large collection of Atari 2600 games provided by the Arcade Learning Environments (ALE) (Bellemare et al. 2013) is of particular interest as examples of relatively challenging *real* (albeit outdated) video games. DeepMind Lab (Beattie et al. 2016) is another platform of note that provides support for training RL agents in 3D environments. Although many of these platforms were not built for ABD research, some have been used to training game playing agents in this context (e.g. (Wilkins et al. 2020)). They are of limited use in identification as they don't (intentionally) contain bugs.

Benchmarking is one of the core motivations for the platform. The environments themselves are important in this, particularly for exploration. However, static datasets may be more appropriate for benchmarking identification. As discussed in chapter 3, there is an important interaction between search and identification - the data used to train (and test) identification models are collected by an agent. To ensure benchmarking is fair this data should be unchanged across experiments. The simplest way to achieve this is to generate a static dataset using an agent with a particular policy. The use of static datasets is further motivated by ease of use. It is computationally expensive to generate fresh data, meaning longer experiment times. Static datasets allow ML practitioners to try their hand at identification with a familiar set up and at minimal cost.

With these considerations in mind, WOB has explicit support for generating datasets given an environment and an agent. A collection of datasets have been generated using the built-in agents and environments. The datasets are large (up to 500k) collections of labelled observations and actions. Links to datasets can be found in the platform documentation.

These datasets are to the best of our knowledge the first of their kind. A thorough search of the relevant literature suggests that there are very few datasets that are suitable for training agents[4], see the list below:

- Politowski et al. 2020 provides a dataset of video game *problems*, these are curated bug reports. The dataset cannot be used to train bug identification models and is instead meant to inform human testers. It could perhaps be used to develop intuition for good inductive biases.
- Wilkins et al. 2020 provides a dataset of bugged and bug free observations collected in 7 Atari 2600 games made available by ALE. Bugs are introduced directly by modifying pixel values. Without access to a games source code it is difficult to generate realistic issues. This dataset is used in experiments we perform in chapter 5.
- Chen et al. 2021a provides a similarly labelled dataset, with some observations being modified at a pixel level and some genuine bugs captured from gameplay. The dataset looks promising, but further inspection of the data revealed some potential problems and the authors have neglected to respond to an inquiry.
- Liu et al. 2020 provides a dataset of simple GUI issues in form-like Android applications. These are generally much simpler than those seen in video games, but the data is very useful for ABD in their setting, and to our knowledge is the only one of its kind.

4.2 Regression Testing World of Bugs

Like any software, WOB is susceptible to bugs and requires testing. The stability of the built-in environments and bugs are of particular interest. It is important that even as the platform code is updated they remain

---

[4] Recently (at the time of writing) (Taesiri et al. 2022) has curated the GamePhysics dataset to support their work. It looks very promising, but it does not appear to be publicly available.

(a) Forza Horizon 5 - rainbow texture bug (b) Grand Theft Auto - bump map bug (c) Terraria - texture alpha bug (d) Oblivion - level of detail bug (e) WOB - UV corruption bug

Figure 4.8: Examples of different texture corruption bugs.

stable to ensure researchers can fairly compare their approaches. This section explores a very simple learning-based approach to regression testing the WOB platform. The experiments are not a serious attempt at the bug identification problem (this can be found in the next chapter), but highlights some of the challenges that were outlined in the last chapter. In the experiment, the bugs that are implemented in the platform are actually features to be tested, the aim is to ensure that these features work as intended following an update to the wider code.

Texture Corruption: An Example

The environment that is considered is `World-v0` and we wish to test whether the texture corruption bug is working correctly. Texture corruption is a simple and common bug that appears during development. The issue can be rooted in a number of places, ranging from corrupted source image files or UV maps to issues with the rendering pipeline or shaders. Examples are shown in Fig. 4.8. Given the variety of texture corruption bugs and their graphical nature, writing comprehensive tests can be difficult and it is often left to human testers to find the issue.

Here the texture corruption bug is treated as a feature as we are interested in testing the platform. The effect of enabling the bug should be kept constant for a particular game environment as the platform is updated. Perhaps the simplest issue that might arise after an update is that texture corruption fails to manifest after the API command `env.enable_bug("TextureCorruption")`. Enabling texture corruption should cause a randomly selected texture on a particular game object to corrupt, a relatively simple test can be written to check for this as follows:

```
def test_texture_corruption():
    env = worldofbugs.make("World-v0")
    env.enable_bug("TextureCorruption")
    observations = collect_data(env, agent)
    score = agent.classify(observations)
    assert score < SCORE_TRESHOLD
```

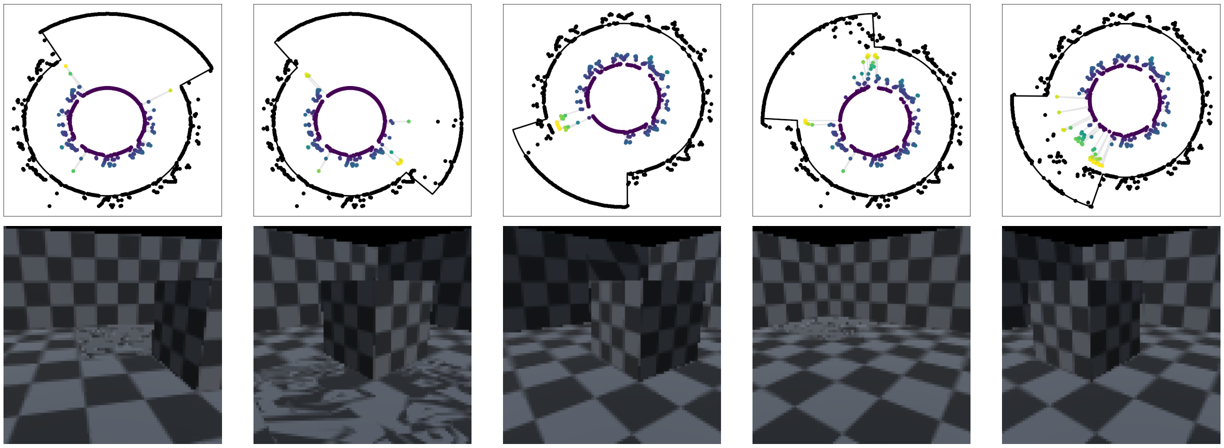
Experimental Setup

Since we have the luxury of a stable version of `World-v0` and access to a working bug mask, a supervised learning approach to identification may be used. A classifier can be trained to identify the texture corruption in the agent's observation using the mask as a label. The test will provide the classifier with new (post-update) observational data, if the classifier fails to properly identify the texture corruption the unit test should fail. This sounds straightforward, but there is a lurking problem. After the update an environment might *break* in any number of ways, not just in the way we are expecting. This can impact the classifier's estimate in unexpected ways. The problem of checking whether the texture corruption has correctly manifest is therefore also a problem of checking the overall stability of the environment. This is explored in the experiments to follow, where the goal is ultimately to create a robust test for texture corruption as a *feature*.

Given the simplicity of the `World-v0` environment, a policy that rotates the camera 360° is sufficient to allows the agent to view every part of the environment. A simple AlexNet-like (Krizhevsky et al. 2017) classifier is trained on data collected in 100 trajectories for 100 epochs. To avoid problems with imbalanced data, only the unique images are used in training, totalling approximately 20k observations. Observations are labelled (1) if a corrupted texture is present and (0) otherwise. The following issues are introduced by updates to the platform after training:

- (B1) Global illumination changed.
- (B2) Agent starting position changed.
- (B3) Corrupted texture failed to manifest at all.
- (B4) Texture corruption was not rendered in the mask.
- (B5) A missing texture bug is also present in the episode.

(B1) and (B2) are systemic issues that change every observation, (B3-5) are local and only impact specific observations. Since we are testing texture corruption, (B3) and (B4) are of particular interest. (B1), (B2)



(N1) (N2) (N3) (N4) (N5)

Figure 4.9: Classification performance visualised for five unseen test trajectories. The five radial plots in the top row each show the the agents prediction and classification error for a full rotation of the agent (360°) as it gathers and classifies observations. The inner coloured ring shows the classification error: purple = low error, yellow = high error. The stepped outer ring shows the agent's prediction, larger radius = 1 (texture bug present), smaller radius = 0 (bug not present). Each point plotted represents a single observation made by the agent. The solid black line in the stepped outer ring shows the ground truth. The plots show that the agent is proficient at classifying the texture corruption bug from visual observations.The bottom row shows a single example observation containing a texture corruption bug for each of five test trajectories. Note the subtlety of (N5), the texture has been flipped horizontally and is occluded.

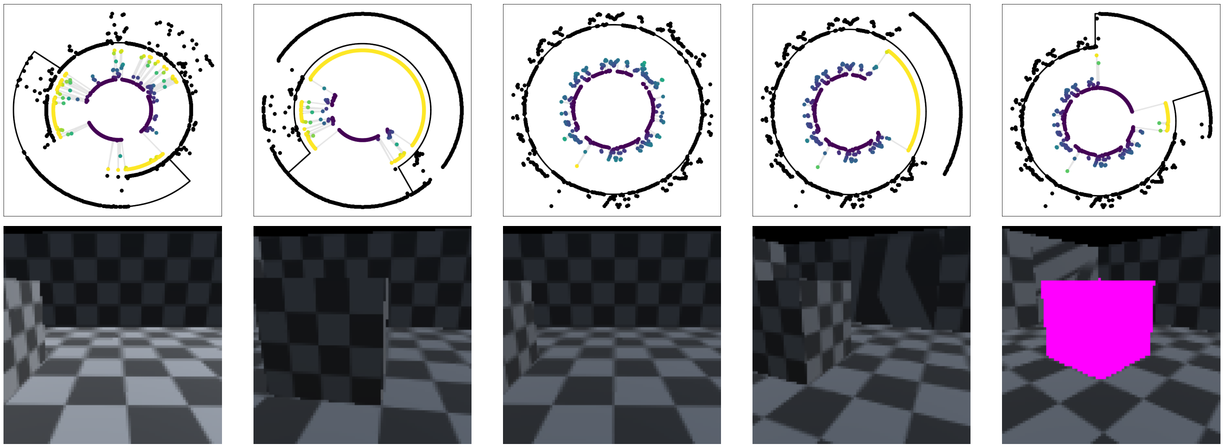
and (B5) are used to check robustness.

Results & Discussion

Performance on normal unseen test episodes is visualised in Fig. 4.9. Each episode contains one instance of a corrupted texture (across multiple observations) with no platform update. The classifier performs well on all test trajectories, only misclassifying on some particularly challenging examples at points where the texture corruption is not in full view or manifests in an unusual way. This bodes well for supervised learning approaches to the bug identification as it shows that a relatively simple classifier can be used to identify bugs from visual observations.

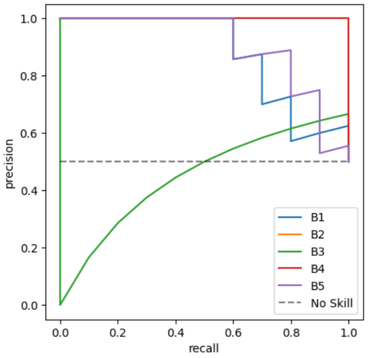
Performance on normal unseen post-update test episodes is visualised in Fig. 4.10. It is clear that the performance after an update is worse, with (B3) being the exception. The question is now, what score should be used to measure the abnormality of each episode. One possibility is to simply measure the per-episode classification error. This gives the result presented in Fig. 4.11.

Clearly, the per-observation classification task is not a good way of identifying (B3). One could try to further classify episodes by looking at the per-observation error statistics, but better may be to design



(B1) (B2) (B3) (B4) (B5)

Figure 4.10: Classification performance visualised for five unseen test trajectories that contain the following bugs: (B1) Global illumination changed, (B2) Agent starting position changed, (B3) Corrupted texture failed to manifest at all, (B4) Texture corruption was not rendered in the mask, (B5) A missing texture bug is also present in the episode. The five radial plots in the top row show predictions and classification error as described in Fig. 4.9. It is clear that the agents performance is poor on these unseen bugs. As discussed previously, a common strategy for classifying anomalies is to use performance to identify anomalies. With the exception of (B3) this would be an effective way to identify problems here. This is discussed further in section 4.2, see also Fig. 4.11 for numerical results.



(a) Precision Recall Curves

B1		
	A. 1	A. 0
P. 1	7	1
P. 0	3	9

B2		
	A. 1	A. 0
P. 1	10	0
P. 0	0	10

B3		
	A. 1	A. 0
P. 1	10	5
P. 0	0	5

B4		
	A. 1	A. 0
P. 1	10	0
P. 0	0	10

B5		
	A. 1	A. 0
P. 1	8	1
P. 0	2	9

(b) Confusion Matrices

Figure 4.11: Results when using classification error as a normality score. (a) shows precision-recall curves for each bug (B1-B5) and (b) shows associated confusion matrices (A. = Actual, P. = Predicted). The trained classifier was tested on a 20 episodes (representing the running of a regression test 20 times for different data) for each bug, 10 of these were *pre-updated* (they contain no bug) labelled 0, and 10 were *post-updated* and contain a bug and are labelled 1. The normality score for a episode is computed as the sum of classification errors for each observation in the episode. This score is used to classify the episode and ultimate determine whether the regression test *passes* (0) or *fails* (1). The pre-update scores are computed over unseen episodes rather than over the training data. This is because the training error is going to be smaller than the error for these unseen episodes. In practice, it is better to use a threshold based (conservatively) on validation error, which is effectively what these results are showing. See section 4.2 for discussion.

a classifier for entire episodes. This would be much more computationally expensive, but might also deal with a problem present in (B5). The problem with (B5) is that the texture missing bug may be hidden if it appears in the same observations as the texture corruption bug (as shown in Fig. 4.10). The classifier appears to confuse the missing texture for a corrupted texture. Another option for correctly identifying (B5) would be to attempt to predict the entire mask rather than just a scalar label.

(B1) and (B2) offer some insight into how key variable changes such as these might impact an agents performances. It points to the fact that small changes in the input distribution can have a dramatic impact on predictions. In this case it leads to good performance as we are actively trying to identify them as problems. But more generally the update may be an intended one and the agent would need to take this into account, for example by learning *invariance* to certain changes. For (B2) this might be done simply by having a more sophisticated explorative strategy, making the model invariant to the point of view or position of the agent. (B1) is harder to deal with, especially if lighting issues are a potential concern since being completely invariant to lighting conditions may prevent a model from identifying unintended changes. Dealing with intentional changes in general is a very challenging problem, and is one that is not addressed in this thesis despite our focus on regression testing. First, it is important to ascertain whether novelty detection may be used to identify bugs and what kinds of methods are applicable, this is our focus.

4.3 Conclusions

In this chapter World of Bugs was presented as a platform that supports the training and evaluation of ABD agents. The platform in its current state satisfies the aims and objectives set out in the introduction of the chapter. (1) The platform provides a set of diverse bugs with supervision in a number of non-trivial environments. (2) The platform interface and API allow control over when, where and how these bugs manifest. (3) A number of datasets have been generated that will support benchmarking approaches to identification. The environments that are currently implemented provide a means of experimenting with agents that can both play and identify bugs. (4) The design guidelines and tools that have been implemented and presented in this chapter should make it easier to implement new bugs in the platform.

In the later part of the chapter, we gave a demonstration of the use of testing agents with learning capabilities for regression testing the platform itself. The demonstration highlighted many of the important issues raised in chapter 3. Namely, the importance of choosing a good measure of normality, the trade-offs in working with observations or entire episodes and the potential issues faced by intentional changes that introduce distributional shift such as lighting. In the chapters to follow the platform will again be used in experiments where we will attempt to highlight the benefits of using learning for the purposes of testing, namely, capability and reusability.

4.3.1 Current Platform Limitations & Future Work

The platform is meant only as a means to experiment with different approaches to ABD. The practical deployment of these models is still an open issue which the platform does little to address.

With regard to the practical operation of the platform, although ML-Agents supports parallel agents running in the same instance of Unity, this is not currently supported by WOB as OpenAI gym supports only single agent interactions. It is however possible to create multiple instances of any WOB environment running as separate processes to enable parallel training. Fine-grained control over properties of the built-in environments via the Python API (beyond enabling bugs and changing behaviours) is also desirable.

The focus of WOB is currently on bugs that manifest visually (see Fig. 4.6) and other bugs that impact game mechanics, or progression bugs such as getting stuck. Issues with audio as another important output modality are not currently supported, although many of the same design decisions apply. A similar (1D) mask could be used to indicate issues.

The bugs implemented only scratch the surface of the vast assortment of bugs that may be exhibited more generally. Expanding this list with the help of the ABD research community is left as ongoing future work. There are plans to grow the list of available bugs into more challenging areas such as in physics, narrative, NPC behaviours, more complex invalid information access bugs and those that are more game specific. For this to happen, richer and more complex game environments are required. Again, the hope is that over time the community will provide more interesting environments that exhibit a wider variety of bugs.

Chapter 5

Contrastive Learning for Automated Bug Identification

This chapter is an extended version of our paper: *A Metric Learning Approach to Anomaly Detection in Video Games* (Wilkins et al. 2020). The focus is on regression testing as outlined in chapter 3. We are interested in developing capable and reusable agents that can be trained to identify a broad range of bugs using weak supervision.

In pursuit of this, we develop State-State Siamese Networks (S3N), a contrastive learning algorithm that attempts to learn the structure of an agent's experiences. S3N maps observations, which in this chapter are the fully rendered images that would be seen by a human player, to a low dimensional embedding space in which the distance between embedded observations are representative of their temporal relationship. If observations are temporally close in the agent's experience, then they are close in the learned embedding. The *closeness* of subsequent observations may be used as a measure of their novelty (or abnormality). Novel observations tend to be further from each other than observations that have been seen during training.

In our thorough experimentation and investigation of S3N and its ability to identify bugs, we demonstrate many of the benefits of learning-based approaches to testing, namely: reusability (through decoupling) - not game specific and game implementation independent; applicability - a single test can identify a range of bugs; capability - can identify bugs that are otherwise difficult to identify with guards. We additionally show empirically that the approach can be used to identify systemic bugs, which are in general much harder to identify than bugs that occur on a per-observation or transition level. The bugs we explore include: geometry clipping; unintended shortcuts; player out of bounds; rich graphical bugs like those presented in chapter 4; and freezing and lag.

In an attempt to familiarize the reader with key concepts, the chapter begins with background on self-supervised representation learning. In section 5.2 we present the theory and algorithm behind S3N. Then in section 5.2.1 we investigate the embeddings S3N produces and their properties. This is in an attempt to verify the claim that closeness, or *distance* can be used as a normality score. We then present experiments that aim to identify bugs in environments of increasing complexity, along with a detailed empirical analysis in section 5.3. Finally, we discuss related work in 5.4, and summarise in 5.5.

5.1 Self-Supervised Representation Learning

Learning representations is one of the most important problems in the study of AI. Broadly, it is the automatic extraction of features[1] via some optimization or learning procedure. *Features* is a broad term that refers to any regularity, symmetries, patterns, or characteristics of data. Perhaps the most famous example of representation learning is simply in the use of neural networks. The network parameters and their interactions encode information about the task, or the relationship between inputs and outputs (Rumelhart et al. 1986). This kind of representation reflects the dataset or task as a whole. In problem domains such as data compression and dimensionality reduction the interest is in learning representations of particular observations. Rather than looking at a model's parameters, the model is trained to produce a representation of each input observation. That is, it learns a transformation $f_\theta : \mathcal{X} \rightarrow \mathcal{Z}$ where $\mathcal{Z}$ is often referred to as an embedding space, or latent space.

In this regime, the focus is on learning representations that are most useful for some downstream task, which might for example be decision-making, classification, or the identification of novelty or abnormality. Determining how best to do this is an ongoing research endeavour and might be considered one of the central problems in AI. There is relatively broad agreement as to what properties are desirable, for example, as described by terms such as *invariance*, *disentanglement* and *causality* (Bengio et al. 2014). We will touch briefly on these topics in this chapter, and dive into more depth regarding causality in chapter 6.

As we have already stated, in this chapter our primary focus is novelty. Novelty is fundamentally about representation. It asks about *similarity* or *equivalence* - how *similar* is my new observation x to my prior observations or experience. The key is in choosing or *learning* a similarity measure that is suited to the

---

[1] not be confused with features as in intentional updates to a video game.

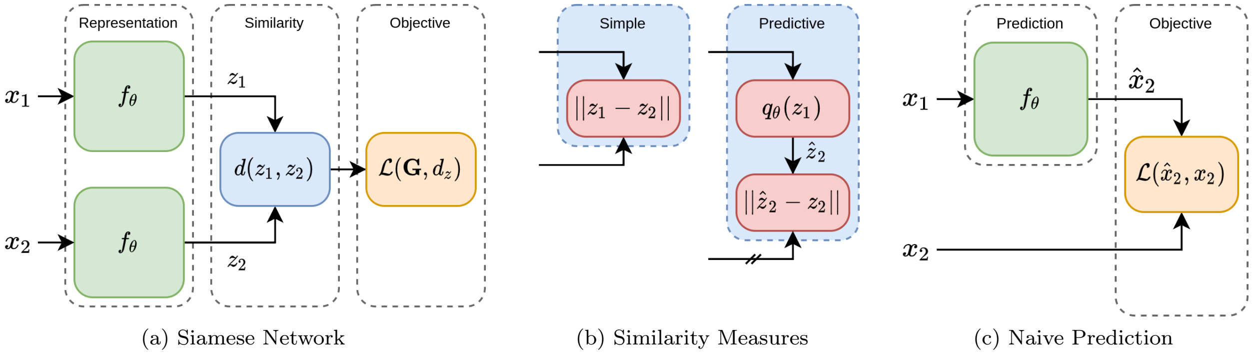
task at hand. As exemplified by our pebble example in chapter 3, at one extreme, novelty is defined as non-equality, meaning that any new observation that does not exactly match a prior observation is identified as novel. Clearly, this is not desirable for our setting, there is a balance to be found - an agent should be *invariant* to, or group together certain features of an observation but not others. As we will see in the sections to follow, the Self-Supervised Learning (SSL) paradigm gives us the means to learn useful invariances.

Weak Supervision

In this learning paradigm, weak supervision in the form of a semantic graph **G** that specifies relations between observations is given to the agent. The agent aims to construct an embedding space that, according to a given metric (e.g. Euclidean distance), respects the relationships specified by **G** (Balestriero et al. 2022). The use of weak supervision of this kind is exemplified by work in one-shot facial recognition (e.g. (Schroff et al. 2015)). Any two images of the same persons face are related according to **G**. Models are trained to produce an embedding in which images of the same face are close, and images of different faces are far according to the given metric. Provided the dataset is rich enough, built into this are a number of invariances, such as to illumination, background, facial position and expression. One might also include other *out-of-distribution* observations such as images of completely different objects, leading to other non-trivial invariances.

Part of what makes self-supervised learning attractive is precisely this ability to encode invariances via specifying **G**. Many approaches have made extensive use of data augmentation as a means to do this (see e.g. (Chen et al. 2020b) and derivatives), although it is not always clear what invariances will result; the augmentation has to be done carefully. Using views of the same object, or face as in facial recognition, is a common technique in other domains. Cropping images to contain only part of an object, although much more crude than taking multiple photos from different view points, can be used to a similar effect. Other augmentations include changes to illumination, colour, visual style, or to rearrange/permute sequences (Xu et al. 2019), among others.

Various metrics and self-supervised objectives have been proposed. For the most part metrics are interchangeable, with some offering potential benefits in different problem domains. To name some, along with their most prominent or recent associated work: cross-correlation as in Barlow Twins (Zbontar et al. 2021); Euclidean distance as in BYOL (Niizumi et al. 2021); cosine-similarity as in SimSam (Chen et al. 2020d) and SimCLR (Chen et al. 2020b); and mutual information as in (Bachman et al. 2019). See (Le-Khac et al. 2020) for an extensive review of metrics and objectives. Broadly, approaches can be categorised into contrastive and non-contrastive. The essential difference being that contrastive methods make explicit use of negative



(a) Siamese Network (b) Similarity Measures (c) Naive Prediction

Figure 5.1: Illustration of the Siamese network architecture (a). Two or more examples are fed through the same network and subsequently compared. The comparison step (b) may include additional parametrized transformations, for example when attempting to predict one representation from another. (c) shows a naive model architecture that is attempting to predict one observation from another, in section 5.2.1 it is demonstrated why this is a bad approach.

examples, or *dissimilarity* among inputs.

Contrastive Learning

In contrastive learning the setup is typically as follows: for each observation x there are associated positive and negative observations x^+ and x^-, these associations are specified in $\mathbf{G}$. x^+ and x^- are used explicitly in a contrastive objective such as triplet loss (Schroff et al. 2015), angular loss (Wang et al. 2017) or otherwise, to learn a measure of similarity. We make extensive use of triplet loss in later sections, it is given below:

$$\mathcal{L}(x, x^+, x^-) = max(||f_\theta(x) - f_\theta(x^+)|| - ||f_\theta(x) - f_\theta(x^-)|| + \alpha, 0) \tag{5.1}$$

where f_θ is typically a neural network parametrized by θ, x is referred to as the anchor and α is the margin parameter. α prevents trivial solutions being learned. Triplet loss is derived from the desired property:

$$||f_\theta(x) - f_\theta(x^+)|| + \alpha < ||f_\theta(x) - f_\theta(x^-)|| \tag{5.2}$$

Put simply, triplet loss (and other contrastive objectives) attempt to bring positive pairs close in the embedding, while ensuring that negative examples are far. For triplet loss, negatives are at least as far as positive.

Contrastive learning has a long history. Its roots are in a field known as metric learning which predates even self-supervised learning. Early works in metric learning generally looked at learning linear models (e.g.

Mahalanobis distance (Weinberger et al. 2005)) with a focus on learning metrics (as opposed to more general measures). More recently, approaches are making use of deep learning (Kaya et al. 2019), and the boundary between what might be called metric learning and self-supervised learning - contrastive or non-contrastive, is blurred. We tend to use metric learning and contrastive learning interchangeably.

Non-Contrastive Learning

More recently, it was realized that negative examples are not required, at least not explicitly. It is enough to minimize the distance between positive pairs to produce useful representations in the self-supervised regime (Chen et al. 2020d). Despite the concern that embeddings might collapse (i.e. positive pairs are similar if they are the same), a flurry of new works including some of those we have already mentioned (e.g. BYOL) have demonstrated that no such collapse occurs as long as some tricks are used. Namely, that in the gradient-based optimization, to restrict gradients on one side of the comparison (known as the stop-gradient operation). The reasons for the non-collapse are beyond the scope of our work (refer to (Tian et al. 2021) for further details).

Siamese Networks

The above-mentioned works, and many other recent works in this area make use of Siamese networks. The term refers to the situation in which the same neural network is used to transform multiple observations (usually two) simultaneously for subsequent comparison as illustrated in Fig. 5.1 (a). The only distinction from the usual batching setup is the comparison step. More complex architectures use different network heads or introduce asymmetry e.g. stop-gradient or momentum encoders as in BYOL (see Fig. 5.1 (b)).

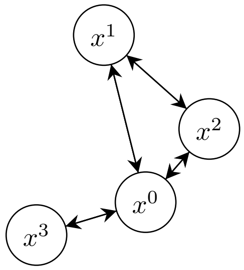
5.2 State-State Siamese Networks

State-state Siamese Networks (S3N) is a self-supervised[2] contrastive learning approach to learning representations for high-dimensional time-series data such as video (Wilkins et al. 2020), or more generally for learning representations of graphs where vertices have some associated data.

The approach was originally developed with the aim of identifying graphical anomalies in video games. The main idea is to learn a similarity that depends on the relative time at which an agent has a particular experience, and to use this similarity to identify novelty. In essence, observations that are closer in time

---

[2] and semi-supervised when applied to regression testing.



$$\begin{array}{c|cccc} & x^0 & x^1 & x^2 & x^3 \\ \hline x^0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ x^1 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ x^2 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ x^3 & 1 & 0 & 0 & 0 \end{array}$$

$$\begin{aligned} &A_{10}[max(D_\theta(x^1,x^0) - D_\theta(x^1,x^3) + \alpha, 0)] + \\ &A_{12}[max(D_\theta(x^1,x^2) - D_\theta(x^1,x^3) + \alpha, 0)] + \\ \\ &A_{20}[max(D_\theta(x^2,x^0) - D_\theta(x^2,x^3) + \alpha, 0)] + \\ &A_{21}[max(D_\theta(x^2,x^1) - D_\theta(x^2,x^3) + \alpha, 0)] + \\ \\ &\tfrac{1}{2}A_{30}[max(D_\theta(x^3,x^0) - D_\theta(x^3,x^1) + \alpha, 0) + \\ &\quad max(D_\theta(x^3,x^0) - D_\theta(x^3,x^2) + \alpha, 0)] \end{aligned}$$

Figure 5.2: Simple example of S3N when with a known MDP. The algorithm takes all pair-wise combinations of positive/negative examples. Values are weighted by transition probability and negative count. The graph shown is the 2D embedding created by S3N with a simple linear transform i.e. $D_W(x^i, x^j) = ||Wx^i - Wx^j||_2$. Relative distances between x^i have been preserved in the visualization. Each x^i is a one hot vector $\{0,1\}^4$ with $W \in \mathbb{R}^{2\times4}$. Notably, self-transitions have not been included, this is discussed further in section 5.2.1

in an agent's stream of experience are mapped closer in the learned embedding space than those that are experienced at different times. The fact that observations are experienced together in time is reflective of their relationship in the transition function (or semantic graph) i.e. $x_t \to x_{t+1}$. The similarity, or (inversely) distance between observations in the embedding can be used directly as a normality measure, with larger distances at inference time signifying novelty.

To outline the method more formally, the semantic graph **G** is given by the MDP that defines the environment. S3N selects positive and negative examples based on the neighbourhood of a state (or observation). With a triplet objective, positive pairs (x, x^+) and negative pairs (x, x^-) are given by $x^+ \in N(x)$ and $x^- \notin N(x)$. In simpler discrete settings where the MDP is known, one can train S3N by making direct use of the transition matrix, see Fig. 5.2.

In practice we do not have access to the transition function, otherwise the bug identification problem would be one of formal verification. S3N can still be used in this setting, although it gives no guarantees of correctness. Instead, we have access to the trajectories sampled by an agent. We can approximate the graph embedding procedure using these samples. Given a trajectory of observations $\tau = \{x_0, x_1, \cdots, x_T\}$ for a particular anchor example x_t, the next observation x_{t+1} forms a positive pair. Since we don't know which other observations might form positive pairs, at least without keeping track which may become prohibitively expensive, we assume all other observations are negative. This is illustrated below in a distance matrix.

$$\begin{array}{c|cccc}
 & x_1 & x_2 & \cdots & x_T \\
\hline
x_0 & D_\theta(x_0, x_1) & D_\theta(x_0, x_2) & \cdots & D_\theta(x_0, x_T) \\
x_1 & D_\theta(x_1, x_1) & D_\theta(x_1, x_2) & & D_\theta(x_1, x_T) \\
\vdots & \vdots & & \ddots & \vdots \\
x_{T-1} & D_\theta(x_{T-1}, x_1) & D_\theta(x_{T-1}, x_2) & \cdots & D_\theta(x_{T-1}, x_T)
\end{array} \qquad D_\theta(x_i, x_j) = ||f_\theta(x_i) - f_\theta(x_j)||$$

Each row of the distance matrix forms a single positive pair, with $T-1$ negative pairs for a particular anchor indexed by row (i.e. x_0 in the first row). Making use of triplet loss, the learning objective is then:

$$\operatorname*{argmin}_{\theta} \sum_{\tau} \mathcal{L}(\tau;\theta) \qquad \mathcal{L}(\tau;\theta) = \frac{1}{(T-1)^2} \sum_{i=0}^{T-1} \Big[\sum_{j=1}^{T} max(D_\theta(x_i, x_{i+1}) - D_\theta(x_i, x_j) + \alpha, 0) \Big]$$

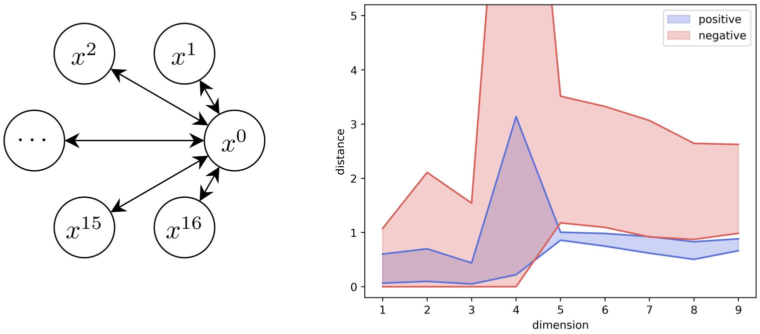
The assumption that all other observations in a trajectory are negative seems extreme, but assuming the MDP in question is relatively sparse, or that the agent doesn't repeatedly visit the same states it will not have a large impact on the resulting embedding. For small environments e.g. in Fig. 5.2 it may cause problems, but as we will see in later experiments, for larger ones it works reasonably well.

Rather than computing the distance over entire trajectories, in practice a mini-batching approach can be taken. The distance matrix is computed on a per-batch basis using the other elements in a batch as negative examples. In our later experiments, batches of observation are uniformly sampled from a dataset formed of many trajectories.

By following the procedure outlined above, the hope is that the support of the transitional distribution for each state is in some sense captured by the learned similarity measure. That is, smaller distances between observations indicate that a transition was observed in the training data, and the converse for larger distances. In actuality, the distances are influence by the topology of the MDP, the embedding dimensionality, the axioms of the metric used and the flexibility/expressibility of the model. These are explored in the sections to follow.

5.2.1 Investigating Embeddings

In this section we attempt to build some intuition for why graph embedding approaches such as S3N might be useful for identifying abnormality. In doing so, we explore some of the properties that S3N exhibits when the



(a) (b)

Figure 5.3: Embedding of 16-star graph G (a) in different dimensions. The distance condition $D_\theta(x^0, x^i) + \alpha < D_\theta(x^i, x^j)$ where $i, j > 0$ should be respected if we are to use distance as a measure of abnormality. Recall that abnormality is an observed relation (edge) in the agent's experience that is not in $\mathbf{G}$. In (b) we see that the embedding dimension required for this condition to be satisfied is 5, i.e. where the minimum negative distance $\geq$ maximum positive distance. See text for further details, and Appendix A.2.5 for more examples.

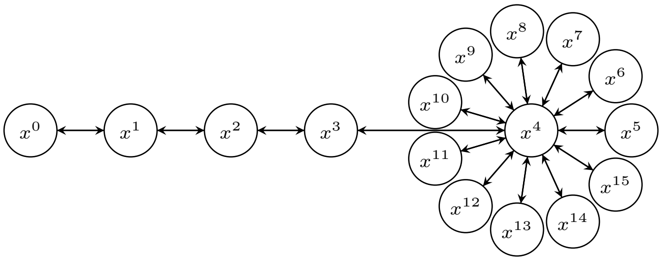
MDP is known. We also expand upon what is presented above by introducing the notion of pseudo-metrics (metrics that do not conform to the usual axioms), to deal with self-transitions and transition directionality.

Embedding Dimension

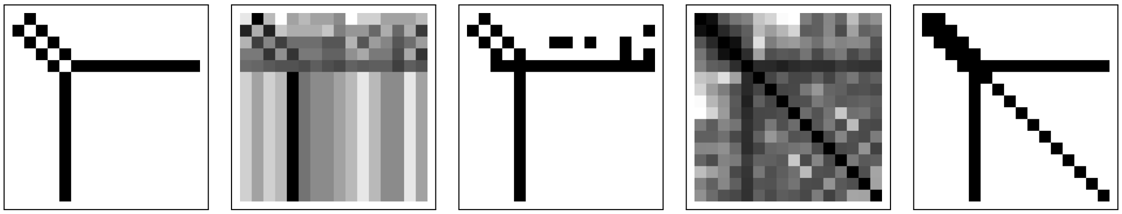
The triplet objective aims to satisfy the condition $D_\theta(x, x^+) + \alpha < D_\theta(x, x^-)$. Of course, different graphs require different embedding dimensions for this condition to be satisfied. If we are to use S3N to identify novelty or abnormality, it is especially important that this condition is met. When an agent encounters a new pair (x_t, x_{t+1}), if it is negative (there is no edge in $\mathbf{G}$) then the distance should reflect that and be relatively large. As a sanity check, in Fig. 5.3 we check this by embedding an N-star graph in different dimensions. Further examples can be found in Appendix A.2.5.

High-Entropy Transitions

In chapter 3 we mentioned briefly that predictive models (those that try to estimate x_{t+1} from x_t and a_t) could be used to identify novelty by using prediction error as a normality score. Models that produce a point prediction (e.g. the expected next observation) suffer from a problem. Prediction errors will be high in cases where a transition distribution is not well captured by the point estimate and this may lead to false positives. In Fig. 5.4 we demonstrate this problem with a simple example that compares S3N with a predictive approach. The example highlights why methods such as S3N, or more generally, methods that make proper use of embeddings at inference time should be generally preferred. A simple 3-layer



Confusion matrix				
	Forward		S3N	
	P	N	P	N
T	30	210	30	217
F	0	9	0	16



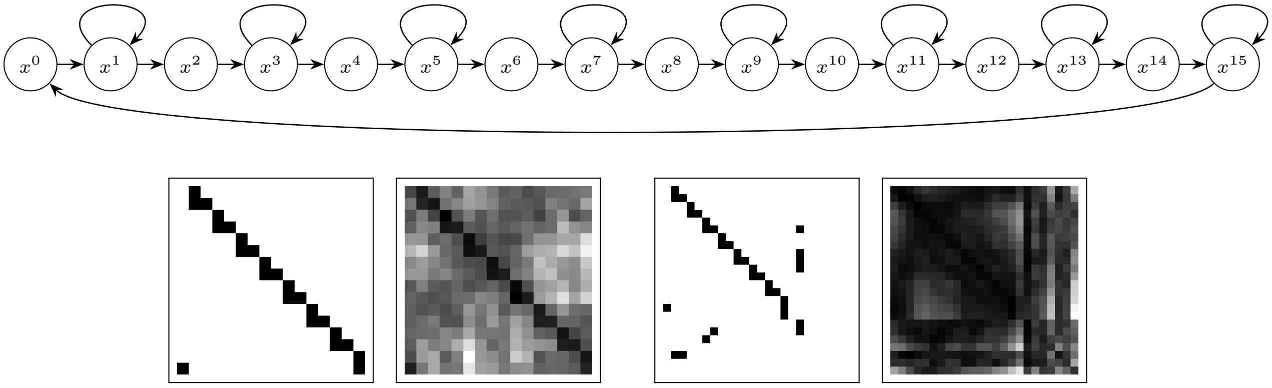
(a) (b) (c) (d) (e)

Figure 5.4: Example of a high-entropy transition and the problems that they can cause. (a) shows the graph adjacency matrix. (b) and (c) show the score and classification for the forward model respectively for every possible pair of states. (d) and (e) similarly show the score and classification for S3N respectively. The high-entropy transition at x^4 can be seen clearly in (b) as a band of relatively high scores. This band leads to a number of subsequent misclassifications (c). The threshold was chosen to balance sensitivity and specificity which handles the imbalance of labels. This choice leads to false negatives (classifying as normal when actually abnormal). With a slight variation in threshold the normal x^4 transition would be misclassified as abnormal. The problem is avoided entirely by S3N, where the only misclassifications are of self-transitions. In Fig. 5.5 we remedy the self-transition problem by making use of pseudo-metrics. Embedding dimension 6, and margin $\alpha = 0.2$. As in experiments to follow, the embedding dimension was chosen to give S3N sufficient space for the embedding (see Appendix. A.2.5 for a more in-depth discussion of embedding dimension), and the margin α was chosen based on experience gained from prior work, as long as α is in a reasonable range (e.g. $[10e^{-2} - 10e^{2}]$) it has little impact the result for such simple graphs.

MultiLayer Perceptron (MLP) is trained to minimize Mean Squared Error (MSE) between an observation and its neighbours. The predictive model is estimating the expected next observation. S3N uses the same MLP architecture and hyperparameter choices. The normality threshold in both cases is chosen according to g-mean, as will also be the case for subsequent sections.

Self-Transitions

Metrics used for contrastive learning by definition satisfy the *identity of indiscernibles* axiom $d(x, x) = 0$. This poses a problem for abnormal self-transitions, a classic example of this in video games is the so-called freeze bug. A metric that has this property will always give a score of zero for such bugs, leading to a false negative. This can be seen in Fig. 5.4 and in our later experiments with Atari games (see section 5.3.3). To remedy, one can make use of a pseudo-metric. For example $||Wf_\theta(x^i) - f_\theta(x^j)||_2$ where W are parameters



(a) (b) (c) (d)

Figure 5.5: Example of S3N with an asymmetric pseudo-metric. (a) shows the graph adjacency matrix of a directed ring graph with self-transitions at every other state. It also shows the S3N's classification (accuracy of 1.0). (b) shows the score (distance) associated with each transition. Note the difference between this score and that of Fig. 5.4.e, self-transitions are no longer incorrectly classified. Normal states are one-hot vectors, in (d) and (e) eight new abnormal states that have additional ones in some positions are introduced. They were not seen during training. Increasing the embedding dimension from 8 to 16 leads to a perfect classification, otherwise there are 11 false negatives. Embedding dimension 8, $\alpha = 0.05$.

to be optimized. Or otherwise introduce asymmetry into the metric, for example by introducing dummy variables $||f_\theta(x_L^i) - f_\theta(x_R^j)||$ where $x_L = [0, x], x_R = [1, x]$. In Fig. 5.5 the former approach is demonstrated with a metric that projects the LHS input using a 2 layer MLP.

Asymmetric Transitions

In the previous section we saw that by introducing a pseudo-metric we could avoid misclassifying self-transitions. This also addresses a potential problem that might arise from the symmetry axiom $d(x, y) = d(y, x)$ as shown in Fig. 5.5. Without a pseudo-metric S3N would not identify a situation in which a player was able to incorrectly go *in-reverse*, they may, for example, be permitted to transition from one level to the next, but not back again. The possible benefits of asymmetric metrics are not explored further in this chapter as the bugs (other than freezing) we are interested in do not require this and using symmetric metrics is generally simpler. A more in-depth exploration of this is left as future work.

5.3 Experiments

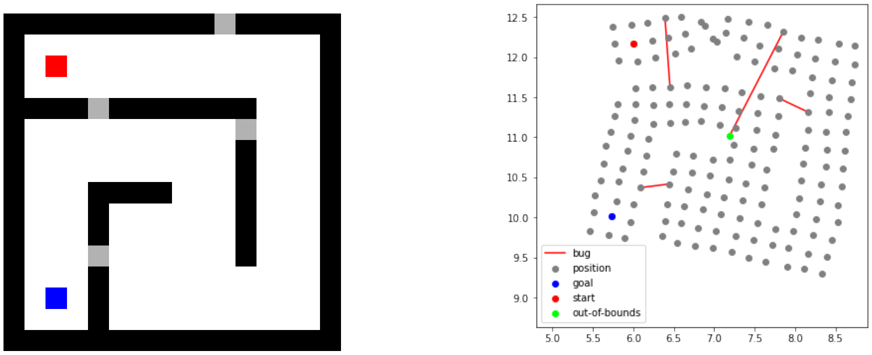
Now that we have provided intuition for why S3N works (at least in principle), we move on to more complex settings. Namely, those for which the MDP is not known and we must therefore use an approximate procedure to learn the embedding. This procedure was outlined in section 5.2. It should become apparent in subsequent sections that it is a good approximation provided we have enough trajectories to work with. In the first set of experiments we work with relatively simple grid environments in which a player is able to move around in the cardinal directions on a 2D plane. We then move to more complex environments such as Atari 2600 games, those created in World of Bugs and real world video surveillance footage.

5.3.1 Identifying Unintended Shortcuts

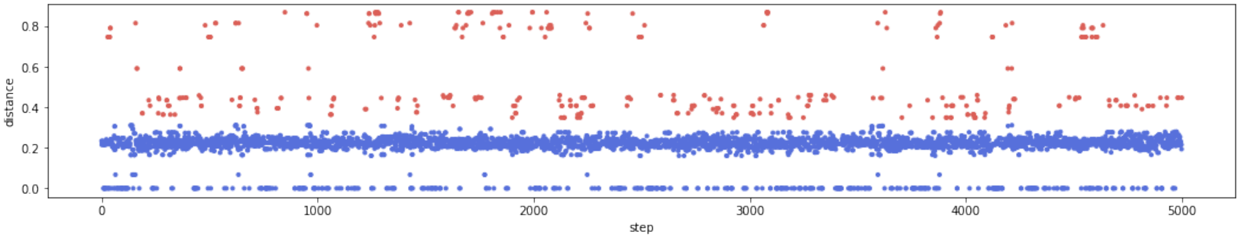
Here we investigate the so-called *unintended shortcut* bug, a kind of bug that allows players to take an unintended shortcut to the goal. We experiment in the Explorer-v0 environment (see Appendix. B.1.2) in which the player can move in the four cardinal directions to navigate through a maze to the goal. Shortcuts were introduced by removing collision detection for certain walls of the maze, allowing the player to pass through them. Results are shown in Fig. 5.6. S3N was trained on 50k normal observations (enough to cover all possible transitions) collected by a uniformly random policy. To avoid bias towards observations at the beginning of a trajectory (i.e. surrounding the players initial position) and to balance the training dataset, the goal was removed and all observations were collected in the same trajectory. A policy that explores in a principled fashion could have been used to the same effect. The network architecture used is a simple 4 layer MLP, further details can be found in Appendix. A.2.3. In this proof-of-concept example, S3N achieves an accuracy of 1.0.

Pixel Space Discontinuity

In the experiment above, S3N learnt an embedding which reflected the players position in a coherent manner. We might expect that this is because the player smoothly transitions from one state to another. In many video games there are large discontinuities between observations, for example, when bringing up a menu, between levels, due to an explosion, or otherwise. Since distance is being used directly as a measure of normality, we had better be sure that S3N can deal with such discontinuities. The Explorer-v0 environment used in the previous section is modified to contain large discontinuities in pixel space at certain player locations. Fig. 5.7 visualizes the embedding. Although it looks distorted when compared to Fig. 5.6, it can still be used to the same effect, especially if the embedding dimension is increased. The failure to converge to an embedding that clearly reflects the players position is likely due to the discontinuities. The neural network, which was again a simple MLP, needed to learn a much more complex transformation. There is no significant correlation between the embedding distances and the pixel-wise distances as measured by a correlation coefficient of -0.068 on a trajectory of 5k observations. This is an indicator that S3N is indeed

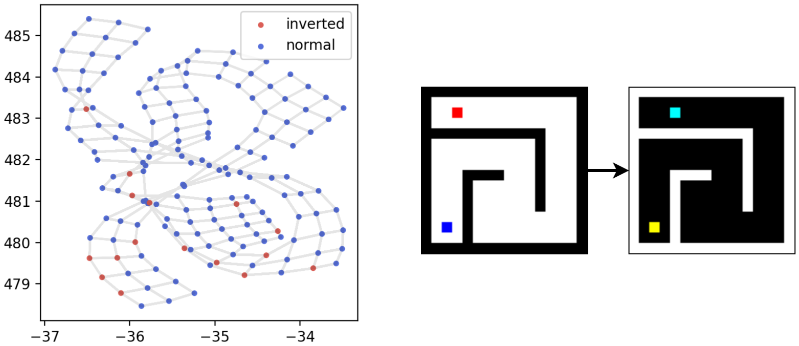


(a) Maze (b) Embedding



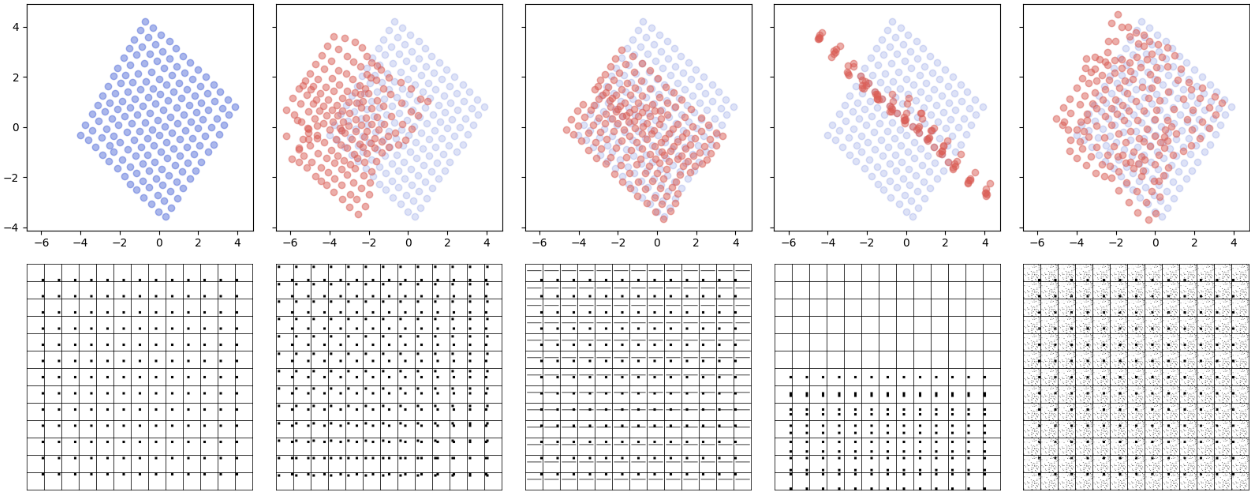
(c) Score

Figure 5.6: (a) shows the environment a 16x16 maze, red indicates the player at their starting position and blue the goal. The player may move in the cardinal directions one cell at a time, or immediately to the other side of the wall if colliding with a wall that has its hit box removed (those in grey). (b) shows the embedding space that has been learned by S3N, which each axis representing one of the two dimensions of the space. The red lines show the abnormal transitions in the embedding space. The green point shows the embedding for *out-of-bounds* observation, i.e. the observation where no player is present as they made it out of the maze boundary. (c) shows the scores (distances) that S3N assigns to each transition in a trajectory containing 5k observations. In this trajectory, the player may pass through any of the inner walls, effectively corresponding to the situation where the collision system has completely failed. The score associated with each abnormal transition is always higher than those associated with normal transitions. S3N achieves an accuracy of 1.0 on this proof-of-concept task if the decision boundary is trivially set to the maximum normal score.



(a) (b)

Figure 5.7: The 2D embedding space (a) learned in the modified Explorer-v0 environment with large pixel discontinuities (b). See text for details.



(a) Normal (b) Block (c) Line (d) Split (e) SP Noise

Figure 5.8: S3N is trained to produce a 2-dimensional embedding of the Alone-v0 environment. In this environment, the agent (black square) is free to move around a room in the cardinal directions, or remain in place. The top row shows the learned embedding over all possible observations. Blue shows normal data and red shows data that has been corrupted in some way. The bottom row shows the original observations as a 28 × 28 pixel image, each observation has a corresponding point in the embedding above. There are 13 × 13 possible player positions. (a) shows only normal data, (b) a block is added top left, (c) a thin line is added, (d) observations are reflected top to bottom, (e) salt and pepper noise is added. Corruptions are constant over all observations in each case, see text for details.

learning something more interesting than just pixel-wise distance.

5.3.2 Identifying Systemic Bugs

A systemic bug, as defined in chapter 3, is caused by a fault that impacts a large part of the state space or transition function. The resulting bugs tend to share some semantic features, for example, in a total failure of the collision system, the player can pass through solid objects. In an unsupervised setting, these kinds of bugs can be difficult to identify (at least without strong inductive bias) since there may be no examples of the intended behaviour in the data (e.g. of the player colliding with a solid wall).

To illustrate how systemic bugs might be identified in a semi-supervised regime for the purposes of regression testing, S3N is trained to produce a 2-dimensional embedding of normal experience in the Alone-v0 environment. In this environment the player is situated in a small room and can move in any of the four cardinal directions, or choose to remain stationary. Observations are a 28 × 28 image from a birds-eye perspective, the player is shown as a black square. The players position fully determines the state of

Bug	$\mu_t(\Delta(\tau))$	$\mu_i(EMD_i)$	95% conf.
Normal	0.371	38.826	± 4.210
Block	0.321	66.587	± 2.924
Line	0.329	84.484	± 2.767
Split	0.344	366.782	± 4.530
Salt & Pepper	0.401	111.907	± 1.791

Figure 5.9: Global distance statistics for trajectories of length 500 in the Alone-v0 environment (see Fig. 5.8). $\mu_t(d_\tau)$ is the mean distance between the current and next state i.e. $1/|\tau| \sum D_\theta(x_t, x_{t+1})$. $EMD_i := EMD(\Delta(\tau), \Delta(\tau_i^*))$ is the earth mover distance, a measure of the difference between distributions, see A.2.4. The distribution of distances for each trajectory was independently compared to that of 50 normal reference trajectories $\Delta(\tau_i^*)$ to see if the EMD might be used to determine whether a bug is present. The EMD is smaller when comparing normal trajectories and so in this instance the bugs can be successfully identified.

the environment. Training data is collected using a uniformly random policy. The learned embedding is visualized in Fig. 5.8.

The faults we introduce distort the data manifold. The hope is that this is reflected by S3N when projecting abnormal examples into the learned embedding space. Looking at the visualizations in Fig. 5.8 this is indeed the case. The distances between observations both increase and decrease in places. While the use of the individual distances between observations may still identify the bug in some transitions, they will be missed for smaller distances. It also may not be clear that the bug is systemic by just looking at individual distances. This is not such a big issue, as long as we identify one or more instances a developer will be notified of the problem and a fix would hopefully extend to all instances. We might identify the issue as systemic, thus providing the developer with more information, by analysing distance statistics of a trajectory (or collection of trajectories), see Fig. 5.9.

5.3.3 Atari 2600

This collection of experiments on Atari 2600 games is presented in our paper (Wilkins et al. 2020). They are the first examples of S3N applied to more interesting video games. The aim here is to demonstrate the effectiveness of S3N in identifying some relatively simple bugs in a more challenging setting.

Experimental Setup

Experiments are performed on seven Atari games[3] that have previously been made available as part of the Arcade Learning Environment (ALE) (Bellemare et al. 2013) and OpenAI Gym (Brockman et al. 2016). Each game was chosen with a specific motivation in mind - to test S3N's ability to deal with large pixel

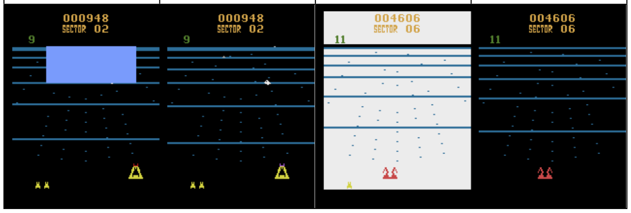
---

[3] Beam Rider, Breakout, Enduro, Pong, Qbert, Seaquest, and Space Invaders

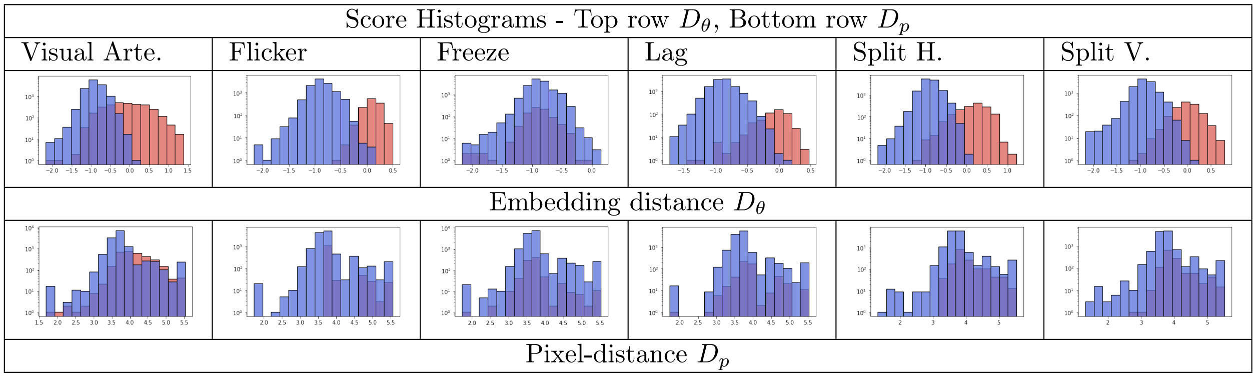
Precision-Recall AUC			
Bug	D_p	D_θ	NS
Visual Artefact	0.461	**0.878**	0.192
Flicker	0.151	**0.997**	0.096
Freeze	**0.051**	0.049	0.050
Lag	0.110	**0.916**	0.044
Split Horizontal	0.231	**0.963**	0.098
Split Vertical	0.204	**0.970**	0.098

(a)

Max D_θ (normalized)		Max D_p (normalized)	
D_θ	D_p	D_θ	D_p
1.	0.294	0.208	1.

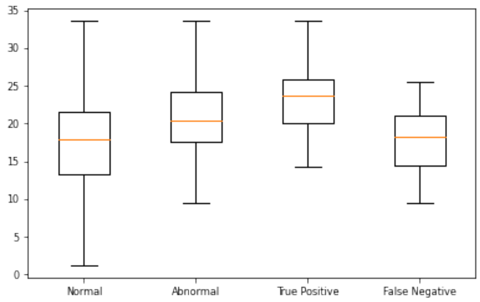


(b)



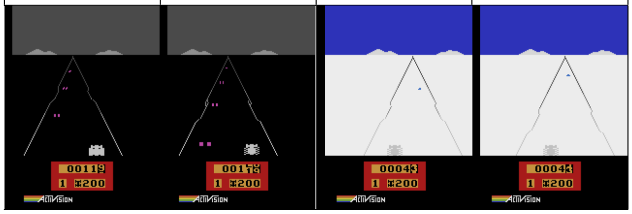
(c)

Figure 5.10: Comparison of the learned metric D_θ with pixel-wise distance D_p on the Beam Rider environment. S3N is able to deal with the large pixel discontinuities (flashing) that are exhibited frequently in Beam Rider as highlighted in (b). As such, it performs well in identifying the various visual artefacts (a) and (c). (c) visualizes the scores (log scale x-axis), blue = normal, red = abnormal. The score counts (y-axis) are also on a log scale. Ideally we should see blue on the left and red on the right. NS = No Skill. Detailed numerical results for this environment can be found in Fig. D.1 in the Appendix.



(a)

Example TP		Example FN	
$logD_\theta$	D_p	$logD_\theta$	D_p
0.247	18.41	-0.822	18.33



(b)

Figure 5.11: (a) shows a pixel-distance (D_p) box plots in the Enduro environment for the lag bug. S3N has the worst performance for identifying lag in this environment. The plots show pixel distance for normal and abnormal transitions (ignoring self-transitions), and S3Ns true positive (abnormal when actually normal) and false negative (normal when actually abnormal).

discontinuities including flashing and scene changes (e.g. Enduro/Beam Rider), embed cyclic/acyclic graphs (e.g. Pong/Seaquest), or to deal with a high combinatorial dimensionality (e.g. Breakout and its arrangement of bricks).

As usual, S3N was trained in a semi-supervised fashion, and later tested on unseen data that contains bugs. Observations for training and testing were collected using the Stable Baselines (Hill et al. 2018) implementation of Advantage Actor-Critic (A2C) (Mnih et al. 2016). Observations total approximately 200k per game. Bugs have been artificially introduced into approximately half of the collected trajectories for each game at a rate of 0.01. These include freezing, flickering and various visual artefacts (see Fig. 5.12). The first 100k (bug-free)[4] examples are used for training, and the last 100k (bugged) for testing. As we noted in chapter 4, introducing bugs in this way has some limitations, they are not as realistic as they could be, making them somewhat easier to identify. This restricts us to the regression testing setting. Still, these bugs represent a proof-of-concept and identification is still non-trivial. The flickering and freezing effects are realistic. The dataset and code is publicly available as part of the supplementary material of the original work here[5].

A vanilla version of S3N with a proper metric (Euclidean distance) that is subject to the issues outlined in 5.2.1 is used. Self-transitions are omitted from the results to avoid biasing the performance metrics. Doing so doesn't narrow the applicability of our approach, or bias the performance (positively or negatively) on identifying issues outside those related to self-transitions. By using a proper metric S3N will not be able to identify freezing effects (as is reflected in the results), or any of the intermediate frozen observations in the lag bug. In practice, when identifying lag it is enough to identify the "skipping" effect that occurs in the last transition. This is taken into account when evaluating performance on this bug. The full numerical results for identifying bugs in each environment are presented in Appendix. A.2.1. The most interesting of these results is discussed in the next section.

Discussion

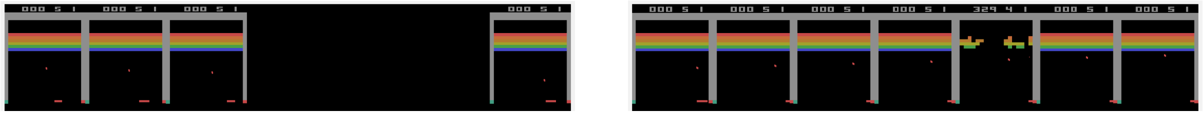
S3N performs well in identifying all bugs with the exception of freezing. Perhaps the most interesting result is in the identification of lag. The "skip effect" is generally small (on the order of the difference between normal frames) but it can still be identified, see Fig. 5.11. The two split bugs also offer some subtle issues which S3N is for the most part able to identify (see 5.13 for some examples).

Unsurprisingly, given its strong inductive bias towards modelling dynamics, S3N is able to identify sudden

---

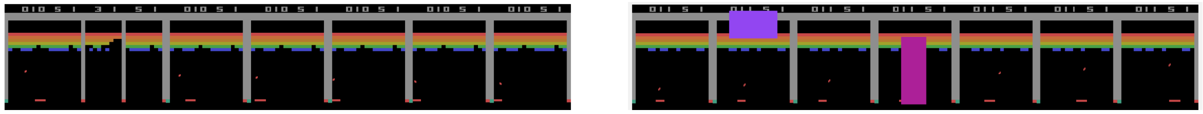
[4] Recall that the training data must be bug free because S3N is learning distances between observations that are normal.

[5] https://www.kaggle.com/benedictwilkinsai/atari-anomaly-dataset-aad



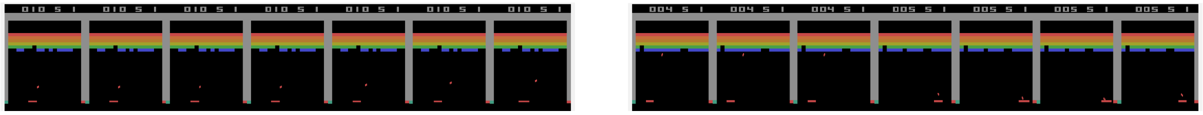
(a) Flicker - observations are blacked out.

(b) Split horizontal - observations have their top/bottom half swapped with another in the trajectory.



(c) Split vertical - observations have their left/right half swapped with another in the trajectory.

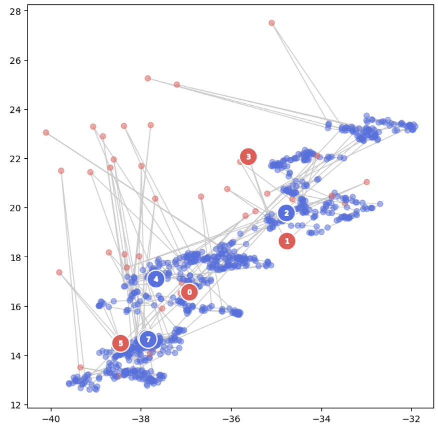
(d) Visual artefact - random blocks of colour of varying size are introduced.



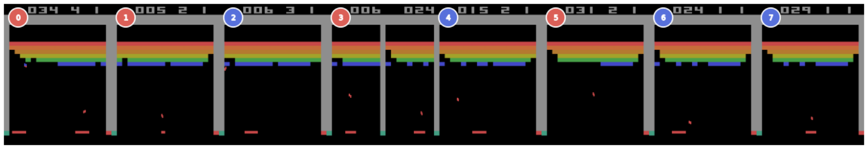
(e) Freeze - observations are repeated for N steps.

(f) Lag - observations are repeated for N steps, overwriting any subsequent observations.

Figure 5.12: Examples of the bugs that have been introduced into the Atari game Breakout.



(a) S3N 2-dimensional Embedding



(b) Observations

Figure 5.13: A visualisation of a 2D S3N embedding space of a Breakout trajectory that contains the *split vertical* bug (a). The red points in the plot are bugged observations (images) projected by the S3N network into 2 dimensional space, similarly the blue points are projected normal observations. Numbered examples of observations can be seen in (b). The plot serves to highlights the relatively large distances between consecutive observations that are normal and abnormal.

and significant (in terms of pixel-wise distance) changes to observations. This is highlighted by its performance in identifying visual artefacts, and most strongly in identifying flickering which exhibits the most significant change between observations. As in section 5.3.1, we might again be suspicious that S3N has just learned the pixel-wise distance $D_p(x_t, x_{t+1}) = ||x_t - x_{t+1}||_2$ rather than some more interesting metric that is more representative of the agent's experience. In certain environments, namely, those in which normal transitions are smooth (small in D_p), D_p may be used directly as a score to obtain performance similar to S3N. This follows from the fact that the visual bugs we consider here tend to manifest suddenly with a relatively large D_p. In later experiments this will not be so. For the moment, we show that S3N learns a more meaningful metric in the Beam Rider environment. In this environment transitions tend to be much more varied as each time the player is hit by an enemy the screen flashes. The metric learned by S3N takes this into account and still achieves good performance. This result is shown in Fig. 5.10.

This application of S3N to identifying a range of bugs, without being specifically tailored to any one of them is encouraging. In the next sections we will explore bug identification in even more complex settings, with bugs that are much more realistic.

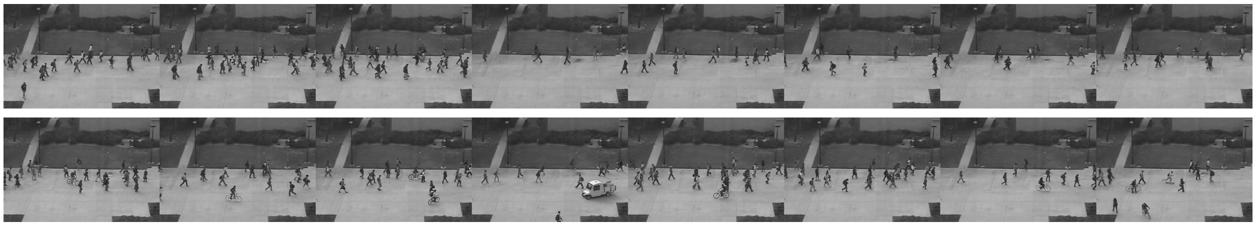
5.3.4 Video Surveillance

As discussed in chapter 4, there is a distinct lack of datasets that are suitable for bug identification. Unfortunately this means that a fair and in-depth comparison of our approach with approaches used in related domains (e.g. in video surveillance) is difficult. Generally, it would be unfair to draw any conclusions about approaches that are tailored to a particular domain based on their performance in another domain. Despite this, here S3N is compared with approaches developed for anomaly and novelty detection in the domain of video surveillance.

S3N is not tailored to any particular domain or task, but like any approach it contains biases that may make it more suited to some domains than others. It may however be at some disadvantage against those approaches that are tailored to a specific task. With this in mind, the datasets that are closest methodologically to ours (i.e. permit semi-supervised training), and that play to the strengths of S3N have been selected for comparison. To reach comparative performance we introduce some tricks, these do not fundamentally alter the workings of the algorithm but are used to deal with issues such as label noise or small datasets.

Video surveillance and monitoring is one domain that has seen numerous high-quality datasets and an influx of work over the last years (Kumari et al. 2022; Patil et al. 2017). The experiments to follow are

(a) Pedestrian 1



(b) Pedestrian 2

Figure 5.14: The UCSD Pedestrian datasets are composed of a collection of videos of pedestrians walking in a park. Anomalies are temporally extended and include skateboarders, bikes, wheelchair users, people walking on grass, food carts and various other kinds of vehicles. A selection of normal (top row) and anomalous (bottom row) observations are shown in (a) and (b). Both datasets are relatively small, pedestrian 1 contains 6800 training and 7200 test observations, and pedestrian 2 contains 2550 training and 2010 test observations.

performed on the well known UCSD Pedestrian dataset (Mahadevan et al. 2010; Li et al. 2014), see Fig. 5.14 for dataset details.

Experimental Setup

As the datasets are relatively small, rather than train S3N on a frame-by-frame basis, frames are split into patches each of size 32 × 32 pixels with an eight pixel overlap. This artificially increases the size of the training dataset by two orders of magnitude and ultimately allows S3N to identify novelty. Patches are treated as if they were full observations, positive pairs are formed by patches at the same location in consecutive frames. Negative pairs are formed by patches at non-consecutive times as usual, as well as any patches not at the same image location. The score for a particular frame is computed as the sum of the individual patch scores (i.e. Δ_1 per patch). Additionally, we take the mean of frame scores over a window either side of a particular frame. This smooths the scores, leveraging additional temporal information to result in better performance when identifying longer temporally extended anomalies like those seen in this task. Cosine distance $D_\theta(x_i, x_j) = 1 - \left[(f_\theta(x_i) \cdot f_\theta(x_j))/||f_\theta(x_i)|| \cdot ||f_\theta(x_j)||\right]$ is used as the embedding metric.

Results for the UCSD Pedestrian datasets					
		Ped. 2		Ped. 1	
Ref.	Year	EER	AUC	EER	AUC
Xu et al. 2015	2015	0.17	0.908	0.16	0.921
Sabokrou et al. 2015	2015	0.19	-	-	-
Sabokrou et al. 2016	2016	0.15	-	-	-
Hasan et al. 2016	2016	0.217	0.900	0.279	0.810
Ravanbakhsh et al. 2017	2017	0.14	0.935	0.08	0.974
Sabokrou et al. 2017	2017	**0.082**	-	0.091	-
Sabokrou et al. 2018	2018	0.13	-	-	-
Ravanbakhsh et al. 2018	2018	0.11	0.955	**0.07**	0.968
Liu et al. 2018	2018	-	0.954	-	0.831
Zhou et al. 2019	2019	0.103	0.949	-	-
Nguyen et al. 2019	2019	-	0.962	-	-
Chang et al. 2022	2022	-	**0.967**	-	-
Ours (S3N)	**2020**	**0.135**	**0.916**	**0.257**	**0.804**
Pixel-wise distance	2020	0.122	0.933	0.172	0.914

Figure 5.15: Comparison of performance on the UCSD Pedestrian 2 dataset. Both EER (Equal Error Rate) and ROC-AUC are presented where reported in their respective works. S3N performs worse than some of the more recent approaches that have been tailored to the task. Given that S3N is a non-tailored novelty detector it performs reasonably well. In fact, it compares well to another similarly non-tailored approach to novelty/abnormality detection that is (Sabokrou et al. 2018). We also report a result using pixel-wise distance, which is surprisingly high. See text for further discussion.

Discussion

In the results presented in Fig. 5.15 we compare performance on the Pedestrian 1 and 2 dataset taking reported results from various other works. S3N performs relatively well considering many of the works are specifically designed for the task. We also present a simple pixel-wise distance result, frames are similarly split into patches, the patch with the maximum score is selected as the frame score. It performs surprisingly well, but like S3N does not capture some of the more subtle abnormality that manifests (e.g. bikes behind other pedestrians). While S3N is able to identify some issues present in the video, enough that it could be of some practical use, it is limited by the need to introduce tricks (specifically averaging over frames) highlights a limitation of the approach. S3N as it has been presented is very good at identifying sudden changes, or non-systemic bugs such as those seen thus far. It does so in a way that is invariant to large changes in the magnitude of its observations as highlighted in the experiment in the Beam Rider environment. Using the distance between subsequent *test* observations tends to work well if the transitions are from normal to abnormal, or vice versa, but may fail if they are abnormal to abnormal. The model may in some sense end up generalising *too well* if the abnormal transitions follow a similar pattern of transition to that of normal transitions. To give a contrived example, one might imagine a pedestrian wearing brightly coloured clothes, since clothes don't tend to impact the dynamics of walking, S3N may be blind to this novel observation. This kind of content blindness is something that manifests rather severely in the next set of experiments. For a more intuitive demonstration of the problem see Appendix. A.2.5.

5.3.5 World of Bugs

This set of experiments makes use of the platform presented in chapter 4. The bugs that are explored here are the most challenging and realistic so far. The testing agent observes the environment from a first-person perspective, the bugs manifest over many observations and tend not appear suddenly (i.e. over single transitions). As discussed in the last section, S3N can suffer from a kind of content blindness. This is especially true if the environment transitions are not very diverse. This limitation of S3N is apparent in the experiments to follow, but nevertheless it is able to identify *some* issues.

In addition to S3N, experiments are performed with two other contrastive learning approaches. Both make use of data augmentation to generate negatives for use in the contrastive objective. The first approach is inspired by (Masana et al. 2019), where normal observations are augmented and treated as negative. Positive observations are those that appear in the training data. The normality score is defined as $D(\mu(f_\theta(X)), f_\theta(x_i))$ where $\mu(f_\theta(X))$ is the mean of the embedded training set.

The second approach is a kind of hybrid of S3N and the above, it learns an embedding of observation *pairs*. Positives are pairs (x_t, x_{t+1}) that appear in the training set, negatives are pairs that have been augmented in some way. The S3N inspired data augmentation swaps one observation in the pair for another in the training set. Other augmentations that are used in both approaches include modifying the image by altering brightness, contrast, rotation and blurring. The score is defined as above as the difference between newly embedded observation pairs and the mean of the embedded training set. Triplet loss is used for both approaches.

Experimental Setup

The environment used is the Maze-v1, which is a simpler version of the Maze-v0 environment presented in chapter 4 (see section 4.1.2). The following 10 bugs are implemented in this environment: geometry corruption; missing object; screen tearing; texture corruption; Z-fighting; unintended object; terrain hole; unintended shortcut; player out of bounds; high force. The bugs visualized in Fig. 5.17 with further details in Appendix. B.3.2 and in the bug glossary. Models are trained on 60k normal observations gathered using the built-in navigation policy described in chapter 4. The same AlexNet architecture and hyperparameter

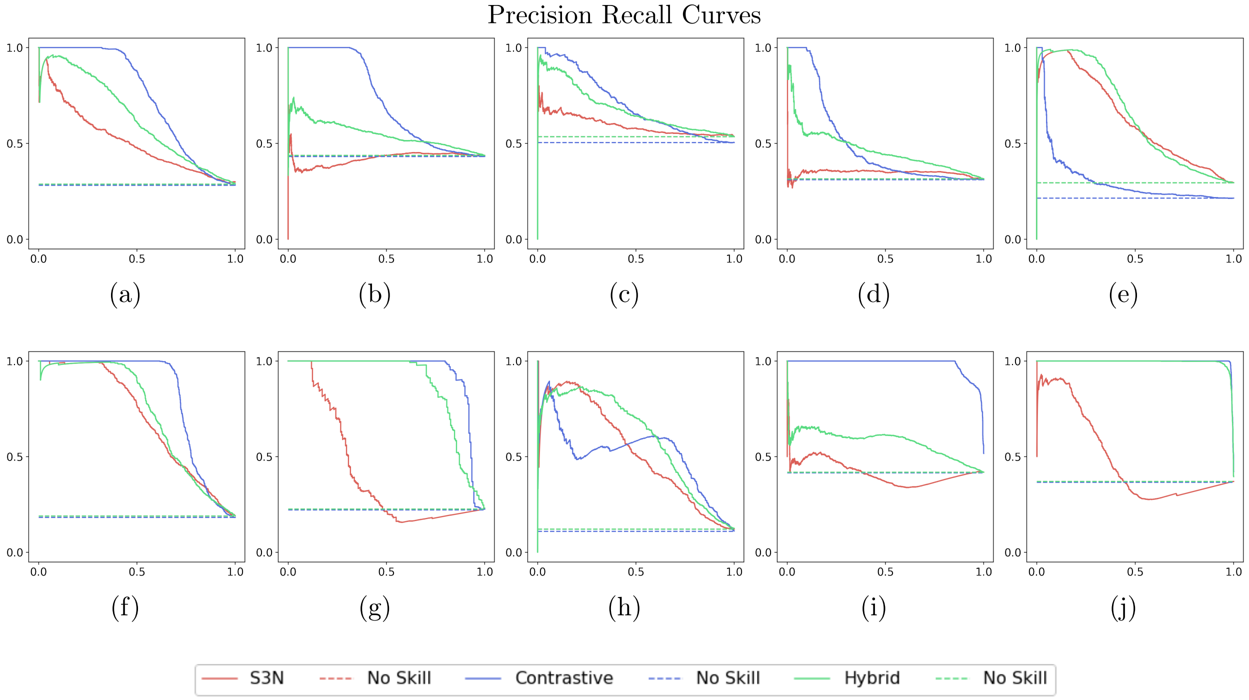


Figure 5.16: Precision recall curves for the Maze-v1 environment. Axis show recall on x and precision on y. Each plot shows the result for a particular bug: (a) geometry corruption; (b) missing object; (c) screen tearing; (d) texture corruption; (e) Z-fighting; (f) unintended object; (g) terrain hole; (h) unintended shortcut; (i) player out of bounds; (j) high force; See text for discussion and Fig. 5.17 for an example of each bug. No skill shows the performance of a random classifier. It is the same for S3N and hybrid as they are both transition models and work with pairs of observations unlike contrastive which only works with individual observations.

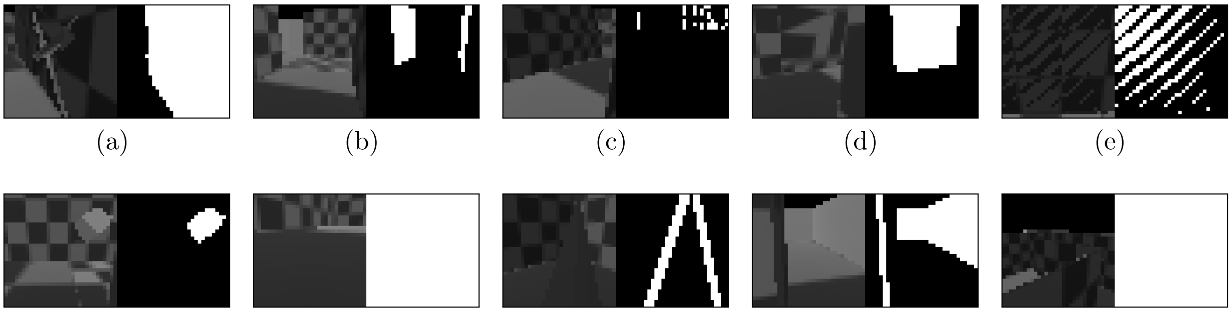
choices are used in each of the three approaches. Euclidean distance is used as the embedding metric. A summary of results in the form of Precision-Recall curves is presented in Fig. 5.16. See Appendix. A.2.1 for numerical results and further details.

Discussion

The results are mixed, but encouraging overall. S3N lags behind the other two contrastive approaches in identifying many of the bugs. It exhibits content blindness for those problems that are systemic, or do not exhibit sudden changes. Still, it performs reasonably well in identifying (e) Z-fighting, (f) unintended object and (h) unintended shortcut[6]. These bugs, or at least the way they manifest here, align with the inductive biases S3N possesses. In (e) the textures flicker rapidly, in (f) the object sweeps through the environment at a relatively high speed, and in (h) there is a sudden change from being very close to a wall to a view from

---

[6] there are some labelling issues with this bug which explain to some degree the reduced performance on this issue across the board. This is discussed in Appendix. A.2.3



(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

Figure 5.17: Examples of each bug implemented in the Maze-v1 environment. (a) geometry corruption; (b) missing object; (c) screen tearing; (d) texture corruption; (e) Z-fighting; (f) unintended object; (g) terrain hole; (h) unintended shortcut; (i) player out of bounds; (j) high force;

the other side. While not a perfect identifier, S3N would be enough to alert a developer and provide some value in testing if the threshold was set high to avoid false positives. The trade offs discussed in chapter 3 become apparent here. Some certainty in the test result is sacrificed for the sake of other desirable traits - decoupling and reusability.

The other two contrastive learning approaches reach much better performance on the majority of the bugs tested. This is even to the point where they could be used in a production system for the following: (j) high force, (i) player out of bounds, (g) terrain hole, (f)* unintended object. (j), (i) and (g) belong to the same class of *out of bounds* bugs, but are visually distinct. Typically, (g) and (j) can be identified with a check on the players position or velocity. (i) is slightly more involved for complex level layouts. The benefit of using a learning based approach here is that they can capture context from experience. To give an example, say there is a multi-layered level layout, and the player falls through a floor down to the next. Writing a guard on the players vertical position is now more involved than just a blanket rule like `player.y > LOWERBOUND`. For a learning based approach this doesn't change things, the same test can be used. Although performance is high for (f), this bug has potential for great diversity - there is a virtually unlimited number of objects that could be present. In addition, the object here is previously *unseen* by the agent, objects that have been seen but are wrongly placed are likely more difficult to identify. While the result is encouraging, further experimentation is required before any claims can be made about identifying this kind of bug in general. The results for the bugs (a) geometry corruption and (b) missing object are encouraging, but are similar to (f) in that making claims is problematic.

The bugs (c) screen tearing and (d) texture corruption have relatively low performance. The performance on (c) might be explained by incompatible inductive bias - the bug shifts a strip of the observation some number of pixels horizontally but otherwise leaves it unaffected. While glaringly obvious to us because of the dislocation at the edges of the strip, for a translation invariant convolutional neural network, it is probably less so. Further experimentation is required to get the right inductive bias for this issue. The texture corruptions here are very mild, if an agent is very close to the textured object it may not be possible to identify the issue from a single observation. In short, the task is challenging.

As one might expect given the diversity of bugs that might manifest video games, none of the approaches presented are a magic bullet. Finding the right inductive biases for each bug is key, somewhat ironically given that our original aim was to avoid having to craft guards for this purpose. The point here is that these learning based methods are much more powerful (in terms of breadth and depth of what they can identify) than any collection of guards we can write. In the approaches we have seen, it is clear that data augmentation plays an important role in producing the right biases. As stated in the introduction to this chapter, this is one of the reasons contrastive learning is particularly suited to the task. It may be that S3N stands to benefit from other augmentation schemes beyond temporal views (as can be seen in the hybrid approach). Developing a method to reduce content blindness is another possible direction for improvement.

5.4 Related Work

5.4.1 Temporal Metric Learning

There are a number of works that make use of temporal associations when performing metric learning. In (Xu et al. 2019) video clips are randomly selected from a video then shuffled and the network is asked to predict their original order. The model is used downstream to retrieve videos that share similar content. (Kim et al. 2018) takes a similar approach with the addition of a spatial shuffle (similar to our experiments in video surveillance), to learn video representations. The learned representations are used downstream for human action recognition.

Time-contrastive networks (TCNs) were developed in (Sermanet et al. 2018) independently of our work. The approach follows our embedding approximation procedure closely, where positive examples (video frames) are chosen from a window around an anchor, and negative are chosen from further away in the same video clip. The work also introduces multiple views (i.e. multiple videos of the same scene recorded from different perspectives), with positive/negative pairs additionally being taken from these. Their downstream application is imitation learning, to train robots to imitate human action by watching video demonstrations. The use of different temporal views is a very natural extension from using spatial views, we (nor they) claim this as any kind of primary contribution. The main distinction with our work is in the differing use of the learned representations and overcoming of domain specific problems. Our exploration of similarity as a direct means to identify novelty in temporal data is our primary contribution as is discussed further in the next section.

5.4.2 Self-Supervised Learning for Anomaly Detection

Self-supervised learning (SSL) has been applied to anomaly detection in various ways. The most common approach taken when applying SSL methods to this problem is the following: (1) learn representations using contrastive learning, positive examples are in-distribution, negative examples are Out-Of-Distribution (OOD) (2) Train a simple classifier (e.g One-Class Support Vector Machine (OCSVM) or KDE) using these representations to identify anomalies. (Li et al. 2021) follows this process, data is augmented to contain anomalies that are similar to those they wish to identify and their model is trained contrastively. A Gaussian Kernel Density Estimator is trained on the resulting representations and low density examples are classified as abnormal. In (Alaverdyan et al. 2020) Siamese auto-encoders are used to learn representations of healthy brain tissue with the aim of identifying lesions. The approach tries to align brain images of different healthy patients in the latent space of an auto-encoder, OCSVMs are used to classify abnormality from the learned representations.

Other approaches do not rely on a subsequent classifier and instead use the representations directly and make use of data-augmentation. In (Masana et al. 2019), both augmented and curated negatives (real images of a different class) are used. Like us they use the learned similarity directly as a measure of abnormality. (Sehwag et al. 2021) make use of spatial views, representations are clustered, and test examples are compared to the cluster centroids. They also experiment with class labelled data to improve the selection of negative examples (negatives are taken from different classes). (Tack et al. 2020) uses the similarity between a newly observed example and the closest training example, along with the norm of the representation vector as a score. They augment data in various ways: introducing Gaussian noise, blurring, rotations and views, among others.

The reader is referred to survey (Hojjati et al. 2022) for a more complete review of SSL for anomaly and novelty detection. Our work differs from many of the above works, including many in the referenced survey in that S3N does not make use of a classifier or perform any kind of clustering. Our work is most similar to (Tack et al. 2020) which was done independently and published around the same time. Like them, we use similarity directly and the negative (or OOD) examples are created using simple data augmentations, in our case primarily temporal views. Comparing test examples to their closest training example also has parallels with our work, except that we compare with another *test* example that is temporally closest. While this approach has some limitations as we have seen in the last set of experiments, for certain kinds of problems it works well.

5.5 Conclusions

In this chapter contrastive learning was explored as a means to identify novelty for the purposes of regression testing in video games. S3N was developed and motivated as an approach to learning representations of an agent's experience playing games. It was shown empirically that S3N and other contrastive learning methods can identify a range of different bugs (applicability), some of which would otherwise be difficult to identify with handwritten guards (capability), in various different environments (reusability) all by looking at the screen as a human player would see it (decoupled). This chapter serves as a proof-of-concept for the application of learning to regression testing.

5.5.1 Limitations

S3N is limited in various ways. Firstly, it has a strong inductive bias towards modelling dynamics rather than content. We saw that this led to the content blindness problem, but does mean it is able to identify bugs that are related to dynamics. S3N also works over single transitions. Many other approaches to novelty detection in video (outside contrastive learning, e.g. (Lee et al. 2018)) consider wider temporal frames, this helps to give relevant context. This context is important for identifying certain kinds of bugs (e.g. unintended shortcut).

Aside from the individual limitations of the approach we took, our experimental setting is also limited. The problem of distinguishing intentional and unintentional updates was not investigated. An agent equipped with the methods presented in this chapter would identify new features as well as bugs as novel. In chapter 3 we outlined a simple procedure which may allow this distinction to be made. As a brief reminder, it involves alternating running the agent to obtain what has changed in the new version, fixing any bugs and leaving any features until only features remain, then retraining the agent. In this setup, an agent is a tool that summarizes what is different from the previous version. User studies are required to see if this would be too cumbersome in practice. There are also important technical details to work out, such as how an agent might deal with intended global changes to its experience (e.g. global lighting changes) so as not to flag all of its experience as novel.

5.5.2 Future Work

The results we have obtained are encouraging, but there is still much room for improvement and many possible avenues that could be taken in pursuit of better performance. Our focus was on self-supervised learning approaches as we believed this would give the best opportunity for shaping the biases that are required for identifying the diverse set of bugs that appear in video games. There are many other approaches to novelty detection that we did not explore which have been developed for use in related domains, such as those outlined in chapter 3. A more thorough comparison of these methods for bug identification would likely yield insights into which inductive biases work well. Another important future direction is to expand the experimental setting and explore how agents might deal with intended updates in regression testing.

In the next chapter, we leave contrastive learning, novelty detection and regression testing behind, and move on to another important problem in bug identification. Namely, to identify problems with the player's interaction with the game environment.

Chapter 6

Disentangling Reafference for Action Contingent Bug Identification

This chapter is an extended version of our paper *Disentangling Reafferent Effects by Doing Nothing* (Wilkins et al. 2023). The work is focused on learning models of action and their effects on observation. What sets video games apart from other kinds of media entertainment is their interactivity. As players, we are able to quickly grasp what impact we can have on the game through our actions. The first thing that a human player will do after picking up a new game is try out the controls. Our ability to form an internal model of the effects of our actions, even when they look nothing like those that we can perform in the real world, appears to be crucial for both game playing (e.g. for planning) and for identifying bugs that relate to action. We will refer to the problem of identifying action-related bugs such as the failure to open a door when the preconditions are met (e.g. unlocked) or unresponsiveness, as *action contingent bug identification.*

There are a number of avenues that we might take in developing learning methods for modelling action. An agent might, for example, model the associational relationship between action and observation (or effect), but this is limiting in a number of ways. The biggest problem is that external influences that are spuriously correlated with our action will also be modelled. Ideally, we want a model of action that is *independent* of external influences, i.e. a model of only what is *caused* by action. In other words, the agent should be able to distinguish between the *sensory effects* (changes in its observation) that are *self-caused* and those that are *externally-caused*, modelling these independently. In drawing this distinction and with a model of self-caused sensory effects, an agent would be able to ask questions of the form: *did my action have the* *expected effect?* and answer without conflating external influences, such as the actions of another agent, or other environmental occurrences.

Although originally motivated by the problem of action contingent bug identification, learning the distinction between sensory effects that are self-caused and those that are externally-caused is of much broader significance. In fact, our paper is presented without reference to the bug identification problem. It is instead presented as part of the more general discussion of AI with reference to work in the biological and cognitive sciences, robotics and causality. For the most part, this is how it will be presented in this chapter, with only occasional reference to bug identification. As such, it may require some reorientation on the part of the reader who has followed earlier chapters closely. Best efforts have been made to keep relevant concepts and notation consistent. It is also assumed that the reader is somewhat familiar with the fundamentals of causal inference[1].

To better ground the work in the theme of this thesis we develop the notion of *metamorphic action relations*. Since simply learning a model of action is not enough to make statements about intended behaviour, we instead resolve the test oracle problem using metamorphic testing (Chen et al. 2020c). A developer will need to specify relations between actions, for example, to say that the actions `OPEN` and `CLOSE` are inversely related. Given these metamorphic relations, an agent equipped with a causal model of its actions is able to determine whether there is a bug in the portion of the game code that handles player interaction.

The chapter is structured as follows, in section 6.1 we briefly present metamorphic testing and how one might go about doing action contingent bug identification with metamorphic relations. Section 6.2 forms the bulk of the chapter, it presents a thorough investigation of the problem of modelling self-caused effects and distinguishing them those that are externally-caused. In the first part of the section, the problem is framed as one of causal inference and a solution is derived. In the latter part, the approach is experimentally validated in three environments. In section 6.3 we present further experiments and discussion, this time centred around the problem of action contingent bug identification. Related work is presented in section 6.4, and finally the chapter is summarized, and future work is discussed in section 6.5.

6.1 Metamorphic Action Relations

In previous chapters we have dealt with the test oracle problem by essentially assuming it away, using a previous version of the game as an oracle. This is only possible in regression testing, where we have such

---

[1] (Neal 2020) offers an excellent introduction to the relevant concepts.

(a) x_1 (b) x_2 (c) x_3

Figure 6.1: Illustration of a player (the knight) opening and closing a door. The environment transitions as $x_1 \rightarrow x_2 \rightarrow x_3$ with the agent taking actions `OPEN` and then `CLOSE`. In the meantime, another agent (the goblin) is moving in the environment, and so $x_1 \neq x_3$ despite `CLOSE` being the inverse of `OPEN`. See text for details.

a version of the game. In this chapter we explore metamorphic testing as an alternative. The idea is to use the video game (in its current version) as its own test oracle by comparing multiple input/output pairs. Recalling the example presented in chapter 2: consider the function `merge(L1,L2)` that takes two lists and merges them into one ordered list. Since the output list is ordered, the following metamorphic relation holds: `merge(L1,L2) == merge(L2,L1)`.

Rather than lists, we are working with observations and actions. The function we wish to test is the environments transition function $\mathcal{T}$. Or, more specifically, the part of the transition function which handles how the player's actions influence observations (or the environment's state).

Consider for a moment a simple environment like that depicted in Fig. 6.1. Let the actions `OPEN` and `CLOSE` be the inverse of each other. Then, the following metamorphic relation can be defined:

$$x = \mathcal{T}(\mathcal{T}(x, \texttt{OPEN}), \texttt{CLOSE}) \tag{6.1}$$

In plain language this simply says: if the player opens the door, then closes it, they should again observe the door closed. If this relation doesn't hold for some x[2] and some ordering of the actions, then we can be sure there is a bug - either the door failed to open, or it failed to close. But there is an important caveat, the relation assumes the environment to be static - there are no changes in x that are not as a result of the player's action. Imagine for a moment that there is another agent in this environment (as in Fig. 6.1)

---

[2] the preconditions for action execution must also hold.

who can also perform actions. If this agent does so while the player is testing the relation, then clearly the environment will not end up in its original state and the relation doesn't hold. For it to hold, we need to isolate the change in observation that is due to the player's action, ignoring any other occurrences or external influences (the other agent or otherwise).

In more traditional testing terms, one might take steps to remove external influences completely. For example, by monitoring only the internal state of the door. In more complex instances, say where there is another agent who may also decide to open or close the door, we face the same issue. We might go further, and also remove anything that can influence the doors state, effectively placing the door and the player in a void.

Constructing what are effectively stripped-down test environments might get quite cumbersome if we are testing lots of different interactions. Instead, it would be better if like a human, a software agent, could look at the state of the world and disentangle the various factors that might influence whatever interaction is being tested, ignoring any that were not relevant to the given test. This is a very intuitive idea for us humans, we are (for the most part) able to recognise that opening a door is the inverse of closing it without conflating irrelevant environmental factors. Underlying this is our ability to attribute the cause of an occurrence to, for example, oneself or to another agent. Simply by taking actions and observing the result, we seem to be able to construct a kind of causal model that describes how said actions influence the environment or at least our perception of it.

The next section explores how an agent might create a distinction between what *it* does, and the other environmental happenings. Informed by the relevant literature in biology, the cognitive sciences, and AI, we attempt to operationalize the problem and develop a mechanism that enables an agent to draw the required distinction and subsequently learn the effects of its actions.

6.2 Disentangling Reafferent Effects

A seminal work (Holst 1954) on this distinction in biological agents coined the terms *reafference* and *exafference* to mean: the parts of an observation that are caused by the agent's own action, and the parts of an observation that are caused by external conditions or events respectively. The subtlety of the distinction is highlighted by Von Holst in his seminal paper:

> *If I shake the branch of a tree, various receptors of my skin and joints produce a reafference, but if I place my hand on a branch shaken by the wind, the stimuli of the same receptors produce an*

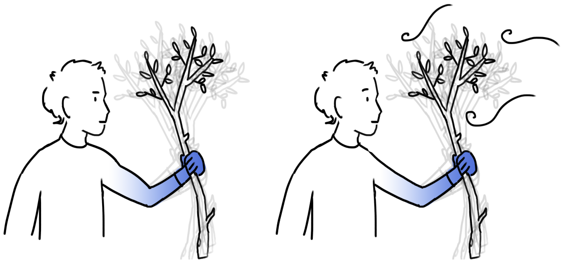

(a) (b)

Figure 6.2: Illustration of reafference (a) and exafference (b) and the subtlety of the distinction.

exafference.

The distinction has played a central role in developing theories to explain a broad range of physiological phenomena (Holst 1954; Blakemore et al. 2000; Wolpert et al. 2001; Medendorp 2011; Fukutomi et al. 2020). It has also appeared in investigations of higher-level cognitive functions. For example in mirroring (Rajmohan et al. 2007; Blakemore et al. 2005; Gallese et al. 1998) (theory of mind), the sense of agency (Haggard 2017) and the early development of *self*[3] (Lewis 2012; Jékely et al. 2021).

A theory of the mechanism that underpins these theories was made precise in (Miall et al. 1996), in which a comparative theory of reafference was introduced, see Fig. 6.3. The efference copy (or corollary discharge (Sperry 1950)), a copy of an internal outward motor signal or *action*, along with a forward model is used to estimate the reafferent sensory consequences (Kawato 1999; Wolpert et al. 1995; Wolpert et al. 2001). The estimate is compared with subsequent sensory data, and any error is attributed to exafference. The theory is illustrated in Fig. 6.3, along with an overview of our new proposal.

The new proposal aims to address a significant limitation of the comparative theory - that it is inherently *associational*. This means it is at best only an approximate distinction of reafference and exafference can be obtained. Since these are inherently *causal* notions, the theory is not sufficient. Why this is the case will be made more clear shortly in our discussion of reafference in AI. Further details of the comparative model and the surrounding literature can be found in the original paper (Wilkins et al. 2023).

6.2.1 Reafference in AI

In early work on reasoning about action in AI, the causal relationships between actions and their effects are assumed known and are represented as a program expressed in a logical form. The agent can query the logic

---

[3] there is some debate surrounding the extent of the role of reafference in the development of agency or *self* (Zaadnoordijk et al. 2019)

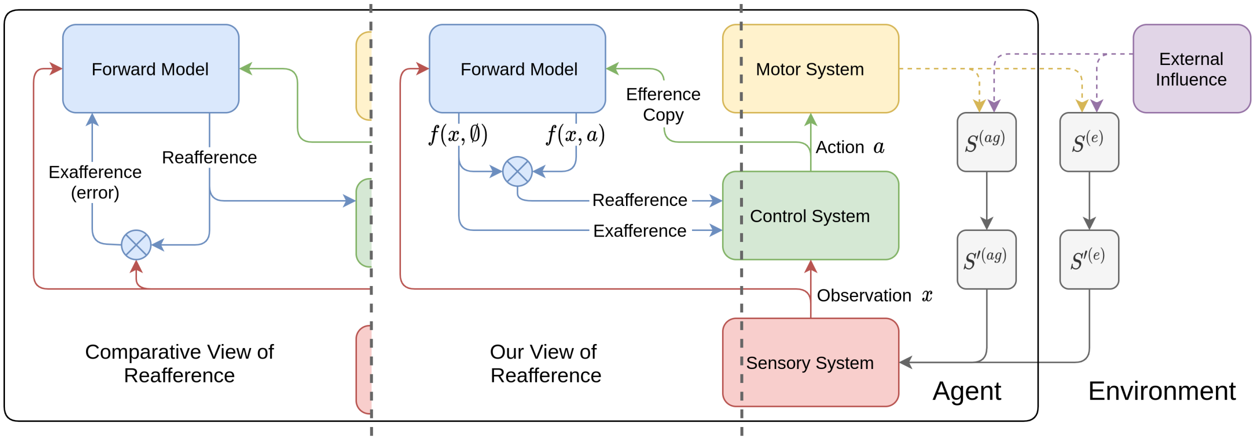


Figure 6.3: Comparative view of reafference vs. our view of reafference. In the comparative view the forward model estimates reafferent effects from the Efference Copy and the agent's observation. Reafference is compared with subsequent observations to determine exafference. It is not clear how the forward model comes to model reafference. In our view, the forward model estimates the total effect of taking a particular action $f(x, a)$ and this is subtracted from a *counterfactual* estimate of exafference $f(x, \emptyset)$ to obtain an estimate of reafference, where $\emptyset$ is the *do nothing* action. The forward model is trained according to the procedure outlined later in Alg. 1. The sections to follow outline this view in detail. The dotted edges indicate that actions may or may not have an effect on some aspect of the internal/external state.

program with an action to obtain the logical consequences, these are the self-caused effects of the agents action. Similarly, in robotics, an agent may devise a plan given an *a priori* forward model that is formed from the relevant laws of physics and the mechanical properties of its body. This is a kind of causal model that is based on *our* knowledge of physics; we have taken time to disentangle the relevant causal relationships ourselves.

Although planning in these settings is still a challenging problem and useful in many applications, we cannot always rely on knowing, or being able to specify the causal relations that are required for building a reafferent model for the agent. It is not always clear how each aspect of the agent's observation is related to its action; vision is a particularly difficult example. If as in the previous chapter, we wish to work with the fully rendered screen as the player sees it, specifying how the player's actions alter pixels on the screen is going to be impractical, bordering on impossible. This has led to the development of methods that instead try to learn or discover the relevant relationships.

This is what biological agents do in one form or another through their experience, or perhaps through their evolution. One view that is related to the comparative theory of reafference is that the predictability of a sensory signal determines whether it is reafferent or exafferent, with the reafferent signal being the more predictable. It is *easy* to estimate the reafferent signal produced by shaking a tree branch, but more difficult to predict the observed exafference that is due to the wind. The implementation of this view is typically quite crude, and rather ends up as a model of the association between action and sensory effect. One possible implementation removes external influence altogether, making exafference *unpredictable* only to the extent that a model has not previously seen these influences and is therefore bad at predicting their effects.

In (Schroder-Schetelig et al. 2010), a bipedal robot learns to walk. The forward model is trained when the robot is situated on a flat surface and the robot is later tested on sloped surfaces. The robot is successful in stabilising itself in the new sloped environment using its forward model and exafferent error signal. This approach has again deferred the problem of disentanglement, leaving it up to us to determine a suitable training environment. Although the agent is now free to learn the effects of its actions, it is generally difficult to create an environment that is free of external influence. In this instance, the learned forward model suffers from bias. It has not disentangled the effect on its observation that is due to gravity. Clearly the gravitational effect is not due to the agent's action and should be considered exafferent. If we sent this robot to Mars it would fail to stabilise since the forward model is working with the measure of Earth's gravity. One might argue that gravity need not be modelled as exafference if we are not on an interplanetary mission since it is constant on Earth. Nevertheless, there is a conceptual issue to address and that is that the gravitational effect is not *caused* by the agent's action, and that there are other similar variables of interest that may be difficult to control for. The assumption that underpins this experiment can be found in a number of other works (Bechtle et al. 2016; Schillaci et al. 2016). The essential issue with the approach is that the choice of environment determines what effects are considered reafferent.

In video games, where we are not constrained by physical laws, it may be much easier to control for external influences such as gravity. Nevertheless, doing so requires active decisions on the part of the developer to set up suitable test environments, and in some cases it may not be clear what to control for. Ideally, we would like the agent to learn reafferent effects without needing to set up specific test environments.

Going further still, in order to maximize return and therefore solve the task it has been given, a Reinforcement Learning (RL) agent must model the long-term effects of its action and typically does so via its value function. It is not clear to what extent model-free RL agents learn reafference. They are able to exploit and maximize return, but likely work with an associative model of an action's effect on observation (or return) rather than a causal one. What is clear is that RL agents are attentive to only those aspects relevant for maximizing return (Lapuschkin et al. 2019). A similar phenomenon is seen in the other learning paradigms, most clearly in supervised learning (Geirhos et al. 2020). This selective associational modelling of reafferent signals by RL agents leads to less robust policies, worse generalization performance, and exacerbates problems with learning long-term dependencies between action and return. If for example, early in training an agent finds that particular aspects of (or effects on) its observation lead to reward in the short-term, it may neglect to model those aspects that turn out to be relevant for obtaining more reward long-term.

In addressing some of these open problems, a number of works have tried to provide agents with a means to better learn the effects of their actions, often by introducing notions that are related to causality, such as counterfactuals (Buesing et al. 2018; Mesnard et al. 2021) or *imagination* (Schrittwieser et al. 2020). They have also been a key motivation for works most closely related to ours (Bellemare et al. 2012; Corcoll et al. 2020).

6.2.2 Reafference: A Causal Estimand

Now that importance of causality in this problem has been highlighted, we begin formalising reafference as a problem of causal inference. We again consider a Markov Decision Process (MDP) as in previous chapters. Actions are selected by an agent according to a policy $\pi(A_t = a_t | X_t = x_t)$ or by intervention $do(A_t = a_t)$ (i.e. to choose an action with no regard for observation, see next section). The time index t is useful for determining the direction of causal relations, however going forward we drop it as we are generally interested in the effect of an action on a particular observation x. We instead use X' to refer to the possible next observations $(i.e. X_{t+1})$ an agent might have after X_t. The causal graph that we work with is presented in Fig. 6.4, it represents a single step in the MDP.

Decisions, Actions & Interventions

To act is to bring about change in an environment. One popular formalisation of action comes from decision theory, where an agent's decision is represented as a variable A whose outcome is an action a. The effects of the action are determined by the relationship between A and the variables that represent the state of the environment. A decision is made by an agent given its observations and beliefs about the world, which are themselves variables, in pursuit of a goal. This is the view we have taken in earlier chapters.

In Pearl's conception of causal inference (Judea Pearl 2000), actions are instead represented as interventions, that is, changes to the underlying causal relationships between variables, and not as outcomes of decision variables. In their simplest form, they fix the value of a variable, for example $do(S' = s)$ would change the causal mechanism that gives rise to the next state of the world from say $S' := \mathcal{T}(S, A)$ to simply $S' := s$. $:=$ is used to mean *assignment* as distinct from equality.

To determine causal relations in practice we find ourselves intervening on variables that look very similar

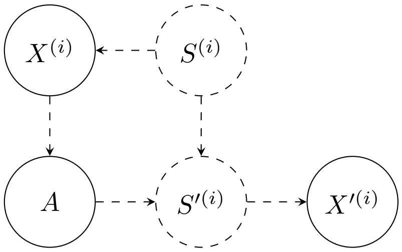


Figure 6.4: The causal graph that we are working with. At some moment in time, each $S^{(i)}$ is a random variable representing some part of the state e.g. a component of the agent position. Each $X^{(i)}$ is a variable representing some part of the agent observation. There is a many-to-one causal relationship between variables $S^{(i)}$ and each $X^{(i)}$, this relation is defined by the agents sensory mechanism. A is a random variable that represents the agent's decision. Again, there is a one-to-many causal relation between variables $X^{(i)}$ and A, which is defined by the agents decision-making mechanism. $S'^{(i)}$ and $X'^{(i)}$ are random variables representing parts of the *next* state and observation respectively. In the diagram, dashed nodes indicate unobserved variables (the agent never observes the outcomes of the underlying state). Dashed edges show the direction of causation as in a standard causal graph. They additionally indicate that some edges may be missing between variables in the corresponding collections (i.e. indicate the many-to-one relations). A particular variable that is present in the agent's observation $X^{(i)}$ may not have any bearing on its decision. Similarly, the action the agent decides upon may not affect every part of the state $S'^{(i)}$.

to what might be called decision variables in decision theory. To be concrete, consider the following prevalent introductory example: a new treatment T is to be tested for its effectiveness in combating a particular disease. The effect of T on patient health Y is to be estimated to determine whether the drug is suitable for wider use. Although T is typically referred to as the treatment variable, it in fact represents a medical practitioner's decision to give ($T = 1$), or not give ($T = 0$) treatment. The intervention is therefore a modification of the decision making mechanism; $do(T = 1)$ will fix the value of the decision to give treatment, regardless of patient heath, age, sex, etc. Of course, causal inference is more general than this, one can intervene on any observed state variable, not just those that look like decision variables.

For the purposes of our work, an action is the outcome of a decision variable. An intervention is a modification of the mechanism that determines the outcome of the decision variable, $do(A = a)$ sets the agent's decision to the action a without regard for prior observations or beliefs. This allows us to formalize reafference as *the changes in an agent's observation X' that are due to changes in the decision variable A.*

Doing Nothing

In the treatment problem presented above, T is a binary variable whose outcome is determined by the practitioners decision, give treatment, do not, or more generally, act, do not. Not acting, or *doing nothing* turns out to be crucial in determining reafference. This action, which we denote $\emptyset$ and refer to as the *null-action*, sets a baseline that allows the agent to reason as follows: *If I do nothing, there are only exafferent* *effects.* For many environments the choice of $\emptyset$ is quite natural, for example, as the action with least (ideally zero) expenditure of energy. Or, in videos games, where the action is commonly referred to as *noop* (no operation) and represents an absence of input from the player. The decision variable A may be continuous or discrete, but requires such a null-action to be explicitly chosen.

Potential Outcomes

The potential outcome (Rubin 1974) of an agent's observation x taking action a is denoted as $X'(A = a)$ or making the inner variable implicit for conciseness $X'(a)$. Potential outcomes are the possible observations an agent might make one time-step after observing x. A potential outcome is either observed or not observed as illustrated in the table below.

X	A	$X'(\emptyset)$	$X'(a)$
x_1	$\emptyset$	x_2	?
x_3	$\emptyset$	x_3	?
x_4	a	?	x_5
x_2	a	?	x_6

A potential outcome $X'(a)$ is said to be *factual* when it is observed - the agent actually took action a that lead to the outcome. A potential outcome $X'(a)$ is said to be *counterfactual* if it unobserved - the agent took an action other than a. We are interested in filling in the missing values of this table (the counterfactuals), that is, to ask questions like: *what would have happened if I had done a instead of $\emptyset$?* or visa-versa.

Reafferent Effects

The Average Causal Effect (ACE)[4], is a common causal estimand of interest. The Average Reafferent Effect (ARE) turns out to be well captured by this estimand and corresponds to the ACE of action on observation.

$$\delta(a) = \mathbb{E}[X'(a)] - \mathbb{E}[X'(\emptyset)] \tag{6.2}$$

---

[4] sometimes referred to as the Average Treatment Effect (ATE)

The ARE is taken over all observations. As an illustrative example, consider the following Structural Causal Model (SCM):

$$\begin{aligned} &A := Bernoulli(p_a) && Y := \mathcal{N}(0,1) && Y' := (A * Z) + Y \\ &Z := Bernoulli(p_z) && Z' := Bernoulli(p_z) && \end{aligned}$$

It depicts an agent taking a decision A to move forward in a simple environment. The action is chosen by a Bernoulli policy where p_a is the probability of taking action $A = 1$. If $A = 1$ the agent will attempt to move, if $A = 0$ the agent will remain stationary. Y is the agent's current position, Y' is the agent's next position, and Z indicates whether the space the agent wishes to occupy is already occupied ($Z = 0$) or is empty ($Z = 1$) where p_z is the probability of $Z = 1$. The agent's observation consists of (Y, Z), and subsequently (Y', Z'). $Z = 1$ is a precondition for the successful execution of the action $A = 1$, otherwise Y' takes the value of Y. By inspection[5] it can be seen that $\delta(1) = (p_z, 0)$. Similarly if the agent were to take action $A = 0$ then $\delta(0) = (0, 0)$.

The Individual Reafferent Effect (IRE)[6], or simply, the *reafferent effect*, is defined by conditioning on a specific observation:

$$\delta(a|X = x) = \mathbb{E}[X'(a)|X = x] - \mathbb{E}[X'(\emptyset)|X = x] \tag{6.3}$$

In the example above, we need only to condition on Z as the reafferent effect is invariant to Y. Again by inspection, $\delta(1|Z = 0) = (0, 0)$ and $\delta(1|Z = 1) = (1, 0)$, corresponding to the unsuccessful/successful act of moving into the filled/empty space respectively. Here we are computing the reafferent effect for each observation, that is, for every observed value of Z and Y. Crucially, we are able to compute the reafferent effect here because we have access to the underlying SCM. More generally, the agent does not know the mechanism, and the problem becomes one of estimating reafference from experience.

6.2.3 Estimating Reafference

Using the formalism presented in the previous sections, we develop a simple algorithm based on the difference-in-differences (DID) method[7] (*Difference-in-Difference Estimation | Columbia Public Health* n.d.) (see Fig.

---

[5] see Appendix A.3.5

[6] this terminology carries over from treatment effect estimation, where we are interested in the effect the treatment will have for some *individual* patient rather than its average over the population.

[7] DID is also referred to as the *controlled before-and-after study*.

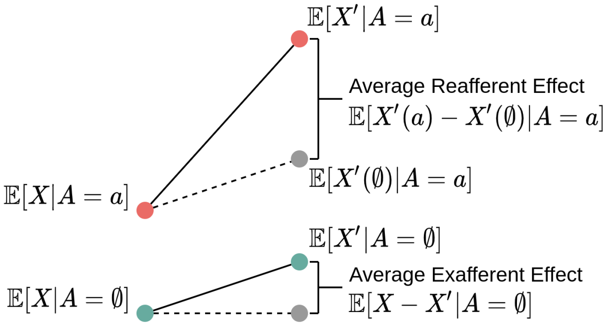


Figure 6.5: Difference-in-differences (DID) to get the Average Reafferent Effect (ARE). The Individual Reafferent Effect (IRE) can be found similarly by conditioning on the current observation x, with realisations of X' being the possible observations after x.

6.5) that trains a forward model to disentangle and estimate reafferent and exafferent effects. DID estimates the ARE simply by computing the difference in the expected outcomes of two groups. The groups in our case are the observations for which an agent takes action a, and the observations for which the agent takes no action $\emptyset$ (the control group). Various assumptions are required in order to do this, these are presented in section 6.2.3. Before this we examine the problem in more detail to help build intuition for the algorithm that is developed shortly after.

Counterfactual Structure of Reafference

Consider Fig. 6.6 it depicts two possible futures in the Atari Freeway environment[8]. In this environment the agent (a chicken) is attempting to cross a road while cars drive past. The agent, can move forward, backward, or stay where it is (null-action). The agent is presented with visual observations (images) and should disentangle the reafference, movement of its body, from exafference, cars driving past. In one possible future, the agent takes no action. This leaves only the exafferent effect $X'(\emptyset) - x$ which the agent observes (blue border). In another possible future the agent takes the action a (move forward) and observes the total effect $X'(a) - x$. This effect is a combination of its own movement and the movement of the cars. Rather than the total effect, we are interested in estimating the reafferent effect $X'(a) - X'(\emptyset)$, in other words, the movement of *only* the chicken. Unfortunately the potential outcome $X'(\emptyset)$ is not observed (red border) in this possible future and so is a *counterfactual* quantity. This is known as the *fundamental problem of causal inference* (Paul W. Holland 1986) - we (the agent) generally do not or cannot observe both $X'(a)$ and $X'(\emptyset)$ as they appear in different possible futures.

---

[8] see Appendix. B.2.8 for environment details

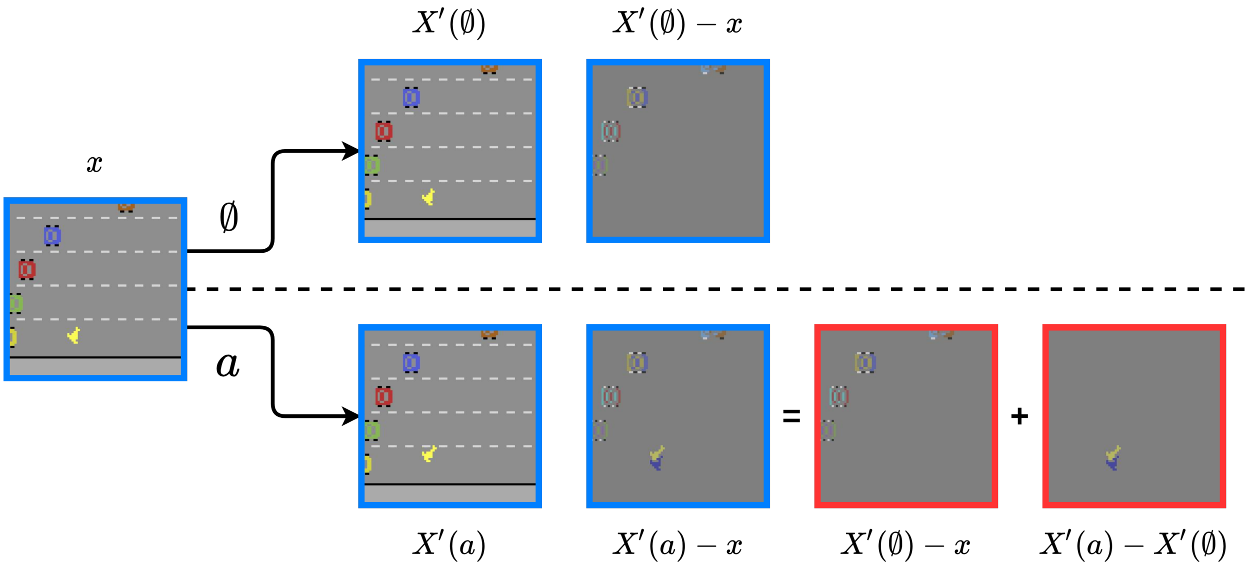


Figure 6.6: Counterfactual structure of the reafference problem. Effects have been normalised to $[0-1]$ for visualisation purposes. See text for discussion.

If we are interested in estimating the ARE then (under some assumptions) this problem can be overcome, but in this example (treating pixels as random variables) the ARE is not very useful. To estimate the reafferent effect, we need examples from both possible futures that are comparable and a model that is capable of interpolating (or extrapolating) well. Ideally, the agent would experience both possible futures, which in some restricted settings and in simulation is a possibility. In video games, the agent may experience both if it plays the game multiple times (or revisits certain states) but in general it will not be possible to try all actions in all states.

Assumptions

In order to estimate (average) reafferent effect using the DID method (or similar) a number of assumptions are required. Estimating the individual reafferent effect requires the additional assumption that *comparable* examples from both possible futures are observed. The other assumptions are listed below:

- **Positivity:** $P(A = a) > 0$ all actions must have some probability of being taken.

- **Consistency:** $\mathcal{A} = a \implies X' = X'(a)$. There is no unmeasured variation in how the decision is applied to X that causally influences the potential outcome. For our purposes, it says that there is only one way to take an action. For an in-depth discussion of this assumption see (VanderWeele 2009).

- **No interference:** The potential outcomes of an observation are unaffected by action taken for other observations. This assumption is violated if for example, actions have effects which are extended in time.

- **Modularity:** Interventions on A do not change the mechanism of any other variable.[9]

- **Unconfoundedness:**[10] Potential outcomes are independent of the agent's decision. There are no unknown confounders in the formalisation presented, the environment state contains all variables that might influence the next state (other than the agent's action). Although not all variables are observed by the agent, only those that the agent does observe have immediate causal relationships to its action, in other words, an agent makes a decision based only on what it observes. As such, conditioning on the observation X is enough to satisfy the backdoor criterion [11] and ensure unconfoundedness.

  Unconfoundedness can be broken in an alternative formulation in which actions are also dependent on unobserved state variables. Although it seems counter-intuitive that actions could be decided based on something that is not observed, it is helpful to remember that, at least for biological agents, taking action is not an instantaneous process. The line between agent and environment is not as clear as it is for artificial agents. A may be influenced by the state of the agent's body, not all of which forms part of X. If the value of A is *measured* after this happens then S can be said to directly influence A. Another source of confounding might come from an agent's beliefs. As they are derived from previous observations, unless they are properly conditioned upon, backdoor paths may be present. To avoid these confounding issues, we assume agents take action based only on the current observation, and that actions are not influenced by S. These assumptions are reflected in the causal graph presented in Fig. 6.4 but may not be realistic for more complex settings.

- **Parallel trends:** The exafferent effect is the same regardless of the action taken for a particular observation. This is required if we are to use DID.

- **Time independence:** Reafferent (and exafferent) effects do not change with on time. In other words, the environment is stationary. This assumption is broken if, for example, there is an unobserved state variable that interacts with time and with the action to produce its effect, such as age in biological agents. This assumption can be weakened, effects may change with time if this happens at a rate

---

[9] modularity is sometimes referred to as the *no fat hand* assumption.

[10] unconfoundedness is sometimes referred to as strong ignorability or exchangeability.

[11] see (Neal 2020) for details on the backdoor criterion.

substantially slower than the agent can learn. An experiment where the assumption is violated is presented in Appendix. A.3.4.

Disentangling Reafference

Now that we have built some intuition around the problem, our algorithm is presented (see Alg. 1)[12]. The algorithm uses Stochastic Gradient Descent (SGD) to train a forward model f_θ to estimate the total afferent effect. $f_\theta(x,a)$ is estimating the total effect $\mathbb{E}[X'|A=a,X=x]-\mathbb{E}[X|A=a,X=x]$. $f_\theta(x,\emptyset)$ is estimating the exafferent effect $\delta(\emptyset|X=x)$, which as we have seen is also the total effect for action $\emptyset$. The reafferent effect $\delta(a|X=x)$ is estimated as $f_\theta(x,a)-f_\theta(x,\emptyset)$. The reafferent estimand is expanded below:

$$\big[\,\mathbb{E}[X'|A=a,X=x]-\mathbb{E}[X|A=a,X=x]\big]- \tag{6.4}$$

$$\big[\,\mathbb{E}[X'(\emptyset)|A=a,X=x]-\mathbb{E}[X(\emptyset)|A=a,X=x]\big] \tag{6.5}$$

It is assumed that the observed and counterfactual expectations of the current observation (in red) are equal. This is trivially true if we are estimating the individual effect as X is observed[13]. When taking action a, the counterfactual quantity (in black) is estimated by extrapolating from observed instances of doing nothing. The reafferent effect is 0 when $a=\emptyset$ since the same model is used to compute both observed and counterfactual quantities. The estimated total effects for both a and $\emptyset$ are compared with the ground truth to obtain the loss.

The model is trained on batches of experience triples $(\mathbf{x},\mathbf{a},\mathbf{x}')$, each is a vector of observations/actions which are collected from multiple copies of the environment. This is a common trick that was devised in the deep reinforcement learning literature (Mnih et al. 2016) to try to bring elements of the mini-batch closer to IID. A replay buffer (Lin 1992; Schaul et al. 2016) might also be used to further reduce temporal correlation. Fig. 6.7 intuitively illustrates what is learned by Alg. 1 in a simple grid environment.

Estimating Average Reafferent Effects (ARE)

To estimate the ARE using Alg. 1 we might provide only the action as input. This may introduce bias since X is no longer conditioned upon. To remedy, the agent can perform a randomized trial by following a uniformly random policy, this will ensure that red terms in Eq. 6.5 are equal, provided there is no sampling

---

[12] See Appendix A.3.3 for further technical details on Alg. 1.

[13] When estimating the ARE this is true if there is no systematic bias introduced by the agent's policy, see next section for details.

Algorithm 1: Estimating Effects via SGD

```
Objective function $\mathcal{L}$ (MSE); Model $f_\theta$; Policy $\pi$;
Environments $env^{(i)}$;
Initial observations $x^{(i)} \sim env^{(i)}$;
while stopping criteria not met do
    Sample actions $a^{(i)} \sim \pi(x^{(i)})$;
    Take actions $x'^{(i)} \sim env^{(i)}(a^{(i)})$;
    Ground truth (total) effect $\delta = \mathbf{x}' - \mathbf{x}$;
    Estimate exafference $\hat{\delta}_\emptyset = f_\theta(\mathbf{x}, \emptyset)$;
    Estimate reafference $\hat{\delta}_a = f_\theta(\mathbf{x}, \mathbf{a}) - \hat{\delta}_\emptyset$;
    Gradient $\nabla_\theta \mathcal{L}(\hat{\delta}_a + \hat{\delta}_\emptyset, \delta; \theta)$;
    Apply Gradient Update $\theta \leftarrow \theta - \eta \nabla_\theta$;
    $x \leftarrow x'$;
end
```

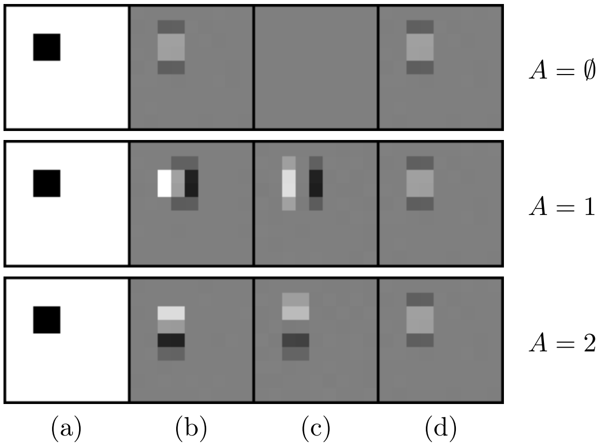


Figure 6.7: Disentangling in the Alone-v1 environment (see Appendix. B.1.1). (a) shows the agent's observation, a 9x9 pixel image, (b) shows the estimated total effect, (c) shows the estimated reafferent effect and (d) shows the estimated exafferent effect. In this environment, the agent can take the actions `NORTH,EAST,SOUTH,WEST` or $\emptyset$ to move the agent (shown as a black square) in the given direction. At each step the agent is randomly pushed north/south with probability 0.25, or remains in its current location with probability 0.5. This leads to the exafferent effect seen with action $A = \emptyset$. For actions $A = 1$ (`EAST`) and $A = 2$ (`SOUTH`), the reafferent effect is modelled with this random perturbation in mind. Since the most likely outcome is an absence of noise, for $A = 1$ the most likely displacement is eastward, with less chance of ending up north or south eastward (indicated by the fainter values). The case is similar for $A = 2$.

bias introduced by the policy. Sampling bias results from the fact that each observation may depend on those collected previously, essentially this is the policy dependence issue that was discussed in chapter 3. If sampling bias can be dealt with then estimating the ARE may be useful in some settings. Particularly for environments in which the effect is similar (or constant) across observations, in which case learning the ARE will give a better estimate in small sample size regimes.

Fig. 6.8 demonstrates the use of Alg. 1 to estimate the ARE in a simple environment in which reafference

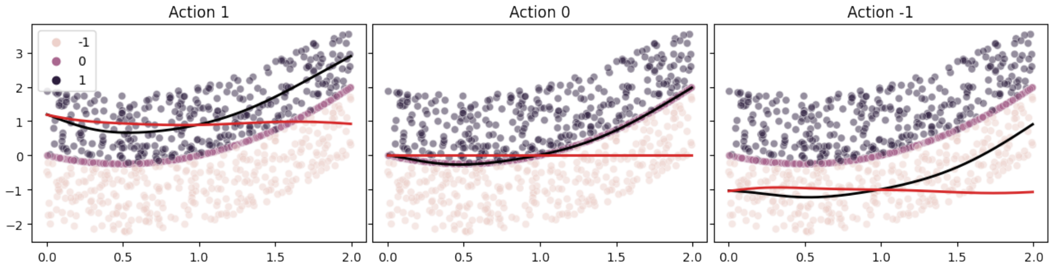
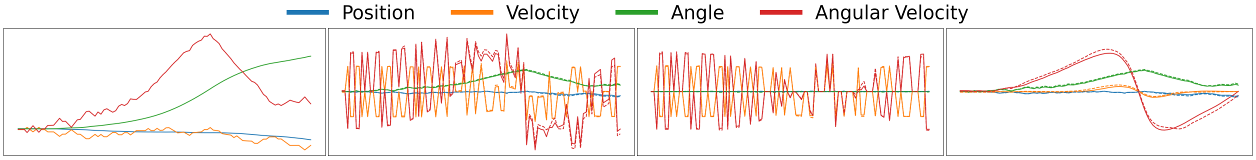


Figure 6.8: Example of estimating the ARE where action effects are the same (in expectation). The plots show the observed effects $x' - x$ (y-axis) plotted against the observation x (x-axis). Each graph shows estimates for a particular action. The red lines show the estimated reafferent effect for different values of x. It remains constant and so can be averaged over to get an estimate of the ARE that is useful. The black line shows the expected total effect, the upward trend is a result of the exafferent effect.



(a) (b) (c) (d)

Figure 6.9: Disentangling reafference and exafference in Cartpole. Observations and (estimated) effects are shown over time, with the agent taking an action at each step. Graphs show (a) observation, (b) total effects, (c) reafferent effects, and (d) exafferent effects. Estimated effects are shown as dotted lines. An action is an instantaneous force applied to the cart which has an instantaneous effect on the carts (angular) velocity. Actions do not have an instantaneous effect on the carts position or angle.

is equal in each observation and there is no sampling bias. The SCM for the environment is given below:

$$A := \pi(\{-1, 0, 1\}) \qquad X := U_X \qquad X' := X^2 + A \cdot U_{X'}$$

where π is a discrete uniformly random decision variable that selects actions from the set {-1,0,1}, and U_X and $U_{X'}$ are continuous uniformly random variables on the interval [0,2]. A two layer MLP is trained to estimate the effects using 1k observations. The ARE is estimated as 0.951 for $A = 1$; -1.015 for $A = -1$. The true reafferent effects are $1, -1$ respectively. The average exafferent effect ($A = 0$) is estimated as 0.317 and its true value is $1/3$. Notice that the population for which we are estimating the ARE is Independent and Identically Distributed (IID).

6.2.4 Experiments: Disentangling Effects

Here we demonstrate the use of Alg. 1 in a series of experiments on three different environments. Each showcases, or has a parallel with, an important concept or experiment performed in related fields. Each experiment is described in the sections to follow. Further details, as well as additional experiments are presented in the Appendix. A.3. All code and data is publicly available[14] as part of the supplementary material of the paper.

(i) Cartpole

In this experiment we show that an agent equipped with Alg. 1 can properly recover both reafferent and exafferent effects in a simple physical simulator environment, namely the Cartpole environment. Crucially, this includes properly modelling the constant gravitational effect, which is in contrast to the bipedal robot example (Schroder-Schetelig et al. 2010) presented earlier in section 6.2.1 in which the gravitational effect was absorbed into the forward model.

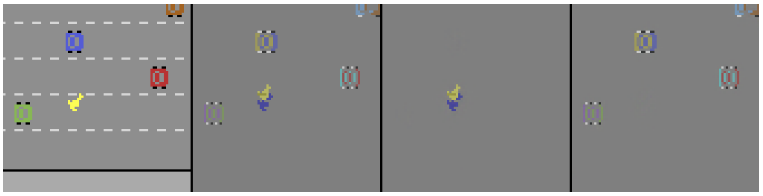
The Cartpole environment is a simple physical system with a cart that moves along a horizontal axis and a pole that should be balanced on top. This version of the environment has three actions $[-\beta, 0, \beta]$, each applies a horizontal force of magnitude β to the cart, with 0 applying no force (null-action). The agent observes the cart's position and velocity, and the pole's angle and angular velocity. Exafferent effects are due to gravitational acceleration, or velocities produced by previous actions. Reafferent effects are changes in the next observation that are due to the force applied to the cart by the agent. Results are shown in Fig. 6.9. Actions have an instantaneous effect only on the velocities and not on position or angle, this is an environment implementation detail that is reflected in the result.

(ii) Atari Freeway

The aim of this experiment is to show that our approach can separate the *body* of an agent from the rest of the environment. Additionally, it is to show that reafference may also be learned in the visual domain. We use the Atari Freeway environment as described previously section 6.2.3. The agent's body is meant loosely as the pixel representation of the agent. The agent does not explicitly recognize its body as an independent entity as we have given it no capacity to do so. It is however able to create a clean separation between reafference and exafference, where reafference happens to correspond to effects local to the region that we would call its body. Results are presented in Fig. 6.10. The result is what we would expect, it matches the ground truth as presented previously in section 6.2.3. The ability to distinguish body from environment is an important first step in developing embodied agents. Learning reafference is an avenue that may better enable agents to take this step. A deeper exploration of reafference in this context is needed, but is beyond

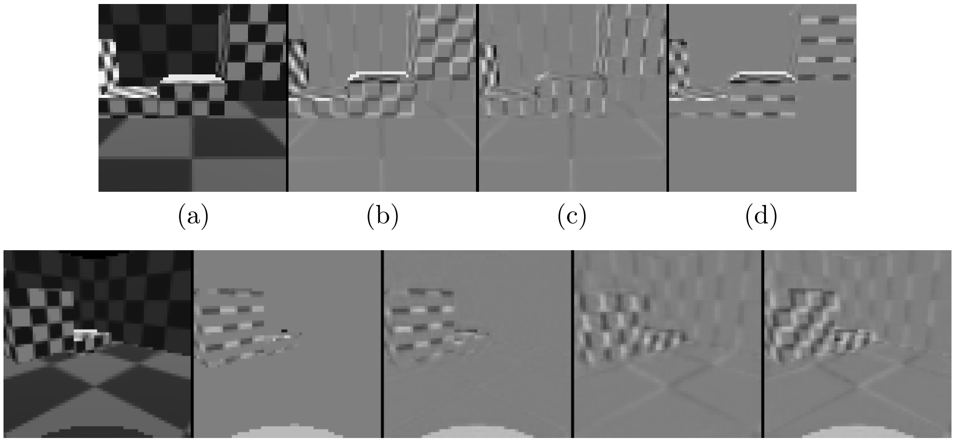
---

[14] https://github.com/BenedictWilkins/disentangling-reafference



(a) (b) (c) (d)

Figure 6.10: Disentangling effects in the Atari Freeway environment. Images show (a) current observation x, (b) ground truth total effect $x' - x$, (c) estimated reafferent effect, and (d) estimated exafferent effect. Effects are scaled $[-1, 1] \rightarrow [0, 1]$.



(a) (b) (c) (d)

(a) (b*) (b) (c) (d)

Figure 6.11: Disentangling in Artificial Ape environment. The top row shows results for the first experiment without the rotating platform, the bottom row shows results for the second experiment with congruent reafference and exafference. In each, (a) shows observation (b*) shows ground truth total effect (b) shows the estimated total effect (c) shows the estimated reafferent effect, and (d) shows the estimated exafferent effect. In the second experiment, the congruent effects cancel each other out (the agent and platforms rotate in different directions). Effects are scaled $[-1, 1] \rightarrow [0, 1]$

the scope of this thesis.

(iii) Artificial Ape

The neural mechanisms that underpin reafference, and in particular the comparative theory as presented earlier in Fig. 6.3, has been investigated in numerous works. In one study (Cullen 2004; Roy et al. 2004), an ape was placed on a rotating platform, restrained but with some freedom to move its head. The authors studied vestibular neuron signals and found that there was indeed a distinction made between passive (exafferent) and active (reafferent) movement of the head. In the experiments to follow, we show that an analogous distinction is made by an artificial agent using our method. In contrast to the previous experiments, here the exafferent and reafferent signals are *congruent*, meaning they have the *same* effect on the senses. The distinction is therefore more subtle and analogous to Von Holst's original example of the tree-branch blowing in the wind.

In the spirit of the study with apes, the Artificial Ape environment[15], which was developed using the WOB platform (Wilkins et al. 2022), places an agent into a 3D scene with a collection of moving cubes. The agent can rotate its view left and right or maintain its current view (null-action). The movement of the cubes is independent of the agent's action and so is always exafferent. The agent stands on a platform that is rotated randomly, rotating the agent's perspective with it.

In the first experiment we show that the agent is able to distinguish the movement of the cubes from the reafferent perspective shifts. In the second experiment, we introduce the passive exafferent perspective shift by randomly rotating the platform and show that the agent is further able to distinguish the congruent exafferent perspective shifts from the reafferent perspective shifts. Results for both are presented in Fig. 6.11. The agent's action is to rotate its view by a small angle, leading to the highlighted vertical edges seen in the estimated reafferent effect. The chequered cubes in the scene are moving up/down relative to the view leading to the highlighted horizontal edges seen in the predicted exafferent effect.

In the second experiment, to aid interpretation by reducing aleatoric uncertainty, the colour of the platform is an indicator of the future direction of motion of the platform. An analogue in the experiment with apes might be the whirr of the platform motor, feeling of acceleration in the rest of the body, or the fact that the platform is already rotating in some direction. This introduces some incongruence to the effects, however the vast majority of effects are congruent[16]. The result shows a cancellation effect in the signals that is similar to what is observed in the original experiment when the ape's head and platform rotate in opposite directions. The result suggests that our approach is viable as even in the more subtle case of congruence the agent is able to create the desired distinction.

6.3 Action Contingent Bug Identification

Now that the agent has a means to learn the effects of its actions, it can perform metamorphic testing as outlined in section 6.1. Metamorphic action relations can now be expressed in terms of reafferent effects rather than in terms of the full transition function. For example, we might define a relation between actions

---

[15] see Appendix. B.3.3 for further environment details.

[16] Another experiment with truly congruent effects is presented in Appendix. A.3.4

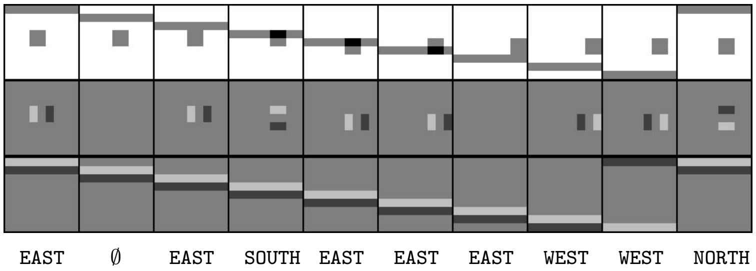


Figure 6.12: A short trajectory from Scan-v0 environment visualised. The top row of images shows the ground truth observation, the middle row shows reafference, and the bottom row shows exafference. Note the failure of action 7 (`EAST`), the agent cannot move outside the play area.

a and $\bar{a}$ that are the inverse of each:

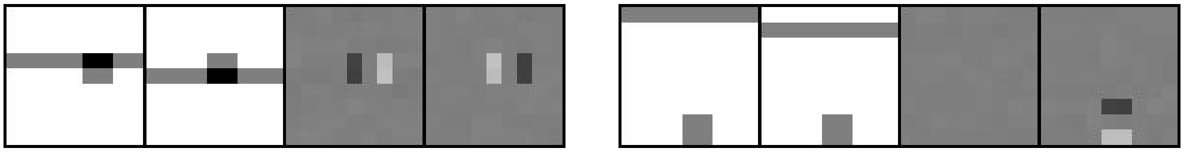
$$\delta(a|x) + \delta(\bar{a}|\mathcal{T}(x,a)) = 0 \tag{6.6}$$

where $\delta(a|x)$ is the reafferent effect of action a on observation x. In contrast to previous chapters, the reafferent forward model is trained in an unsupervised manner. The training data may contain any number of bugs, and we are defining relations in an attempt to catch these bugs. That is, the test oracle problem is being addressed by *us*. By having an agent learn reafference we have a means to specify metamorphic relations which might otherwise be difficult to specify, as was highlighted in section 6.1. In the sections to follow, we perform some simple experiments demonstrating how an agent might be used in this way.

6.3.1 Investigating Metamorphic Action Relations

The first experiment takes place in the Scan-v0 environment. The environment is very similar to the now familiar Alone-v0 environment, but is modified to contain a line that scans from top to bottom as illustrated in Fig. 6.12. The scanning line introduces exafference into the agent's observation. A simple forward model is trained by Alg. 1 such that it can disentangle exafference and reafference in this environment. The metamorphic relation in Eq. 6.6 should hold in this environment for the action pairs (`NORTH`, `SOUTH`), (`EAST`, `WEST`) and (∅,∅) in both directions.

To test, an agent takes a series of actions to produce some trajectory τ. For each observation x_t in τ, the agent computes the reafferent effect estimate $\hat{\delta}(a_t|x_t)$ and the second estimate $\hat{\delta}(\bar{a}_t|x_{t+1})$ where $\bar{a}_t$ is the inverse of a_t. The two estimates cannot be compared with equality because there will likely be some small error or noise in the estimate. But the equality can be approximated using a distance between them i.e. $||\hat{\delta}(a_t|x_t) - \hat{\delta}(\bar{a}_t|x_{t+1})||_2$. If the distance is relatively large (above the baseline error noise) then the relation



(a) relation satisfied

(b) relation violated

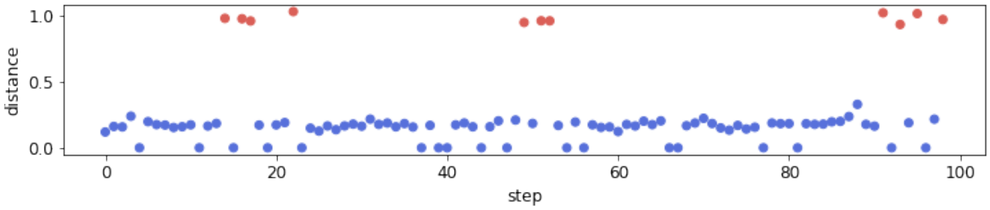


Figure 6.13: Results for applying the metamorphic relation in Eq. 6.6 to the Scan-v0 environment for pairs of inversely related actions. (a) shows the metamorphic relation satisfied for the actions (`EAST`, `WEST`). In (b) the agent attempts to move outside the play area (i.e. takes action `SOUTH`) which has no effect, the inverse action `NORTH` has an effect and the relation is therefore violated. Attempting to go out of bounds therefore constitutes a bug in this environment, at least according to the metamorphic relation that has been given. In each case, the agent *imagines* what would happen if it was to take the inverse action in x_{t+1}, it does not need to actually take the action. The plot shows the distance associated with each relation over a trajectory of length 100. Red points are those that violate the relation, blue are those that satisfy it. The baseline error noise is around 0.2.

has been violated. As it turns out, this environment does violate the relation, see Fig. 6.13 (b). This happens when an agent attempts to move out of bounds, the attempted action (e.g. `WEST` into the westward wall) will have no effect. If this is not a bug, then we can with relative ease, modify the metamorphic relation to include this case, for example, by qualifying it with some subset of the possible observations.

Identifying Unresponsiveness

Unrepsonsiveness is a simple kind of bug that is similar to a freeze bug, but that only impacts action. Other environmental processes (exafference) will continue as normal. When the player attempts to take any action, it will fail and have no effect. This might be checked with following iterative metamorphic relation.

$$0 < \sum_{t=0}^{|\mathcal{A}|} |\delta(a^{(t)}|x_t)| \tag{6.7}$$

where $a^{(t)}$ is an action in the set of actions available to the player $\mathcal{A}$. Actions are applied in subsequent states which result in each observation x_t. This reflects exactly how a human player would identify the unresponsiveness issue, by first trying some action in $\mathcal{A}$, finding that it doesn't work, trying the next, and so

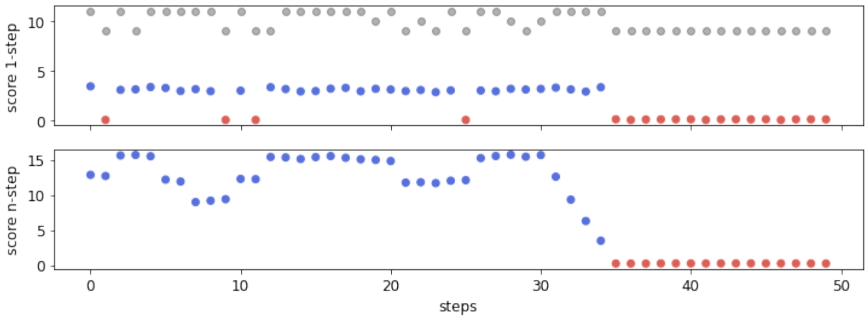


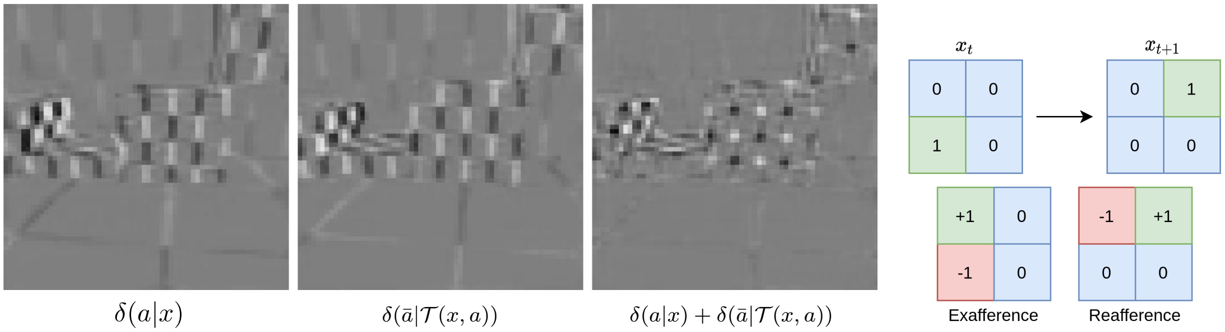
Figure 6.14: Identifying Unresponsiveness. Top plot shows the score $|\delta(a_t|x_t)|$ for the action taken. The grey points show $||x_{t+1} - x_t||$ i.e. the total change in the environment. One can imagine that this will be highly variable in more complex environments and so is not suitable as a score generally. The bottom plot shows the relation in eq. 6.7, each point is the sum of scores over the next 5 actions (the total number of available actions). After observing a sufficiently low score (red in the top plot) the agent will try each of the other actions in sequence. A red point in this plot indicates the presence of an unresponsiveness bug.

on. After each action is attempted, the environment may change independently, hence the need to formulate the test in terms of reafference.

Metamorphic relations such as the one presented in Eq. 6.7 require the agent to experiment in order to identify the bug. That is, perform experience-based testing. Incentivizing such behaviour is an important direction that has yet to be explored in the automated video game testing literature. We make no meaningful attempt at this here. In the experiment outlined in Fig. 6.14, the agent is given decision rules that produce the required behaviour - the initial failure of an action will prompt taking each available action one after another. With a forward model, one could just as well compare the same estimate for each action for the same observation. In this way the approach differs from how a human would test unresponsiveness, they are able to do after-the-fact reasoning i.e. disentangle reafference after making their next observation. It is not immediately clear how this kind of disentanglement could be done. If solved, it resolves problems with predicting stochastic effects since leveraging future information ensures determinacy.

Visual Reafference

The reafferent model that has been developed works by estimating changes in the variables in the agent's observation. When trained on images, each pixel in the image is a variable. In the experiments above we saw that the forward model might be used with visual observations. This was only possible because observation had properties that made writing relations relatively straightforward. More generally, writing these relations for visual observations is challenging. This is best highlighted with an example. Recall the Artificial Ape environment, the actions to rotate agents view `LEFT` and `RIGHT` are inversely related, at least with respect to



$\delta(a|x)$ $\delta(\bar{a}|\mathcal{T}(x,a))$ $\delta(a|x) + \delta(\bar{a}|\mathcal{T}(x,a))$

(a) (b)

Figure 6.15: When reafference depends on exafference. (a) shows the inverse relation (eq. 6.6) in the Artificial Ape environment. (b) shows this at a pixel level. Note how subtracting exafference from x_{t+1} will not reverse the action of the cubes. See text for further discussion.

the player's rotation. We might think then that the same will hold for the visual presentation of the scene, as was the case in section 6.3.1 for movement. But this is not so, as shown in Fig. 6.15. The reason is that the reafferent effect depends upon the exafferent effect for these variables. This unfortunately means that we cannot easily ask the forward model counterfactual questions about exafference, such as *what would have happened if the cubes had not moved?* This is in contrast to previous examples where reafference and exafference *are* independent and analogous questions can be asked simply by subtracting the exafferent signal from the observation. In terms of defining metamorphic relations in pixel space this is quite a significant limitation. We should instead prefer to train the model on variables where the action relations can be more easily defined. Doing so would adversely impact reusability, unless these variables can be discovered, e.g. by learning causal relations over abstract representations of the environment. See section 6.5.2 for further discussion.

6.3.2 Discussion

Why do we need learning, isn't there an easier way?

The short answer is, yes. But only as long as it is possible to set the state of the video game. If the agent can take the null action, record the exafferent effect, then revert to the state prior to doing nothing. It can then take another action and recover the reafferent effect. Setting the game state to *exactly* what it was a short time ago is not always easy to do, and many games do not support this by default (consider the difficulty of dealing with parallel compute or states on the GPU). Whether implementing *state setting* functionality, which will likely need to be a consideration from the start of development, is more difficult than training a reafferent forward model will be project dependent.

We might simply return to tradition and specify how actions affect certain parts of the state. But again, we would not get the benefits of using intelligent automation. A reafferent forward model is reusable in any new project or after an update to the game, it just needs training. It also strongly decouples testing since (with certain assumptions met) it can work with the visual presentation of the game. Aside from these benefits, the reafferent forward model might also help in making agents more proficient game players[17]. Data also should not be an issue since testing agents will generate a very large amount during play.

What if doing nothing does something?

A critical assumption is that the doing nothing has no effect as this action is the control for determining reafference. It may be that due to a bug in the game this action instead does *something*. This will bias the reafference estimates and potentially invalidate metamorphic relations. Unfortunately there is no easy check an agent can perform for this, at least not without some help from a human who already knows what doing nothing means. One could write a metamorphic relation that identifies the issue, but this would have to be over exafferent effects since the reafferent effect for doing nothing would be estimated as zero even if the assumption is violated.

Along a similar line, some games do not admit a null action. Turn-based games such as chess, where players cannot skip their turn are an example. Almost all video games have a null action of some kind since a human can't be expected to continuously provide inputs.

What about stochastic effects?

In real video games, it is more likely that bugs like unresponsiveness manifests stochastically (at least if we admit partial observability). In such cases it is difficult to write metamorphic relations while making use of a reafferent forward model. This is because the forward model is only modelling the expected effect. Modelling stochastic transitions beyond simply modelling the expectation is a very challenging problem in general. A seemingly simple approach where forecasting is not required (as is the case here) is to provide both the current and next observation as input to the model. Unfortunately, this is not applicable where counterfactuals are involved (i.e. in Alg. 1) as we do not have access to the *next* observation as input to estimate counterfactual exafference, as this is itself a counterfactual quantity. There are existing methods that aim to estimate transition distributions (e.g. (Hafner et al. 2022)) and work in estimating causal effect

---

[17] this has not been verified empirically for this work, but other similar works have shown that there is some benefit to doing this (Corcoll et al. 2020; Bellemare et al. 2012).

distributions (Rhodes et al. 2020), but it is not obvious how they would be applied here. Exploring stochastic effects is left as future work.

Multi-step Reafference

Reafference as defined thus far considers only single-step effects, that is, the effect of the agent's action on the next observation. In practice, effects may be extended in time, may be delayed, or we may want to model the effect of an action over multiple time steps. Each case is essentially asking *what effect does action a_t have on observation x_{t+n}?*. To estimate multi-step effects, the model would need to compute counterfactual estimates for all intermediate actions during training. Then at test time take $n-1$ consecutive null-actions for comparison. An exploration of this is beyond the scope of this thesis.

Biological Plausibility

In some of the earliest work on reafference, Helmholtz noted that if one presses gently on their eye the world appears to move, however remains stationary when the eye is moved by the extraocular muscles (Helmholtz et al. 1924). This suggests that there is some reafferent modulation of the signal in the latter case that keeps the world stationary whenever our eyes saccade. It might also suggest a gap in biological reafference, since in the first instance the eye movement is also self-caused, just by a different motor mechanism and yet it is treated as exafferent. The approach to modelling reafference that is developed here has no such gaps. This might be an indicator that the mechanism behind biological reafference differs in some important way, or just that evolution has found shortcuts in cases where modelling such effects is not necessary.

6.4 Related Work

6.4.1 Metamorphic Testing

To the best of our knowledge, this is a novel problem setting as there are no works that look at metamorphic testing of action effects[18]. A loosely related setting is testing agent behaviour to ensure that it will behave as intended. A seminal work in this area developed DeepTest (Tian et al. 2018). DeepTest is a learning-based framework which tests (via metamorphic relations) the behaviour of autonomous vehicles. The method

---

[18] (Liaqat et al. 2018) claims to be doing metamorphic testing for the game of chess, but in actuality they appear to be doing traditional guard-based testing, the key feature of metamorphic testing (comparing multiple input-output pairs with relations) is missing from the work.

involves checking whether the agent will produce the same action after perturbing its observation. The framework is fairly general, it could for example be applied to testing NPC behaviours.

6.4.2 Disentangling Reafference

Contingency Awareness

Contingency awareness (Watson 1966), a close conceptual relative of reafference is investigated in (Bellemare et al. 2012). The term *contingent regions* was coined to mean the region of an observation that is affected by an agent's action. From a causal perspective, (Corcoll et al. 2020) defines a measure similar to that used to determine contingent regions. These measures are similar to our work in that action effects are compared to determine a causal relation. However, they do not determine the causal *extent* of the relation. It is noted in (Corcoll et al. 2020) that a special *do nothing* would not work well for estimating effects, arguing that doing nothing still has an effect on the observation (or at least the return). We believe this to be a conceptual oversight. While it is true that there is an effect on the observation, this effect should be ascribed to environmental influence. If the null-action is taken there is by our definition no edge from A to X' in the causal graph. This baseline is what allows us to determine the extent of the causal relation between the other actions and the agent's observation.

Associational Approaches

The following approaches take advantage of strong regularisation or implicit model biases to perform some kind of disentanglement. The general idea is to condition on actions and inspect the internals of a model, or use salience maps, to determine the controllable regions of the observation. (Choi et al. 2019) takes advantage of spatial attention mechanisms and trains an inverse-dynamics model, (Yang et al. 2019) uses an action-conditioned beta-VAE, similarly (Zhong et al. 2020) uses an action-information bottleneck with strong regularisation and (Oh et al. 2015) learns action-conditioned dynamics models. These works differ from ours in that they learn associational relations between action and observation. Additionally, measures of the relation are strongly subject to hyper-parameter choices and are biased by exafferent effects.

Controllable Factors of Variation

One line of work (Thomas et al. 2018; Bengio et al. 2017; Sawada 2018) defines and makes use of *selectivity* as a measure of what they call *independent controllable factors of variation*. These factors correspond to aspects of the environment that are *controllable* independently of other aspects, for example, the chicken in the Freeway environment. There are parallels with our work in that the changes in these factors would for the most part be represented as reafferent effects. However, rather than effects, they are more abstract latent representations of what can be independently *done* in an environment. In Cartpole for example, the cart and pole as a whole would be modelled as a factor, and the chicken in Freeway as another. The factors in Artificial Ape are less clear.

6.5 Conclusions

This chapter investigated reafference in the context of artificial agents and AI and its application to action contingent bug identification. Reafference was formalised as a causal estimand that can be estimated by a counterfactual comparison of doing nothing and doing something. The formalisation presented offers a new and alternative perspective on how agents come to know the effects of their actions or otherwise distinguish self-caused from externally-caused sensory effects.

Knowing the effects of one's action is useful for solving downstream tasks such as planning, or in this instance, action contingent bug identification. Assuming the agent's observations have certain properties, metamorphic relations can be defined in terms of reafferent effects over visual observations. With these relations and some relatively straightforward experimentation by the agent, action contingent bugs such as unresponsiveness can be identified. Identifying issues with how an agent interacts with its environment is a novel problem domain. The work presented here is a first step towards a solution which, given further refinement, will likely be a valuable addition to a future learning-based testing toolkit.

6.5.1 Limitations

Regarding reusability, the fact that metamorphic action relations need to be specified for each new environment takes away from our goal of having a reusable identifier for action related issues. There were also problems with specifying relations over stochastic effects, particularly because our method only models *expected effects*, which may not be representative of the underlying distribution. When estimating *individual* reafferent effects in the general setting where environments cannot be reset our method relies on the generalization ability of the forward model. Fortunately for bug identification, states may be visited more than once (in different trajectories, or the same trajectory) which alleviates the problem.

6.5.2 Future Work

One interesting direction for future work in action contingent bug identification is in the development of agents that can discover metamorphic relations from the learned reafferent effects. This is rather than specifying them manually which hinders reusability. As long as the relations could be presented to the tester in a straightforward fashion (e.g. as probabilistic rules), a tester could then evaluate the discovered relations to see if they match with what is intended for the particular game. Along similar lines, an agent might simply report statistics over effects. One can imagine for example, that the effect of the action `OPEN` should always be the same in the presence of an (unlocked) door, any deviation might constitute a bug. One might apply notions of statistical normality to the distributions of effects for each action.

In our experiment with visual effects in section 6.3.1 we saw that asking counterfactual questions did not always give the result we might expect. Another interesting direction for future work is to learn the causal relationship between action and internal abstract representations of observation rather than over the raw sensory input as our method does. Working with representations that have clear independence relations and are affected linearly by action (as is the case with many internal state variables, e.g. position) may also make it easier for an agent to discover metamorphic action relations. To keep agents as reusable as possible, one might make use of the work done in disentangled representation learning (Locatello et al. 2019) to learn these representations from visual experience.

Chapter 7

Conclusions

Producing fun, high-quality, bug-free games is the priority for anyone in the business of video game development. The introduction of intelligent software agents into the development toolkit promises to revolutionise video game testing. By locating bugs and evaluating important design criteria through play, these agents promise to save countless hours of manual testing, ultimately leading developers to produce better games.

In this thesis, we aimed to develop intelligent testing agents that can identify bugs by learning from their experience. More specifically, to develop agents that are capable, in that they can identify bugs that may otherwise be difficult using traditional methods, and reusable in that they can be used across projects or versions of a game with minimal manual alteration. In pursuit of this aim, four research objectives were set, each being an important step in the development of these agents. They are reiterated here:

Objective 1: To develop learning objectives that will allow agents to identify bugs in video games, especially those that would otherwise require human involvement to be identified.

Objective 2: To develop capable and reusable agents that can make use of *weak* supervision to perform regression testing.

Objective 3: To investigate and pinpoint one or more of the capabilities that enable humans to be proficient testers, and attempt to operationalize them with the goal of developing more general testing agents.

Objective 4: To curate and make available the data required to train and evaluate video game testing agents. The data must be labelled and include diverse examples of realistic video game bugs.

In the next section, each chapter is evaluated with respect to its achievement of these objectives. In doing so, we summarise the key contributions, challenges, and lessons learned.

7.1 Thesis Summary

7.1.1 Chapter 3: Learning and Bug Identification

In chapter 3, we explored the relationship between experience, learning, bug identification, anomaly detection, and the test oracle problem. We argued for the use of learning in video game testing as a means of dealing with our limited ability to formally or exactly specify what is intended (or unintended). At a high-level, testing agents require additional information to fill in the gaps left by our informal specification of their objective. As is the case for a human tester, this information must come from broader experience. The task of specifying intended behaviour is therefore to give a testing agent ways of leveraging or learning from its experience to achieve the task at hand. This is the essential problem faced in machine learning and AI more broadly. In this framing, bug identification has strong parallels with the field of anomaly detection; these were examined in some depth. This chapter strongly motivated the main thesis objectives, and indirectly contributed to their achievement by highlighting the key challenges involved and giving direction to subsequent research. The main contribution of the chapter lies in our analysis and framing of the problem as one in which AI (in particular agents with learning capabilities) plays a principal role.

Key Takeaways

Developing general video game testing agents will require progress on some of the most important problems in AI. The test oracle problem is a quintessential example of the AI alignment problem as agents must act to achieve test goals while only having them specified abstractly or informally. While general testing agents would provide the most value, the current research in anomaly detection and ML is poised to make major contributions to the video game testing toolkit. This is not just in game playing, which has occupied the attention of more recent work in this area, but also in the identification of bugs, which is comparatively understudied. Where automated bug detection is concerned, these two problems are deeply entangled, to such an extent that they may even be treated as one.

7.1.2 Chapter 4: A Platform for Automated Bug Detection in Video Games

In chapter 4 we made significant progress in addressing the fourth thesis objective. World of Bugs WOB (Wilkins et al. 2022) is the first example of a platform that makes available realistic video game bugs in varied and challenging environments. We have generated multiple, high-quality, freely available datasets that researchers can use to train and evaluate their approaches to automated bug detection. Three of these datasets were used to evaluate the work presented in this thesis, first in testing the platform itself as part of this chapter, then in chapter 5 in our investigation of contrastive learning for regression testing, and finally in chapter 6 when evaluating our approach to disentangling sensory effects. By bringing together key existing tools and technology, improving upon it and implementing numerous video game environments and bugs, the platform is supporting research into both learning-based bug identification and game-playing. Documentation, code, datasets, and related research can be found in the platform repository[1].

In the latter part of the chapter, we took tentative steps towards addressing the first and second thesis objectives by showing that a simple neural network classifier could identify the texture corruption bug in visual observations collected by an agent. We then showed that the agent could subsequently identify *unintended* changes to the WOB environment under test by using classification error as a measure of novelty. This simple demonstration highlighted many of the challenges that were discussed in chapter 3, especially the difficulties we face in distinguishing intended and unintended behaviour, but did little to address them. Nevertheless, the results were promising and served to highlight the potential of learning-based approaches.

Key Takeaways

The availability of data is important for engaging the wider research community in the problem of Automated Bug Detection (ABD). For the first time, this data is now freely available. The platform's purpose is to support research in this area and encourage progress on AI and learning-based solutions to the bug identification problem. The platform has opened up the opportunity for research in important directions, such as developing more efficient exploration directed by knowledge of the bugs that are to be identified.

7.1.3 Chapter 5: Contrastive Learning for Automated Bug Identification

In chapter 5 we explored contrastive Self-Supervised Learning (SSL) as a means to regression test video games (Wilkins et al. 2020). We developed State-State Siamese Networks (S3N) as an approach to novelty detection that worked by contrasting learned representations of an agent's experience. S3N was successful in identifying certain bugs in 2D games, including unintended shortcut bugs in a maze game, and various visual artefacts introduced into a selection of Atari 2600 games. It also performed reasonably well in identifying some challenging bugs presented in the 3D WOB environment `Maze-v1`, including Z-fighting, unintended object and unintended shortcut. We tested two other contrastive approaches that made use of data augmentation to instil biases that were suited to identifying the other bugs present in the environment, including terrain

---

[1] https://benedictwilkins.github.io/world-of-bugs/

hole, player out of bounds and high force.

Some of these bugs may be identified using more traditional approaches, but the major advantage of using the learning-based tests is reusability. Each of the games in our experiments was tested using the same contrastive learning algorithm[2]. The reusability, in this case, comes primarily from the fact that the agent is operating in the visual domain, meaning it does not need to be integrated with the underlying game implementation. S3N was able to identify the bugs that matched its built-in normality biases, which were specifically around the game's dynamics (or how the agent's experiences change through time). The approaches we experimented with also demonstrated some capability as they were able to identify bugs like unintended shortcut and Z-fighting. Writing guards to identify these bugs is not straightforward even when operating directly on the game's state. In developing S3N and demonstrating reusability and capability we have made substantial progress towards the first and second thesis objectives. Our work is among the first to demonstrate the use of learning in identifying bugs in the visual domain.

The results we obtained are very encouraging, however, the setting in which we evaluated our approaches is limited in two crucial ways which urgently require further investigation if these solutions are to be considered industrially viable. (1) the fact that a working version of the game is required as a training oracle, and (2) that we did not attempt to distinguish between intentional and unintentional changes to a game, all changes were assumed unintentional. Both limit our methods to a very specific kind of regression testing. Despite these limitations, the work is an important first step in developing more intelligent testing agents that operate over the same input/output domain as human testers.

Key Takeaways

Testing agents equipped with modern machine learning methods can be used to identify video game bugs in the visual presentation of the game as would be seen by a human player. Working in the visual domain decouples the tests from a game's underlying implementation, making them highly reusable. Game-specific details may be learned rather than manually specified, further improving reusability. With the right inductive biases, the same agent can be used to identify bugs of various kinds, including some that would usually require human testers to find. These claims were verified in a highly restricted setting. Further work is required to make them viable in practice.

---

[2] different hyperparameters were used, but these might be chosen automatically using auto-ML methods.

7.1.4 Chapter 6: Disentangling Reafference for Action Contingent Bug Identification

In chapter 6 we gained insight into an important capability of the quintessential test oracle - a human tester. We started by reasoning that one of the most fundamental aspects of any video game is player interaction. Among the first things that a new (human) player does is experiment with the controls to gain some understanding of how they can influence the game world. In doing so, an internal model of the effects of their actions is constructed, which is updated and refined as they continue to play. The construction of such a model is the capability that we chose to investigate as it is important for playing and planning, but is also crucial for identifying bugs that are related to action, such as unresponsiveness.

In our investigation, we examined the biological and cognitive sciences literature and drew parallels with the work that has been done in AI in this area. It was clear that much of the work was focused on the associational relationship between action and effect, but that a stronger *causal* relationship was much more desirable. We developed a causal framework that captured the relationship between an agent's action and its immediate causal effect. We then gave an algorithm that enabled an agent to learn this relationship by leveraging a counterfactual comparison of *doing nothing* and *doing something.* We evaluated the method on three different environments and found that agents were indeed able to learn the desired causal relations. To ground the work in the theme of this thesis, we devised the notion of *metamorphic action relations* as an instance of metamorphic testing that made use of learned action effects. This dealt with the test oracle problem and enabled an agent with this capability to perform action contingent bug identification.

Our agents were limited by the fact that the causal model was over the raw sensory input space (e.g. pixel space), rather than over abstract representations of observation. While they were learning a causal model, it was not a *general* model of action comparable to that of a human tester. Developing more general models of action in which counterfactual questions can be asked serves as an important but extremely challenging direction for future work. Our formalization and method serve as a rudimentary step in developing more general biologically inspired agents, both in video game testing and in AI more broadly.

Key Takeaways

The construction of an internal causal model of our environment and how our actions affect it plays a central role in many cognitive processes. The ability to construct and use these models effectively is part of what makes humans such proficient testers. An agent that can learn a model of action is far more reusable than one that requires this model to be specified. Such a model enables the agent to reason about the effects of its action and experiment in order to search for and identify bugs.

7.2 Limitations

The specific limitations of the work in this thesis are presented at the end of each chapter. Here the broader thesis-wide limitations are given.

Much of the work in this thesis focused on showing that agents with learning capabilities can be used to identify bugs. Our experiments are idealized in many ways and do not reflect the full complexity that is faced during the video game development and testing process. Much of this complexity was highlighted in chapter 3. Some of the most pressing issues that we did little to address include: How a testing agent might deal with the changes to its environment brought on by the development process, how to deal with the sampling bias introduced by the agent's policy in large environments, and how to address the test oracle problem in earlier stages of development/testing. Aside from the more technical issues, there are also those that relate to human-computer interaction, such as, how test results obtained by a testing agent might be reported to developer for maximum productivity. As well as more practical deployment problems, such how these agents can be integrated into existing testing/development toolkits and life-cycles. Addressing these problems is essential if agents are to be practically deployed as we have envisioned.

7.3 Future Work

Future work is discussed at the end of each chapter. Here we present broad directions for future work that look beyond our experiments and discussion in each of the thesis chapters.

- Not all the bugs that have been made available in the WOB platform have been explored in this thesis. The bugs that went unexplored are those that were not identifiable with the methods that this thesis develops, which tend to be more in the visual domain. An example is the stuck bug present in the `GettingStuck-v0` environment. While there may be visual indicators that might be exploited to identify the bug, it might be better dealt with using methods developed for reachability analysis. Along similar lines, video games are as diverse as the bugs they exhibit, and while we were varied in the games we tested, there are still many genres that remain unexplored.

- The kind of reusability that was demonstrated in chapter 5 required that the agent was retrained on each new video game. In the future, an agent that, at most, requires fine-tuning on new video games would be preferable. This may be within reach for certain always bugs such as geometry corruption. Developing reusable tests for always bugs (e.g. using techniques in transfer learning) is an important direction for future work.

- At various points in this thesis we alluded to *experimentation* and *experience-based testing* as an integral part of the testing process. It is not enough that an agent simply plays a game with the hope that it happens upon a bug here and there. An experienced human tester will actively experiment and direct their play towards finding certain kinds of problems. This is important for making the search process more efficient. Integrating knowledge of bugs into behaviour may be easier if both are learned. Developing testing agents with this capability is an open problem and an exciting direction for future work.

Bibliography

Alaverdyan, Zaruhi, Julien Jung, Romain Bouet, and Carole Lartizien (2020). “Regularized siamese neural network for unsupervised outlier detection on brain multiparametric magnetic resonance imaging: Application to epilepsy lesion screening”. In: *Medical Image Analysis* 60, p. 101618. ISSN: 1361-8415. DOI: `https://doi.org/10.1016/j.media.2019.101618`. URL: `https://www.sciencedirect.com/science/article/pii/S1361841519301562`.

Alenezi, Mamdouh and Shadi Banitaan (Dec. 2013). “Bug Reports Prioritization: Which Features and Classifier to Use?” In: *2013 12th International Conference on Machine Learning and Applications*. Vol. 2, pp. 112–116. DOI: `10.1109/ICMLA.2013.114`.

Aljundi, Rahaf, Daniel Olmeda Reino, Nikolay Chumerin, and Richard E. Turner (Sept. 2022). “Continual Novelty Detection”. en. In: *arXiv*. URL: `http://arxiv.org/abs/2106.12964`.

Allamanis, Miltiadis, Earl Barr, Premkumar Devanbu, and Charles Sutton (2017). “A Survey of Machine Learning for Big Code and Naturalness”. In: *ACM Computing Surveys* 51. DOI: `10.1145/3212695`.

Andrews, Anneliese A., Jeff Offutt, and Roger T. Alexander (July 2005). “Testing Web applications by modeling with FSMs”. en. In: *Software & Systems Modeling* 4.3, pp. 326–345. ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-004-0077-7`. URL: `http://link.springer.com/10.1007/s10270-004-0077-7` (visited on 02/20/2023).

Ariyurek, Sinan, Aysu Betin-Can, and Elif Surer (2020). “Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing”. In: *IEEE Conference on Computatonal Intelligence and Games, CIG*. Vol. 2020-Augus. arXiv: 2003.07813 ISSN: 23254289. IEEE Computer Society, pp. 25–32. ISBN: 978-1-72814-533-4. DOI: `10.1109/CoG47356.2020.9231670`. (Visited on 09/30/2021).

Aubret, Arthur, Laetitia Matignon, and Salima Hassas (Aug. 2019). “A survey on intrinsic motivation in reinforcement learning”. In: *arXiv*. URL: `http://arxiv.org/abs/1908.06976`.

Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr (Jan. 2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing". en. In: *IEEE Transactions on Dependable and Secure Computing* 1.1, pp. 11–33. ISSN: 1545-5971. DOI: `10.1109/TDSC.2004.2`.

Azadvar, Ahmad and Alessandro Canossa (Aug. 2018). "UPEQ: ubisoft perceived experience questionnaire: a self-determination evaluation tool for video games". en. In: *Proceedings of the 13th International Conference on the Foundations of Digital Games*. Malmö Sweden: ACM, pp. 1–7. ISBN: 978-1-4503-6571-0. DOI: `10.1145/3235765.3235780`.

Bachman, Philip, R Devon Hjelm, and William Buchwalter (2019). "Learning Representations by Maximizing Mutual Information across Views". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc.

Bainbridge, Wilma Alice and William Sims Bainbridge (Feb. 2007). "Creative Uses of Software Errors: Glitches and Cheats". In: *Social Science Computer Review* 25.1, pp. 61–77. ISSN: 0894-4393, 1552-8286. DOI: `10.1177/0894439306289510`. URL: `http://journals.sagepub.com/doi/10.1177/0894439306289510` (visited on 05/27/2022).

Baker, Bowen, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch (Feb. 2020). "Emergent Tool Use From Multi-Agent Autocurricula". In: *arXiv*. URL: `http://arxiv.org/abs/1909.07528` (visited on 10/19/2022).

Balestriero, Randall and Yann LeCun (June 2022). "Contrastive and Non-Contrastive Self-Supervised Learning Recover Global and Local Spectral Embedding Methods". In: arXiv:2205.11508 [cs, math, stat]. DOI: `10.48550/arXiv.2205.11508`. URL: `http://arxiv.org/abs/2205.11508` (visited on 12/08/2022).

Barr, Earl T., Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo (2015). "The Oracle Problem in Software Testing: A Survey". en. In: *IEEE Transactions on Software Engineering* 41.5, pp. 507–525. ISSN: 0098-5589, 1939-3520. DOI: `10.1109/TSE.2014.2372785`.

Barthet, Matthew, Antonios Liapis, and Georgios N. Yannakakis (Sept. 2021). "Go-Blend behavior and affect". en. In: arXiv:2109.13388 [cs]. URL: `http://arxiv.org/abs/2109.13388`.

Beattie, Charles, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen (Dec. 2016). "DeepMind Lab". en. In: *arXiv*. URL: `http://arxiv.org/abs/1612.03801` (visited on 06/07/2022).

Bechtle, Sarah, Guido Schillaci, and Verena V. Hafner (Sept. 2016). “On the sense of agency and of object permanence in robots”. In: *2016 Joint IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*. ISSN: 2161-9484, pp. 166–171. DOI: `10.1109/DEVLRN.2016.7846812`.

Becker, Alexander and Daniel Görlich (2020). “What is Game Balancing? - An Examination of Concepts”. In: *ParadigmPlus* 1, pp. 22–41. DOI: `10.55969/paradigmplus.v1n1a2`.

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling (2013). “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47, pp. 253–279. ISSN: 1076-9757. DOI: `10.1613/jair.3912`. arXiv: `1207.4708`.

Bellemare, Marc G., Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos (Nov. 2016). “Unifying Count-Based Exploration and Intrinsic Motivation”. In: arXiv:1606.01868 [cs, stat]. DOI: `10.48550/arXiv.1606.01868`. URL: `http://arxiv.org/abs/1606.01868` (visited on 09/19/2022).

Bellemare, Marc G., Joel Veness, and Michael Bowling (2012). “Investigating contingency awareness using Atari 2600 games”. In: *Proceedings of the National Conference on Artificial Intelligence*. Vol. 2, pp. 864–871. ISBN: 9781577355687.

Bellman, Richard (1954). “The theory of dynamic programming”. en. In: *Bulletin of the American Mathematical Society* 60.6, pp. 503–515. ISSN: 0273-0979, 1088-9485. DOI: `10.1090/S0002-9904-1954-09848-8`. URL: `https://www.ams.org/bull/1954-60-06/S0002-9904-1954-09848-8/`.

Benedict Wilkins (n.d.). *World of Bugs*. URL: `https://benedictwilkins.github.io/world-of-bugs/` (visited on 11/29/2022).

Bengio, Emmanuel, Valentin Thomas, Joelle Pineau, Doina Precup, and Yoshua Bengio (2017). “Independently Controllable Features”. In: *arXiv*. URL: `http://arxiv.org/abs/1703.07718`.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (Apr. 2014). “Representation Learning: A Review and New Perspectives”. en. In: *arXiv*. URL: `http://arxiv.org/abs/1206.5538`.

Bengio, Yoshua, Li Yao, Guillaume Alain, and Pascal Vincent (2013). “Generalized Denoising Auto-Encoders as Generative Models”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’13. Lake Tahoe, Nevada: Curran Associates Inc., 899–907.

Bergdahl, Joakim, Camilo Gordillo, Konrad Tollmar, and Linus Gisslen (2020). “Augmenting Automated Game Testing with Deep Reinforcement Learning”. In: *IEEE Conference on Computatonal Intelligence and Games, CIG*. Vol. 2020-Augus. arXiv: 2103.15819 ISSN: 23254289. IEEE Computer Society, pp. 600–603. ISBN: 978-1-72814-533-4. DOI: `10.1109/CoG47356.2020.9231552`. (Visited on 09/30/2021).

Berner, Christopher, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang (2019). "Dota 2 with Large Scale Deep Reinforcement Learning". In: *arXiv*. URL: `http://arxiv.org/abs/1912.06680`.

Bissyandé, Tegawendé F., David Lo, Lingxiao Jiang, Laurent Réveillère, Jacques Klein, and Yves Le Traon (Nov. 2013). "Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. ISSN: 2332-6549, pp. 188–197. DOI: `10.1109/ISSRE.2013.6698918`.

Blakemore, S. J., D. Wolpert, and C. Frith (Aug. 2000). "Why can't you tickle yourself?" eng. In: *Neuroreport* 11.11, R11–16. ISSN: 0959-4965. DOI: `10.1097/00001756-200008030-00002`.

Blakemore, Sarah Jayne and Chris Frith (2005). "The role of motor contagion in the prediction of action". In: *Neuropsychologia*. Vol. 43. 2 SPEC. ISS. Elsevier Ltd, pp. 260–267. DOI: `10.1016/j.neuropsychologia.2004.11.012`.

Bogdoll, Daniel, Maximilian Nitsche, and J. Marius Zollner (June 2022). "Anomaly Detection in Autonomous Driving: A Survey". en. In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. New Orleans, LA, USA: IEEE, pp. 4487–4498. ISBN: 978-1-66548-739-9. DOI: `10.1109/CVPRW56347.2022.00495`. URL: `https://ieeexplore.ieee.org/document/9857500/` (visited on 10/10/2022).

Boller, Sharon (July 2013). *Learning Game Design: Game Mechanics*. URL: `http://www.theknowledgeguru.com/learning-game-design-mechanics/` (visited on 09/26/2022).

Brando, Axel, Jose A. Rodríguez-Serrano, Mauricio Ciprian, Roberto Maestre, and Jordi Vitrià (July 2018). "Uncertainty Modelling in Deep Networks: Forecasting Short and Noisy Series". en. In: *arXiv*. URL: `http://arxiv.org/abs/1807.09011` (visited on 02/20/2023).

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba (2016). "OpenAI Gym". In: *CoRR* abs/1606.01540. arXiv: `1606.01540`. URL: `http://arxiv.org/abs/1606.01540`.

Buesing, Lars, Theophane Weber, Yori Zwols, Sebastien Racaniere, Arthur Guez, Jean-Baptiste Lespiau, and Nicolas Heess (Nov. 2018). "Woulda, Coulda, Shoulda: Counterfactually-Guided Policy Search". en. In: *arXiv*. URL: `http://arxiv.org/abs/1811.06272` (visited on 07/22/2022).

bug (n.d.). en. URL: https://dictionary.cambridge.org/dictionary/english/bug (visited on 09/21/2022).

Buhl, Christian and Fazeel Gareeboo (2012). “Automated testing: a key factor for success in video game development. Case study and lessons learned.” en. In: *Proceedings of Pacific NW Software Quality Conferences*, pp. 1–15.

Burda, Yuri, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros (Aug. 2019). “Large-scale study of curiosity-driven learning”. In: *7th International Conference on Learning Representations, ICLR 2019*. arXiv: 1808.04355. URL: http://arxiv.org/abs/1808.04355 (visited on 08/15/2019).

Campbell, Murray, A. Joseph Hoane, and Feng-hsiung Hsu (Jan. 2002). “Deep Blue”. en. In: *Artificial Intelligence* 134.1, pp. 57–83. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291 (visited on 06/24/2022).

Castro, Pablo Samuel, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare (2018). “Dopamine: A Research Framework for Deep Reinforcement Learning”. In: *arXiv*. URL: http://arxiv.org/abs/1812.06110.

Chalapathy, Raghavendra and Sanjay Chawla (Jan. 2019). “Deep Learning for Anomaly Detection: A Survey”. In: arXiv: 1901.03407. URL: http://arxiv.org/abs/1901.03407 (visited on 07/15/2019).

Chandola, Varun, Arindam Banerjee, and Vipin Kumar (July 2009). “Anomaly Detection: A Survey”. In: *ACM Computing Surveys* 41.3, pp. 1–58. ISSN: 03600300. DOI: 10.1145/1541880.1541882. URL: http://portal.acm.org/citation.cfm?doid=1541880.1541882 (visited on 08/12/2019).

Chang, Kenneth, Batu Aytemiz, and Adam M. Smith (2019). “Reveal-More: Amplifying Human Effort in Quality Assurance Testing Using Automated Exploration”. en. In: *2019 IEEE Conference on Games (CoG)*. London, United Kingdom: IEEE. ISBN: 978-1-72811-884-0. DOI: 10.1109/CIG.2019.8848091. URL: https://ieeexplore.ieee.org/document/8848091/ (visited on 09/19/2022).

Chang, Yunpeng, Zhigang Tu, Wei Xie, Bin Luo, Shifu Zhang, Haigang Sui, and Junsong Yuan (Feb. 2022). “Video anomaly detection with spatio-temporal dissociation”. en. In: *Pattern Recognition* 122, p. 108213. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2021.108213. URL: https://www.sciencedirect.com/science/article/pii/S0031320321003940 (visited on 12/12/2022).

Chen, Ke, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang (Aug. 2021a). “GLIB: towards automated test oracle for graphically-rich applications”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, pp. 1093–1104. ISBN: 978-

1-4503-8562-6. DOI: 10.1145/3468264.3468586. URL: https://doi.org/10.1145/3468264.3468586 (visited on 06/11/2022).

Chen, Lin-Kung, Yen-Hung Chen, Shu-Fang Chang, and Shun-Chieh Chang (Sept. 2020a). “A Long/Short-Term Memory Based Automated Testing Model to Quantitatively Evaluate Game Design”. en. In: *Applied Sciences* 10.19, p. 6704. ISSN: 2076-3417. DOI: 10.3390/app10196704. URL: https://www.mdpi.com/2076-3417/10/19/6704 (visited on 06/11/2022).

Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba (July 2021b). “Evaluating Large Language Models Trained on Code”. In: *arXiv*. URL: http://arxiv.org/abs/2107.03374.

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton (June 2020b). “A Simple Framework for Contrastive Learning of Visual Representations”. en. In: *arXiv*. URL: http://arxiv.org/abs/2002.05709 (visited on 12/12/2022).

Chen, Tsong Yueh, S. C. Cheung, and Siu-Ming Yiu (2020c). “Metamorphic Testing: A New Approach for Generating Next Test Cases”. In: URL: https://arxiv.org/abs/2002.12543.

Chen, Xinlei and Kaiming He (Nov. 2020d). “Exploring Simple Siamese Representation Learning”. en. In: *arXiv*. URL: http://arxiv.org/abs/2011.10566 (visited on 12/08/2022).

Choi, Jongwook, Yijie Guo, Marcin Moczulski, Junhyuk Oh, Neal Wu, Mohammad Norouzi, and Honglak Lee (2019). “Contingency-aware exploration in reinforcement learning”. In: *7th International Conference on Learning Representations, ICLR 2019*. International Conference on Learning Representations, ICLR. arXiv: 1811.01483. URL: http://arxiv.org/abs/1811.01483.

Corcoll, Oriol and Raul Vicente (2020). “Disentangling causal effects for hierarchical reinforcement learning”. In: *arXiv*. URL: http://arxiv.org/abs/2010.01351.

Crispin, Lisa and Janet Gregory (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. 1st ed. Addison-Wesley Professional. ISBN: 0-321-53446-8.

Cullen, Kathleen E (Dec. 2004). "Sensory signals during active versus passive movement". en. In: *Current Opinion in Neurobiology* 14.6, pp. 698–706. ISSN: 0959-4388. DOI: 10.1016/j.conb.2004.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0959438804001540 (visited on 07/11/2022).

De Lange, Matthias, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars (2021). "A continual learning survey: Defying forgetting in classification tasks". en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1. ISSN: 0162-8828, 2160-9292, 1939-3539. (Visited on 03/04/2023).

Delaurentis, Daniel, Jitesh Panchal, Ali Raz, Prajwal Balasubramani, Apoorv Maheshwari, Adam Dachowicz, and Kshitij Mall (Aug. 2021). "Toward Automated Game Balance: A Systematic Engineering Design Approach". In: *2021 IEEE Conference on Games (CoG)*. DOI: 10.1109/CoG52621.2021.9619032.

Dhasade, Akash Balasaheb, Akhila Sri Manasa Venigalla, and Sridhar Chimalakonda (Feb. 2020). "Towards Prioritizing GitHub Issues". In: *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference*. ISEC 2020. New York, NY, USA: Association for Computing Machinery, pp. 1–5. ISBN: 978-1-4503-7594-8. DOI: 10.1145/3385032.3385052. URL: https://doi.org/10.1145/3385032.3385052 (visited on 10/07/2022).

Di Mattia, Federico, Paolo Galeone, Michele De Simoni, and Emanuele Ghelfi (June 2019). "A Survey on GANs for Anomaly Detection". In: *arXiv*. URL: http://arxiv.org/abs/1906.11632 (visited on 10/27/2019).

Difference-in-Difference Estimation | Columbia Public Health (n.d.). URL: https://www.publichealth.columbia.edu/research/population-health-methods/difference-difference-estimation (visited on 03/02/2023).

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby (June 2021). "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *arXiv*. URL: http://arxiv.org/abs/2010.11929 (visited on 01/16/2023).

Fatta, Hanif Al, Zulisman Maksom, and Mohd Hafiz Zakaria (Nov. 2019). "Fun, Play and Challenge Factors for Playability Measurement of Game-Based Learning". In: *International Journal of Advanced Trends in*

Computer Science and Engineering 8, pp. 305–316. ISSN: 22783091. DOI: 10.30534/ijatcse/2019/5281.52019.

Feng, Sidong and Chunyang Chen (May 2022). “GIFdroid: automated replay of visual bug reports for Android apps”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. New York, NY, USA: Association for Computing Machinery, pp. 1045–1057. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510048. URL: https://doi.org/10.1145/3510003.3510048 (visited on 10/07/2022).

Ferdous, Raihana, Fitsum Kifetew, Davide Prandi, I. S. W. B. Prasetya, Samira Shirzadehhajimahmood, and Angelo Susi (2021). “Search-Based Automated Play Testing of Computer Games: A Model-Based Approach”. en. In: *Search-Based Software Engineering*. Ed. by Una-May O’Reilly and Xavier Devroey. Vol. 12914. Series Title: Lecture Notes in Computer Science. Springer International Publishing, pp. 56–71. ISBN: 978-3-030-88105-4 978-3-030-88106-1. DOI: 10.1007/978-3-030-88106-1_5. URL: https://link.springer.com/10.1007/978-3-030-88106-1_5 (visited on 01/04/2023).

Finn, Chelsea, Sergey Levine, and Pieter Abbeel (Mar. 2016). “Guided Cost Learning: Deep Inverse Optimal Control Via Policy Optimization”. In: *33rd International Conference on Machine Learning, ICML 2016* 1. arXiv: 1603.00448 Publisher: International Machine Learning Society (IMLS), pp. 95–107. URL: http://arxiv.org/abs/1603.00448.

Florian, Razvan V (2007). *Correct equations for the dynamics of the cart-pole system.*

Fu, Chen, Yuxin Deng, David N. Jansen, and Lijun Zhang (2017). “On Equivalence Checking of Nondeterministic Finite Automata”. en. In: *Dependable Software Engineering. Theories, Tools, and Applications.* Ed. by Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang. Vol. 10606. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 216–231. ISBN: 978-3-319-69482-5 978-3-319-69483-2. DOI: 10.1007/978-3-319-69483-2_13. URL: http://link.springer.com/10.1007/978-3-319-69483-2_13 (visited on 10/31/2022).

Fuchs, Florian, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Duerr (Aug. 2020). “Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning”. In: *arXiv*. arXiv: 2008.07971 Publisher: arXiv. URL: http://arxiv.org/abs/2008.07971 (visited on 01/18/2021).

Fukutomi, Matasaburo and Bruce A. Carlson (2020). “A History of Corollary Discharge: Contributions of Mormyrid Weakly Electric Fish”. In: *Frontiers in Integrative Neuroscience* 14. ISSN: 16625145. DOI: 10.3389/fnint.2020.00042.

Gabriel, Iason (Sept. 2020). “Artificial Intelligence, Values, and Alignment”. en. In: *Minds and Machines* 30.3, pp. 411–437. ISSN: 1572-8641. DOI: 10.1007/s11023-020-09539-2.

Gallese, Vittorio and Alvin Goldman (Dec. 1998). "Mirror neurons and the simulation theory of mind-reading". en. In: *Trends in Cognitive Sciences* 2.12, pp. 493–501. ISSN: 1364-6613. DOI: `10.1016/S1364-6613(98)01262-5`. URL: `https://www.sciencedirect.com/science/article/pii/S1364661398012625` (visited on 08/13/2022).

García-Sánchez, Pablo, Alberto Tonda, Antonio M. Mora, Giovanni Squillero, and Juan Julián Merelo (Aug. 2018). "Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone". en. In: *Knowledge-Based Systems* 153, pp. 133–146. ISSN: 09507051. DOI: `10.1016/j.knosys.2018.04.030`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0950705118301953` (visited on 06/11/2022).

Geirhos, Robert, Jorn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A. Wichmann (Nov. 2020). "Shortcut learning in deep neural networks". en. In: *Nature Machine Intelligence* 2, pp. 665–673. ISSN: 2522-5839. DOI: `10.1038/s42256-020-00257-z`. URL: `https://www.nature.com/articles/s42256-020-00257-z` (visited on 07/12/2022).

Georgios N. Yannakakis and Julian Togelius (2018). *Artificial Intelligence and Games*. en. Vol. 2. Springer.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014). "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Vol. 27. Curran Associates, Inc.

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy (2015). "Explaining and Harnessing Adversarial Examples". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: `http://arxiv.org/abs/1412.6572`.

Gordillo, Camilo, Joakim Bergdahl, Konrad Tollmar, and Linus Gisslén (2021). "Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents". In: *2021 IEEE Conference on Games (CoG)*. DOI: `10.1109/cog52621.2021.9619048`.

Gravina, Daniele, Antonios Liapis, and Georgios Yannakakis (July 2016). "Surprise Search: Beyond Objectives and Novelty". en. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. Denver Colorado USA: ACM, pp. 677–684. ISBN: 978-1-4503-4206-3. DOI: `10.1145/2908812.2908817`.

Gudmundsson, Stefan Freyr, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling, and Lele Cao (2018). "Human-Like Playtesting with Deep Learning". In: *IEEE Conference on Computatonal Intelligence and Games, CIG*. Vol. 2018-Augus. ISSN: 23254289. IEEE Computer Society. ISBN: 978-1-5386-4359-4. DOI: `10.1109/CIG.2018.8490442`. (Visited on 09/30/2021).

Hafner, Danijar, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba (Feb. 2022). “Mastering Atari with Discrete World Models”. In: arXiv:2010.02193 [cs, stat]. DOI: `10.48550/arXiv.2010.02193`. URL: `http://arxiv.org/abs/2010.02193` (visited on 01/09/2023).

Haggard, Patrick (Apr. 2017). “Sense of agency in the human brain”. en. In: *Nature Reviews Neuroscience* 18.4. Number: 4 Publisher: Nature Publishing Group, pp. 196–207. ISSN: 1471-0048. DOI: `10.1038/nrn.2017.14`. URL: `https://www.nature.com/articles/nrn.2017.14` (visited on 07/23/2022).

Hasan, Mahmudul, Jonghyun Choi, Jan Neumann, Amit K. Roy-Chowdhury, and Larry S. Davis (June 2016). “Learning Temporal Regularity in Video Sequences”. en. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, pp. 733–742. ISBN: 978-1-4673-8851-1. DOI: `10.1109/CVPR.2016.86`. URL: `http://ieeexplore.ieee.org/document/7780455/` (visited on 12/12/2022).

Hazan, Elad, Sham M. Kakade, Karan Singh, and Abby Van Soest (Dec. 2018). “Provably Efficient Maximum Entropy Exploration”. In: *arXiv*. URL: `http://arxiv.org/abs/1812.02690` (visited on 11/05/2019).

Helmholtz, Hermann von and James PC Southall (1924). *Helmholtz’s treatise on physiological optics*. Vol. 1. Optical Society of America. URL: `https://doi.org/10.1037/13536-000`.

Hernández Bécares, Jennifer, Luis Costero Valero, and Pedro Pablo Gómez Martín (Jan. 2017). “An approach to automated videogame beta testing”. en. In: *Entertainment Computing* 18, pp. 79–92. ISSN: 1875-9521. DOI: `10.1016/j.entcom.2016.08.002`. URL: `https://www.sciencedirect.com/science/article/pii/S1875952116300234` (visited on 10/14/2022).

Hill, Ashley, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu (2018). *Stable Baselines*. URL: `\url{https://github.com/hill-a/stable-baselines}`.

Ho, Jonathan and Stefano Ermon (2016). “Generative Adversarial Imitation Learning”. In: *Advances in Neural Information Processing Systems*. arXiv: 1606.03476 ISSN: 10495258, pp. 4572–4580. URL: `https://arxiv.org/abs/1606.03476v1`.

Hojjati, Hadi, Thi Kieu Khanh Ho, and Narges Armanfard (May 2022). “Self-Supervised Anomaly Detection: A Survey and Outlook”. en. In: arXiv:2205.05173 [cs]. URL: `http://arxiv.org/abs/2205.05173` (visited on 12/14/2022).

Holmgård, Christoffer, Michael Cerny Green, Antonios Liapis, and Julian Togelius (Feb. 2018). "Automated Playtesting with Procedural Personas through MCTS with Evolved Heuristics". en. In: arXiv:1802.06881 [cs]. URL: http://arxiv.org/abs/1802.06881 (visited on 09/19/2022).

Holst, E. von (1954). "Relations between the central Nervous System and the peripheral organs". In: *The British Journal of Animal Behaviour* 2.3, pp. 89–94. ISSN: 09505601. DOI: 10.1016/S0950-5601(54)80044-X.

Ibrahimzada, Ali Reza, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand (2022). "Perfect is the enemy of test oracle". en. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore Singapore: ACM, pp. 70–81. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549086.

IEEE Standard Classification for Software Anomalies (2010). In: *IEEE Std 1044-2009*, pp. 1–25. DOI: 10.1109/IEEESTD.2010.5399061.

IEEE Standard Glossary of Software Engineering Terminology (Dec. 1990). In: *IEEE Std 610.12-1990*, pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

Iftikhar, Sidra, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood (Sept. 2015). "An automated model based testing approach for platform games". en. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ottawa, ON, Canada: IEEE, pp. 426–435. ISBN: 978-1-4673-6908-4. DOI: 10.1109/MODELS.2015.7338274. URL: http://ieeexplore.ieee.org/document/7338274/ (visited on 01/04/2023).

Imran, Mia Mohammad, Agnieszka Ciborowska, and Kostadin Damevski (May 2021). "Automatically Selecting Follow-up Questions for Deficient Bug Reports". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. ISSN: 2574-3864, pp. 167–178. DOI: 10.1109/MSR52588.2021.00029.

Isaksen, Aaron, Dan Gopstein, Julian Togelius, and Andy Nealen (June 2018). "Exploring Game Space of Minimal Action Games via Parameter Tuning and Survival Analysis". en. In: *IEEE Transactions on Games* 10.2, pp. 182–194. ISSN: 2475-1502, 2475-1510. DOI: 10.1109/TCIAIG.2017.2750181. URL: https://ieeexplore.ieee.org/document/8030128/ (visited on 06/11/2022).

Izadi, Maliheh, Kiana Akbari, and Abbas Heydarnoori (Mar. 2022). "Predicting the objective and priority of issue reports in software repositories". en. In: *Empirical Software Engineering* 27.2, p. 50. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-021-10085-3. URL: https://link.springer.com/10.1007/s10664-021-10085-3 (visited on 10/07/2022).

Jékely, Gáspár, Peter Godfrey-Smith, and Fred Keijzer (2021). "Reafference and the origin of the self in early nervous system evolution". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 376.1821. ISSN: 14712970. DOI: 10.1098/rstb.2019.0764. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rstb.2019.0764.

Judah, Kshitij, Alan Fern, Prasad Tadepalli, and Robby Goetschalckx (June 2014). "Imitation Learning with Demonstrations and Shaping Rewards". en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 28.1. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v28i1.9024. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9024 (visited on 10/17/2022).

Judea Pearl (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press. ISBN: 978-0-521-77362-1.

Juliani, Arthur, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange (2018). "Unity: A General Platform for Intelligent Agents". In: *arXiv*. ISSN: 2331-8422.

Kallis, Rafael, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella (Sept. 2019). "Ticket Tagger: Machine Learning Driven Issue Classification". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. ISSN: 2576-3148, pp. 406–409. DOI: 10.1109/ICSME.2019.00070.

Kasurinen, Jussi and Kari Smolander (2014). "What do game developers test in their products?" In: *International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, pp. 1–10. ISBN: 9781450327749. DOI: 10.1145/2652524.2652525. URL: http://dl.acm.org/citation.cfm?doid=2652524.2652525.

Kawato, Mitsuo (1999). "Internal models for motor control and trajectory planning". In: *Current Opinion in Neurobiology*. Vol. 9. 6. Current Biology Ltd, pp. 718–727. DOI: 10.1016/S0959-4388(99)00028-8. URL: https://pubmed.ncbi.nlm.nih.gov/10607637/.

Kaya, Mahmut and Hasan Şakir Bi lge (Sept. 2019). "Deep Metric Learning: A Survey". en. In: *Symmetry* 11.9. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute, p. 1066. ISSN: 2073-8994. DOI: 10.3390/sym11091066. URL: https://www.mdpi.com/2073-8994/11/9/1066 (visited on 11/08/2022).

Keehl, Oleksandra and Adam M. Smith (Aug. 2018). "Monster Carlo: An MCTS-based Framework for Machine Playtesting Unity Games". en. In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. Maastricht: IEEE, pp. 1–8. ISBN: 978-1-5386-4359-4. DOI: 10.1109/CIG.2018.8490363. URL: https://ieeexplore.ieee.org/document/8490363/ (visited on 06/11/2022).

Khaliq, Zubair, Sheikh Umar Farooq, and Dawood Ashraf Khan (2022). “Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect”. en. In: URL: `http://arxiv.org/abs/2201.05371` (visited on 01/04/2023).

Kim, Dahun, Donghyeon Cho, and In So Kweon (Nov. 2018). “Self-Supervised Video Representation Learning with Space-Time Cubic Puzzles”. en. In: arXiv:1811.09795 [cs]. URL: `http://arxiv.org/abs/1811.09795` (visited on 11/08/2022).

Kingma, Diederik P. and Max Welling (2014). “Auto-encoding variational bayes”. In: *2nd International Conference on Learning Representations, ICLR 2014*. (Visited on 01/14/2020).

Kiran, B. Ravi, Dilip Mathew Thomas, and Ranjith Parakkal (2018). “An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos”. In: 4.2, p. 36. ISSN: 2313433X. DOI: `10.3390/jimaging4020036`. arXiv: `1801.03149`. URL: `http://www.mdpi.com/2313-433X/4/2/36`.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (May 2017). “ImageNet classification with deep convolutional neural networks”. en. In: *Communications of the ACM* 60.6, pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: `10.1145/3065386`. URL: `https://dl.acm.org/doi/10.1145/3065386` (visited on 02/26/2023).

Kumari, Pratibha, Anterpreet Kaur Bedi, and Mukesh Saini (Apr. 2022). “Multimedia Datasets for Anomaly Detection: A Review”. en. In: *arXiv*. URL: `http://arxiv.org/abs/2112.05410` (visited on 09/08/2022).

Lapuschkin, Sebastian, Stephan Wäldchen, Alexander Binder, Grégoire Montavon, Wojciech Samek, and Klaus Robert Müller (2019). “Unmasking Clever Hans predictors and assessing what machines really learn”. In: *Nature Communications* 10.1. ISSN: 20411723. DOI: `10.1038/s41467-019-08987-4`. arXiv: `1902.10178`. URL: `http://arxiv.org/abs/1902.10178http://dx.doi.org/10.1038/s41467-019-08987-4`.

Lavalle, Steven M. (1998). “Rapidly-Exploring Random Trees: A New Tool for Path Planning”. In: *The annual research report*.

Le-Khac, Phuc H., Graham Healy, and Alan F. Smeaton (2020). “Contrastive Representation Learning: A Framework and Review”. en. In: *IEEE Access* 8. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2020.3031549`.

Lee, Sangmin, Hak Gu Kim, and Yong Man Ro (Apr. 2018). “Stan: Spatio- temporal adversarial networks for abnormal event detection”. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. Vol. 2018-April. arXiv: 1804.08381 ISSN: 15206149. IEEE, pp. 1323–1327. ISBN: 978-1-5386-4658-8. DOI: `10.1109/ICASSP.2018.8462388`. URL: `https://ieeexplore.ieee.org/document/8462388/` (visited on 11/05/2019).

Lehman, Joel, Jeff Clune, Dusan Misevic, Christoph Adami, Lee Altenberg, Julie Beaulieu, Peter J. Bentley, Samuel Bernard, Guillaume Beslon, David M. Bryson, Patryk Chrabaszcz, Nick Cheney, Antoine Cully, Stephane Doncieux, Fred C. Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Frénoy, Christian Gagné, Leni Le Goff, Laura M. Grabowski, Babak Hodjat, Frank Hutter, Laurent Keller, Carole Knibbe, Peter Krcah, Richard E. Lenski, Hod Lipson, Robert MacCurdy, Carlos Maestre, Risto Miikkulainen, Sara Mitri, David E. Moriarty, Jean-Baptiste Mouret, Anh Nguyen, Charles Ofria, Marc Parizeau, David Parsons, Robert T. Pennock, William F. Punch, Thomas S. Ray, Marc Schoenauer, Eric Shulte, Karl Sims, Kenneth O. Stanley, François Taddei, Danesh Tarapore, Simon Thibault, Westley Weimer, Richard Watson, and Jason Yosinski (Nov. 2019). “The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities”. en. In: arXiv:1803.03453 [cs]. URL: http://arxiv.org/abs/1803.03453 (visited on 10/19/2022).

Lehman, Joel and Kenneth O. Stanley (June 2011). “Abandoning Objectives: Evolution Through the Search for Novelty Alone”. en. In: *Evolutionary Computation* 19.2, pp. 189–223. ISSN: 1063-6560, 1530-9304. DOI: 10.1162/EVCO_a_00025. URL: https://direct.mit.edu/evco/article/19/2/189-223/1365.

Levy, L. and J. Novak (2010). *Game Development Essentials: Game QA & Testing*. Delmar/Cengage Learning. ISBN: 9781435439474.

Lewis, Chris, Jim Whitehead, and Noah Wardrip-Fruin (2010). “What went wrong: A taxonomy of video game bugs”. In: *FDG 2010 - Proceedings of the 5th International Conference on the Foundations of Digital Games*. ACM Press, pp. 108–115. ISBN: 9781605589374. DOI: 10.1145/1822348.1822363.

Lewis, Michael (2012). *Social cognition and the acquisition of self*. Springer Science & Business Media. URL: https://doi.org/10.1007/978-1-4684-3566-5.

Li, Chun-Liang, Kihyuk Sohn, Jinsung Yoon, and Tomas Pfister (June 2021). “CutPaste: Self-Supervised Learning for Anomaly Detection and Localization”. en. In: *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Nashville, TN, USA: IEEE, pp. 9659–9669. ISBN: 978-1-66544-509-2. DOI: 10.1109/CVPR46437.2021.00954. URL: https://ieeexplore.ieee.org/document/9578875/ (visited on 12/14/2022).

Li, Ning, Martin Shepperd, and Yuchen Guo (2020). “A systematic review of unsupervised learning techniques for software defect prediction”. en. In: *Information and Software Technology* 122, p. 106287. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2020.106287. (Visited on 11/02/2022).

Li, Weixin, Vijay Mahadevan, and Nuno Vasconcelos (2014). “Anomaly detection and localization in crowded scenes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.1, pp. 18–32. ISSN: 01628828. DOI: `10.1109/TPAMI.2013.111`.

Liaqat, Aisha and Muddassar Azam Sindhu (Nov. 2018). “A Metamorphic Relation based Approach for Testing a Chess Game”. en. In: *2018 14th International Conference on Emerging Technologies (ICET)*. Islamabad: IEEE, pp. 1–6. ISBN: 978-1-5386-8143-5. DOI: `10.1109/ICET.2018.8603603`. URL: `https://ieeexplore.ieee.org/document/8603603/` (visited on 01/04/2023).

Lin, Long-Ji (May 1992). “Self-improving reactive agents based on reinforcement learning, planning and teaching”. en. In: *Machine Learning* 8.3, pp. 293–321. ISSN: 1573-0565. DOI: `10.1007/BF00992699`. URL: `https://doi.org/10.1007/BF00992699` (visited on 07/02/2022).

Ling, Carlos, Konrad Tollmar, and Linus Gisslén (Oct. 2020). “Using Deep Convolutional Neural Networks to Detect Rendered Glitches in Video Games”. en. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 16.1. Number: 1, pp. 66–73. ISSN: 2334-0924. DOI: `10.1609/aiide.v16i1.7409`.

Liu, Wen, Weixin Luo, Dongze Lian, and Shenghua Gao (Mar. 2018). “Future Frame Prediction for Anomaly Detection – A New Baseline”. In: arXiv:1712.09867 [cs]. DOI: `10.48550/arXiv.1712.09867`. URL: `http://arxiv.org/abs/1712.09867` (visited on 12/12/2022).

Liu, Zhe, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang (Dec. 2020). “Owl eyes: spotting UI display issues via visual understanding”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’20. New York, NY, USA: Association for Computing Machinery, pp. 398–409. ISBN: 978-1-4503-6768-4. DOI: `10.1145/3324884.3416547`. URL: `https://doi.org/10.1145/3324884.3416547` (visited on 06/11/2022).

Locatello, Francesco, Stefan Bauer, Mario Lucic, Gunnar Raetsch, Sylvain Gelly, Bernhard Schölkopf, and Olivier Bachem (2019). “Challenging common assumptions in the unsupervised learning of disentangled representations”. In: *International Conference on Machine Learning*.

Mahadevan, Vijay, Weixin Li, Viral Bhalodia, and Nuno Vasconcelos (June 2010). “Anomaly detection in crowded scenes”. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. ISSN: 1063-6919, pp. 1975–1981. DOI: `10.1109/CVPR.2010.5539872`.

Makantasis, Konstantinos, Antonios Liapis, and Georgios N. Yannakakis (Jan. 2023). “The Pixels and Sounds of Emotion: General-Purpose Representations of Arousal in Games”. In: *IEEE Transactions on Affective Computing* 14.1, pp. 680–693. ISSN: 1949-3045. DOI: `10.1109/TAFFC.2021.3060877`.

Makhzani, Alireza, Jonathon Shlens, Navdeep Jaitly, and Ian Goodfellow (2016). “Adversarial Autoencoders”. In: *International Conference on Learning Representations*. URL: http://arxiv.org/abs/1511.05644.

Mao, Ke, Mark Harman, and Yue Jia (July 2016). “Sapienz: multi-objective automated testing for Android applications”. en. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. Saarbrücken Germany: ACM, pp. 94–105. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931054. URL: https://dl.acm.org/doi/10.1145/2931037.2931054 (visited on 09/22/2022).

Markou, Markos and Sameer Singh (Dec. 2003). “Novelty Detection: A Review—Part 1: Statistical Approaches”. en. In: *Signal Processing* 83.12, pp. 2481–2497. ISSN: 0165-1684. DOI: 10.1016/j.sigpro.2003.07.018. URL: https://www.sciencedirect.com/science/article/pii/S0165168403002020 (visited on 10/04/2022).

Marselis, Rik and Bert Linker (2014). “Organize Your Testing Using Test Varieties and Coverage Types”. en. In: *The Magazine for Professional Testers*.

Masana, Marc, Idoia Ruiz, Joan Serrat, Joost Van De Weijer, and Antonio M. Lopez (2019). “Metric learning for novelty and anomaly detection”. In: *British Machine Vision Conference 2018, BMVC 2018*. arXiv: 1808.05492.

Maschler, Benjamin and Michael Weyrich (June 2021). “Deep Transfer Learning for Industrial Automation: A Review and Discussion of New Techniques for Data-Driven Machine Learning”. In: *IEEE Industrial Electronics Magazine* 15.2, pp. 65–75. ISSN: 1941-0115. DOI: 10.1109/MIE.2020.3034884.

Medel, Jefferson Ryan and Andreas Savakis (2016). “Anomaly Detection in Video Using Predictive Convolutional Long Short-Term Memory Networks”. In: *arXiv*. URL: http://arxiv.org/abs/1612.00390.

Medendorp, W. Pieter (2011). “Spatial constancy mechanisms in motor control”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 366, pp. 476–491. ISSN: 14712970. DOI: 10.1098/rstb.2010.0089.

Melhart, David, Ahmad Azadvar, Alessandro Canossa, Antonios Liapis, and Georgios N. Yannakakis (Aug. 2019). “Your Gameplay Says It All: Modelling Motivation in Tom Clancy’s The Division”. In: *2019 IEEE Conference on Games (CoG)*. ISSN: 2325-4289, pp. 1–8. DOI: 10.1109/CIG.2019.8848123.

Mesnard, Thomas, Théophane Weber, Fabio Viola, Shantanu Thakoor, Alaa Saade, Anna Harutyunyan, Will Dabney, Tom Stepleton, Nicolas Heess, Arthur Guez, Éric Moulines, Marcus Hutter, Lars Buesing, and Rémi Munos (Dec. 2021). “Counterfactual Credit Assignment in Model-Free Reinforcement Learning”. en. In: *arXiv*. URL: http://arxiv.org/abs/2011.09464 (visited on 07/22/2022).

Miall, R. C. and D. M. Wolpert (Nov. 1996). “Forward Models for Physiological Motor Control”. en. In: *Neural Networks*. Four Major Hypotheses in Neuroscience 9.8, pp. 1265–1279. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(96)00035-4.

Miller, Barton P., Lars Fredriksen, and Bryan So (Dec. 1990). “An empirical study of the reliability of UNIX utilities”. In: *Communications of the ACM* 33.12, pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279 (visited on 09/20/2022).

Mirarab, Siavash and Ladan Tahvildari (2007). “A Prioritization Approach for Software Test Cases Based on Bayesian Networks”. en. In: *Fundamental Approaches to Software Engineering*. Ed. by Matthew B. Dwyer and Antónia Lopes. Vol. 4422. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 276–290. ISBN: 978-3-540-71288-6 978-3-540-71289-3. DOI: 10.1007/978-3-540-71289-3_22. URL: http://link.springer.com/10.1007/978-3-540-71289-3_22 (visited on 12/07/2022).

Mitchell, Tom M. (1997). *Machine Learning*. en. McGraw-Hill series in computer science. New York: McGraw-Hill. ISBN: 978-0-07-042807-2.

Mnih, Volodymyr, Adria Puigdomenech Badia, Lehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu (Feb. 2016). “Asynchronous methods for deep reinforcement learning”. In: *33rd International Conference on Machine Learning, ICML 2016*. Vol. 4. arXiv: 1602.01783. International Machine Learning Society (IMLS), pp. 2850–2869. ISBN: 978-1-5108-2900-8. URL: http://arxiv.org/abs/1602.01783 (visited on 04/13/2021).

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013). “Playing atari with deep reinforcement learning”. In: *arXiv*.

Mozgovoy, Maxim and Evgeny Pyshkin (2018). “Unity Application Testing Automation with Appium and Image Recognition”. en. In: *Tools and Methods of Program Analysis*. Vol. 779. Cham: Springer International Publishing, pp. 139–150. ISBN: 978-3-319-71733-3 978-3-319-71734-0. DOI: 10.1007/978-3-319-71734-0_12. URL: http://link.springer.com/10.1007/978-3-319-71734-0_12 (visited on 10/14/2022).

Mugrai, Luvneesh, Fernando Silva, Christoffer Holmgard, and Julian Togelius (Aug. 2019). “Automated Playtesting of Matching Tile Games”. en. In: *2019 IEEE Conference on Games (CoG)*. London, United Kingdom: IEEE, pp. 1–7. ISBN: 978-1-72811-884-0. DOI: 10.1109/CIG.2019.8848057. URL: https://ieeexplore.ieee.org/document/8848057/ (visited on 06/11/2022).

Mutti, Mirco, Lorenzo Pratissoli, and Marcello Restelli (May 2021). “Task-Agnostic Exploration via Policy Gradient of a Non-Parametric State Entropy Estimate”. en. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.10. Number: 10, pp. 9028–9036. ISSN: 2374-3468. DOI: 10.1609/aaai.v35i10.17091. URL: https://ojs.aaai.org/index.php/AAAI/article/view/17091 (visited on 02/10/2023).

Nantes, Alfredo, Ross Brown, and Frederic Maire (2008). “A framework for the semi-Automatic testing of video games”. In: *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2008*, pp. 197–202. ISBN: 978-1-57735-391-1. URL: http://glintercept.nutty.org/index.html/ (visited on 01/21/2020).

— (2013). “Neural network-based detection of virtual environment anomalies”. In: *Neural Computing and Applications* 23.6, pp. 1711–1728. ISSN: 09410643. DOI: 10.1007/s00521-012-1132-x.

Neal, Brady (2020). *Introduction to Causal Inference from a Machine Learning Perspective*. en.

Nguyen, Trong Nguyen and Jean Meunier (Oct. 2019). “Anomaly Detection in Video Sequence With Appearance-Motion Correspondence”. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. ISSN: 2380-7504, pp. 1273–1283. DOI: 10.1109/ICCV.2019.00136.

Niizumi, Daisuke, Daiki Takeuchi, Yasunori Ohishi, Noboru Harada, and Kunio Kashino (Apr. 2021). “BYOL for Audio: Self-Supervised Learning for General-Purpose Audio Representation”. en. In: *arXiv:2103.06695 [cs, eess]*. URL: http://arxiv.org/abs/2103.06695 (visited on 01/14/2022).

Offutt, Jeff and Aynur Abdurazik (1999). “Generating Tests from UML Specifications”. In: *«UML»’99 — The Unified Modeling Language*. Ed. by Robert France and Bernhard Rumpe. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 416–429. ISBN: 978-3-540-46852-3.

Oh, Junhyuk, Xiaoxiao Guo, Honglak Lee, Richard Lewis, and Satinder Singh (2015). “Action-conditional video prediction using deep networks in Atari games”. In: *Advances in Neural Information Processing Systems*, pp. 2863–2871. arXiv: 1507.08750.

Oudeyer, Pierre-Yves and Frederic Kaplan (2009). “What is intrinsic motivation? A typology of computational approaches”. In: *Frontiers in Neurorobotics* 1. ISSN: 1662-5218. URL: https://www.frontiersin.org/articles/10.3389/neuro.12.006.2007 (visited on 10/14/2022).

Paavilainen, Janne (Jan. 2020). “Defining playability of games: functionality, usability, and gameplay”. en. In: *Proceedings of the 23rd International Conference on Academic Mindtrek*. Tampere Finland: ACM, pp. 55–64. ISBN: 978-1-4503-7774-4. DOI: 10.1145/3377290.3377309.

Pang, Guansong, Chunhua Shen, Longbing Cao, and Anton van den Hengel (Mar. 2022). “Deep Learning for Anomaly Detection: A Review”. en. In: *ACM Computing Surveys* 54.2. arXiv: 2007.02500, pp. 1–38.

ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3439950. URL: http://arxiv.org/abs/2007.02500 (visited on 03/28/2022).

Pascarella, Luca, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli (2018). “How is Video Game Development Different from Software Development in Open Source?” In: *Proceedings of the 15th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 392–402. ISBN: 9781450357166. DOI: 10.1145/3196398.3196418. URL: https://doi.org/10.1145/3196398.3196418.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell (May 2017). “Curiosity-driven exploration by self-supervised prediction”. In: *34th International Conference on Machine Learning, ICML 2017*. Vol. 6. arXiv: 1705.05363, pp. 4261–4270. ISBN: 978-1-5108-5514-4. URL: http://arxiv.org/abs/1705.05363 (visited on 08/15/2019).

Patil, N. and Prabir Kumar Biswas (July 2017). “A Survey of Video Datasets for Anomaly Detection in Automated Surveillance”. In: *Proceedings - 2016 6th International Symposium on Embedded Computing and System Design, ISED 2016*. Institute of Electrical and Electronics Engineers Inc., pp. 43–48. ISBN: 978-1-5090-2541-1. DOI: 10.1109/ISED.2016.7977052. (Visited on 08/10/2020).

Paul W. Holland (1986). “Statistics and Causal Inference”. en. In: *Journal of the American Statistical Association*, pp. 945–960.

Pfau, Johannes, Antonios Liapis, Georg Volkmar, Georgios N. Yannakakis, and Rainer Malaka (2020). “Dungeons & Replicants: Automated Game Balancing via Deep Player Behavior Modeling”. In: *2020 IEEE Conference on Games (CoG)*, pp. 431–438. DOI: 10.1109/CoG47356.2020.9231958.

Pfau, Johannes, Jan David Smeddinck, and Rainer Malaka (2017). “Automated game testing with ICARUS: Intelligent completion of adventure riddles via unsupervised solving”. In: *CHI PLAY 2017*. Association for Computing Machinery, Inc, pp. 153–163. ISBN: 9781450351119. DOI: 10.1145/3130859.3131439.

Pini, Stefano, Olfa Ben Ahmed, Marcella Cornia, Lorenzo Baraldi, Rita Cucchiara, and Benoit Huet (Nov. 2017). “Modeling multimodal cues in a deep learning-based framework for emotion recognition in the wild”. en. In: *Proceedings of the 19th ACM International Conference on Multimodal Interaction*. Glasgow UK: ACM, pp. 536–543. ISBN: 978-1-4503-5543-8. DOI: 10.1145/3136755.3143006.

Politowski, Cristiano, Yann-Gaël Guéhéneuc, and Fabio Petrillo (Mar. 2022). “Towards Automated Video Game Testing: Still a Long Way to Go”. In: *arXiv*. URL: http://arxiv.org/abs/2202.12777 (visited on 05/14/2022).

Politowski, Cristiano, Fabio Petrillo, and Yann Gael Gueheneuc (2021a). “A Survey of Video Game Testing”. In: *Proceedings - 2021 IEEE/ACM International Conference on Automation of Software Test, AST 2021*. arXiv: 2103.06431, pp. 90–99. ISBN: 978-1-66543-567-3. DOI: 10.1109/AST52587.2021.00018. URL: http://bit.ly/2M41lHG (visited on 11/29/2021).

Politowski, Cristiano, Fabio Petrillo, Gabriel C. Ullmann, and Yann-Gaël Guéhéneuc (June 2021b). “Game industry problems: An extensive analysis of the gray literature”. en. In: *Information and Software Technology* 134, p. 106538. ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106538. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000252 (visited on 06/11/2022).

Politowski, Cristiano, Fabio Petrillo, Gabriel Cavalheiro Ullmann, Josias de Andrade Werly, and Yann-Gaël Guéhéneuc (June 2020). “Dataset of Video Game Development Problems”. en. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. Seoul Republic of Korea: ACM, pp. 553–557. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387486. URL: https://dl.acm.org/doi/10.1145/3379597.3387486 (visited on 06/11/2022).

Prasetya, I.S.W.B., Maurin Voshol, Tom Tanis, Adam Smits, Bram Smit, Jacco Van Mourik, Menno Klunder, Frank Hoogmoed, Stijn Hinlopen, August Van Casteren, Jesse Van De Berg, Naraenda G.W.Y. Prasetya, Samira Shirzadehhajimahmood, and Saba Gholizadeh Ansari (2020). “Navigation and exploration in 3D-game automated play testing”. In: *Proceedings of the 11th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation*. Association for Computing Machinery, Inc, pp. 3–9. ISBN: 9781450381017. DOI: 10.1145/3412452.3423570.

Puterman, Martin L (1994). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons. ISBN: 9780471619772. DOI: 10.1002/9780470316887.

Quinones, Jorge R. and Antonio J. Fernandez-Leiva (2020). “XML-Based Video Game Description Language”. en. In: *IEEE Access* 8, pp. 4679–4692. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2962969. URL: https://ieeexplore.ieee.org/document/8945249/ (visited on 10/11/2022).

Radford, Alec, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever (Feb. 2021). “Learning Transferable Visual Models From Natural Language Supervision”. en. In: *arXiv*. URL: http://arxiv.org/abs/2103.00020 (visited on 01/12/2023).

Radomski, Stefan and Tim Neubacher (2015). “Formal Verification of Selected Game-Logic Specifications”. In: *Proceedings of the 2nd EICS Workshopon Engineering Interactive ComputerSystems with SCXML*, pp. 30–34.

Rajmohan, V. and E. Mohandas (2007). “Mirror neuron system”. In: *Indian Journal of Psychiatry* 49.1, pp. 66–69. ISSN: 0019-5545. DOI: 10.4103/0019-5545.31522. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2900004/ (visited on 07/22/2022).

Ramachandra, Bharathkumar, Michael J. Jones, and Ranga Raju Vatsavai (Aug. 2020). “A Survey of Single-Scene Video Anomaly Detection”. en. In: *arXiv*. URL: http://arxiv.org/abs/2004.05993 (visited on 02/19/2023).

Ravanbakhsh, Mahdyar, Moin Nabi, Enver Sangineto, Lucio Marcenaro, Carlo Regazzoni, and Nicu Sebe (Aug. 2017). “Abnormal Event Detection in Videos using Generative Adversarial Nets”. en. In: arXiv:1708.09644 [cs]. URL: http://arxiv.org/abs/1708.09644 (visited on 12/12/2022).

Ravanbakhsh, Mahdyar, Enver Sangineto, Moin Nabi, and Nicu Sebe (Nov. 2018). “Training Adversarial Discriminators for Cross-channel Abnormal Event Detection in Crowds”. en. In: arXiv:1706.07680 [cs]. URL: http://arxiv.org/abs/1706.07680 (visited on 12/12/2022).

Reiterman, J., V. Rödl, and E. Šinˇajová (1989). “Geometrical embeddings of graphs”. en. In: *Discrete Mathematics* 74.3, pp. 291–319. ISSN: 0012365X. DOI: 10.1016/0012-365X(89)90142-8. URL: https://linkinghub.elsevier.com/retrieve/pii/0012365X89901428 (visited on 11/28/2022).

Ren, Pengzhen, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B. Gupta, Xiaojiang Chen, and Xin Wang (Dec. 2021). “A Survey of Deep Active Learning”. In: *arXiv*. DOI: 10.48550/arXiv.2009.00236. URL: http://arxiv.org/abs/2009.00236 (visited on 02/22/2023).

Rhodes, William and Gerald Gaes (Aug. 2020). “Estimating the Distribution of Treatment Effects From Random Design Experiments”. en. In: *Evaluation Review* 44.4, pp. 262–294. ISSN: 0193-841X, 1552-3926. DOI: 10.1177/0193841X20906232. URL: http://journals.sagepub.com/doi/10.1177/0193841X20906232 (visited on 01/09/2023).

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer, pp. 234–241.

Roohi, Shaghayegh, Jari Takatalo, Christian Guckelsberger, and Perttu Hämäläinen (2018). “Review of intrinsic motivation in simulation-based game testing”. en-GB. In: *Conference on Human Factors in Computing Systems - Proceedings* 2018-April. DOI: 10.1145/3173574.3173921. (Visited on 04/03/2022).

Ross, Stéphane, Geoffrey J Gordon, and J Andrew Bagnell (2011). “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning”. en. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635.

Roy, Jefferson E. and Kathleen E. Cullen (Mar. 2004). “Dissociating Self-Generated from Passively Applied Head Motion: Neural Mechanisms in the Vestibular Nuclei”. In: *The Journal of Neuroscience* 24.9, pp. 2102–2111. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.3988-03.2004. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6730417/ (visited on 07/11/2022).

Rubin, Donald B. (Oct. 1974). “Estimating causal effects of treatments in randomized and nonrandomized studies.” en. In: *Journal of Educational Psychology* 66.5, pp. 688–701. ISSN: 1939-2176, 0022-0663. DOI: 10.1037/h0037350. URL: http://doi.apa.org/getdoi.cfm?doi=10.1037/h0037350 (visited on 12/24/2022).

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (Oct. 1986). “Learning representations by back-propagating errors”. en. In: *Nature* 323.6088, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 05/24/2022).

Runeson, Per (2006). “A Survey of Unit Testing Practices”. In: *IEEE Software* 23. DOI: 10.1109/MS.2006.91.

Rushby, John (2008). “Automated Test Generation and Verified Software”. en. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 161–172. ISBN: 978-3-540-69147-1 978-3-540-69149-5. DOI: 10.1007/978-3-540-69149-5_18.

Ryan, Richard M., C. Scott Rigby, and Andrew Przybylski (Dec. 2006). “The Motivational Pull of Video Games: A Self-Determination Theory Approach”. en. In: *Motivation and Emotion* 30.4, pp. 344–360. ISSN: 0146-7239, 1573-6644. DOI: 10.1007/s11031-006-9051-8.

Sabokrou, M., M. Fathy, and M. Hoseini (2016). “Video anomaly detection and localisation based on the sparsity and reconstruction error of auto-encoder”. en. In: *Electronics Letters* 52.13, pp. 1122–1124. ISSN: 1350-911X. DOI: 10.1049/el.2016.0440.

Sabokrou, Mohammad, Mahmood Fathy, Mojtaba Hoseini, and Reinhard Klette (June 2015). “Real-time anomaly detection and localization in crowded scenes”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. ISSN: 2160-7516, pp. 56–62. DOI: 10.1109/CVPRW.2015.7301284.

Sabokrou, Mohammad, Mohsen Fayyaz, Mahmood Fathy, and Reinhard Klette (Apr. 2017). “Deep-Cascade: Cascading 3D Deep Neural Networks for Fast Anomaly Detection and Localization in Crowded Scenes”. In: *IEEE Transactions on Image Processing* 26.4, pp. 1992–2004. ISSN: 1941-0042. DOI: 10.1109/TIP.2017.2670780.

Sabokrou, Mohammad, Mohammad Khalooei, Mahmood Fathy, and Ehsan Adeli (May 2018). "Adversarially Learned One-Class Classifier for Novelty Detection". en. In: *arXiv*. URL: http://arxiv.org/abs/1802.09088 (visited on 12/12/2022).

Salehi, Mahsa and Lida Rashidi (May 2018). "A Survey on Anomaly detection in Evolving Data". In: *ACM SIGKDD Explorations Newsletter* 20.1. Publisher: ACM, pp. 13–23. ISSN: 19310145. DOI: 10.1145/3229329.3229332. URL: http://dl.acm.org/citation.cfm?doid=3229329.3229332 (visited on 07/15/2019).

Samvelyan, Mikayel, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Küttler, Edward Grefenstette, and Tim Rocktäschel (Nov. 2021). *MiniHack the Planet: A Sandbox for Open-Ended Reinforcement Learning Research*. Tech. rep. arXiv:2109.13202. arXiv:2109.13202 [cs, stat] type: article. arXiv. DOI: 10.48550/arXiv.2109.13202. URL: http://arxiv.org/abs/2109.13202 (visited on 06/07/2022).

Santos, Ronnie E.S., Cleyton V.C. Magalhes, Luiz Fernando Capretz, Jorge S. Correia-Neto, Fabio Q.B. Da Silva, and Abdelrahman Saher (Oct. 2018). "Computer games are serious business and so is their quality: Particularities of software testing in game development from the perspective of practitioners". In: *International Symposium on Empirical Software Engineering and Measurement*. ISSN: 19493789. New York, NY, USA: IEEE Computer Society, pp. 1–10. ISBN: 978-1-4503-5823-1. DOI: 10.1145/3239235.3268923. URL: https://dl.acm.org/doi/10.1145/3239235.3268923 (visited on 07/27/2020).

Sawada, Yoshihide (2018). "Disentangling Controllable and Uncontrollable Factors of Variation by Interacting with the World". In: *arXiv*. arXiv: 1804.06955. URL: http://arxiv.org/abs/1804.06955.

Schaal, Stefan (1997). "Learning From Demonstration". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 1040–1046. URL: http://www.cc.gatech.edulfac/Stefan.Schaal.

Schaul, Tom (Aug. 2013). "A Video Game Description Language for Model-Based or Interactive Learning". en. In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. Niagara Falls, ON, Canada: IEEE, pp. 1–8. ISBN: 978-1-4673-5311-3 978-1-4673-5308-3. DOI: 10.1109/CIG.2013.6633610. URL: http://ieeexplore.ieee.org/document/6633610/ (visited on 10/17/2022).

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver (2016). "Prioritized Experience Replay". en. In: *arXiv*. URL: http://arxiv.org/abs/1511.05952.

Schell, Jesse (2008). *The art of game design: a book of lenses*. en. OCLC: ocn213839335. Amsterdam ; Boston: Elsevier/Morgan Kaufmann. ISBN: 978-0-12-369496-6.

Schillaci, Guido, Claas-Norman Ritter, Verena Vanessa Hafner, and Bruno Lara (July 2016). “Body Representations for Robot Ego-Noise Modelling and Prediction. Towards the Development of a Sense of Agency in Artificial Agents”. en. In: MIT Press, pp. 390–397. DOI: 10.1162/978-0-262-33936-0-ch065. URL: https://direct.mit.edu/isal/article/doi/10.1162/978-0-262-33936-0-ch065/99437/Body-Representations-for-Robot-Ego-Noise-Modelling (visited on 07/30/2022).

Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver (Dec. 2020). “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”. In: *Nature* 588, pp. 604–609. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://arxiv.org/abs/1911.08265 (visited on 08/01/2022).

Schroder-Schetelig, Johannes, Poramate Manoonpong, Florentin Worgotter, J Schroder-Schetelig, P Manoonpong, and · F Worgotter (2010). “Using efference copy and a forward internal model for adaptive biped walking”. In: *Autonomous Robots* 29, pp. 357–366. DOI: 10.1007/s10514-010-9199-7. (Visited on 08/04/2021).

Schroff, Florian, Dmitry Kalenichenko, and James Philbin (2015). “FaceNet: A unified embedding for face recognition and clustering”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, pp. 815–823. ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298682.

Schultz, Charles P, Robert Bryant, and Tim Langdell (2005). *Game Testing All in One*. ISBN: 1592003737.

Segura, Sergio, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen (2020). “Metamorphic Testing: Testing the Untestable”. en. In: *IEEE Software* 37.3, pp. 46–53. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2018.2875968.

Sehwag, Vikash, Arjun Nitin Bhagoji, Liwei Song, Chawin Sitawarin, Daniel Cullina, Mung Chiang, and Prateek Mittal (Nov. 2019). “Analyzing the Robustness of Open-World Machine Learning”. In: *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*. AISec’19. New York, NY, USA: Association for Computing Machinery, pp. 105–116. ISBN: 978-1-4503-6833-9. DOI: 10.1145/3338501.3357372.

Sehwag, Vikash, Mung Chiang, and Prateek Mittal (2021). “SSD: A Unified Framework for Self-Supervised Outlier Detection”. In: abs/2103.12051. URL: https://arxiv.org/abs/2103.12051.

Sen, Koushik, Darko Marinov, and Gul Agha (Sept. 2005). “CUTE: a concolic unit testing engine for C”. In: *ACM SIGSOFT Software Engineering Notes* 30.5, pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750. URL: https://doi.org/10.1145/1095430.1081750 (visited on 09/20/2022).

Sermanet, Pierre, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, and Sergey Levine (Mar. 2018). “Time-Contrastive Networks: Self-Supervised Learning from Video”. en. In: *arXiv*. URL: http://arxiv.org/abs/1704.06888 (visited on 11/08/2022).

Sestini, Alessandro, Joakim Bergdahl, Konrad Tollmar, Andrew D Bagdanov, and Linus Gisslén (2022). “Towards Informed Design and Validation Assistance in Computer Games Using Imitation Learning”. en. In: *NeurIPS Workshop on Human in the Loop Learning*.

Settles, Burr (1995). “Active Learning Literature Survey”. In: *Science* 10.3, pp. 237–304.

Shalit, Uri, Fredrik D Johansson, and David Sontag (2017). “Estimating individual treatment effect: generalization bounds and algorithms”. en. In: pp. 3076–3085.

Shapley, L. S. (1953). “Stochastic Games*”. In: *Proceedings of the National Academy of Sciences* 39.10, pp. 1095–1100. DOI: 10.1073/pnas.39.10.1095. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.39.10.1095. URL: https://www.pnas.org/doi/abs/10.1073/pnas.39.10.1095.

Shi, Lin, Fangwen Mu, Yumin Zhang, Ye Yang, Junjie Chen, Xiao Chen, Hanzhi Jiang, Ziyou Jiang, and Qing Wang (May 2022). “BugListener: identifying and synthesizing bug reports from collaborative live chats”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. New York, NY, USA: Association for Computing Machinery, pp. 299–311. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510108. URL: https://doi.org/10.1145/3510003.3510108 (visited on 10/07/2022).

Shin, Yuchul, Jaewon Kim, Kyohoon Jin, and Young Bin Kim (2020). “Playtesting in Match 3 Game Using Strategic Plays via Reinforcement Learning”. en. In: *IEEE Access* 8, pp. 51593–51600. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2980380. URL: https://ieeexplore.ieee.org/document/9034187/ (visited on 06/11/2022).

Shirzadehhajimahmood, Samira, I. S. W. B. Prasetya, Frank Dignum, Mehdi Dastani, and Gabriele Keller (Aug. 2021). “Using an agent-based approach for robust automated testing of computer games”. In: *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation*. Association for Computing Machinery (ACM). ISBN: 978-1-4503-8623-4. DOI: 10.1145/3472672.3473952.

Silva, Fernando de Mesentier, Igor Borovikov, John Kolen, Navid Aghdaie, and Kazi Zaman (Nov. 2018). “Exploring Gameplay With AI Agents”. en. In: *arXiv*. URL: http://arxiv.org/abs/1811.06962 (visited on 06/11/2022).

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis (Jan. 2016). “Mastering the game of Go with deep neural networks and tree search”. en. In: *Nature* 529.7587, pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://www.nature.com/articles/nature16961.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis (Dec. 2017). “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: DOI: 10.48550/arXiv.1712.01815. URL: http://arxiv.org/abs/1712.01815.

Sinan, Aysu Betin-Can, and Elif Surer (Oct. 2019). “Automated Video Game Testing Using Synthetic and Human-Like Agents”. In: *IEEE Transactions on Games* 13.1. arXiv: 1906.00317 Publisher: Institute of Electrical and Electronics Engineers (IEEE), pp. 50–67. ISSN: 2475-1502. DOI: 10.1109/TG.2019.2947597. (Visited on 07/30/2020).

Singh, Satinder, Andrew G Barto, and Nuttapong Chentanez (2005). “Intrinsically motivated reinforcement learning”. In: *Advances in Neural Information Processing Systems.* ISSN: 10495258. ISBN: 0-262-19534-8.

Smith, Adam M. and Michael Mateas (Sept. 2011). “Answer Set Programming for Procedural Content Generation: A Design Space Approach”. en. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3, pp. 187–200. ISSN: 1943-068X, 1943-0698. DOI: 10.1109/TCIAIG.2011.2158545. URL: http://ieeexplore.ieee.org/document/5783900/ (visited on 10/11/2022).

Smith, Adam M., Mark J. Nelson, and Michael Mateas (Aug. 2010). “LUDOCORE: A logical game engine for modeling videogames”. en. In: *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games.* Copenhagen, Denmark: IEEE, pp. 91–98. ISBN: 978-1-4244-6295-7. DOI: 10.1109/ITW.2010.5593368. URL: http://ieeexplore.ieee.org/document/5593368/ (visited on 10/11/2022).

Software bug Definition (n.d.). en. URL: https://www.lawinsider.com/dictionary/software-bug (visited on 09/21/2022).

Software testing techniques (1990). USA: John Wiley & Sons, Inc. ISBN: 0442245920.

Song, Yang and Oscar Chaparro (Nov. 2020). “BEE: a tool for structuring and analyzing bug reports”. en. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Virtual Event USA: ACM, pp. 1551–1555. ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3417928. URL: https://dl.acm.org/doi/10.1145/3368089.3417928 (visited on 10/07/2022).

Sperry, R. W. (1950). “Neural basis of the spontaneous optokinetic response produced by visual inversion”. In: *Journal of Comparative and Physiological Psychology* 43.6, pp. 482–489. ISSN: 00219940. DOI: 10.1037/h0055479. URL: /record/1951-03616-001.

Stadie, Bradly C., Sergey Levine, and Pieter Abbeel (Nov. 2015). “Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models”. en. In: *arXiv*. URL: http://arxiv.org/abs/1507.00814 (visited on 11/20/2022).

Stahlke, Samantha, Atiya Nova, and Pejman Mirza-Babaei (2020). “Artificial Players in the Design Process: Developing an Automated Testing Tool for Game Level and World Design”. en. In: *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*. Virtual Event Canada: ACM, pp. 267–280. ISBN: 978-1-4503-8074-4. DOI: 10.1145/3410404.3414249. URL: https://dl.acm.org/doi/10.1145/3410404.3414249 (visited on 09/19/2022).

Tack, Jihoon, Sangwoo Mo, Jongheon Jeong, and Jinwoo Shin (2020). “CSI: Novelty Detection via Contrastive Learning on Distributionally Shifted Instances”. en. In: *arXiv*. URL: http://arxiv.org/abs/2007.08176 (visited on 12/24/2022).

Taesiri, Mohammad Reza, Finlay Macklon, and Cor-Paul Bezemer (Mar. 2022). “CLIP meets GamePhysics: Towards bug identification in gameplay videos using zero-shot transfer learning”. en. In: arXiv:2203.11096 [cs]. URL: http://arxiv.org/abs/2203.11096 (visited on 01/04/2023).

Tamm, Matilda, Olivia Shamon, Hector Anadon Leon, Konrad Tollmar, and Linus Gisslen (Aug. 2022). “Automatic Testing and Validation of Level of Detail Reductions Through Supervised Learning”. en. In: *2022 IEEE Conference on Games (CoG)*. Beijing, China: IEEE, pp. 191–198. ISBN: 978-1-66545-989-1. DOI: 10.1109/CoG51982.2022.9893682. URL: https://ieeexplore.ieee.org/document/9893682/.

Tang, Haoran, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel (Dec. 2017). “#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning”. In: *arXiv*. DOI: 10.48550/arXiv.1611.04717. URL: http://arxiv.org/abs/1611.04717 (visited on 10/14/2022).

Tasfi, Norman (2016). *PyGame Learning Environment*. Publication Title: GitHub repository. URL: https://github.com/ntasfi/PyGame-Learning-Environment.

Tassa, Yuval, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller (Jan. 2018). “DeepMind Control Suite”. en. In: *arXiv*. URL: http://arxiv.org/abs/1801.00690 (visited on 06/07/2022).

Terry, J. K, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sulivan, Luis Santos, Rodrigo Perez, Caroline Horsch, Clemens Dieffendahl, Niall L Williams, Yashas Lokesh, Ryan Sullivan, and Praveen Ravi (2020). "PettingZoo: Gym for Multi-Agent Reinforcement Learning". In: *arXiv preprint arXiv:2009.14471*.

Thomas, Valentin, Emmanuel Bengio, William Fedus, Jules Pondard, Philippe Beaudoin, Hugo Larochelle, Joelle Pineau, Doina Precup, and Yoshua Bengio (2018). "Disentangling the independently controllable factors of variation by interacting with the world". In: *arXiv*. arXiv: 1802.09484. URL: http://arxiv.org/abs/1802.09484.

Tian, Yuandong, Xinlei Chen, and Surya Ganguli (Oct. 2021). "Understanding self-supervised Learning Dynamics without Contrastive Pairs". en. In: arXiv:2102.06810 [cs]. URL: http://arxiv.org/abs/2102.06810 (visited on 12/12/2022).

Tian, Yuchi, Kexin Pei, Suman Jana, and Baishakhi Ray (2018). "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 303–314. ISBN: 9781450356381. DOI: 10.1145/3180155.3180220. URL: https://doi.org/10.1145/3180155.3180220.

Todorov, Emanuel, Tom Erez, and Yuval Tassa (2012). "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

VanderWeele, Tyler J. (Nov. 2009). "Concerning the Consistency Assumption in Causal Inference". en-US. In: *Epidemiology* 20.6, p. 880. ISSN: 1044-3983. DOI: 10.1097/EDE.0b013e3181bd5638. URL: https://journals.lww.com/epidem/fulltext/2009/11000/concerning_the_consistency_assumption_in_causal.18.aspx (visited on 12/24/2022).

Varvaressos, Simon, Kim Lavoie, Alexandre Blondin Massé, Sébastien Gaboury, and Sylvain Hallé (2014). "Automated bug finding in video games: A case study for runtime monitoring". In: *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*. IEEE Computer Society, pp. 143–152. ISBN: 9780769551852. DOI: 10.1109/ICST.2014.27.

Vinyals, Oriol, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver (Nov. 2019). “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. en. In: *Nature* 575.7782, pp. 350–354. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/s41586-019-1724-z`. URL: `http://www.nature.com/articles/s41586-019-1724-z`.

Vinyals, Oriol, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing (Aug. 2017). *StarCraft II: A New Challenge for Reinforcement Learning*. Tech. rep. arXiv:1708.04782. arXiv. DOI: `10.48550/arXiv.1708.04782`. URL: `http://arxiv.org/abs/1708.04782` (visited on 06/07/2022).

Volz, Vanessa, Günter Rudolph, and Boris Naujoks (Mar. 2016). “Demonstrating the Feasibility of Automatic Game Balancing”. en. In: arXiv:1603.03795 [cs]. URL: `http://arxiv.org/abs/1603.03795` (visited on 10/10/2022).

Wang, Jian, Feng Zhou, Shilei Wen, Xiao Liu, and Yuanqing Lin (2017). “Deep Metric Learning with Angular Loss”. In: *arXiv*. arXiv: `1708.01682`. URL: `http://arxiv.org/abs/1708.01682`.

Washburn, Michael, Pavithra Sathiyanarayanan, Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird (2016). “What went right and what went wrong: An analysis of 155 postmortems from game development”. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, pp. 280–289. ISBN: 9781450341615. DOI: `10.1145/2889160.2889253`.

Watson, John S (1966). “The Development and Generalization of Contingency Awareness in Early Infancy: Some Hypotheses”. In: *Merrill-Palmer Quarterly of Behavior and Development* 12.2, pp. 123–135. URL: `http://www.jstor.org/stable/23082793%0Ahttp://www.jstor.org/stable/23082793%0Ahttp://www.jstor.org/stable/23082793`.

Weinberger, Kilian Q, John Blitzer, and Lawrence Saul (2005). “Distance Metric Learning for Large Margin Nearest Neighbor Classification”. In: *Advances in Neural Information Processing Systems*. Ed. by Y. Weiss, B. Schölkopf, and J. Platt. Vol. 18. MIT Press, pp. 1473–1480. URL: `https://proceedings.neurips.cc/paper/2005/file/a7f592cef8b130a6967a90617db5681b-Paper.pdf`.

Wikipedia contributors (2022). *List of Unity games — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=List_of_Unity_games&oldid=1091890960`. [Online; accessed 8-June-2022].

Wilkins, Benedict and Kostas Stathis (2022). "World of Bugs: A Platform for Automated Bug Detection in 3D Video Games". In: *2022 IEEE Conference on Games (CoG)*, pp. 520–523. DOI: 10.1109/CoG51982.2022.9893616.

— (2023). "Disentangling Reafferent Effects by Doing Nothing". In: *The Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI Press.

Wilkins, Benedict, Chris Watkins, and Kostas Stathis (2020). "A Metric Learning Approach to Anomaly Detection in Video Games". In: *2020 IEEE Conference on Games (CoG)*, pp. 604–607. ISBN: 9781728145334. DOI: 10.1109/CoG47356.2020.9231700. arXiv: 2005.10211.

Wolpert, Daniel M and J Randall Flanagan (2001). "Motor prediction". In: *Current Biology*. Vol. 11. 18, R729–32. DOI: 10.1016/s0960-9822(01)00432-8.

Wolpert, Daniel M., Zoubin Ghahramani, and Michael I. Jordan (1995). "An internal model for sensorimotor integration". In: *Science* 269.5232, pp. 1880–1882. ISSN: 00368075. DOI: 10.1126/science.7569931.

Wong, W., Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa (2016). "A Survey on Software Fault Localization". In: *IEEE Transactions on Software Engineering* 42, pp. 1–1. DOI: 10.1109/TSE.2016.2521368.

Wooff, D.A., M. Goldstein, and F.P.A. Coolen (2002). "Bayesian graphical models for software testing". In: *IEEE Transactions on Software Engineering* 28.5. Conference Name: IEEE Transactions on Software Engineering, pp. 510–525. ISSN: 1939-3520. DOI: 10.1109/TSE.2002.1000453.

Xu, Dan, Elisa Ricci, Yan Yan, Jingkuan Song, and Nicu Sebe (Oct. 2015). "Learning Deep Representations of Appearance and Motion for Anomalous Event Detection". en. In: *arXiv*. URL: http://arxiv.org/abs/1510.01553 (visited on 12/12/2022).

Xu, Dejing, Jun Xiao, Zhou Zhao, Jian Shao, Di Xie, and Yueting Zhuang (June 2019). "Self-Supervised Spatiotemporal Learning via Video Clip Order Prediction". en. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, pp. 10326–10335. ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.01058. URL: https://ieeexplore.ieee.org/document/8953292/ (visited on 11/08/2022).

Yang, Jie, Ruijie Xu, Zhiquan Qi, and Yong Shi (2021). "Visual Anomaly Detection for Images: A Survey". In: *arXiv*. URL: http://arxiv.org/abs/2109.13157 (visited on 01/03/2022).

Yang, John, Gyuejeong Lee, Simyung Chang, and Nojun Kwak (2019). "Towards Governing Agent's Efficacy: Action-Conditional β-VAE for Deep Transparent Reinforcement Learning". In: *Proceedings of Machine Learning Research*. Vol. 101, pp. 32–47.

Yee, Nick (Dec. 2006). “Motivations for Play in Online Games”. en. In: *CyberPsychology & Behavior* 9.6, pp. 772–775. ISSN: 1094-9313, 1557-8364. DOI: `10.1089/cpb.2006.9.772`.

Zaadnoordijk, Lorijn, Tarek R Besold, and Sabine Hunnius (May 2019). “A match does not make a sense: on the sufficiency of the comparator model for explaining the sense of agency”. In: *Neuroscience of Consciousness* 2019. ISSN: 2057-2107. DOI: `10.1093/nc/niz006`. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6511607/` (visited on 07/30/2022).

Zbontar, Jure, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny (June 2021). “Barlow Twins: Self-Supervised Learning via Redundancy Reduction”. In: *arXiv*. DOI: `10.48550/arXiv.2103.03230`. URL: `http://arxiv.org/abs/2103.03230` (visited on 12/08/2022).

Zhao, Yan, Weihao Zhang, Enyi Tang, Haipeng Cai, Xi Guo, and Na Meng (2021). “A Lightweight Approach of Human-Like Playtesting”. In: *arXiv* abs/2102.13026. URL: `https://arxiv.org/abs/2102.13026`.

Zheng, Alice, Michael Jordan, Ben Liblit, Mayur Naik, and Alex Aiken (2006). *Statistical debugging: Simultaneous identification of multiple bugs*. Vol. 2006. Journal Abbreviation: ICML 2006 - Proceedings of the 23rd International Conference on Machine Learning Pages: 1112 Publication Title: ICML 2006 - Proceedings of the 23rd International Conference on Machine Learning. DOI: `10.1145/1143844.1143983`.

Zheng, Yan, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan (Nov. 2019). “Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning”. en. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 772–784. ISBN: 978-1-72812-508-4. DOI: `10.1109/ASE.2019.00077`. URL: `https://ieeexplore.ieee.org/document/8952543/` (visited on 05/10/2022).

Zhong, Yuanyi, Alexander Schwing, and Jian Peng (2020). “Disentangling Controllable Object Through Video Prediction Improves Visual Reinforcement Learning”. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. Vol. 2020-May, pp. 3672–3676. ISBN: 9781509066315. DOI: `10.1109/ICASSP40776.2020.9053819`. arXiv: `2002.09136`.

Zhou, Joey Tianyi, Jiawei Du, Hongyuan Zhu, Xi Peng, Yong Liu, and Rick Siow Mong Goh (Oct. 2019). “AnomalyNet: An Anomaly Detection Network for Video Surveillance”. In: *IEEE Transactions on Information Forensics and Security* 14.10, pp. 2537–2550. ISSN: 1556-6021. DOI: `10.1109/TIFS.2019.2900907`.

Zhu, Yuxiang and Minxue Pan (Oct. 2019). “Automatic Code Summarization: A Systematic Literature Review”. en. In: *arXiv*. arXiv:1909.04352 [cs]. URL: `http://arxiv.org/abs/1909.04352`.

Ziebart, Brian D., Andrew Maas, J. Andrew Bagnell, and Anind K. Dey (2008). "Maximum Entropy Inverse Reinforcement Learning". In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. Chicago, Illinois: AAAI Press, 1433–1438. ISBN: 9781577353683.

Zuo, Max, Logan Schick, Matthew Gombolay, and Nakul Gopalan (Apr. 2022). "Efficient Exploration via First-Person Behavior Cloning Assisted Rapidly-Exploring Random Trees". en. In: *arXiv*. URL: `http://arxiv.org/abs/2203.12774` (visited on 10/13/2022).

Acronyms

ABD Automated Bug Detection. 21, 22, 25, 34, 56, 83, 84, 161, 198

ARE Average Reafferent Effect. 138, 140, 141, 143, 198

AUC Area Under Curve. 198

CI Causal Inference. 7, 198

CNS Central Nervous System. 198

FSM Finite State Machine. 198

GUT Game Under Test. 17, 18, 55, 56, 69, 80, 198

IID Independent and Identically Distributed. 143, 145, 198

IRE Individual Reafferent Effect. 139, 140, 198

LLM Large Language Model. 32, 198

MCTS Monte-Carlo Tree Search. 40, 41, 198

MDP Markov Decision Process. 56, 57, 136, 198

MLP MultiLayer Perceptron. 109–111, 198

MSE Mean Squared Error. 109, 198

NPC Non-Player Character. 34, 61, 91, 100, 155, 198

OOD Out-Of-Distribution. 66, 125, 198, 219

QA Quality Assurance. 26, 198

RL Reinforcement Learning. 36, 198

ROC Receiver Operator Characteristic. 198

S3N State-State Siamese Networks. 22, 161, 198

SCM Structural Causal Model. 139, 145, 198

SGD Stochastic Gradient Descent. 143, 144, 198

SSL Self-Supervised Learning. 7, 21, 22, 66, 75, 103, 161, 198, 246

SUT System Under Test. 32, 198

TDD Test Driven Development. 30, 198

Glossary

absorbing state A state with a transition only to itself. . 198

acceptance testing A testing phase where the whole application against the requirements. This may involve testing *in the wild* with real users. . 15, 27, 198

action In decision theory, an action is a label, index (e.g. $a_1, a_2, \cdots, a_n$) or value in a continuous range that has some correspondance with what an agent can *do* to change the environment in which it resides. *see also action space and intervention.* . 34, 198

action space The space of possible actions that an agent may take. For example, $\{1, 2, \cdots, n\}$ if the agent may take n possible discrete actions, or $[a - b]$ if the action is continuous. *see also action.* . 57, 198

afference Any incoming sensory signal or effect on an agent's sensory system. *see also observation* . 143, 198

agent An agent is anything that, given its observations and beliefs about the world in which it resides, takes action to acheive its goals. There are many examples, including game testers, humans, animals, software agents and game playing agents. . 198, 201, 203, 204, 207, 209

aleatoric uncertainty Aleatoric uncertainty is uncertainty due to the inherent randomness of a system. This kind of uncertainty cannot be reduced, at least not without atlering the system itself.. 148, 198

alignment To align an agent's behaviour with our goals or intentions. The alignment problem refers to the challenges faced when trying to do this. Alignment is often talked about with reference to safety, but broadly it means to have agents or AI systems do what we want in the face of vague or informal instruction. . 47, 53, 70, 160, 198

alpha testing A testing phase where software is tested by a relatively small select group of users or team members in a staged or lab environment.. 27, 198

always bug Always bugs are class of bugs that are common to the vast majority of video games and are always considered bugs. Examples include geometry corruption and . Not all bugs are always bugs, what is a bug in one game might be a feature in other. An example is the high force bug. In some games applying large forces to objects may be part of a game mechanic. . 68, 165, 198

analysis The process of recreating, checking and fixing the reported bug, also called *debugging*.. 25, 29, 198

applicability A test is applicable if it can identify a range of different bugs. In traditional testing terms this is not always desirable as it violates a key testing principle - that tests should be simple, modular and test for *one thing*. Where learning is involved, general purpose test cases can be created which may let an agent identify different bugs *for free*, for example by learning what is normal under a closed world assumption. In other words, a single objective function might support the identification of many different kinds of bugs. . 101, 198

avatar The player's representation in the game, often a humanoid character that they can control. . 26, 198

beta testing A testing phase with a large group of users typically done after alpha testing. The goal is to get real-world feedback on the product.. 27, 45, 198

black box testing Any form of tesing where the tester does not have knowledge of the underlying implementation of the program under test. *see also white box testing* . 27, 43, 198

bug detection The process of searching for and identifying a bug in the game. Search typically means actually playing some portion of the game. Identification means to recognize that there is mismatch between observed behaviour and intended behaviour. . 25, 28, 198

capability In the context of untestable software, a test (or agent) is capable if it can identify a bug for which manually writing rules or guards to do the same job would be very difficult. The example used in chapter 2 is that of a handstanding horse (see Fig. 2.4). . 7, 17–19, 81, 100, 101, 159, 162, 198

catastrophic forgetting The tendency of an artificial neural network to abruptly and completely forget previously learned information upon learning new information. . 53, 69, 198

code coverage The proportion of statements in code that were actually executed during a series of tests or runs of a program. . 16, 29, 33, 35, 198

code coverage The proportion of states that were reached by an agent in a series of runs. . 198

contingent region The region of an agent's observation which may be affected by taking any action currently avaliable to the agent. See Bellemare et al. 2012 for a formal definition. . 198

continual learning A problem setting in Machine Learning (ML), where an agent should learn a model for a number of tasks sequentially, each of which may follow different data distributions while being able to remember and apply knowledge from previous tasks to new tasks. An example in testing is to learn about one kind of bug in some initial version of the game, then about another in an updated version and be able to remember and identify both in the updated version even under distributional shift. Continual learning is otherwise known as *sequential, incremental, or life-long* learning. . 69, 198

counterfactual Contrary to what has been observed or is *factual.* A counterfactual expresses what has *not* happened. Counterfactual questions often look something like the following: If I had not taken action a, and instead took action b, what would have happened? . 140, 198

coverage-based testing A kind of testing that where the aim is to maximise code coverage so as to uncover as many bugs as possible. For large programs this can be difficult to acheive. *see also experience-based testing* . 43, 198

crash A program crashes, exiting from normal execution, when it reaches a state from which it cannot continue. Examples of crashes include attempting to divide by zero, segmentation faults, or invalidating guards or pre-/post-conditions in an unrecoverable fashion. . 15, 16, 84, 198

curse of dimensionality The curse of dimensionality refers to the fact that the number of samples needed to estimate an arbitrary function with a given level of accuracy grows exponentially with the function's dimensionality.. 76, 198

decision boundary The boundary over which a decision changes, on one side of the boundary a learner may answer *test pass* and on the other *test fail.* . 52, 198

decoupling A test is decoupled if it does not require access to implementation details, it treats the GUT as a black box. A test is said to be strongly decoupled if it works with a program's input/output directly. In video games, this means with the player's action and audiovisual observation. All games share this action-observation API at a high-level, so tests that operate on this input/output space have the potential to be widely reusable. *see also reusability* . 74, 81, 153, 198

deterministic An environment is deterministic for every state there is exactly one possible next state for each action taken. . 198

developer A developer is anyone who brings the software closer to the requirements, by developing code, 3D models, artwork or other assets, writing narratives, designing characters etc. . 26, 198

distributional shift The change in the underlying distribution of data from when a model is trained and when it is used. This is a problem in non-stationary environments and the principle problem faced in continual learning. . 69, 198

dominant strategy A strategy is dominant if there exists no other strategy that can *win* over it. . 198

efference Any outgoing motor signal or action. . 198

embedding An embedding space is the space in which data resides after a process of dimensionality reduction or manifold learning. For example, images may be reduced to low-dimensional vector representations. These vectors are sometimes refered to as *embeddings*. . 32, 49, 198

endemic fault An endemic fault that shifts probability mass from intended states to unintended states in the transition distribution for a particular state (and action). An example might be a disabled collision box for a wall. Whenever the agent encounters the wall and takes action to bump into, they will pass through it.. 72, 198

environment The "world" or universe in which an agent resides. In other words, the surroundings of the agent. Environments are typically stateful - consisting of quantities or qualities that change with time, and that can be influenced through the action of an agent. *see also state* . 198

epistemic uncertainty Epistemic uncertainty is uncertainty due to lack of knowledge about a system.. 60, 198

error A mistake or conceptual misunderstanding of a human "IEEE Standard Classification for Software Anomalies" 2010. Errors are not to be confused with exceptions. *see also fault, failure*. . 27, 198

estimand A statistical quantity that is to be estimated.. 76, 138, 143, 198

exafference Any incoming sensory signal or effect on an agent's sensory system that is caused by external (environmental) conditions, processes or events. . 132, 198

exception Exceptions are the result of a failed guard, pre/post condition or constraint that occurs when running a program. . 16, 198

expected behaviour Expected behaviour is a tester's model or interpretation of intended behaviour. This has been derived from various sources of information (e.g. communicating with colleagues/stakeholders/players, playing the GUT, playing other games, real-world experience, etc.). In testing, the tester is comparing expected behaviour to observed behaviour. Expected behaviour is used to make explicit the fact that testers are always filling in gaps (e.g. using common sense) and may not fully grasp what is intended. This is made clearer when thinking in terms of automation. An agent may similarly try to model intended behaviour guided by using supervision from an oracle (a human tester). But more generally, expected behaviour may refer to a (statistical) model of the agent's environment which it has formed through experience. This model may or may not overlap with intended behaviour, it is our job as designers of the agent to ensure that there is overlap by tuning inductive biases, exposing the agent to the right information, or otherwise giving supervision. *see also observed behaviour, intended behaviour.* . 22, 51, 62, 70, 74, 198

experience An experience is an observation, or part of it, or any collection of observations (i.e. part of a trajectory) with or without the associated actions. Experiences may be abstractly represented by an agent i.e. they are not restricted to *raw* sensory input. The term is used to generally refer to that which is used as the input to a bug identifier. The precise form of the experience varies with the approach taken. . 57, 198

experience-based testing A kind of testing that is directed by one's prior experience of how, when and where bugs tend to manifest. On their search for bugs an experienced human game tester may direct their search so as to find bugs more efficiently. *see also coverage-based testing* . 43, 151, 165, 198

explorative testing A kind of testing that is focused on finding bugs that are not anticipated in advanced (i.e. there was no test written that would otherwise uncover the bug). . 52, 53, 198

failure Observable unexpected or incorrect behaviour of a system "IEEE Standard Classification for Software Anomalies" 2010. *see also error, fault.* . 28, 198

fault A manifestation of an error in software "IEEE Standard Classification for Software Anomalies" 2010. *see also error, failure.*. 28, 198

fully observable An environment is fully observable if an agent is able to observe the full environment state. . 78, 198

fun-factor An important non-functional requirement in video games, is the game *fun* to play? *see also playability*. . 15, 198

g-mean g-mean is a measure used to select a threshold in imbalanced classification problems. It tries to balance sensitivity and specificity and is computed as $sqrt(Sensitivity * Specificity)$. . 109, 198

game balancing The problem of choosing difficulty settings or fine-tuning game mechanics so that a game appears fair, is not too easy or too difficult. . 15, 16, 198

game loop The game loop is a software pattern that manages the running of a video game. The game loop runs continuously through the lifetime of the game, processes user input and updates the game state and graphics. . 34, 198

game mechanic A rule that governs or guides the player's actions and how the game responds to them Boller 2013. For example, the ability to move forward, jump, communicate with other players or NPCs, etc. . 40, 68, 100, 198

group anomaly Observations that when taken individually are normal but are abnormal when taken as part of a group according to some well-defined notion of normality. Group anomalies are also known as *conditional*; *contextual*; *collective* (when a group is the complete dataset); *temporally extended* (for time-series) or *behavioural*. . 60, 198

guard A conditional statement or rule that checks the state of a program to ensure it is valid. Many programming languages have a means to express guards directly, e.g. the `assert` statement, or `try catch` blocks. Aside from appearing in the usual code (e.g. for validing input to a function) they also appear in test cases. Guards are a traditional form of testing and are hand written for purpose. . 16, 21, 32, 43, 198

inductive bias The set of implicit or explicit assumptions that determine how a machine learning model will make predictions outside its finite training data. . 47, 64, 198

integration testing A testing phase where units, modules or components of an application are tested together to ensure they work together as intended. . 15, 27, 198

intended behaviour The behaviour of a system that is desired as outlined by the totality of the requirements. By testing software, we are trying to convience ourselves that the observed behaviour matches the intended behaviour. This is done by comparing our interpretation of intended behaviour (which is refered to as expected behaviour) to what is observed during testing. In large projects the intended behaviour may not be known in its entirety to any individual team member, and in many cases it can only be communicated informally. *see also observed behaviour, expected behaviour, untestable*. . 7, 16, 18, 26, 28, 30, 43, 51, 56, 62, 70, 74, 160, 198

intended state An intended state is one that is part of the of the video game that wish to create. More specifically, for a given transition an intended state is part of the support of the intended transition distribution. . 72, 198

intervention An intervention is a kind of action that changes the causal mechanism that governs the outcome of a particular variable in some system of interest. Crucially, the action removes any dependence this variable has on the other variables in the system. . 136, 198

intial state A state at the start of a trajectory. In transition systems such as Markov Decision Processes (MDP) or Finite State Automata (FSA) a set of initial states is explicitly chosen. . 198

irreducible A Markov chain is irreducible if every state can be reached from every other state. . 78, 198

level Any space available to the player during the course of completion of an objective, otherwise called a map or stage. . 16, 66, 71, 123, 198

metamorphic testing A method for addressing the test oracle problem that uses the software itself as a test oracle. The method has developers specify *metamorphic relations* which are relations or constraints over sets of input/ouput pairs. For example, the relation `merge(L1,L2) == merge(L2,L1)` for a function `merge` which combines two lists and sorts the result. . 17, 23, 130, 131, 198

non-stationary An environment is non-stationary if its dynamics (transition function) change with time. The changes may occur during the agent's experience (i.e. in a single trajectory) or between trajectories, or both. . 69, 81, 198, 237

objective function A function that is optimised during learning. . 18, 74, 198

objectness The tendancy to discretize perception into objects. . 73, 198

observation An agent's sensory input, it is an approximation of the state of the environment in which the agent is currently situated. *see also observation space*. . 198

observation space The space of possible observations that an agent may have. For example: $[0-1]^{3\times H\times W}$ for RGB images. *see also observation*. . 198

observed behaviour The behaviour of a system or program that is seen during testing. *see also expected behaviour, intended behaviour*. . 28, 43, 198

partially observable An environment is partially observable if the agent does not observe the full state of the environment. . 59, 198

playability A non-functional requirement that concerns the overall quality of game play, is the game fun? Does it provide the correct level of challenge? Are the controls intuitive? Is the story engaging? Are the visuals attractive? See Fatta et al. 2019, see also Paavilainen 2020 for some discussion on playability as a functional requirement (as distinct from player experience). . 15, 26, 27, 50, 198

player controller The player controller is the implementation of a players actions. It defines the effects of each action on the environment. . 198

playtesting Testing a video game by actually playing it. Playtesting is done to evaluate game design and playability, and to find bugs. It may be done by internal testers, players, or software agents. . 15, 27, 36, 41, 42, 52, 53, 198

playthrough To playthrough a video game is to play it from start to finish. A playthrough might also refer to what is otherwise called a trajectory or episode. . 56, 198

point anomaly A single observation that is considered abnormal acoording to some well-defined notion of normality. . 60, 198

policy An agent's decision making procedure or mapping from observations to actions, denoted as $\pi : \mathcal{X} \rightarrow \mathcal{A}$. A policy may be stochastic in that, given an observation it assigns a probability to each action and chooses randomly according to this distribution. . 35, 198

positive recurrent A state in a Markov chain is positive recurrent if the expected number of steps taken before returning to the state is finite. . 78, 198

potential outcome Given an observation x, potential outcomes are the possible outcomes (next observations) that may be observed by taking a particular action. They represent the possible futures that may or may not be observed. Only one potential outcome is ever realized, or is said to be *factual*, the others are *counterfactual*. . 138, 198

reafference Any incoming sensory signal or effect on an agent's sensory system that is caused by the agent's own action. . 132, 198

regression When a feature worked previously but no longer works. *see also regression testing*. . 66, 198

regression testing A kind of testing that ensures that already existing software components don't break after an update.. 19, 21, 22, 27, 48, 52, 81, 88, 95, 159, 198

repeatability A test is repeatable if as part of a testing automation framework it can be run over and over without the need for human involvement. A human should only need to get involved if the test fails. . 29, 44, 198

reporting The process of documenting a bug and informing the relevant programmer, artist or other development team member of the bug's existence and requesting a fix.. 25, 28, 198

requirements Software requirements are a description of the features, functionalities and constraints on a system "IEEE Standard Glossary of Software Engineering Terminology" 1990. . 26, 198

reusability A test is reusable if it can be used in multiple projects or across versions of a game with no alterations. The most reusuable tests are those that are strongly decoupled from a games underlying implementation. *see also decoupling* . 7, 18, 19, 29, 44, 54, 56, 74, 81, 100, 101, 152, 153, 156, 159, 162, 198

smoke testing Testing that covers the most critical aspects of the software, usually relating to stability (i.e. does it crash with expected input?) or basic functionality. It is used to determine whether the software meets the minimum requirements before release.. 27, 198

state A collection of grounded variables that fully capture the environment at any particular moment. The state of a video game at time t is the collection of values in computer memory at that moment (this is sometimes called the RAM state). *see also state space* . 17, 198

state space The space of possible states that exist in an environment. In chess for example, this is all possible configurations of chessmen that can be reached from the starting position by making valid moves. *see also state* . 57, 198

static An environment is static if changes to its state are made only by a single agent. There are no other environmental processes (e.g. other agents) are acting to change the state. . 131, 198

stationary An environment is stationary if its dyanmics (or transition function) do not change with time. . 198

stationary distribution The stationary distribution of a Markov chain x satisfies the following equation $x = xP$ for some chosen initial distribution of states, where P is the transition matrix. . 78, 198

stochastic An environment is stochastic if for a given action in some state there is the possibility of transitioning to multiple different states. The transition to each possible next state has an associated probability. . 198

system testing A testing phase in which all of the components in an application are tested together to ensure the system as a whole works as intended. . 27, 198

systemic fault A systemic fault is similar to an endemic fault, but has wider reaching consequences or impacts many different transitions. An example might be a completely broken collision detection system. Objects that are supposed to be solid, are not. . 51, 72, 198

terminal state The state at the end of a trajectory. This may an absorbing state or state that only has an edge to one or more intial states. . 198

test oracle A test oracle determines whether the input-output pairs generated by executing a program are correct or intended. A human tester is an example of a test oracle. The *test oracle problem* simply refers to the challenge of specifying or implementing a test oracle, especially in cases where it is not entirely clear what constitues intended behaviour. *see also untestable software* . 16, 31, 33, 43, 46, 51, 69, 130, 149, 160, 163, 198

tester A tester's job is to uncover bugs in software, evaluate design criteria (e.g. fun-factor), or otherwise improve software by performing tests. They are verifying that the software works as intended. *see also intended behaviour* . 26, 198

trajectory A trajectory τ is a sequence of alternating observations (or states) and actions $x_0, a_0, x_1, a_1, \cdots, x_{T-1}, a_{T-1}, x_T$ that represent the experience an agent has when interacting with an environment. Trajectories might also be refered to as *episodes*, *runs* or . . 56, 198

transfer learning The problem of taking knowledge learned in one task and using it to solve another task. For example, learning about a certain bug in one game, and then using this knowledge to identify a simlar bug in another game. . 19, 198

unit testing A testing phase where the smallest parts or *units* of an application are independently tested.. 15, 26, 30, 198

untestable Untestable software is a class of software for which it is extremely difficult or impossible to write simple collections of rules that describe the intended behaviour. Examples include, compilers, search engines, machine learning systems, physics simulators and video games. . 16, 33, 45, 53, 57, 198

verification The process of checking that a particular bug has indeed been fixed.. 25, 29, 198

white box testing Any form of testing where the tester has knowledge of the underlying implementation of the program under test. *see also black box testing* . 27, 43, 198

zero-shot learning A problem formulation in machine learning where at test time a model encounters examples of a class that was not seen during training. For example, to recognise that a dog has ears, having never seen a dog before. . 49, 198

Glossary of Video Game Bugs

black screen A bug that leads the screen to be rendered black (or another block colour). It usually results from a total failure of the rendering system. The screen might otherwise quickly swap between failing to render (black) and rendering normally, this is usually a sign that there is an error with some part of the rendering process (e.g. in a shader that is trying to render something specific but failing). . 90, 91, 198, 220

camera clipping From certain view points, when the camera frustum is too close to an object, some of the object's geometry is culled, allowing the player to see through or inside the object. . 85, 88, 90, 198

freeze A bug that causes the running of the game (or its presentation) to hault temporarily. It might be called *lag* if subsequent frames are skipped, or a *hang* if the game continues without skipping. Freezing may be unrecoverable and require a restart, but is usually related to performance problems. . 22, 85, 91, 102, 109, 110, 150, 198

geometry clipping A collision bug that causes solid geometry to overlap in some significant way. . 22, 68, 85, 88, 90, 101, 198, 248

geometry corruption A geometry is corrupted when some of its vertices are incorrectly placed relative to the other vertices. This bug tends to happen during animations that depend on physics with flexible/dynamic geometries, but can also happen with static geometry, although this is less common. . 22, 48, 68, 85, 90, 121–123, 165, 198, 247, 263–265

high force When the player or other object suddenly experiences a high force which flings them at incredible speed in some direction. This can cause objects to clip through others due to the great speeds involved. . 22, 68, 121–123, 162, 198, 248, 263–265

invalid information access A bug in which the player is able to gain access to information they would otherwise not have. In extreme cases, this information may trivialise key challenges that the player faces (e.g. reveal the solution to a puzzle, location of a treasure, etc.). . 88, 89, 91, 100, 198

level of detail bug Under normal circumstances, as a player gets further from a detailed texture or object, the texture/geometry is down sampled to make rendering more efficient. As the player moves closer again, the texture/geometry should be upsampled to again show detail. If this fails to happen the world begins to look very low resolution. . 95, 198

missing object When an object is supposed to be present in the game but isn't. It may not be rendered, or may not exist entirely. *see also unintended object*. . 121–123, 198, 247, 263–265

perspective aliasing A kind of shadow artefact. It occurs when the mapping of pixels in view space to texels in the shadow map is not at a one-to-one ratio.. 48, 198

player out of bounds A bug where the player escapes the playable area (e.g. the bounds of a level). *see also terrain hole* . 22, 71, 73, 74, 78, 84, 85, 88, 92, 102, 121–123, 162, 198, 248, 263–265

progression bug Any kind of bug that hinders a player from progressing towards to the goal of a game. *see also stuck bug* . 198, 246

screen tearing Screen tearing happens when a monitor's refresh rate doesn't match the GPU frame rate. This leads to different parts of multiple frames being rendered to a single frame, leading to a *tearing* effect on screen. . 88, 90, 91, 121–123, 198, 247, 263–265

stuck bug A kind of progression bug where the player gets trapped in a certain area, for example, they fall into a hole and cannot jump out, or get stuck inside some geometry. Typically resetting from the last checkpoint will allow the player to continue as normal as long as they avoid the same trap, this is distinct from more general progression bugs which may never permit the player to progress beyond a certain point. . 16, 84, 85, 88–90, 100, 164, 198

terrain hole A kind of collision bug where the player falls through solid ground. *see also player out of bounds* . 38, 84, 88, 90, 92, 121–123, 161, 198, 248, 263–265

texture alpha bug A bug that renders a texture's alpha channel as something other than transparent. . 95, 198

texture corruption A texture may be rendered incorrectly / become corrupt for various reasons, for example, when texture offsets are incorrectly set or the UV map is incorrect. It can be difficult to distinguish texture corruption from other rendering related issues, such as level of detail, or lighting issues. . 22, 48, 88, 90, 95, 99, 121–123, 161, 198, 219, 246, 247, 263–265

texture missing Textures are assigned to objects for rendering, but if the texture is missing (e.g. the file doesn't exist) then the object won't be rendered correctly. A bright and noticable default texture is often used in such cases so that the issue can be easily identified by a human during testing. . 65, 68, 90, 99, 198

unintended object When an object is present in the game but isn't supposed to be. *see also missing object*. . 22, 121–123, 161, 198, 248, 263–265

unintended shortcut A bug that allows the player to take a shortcut to the goal. This usually involves them skipping some content and makes reaching the goal significantly easier. . 22, 102, 111, 121–123, 126, 161, 162, 198, 225, 248, 263–265

unresponsive A bug that prevents the player from taking action. The environment may evolve around them, but any player input is effectively ignored. . 21, 129, 150, 156, 163, 198

Z-clipping Rendering pipelines often have layers which are queued and rendered in order. Each layer contains a collection of objects to render. Objects that are rendered in the first layers are replaced or blended with pixels in later layers. If an object is placed in the wrong layer, it may be rendered on top of others which can lead to some strange visual effects. . 90, 198

Z-fighting Z-Fighting happens when two surface geometries have the same depth. The renderer does not know which to show first and this results in a mixing of textures from the two surfaces. A flickering effect may also occur when a player shifts their view. . 85, 90, 121–123, 161, 162, 198, 247, 263–265

Appendix A

Technical Appendix

This chapter of the appendix contains technical details for the main thesis chapters, including derivations, algorithm details, reproducibility information and additional experiments. Sections are organized by chapter and contain sections that discuss results, reproducibility, and additional experiments. Numerical results are presented at the end of the appendix, links to the relevant figures have been organized in the results section for each chapter. This thesis also has supplementary material which contains datasets and code for reproducing experiments that are specific to the thesis (i.e. not included as part of any supplementary material for the published papers). This can be found here[1].

---

[1] https://github.com/BenedictWilkins/thesis-reproduce

A.1 Chapter 4

A.1.1 Reproducibility Checklist

Compute Requirements

All experiments were run on a single 12 cpu core 32gb RAM machine with an NVIDIA GeForce RTX 2070 GPU. Experiments can quickly and easily be reproduced on a mid to high-end personal computer.

Code Dependencies

Code is written in python 3.8 and models are developed with pytorch and run with CUDA. Code and a full list of dependencies can be found in the supplementary material.

Data Dependencies

The dataset used was generated from the `World-v0` environment. Unintentional updates were introduced by manually modifying the `World-v0` environment in the Unity editor, or by modifying the data collection code. There is currently no support for introducing these kinds of changes in an automated fashion. Links and further details regarding data can be found in the supplementary material. The dataset (and pre-trained model) can also be found here[2].

Randomness & Seeding Environments

Random seeds are not provided as they are not required to obtain equivalent performance. Datasets are available where required.

Evaluation metrics

We showed precision-recall curves and confusion matrices for each of the 5 bugs. They were computed given 10 episodes for each bug and 10 (unseen) normal episodes using the average episode classification error as a normality score. The same 10 normal episodes were used for each bug. These metrics are sufficient to show that the model is working as expected. No further details are given as we are not expecting any comparison to be made.

---

[2] https://figshare.com/articles/dataset/WOB_Texture_Corruption/22210660

Algorithm Stability & Hyperparameters

The experiment was performed without changes to the initial hyperparameter settings. They were set based on our experience performing similar classification tasks. The training procedure is likely very stable as the task is a straightforward classification.

A.1.2 Experiment Details

Regression Testing World-v0

We trained a single model to classify the texture corruption bug. The training dataset contained ∼20k unique observations. All duplicate observations were removed to balance the training data. The model took in the order of minutes to train on our hardware. The training duration was stopped arbitrarily at 100 epochs.

```
==========================================================================================
Layer (type:depth-idx)               Output Shape              Param #
==========================================================================================
Classifier                           --                        --
├─Sequential: 1-1                    [2, 32, 6, 6]             --
│    └─Conv2d: 2-1                   [2, 8, 39, 39]            1,184
│    └─LeakyReLU: 2-2                [2, 8, 39, 39]            --
│    └─Conv2d: 2-3                   [2, 16, 17, 17]           6,288
│    └─LeakyReLU: 2-4                [2, 16, 17, 17]           --
│    └─Conv2d: 2-5                   [2, 32, 6, 6]             25,120
│    └─LeakyReLU: 2-6                [2, 32, 6, 6]             --
├─Sequential: 1-2                    [2, 1]                    --
│    └─Linear: 2-7                   [2, 1024]                 1,180,672
│    └─LeakyReLU: 2-8                [2, 1024]                 --
│    └─Linear: 2-9                   [2, 1024]                 1,049,600
│    └─LeakyReLU: 2-10               [2, 1024]                 --
│    └─Linear: 2-11                  [2, 1]                    1,025
==========================================================================================
Total params: 2,263,889
Trainable params: 2,263,889
Non-trainable params: 0
Total mult-adds (M): 13.51
==========================================================================================
Input size (MB): 0.17
Forward/backward pass size (MB): 0.32
Params size (MB): 9.06
Estimated Total Size (MB): 9.54
==========================================================================================
```

```
learning rate = 0.0005,
batch_size = 256
optim = torch.nn.Adam,
criterion = torch.BCEWithLogitsLoss
epochs = 100
training_data_size = 19389
unique_only = True
bug_ratio = 0.62
```

A.1.3 Additional Experiments

Action Classification

While regression testing the `World-v0` environment in section 4.2 a simple model was trained to classify the texture corruption bug. The classification error was used as a normality score. This worked reasonably well for identifying the unintended platform updates. The assumption here is that the model will perform badly when observations are Out-Of-Distribution (OOD), but it is not obvious that this will be the case. If an unintentional update is subtle (in that the novel features were not important for the classification task) then the model may still perform well and the score will not be useful.

In a later experiment we tried a similar approach in an attempt to identify the bugs found in `World-v0` environment. A classification target that is analogous to the bug mask (which of course would not be present in a real game) is the action that the agent takes. A classifier was trained to classify the action taken given a pair of observations. (i.e $x_t \rightarrow x_{t+1}$). Other than for the simplest bugs (e.g. black screen) this approach performed very poorly. This is because the bugs do interfere greatly with the effect of the agent's action, which in this environment is to globally shift pixels.

A.2 Chapter 5

A.2.1 Results

Numerical results for the experiments presented in Chapter 5: Contrastive Learning for Automated Bug Identification can be found in the figures listed below. The figures themselves can be found at the end of the appendix. There are no numerical results for experiments (1) Identifying Unintended Shortcuts and (2) Identifying Systemic Bugs.

(3) Atari Results	
Experiment	**Figure Ref.**
Beam Rider	D.1
Breakout	D.2
Enduro	D.3
Pong	D.4
Qbert	D.5
Sea Quest	D.6
Space Invaders	D.7

(4) PED Results	
Experiment	**Figure Ref.**
Ped. 1 & 2	D.8

(5) WOB Results	
Model	**Figure Ref.**
S3N	D.9
Contrastive	D.10
Hybrid	D.11

A.2.2 Reproducibility Checklist

Compute Requirements

All experiments, with the exception of the Atari experiments were run on a single 12 cpu core 32gb RAM machine with an NVIDIA GeForce RTX 2070 GPU. The Atari experiments were initially run on AWS g4dn.8xlarge instances, but were later ported to the RTX hardware.

Code Dependencies

Code is written in python 3.8 and models are developed with pytorch and run with CUDA. Code for the Atari experiments can be found with the supplementary material of the original paper (Wilkins et al. 2020) here[3]. Code for the other experiments can be found in the supplementary material of this thesis.

Data Dependencies

1. Identifying Shortcuts: uses the `Explorer-v0` environment, which is an alias for the `bugged-explorer-v0` environment and can be found here[4]. A static dataset is available as part of the supplementary material.

---

[3] https://github.com/BenedictWilkins/S3N

[4] https://github.com/BenedictWilkins/gym-explorer commit hash: 83f8c9d7a3f420db40379fee1a1d9b13c38cc12e

2. Identifying Systemic Bugs: uses the `Alone-v0` environment which can be found in here[5]. A static dataset is available as part of the supplementary material of this thesis.

3. Atari: data can be found as part of the supplementary material of our paper (Wilkins et al. 2020) here[6].

4. Video Surveillance: the UCSD Ped. 1 and 2 datasets are publicly available.

5. WOB: uses the `Maze-v1` environment that is part of the WOB platform. The data used in experiments can be found as part of the supplementary material and also here[7].

Randomness & Seeding Environments

Random seeds are not provided as they are not required to obtain equivalent performance. Datasets are available where required.

Evaluation Metrics

Evaluation metrics go beyond accuracy and other simple measures. There is some discussion as to which measures are most applicable for the experiments performed. See section A.2.4 below. In experiment (3) the measures presented by other works are also used, namely Receiver Operator Characteristic AUC and Equal Error Rate (EER). A more extensive set of performance measures is given in the numerical results (3) to allow for better comparison with possible future work (see section A.2.1).

Algorithm Stability & Hyperparameters

Hyperparameter choices for each experiment are presented in the sections below. The simpler experiments (1) and (2) are fairly robust to these choices. In (3) experiments were repeated numerous times with different hyperparameter settings, the most impactful was the embedding dimension. Other hyperparameters had little impact under reasonable settings.

The results in (4) were very sensitive to hyperparameter choices, particularly to the image patch size. They were also sensitive to the choice of window size used to average over scores. This was chosen to be the best among at least ten trials with different choices.

---

[5] https://github.com/BenedictWilkins/gym-pygame commit hash: b5616d4713f46ee6efaa13703e2ec899bc4a3a7b
[6] https://www.kaggle.com/datasets/benedictwilkinsai/atari-anomaly-dataset-aad
[7] https://www.kaggle.com/datasets/benedictwilkinsai/wob-mazev1-dataset

In (5) given the poor performance, some effort was made to improve the results, this led to many runs with different hyperparameters settings. The choice of hyperparameters did not impact the results greatly, despite our best efforts performance was not significantly improved for any particular setting. Changing the network architecture also had almost no impact, various architectures were tried, including MLPs, AlexNets, and vision transformers (ViT) (Dosovitskiy et al. 2021). After some experimentation we settled on an AlexNet architecture as it was reasonably quick to train and offered slightly better performance than the MLP architecture. The ViT architecture seemed to produce nonsense results, we did not have time to investigate this further. The lack of improvement pointed to a more systematic problem with the approach which was discussed at some length in the chapter (see also A.2.5). The same architecture was used for the other two contrastive learning approaches in an attempt to keep the experiments fair (the AlexNet was not optimized for performance with the S3N objective). Different combinations of data augmentation were tried for these approaches, we settled on those that gave the best overall performance for the bugs collectively. The augmentations were taken from the `torchvision` image transform library. No other hyperparameter tuning was performed for these approaches.

A.2.3 Experiment Details

Experiment (1): Unintended Shortcuts

The model trained here is a simple one. The model could be trained in a matter of minutes on our hardware before reaching perfect accuracy. Training was stopped when the embedding looked reasonable (it was visualized as a graph after each training epoch), this was at around 50 epochs. Approximately 50k observations were collected using a uniform random policy for training. As the environment is small with no difficult puzzles there is no exploration issue, the dataset covers the environment's observation space.

```
=========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
=========================================================================================
Model                                    --                        --
├─Sequential: 1-1                        [1, 2]                    --
│    └─Linear: 2-1                       [1, 128]                  32,896
│    └─LeakyReLU: 2-2                    [1, 128]                  --
│    └─Linear: 2-3                       [1, 128]                  16,512
│    └─LeakyReLU: 2-4                    [1, 128]                  --
│    └─Linear: 2-5                       [1, 128]                  16,512
│    └─LeakyReLU: 2-6                    [1, 128]                  --
│    └─Linear: 2-7                       [1, 2]                    258
=========================================================================================
Total params: 66,178
Trainable params: 66,178
Non-trainable params: 0
Total mult-adds (M): 0.07
=========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.26
Estimated Total Size (MB): 0.27
=========================================================================================
```

```
env_id = "explorer-bugged-v0"
optimizer = torch.optim.Adam
margin = 0.05
learning rate = 0.005
batch_size = 64
epochs = 50
training_dataset_size = 50k
policy = uniform random
trajectories = 1
latent_shape = (2,)
input_shape = (1,16,16)
random_seed = NA
```

Experiment (2): Systemic Bugs

Again the model is simple and can be trained quickly. Enough data (20k observations) to cover the environment's observation space was collected by a uniform random policy. Exploration is again easy. The model was trained as in experiment (1) until the embedding looked reasonable.

```
----------------------------------------------------------------
Layer (type)            Output Shape       Param #
================================================================
Linear-1                [-1, 256]       200,960
LeakyReLU-2                [-1, 256]            0
Linear-3                [-1, 256]        65,792
LeakyReLU-4                [-1, 256]            0
Linear-5                  [-1, 2]           514
================================================================
Total params: 267,266
Trainable params: 267,266
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 1.02
Estimated Total Size (MB): 1.03
----------------------------------------------------------------
```

```
env_id = "pygame/Alone-v0"
batch_size = 256
optimizer = torch.nn.Adam
learning_rate = 0.0005
action_shape = (4,)
state_shape = (1,28,28)
latent_shape = (2,)
epochs = 20
trajectory_length = 1000
dataset_size = 20k
margin = 0.2
policy = uniform_random
```

Experiment (3): Atari

The models used here are rather larger but in hindsight need not be. They were initially trained using AWS g4dn.8xlarge machines. They were later retained using our NVIDIA GTX setup, it is these models for which results are presented. The models were trained for around an hour on the *RAW* partition of the AAD dataset. Very little hyperparameter tuning was done. Experiments were run for different embedding dimensions (2,8,64,128,256). After 2 all gave similar results. The best results among these runs are those that are presented.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
CNet2                                    --                        --
├─Conv2d: 1-1                            [1, 16, 104, 79]          784
├─LeakyReLU: 1-2                         [1, 16, 104, 79]          --
├─Conv2d: 1-3                            [1, 32, 101, 76]          8,224
├─LeakyReLU: 1-4                         [1, 32, 101, 76]          --
├─Conv2d: 1-5                            [1, 64, 98, 73]           32,832
├─LeakyReLU: 1-6                         [1, 64, 98, 73]           --
├─Linear: 1-7                            [1, 64]                   29,302,848
==========================================================================================
Total params: 29,344,688
Trainable params: 29,344,688
Non-trainable params: 0
Total mult-adds (M): 333.75
==========================================================================================
Input size (MB): 0.40
Forward/backward pass size (MB): 6.68
Params size (MB): 117.38
Estimated Total Size (MB): 124.46
==========================================================================================
```

```
batch_size = 128
episodes = 20
epochs = 12
embedding dimension = 64 or 256
learning_rate = 0.0005
margin = 0.2
input_shape': (3, 210, 160)
```

Experiment (4): PED

Many hyperparameter settings were tried in pursuit of better performance. After this experiment it was clear that content blindness was a serious problem with our approach. Using image patches rather than the full image gave a clear boost in performance. Given the surprisingly good performance of the simple distance classifier and the model architecture we settled on here, it is clear that one can reach “good” performance just by looking at the pixel dynamics. The approaches that report better results must go beyond this, which in this instance S3N is not capable of doing as the training data is quite limited. It simply does not have the right inductive biases for the full task.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      --                        --
├─Sequential: 1-1                        [2, 1, 32, 32]            --
│    └─ResBlock2D: 2-1                   [2, 1, 32, 32]            --
│    │    └─Conv2d: 3-1                  [2, 128, 32, 32]          1,280
│    │    └─ReLU: 3-2                    [2, 128, 32, 32]          --
│    │    └─Conv2d: 3-3                  [2, 1, 32, 32]            129
│    └─LeakyReLU: 2-2                    [2, 1, 32, 32]            --
├─Linear: 1-2                            [2, 1024]                 1,049,600
├─Linear: 1-3                            [2, 1024]                 1,049,600
├─Linear: 1-4                            [2, 8]                    8,200
==========================================================================================
Total params: 2,108,809
Trainable params: 2,108,809
Non-trainable params: 0
Total mult-adds (M): 7.10
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 2.15
Params size (MB): 8.44
Estimated Total Size (MB): 10.59
==========================================================================================
```

```
batch_size = 512
learning_rate = 0.0025
latent_shape = 8
epochs = 20
margin = 0.05
embedding_metric = Cosin_distance
optimizer = torch.optim.Adam
patch_size = 32
patch_stride = 8
```

Experiment (5): WOB

The encoder forming the Siamese network in each approach is given below. Other architectures were tried (see section A.2.2). Training took less than one hour on our hardware in each case (longer for the two approaches that required data augmentation). We did not experiment extensively with different augmentations, it is possible that better performance could be reached with different augmentations.

We saw that performance on the unintended shortcut bug was relatively poor and believe this is due to a problem with labelling. A bug was labelled as such only when the agent’s camera clipped through the wall. Until that moment, the agent approaches the wall and its body begins to clip through it, these states close to the wall would normally be unreachable but are not labelled as such. This highlights the difficulty with labelling certain bugs.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
AlexNet32                                --                        --
├─Conv2d: 1-1                            [1, 16, 26, 26]           800
├─LeakyReLU: 1-2                         [1, 16, 26, 26]           --
├─Conv2d: 1-3                            [1, 32, 20, 20]           25,120
├─LeakyReLU: 1-4                         [1, 32, 20, 20]           --
├─Conv2d: 1-5                            [1, 64, 14, 14]           100,416
├─LeakyReLU: 1-6                         [1, 64, 14, 14]           --
├─Conv2d: 1-7                            [1, 128, 8, 8]            401,536
├─LeakyReLU: 1-8                         [1, 128, 8, 8]            --
├─Conv2d: 1-9                            [1, 256, 2, 2]            1,605,888
├─LeakyReLU: 1-10                        [1, 256, 2, 2]            --
├─Flatten: 1-11                          [1, 1024]                 --
├─Linear: 1-12                           [1, 1024]                 1,049,600
├─LeakyReLU: 1-13                        [1, 1024]                 --
├─Linear: 1-14                           [1, 8]                    8,200
==========================================================================================
Total params: 3,191,560
Trainable params: 3,191,560
Non-trainable params: 0
Total mult-adds (M): 63.45
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.37
Params size (MB): 12.77
Estimated Total Size (MB): 13.14
==========================================================================================
```

```
env_id = "WOB/Maze-v1"
optimizer = torch.optim.Adam
margin = 0.2
learning rate = 0.0005
batch_size = 128
metric = L22
training_dataset_size = 60k
policy = random navigation (WOB built-in)
trajectories = 3
latent_shape = (8,)
input_shape = (1,32,32)
random_seed = NA
epochs.S3N = 10
epochs.Contrastive = 20
epochs.Hybrid = 20
augmentation.S3N = NA
augmentation.Contrastive = [
  ColorJitter(contrast=0.5, brightness=0.5, saturation=0.5),
  RandomRotation(degrees=(5, 355)),
  GaussianBlur(kernel_size=3)
]
augmentation.Hybrid = [
  RandomPairApply(*augmentation.Contrastive),
  RandomSwap()
]
```

A.2.4 Performance Measures

Earth Mover Distance (EMD)

On occasion, for example when attempting to identify systemic bugs (see section 5.3.2), we may want to compare global distance statistics. The distribution of distances $\Delta(\tau^*)$ for some normal reference trajectory τ^* might be compared with that of new (possibly buggy) trajectories. The earth mover distance (EMD), sometimes referred to as the Wasserstein-1 metric, can be used for this purpose. This metric has some desirable properties over say, the Kullback–Leibler divergence: it is symmetric, is defined for distributions with differing support, and can be computed efficiently for histograms of equal bin size (which is the setting we find ourselves in).

Area Under Receiver Operator Characteristic Curve (AUC-ROC)

The AUC-ROC score can be interpreted as the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. Positive and negative correspond to abnormal and normal respectively. AUC-ROC has some issues with severely imbalanced data. In our setting data may be balanced or imbalanced, AUC-ROC is used where suitable.

Area Under Precision-Recall Curve (AUC-PR)

The AUC-PR is better suited to imbalanced data, and is of particular importance when we care more about identifying positives (abnormality). It gives more importance to finding bugs than normality, which is generally desirable in our setting.

A Note on the Uniform Distance Statistic (UDS)

In the identification of abnormal observations or transitions, distance will be used as a score to produce a global ranking. It is therefore important that distances are globally consistent (they have the same scale), which may not be so if the approximation is bad. We do not want to be in a situation where distances for certain observations are vastly greater than for others since this may lead to misclassifications. When the graph is known, the local constraints tend to lead to a globally coherent score. It is less clear that this tendency for global coherence will persist in the approximate regime. The uniform distance statistic (UDS) was developed in our paper (Wilkins et al. 2020) with the intention of measuring the degree to which the embedding respects global consistency.

The original statistic, which was defined as the standard deviation of residual distance $max(\sigma(\Delta_1 - \alpha), 0)$, did not take into account the skew of the distribution or zero cut-off. Intuitively, we are interested in the right tail of the distance distribution, and specifically in how much these values differ from the average. Using the standard deviation seems like a natural choice, but it will be biased by the skew. One solution is to assume a log-normal distribution. A log transform of the distances will then give a normal distribution and the standard deviation may be interpreted as usual. Empirically we have found that the distance distributions are approximately log-normal, making this trick applicable (although a test should still be done to validate the assumption). Self-transitions cause problems if we are using a proper metric so zero distances are removed from the computation where required. The UDS is computed as follows:

$$UDS(\tau) = \sigma(log(\Delta(\tau))) \tag{A.1}$$

This measure should be used in place of the one presented in the original paper, which upon further investigation is flawed. Although the measure is not presented in our results, it was useful in our initial exploration of S3N.

A.2.5 Additional Experiments

Content Blindness

Content blindness is an issue that was observed in the surveillance video experiment and then later in the WOB experiments. The problem in essence is that S3N tends to learn the dynamics of the environment without learning about the content of the observation. The issue is illustrated in the Alone-v0 environment in Fig. A.1. The problem needs further investigation, one direction to explore may be to introduce additional

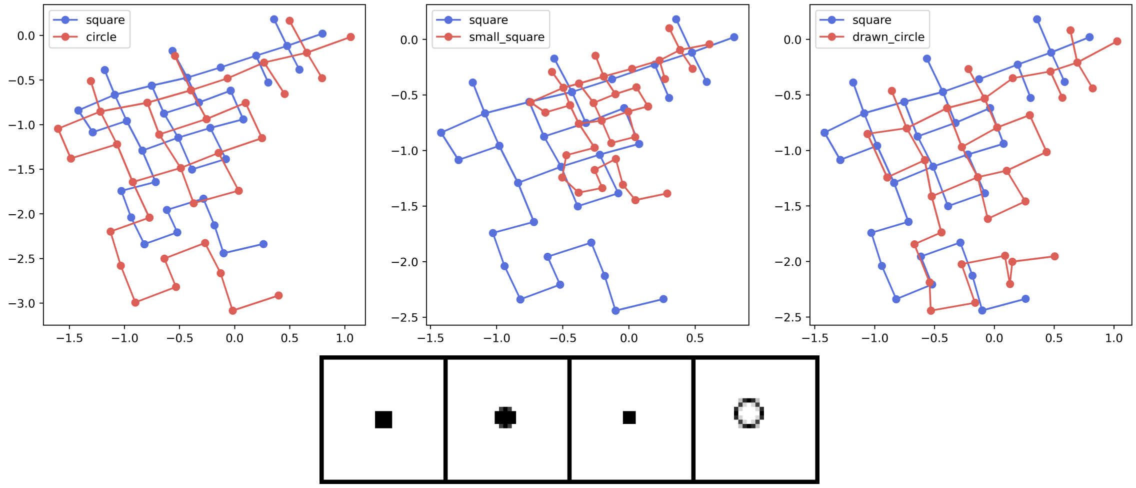


Figure A.1: Demonstration of content blindness in S3N. In the environment the player is represented as a square that moves in the cardinal directions. If the square is replaced with one of the shapes shown above then content blindness manifests. This may manifest in different ways for different environments (e.g. in the experiments with `Explorer-v0`).

data augmentation (as was done in the WOB experiments).

Embedding Dimensions of Common Graphs

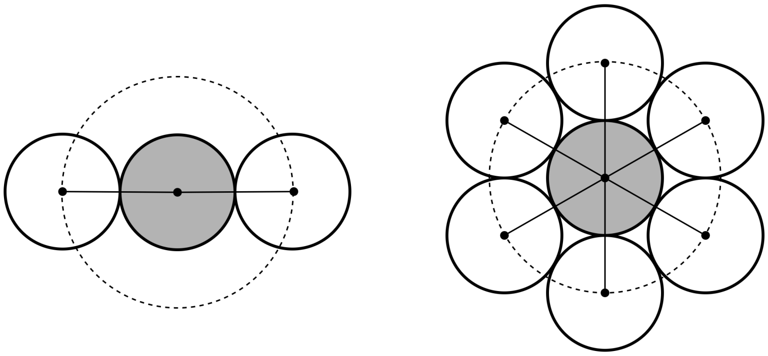
Examples of graph embeddings for graphs beyond those presented in chapter 5. In Fig. A.3 we are checking the condition $D_\theta(x, x^+) + \alpha < D_\theta(x, x^-)$ in each embedding dimension. The intervals shown in each plot show the min/max positive and negative distances over the complete graph.

Sphericity

Embedding graphs in Euclidean space provides a means to study graphs through geometry (Reiterman et al. 1989). There are some theoretical results for graph embeddings, in settings where nodes do not have associated data (as we do). These results give bounds on the number of dimensions required to construct embeddings with certain properties, such as the one we consider $D(x, x^+) \leq D(x, x^-) + \alpha$. The *sphericity* $sph(G)$ of a graph G, refers to the minimum dimension required to construct a graph embedding with the following closely related properties[8] $D(x, x^+) \leq \beta$, $D(x, x^-) > \beta$. These properties impose a global structure on the embedding, which is slightly stricter than the vertex relative constraint imposed by triplet loss. A

---

[8] recall that $x^+ \in N(x)$ and $x^- \notin N(x)$.



(a) $d = 1, k = 2$ (b) $d = 2, k = 6$

d	k
1	2
2	6
3	12
4	24
5	40-44
6	72-78
7	126-134
8	240

(c)

Figure A.2: Kissing numbers visualized in (a) 1 and (b) 2 dimensions. Known kissing numbers for dimensions 1-8.

lower bound sphericity is derived in (Reiterman et al. 1989) as

$$sph(G) \geq \frac{log_2 \alpha(G)}{log_2(2r(G)+1)}$$

where $a(G)$ is the *maximal independent set size of* G and $r(G)$ *is the radius of* G, see the work for further details.

Star-Graphs & Kissing Number

Another set of theoretical results concerning the special case star-graphs is related to the sphere packing problem, which in general is unsolved. The *kissing number* is defined as the maximum number of non-overlapping unit spheres arranged such that they touch a common unit sphere, see Fig. A.2. The kissing number k for dimension d gives us the largest star graph (k-star) that can be embedded in this dimension. Of course, in Fig. A.3 we see that S3N respects these limits, requiring 3 dimensions to embed a 7-star graph, and 8 for a 64-star graph. Note that the latter used many more dimensions than needed, a 64-star graph can be embedded in $d = 6$.

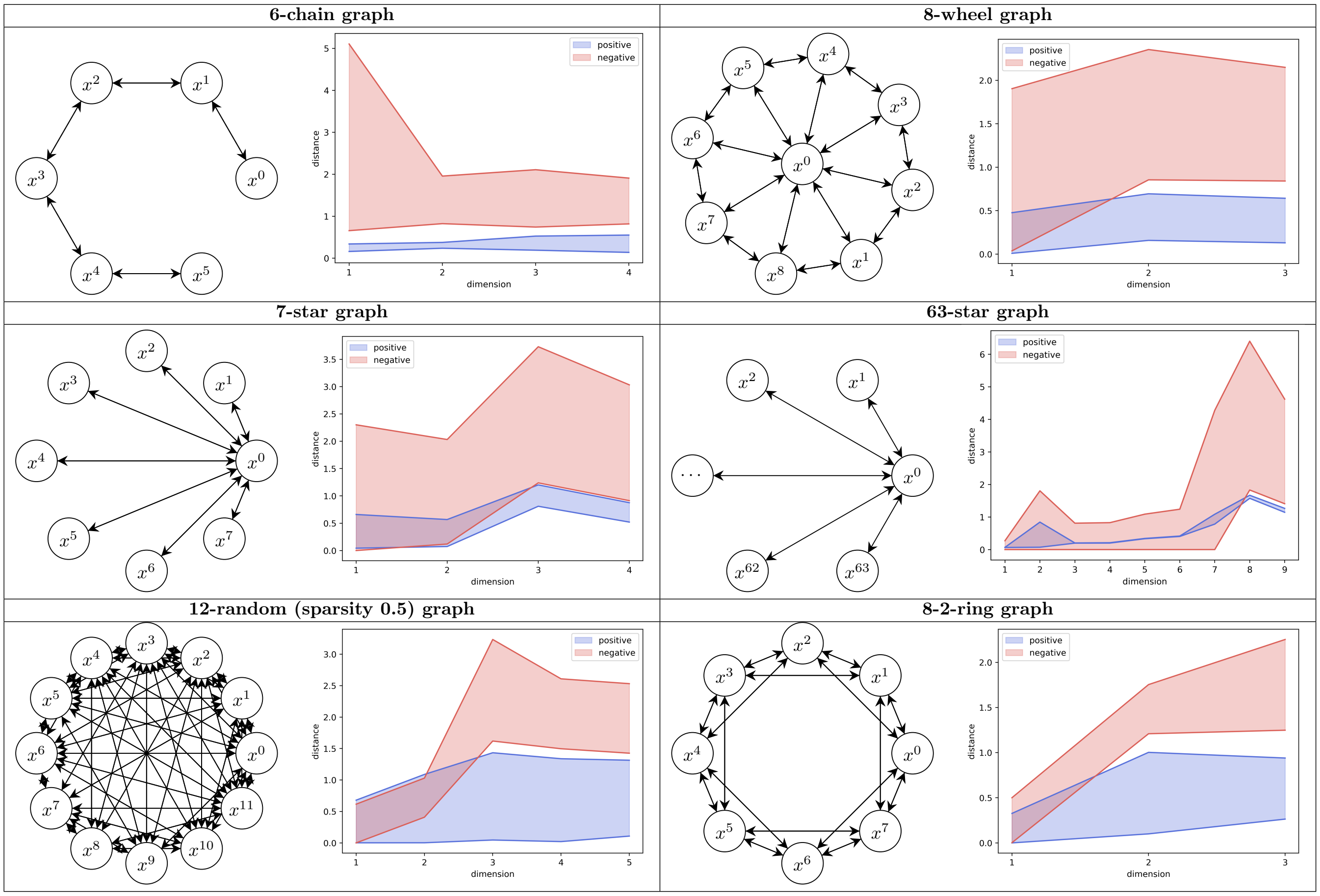


Figure A.3: Examples of the embedding dimension required for different graphs.

A.3 Chapter 6

A.3.1 Reproducibility Checklist

Compute Requirements

All experiments were run on a single 12 cpu core 32gb RAM machine with an NVIDIA GeForce RTX 2070 GPU. Experiments can quickly and easily be reproduced on a mid to high-end personal computer.

Code Dependencies

Code is written in python 3.8 and models are developed with pytorch and run with CUDA. Other dependencies are specified in the `setup.py` file in the code repository.

Randomness & Seeding Environments

We have not provided the seeds used to generated data in each environment (e.g. the seed for the random policy used) or for SGD as all results can be reproduced easily without them.

Evaluation metrics

The results are difficult to evaluate numerically. Our work is not a *delta*, so there are no measures of performance to compare to from previous/related work.

Even in cases where ground truth reafference/exafference is available (Cartpole and Freeway) any metric that compares estimates to the ground truth would not necessarily give a good indicator of a model's ability to disentangle the two. Instead, it may just reflect whether the model was able to learn an accurate representation of the environment dynamics (this might happen if a model is not expressive enough to capture the true dynamics). We have tried to set up experiments in such a way that a single example of the disentanglement presented in the paper would give a good idea of the performance by *eye-balling*. Granted this is not ideal, but the samples presented in the paper are representative (i.e. not cherry picked for accuracy, only for clarity). Videos of the disentanglement for full runs through different environments are available in the supplementary files.

Algorithm Stability & Hyperparameters

Our algorithm is very robust to hyperparameter choice and given their limited number, we have not presented any detailed discussion of them in the main body of the paper. Our initial selection of Adam with a learning rate (0.0005) was kept in all runs. Batch sizes were varied only to allow training with limited GPU memory, and did not have any noticeable impact performance.

The most impactful hyperparameter choice was the neural network architecture used. For Cartpole we settled on a simple MLP network after only a few runs which each took only a few minutes on our hardware. For Freeway and Artifical Ape, we tried various architectures, including a simple AlexNet-like architecture, and after a handful of runs (less than 10) settled on the UNet architecture presented in the work. The simpler architectures we tried did not capture the environment dynamics well, or were slower to train. A single training run took in the order of tens of minutes with our setup.

We tried using batch normalisation in a few runs to the detriment of the exafference estimates as they became nonsensical. This needs further investigation.

A.3.2 Experiment Details

Experiment (1): Cartpole

The model is a 4 layer MLP with tanh activation with approx. 500k parameters. The action is represented as a one-hot vector and is concatenated with the observation in the initial network layer. The model was trained for 100 epochs on 100k examples that were collected using a uniform random policy and stored. The Adam optimiser with a learning rate of 0.0005 was used.

```
==========================================================================================
Layer (type:depth-idx)               Output Shape           Param #
==========================================================================================
CartPoleNet                          [2, 4]                 --
├─Sequential: 1-1                    [2, 4]                 --
│    └─Linear: 2-1                   [2, 512]               4,096
│    └─Tanh: 2-2                     [2, 512]               --
│    └─Linear: 2-3                   [2, 512]               262,656
│    └─Tanh: 2-4                     [2, 512]               --
│    └─Linear: 2-5                   [2, 512]               262,656
│    └─Tanh: 2-6                     [2, 512]               --
│    └─Linear: 2-7                   [2, 4]                 2,052
==========================================================================================
Total params: 531,460
Trainable params: 531,460
Non-trainable params: 0
Total mult-adds (M): 1.06
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.02
Params size (MB): 2.13
Estimated Total Size (MB): 2.15
==========================================================================================
```

```
state_shape = (4,)
action_shape = (3,)
latent_shape = (512,)
epochs = 100
learning_rate = 0.0005
criterion = torch.nn.MSE
optimiser = torch.optim.Adam
action_conditional = cat_first
exafferent_stop_grad = True
noop_index = 0
```

Experiment (2): Atari Freeway

The model trained follows the UNet architecture (Ronneberger et al. 2015). We found that this architecture worked better than those without residual connections. Actions are embedded using a single linear layer, then introduced into the UNet by an element-wise product with the output of the encoder portion of the network. The network has approx. 33M parameters. It was trained for 50 epochs on 5k observations/actions collected using a uniform random policy and stored. The Adam optimiser with a learning rate 0.0005 was used.

This environment is not Markovian, which leads to some issues with prediction at certain points. Namely, when the chicken gets hit by a car, the agent's actions become ineffective and the agent is moved backward some steps. The model will predict the usual reafferent effect if the forward/backward action is taken as there is no indication that the chicken has previously been hit in the current observation. This can be resolved by using frame stacking, or by introducing some kind of memory (e.g. with LSTM). The issue is not one that is relevant for demonstrating the effectiveness of our approach.

```
===================================================================================================
Layer (type:depth-idx)                     Output Shape            Param #
===================================================================================================
UNet                                       [2, 3, 84, 84]          --
├─DoubleConv: 1-1                          [2, 32, 84, 84]         --
│    └─Sequential: 2-1                     [2, 32, 84, 84]         --
│    │    └─Conv2d: 3-1                    [2, 32, 84, 84]         864
│    │    └─Identity: 3-2                  [2, 32, 84, 84]         --
│    │    └─LeakyReLU: 3-3                 [2, 32, 84, 84]         --
│    │    └─Conv2d: 3-4                    [2, 32, 84, 84]         9,216
│    │    └─Identity: 3-5                  [2, 32, 84, 84]         --
│    │    └─LeakyReLU: 3-6                 [2, 32, 84, 84]         --
├─Down: 1-2                                [2, 64, 42, 42]         --
│    └─Sequential: 2-2                     [2, 64, 42, 42]         --
│    │    └─MaxPool2d: 3-7                 [2, 32, 42, 42]         --
│    │    └─DoubleConv: 3-8                [2, 64, 42, 42]         55,296
├─Down: 1-3                                [2, 128, 21, 21]        --
│    └─Sequential: 2-3                     [2, 128, 21, 21]        --
│    │    └─MaxPool2d: 3-9                 [2, 64, 21, 21]         --
│    │    └─DoubleConv: 3-10               [2, 128, 21, 21]        221,184
├─Down: 1-4                                [2, 256, 10, 10]        --
│    └─Sequential: 2-4                     [2, 256, 10, 10]        --
│    │    └─MaxPool2d: 3-11                [2, 128, 10, 10]        --
│    │    └─DoubleConv: 3-12               [2, 256, 10, 10]        884,736
├─Down: 1-5                                [2, 512, 5, 5]          --
│    └─Sequential: 2-5                     [2, 512, 5, 5]          --
│    │    └─MaxPool2d: 3-13                [2, 256, 5, 5]          --
│    │    └─DoubleConv: 3-14               [2, 512, 5, 5]          3,538,944
├─DiagLinear: 1-6                          [2, 512, 5, 5]          --
│    └─Flatten: 2-6                        [2, 12800]              --
│    └─Linear: 2-7                         [2, 1024]               13,108,224
│    └─Linear: 2-8                         [2, 1024]               4,096
│    └─Linear: 2-9                         [2, 12800]              13,120,000
│    └─View: 2-10                          [2, 512, 5, 5]          --
├─Up: 1-7                                  [2, 256, 10, 10]        --
│    └─ConvTranspose2d: 2-11               [2, 256, 10, 10]        524,544
│    └─DoubleConv: 2-12                    [2, 256, 10, 10]        --
│    │    └─Sequential: 3-15               [2, 256, 10, 10]        1,769,472
├─Up: 1-8                                  [2, 128, 21, 21]        --
│    └─ConvTranspose2d: 2-13               [2, 128, 20, 20]        131,200
│    └─DoubleConv: 2-14                    [2, 128, 21, 21]        --
│    │    └─Sequential: 3-16               [2, 128, 21, 21]        442,368
├─Up: 1-9                                  [2, 64, 42, 42]         --
│    └─ConvTranspose2d: 2-15               [2, 64, 42, 42]         32,832
│    └─DoubleConv: 2-16                    [2, 64, 42, 42]         --
│    │    └─Sequential: 3-17               [2, 64, 42, 42]         110,592
├─Up: 1-10                                 [2, 32, 84, 84]         --
│    └─ConvTranspose2d: 2-17               [2, 32, 84, 84]         8,224
│    └─DoubleConv: 2-18                    [2, 32, 84, 84]         --
│    │    └─Sequential: 3-18               [2, 32, 84, 84]         27,648
├─OutConv: 1-11                            [2, 3, 84, 84]          --
│    └─Conv2d: 2-19                        [2, 3, 84, 84]          99
├─Tanh: 1-12                               [2, 3, 84, 84]          --
===================================================================================================
Total params: 33,989,539
Trainable params: 33,989,539
Non-trainable params: 0
Total mult-adds (G): 2.91
===================================================================================================
Input size (MB): 0.17
Forward/backward pass size (MB): 34.56
Params size (MB): 135.96
Estimated Total Size (MB): 170.69
===================================================================================================
```

```
state_shape = (3,84,84)
action_shape = (3,)
latent_shape = (512,)
epochs = 50
learning_rate = 0.0005
criterion = torch.nn.MSE
optimiser = torch.optim.Adam
action_conditional = DiagLinear
exafferent_stop_grad = True
noop_index = 0
```

Experiment (3): Artificial Ape

The model trained again uses the UNet architecture following that used in Freeway, this time with approx. 10M parameters. It was trained for 50 epochs on 5k observations/actions collected using a uniform random policy and stored. The Adam optimiser with a learning rate 0.0005 was used. The dataset used is publicly available[9].

```
==============================================================================================
Layer (type:depth-idx)                   Output Shape            Param #
==============================================================================================
UNet                                     [2, 1, 64, 64]          --
├─DoubleConv: 1-1                        [2, 16, 64, 64]         --
│    └─Sequential: 2-1                   [2, 16, 64, 64]         --
│    │    └─Conv2d: 3-1                  [2, 16, 64, 64]         144
│    │    └─Identity: 3-2                [2, 16, 64, 64]         --
│    │    └─LeakyReLU: 3-3               [2, 16, 64, 64]         --
│    │    └─Conv2d: 3-4                  [2, 16, 64, 64]         2,304
│    │    └─Identity: 3-5                [2, 16, 64, 64]         --
│    │    └─LeakyReLU: 3-6               [2, 16, 64, 64]         --
├─Down: 1-2                              [2, 32, 32, 32]         --
│    └─Sequential: 2-2                   [2, 32, 32, 32]         --
│    │    └─MaxPool2d: 3-7               [2, 16, 32, 32]         --
│    │    └─DoubleConv: 3-8              [2, 32, 32, 32]         13,824
├─Down: 1-3                              [2, 64, 16, 16]         --
│    └─Sequential: 2-3                   [2, 64, 16, 16]         --
│    │    └─MaxPool2d: 3-9               [2, 32, 16, 16]         --
│    │    └─DoubleConv: 3-10             [2, 64, 16, 16]         55,296
├─Down: 1-4                              [2, 128, 8, 8]          --
│    └─Sequential: 2-4                   [2, 128, 8, 8]          --
│    │    └─MaxPool2d: 3-11              [2, 64, 8, 8]           --
│    │    └─DoubleConv: 3-12             [2, 128, 8, 8]          221,184
├─Down: 1-5                              [2, 256, 4, 4]          --
│    └─Sequential: 2-5                   [2, 256, 4, 4]          --
│    │    └─MaxPool2d: 3-13              [2, 128, 4, 4]          --
│    │    └─DoubleConv: 3-14             [2, 256, 4, 4]          884,736
├─DiagLinear: 1-6                        [2, 256, 4, 4]          --
│    └─Flatten: 2-6                      [2, 4096]               --
│    └─Linear: 2-7                       [2, 1024]               4,195,328
│    └─Linear: 2-8                       [2, 1024]               4,096
│    └─Linear: 2-9                       [2, 4096]               4,198,400
│    └─View: 2-10                        [2, 256, 4, 4]          --
├─Up: 1-7                                [2, 128, 8, 8]          --
│    └─ConvTranspose2d: 2-11             [2, 128, 8, 8]          131,200
│    └─DoubleConv: 2-12                  [2, 128, 8, 8]          --
│    │    └─Sequential: 3-15             [2, 128, 8, 8]          442,368
├─Up: 1-8                                [2, 64, 16, 16]         --
│    └─ConvTranspose2d: 2-13             [2, 64, 16, 16]         32,832
│    └─DoubleConv: 2-14                  [2, 64, 16, 16]         --
│    │    └─Sequential: 3-16             [2, 64, 16, 16]         110,592
├─Up: 1-9                                [2, 32, 32, 32]         --
│    └─ConvTranspose2d: 2-15             [2, 32, 32, 32]         8,224
│    └─DoubleConv: 2-16                  [2, 32, 32, 32]         --
│    │    └─Sequential: 3-17             [2, 32, 32, 32]         27,648
├─Up: 1-10                               [2, 16, 64, 64]         --
│    └─ConvTranspose2d: 2-17             [2, 16, 64, 64]         2,064
│    └─DoubleConv: 2-18                  [2, 16, 64, 64]         --
│    │    └─Sequential: 3-18             [2, 16, 64, 64]         6,912
├─OutConv: 1-11                          [2, 1, 64, 64]          --
│    └─Conv2d: 2-19                      [2, 1, 64, 64]          17
├─Tanh: 1-12                             [2, 1, 64, 64]          --
==============================================================================================
Total params: 10,337,169
Trainable params: 10,337,169
Non-trainable params: 0
Total mult-adds (M): 444.08
==============================================================================================
Input size (MB): 0.03
Forward/backward pass size (MB): 10.13
Params size (MB): 41.35
Estimated Total Size (MB): 51.51
==============================================================================================
```

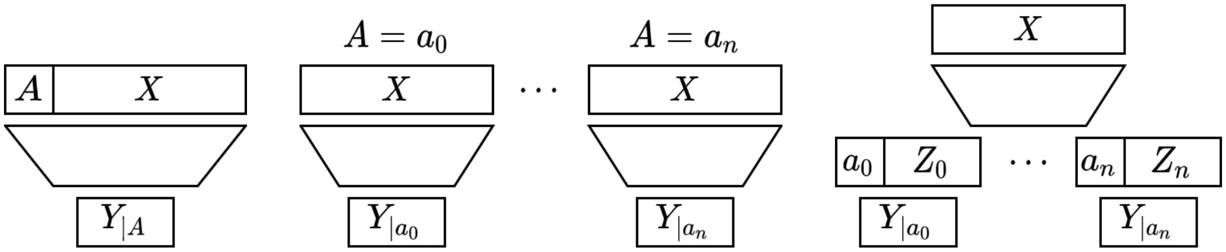
```
state_shape = (1,64,64)
action_shape = (3,)
latent_shape = (512,)
epochs = 50
learning_rate = 0.0005
criterion = torch.nn.MSE
optimiser = torch.optim.Adam
action_conditional = DiagLinear
exafferent_stop_grad = True
noop_index = 1
```

Discussion of Model Architectures

A common problem arises when learning conditional expectations in settings where some variables have a significantly lower dimension/are not required to produce a reasonable estimate. A model may require more training time in order to incorporate these variables into the estimate, or in some cases may end up ignoring

---

[9] https://github.com/BenedictWilkins/disentangling-reafference



(a) Naive Architecture (b) Grouped Architecture (c) Grouped Representation Architecture

Figure A.4: Three kinds of architectures that may be used to learn the conditional outcome. (a) is the naive approach which simply includes A as part of the model input. (b) is a group of models each trained on different subsets of data selected by action. This can lead to higher variances as the models are not training on all data. (c) is a single model that first learns a common representation of the input and then specializes. Each specialised module is trained only when their associated realisation of A is observed in a batch.

them all together. This problem is common in deep learning, but is particularly prevalent in the case of causal inference. Carefully introducing inductive bias into the model by selecting a suitable architecture is one way of dealing with the issue. In the case of discrete actions, independent models can be learned conditional on each realization of the action as illustrated in Fig. A.4.b. These models tend to be less data efficient and have higher variances as they are not trained on the full dataset. There are likely features of X that could be shared among the models, but must be learned by each model independently. Learning shared representations and subsequently specialising is a well-known technique in deep learning (Shalit et al. 2017) and can be applied here, see Fig. A.4.c. Conditioning in the hidden layers of the network provides the additional advantage that more complex mixing of hidden features and the conditional input can be applied. This kind of model still does not use all data. The model used in the Cartpole experiment follows the architecture in A.4.a since X is low dimensional. The other experiments follow the architecture in A.4.c.

A.3.3 Algorithm 6.1 Technical Details

Counterfactual Stop Gradients

$\hat{\delta}_\emptyset$ is included in the objective function twice. To balance terms, the stop-gradient operator (assigning gradients to zero) can be used on $\hat{\delta}_\emptyset$ in the estimate of reafference. This may help to stabilise training, although we have not observed a substantial difference with/without stop-gradients in our experiments.

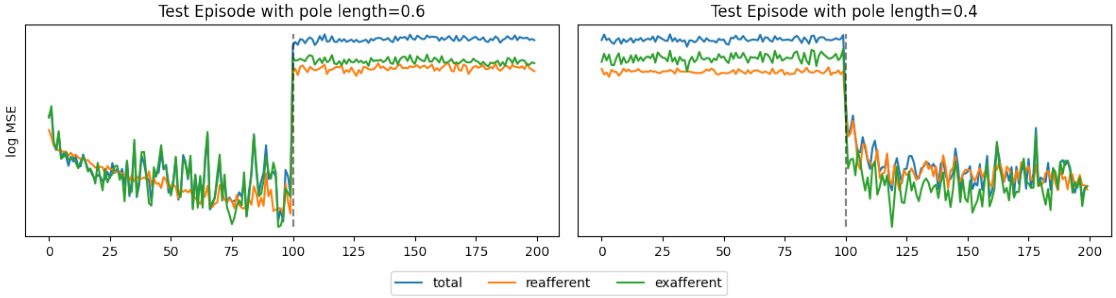


Figure A.5: Demonstration of adaptivity in the Cartpole environment. Treating the cart and pole as the agent's body, the length of the pole l is changed during training to simulate changes that a biological agent might undergo. Changing l changes the dynamics of the system, both reafferent and exafferent effects change as a result and should be relearned by the agent. Each graph shows the performance on a test episode where estimates are compared to the ground truth for differing l. Training starts with $l = 0.6$ and is changed $l = 0.4$ after 100 epochs. During the phase where training l matches test l the error is minimised, otherwise the model performs poorly as is expected. The forward model is able to quickly adapt (see right) its estimates after a change in the agent's body.

Zeroing Reafference

The estimated reafferent effect for the null action should be zero. At the initial stages of training the estimate will be non-zero and small errors later may impact performance. A simple trick that may improve training time and overall performance is simply to set $\hat{\delta}_a = 0$ where $a_i = \emptyset$ in the mini-batch. If interpolation between actions is required (for example, with continuous actions) this may be a detriment to performance since some gradient information is lost and the additional freedom in the estimate at $\emptyset$ might impact neighbouring actions.

Variance of Reafference Estimates

Since reafference is computed from two estimates the variance of its estimate is comparatively large. When reafferent effects are small, the additional epistemic uncertainty can have a proportionally significant impact on the estimate. This is a limitation of our approach, exploring it further is left as future work.

A.3.4 Additional Experiments

Adaptivity in Cartpole

Biological agent's experience change in their motor system over their lifetime, for example, through growth or disease. The same action will consistently give rise to different effects at different times; any model of reafference needs to be capable of adapting to a changing motor system. For bug identification this capability is important for dealing with non-stationarity.

We perform a simple experiment so show that, by continued training with Alg. 1 models can adapt to changes in the agent's body (or environment). This is demonstrated in the Cartpole environment (where the cart and pole is considered the body of the agent) by changing the length of the pole during training. Results are presented in Fig. A.5. The model is able to quickly adapt to changes in the reafferent effect and properly recover both reafference and exafference after a short period of re-training.

Doing Something in Cartpole

In environments where the do-nothing action is not available and where it is possible to intervene on the environments mechanism, it may still be possible to recover the reafferent effects. This situation is analogous to the bi-pedal robot example reviewed in the introduction and can be seen as a kind of inverse of our approach.

In the Cartpole environment, it turns out that the required interventions are $do(G = 0)$, $do(\dot{X} = 0)$, $do(\dot{\theta} = 0)$, these can be derived from the SCM (see section A.3.4), which is in turn derived from the dynamical equations that described the physical system (see (Florian 2007)). With knowledge of the SCM, the reafferent effect can be isolated by zeroing out additive terms that do not depend on the action. With these interventions, the reafferent effect can be estimated as the total effect. Training the reafferent forward model on the intervened mechanism, exafference is then estimated as any error when testing without intervention. This mechanism is closer to the view of reafference illustrated in Fig. 6.3.

Intervention on the environment mechanism can drastically impact how the environment evolves, some states may be difficult (or impossible) to reach using just the agent's action. This issue can be seen in the Cartpole environment. A random policy is unlikely to reach regions of the state space that are otherwise commonly visited without environment intervention. The issue can be mitigated if the intervention is intermittent. Intermittence may be difficult to achieve in some environments, and the choice of when to intervene to obtain the most information about causal effects is non-trivial.

Artificial Ape without an Indicator

If no indicator is present in the Artificial Ape environment, then aleatoric uncertainty is high and the estimates will be difficult to interpret. The distribution of exafferent view shifts is wide without an indicator and is being approximated by its expectation, leading to blurry results. The result of experiment (iii.2) without an indicator can be found in Fig. A.6.

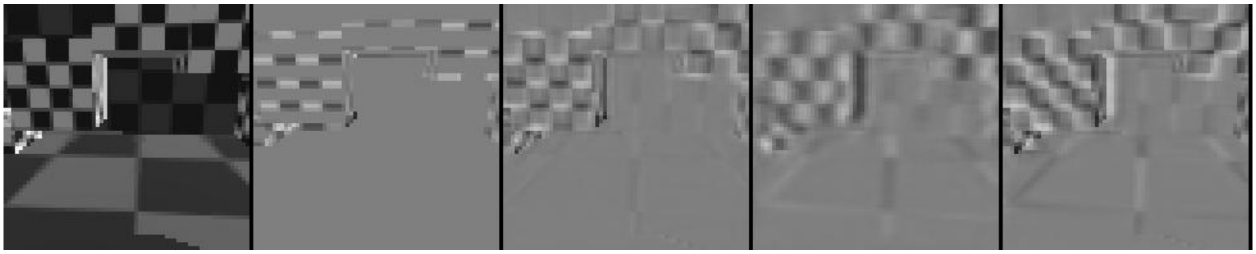

(a) (b) (c) (d) (e)

Figure A.6: Disentangling in Artificial Ape with congruent reafference and exafference without the platform as an indicator. (a) observation (b) ground truth total effect (c) estimated total effect (d) estimated reafferent effect (e) estimated exafferent effect. As in experiment (iii.2) in the main body of the paper, the agent and platform have rotated in opposite directions, leading to a cancellation in the total effect. The estimate of the total effect does not match the ground truth because of the aleatoric uncertainty in this environment. In cases such as these it might be better to use a generative model rather than approximating with expectations as is done here. This is left as a direction for future work.

A.3.5 Miscellaneous

Empty Space Example Calculations

The average reafferent effect of action 1 in the example given in section 6.2.2 can be computed as follows:

$$\begin{aligned}\delta_{Y'}(1) &= \mathbb{E}[Y'(1)] - \mathbb{E}[Y'(0)] \\ &= [\mathbb{E}[A \cdot Z | do(A=1)] + \mathbb{E}[Y]] \\ &\quad - [\mathbb{E}[A \cdot Z | do(A=0)] + \mathbb{E}[Y]] \\ &= 1 \cdot Pr(Z=1) - 0 \cdot Pr(Z=1) \\ &= Pr(Z=1) = p_z\end{aligned}$$

The individual reafferent effect is similar, except we are conditioning on Z, $Pr(Z=1)=1$ when $Z=1$ and 0 when $Z=0$ leading to the stated result. Note that in the example there is no confounding since the agent is essentially performing a randomised trail.

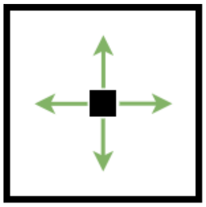
Appendix B

Environments

This chapter presents details for all of the environments used in the various experiments throughout this thesis. OpenAI gym (Brockman et al. 2016) is used to keep the environment API consistent. This API is principally single agent, with a simple environment loop that given an action will provide the agent with its next observation. Datasets that have been used in the experiments presented in this thesis have been archived to ensure that any results are reproducible, links are provided in technical appendix under the *Data Dependencies* sections where applicable.

B.1 Simple Environments

B.1.1 Alone-v0

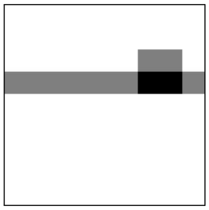


The `Alone-v0` environment is a deterministic environment in which a single agent (a black square) moves around inside a room taking the actions $\mathcal{A} =$ {`NORTH`,`EAST`,`SOUTH`,`WEST`,`NOOP`}. The agent cannot move across the boundary of the room, any action attempting to do so will fail. The environment implementation can be found here[1].

Alone-v1

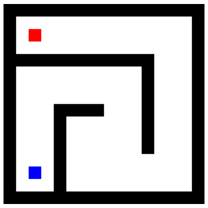
The `Alone-v1` environment is a variant of `Alone-v0` in which the agent is affected by an external force. At each step the agent is randomly pushed north/south with probability 0.25, or is unaffected with probability 0.5.

Scan-v0



The `Scan-v0` environment is another variant of the `Alone-v0` environment. It contains an additional scanning line that continuously moves from top to bottom (then resets) independently of the agent. The environment is a smaller (9×9 pixels) and has no outer border, although the agent is still constrained to moving inside the 9×9 space. The agent and scanning line interact additively as can be seen in the image (right).

B.1.2 Explorer-v0



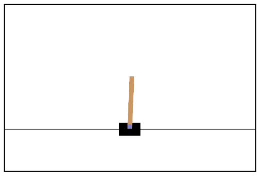
The `Explorer-v0` environment is a deterministic environment in which a single agent (red) can move in a simple maze taking the actions $\mathcal{A} =$ {`NORTH`,`EAST`,`SOUTH`,`WEST`,`NOOP`}. The agent should try to reach the goal (blue) and cannot pass through walls (black). The environment implementation can be found here[2].

B.1.3 Cartpole

---

[1] https://github.com/BenedictWilkins/gym-pygame commit hash: b5616d4713f46ee6efaa13703e2ec899bc4a3a7b

[2] https://github.com/BenedictWilkins/gym-explorer commit hash: 83f8c9d7a3f420db40379fee1a1d9b13c38cc12e



This environment is a modified version of the `Cartpole-v1` environment provided by OpenAI gym (Brockman et al. 2016). It has been modified to support our reafference experiments, see section 6.2.2. The environment is modified to contain the null-action, and is made more interesting by allowing the pole to fall further before resetting the environment. $\mathcal{A} = \{-\beta, 0, \beta\}$. Each action applies a force to the cart (0 applies no force). The agent's observation contains the cart position and velocity, and the pole angle and angular velocity. The forces on the cart/pole are not observed. The Structural Causal Model (SCM) for the environment is given below:

$$\begin{aligned}
A_t &:= \pi(\theta) \\
Y_{t+1} &:= Y_t + dt\dot{Y}_t \\
\dot{Y}_{t+1} &:= \dot{Y}_t + dt\ddot{Y}_t \\
\theta_{t+1} &:= \theta_t + dt\dot{\theta}_t \\
\ddot{Y}_{t+1} &:= \gamma_t + M_p\ddot{\theta}_t cos(\theta_t)/(M_p + M_c)
\end{aligned}$$

$$\begin{aligned}
\dot{\theta}_{t+1} &:= \dot{\theta}_t + dt\ddot{\theta}_t \\
\ddot{\theta}_{t+1} &:= (Gsin(\theta_t) - cos(\theta_t)\gamma_t)/C_t \\
\gamma_t &:= (A_t + M_p\dot{\theta}_t^2 sin(\theta_t))/(M_p + M_c) \\
C_t &:= L(4/3 - M_p cos(\theta_t)^2/(M_p + M_c)
\end{aligned}$$

Where Y is the cart position, θ is the pole angle, $\dot{\square}$ indicates a derivative (velocity, acceleration). M_p and M_c are the masses of the pole and cart respectively, L is the length of the pole, G is the gravitational acceleration. γ_t and C_t are placeholder variables that do not need to be included in the causal graph. The agent observes $X_t = (Y_t, \dot{Y}_t, \theta_t, \dot{\theta}_t)$.

Semi-Implicit Euler Cartpole

The kinematics integrator used in the original environment updates Y_t and θ_t given the previous values of the higher-order terms. Actions therefore only immediately affect the higher-order terms. In the semi-implicit Euler version of Cartpole the SCM is modified as follows:

$$\begin{aligned}
Y_{t+1} &:= Y_t + dt\dot{Y}_{t+1} \\
\theta_{t+1} &:= \theta_t + dt\dot{\theta}_{t+1}
\end{aligned}$$

With these modifications, actions now have an immediate effect on the cart position and pole angle. The experiment presented in section 6.2.2 for the Cartpole environment was rerun with the semi-implicit Euler version. Changes in the angle and position were modelled correctly. The environment is part of the standard

OpenAI gym Python package, the specific modifications can be found here[3]

---

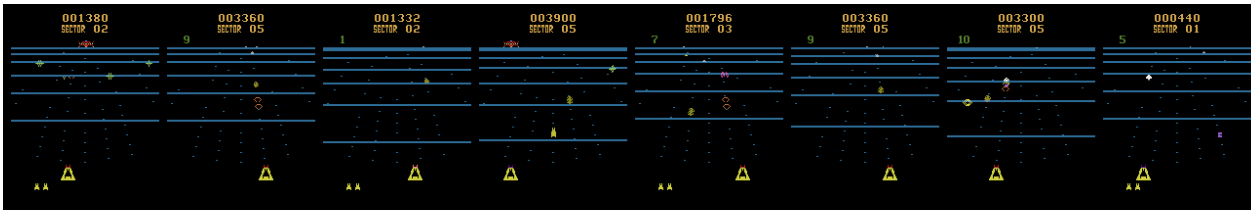
[3] https://github.com/BenedictWilkins/disentangling-reafference commit hash: c594df55ee74cd0c3360ece7c7244df78d1e9ec1

B.2 Atari 2600

Atari 2600 games have served as a long-standing reinforcement learning benchmark. The environments have been made available as part of the Arcade learning environment (ALE) (Bellemare et al. 2013) and OpenAI Gym (Brockman et al. 2016). We make use of 8 of the total 51 games in experiments in this thesis. Various datasets have also been curated to support our experiments and ensure that any results are easily reproducible. Each environment is shown in the sections to follow for reference. Additional information can be found in the associated papers (Wilkins et al. 2020) and (Wilkins et al. 2023), and in the Gym documentation.

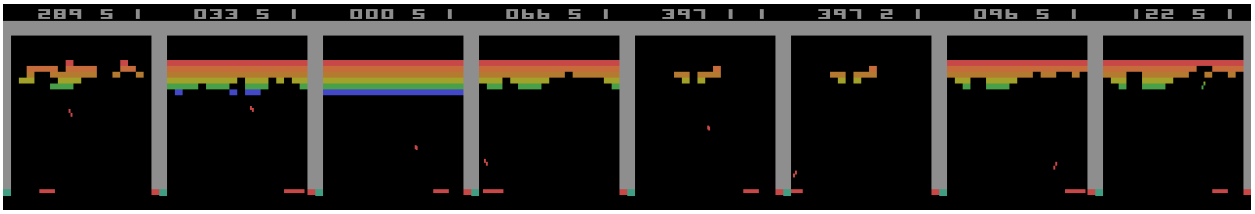
B.2.1 Beam Rider

Gym ID	Notes
`BeamRiderNoFrameskip-v4`	Acyclic and contains flashing (large pixel discontinuities).



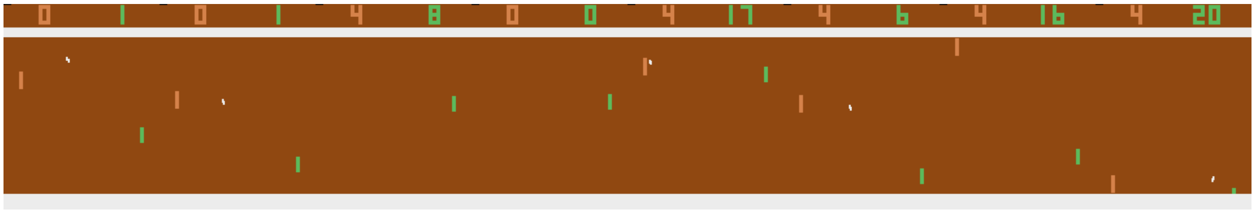
B.2.2 Breakout

Gym ID	Notes
`BreakoutNoFrameskip-v4`	Acyclic and high combinatorial dimension.



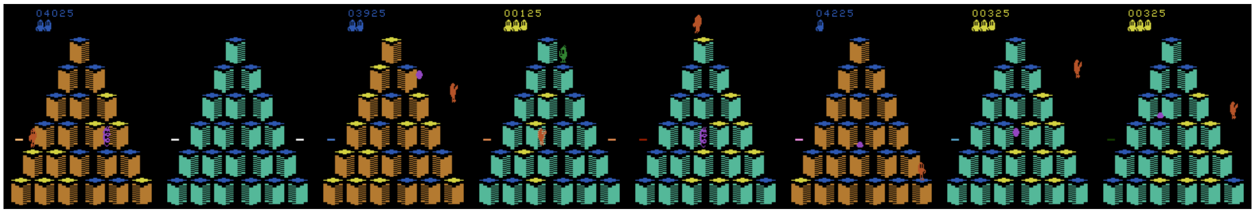
B.2.3 Pong

Gym ID	Notes
`PongNoFrameskip-v4`	Cyclic.



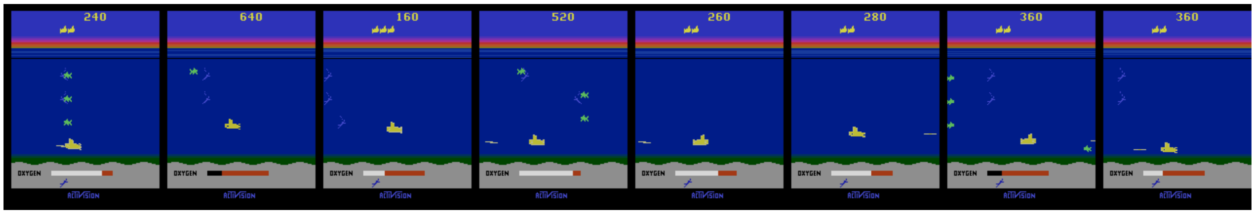
B.2.4 Qbert

Gym ID	Notes
`QbertNoFrameskip-v4`	Cyclic and flashing/scene changes.



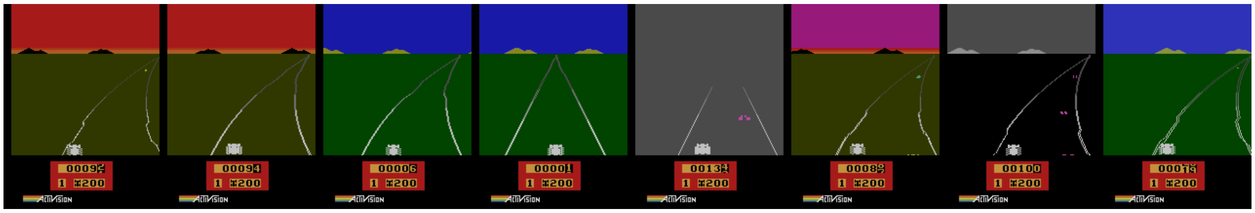
B.2.5 Seaquest

Gym ID	Notes
`SeaquestNoFrameskip-v4`	Acyclic.



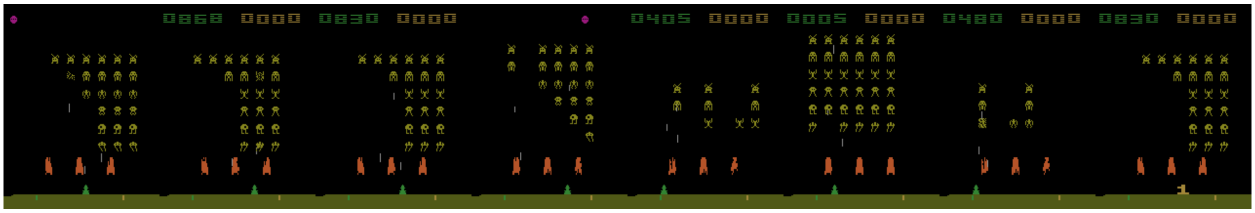
B.2.6 Enduro

Gym ID	Notes
`EnduroNoFrameskip-v4`	Acyclic and significant (pixel-wise) scene changes.



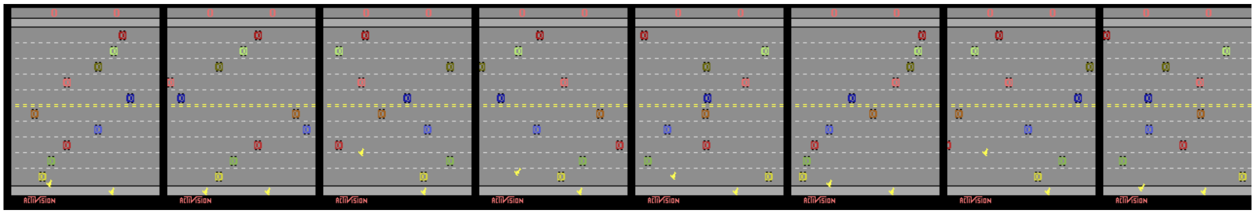
B.2.7 Space Invaders

Gym ID	Notes
`SpaceInvadersNoFrameskip-v4`	Acyclic and high combinatorial dimension.

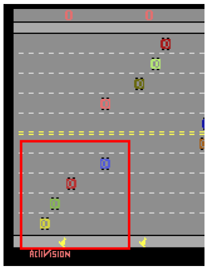


B.2.8 Freeway

Gym ID	Notes
`FreewayDeterministic-v4`	Acyclic.



Modified Freeway



The Freeway environment was modified for use in our reafference experiments presented in section 6.2.2. The agent's observation is an $3 \times 84 \times 84$ colour image, a cropped version of the original observation. The agent may take one of three actions: $\mathcal{A}$ = {`FORWARD`,`BACKWARD`,`NOOP`}, moving the chicken forward, backwards or keeping it stationary. The modified observation is shown in the graphic (right) as outlined by the red rectangle.

B.3 World of Bugs

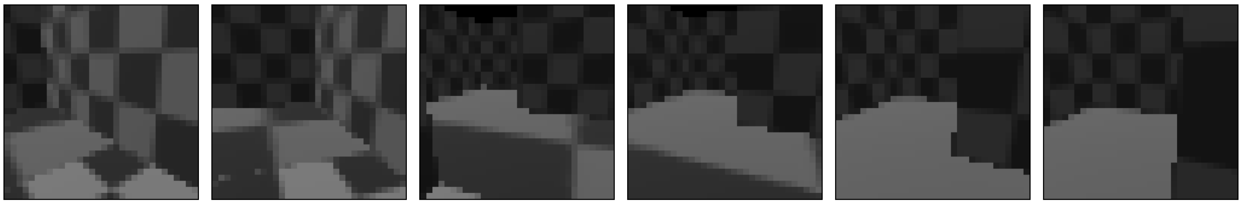
Despite there being a number of environments implemented in the World of Bugs platform (as described in chapter 4), we did not make use of all of them in this thesis. This because the bugs they implement require work which takes us slightly out of scope (e.g. identifying progression bug in the `GettingStuck-v0` environment). Below details of the environments that *are* used in experiments are given. Details for the other environments implemented in the platform can be found in the platform documentation. In each environment the agent observes from a first-person perspective. The bugs that are implemented vary across environments.

B.3.1 World-v0

In the `World-v0` the player is situated in a room surrounded by walls with two static cubes in two of the corners. The agent can take the actions: $\mathcal{A}$ = {`ROTATE_LEFT`, `ROTATE_RIGHT`, `FORWARD`, `NOOP`}. The actions rotate its view left or right, or move the player forward in the direction they are facing. Although there are a good number of bugs implemented in this environment, only the texture corruption bug is investigated in chapter 4. Many of the other bugs are also part of the `Maze-v1` environment (see below) and are investigated in chapter 5.

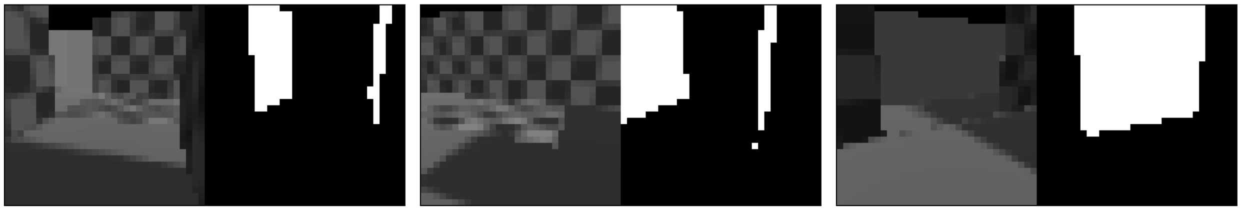
B.3.2 Maze-v1

The `Maze-v1` environment is a simplified (smaller) version of the `Maze-v0` environment that is outlined in chapter 4. This environment is used to demonstrate the identification of a number of different bugs using Self-Supervised Learning (SSL) in chapter 5. The agent can take the actions: $\mathcal{A}$ = {`ROTATE_LEFT`, `ROTATE_RIGHT`, `FORWARD`, `NOOP`}. Observations are $1 \times 32 \times 32$ pixel (grey-scale) images. Examples of the bugs that are used in experiments are shown below.



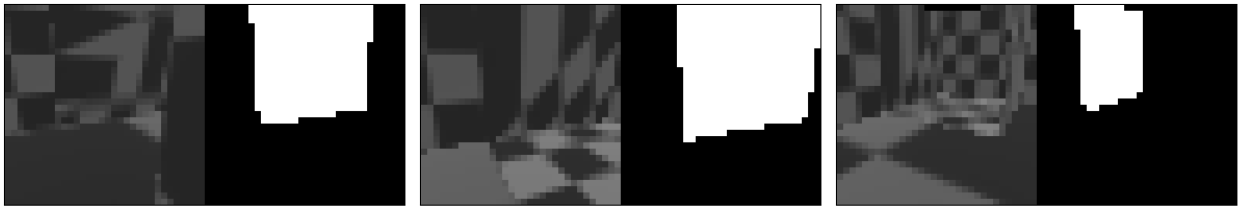
Normal observations (taken every 10th frame).

(a) Geometry corruption

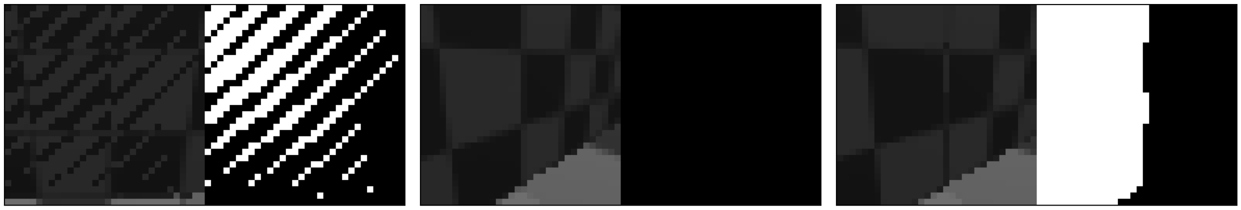


(b) Missing object

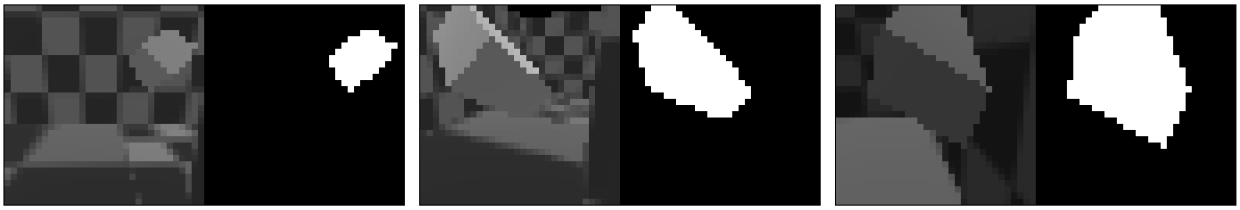
(c) Screen tearing



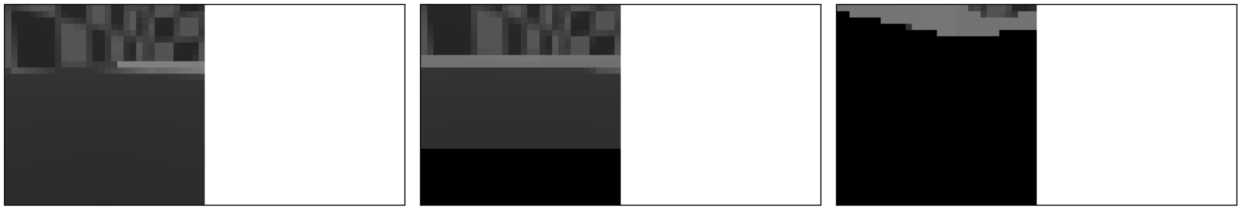
(d) Texture corruption



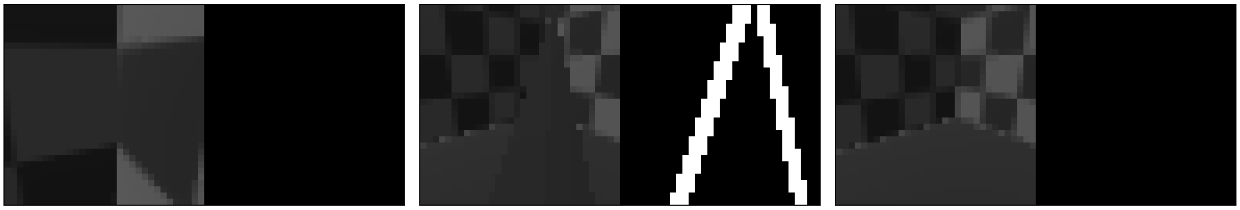
(e) Z-fighting



(f) Unintended object

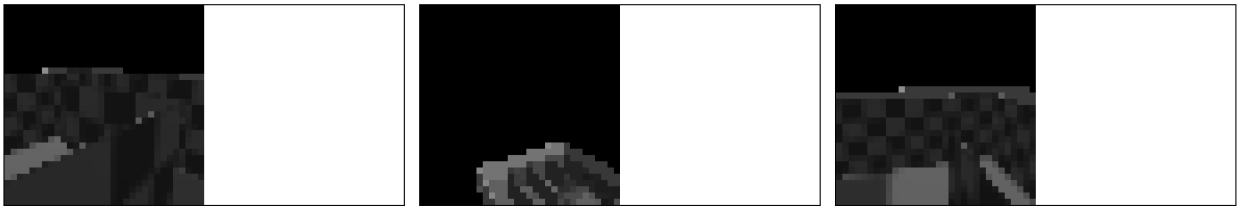


(g) Terrain hole (frames left to right)



(h) Unintended shortcut (Geometry clipping) (frames left to right)

(i) Player out of bounds



(j) High force

B.3.3 Artificial Ape

This environment is used in our reafference experiments under the alias `Artificial Ape`, but the WOB implementations Gym ID is `LookAround-v0`. The environment contains a collection of moving chequered cubes, and a rotating platform that the agent stands on. As the platform rotates so does the player. The player can take the actions $\mathcal{A} = \{$`ROTATE_LEFT, ROTATE_RIGHT, NOOP`$\}$. The dataset used in the experiments can be found in the paper's (Wilkins et al. 2023) supplementary material here[4]. No bugs are implemented in this environment, it was used purely to validate our reafference experiments.

---

[4] https://github.com/BenedictWilkins/disentangling-reafference

Appendix C

Debugging with ChatGPT

The pages to follow show part of several conversations with OpenAI's ChatGPT, a large language model that has been fine-tuned on human feedback and specializes in dialogue. In each prompt we ask it to point out the bug in a given block of code. This is not meant as a thorough exploration of the model's capabilities, the bugs presented are very simple. Even so, it highlights the fact that there has been some progress in the problem of automated analysis or debugging. **Date of all interactions: December 14 2022.**

Prompt » (new conversation)

~~~
Find the bug with this code:

```
for(var i=0; i<5; i++) {
  setTimeout(() => console.log(i), 1000)
}
```
~~~

Response »

~~~
The code will log 5 five times, instead of logging the numbers 0 through 4. This is because the i variable is declared with var, which means it is function-scoped. As a
result, the i variable declared in the for loop will be overwritten on each iteration of the loop, so by the time the setTimeout callback functions are executed, the i
variable will always have the value 5.

To fix this, you can either use the let keyword instead of var to declare the i variable, or you can use an immediately invoked function expression (IIFE) to create a new
variable scope for each iteration of the loop:

```
for(var i=0; i<5; i++) {
  (function(i) {
    setTimeout(() => console.log(i), 1000)
  })(i);
}
```

This will create a new variable i within the IIFE for each iteration of the loop, so the setTimeout callback functions will be able to access the correct value of i.
~~~

Figure C.1: Start of the conversation with ChatGPT. It is asked to find a bug in the given javascript code, is successful and offers a potential fix. See the next conversation C.2.

Prompt »

~~~~
Find the bug with this code:

```
for(var i=0; i>5; i++) {
  setTimeout(() => console.log(i), 1000)
}
```
~~~~

Response » (continued from conversation C.1)

~~~~
The code will never execute the setTimeout callback functions, because the for loop will never run. This is because the loop's exit condition is i > 5, but the i variable
is initialized to 0, which is not greater than 5. As a result, the for loop will not run, and the setTimeout callback functions will never be called.
To fix this, you can either change the exit condition to i < 5, or you can change the initial value of i to a value that is less than 5, such as -1:

```
for(var i=-1; i<5; i++) {
  setTimeout(() => console.log(i), 1000)
}
```

This will cause the for loop to run 5 times, and the setTimeout callback functions will be called with the values 0 through 4.
~~~~

Alternative Response » (new conversation)

~~~~
The bug in this code is that the condition in the for loop will never be satisfied. The condition i>5 will always be false because i is initialized to 0 and the loop will
never run. The code should use the condition i<5 in order to run the loop 5 times.

Additionally, this code uses the var keyword to declare the i variable, which has been deprecated in modern JavaScript in favor of using the let or const keywords. Using
let or const would prevent the variable from being redeclared in the same scope, which can help prevent potential bugs.
~~~~

Figure C.2: ChatGPT is asked to find a different bug in the given javascript code. In the first response its natural language description of the issue is correct, but the suggestion might not be what we expect (starting at -1). It seems to forget that the code also contains the bug in C.1 and suggests it anyway. It also makes a mistake explaining its own code, saying that the loop with output 0 to 4 when it will actually output -1 to 4. In the alternative response, the same question is asked without any previous conversation, this time ChatGPT does not suggest code, but still answers correctly. See the next conversation C.3.

Prompt » (continued from conversation C.2 alternative)

~~~~
Find the bug with this code:

```
X = 1
```
~~~~

Response »

~~~~
This code has several issues that could be considered "bugs."

First, it uses the X variable without declaring it with the let, var, or const keywords. In JavaScript, this will automatically create a global variable, which can cause
unintended behavior and is generally considered bad practice. It's better to always declare variables before using them to avoid this kind of issue.

Second, the code uses an uppercase X as the variable name. In JavaScript, variable names are typically written in camel case (i.e. myVariable), or with underscores between
words (i.e. my_variable). Using uppercase letters in variable names is allowed, but it can be confusing and is not commonly used.

Third, the code does not include a semicolon at the end of the line, which is required in JavaScript. Omitting the semicolon can cause issues with automatic semicolon
insertion, which can lead to unexpected behavior in your code.

In summary, the code can be rewritten as follows to correct these issues:

```
let myVariable = 1;
```

This code declares the myVariable variable with the let keyword, uses camel case for the variable name, and includes a semicolon at the end of the line. This is considered
standard practice in JavaScript.
~~~~

Figure C.3: Continuing the conversation from C.2. ChatGPT is asked about a different block of code. It remembers that javascript was the language of choice (from the previous prompt/response) and explains various problems with this code. See the next conversation C.4.

Prompt » (new conversation)

~~~
Find the bug with this code:

```
X = 1
```
~~~

Response »

```
There does not appear to be any error or bug in the code that you have provided. It is simply a statement that assigns the value of 1 to the variable X.
```

Figure C.4: ChatGPT is asked the same question as in conversation C.3, but this time without context (in a new conversation). It (correctly?) states that there is nothing wrong with the code - this is valid Python code, but not valid C code.

Appendix D

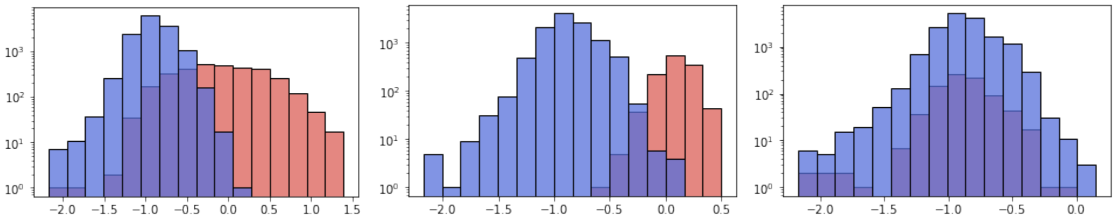
Numerical Results

The pages to follow list detailed numerical results for the various experiments presented in this thesis, especially in chapter 5. See section A.2.1 for guidance.

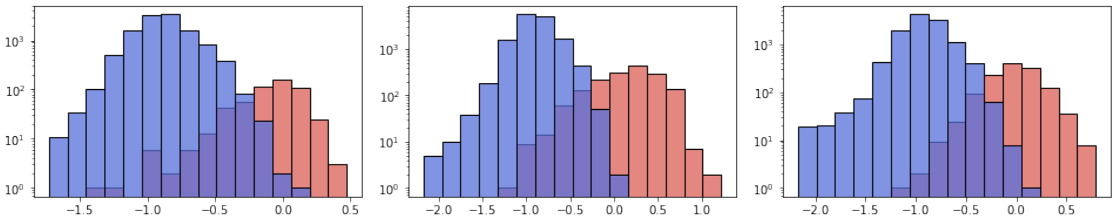
Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.959	0.906	0.932	13622	12347	1275	-0.896
	1 (A)	0.680	0.838	0.751	3232	2709	523	-0.083
Flicker	0 (N)	1.000	0.994	0.997	11361	11296	65	-0.872
	1 (A)	0.949	0.996	0.972	1210	1205	5	0.099
Freeze	0 (N)	0.951	0.525	0.676	15919	8353	7566	-0.870
	1 (A)	0.051	0.486	0.092	832	404	428	-0.879
Lag	0 (N)	0.997	0.961	0.979	11869	11404	465	-0.879
	1 (A)	0.523	0.944	0.673	540	510	30	-0.098
Split Horizontal	0 (N)	0.994	0.970	0.982	14973	14530	443	-0.894
	1 (A)	0.777	0.947	0.853	1630	1543	87	0.148
Split Vertical	0 (N)	0.995	0.977	0.986	11706	11438	268	-0.911
	1 (A)	0.819	0.959	0.884	1267	1215	52	-0.016

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	-0.616	0.872	0.878	0.192	0.934	0.893
Flicker	-0.337	0.995	0.997	0.096	1.000	0.994
Freeze	-0.875	0.505	0.049	0.050	0.494	0.523
Lag	-0.489	0.953	0.916	0.044	0.982	0.960
Split Horizontal	-0.467	0.958	0.963	0.098	0.990	0.968
Split Vertical	-0.447	0.968	0.970	0.098	0.994	0.975

Scores $log(\Delta_1)$



(a) Visual Artefact (b) Flicker (c) Freeze

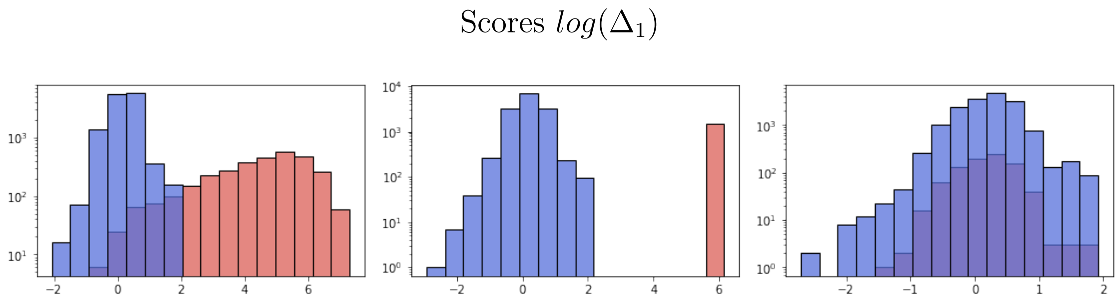


(d) Lag (e) Horizontal Split (f) Vertical Split

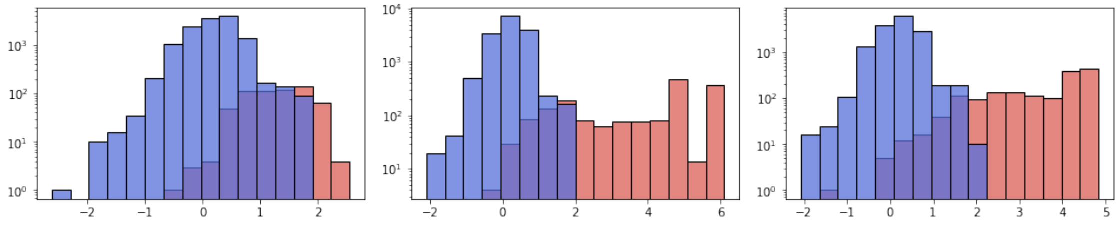
Figure D.1: S3N results for Beam Rider $D = 64$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.991	0.976	0.983	13386	13060	326	0.214
	1 (A)	0.903	0.964	0.933	3150	3037	113	4.429
Flicker	0 (N)	1.000	1.000	1.000	13952	13952	0	0.190
	1 (A)	1.000	1.000	1.000	1468	1468	0	6.040
Freeze	0 (N)	0.950	0.564	0.708	16310	9195	7115	0.212
	1 (A)	0.049	0.434	0.089	850	369	481	0.184
Lag	0 (N)	0.994	0.890	0.939	13083	11642	1441	0.195
	1 (A)	0.272	0.891	0.417	604	538	66	1.287
Split Horizontal	0 (N)	0.993	0.970	0.981	15778	15298	480	0.218
	1 (A)	0.767	0.935	0.843	1689	1580	109	3.822
Split Vertical	0 (N)	0.998	0.972	0.985	14744	14335	409	0.204
	1 (A)	0.791	0.980	0.876	1583	1552	31	3.481

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	1.004	0.970	0.981	0.190	0.988	0.973
Flicker	5.778	1.000	1.000	0.095	1.000	1.000
Freeze	0.308	0.495	0.047	0.050	0.483	0.557
Lag	0.651	0.890	0.623	0.044	0.953	0.890
Split Horizontal	0.914	0.952	0.948	0.097	0.984	0.966
Split Vertical	0.941	0.976	0.979	0.097	0.992	0.973



(a) Visual Artefact (b) Flicker (c) Freeze



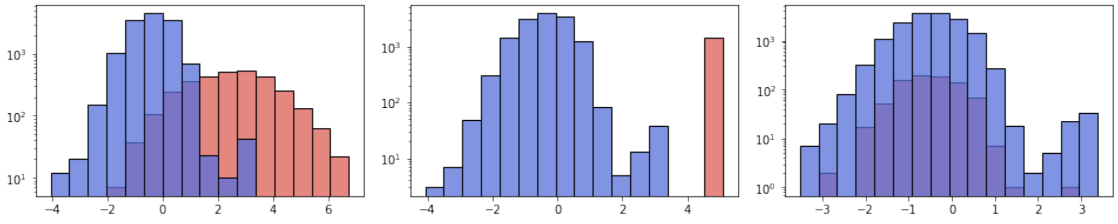
(d) Lag (e) Horizontal Split (f) Vertical Split

Figure D.2: S3N results for Breakout $D = 256$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

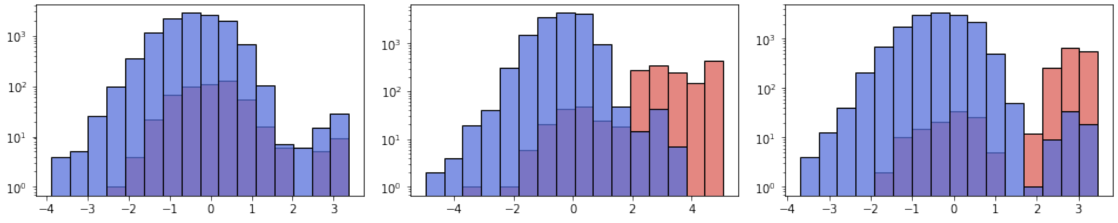
Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.968	0.952	0.960	13460	12816	644	-0.387
	1 (A)	0.810	0.867	0.837	3156	2737	419	2.490
Flicker	0 (N)	1.000	1.000	1.000	13845	13845	0	-0.374
	1 (A)	1.000	1.000	1.000	1453	1453	0	4.927
Freeze	0 (N)	0.948	0.512	0.665	15781	8078	7703	-0.468
	1 (A)	0.048	0.470	0.087	829	390	439	-0.508
Lag	0 (N)	0.971	0.695	0.810	12018	8356	3662	-0.371
	1 (A)	0.067	0.515	0.119	515	265	250	0.031
Split Horizontal	0 (N)	0.990	0.994	0.992	14999	14905	94	-0.309
	1 (A)	0.940	0.911	0.925	1622	1478	144	3.230
Split Vertical	0 (N)	0.992	0.996	0.994	15010	14948	62	-0.357
	1 (A)	0.960	0.929	0.944	1610	1496	114	2.667

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	0.718	0.909	0.915	0.190	0.958	0.936
Flicker	4.796	1.000	1.000	0.095	1.000	1.000
Freeze	-0.450	0.491	0.047	0.050	0.484	0.510
Lag	0.028	0.598	0.082	0.041	0.633	0.688
Split Horizontal	1.385	0.952	0.925	0.098	0.965	0.986
Split Vertical	1.910	0.962	0.917	0.097	0.970	0.989

Scores $log(\Delta_1)$



(a) Visual Artefact (b) Flicker (c) Freeze



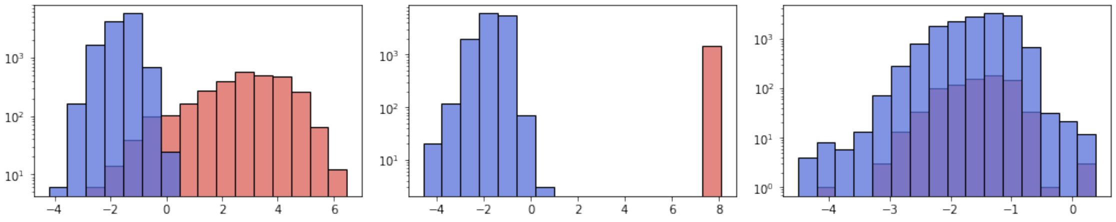
(d) Lag (e) Horizontal Split (f) Vertical Split

Figure D.3: S3N results for Enduro $D = 64$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

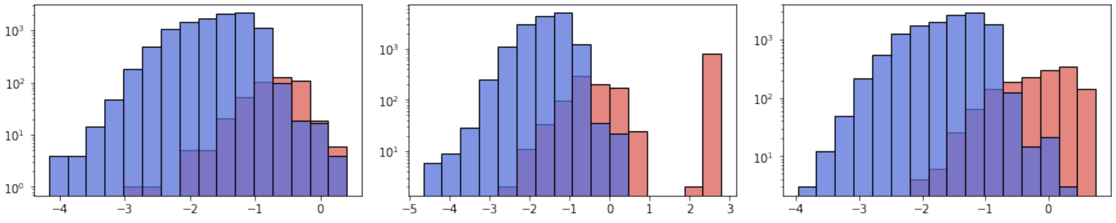
Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.993	0.995	0.994	12413	12354	59	-1.593
	1 (A)	0.980	0.970	0.975	2974	2885	89	2.766
Flicker	0 (N)	1.000	1.000	1.000	14072	14072	0	-1.604
	1 (A)	1.000	1.000	1.000	1502	1502	0	8.109
Freeze	0 (N)	0.951	0.485	0.642	14891	7222	7669	-1.571
	1 (A)	0.051	0.523	0.092	782	409	373	-1.567
Lag	0 (N)	0.991	0.891	0.938	10432	9293	1139	-1.615
	1 (A)	0.242	0.804	0.372	453	364	89	-0.699
Split Horizontal	0 (N)	0.987	0.966	0.976	15186	14663	523	-1.588
	1 (A)	0.735	0.879	0.800	1648	1448	200	1.036
Split Vertical	0 (N)	0.987	0.938	0.962	13257	12432	825	-1.596
	1 (A)	0.608	0.884	0.720	1445	1278	167	-0.164

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	-0.612	0.983	0.987	0.193	0.991	0.990
Flicker	8.109	1.000	1.000	0.096	1.000	1.000
Freeze	-1.525	0.504	0.050	0.050	0.499	0.487
Lag	-0.998	0.846	0.533	0.042	0.921	0.887
Split Horizontal	-0.817	0.921	0.906	0.098	0.966	0.957
Split Vertical	-0.860	0.911	0.894	0.098	0.969	0.933

Scores $log(\Delta_1)$



(a) Visual Artefact (b) Flicker (c) Freeze



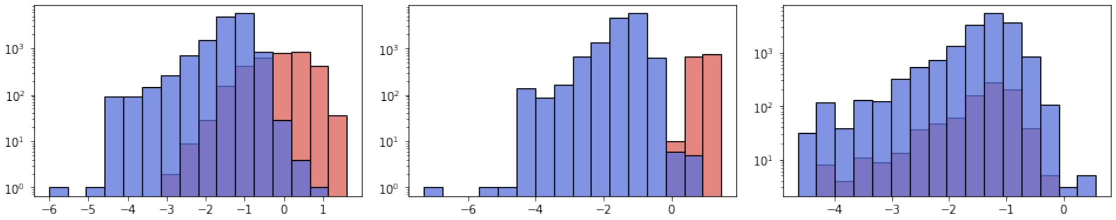
(d) Lag (e) Horizontal Split (f) Vertical Split

Figure D.4: S3N results for Pong $D = 256$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

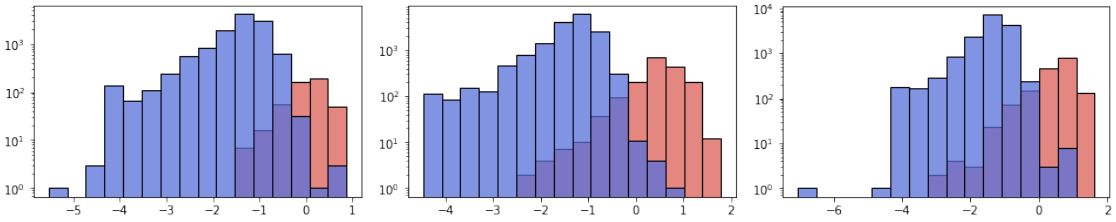
Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.956	0.939	0.948	14239	13369	870	-1.407
	1 (A)	0.759	0.817	0.787	3345	2734	611	-0.093
Flicker	0 (N)	1.000	1.000	1.000	13461	13456	5	-1.429
	1 (A)	0.997	1.000	0.998	1429	1429	0	0.945
Freeze	0 (N)	0.952	0.514	0.668	16871	8670	8201	-1.415
	1 (A)	0.051	0.506	0.093	876	443	433	-1.438
Lag	0 (N)	0.998	0.967	0.982	11790	11396	394	-1.458
	1 (A)	0.535	0.944	0.683	480	453	27	0.025
Split Horizontal	0 (N)	0.996	0.988	0.992	16164	15973	191	-1.412
	1 (A)	0.896	0.961	0.928	1718	1651	67	0.491
Split Vertical	0 (N)	0.994	0.979	0.987	15590	15264	326	-1.415
	1 (A)	0.829	0.947	0.884	1668	1580	88	0.497

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	-0.764	0.876	0.879	0.190	0.931	0.916
Flicker	0.247	1.000	1.000	0.096	1.000	1.000
Freeze	-1.276	0.510	0.050	0.049	0.499	0.513
Lag	-0.633	0.955	0.908	0.039	0.987	0.966
Split Horizontal	-0.493	0.974	0.977	0.096	0.990	0.986
Split Vertical	-0.585	0.963	0.962	0.097	0.984	0.976

Scores $log(\Delta_1)$



(a) Visual Artefact (b) Flicker (c) Freeze



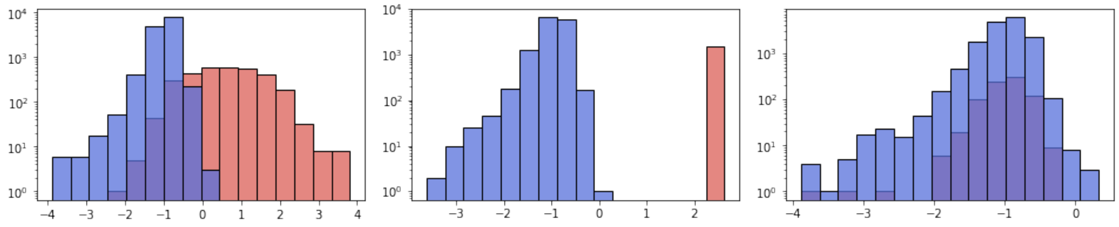
(d) Lag (e) Horizontal Split (f) Vertical Split

Figure D.5: S3N results for Qbert $D = 64$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

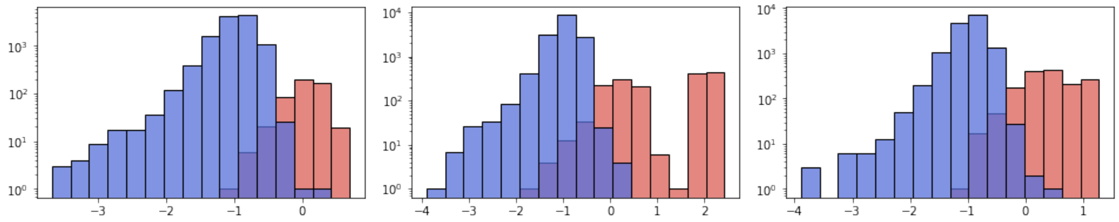
Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.977	0.974	0.975	13487	13134	353	-0.969
	1 (A)	0.889	0.900	0.895	3152	2837	315	0.620
Flicker	0 (N)	1.000	1.000	1.000	13849	13849	0	-0.970
	1 (A)	1.000	1.000	1.000	1487	1487	0	2.582
Freeze	0 (N)	0.952	0.518	0.671	15778	8172	7606	-0.996
	1 (A)	0.051	0.496	0.092	822	408	414	-0.986
Lag	0 (N)	0.999	0.992	0.995	11901	11805	96	-1.001
	1 (A)	0.833	0.972	0.897	494	480	14	0.046
Split Horizontal	0 (N)	0.998	0.993	0.995	15349	15240	109	-0.989
	1 (A)	0.937	0.981	0.958	1655	1623	32	1.161
Split Vertical	0 (N)	0.997	0.994	0.996	14539	14454	85	-0.969
	1 (A)	0.947	0.974	0.961	1563	1523	40	0.412

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	-0.556	0.936	0.949	0.189	0.968	0.960
Flicker	2.528	1.000	1.000	0.097	1.000	1.000
Freeze	-0.950	0.507	0.052	0.050	0.508	0.517
Lag	-0.474	0.982	0.972	0.040	0.995	0.991
Split Horizontal	-0.457	0.987	0.987	0.097	0.993	0.992
Split Vertical	-0.442	0.984	0.987	0.097	0.996	0.992

Scores $log(\Delta_1)$



(a) Visual Artefact (b) Flicker (c) Freeze



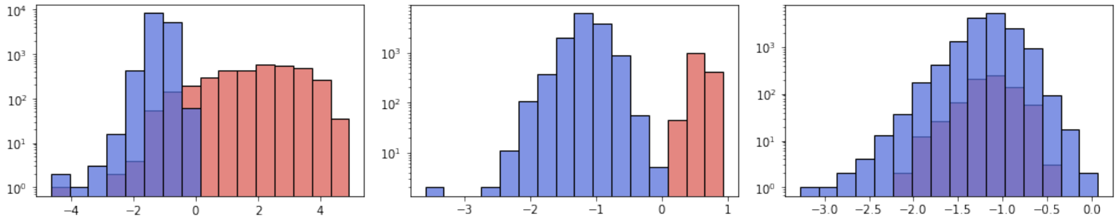
(d) Lag (e) Horizontal Split (f) Vertical Split

Figure D.6: S3N results for Seaquest $D = 64$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

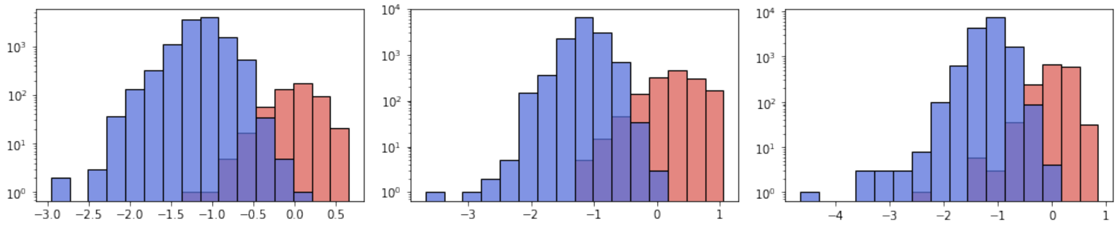
Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(log\Delta_1)$
Visual Artefact	0 (N)	0.988	0.990	0.989	14168	14033	135	-1.133
	1 (A)	0.960	0.950	0.955	3451	3280	171	1.960
Flicker	0 (N)	1.000	1.000	1.000	13420	13420	0	-1.136
	1 (A)	1.000	1.000	1.000	1455	1455	0	0.593
Freeze	0 (N)	0.953	0.583	0.723	14907	8686	6221	-1.142
	1 (A)	0.053	0.450	0.095	777	350	427	-1.136
Lag	0 (N)	0.999	0.989	0.994	11260	11137	123	-1.136
	1 (A)	0.801	0.972	0.878	508	494	14	0.003
Split Horizontal	0 (N)	0.997	0.986	0.992	13147	12967	180	-1.133
	1 (A)	0.888	0.977	0.930	1462	1428	34	0.286
Split Vertical	0 (N)	0.997	0.992	0.995	14254	14140	114	-1.142
	1 (A)	0.931	0.976	0.953	1570	1532	38	0.058

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
Visual Artefact	-0.568	0.970	0.977	0.196	0.984	0.983
Flicker	0.183	1.000	1.000	0.098	1.000	1.000
Freeze	-1.093	0.512	0.052	0.050	0.509	0.576
Lag	-0.604	0.981	0.976	0.043	0.996	0.988
Split Horizontal	-0.602	0.982	0.987	0.100	0.996	0.985
Split Vertical	-0.561	0.984	0.987	0.099	0.995	0.990

Scores $log(\Delta_1)$



(a) Visual Artefact (b) Flicker (c) Freeze



(d) Lag (e) Horizontal Split (f) Vertical Split

Figure D.7: S3N results for Space Invaders $D = 64$. Scores are computed as $log(\Delta_1)$, ignoring self-transitions.

Dataset	Class	Precision	Recall	F1-score	Support	True	False	$\mu(\Delta_1)$
Ped. 1	0 (N)	0.680	0.780	0.727	3130	2440	690	0.145
	1 (A)	0.807	0.716	0.759	4034	2888	1146	0.395
Ped. 2	0 (N)	0.536	0.908	0.674	359	326	33	0.273
	1 (A)	0.976	0.828	0.896	1639	1357	282	0.614

Dataset	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy	EER
Ped. 1	0.201	0.747	0.821	0.563	0.804	0.744	0.257
Ped. 2	0.340	0.867	0.981	0.820	0.916	0.842	0.135

Figure D.8: S3N results for the Pedestrian 1 and 2 datasets.

Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(\Delta_1)$
unintended shortcut	0 (N)	0.955	0.796	0.868	4529	3604	925	0.049
	1 (A)	0.329	0.729	0.454	623	454	169	0.128
player out of bounds	0 (N)	0.587	0.640	0.612	2910	1863	1047	0.047
	1 (A)	0.428	0.373	0.399	2094	782	1312	0.045
screen tearing	0 (N)	0.506	0.548	0.526	1396	765	631	0.046
	1 (A)	0.577	0.535	0.555	1606	859	747	0.052
texture corruption	0 (N)	0.741	0.456	0.564	3429	1562	1867	0.046
	1 (A)	0.356	0.654	0.461	1575	1030	545	0.051
geometry corruption	0 (N)	0.806	0.674	0.734	4278	2882	1396	0.046
	1 (A)	0.425	0.598	0.497	1727	1033	694	0.073
Z-fighting	0 (N)	0.823	0.727	0.772	3527	2563	964	0.046
	1 (A)	0.490	0.628	0.550	1477	927	550	0.101
terrain hole	0 (N)	0.813	0.750	0.780	692	519	173	0.048
	1 (A)	0.324	0.411	0.362	202	83	119	0.105
unintended object	0 (N)	0.929	0.783	0.850	4053	3173	880	0.047
	1 (A)	0.447	0.747	0.559	951	710	241	0.178
high force	0 (N)	0.678	0.788	0.729	2877	2268	609	0.047
	1 (A)	0.502	0.363	0.422	1690	614	1076	0.051
missing object	0 (N)	0.571	0.463	0.512	2815	1304	1511	0.046
	1 (A)	0.445	0.553	0.493	2189	1210	979	0.045

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
unintended shortcut	0.059	0.762	0.528	0.121	0.809	0.788
player out of bounds	0.050	0.489	0.419	0.418	0.436	0.529
screen tearing	0.045	0.541	0.594	0.535	0.557	0.541
texture corruption	0.041	0.546	0.349	0.315	0.564	0.518
geometry corruption	0.052	0.635	0.511	0.288	0.687	0.652
Z-fighting	0.056	0.675	0.629	0.295	0.740	0.697
terrain hole	0.060	0.555	0.420	0.226	0.472	0.673
unintended object	0.060	0.765	0.700	0.190	0.837	0.776
high force	0.061	0.535	0.503	0.370	0.452	0.631
missing object	0.041	0.506	0.422	0.437	0.493	0.502

Figure D.9: S3N results for the `Maze-v1` environment.

Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(\Delta_1)$
unintended shortcut	0 (N)	0.967	0.889	0.927	4586	4078	508	0.242
	1 (A)	0.458	0.757	0.570	567	429	138	0.772
player out of bounds	0 (N)	0.962	0.907	0.934	2925	2654	271	0.323
	1 (A)	0.879	0.950	0.913	2080	1975	105	12.732
screen tearing	0 (N)	0.600	0.649	0.624	1488	965	523	0.208
	1 (A)	0.625	0.576	0.600	1515	873	642	0.295
texture corruption	0 (N)	0.735	0.627	0.677	3451	2164	1287	0.210
	1 (A)	0.375	0.497	0.428	1554	773	781	0.303
geometry corruption	0 (N)	0.858	0.852	0.855	4313	3676	637	0.205
	1 (A)	0.630	0.641	0.635	1693	1085	608	1.351
Z-fighting	0 (N)	0.814	0.611	0.698	3931	2401	1530	0.209
	1 (A)	0.255	0.488	0.335	1074	524	550	0.245
terrain hole	0 (N)	0.970	0.968	0.969	696	674	22	0.214
	1 (A)	0.890	0.894	0.892	199	178	21	13.768
unintended object	0 (N)	0.938	0.984	0.960	4094	4027	67	0.199
	1 (A)	0.906	0.708	0.795	911	645	266	6.011
high force	0 (N)	0.990	0.995	0.993	2893	2879	14	0.196
	1 (A)	0.992	0.983	0.987	1675	1647	28	11.951
missing object	0 (N)	0.684	0.861	0.763	2843	2447	396	0.207
	1 (A)	0.723	0.478	0.576	2162	1034	1128	1.637

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
unintended shortcut	0.353	0.820	0.510	0.110	0.875	0.875
player out of bounds	0.740	0.928	0.983	0.416	0.986	0.925
screen tearing	0.229	0.611	0.706	0.504	0.654	0.612
texture corruption	0.225	0.558	0.493	0.310	0.593	0.587
geometry corruption	0.289	0.739	0.741	0.282	0.787	0.793
Z-fighting	0.219	0.546	0.310	0.215	0.569	0.584
terrain hole	0.436	0.931	0.934	0.222	0.938	0.952
unintended object	0.404	0.835	0.818	0.182	0.871	0.933
high force	0.468	0.989	0.996	0.367	0.996	0.991
missing object	0.301	0.642	0.726	0.432	0.687	0.696

Figure D.10: Contrastive results for the `Maze-v1` environment.

Bug	Class	Precision	Recall	F1-score	Support	True	False	$\mu(\Delta_1)$
unintended shortcut	0 (N)	0.957	0.860	0.906	4529	3897	632	0.385
	1 (A)	0.414	0.716	0.524	623	446	177	2.090
player out of bounds	0 (N)	0.723	0.666	0.693	2910	1938	972	0.385
	1 (A)	0.582	0.646	0.612	2094	1352	742	1.011
screen tearing	0 (N)	0.554	0.593	0.573	1396	828	568	0.312
	1 (A)	0.623	0.585	0.604	1606	940	666	0.388
texture corruption	0 (N)	0.775	0.627	0.693	3429	2150	1279	0.298
	1 (A)	0.426	0.603	0.499	1575	950	625	0.418
geometry corruption	0 (N)	0.837	0.719	0.773	4278	3074	1204	0.312
	1 (A)	0.484	0.654	0.557	1727	1130	597	0.757
Z-fighting	0 (N)	0.810	0.796	0.803	3527	2806	721	0.322
	1 (A)	0.532	0.555	0.543	1477	820	657	0.869
terrain hole	0 (N)	0.943	0.941	0.942	692	651	41	0.329
	1 (A)	0.799	0.807	0.803	202	163	39	28.391
unintended object	0 (N)	0.917	0.893	0.905	4053	3618	435	0.312
	1 (A)	0.589	0.656	0.621	951	624	327	2.195
high force	0 (N)	0.980	0.982	0.981	2877	2824	53	0.308
	1 (A)	0.969	0.966	0.967	1690	1632	58	19.432
missing object	0 (N)	0.657	0.527	0.585	2815	1484	1331	0.309
	1 (A)	0.515	0.646	0.573	2189	1415	774	0.360

Bug	Threshold	G-Mean	PR-AUC	PR-NS	ROC-AUC	Accuracy
unintended shortcut	0.402	0.785	0.596	0.121	0.849	0.843
player out of bounds	0.355	0.656	0.575	0.418	0.682	0.657
screen tearing	0.322	0.589	0.679	0.535	0.628	0.589
texture corruption	0.313	0.615	0.464	0.315	0.655	0.620
geometry corruption	0.348	0.686	0.633	0.288	0.750	0.700
Z-fighting	0.381	0.665	0.648	0.295	0.723	0.725
terrain hole	0.514	0.871	0.875	0.226	0.915	0.911
unintended object	0.433	0.765	0.725	0.190	0.838	0.848
high force	0.574	0.974	0.994	0.370	0.995	0.976
missing object	0.291	0.584	0.549	0.437	0.622	0.579

Figure D.11: Hybrid results for the `Maze-v1` environment.