Detecting Anomalous Collective Behaviours in Simulated Multi-Agent Environments

by

Everton Schumacker Soares

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Everton Schumacker Soares, 2020

Abstract

Several Artificial Intelligence (AI) techniques such as machine learning, evolutionary computing, and Artificial Life (A-life) have been increasingly used to generate emergence of novel behaviours in multi-agent simulations (e.g., commercial games). However, automatically detecting emergent behaviours and recognizing which ones are indeed novel/anomalous still poses a challenging problem. The solution for such a problem may potentially help lead to improvements in quality assurance control (e.g., detection of bugs) and security (e.g., detection of suspicious behaviour in surveillance footage). Some of previously published attempts to detect anomalous behaviour in simulations relied on machine learning techniques. For example, the use of supervised learning models to detect anomalous behaviours requires labelled data not always available at the training time due to the rarity of anomalous behaviours. On another hand, the use of unsupervised learning models usually relies on specific types of behaviour patterns or on having access to internal simulation states. Our approach presented in this thesis uses only unlabelled sequences of readily available visualization frames from simulated multi-agent environments to train a deep variational autoencoder (i.e., a deep artificial neural network). After being trained, the autoencoder can detect anomalous behaviours by comparing its reconstruction error against a threshold. We tested our approach in a predator/prey A-life environment where it proved viable, being even robust to a certain amount of pollution (i.e., the inclusion of anomalous data) in the training data. As a case study, we also applied our approach to a video game where the results are yet inconclusive.

Preface

This thesis is mostly an original work by Everton Schumacker Soares advised by Vadim Bulitko. However, some material here presented is adapted from previously published work co-authored with Kacy Doucet and Morgan Cselinacz from the University of Alberta. We also collaborated with Terence Soule, Samantha Heck, Landom Wright from Polymorphic Games at the University of Idaho.

Chapter 3 and 6 have some material adapted from work published as Everton Schumacker Soares, Vadim Bulitko, Kacy Doucet, Morgan Cselinacz, Terence Soule, Samantha Heck, and Landom Wright, "Learning to Recognize A-life Behaviours", in Proceedings of Advances in Cognitive Systems (ACS) Poster Collection, 2018. I was responsible for programming the approach and the baseline, running the experiments, analyzing the results, writing the paper, and presenting the work at the conference. Vadim Bulitko assisted in designing the experiment, collecting the data for the predator/prey simulation and writing the paper. Morgan Cselinacz and Kacy Doucet were responsible for the statistical analyses as well as assisting in the literature review and writing the paper. Terence Soule, Samantha Heck, and Landom Wright provided the video game data for the experiment and assisted in writing the paper.

Chapter 3, 4, and 5 have some material adapted from work published as Everton Schumacker Soares and Vadim Bulitko, "Deep Variational Autoencoders for NPC Behaviour Classification", in Proceedings of the IEEE conference on Games (CoG), pages 1-4, 2019. I was responsible for designing and performing the experiments, collecting the data, programming

the approach, analyzing the results, and writing the paper. Vadim Bulitko assisted with the design of the experiments, analyzing the result, and writing the paper.

Dedicated to my mother, Marlene Schumacker Soares, and father, Edirlei José Silva Soares.

Acknowledgements

I owe all my accomplishments in this thesis and my graduation to my supervisor Prof. Vadim Bulitko. He was willing to take me in since the beginning of my program, having the patience and wisdom to guide me in the path of deep learning and to help me overcome the numerous challenges in these last few years. I want to thank Nvidia as well, for providing the GPUs used in my experiments.

I appreciate the help from Terence Soule, Samantha Heck, Landon Wright from Polymorphic Games, University of Idaho, for providing free access to their games (to collect data used in this thesis) and also for collaborating on our published paper. I would also like to thank Joshua Sirota, Sergio Poo Hernandes, Kacy Doucet, and Devon Sigurson for the numerous discussions and help in our lab, which allowed me to overcome obstacles in my work. I want to give a special thanks to Morgan Cselinacz, whom I had the pleasure to supervise while she helped with my experiments.

I would also like to thanks my friends Isadora Sophia Garcia Rodopoulos, Gabriel Massaki Wakano Bezerra, Augusto Morgan, and Thiago Henrique Rosales Marques that have been a second family, helping me keep my sanity.

Finally, I would like to thank my mother, Marlene Schumacker Soares, my father Edirlei José Silva Soares, my sister Franciele Schumacker Soares, and my brother Edilmar Schumacker Soares. They always give me all the support for me to pursue my dreams.

Table of Contents

1 Introduction **1**
1.1 Contributions . . . 7
1.2 Thesis organization . . . 8

2 Problem Formulation **9**
2.1 Simulation Framework . . . 9
2.2 Detecting Anomalous Collective Behaviour . . . 11
2.3 Constraints on a Solution . . . 12

3 Related Work **13**
3.1 Detection of Anomalous Population Behaviours . . . 13
3.2 Anomaly Detection in Video Games . . . 15

4 Proposed Approach **17**
4.1 Autoencoders (AE) . . . 17
4.1.1 Stochastic Autoencoders . . . 18
4.1.2 Anomaly Detection with AE . . . 19
4.1.3 Learning a Useful Compression . . . 20
4.1.4 A drawback of an Autoencoder . . . 21
4.2 Variational Autoencoders . . . 23
4.2.1 VAE as a Stochastic Model . . . 24
4.2.2 Training a VAE . . . 25
4.2.3 Anomaly Detection with VAE . . . 28
4.3 Frame Averaging . . . 28
4.4 Summary . . . 29
4.5 The hypothesis . . . 31

5 Empirical Evaluation and Results **32**
5.1 A-life Simulation . . . 34
5.1.1 Input Data . . . 35
5.1.2 Behaviour Classes . . . 35
5.1.3 VAE Implementation . . . 39
5.1.4 Multiple Trials and Majority Class Accuracy . . . 41
5.1.5 Experiment 1: Pure Training Data . . . 41
5.1.6 Experiment 2: Polluted Training Data . . . 46

6 Anomalous Behaviour Detection in a Video Game 48
6.1 *Project Hastur* . . . 48
6.2 Collecting Data . . . 51
6.3 Labeling the Data . . . 52
6.4 Training VAE . . . 53
6.4.1 Results and Discussion . . . 54

7 Future Work 60
7.1 Future Research Directions . . . 60

8 Conclusion 63

Bibliography 64

A A-life Environment: Details 67
A.1 A-life Environment: Parameters . . . 67

B VAE Implementation 70

C *Project Hastur*: Details 96
C.1 Playstyle . . . 96
C.2 Experiment Mode . . . 97

List of Tables

5.1 Mean frequencies of behaviours in our A-life environment. 38
5.2 Number of runs used on each data split. 38
5.3 Accuracy of behaviour classifiers on pure data. B_{R} was collected with $\eta = 50\%$. The top half of the table has the results for the detection of B_{F} anomaly, while the bottom half has the results for the detection of B_{R} anomaly. . . . 44

6.1 Frequencies for each observed behaviour B_{ψ} in the *Project Hastur* data. . . 52
6.2 Number of game runs used in each data split. 56
6.3 Accuracy of behaviour classifiers on *Project Hastur* data. 56

A.1 Parameters for our A-life environemnt. 69

C.1 Values for the parameters on *Project Hastur* experiment mode. 104

List of Figures

1.1 The first *Creatures* game (Creature Labs, 1996). 2
1.2 *Darwin's Demons* (Polymorphic Games, 2017) use GA to evolve NPCs according to the player strategy and settings. 3
1.3 A commercial video game: *Project Hastur* (Polymorphic Games, 2019). . . . 5
1.4 Two behaviours emergent in an A-life predator/prey simulation: prey that actively seek for food (right), and prey that has a distaste for food (left). . 6

2.1 The problem of detecting anomalous collective behaviours in multi-agent simulations. The simulation produce states S_t, while a sequence of $\tau + 1$ states are used to generate features $f(S_{t-\tau}, \ldots, S_t)$. The latter is used as an input to the anomaly detector D_ϕ. 10

3.1 Simulation frames taken from an predator/prey A-life environment used in previous work (Soares et al., 2018). Each image presents a distinct classes of emergent population behaviour in the populations. 15

4.1 An autoencoder. 18
4.2 A normal image (left) used to generate a surrogate anomaly (right) via a random permutation of its pixels. 19
4.3 The flow of information in anomaly detection via a variational autoencoder. 20
4.4 Images with different complexity. More information is necessary to reconstruct a cluster of points (right) than a single point (left). 21
4.5 Twenty points distributed randomly (left), in clusters (center), and in a single line (right). 22
4.6 An image with normal behaviour (left) with agents (triangles) more scattered than an image with anomalous behaviour (right). 23
4.7 A variational autoencoder. 25
4.8 Training variational autoencoder. 26
4.9 The flow of information in anomaly detection via a variational autoencoder. 29
4.10 A image representing the environment state $I(S_t)$ on time step t (left) and its frame averaging $f(S_{t-\tau}, \ldots, S_t)$ considering the past $\tau = 4$ frames (right). Gray dots represent agents within the environment. 30

4.11 Our proposed approach takes unlabelled image visualization from simulated multi-agent environments and used than to train a VAE. We also tune a threshold by selecting the model with best accuracy when detecting surrogate anomalies. We perform frame averaging in the images to add temporal information. 30

5.1 Our predator-prey A-life environment. The grass is shown via shades of green. Orange, purple and blue circles represent grass-liking rabbits, grass-disliking rabbit and wolves respectively. In the left image the majority of the rabbit population are grass-liking rabbits (i.e., the normal behaviour). In the right image the majority is comprised of the grass-disliking rabbits (i.e., an anomalous behaviour). 34
5.2 A single visualization frame from our predator-prey A-life environment with the resource and agent gene visualizations removed (left). A weighted moving average of the frame and its four predecessors captures movements of the agents (right). 36
5.3 Top images are A-life visualization frames. They are converted to input images for the autoencoder and shown in the bottom. The three columns show representatives of classes B_{L}, B_{F}, and B_{R} correspondingly. 37
5.4 Our VAE: the encoder (top) and the decoder (bottom). The encoder consists of a batch-normalization layer, 8 convolutional layers, 2 max-pooling layers and 3 fully connected layers. The decoder has 1 fully connected layer and 8 transposed convolutional layers. Orange rectangles have the stride 1×1 and padding, red rectangles have the same stride but no padding, green rectangles have the stride 2×2 and no padding. Max-pooling layers have the stride 2×2 and all layers have ReLu activation units. 40
5.5 VAE's and GoogLeNet detection accuracy variation according to the probability η of injecting random agents. The error bars show standard deviation. 45
5.6 Effects of pollution degree ρ on VAE test accuracy. 46

6.1 Screen capture taken from *Project Hastur*. 49
6.2 *Project Hastur*: the main and base towers are the white/gray structures surrounded by turrets (purple, blue and red). Proteans are the spider-like NPCs, while civilians are the human-like NPCs. 50
6.3 A original image frame taken from *Project Hastur* (left) and the corresponding features as seen by our VAE (right). 51
6.4 Normal B_{L} image (i.e., no bipedal Proteans nor NPCs capable of swimming or jumping). 53
6.5 Image from class B_{B} (i.e., bipedal Proteans). Red circle marks the presence of bipedal agents. 54
6.6 Image from class $B_{\mathrm{S,J}}$ (i.e., Proteans capable of jumping or swimming). Red circle marks the presence of an agent jumping. 55
6.7 Features $f\left(S_{t-\tau}, \ldots, S_t\right)$ for classes B_{L}, B_{B}, and $B_{\mathrm{S,J}}$ respectively. 56

6.8 The top row shows an image from B_{N} correctly by the $\mathrm{VAE_u}$. The bottom row shows a B_{N} incorrectly classified by the $\mathrm{VAE_u}$. The left column shows the original images, the center column has the images reconstructed by our $\mathrm{VAE_u}$, and the right column has the residual image between original and reconstruction. 57
6.9 The top row shows an image from B_{B} correctly by the $\mathrm{VAE_u}$. The bottom row shows a B_{B} incorrectly classified by the $\mathrm{VAE_u}$. The left column shows the original images, the center column has the images reconstructed by our $\mathrm{VAE_u}$, and the right column has the residual image between original and reconstruction. 58
6.10 The top row shows an image from $B_{\mathrm{S,J}}$ correctly by the $\mathrm{VAE_u}$. The bottom row shows a $B_{\mathrm{S,J}}$ incorrectly classified by the $\mathrm{VAE_u}$. The left column shows the original images, the center column has the images reconstructed by our $\mathrm{VAE_u}$, and the right column has the residual image between original and reconstruction. 59
6.11 A frame average image from our A-life simulation (left), the correspondent reconstruction given by a $\mathrm{VAE_u}$ trained on *Project Hastur* data (middle), and the residual image between them (right). 59

7.1 *Project Hastur*'s default camera angle. 61

C.1 *Project Hastur*: beginning of experiment mode, before the player place any turret or tower. 98
C.2 *Project Hastur*: turret and tower configuration on Generation 0. 98
C.3 *Project Hastur*: turret and tower configuration on Generation 2. 99
C.4 *Project Hastur*: turret and tower configuration on Generation 6. 99
C.5 *Project Hastur*: turret and tower configuration on Generation 11. 100
C.6 *Project Hastur*: turret and tower configuration on Generation 17. 100
C.7 Parameters for *Project Hastur* experiment mode. 103

Chapter 1

Introduction

Artificial Intelligence (AI) is increasingly present in our lives in the form of smartphone assistants, self-driving cars, non-playable characters (NPC) in video games, and other types of technology. Consequently, emergent interactions between AI agents (i.e., AI capable of taking actions) and humans can significantly affect our lives and help us address various societal challenges. Machine Learning, Reinforcement Learning, and bio-inspired computing — such as Genetic Algorithms (GA), Genetic Programming, and Artificial Life (A-life) — are powerful frameworks to study novel behaviours emerging from interactions among AI agents (Sweetser, 2008). Machine Learning and Reinforcement Learning allow AI agents to learn a task during their life time through the optimization of target functions via inference over datasets or via the maximization of expected pleasure/pain reward signals, respectively, which leads to emergent behaviours learned through experience. On the other hand, GA and A-life mimic the natural (i.e., Darwinian) evolution process with natural selection via a explicit definition of a fitness function or via the artificial simulation of natural/biological agents in real life, respectively, leading to emergent behaviour through evolution of a population of agents.

Previous work already showed how these AI techniques can be used to mimic real-life emergence of AI behaviours in multi-agent simulations. For example, Chen et al. (2007) proposed a simulation framework by modelling the relationship between simple and complex

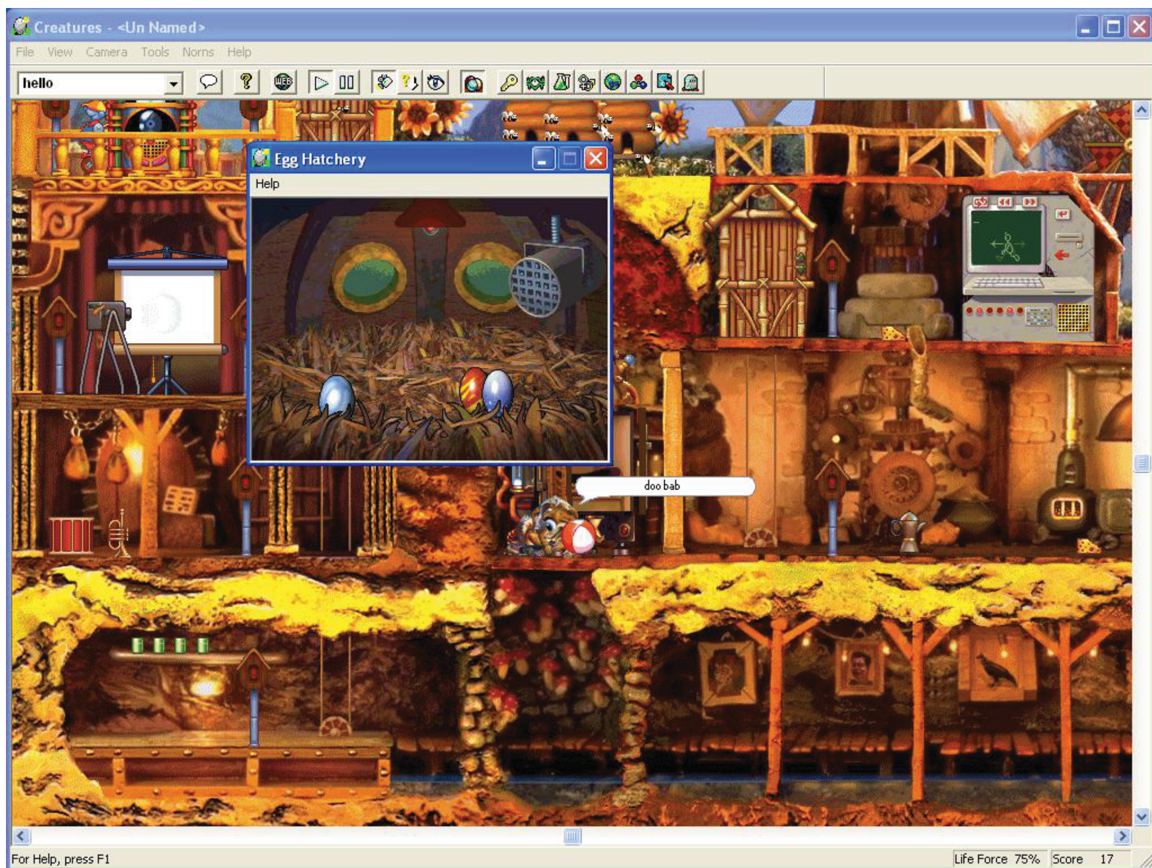

Figure 1.1: The first *Creatures* game (Creature Labs, 1996).

events as a rule-based graph. They evaluated their approach in a predator/prey simulation to detect and study how collective and social behaviours emerge. Ryan et al. (2015) created a multi-agent framework, *Talk of the Town*, to study emergence of communication, which led to agents capable of misremembering information and lying to each other. Bulitko et al. (2017) showed how A-life predator/prey environments can be used to procedurally generate ambience NPCs (e.g., rabbits in a video game environment) with interesting behaviours to the player.

In video games, emergent NPC behaviour can increase immersion by allowing players to interact with NPCs changing the course of their evolution. For example, the game *Creatures* (Creature Labs, 1996) implement populations of A-life agents, called *norns*, each one of them with specific biochemistry and neural network brain (Figure 1.1). Players can

Figure 1.2: *Darwin's Demons* (Polymorphic Games, 2017) use GA to evolve NPCs according to the player strategy and settings.

interact with norms, nursing and breeding them, which leads to several types of evolved behaviours according to the player's actions. This idea proved to be quite successful, leading to the development of other games from the same series: *Creatures 2* (Creature Labs, 1998) and *Creatures 3* (Creature Labs, 1999). Although the norn's evolution was heavily controlled by the players, some interesting behaviours emerged. One such behaviour caused a controversy in the game community when some players created norns with alcoholic behaviour by breeding the creatures with a predisposition in their genes (Creatures Wikia, 2004).

Another example of emergent behaviour can be seen in the commercial game *Darwin's Demons* (Polymorphic Games, 2017), which uses GA to evolve behaviours of enemy NPCs according to each player strategy (Figure 1.2). In this game, the player controls a spaceship with the goal of shooting and destroying NPCs capable of moving on the screen. Previous work (Soule et al., 2017) demonstrated that not only NPC behaviours can emerge accordingly to each playstyle, but also that the adaptive behaviours of NPC increase the difficulty

and replay value of video games. Furthermore, they also noticed that by following a specific strategy that targets more hostile NPCs (e.g., shooting enemies with a non-basic weapon, shooting enemies in the centre of the screen), players can promote the emergence of domesticated NPCs (Soule et al., 2017) that stay in the top corners of the screen. This phenomenon can be game-breaking from a design perspective since non-threatening enemies in a game impose no challenge for the player. As the developers said in an interview (Ronson, 2016):

> "Some very clever players — I mean these are like super, ultra gamers, and they knew it was an evolutionary game — set out to domesticate the aliens. They shot all of the nasty ones first and then they let the dumb ones that didn't fire very much and just stayed up at the top and didn't do anything live for a really long time. And, after several generations, they had [the equivalent of] space cows."

If the automatic detection of emergent domesticated NPC was possible, then this type of behaviour could be tracked and possibly prevented more easily.

Project Hastur (Polymorphic Games, 2019) – another commercial game from the creators of *Darwin's Demons* – also uses GA to evolve NPCs. In this tower-defence game, players must place auto-shooting turrets to defend their towers from spider-like NPCs (Figure 1.3). In this human-versus-AI type of game, the placement of turrets may directly impact the emergence of NPC behaviour, such as NPCs capable of swimming rivers to attack the towers from an unprotected angle. These NPC capable of adapting to playstyles may lead to interesting behaviours (e.g., swimmers) or potential exploit (e.g., domesticated enemies). Effectively detecting emergent behaviours may help to improve quality assurance by preventing potential game exploits and also improve the game experience by replicating interesting behaviours in later games.

The idea of using evolution as a procedural content generation for NPC behaviour was already explored in previous work (Bulitko et al., 2017; Soares et al., 2018; Soares and

Figure 1.3: A commercial video game: *Project Hastur* (Polymorphic Games, 2019).

Bulitko, 2019). Their predator/prey A-life environment evolved two types of behaviours: prey that actively seeks food and was scared of predators (Figure 1.4, right) and prey that avoided food (Figure 1.4, left). Although the first type of behaviour consistently emerged in every simulation run, the second type appeared only in some runs, which was so rare (i.e., infrequent) that it can be considered an outlier (i.e., anomalous) behaviour relative to other behaviours. Although these prey had a distaste for food, they surprisingly survived. This phenomenon happened because this prey type enjoyed following other prey around and forcefully ate any food found along the way. Since the researchers first thought that this behaviour was a bug, they had to perform a time-expensive manual code inspection to understand the nature of such an anomalous behaviour. An automatic detection tool could help researchers to discover other interesting behaviours present in such simulations more quickly.

Ackley and Littman (1991) not only used A-life to simulate/model a predator/prey environment, but also genetically evolved the value function used by each Reinforcement

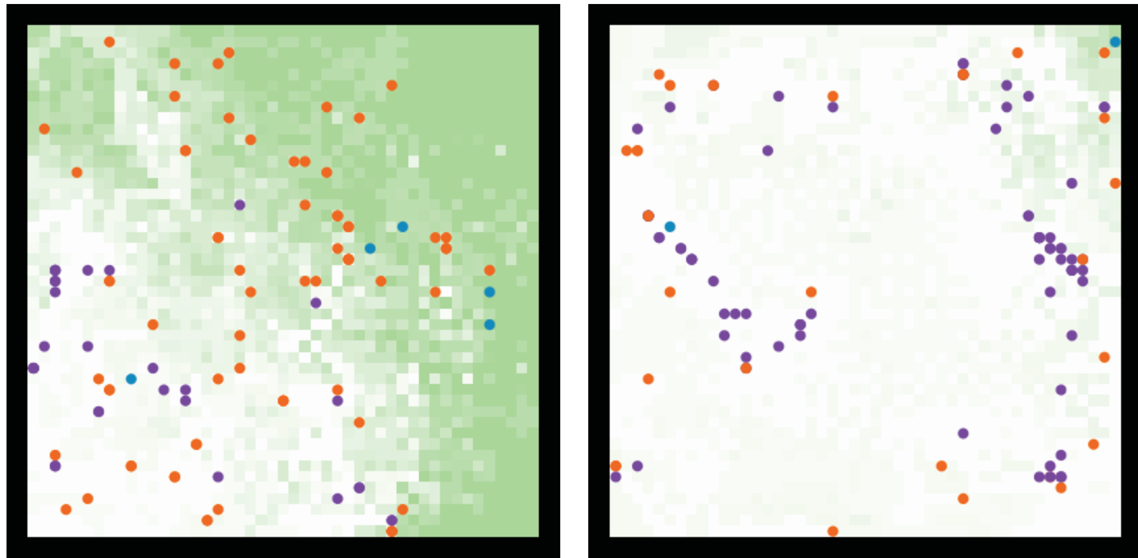

Figure 1.4: Two behaviours emergent in an A-life predator/prey simulation: prey that actively seek for food (right), and prey that has a distaste for food (left).

Leaning agent in their environment. The experiments led to the discovery of emergent novel behaviours such as prey that enjoyed hitting their heads against walls, or prey with an affinity towards predators. Their manual behaviour detection and analysis led to re-discovery of important evolutionary mechanisms (e.g., gene shielding).

Detecting the emergence of such unexpected/surprising behaviour may be not only useful to increase gameplay value but also for quality assurance in video games. While human observers identified these behaviours, it may not always be simple to do so. Detecting emergent behaviours may be challenging due to the intrinsic unpredictability of random algorithms/components often associated with emergent computing. Manual detection can also be prohibitively time-consuming. Even when viable, such manual detection may be prone to human error. In this thesis, we consider any emergent behaviour to be novel when it deviates from the usual behaviours; thus, we assume that novel emergent behaviours are anomalies in the simulation. For this reason, we use anomalous, novel, and abnormal behaviours as interchangeable terms henceforth.

Our thesis proposes the automatic detection of emergent/novel behaviours in multi-agent environments, such as A-life environments and video games. In these simulations, every AI agent is capable of perceiving its surroundings through sensory input and taking actions that might modify the environment's internal states. The environment is every component of the simulation external to the agents, such as natural resources (e.g., grass, water) or obstacles (e.g., walls, rocks). These environments run in discrete time steps (i.e., frames), during which agents interact among themselves, the environment and with a possible human-controlled avatar. We use unlabeled visualization frames taken from two different environments: a predator/prey A-life environment (Figure 1.4) and a commercial game (Figure 1.3). Such data can be collected automatically without any human effort as the behaviour labels are not required. We train a specific type of artificial neural network known as deep variational autoencoder and tune a threshold parameter to detect unusual/anomalous data. Finally, we evaluate our approach empirically in two different settings: an A-life predator/prey environment and a commercial video game.

1.1 Contributions

In this thesis, we made the following contributions:

- We developed a deep convolutional variational autoencoder to detect emergent novel population behaviour in screen-capture streams taken from a predator/prey A-life simulation. Our approach only requires normal data during its training while still being able to detect abnormal behaviours reliably.
- We performed an investigation on how the mixture of normal and abnormal unlabeled data samples may impact the detection performance. We also showed that the autoencoder's performance degrades gracefully as the amount of mislabeled data increases.

- We conducted a case study of how well our approach performs when applied to a video game with genetically evolved NPCs. There, our approach was used to detect anomalous behaviours rarely seen during playtime.

1.2 Thesis organization

We first formulate the problem of detecting anomalous emergent population behaviour in Chapter 2. The constraints on a problem solution are also specified in that chapter. Chapter 3 gives a literature review with the current methods used to detect anomalous behaviours. We consider methods used in video analysis as well as games and A-life simulations. Our proposed methodology is described in Chapter 4. It contains the mathematical definition of an autoencoder and the application of such a model to anomaly detection. This methodology is then evaluated in two different settings: a predator/prey A-life simulation (Chapter 5) and a commercial game (Chapter 6). Chapter 7 propose future work to follow from this research and Chapter 8 presents the conclusions that can be drawn from our experiments. Further algorithms and experimental details are found in Appendices A, B, and C.

Chapter 2

Problem Formulation

This chapter formulates the problem being addressed. First, we state a general framework for the simulations containing emergent behaviours to be detected. After that, we define the concept of *collective or population-level behaviour*, which are the targets of our approach. We then state the problem of detecting anomalous collective behaviour in simulated multi-agent environments. Finally, we state the properties expected from the possible solutions for the problem being addressed. A diagram summarizing the problem formulation is found in Figure 2.1.

2.1 Simulation Framework

According to Drogoul and Ferber (1994), a "simulation consists of artificially reproducing natural phenomena". We expand this definition to match the problem being addressed so that simulations may also artificially implement fictitious scenarios and worlds (e.g., video games). Both types must be composed of artificially (i.e., AI) or human-controlled agents and an environment. Each agent is a component of the simulation that has an internal state and an input sensor to perceive the external environment. The agent is capable of taking actions that may affect the simulation. The environment is every component of the simulation external to the agents.

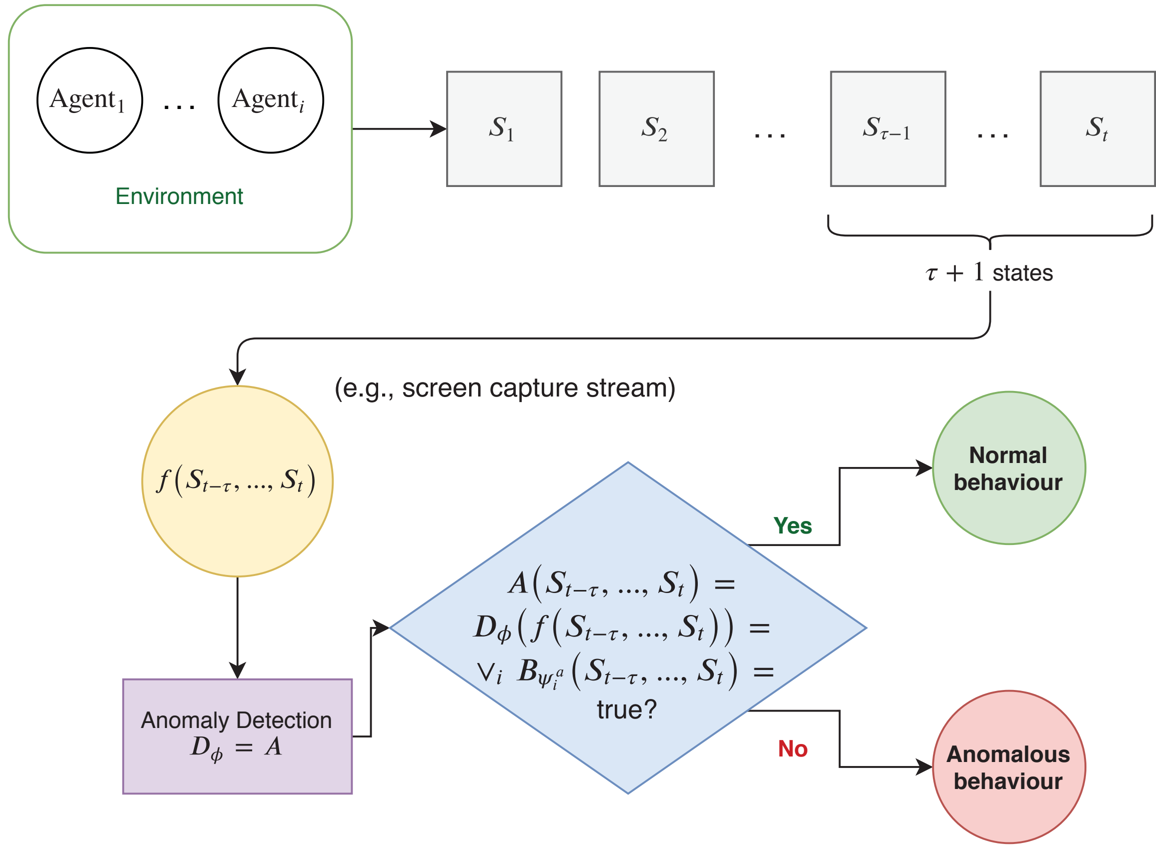

Figure 2.1: The problem of detecting anomalous collective behaviours in multi-agent simulations. The simulation produce states S_t, while a sequence of $\tau+1$ states are used to generate features $f(S_{t-\tau}, \ldots, S_t)$. The latter is used as an input to the anomaly detector D_ϕ.

A simulation may contain one or more populations of agents. For example, a population of predators may be defined as the set of all agents that hunt and eat other agents (i.e., prey). In a simulation, agents may interact with the environment and with each other at discrete time steps, or *simulation frames*, $t = 1, 2, 3, \ldots$. For each $t \in \mathbb{N}$, the simulation has a state defined as:

Definition 2.1.1. The simulation state S_t is all information needed to continue the simulation forward (i.e., for any time step higher than t).

Definition 2.1.2. $\mathbb{S} = \{(S_{t-\tau}, \ldots, S_t) \mid t \in \mathbb{N}, \tau \in \mathbb{N}\}$ is the set of all possible finite sequences of simulation states.

2.2 Detecting Anomalous Collective Behaviour

We consider detection of emergent anomalous/novel *collective behaviours*, defined as:

Definition 2.2.1. Given a collective behaviour ψ, $B_\psi : \mathbb{S} \to \{\text{true, false}\}$ is the collective, or population-level, behaviour predicate that takes a sequence of system states $(S_{t-\tau}, ..., S_t) \in \mathbb{S}$ and outputs true if the behaviour ψ is observed in the sequence, and false otherwise.

For example, consider the clumping behaviour in which prey tends to cluster together to increase their individual chances of survival. Thus, $B_{\text{clumping}}\left(S_{t-\tau}, \ldots, S_t\right) = \text{true}$ means that the simulation state sequence $(S_{t-\tau}, \ldots, S_t)$ has the clumping behaviour.

Let us now assume a human is observing a sequence of frames $(S_{t-\tau}, \ldots, S_t)$. We aim to detect emergent behaviours ψ surprising (i.e., unexpected, rarely seen in multiple simulation runs) to this observer. For example, the space cows described in Chapter 1 was a behaviour unexpected by the developers of *Darwins Demons*; thus, it was a surprising behaviour to them. This definition of surprising and non-surprising behaviour is dependent on the observer. Consider two behaviours ψ_1 and ψ_2. An observer that has only seen behaviour ψ_1 during n simulation runs may assume that this is the normal behaviour for this particular simulation. However, if the same person observes ψ_2 in a particular sequence of states $(S_{t-\tau}, \ldots, S_t)$ in the run $n+1$, they might be surprised by the new behaviour. A second observer, however, may have seen ψ_2 more often in the same simulation might not be surprised by it. Then, we may assume that a behaviour ψ with less probability of being observed in any arbitrary sequence $(S_{t-\tau}, \ldots, S_t)$ is the one with more chances of being considered surprising to any arbitrary observer. Hence, the more anomalous (i.e., less frequent) a behaviour ψ is, the more chances it has of surprising an arbitrary observer.

Definition 2.2.2. $A : \mathbb{S} \to \{\text{true, false}\}$ is the anomaly predicate. A sequence of simulation states $(S_{t-\tau}, ..., S_t) \in \mathbb{S}$ is anomalous if $A\left(S_{t-\tau}, ..., S_t\right) = \text{true}$. The sequence is normal if $A\left(S_{t-\tau}, ..., S_t\right) = false$.

Since a sequence of simulations frames is anomalous when it contains at least one anomalous behaviours from anomalous behavious $\{\psi_1^a, \ldots, \psi_m^a\}$ then $A(S_{t-\tau}, \ldots, S_t) = \vee_i B_{\psi_i^a}$. We measure A's performance via detection accuracy:

Definition 2.2.3. Given a dataset of simulation state sequences $(S_{t-\tau}, \ldots, S_t) \in \mathbb{S}$, the detection accuracy is the number of correctly classified state sequences divided by the dataset size.

If the anomaly detector A is represented in a parameterized framework (e.g., a deep neural network) then the problem of detecting emergent anomalous behaviours becomes the task of finding a parameter vector ϕ that parameterize the function D_ϕ such that $D_\phi(S_{t-\tau}, \ldots, S_t) = A(S_{t-\tau}, \ldots, S_t), \forall (S_{t-\tau}, \ldots, S_t) \in \mathbb{S}$. We must then find ϕ that maximizes the mean detection accuracy of D_ϕ.

2.3 Constraints on a Solution

First, our anomaly/abnormality detector will take features of state sequences as its input. So instead of $A(S_{t-\tau}, \ldots, S_t)$ we will be considering $A(f(S_{t-\tau}, \ldots, S_t))$, where $f : \mathbb{S} \to \mathbb{R}^k$ is a feature function. We constrain the anomaly detection problem by requiring f to be readily available. For example, f can be an existing game or simulation visualization. Secondly, anomalous behaviours may be unknown when searching for parameters ϕ, so a solution approach should not require labelled data for both normal and anomalous data. Finally, the candidate solution should be robust to some inclusion of anomalous behaviours mislabelled as normal behaviours. The detection accuracy should then gracefully degrade when the percentage of anomalous behaviours mislabelled as normal increases.

Chapter 3

Related Work

First, we will review the general application of anomalous population behaviour techniques for several types of environments in Section 3.1. After that, we will discuss some more specific approaches applied to video games in Section 3.2. Some material in this chapter is adapted from our published paper (Soares and Bulitko, 2019).

3.1 Detection of Anomalous Population Behaviours

Population behaviour detection is commonly applied in crowded scenes recorded by surveillance cameras. A social-force model was used to detect anomalous behaviour in publicly available videos composed of escape events (Mehran et al., 2009). Although this approach was capable of detecting population-level anomalies, it relies on specific types of particle motion generated by the optical flow (i.e., pixel motion) in surveillance image frames. Doing so limits the application of the technique, since some anomalies may be independent of the force and velocity field of these particles (e.g., purely morphological anomalies).

Cascades of Dynamic Bayesian Networks (CasDBNs) were used to detect anomalous behaviours in crowded scenes and also to discriminate among different types of anomalies (Loy et al., 2011). CasDBNs are composed of hidden Markov model layers responsible for modelling the behaviours into atomic actions and its components, which is performed accordingly

to spacial and morphological features. However, this approach has two limitations when applied to our problem. First, the model does not take readily available images as input; instead, it takes a semantic decomposition of complex behaviours according to their spatial-temporal visual context. This decomposition requires an initial conversion of frames into blobs composed of object centroids, width and height of bounding boxes, occupancy, the ratio of the dimension, the mean optical flow of the bounding box, and scaled optical flow. Second, the detection depends on a pre-fixed threshold, which is computed via cross-validation on a dataset composed of normal data and real anomalies. If anomalous data is not available, the threshold may not be computed optimally.

Chen et al. (2007) proposed a rule-based framework for the detection and analysis of emergent behaviour in simulations. They modelled an agent-based simulation as a graph in which nodes are events and edges are relationships between simple and complex events. For example, in their predator/prey simulation composed of lions and antelopes, each event can be defined by a set of rules. The emergent behaviours can then be explaining by the cause-effect relation between events in the simulation. For example, the behaviour of starvation of lions is related to the rule of lions eating antelopes and lions over-hunting. Their approach, however, does not solve our problem since it does not take readily available features of simulation states. Furthermore, the modelling of the event graph may not be trivial, since emergent behaviour may depend on partially observable or even wholly hidden rules, such as prey with a genetic affinity to predators (Ackley and Littman, 1991).

More recent work used autoencoders to perform population behaviour anomaly detection (Ribeiro et al., 2018). However, their anomalies involved substantially morphologically different agents (e.g., bicycles in anomalous images versus pedestrians in normal images). Such a distinction between anomalous and normal data may not generally be the case for emergent population behaviours. Previous work (Soares et al., 2018) already showed that agents in an A-life setting might have the same shape among themselves and yet present anomalous population behaviour. Indeed, as shown in Figure 3.1, although all agents have the same shape (triangles), their population behaviour differs.

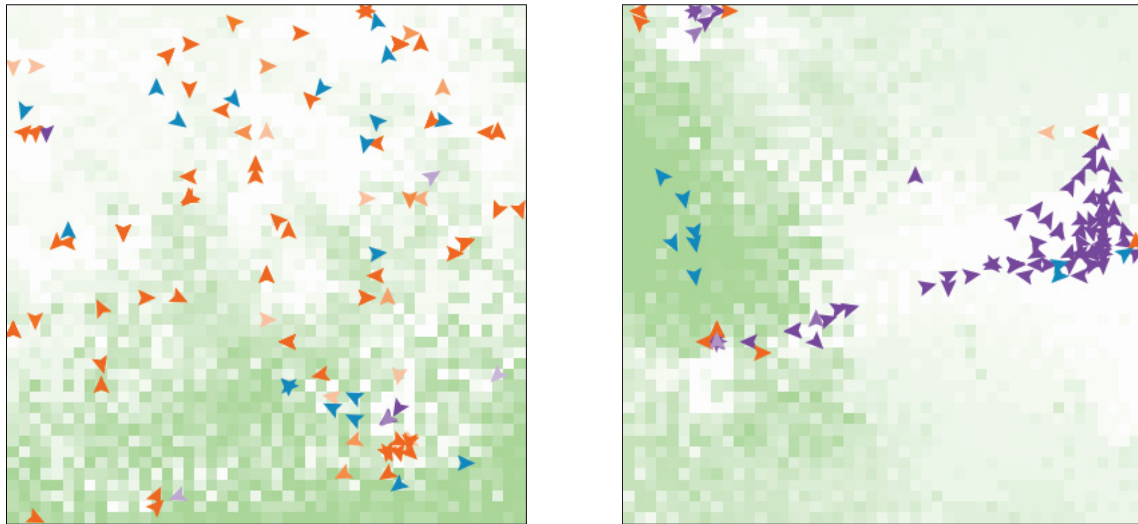

Figure 3.1: Simulation frames taken from an predator/prey A-life environment used in previous work (Soares et al., 2018). Each image presents a distinct classes of emergent population behaviour in the populations.

3.2 Anomaly Detection in Video Games

Player behaviour detection is of interest in on-line games where game modifications, the use of automated players (i.e., bots), and other same anomalous behaviours are considered cheating. For instance, machine learning was used (Ahmad et al., 2009) to detect gold farmers in *EverQuest II* (Daybreak Game Company, 2004). In Massive Multiplayer Online Games (MMOGs), gold farmers are players that dedicate most of their playing time acquiring in-game currency to exchange for real-world currency. The behaviour of gold farmer can then be considered anomalous to the game, since it is unusual and not expected by the developers. Ahmad's approach takes as input logs from the game (e.g., transactions logs, activity form experience logs) and other game-specific data such as character race (e.g., human, elf) and character sex. Their approach consists of using labelled data to create a deductive logit model and an inductive machine learning model. The anomalous labels were given only to the data corresponding to banned players (i.e., players removed from the game after being caught gold farming). Their approach, however, had difficulties detecting gold

farmers, which may have been caused by the specialization of gold farmers into specialized roles distinct from each other. Furthermore, they concluded that the precision rate of their detection might only represent a minimum baseline, since not all gold farmer were necessarily banned, thus leading to potentially mislabelled training data.

Our own work on detecting anomalous population behaviour in a commercial game (Soares et al., 2018) used supervised learning. Doing so required (i) defining anomalous behaviour a priori, (ii) being able to capture large amounts of such rare behaviours for inclusion into the training set, (iii) labelling each datum in the training set as normal/anomalous. These requirements do not fit our desiderata in Section 2.3.

Chapter 4

Proposed Approach

Given the lack of labelled anomalous training data, we propose to use autoencoders (AE) (Protopapadakis et al., 2017; Ribeiro et al., 2018). Specifically, we train a variational autoencoder (VAE) (Kingma and Welling, 2013) to detect anomalous behaviours. A diagram of our approach is in Figure 4.11 with algorithmic details listed below. Some material of this chapter was adapted from our previous publications (Bulitko et al., 2017; Soares et al., 2018; Soares and Bulitko, 2019).

4.1 Autoencoders (AE)

Notationally, $\boldsymbol{x} \in \mathbb{R}^k$ represents an input datum (e.g., an RGB image). AE is a neural network composed of an encoder and a decoder as illustrated in Figure 4.1. We adapt the definitions of AE from previous work (Goodfellow et al., 2016). The encoder is a function $\boldsymbol{h} = g(\boldsymbol{x})$ that converts the input $\boldsymbol{x}$ into a feature vector (i.e., code) with dimension $m \ll k$. The decoder, $\boldsymbol{x'} = u(g(\boldsymbol{x}))$, is responsible for reconstructing the original input given the extracted feature vector $\boldsymbol{h}$. Working together, the encoder and decoder create a reversible compression of $\boldsymbol{x}$ with a code that should provide useful structural information about the training data. They can be machine-learned with the loss function defined as the mean element-wise squared error:

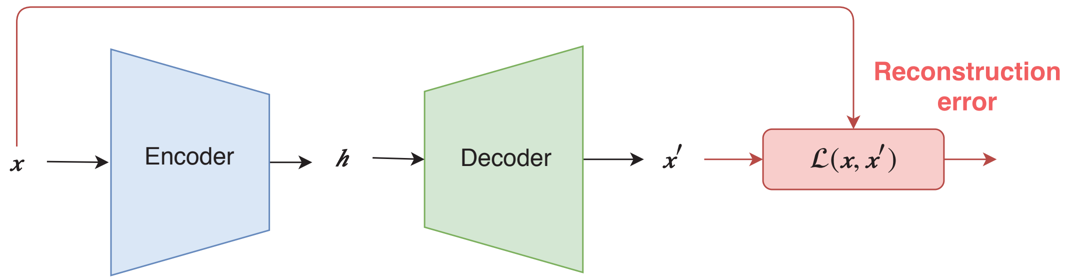

Figure 4.1: An autoencoder.

$$\mathcal{L}\left(\boldsymbol{x}, \boldsymbol{x'}\right) = \sum_{i=1}^{k} \frac{\left(x_k - x'_k\right)^2}{k} \tag{4.1}$$

where x_k and x'_k are the k-th element of the input $\boldsymbol{x}$ and reconstruction $\boldsymbol{x'}$ respectively. Given images as inputs, each x_k is a pixel and the loss can also be called *mean pixel-wise squared error*. Equation 4.1 requires no target labels when training the autoencoder, which satisfies one of the constraints in Chapter 2. Furthermore, the mean pixel-wise square error is a target loss function that suites image inputs, which are readily available features $f\left(S_{t-\tau}, \ldots, S_t\right)$, satisfying another constraint on the problem.

4.1.1 Stochastic Autoencoders

We can also define an autoencoder as a stochastic machine learning model. According to Goodfellow et al. (2016):

> "Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings $p_{\text{encoder}}(\boldsymbol{h} \mid \boldsymbol{x})$ and $p_{\text{decoder}}(\boldsymbol{x} \mid \boldsymbol{h})$."

Hence, we can understand AE as a statistical model that maximizes the likelihood of the correct reconstruction given an input and its code: $p_{\text{autoencoder}}(\boldsymbol{x'}|\boldsymbol{x})$. This model can be trained by reducing the reconstruction error between original and reconstructed input, $\mathcal{L}\left(\boldsymbol{x}, \boldsymbol{x'}\right)$.

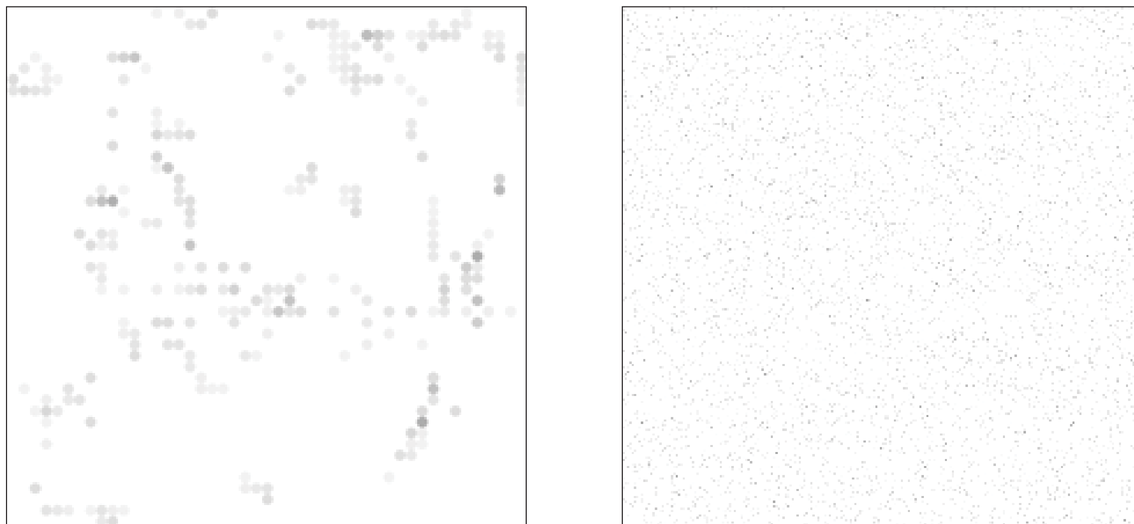

Figure 4.2: A normal image (left) used to generate a surrogate anomaly (right) via a random permutation of its pixels.

4.1.2 Anomaly Detection with AE

As the autoencoder learns to reconstruct training data by passing a high-dimensional image through a lower-dimensional code, it is likely to learn visual patterns in its input images. If such patterns in visualization of normal behaviours (which the training data mostly consists of) are different from patterns specific to visualization of anomalous behaviours (which are mostly absent in the training data), then one may assume that the reconstruction error $\mathcal{L}(\boldsymbol{x}, \boldsymbol{x'})$ will be higher for images of anomalous behaviour.

Thus, we can turn the AE into an anomaly detector by comparing its reconstruction error against a threshold θ as follows: the input image $\boldsymbol{x}$ is classified as normal if $\mathcal{L}(\boldsymbol{x}, \boldsymbol{x'}) \leq \theta$ and anomalous if $\mathcal{L}(\boldsymbol{x}, \boldsymbol{x'}) > \theta$.

The threshold θ has to be tuned to maximize the AE accuracy as a behaviour detector. The tuning process is complicated by the fact that labelled anomalous data is unavailable (Section 2.3). Thus we tune θ against automatically synthesized images of a *surrogate anomaly*. To create such a surrogate-anomaly set, we take a separate set of normal images and perturb each image by shuffling its pixels according to a random permutation (Figure 4.2).

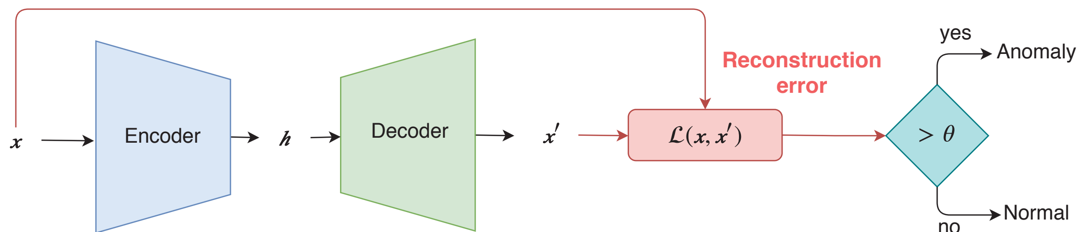

Figure 4.3: The flow of information in anomaly detection via a variational autoencoder.

We then use the two new image sets to select the threshold θ for an already trained AE according to the interquatile range (IQR). The candidate values of θ run the set $\theta_i = Q_3 + \frac{i}{2}\text{IQR}$, $i \in \{0, 1, \ldots, 20\}$. Here $\text{IQR} = Q_3 - Q_1$, Q_1 is the 25th percentile and Q_3 is the 75th percentile (Zhao et al., 2013) of the AE's reconstruction errors on its training set, computed after its training. We choose the θ_i which maximizes the AE's accuracy in classifying the surrogate anomalies. A scheme for our proposed approach is in Figure 4.3. As explained in detail in Section 4.3, our AE receives as input the feature $f\,(S_{t-\tau}, \ldots, S_t)$, thus our method uses $\mathcal{L}(f\,(S_{t-\tau}, \ldots, S_t)\,, \boldsymbol{x}')$, where $\boldsymbol{x}'$ is the reconstruction of $f\,(S_{t-\tau}, \ldots, S_t)$.

4.1.3 Learning a Useful Compression

If the code vector has the same or higher dimension as the input, the encoder may not compress the data at all, giving $\boldsymbol{h} = \boldsymbol{x}$. The dimension of a code layer affects autoencoder's *capacity* to represent the complexity of $\boldsymbol{x}$ (i.e., the amount of information in $\boldsymbol{x}$). Excessive capacity yields trivial and thus useless compression. To learn a non-trivial compression one can encourage sparsity of the AE (Goodfellow et al., 2016). By forcing the code $\boldsymbol{h}$ to be closer to a zero vector (i.e., penalizing large code values), the network is forced to learn more compact codes. This sparsity can be achieved by enforcing a penalization term on $\boldsymbol{h}$ such as the Laplace prior (Goodfellow et al., 2016). Usually, regularization is applied to the weights of an artificial neural network; however, to promote sparsity in the autoencoder we penalize the activation of the encoder's last layer (i.e., code $\boldsymbol{h}$) instead of its weights so the code itself becomes closer to a zero vector.

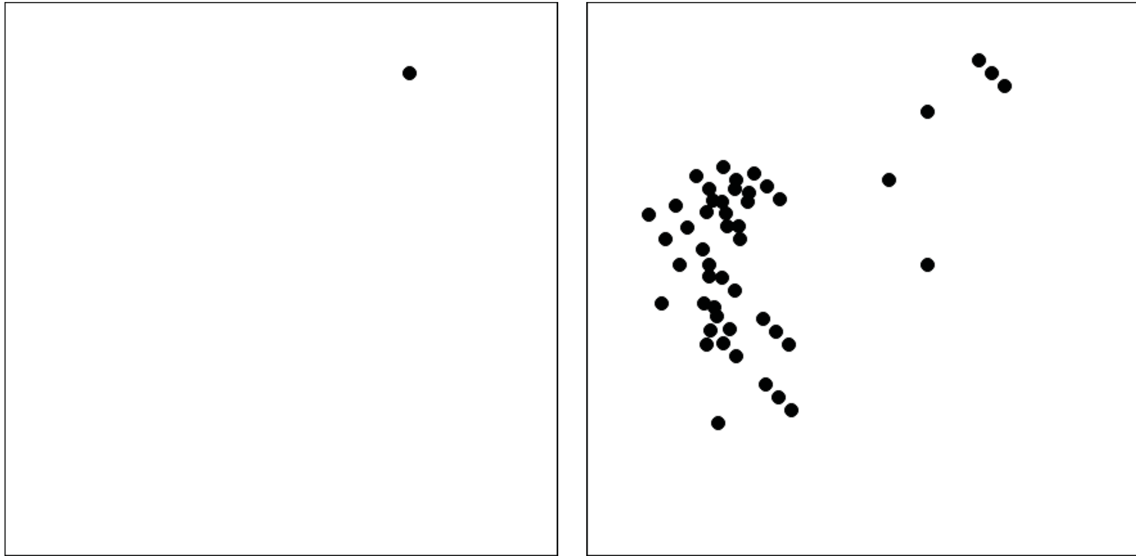

Figure 4.4: Images with different complexity. More information is necessary to reconstruct a cluster of points (right) than a single point (left).

Another way of helping the code to represent useful information is to use *denoising autoencoders* (Goodfellow et al., 2016; Vincent et al., 2008). These AEs are trained to reconstruct the original input by receiving a noisy version of it: $\boldsymbol{x}_{\text{noisy}} = \boldsymbol{x} + \epsilon$, where ϵ is a noise drawn from a Gaussian distribution. Since this AE is now trained to minimize $\mathcal{L}\left(\boldsymbol{x}, u(g(\boldsymbol{x}_{\text{noisy}}))\right)$, where $g(\boldsymbol{x}_{\text{noisy}})$ is the denoising encoder and $u(g(\boldsymbol{x}_{\text{noisy}}))$ is the denoising decoder, the code must provide enough structural information to recover the input from corruption.

4.1.4 A drawback of an Autoencoder

Although both methods in Section 4.1.2 can help an autoencoder learn useful structural patterns from the training data, it can still have a detection accuracy sensitive to the level of complexity present in the input. Since the reconstruction error only measures the difference between input and reconstruction, an AE with enough capacity to reconstruct complex inputs is also likely to reconstruct simpler images even if it was not trained on them specifically. For example, if an AE that can reconstruct clusters of points (Figure 4.4, right)

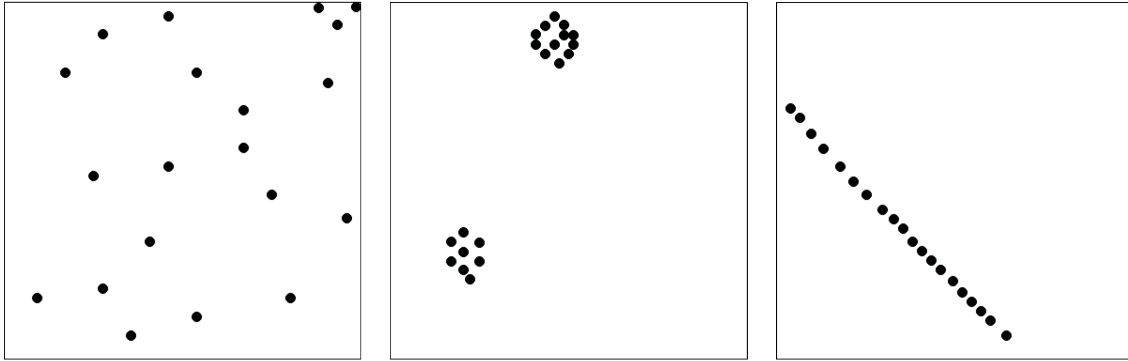

Figure 4.5: Twenty points distributed randomly (left), in clusters (center), and in a single line (right).

is also probably able to reconstruct a single point (Figure 4.4, left). Similarly, Figure 4.5 presents a distribution of 20 points randomly placed (Figure 4.5, left), organized in clusters (Figure 4.5, center), and in a line (Figure 4.5, right). Randomly placed points have a higher complexity compared to clusters or lines, thus requiring a higher code capacity. So an AE trained on randomly placed points as normal data will have a similarly low reconstruction error on clusters or lines, thereby labelling them as normal too.

An AE trained to reconstruct complex input and also able to reconstruct simpler images works well when anomalous images are more complex than normal images. However, if the opposite holds, the AE trained on normal images may also be able to reconstruct anomalous images making it a poor anomaly detector. For example, the AE trained on randomly placed points may learn to encode the coordinate of each point; thus having approximately the same reconstruction error for any pattern of points. In our A-life simulation studied in previous work (Soares et al., 2018; Soares and Bulitko, 2019), the anomalous behaviours tended to have agents grouped in clusters and swarm patterns (Figure 4.6, right). Then anomalous images were probably less complex than normal images (Figure 4.6, left), which would make such an AE a poor anomaly detector.

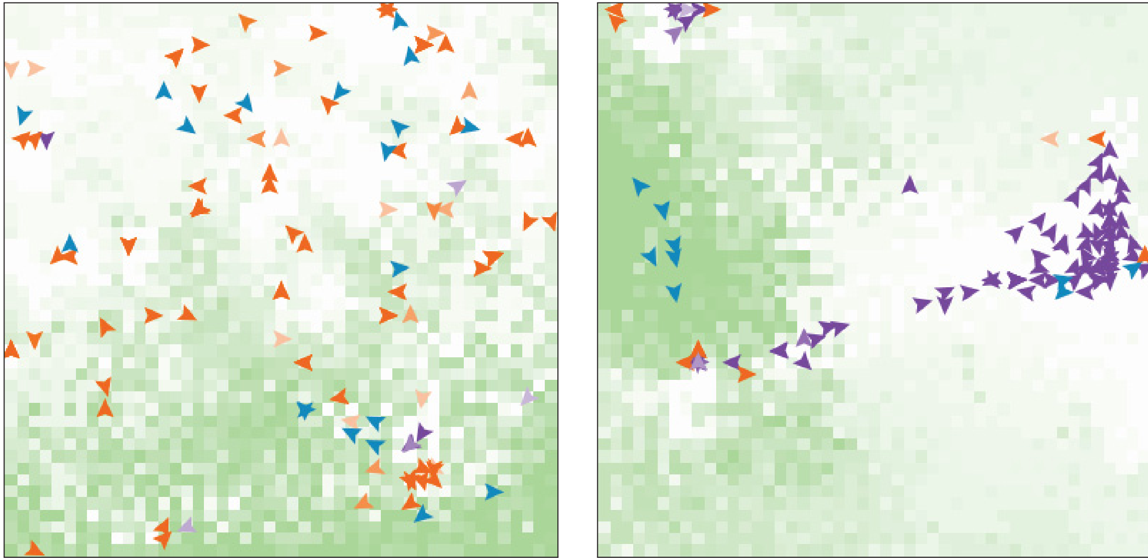

Figure 4.6: An image with normal behaviour (left) with agents (triangles) more scattered than an image with anomalous behaviour (right).

4.2 Variational Autoencoders

One solution to decoupling image complexity and image normality[1] is to use variational autoencoders (VAE) showed in Figure 4.7. The VAE maps an input to a distribution of codes instead of a single code. Then an actual code is sampled from the distribution and decoded into an output (i.e., reconstructed) image. Instead of compressing $\boldsymbol{x}$ into a code vector, a VAE compresses $\boldsymbol{x}$ into two vectors: $\boldsymbol{\mu}(\boldsymbol{x}) \in \mathbb{R}^m$ and $\boldsymbol{\sigma}(\boldsymbol{x}) \in \mathbb{R}^m$ with $m \ll k$. The code $\boldsymbol{z}(\boldsymbol{x}) \in \mathbb{R}^m$ is then drawn from a normal distribution with mean $\boldsymbol{\mu}(\boldsymbol{x})$ and variance $\epsilon \cdot \boldsymbol{\sigma}(\boldsymbol{x})$, where ϵ is a scalar random variable drawn from $\mathcal{N}(0, 1)$. The decoder then reconstructs $\boldsymbol{x}$ using the code $\boldsymbol{z}(\boldsymbol{x})$ as input.

By moving from AE's single code deterministically computed to VAE's code stochastically sampled from a distribution, we decouple input normality from input complexity. To better understand this claim, let us first assume that the input $\boldsymbol{x}$ given to the autoencoder corresponds to features extracted from a sequence of simulation states $(S_{t-\tau}, \ldots, S_t)$. A

[1]By normality we mean the input not being anomalous.

simulation used to generate emergent behaviour usually encompasses one or more stochastic processes (e.g., mutation, cross-over, random initialization) that influence the probability of a particular state sequence $(S_{t-\tau}, \ldots, S_t)$ and thus a particular $\boldsymbol{x}$. We also assume that such stochastic processes are parameterized by an unknown hidden latent variable $\boldsymbol{z}$ and by an unknown parameter vector ξ. Each value of $\boldsymbol{x}$ has a corresponding $\boldsymbol{z}$ value that influenced the stochastic process that generated $\boldsymbol{x}$.

VAEs are designed in such a way that, for each $\boldsymbol{x}$ in the training set, the encoder infers the corresponding value for the latent variable $\boldsymbol{z} = \boldsymbol{z}(\boldsymbol{x})$. We assume that $\boldsymbol{z}(\boldsymbol{x})$ has a normal probability distribution $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, where $\boldsymbol{0}$ is the zero vector mean and $\boldsymbol{I}$ is the identity matrix. The decoder reconstructs the original input back from the inferred value of $\boldsymbol{z}(\boldsymbol{x})$. A VAE parameterized by a parameter vector ϕ and trained only on normal images has an encoder $q_\phi(\boldsymbol{z} = \boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x})$ which gives an approximation to the true conditional probability $p_\xi(\boldsymbol{z}|\boldsymbol{x})$. If the conditional distributions $q_\phi(\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{z}|\boldsymbol{x}, A(\boldsymbol{x}) = \text{true})$ and $q_\phi(\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{z}|\boldsymbol{x}, A(\boldsymbol{x}) = \text{false})$ are distinct and if the VAE has learned only to infer the right values of $\boldsymbol{z}(\boldsymbol{x})$ for normal data (i.e., $A(\boldsymbol{x}) = \text{false}$) then the VAE's anomaly detection should indeed occur independently from the complexity differences between anomalous and normal data. Indeed, by training our VAE only on normal data, it can learn to compute the right values for the mean $\boldsymbol{\mu}(\boldsymbol{x})$ and variance $\boldsymbol{\sigma}(\boldsymbol{x})$ such that $q_\phi(\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{z}|\boldsymbol{x}, A(\boldsymbol{x}) = \text{false})$ is a good approximation for $p_\xi(\boldsymbol{z}|\boldsymbol{x}, A(\boldsymbol{x}) = \text{false})$ for normal data. However, since the VAE never sees anomalies during training, it may compute the wrong value of mean and variance for anomalous data, which leads to a distribution $q_\phi(\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{z}|\boldsymbol{x}, A(\boldsymbol{x}) = \text{true})$ that might completely differ from the true distribution $p_\xi(\boldsymbol{z}|\boldsymbol{x}, A(\boldsymbol{x}) = \text{true})$.

4.2.1 VAE as a Stochastic Model

We use the variatonal calculus adapted from the previous work (Kingma and Welling, 2013) to describe how a proper VAE can be designed. First, given a dataset $\{\boldsymbol{x}^{(i)} \in \mathbb{R}^k\}, i = 1, \ldots, N$, we assume that each $\boldsymbol{x}$ was generated through a stochastic process (e.g., genetic evolution) parameterized by a correspondent unknown latent variable $\boldsymbol{z}^{(i)}$. Consider that each random

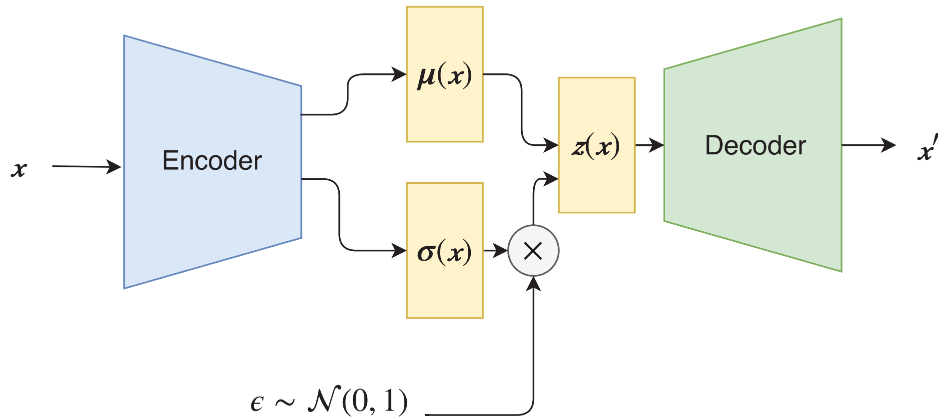

Figure 4.7: A variational autoencoder.

variable $\boldsymbol{z}$ has a prior distribution $p_\xi(\boldsymbol{z})$. Since $\boldsymbol{x}$ is generated by this stochastic process, $\boldsymbol{x}$ is also a random variable with distribution $p_\xi(\boldsymbol{x}) = p_\xi(\boldsymbol{x}|\boldsymbol{z}) \cdot p_\xi(\boldsymbol{z})$. The true prior $p_\xi(\boldsymbol{z})$ and likelihood $p_\xi(\boldsymbol{x}|\boldsymbol{z})$ are a family of probabilities parameterized by ξ, which is also unknown. We assume that it is intractable to compute the posterior density $p_\xi(\boldsymbol{z}|\boldsymbol{x}) = \frac{p_\xi(\boldsymbol{x}|\boldsymbol{z}) \cdot p_\xi(\boldsymbol{z})}{p_\xi(\boldsymbol{x})}$ (Kingma and Welling, 2013), but it can be approximated by the probabilistic encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ parameterized by ϕ. Hence, a VAE has its code $\boldsymbol{z}(\boldsymbol{x})$ computed through the probabilistic encoder $q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x})$, which is used by the probabilistic decoder $q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))$ to reconstruct $\boldsymbol{x}$. Thus, a VAE parameterized by ϕ is designed in such a way that the encoder $q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x})$ and the decoder $q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))$ are approximations for $p_\xi(\boldsymbol{z}|\boldsymbol{x})$ and $p_\xi(\boldsymbol{x}|\boldsymbol{z})$, respectively. The VAE must then be trained in order to find the set of parameters ϕ that maximizes the likelihood $q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))$, which is an approximation for the true likelihood $p_\xi(\boldsymbol{x}|\boldsymbol{z})$.

4.2.2 Training a VAE

We want to make sure that the encoder $q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x})$ is learning the correct approximation to the conditional probability $p_\xi(\boldsymbol{z}|\boldsymbol{x})$. Let us first consider the Kullback-Leibler Divergence (KL-divergence) (Kullback and Leibler, 1951), D_{KL}. This function measures divergence

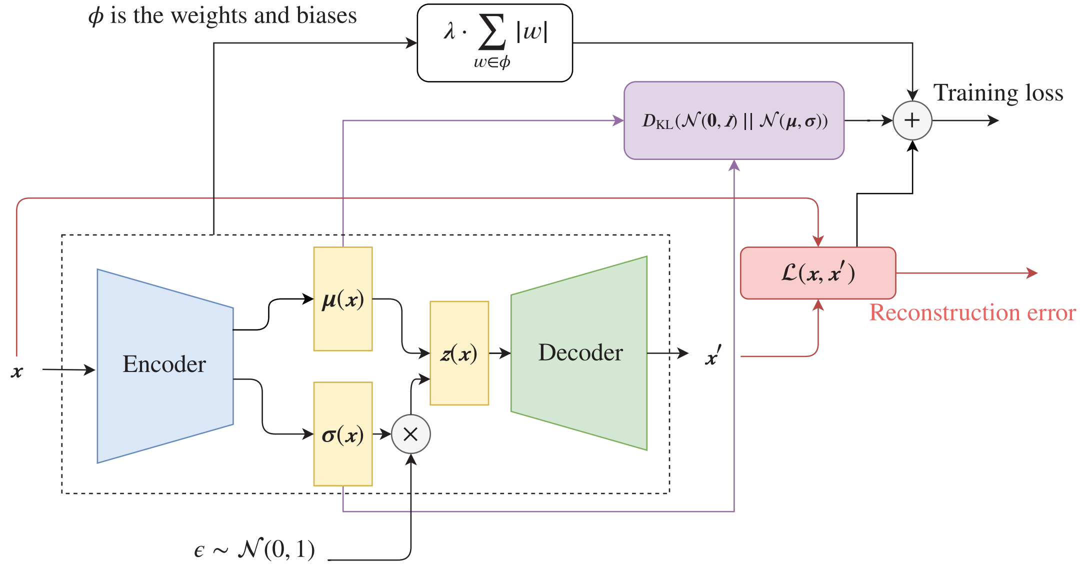

Figure 4.8: Training variational autoencoder.

between two probability distributions $P(x)$ and $Q(x)$:

$$D_{\mathrm{KL}}\left(P \,||\, Q\right) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx = \mathbb{E}_{x \sim P}\left[\log P(x) - \log Q(x)\right] \tag{4.2}$$

We want to find ϕ that minimizes $D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \,||\, p_\xi(\boldsymbol{z}|\boldsymbol{x})\right)$:

$$D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \,||\, p_\xi(\boldsymbol{z}|\boldsymbol{x})\right) = \mathbb{E}_{\boldsymbol{z} \sim q_\phi}\left[\log q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) - \log p_\xi(\boldsymbol{z}|\boldsymbol{x})\right] \tag{4.3}$$

By applying Bayes rules to $p_\xi(\boldsymbol{z}|\boldsymbol{x})$:

$$D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \,||\, p_\xi(\boldsymbol{z}|\boldsymbol{x})\right) = \\ \mathbb{E}_{\boldsymbol{z} \sim q_\phi}\left[\log q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) - \log p_\xi(\boldsymbol{x}|\boldsymbol{z}) - \log p_\xi(\boldsymbol{z})\right] + \log p_\xi(\boldsymbol{x}) \tag{4.4}$$

Minimizing $D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z}|\boldsymbol{x})\right)$ is equivalent to maximizing $-D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z}|\boldsymbol{x})\right)$. Hence, we want select ϕ to maximize:

$$
\begin{aligned}
&- D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z}|\boldsymbol{x})\right) = \\
&\qquad - \mathbb{E}_{\boldsymbol{z}\sim q_\phi}\left[\log q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) - \log p_\xi(\boldsymbol{x}|\boldsymbol{z}) - \log p_\xi(\boldsymbol{z})\right] - \log p_\xi(\boldsymbol{x})
\end{aligned} \tag{4.5}
$$

By rearranging the expectation and adding $\log p_\xi(\boldsymbol{x})$ on both sides of the equation we have:

$$
\begin{aligned}
\log p_\xi(\boldsymbol{x}) - D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z}|\boldsymbol{x})\right) &= \mathbb{E}_{\boldsymbol{z}\sim q_\phi}\left[\log p_\xi(\boldsymbol{x}|\boldsymbol{z})\right] \\
&\quad - \mathbb{E}_{\boldsymbol{z}\sim q_\phi}\left[\log q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) - \log p_\xi(\boldsymbol{z})\right]
\end{aligned} \tag{4.6}
$$

Following from the definition of KL-divergence:

$$
\begin{aligned}
\log p_\xi(\boldsymbol{x}) - D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z}|\boldsymbol{x})\right) &= \mathbb{E}_{\boldsymbol{z}\sim q_\phi}\left[\log p_\xi(\boldsymbol{x}|\boldsymbol{z})\right] \\
&\quad - D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z})\right).
\end{aligned} \tag{4.7}
$$

The distribution $p_\xi(\boldsymbol{x}|\boldsymbol{z})$ is approximated by the decoder $q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))$ which gives us:

$$
\begin{aligned}
\log p_\xi(\boldsymbol{x}) - D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z}|\boldsymbol{x})\right) &= \mathbb{E}_{\boldsymbol{z}\sim q_\phi}\left[\log q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))\right] \\
&\quad - D_{\mathrm{KL}}\left(q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x}) \mid\mid p_\xi(\boldsymbol{z})\right)
\end{aligned} \tag{4.8}
$$

where the term $q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))$ is computed by the decoder and $q_\phi(\boldsymbol{z}(\boldsymbol{x})|\boldsymbol{x})$ is computed by the encoder as $\mathcal{N}(\boldsymbol{\mu}(\boldsymbol{x}), \boldsymbol{\sigma}(\boldsymbol{x}))$. The expectation $\mathbb{E}_{\boldsymbol{z}\sim q_\phi}\left[\log q_\phi(\boldsymbol{x}|\boldsymbol{z}(\boldsymbol{x}))\right]$ can be maximized through the minimization of the reconstruction loss $\mathcal{L}(\boldsymbol{x'}, \boldsymbol{x})$. Since we assumed that the prior $p_\xi(\boldsymbol{z})$ is the normal distribution $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, we can maximize Equation 4.8 by training our VAE to minimize the loss function:

$$
\mathcal{L}(\boldsymbol{x'}, \boldsymbol{x}) + D_{\mathrm{KL}}\left(\mathcal{N}(\mu(\boldsymbol{x}), \sigma(\boldsymbol{x})) \mid\mid \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})\right) \tag{4.9}
$$

where:

$$D_{\mathrm{KL}}\left(\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\sigma}) \mid\mid \mathcal{N}(\mathbf{0},\boldsymbol{I})\right) = \frac{1}{2}\sum_{i=1}^{m}\left(\sigma_i + \mu_i^2 - \log\left(\sigma_i\right) - 1\right). \tag{4.10}$$

By using KL-divergence on $\boldsymbol{\mu}(\boldsymbol{x})$ and $\boldsymbol{\sigma}(\boldsymbol{x})$, which are used to draw $\boldsymbol{z}(\boldsymbol{x})$, the VAE's code becomes sparse (Asperti, 2018). The sparsity of the stochastic code helps solving the problem discussed in Section 4.1.2. Additionally, we also add the L1 regularization term given by $\lambda \cdot \sum_{w\in\phi}|w|$ to obtain weight sparsity, which helps prevent overfitting (Goodfellow et al., 2016). The full training loss is then given by the expression:

$$\mathcal{L}(\boldsymbol{x},\boldsymbol{x'}) + D_{\mathrm{KL}}\left(\mathcal{N}(\mu(\boldsymbol{x}),\sigma(\boldsymbol{x}) \mid\mid \mathcal{N}(\mathbf{0},\boldsymbol{I})\right) + \lambda \cdot \sum_{w\in\phi}|w| \tag{4.11}$$

where ϕ is the set of parameters in the network (weights and bias) and $\lambda \in \mathbb{R}$ is a scaler for the regularization term.

4.2.3 Anomaly Detection with VAE

Similar to Section 4.1.4, we want a VAE (Figure 4.7) with a larger reconstruction error for anomalies. That would allow its use as an anomaly detector by simply measuring the reconstruction error $\mathcal{L}(\boldsymbol{x},\boldsymbol{x'})$. Here we assume that the stochastic process that generates a normal behaviour is different from the one that led to emergent anomalous behaviours (e.g., different play styles in a video game with evolving NPCs). Figure 4.9 summarizes the detector D_ϕ created with a VAE. The topology of our VAE is explained in Section 5.1.3 and details about the implementation are in Appendix B.

4.3 Frame Averaging

Our approach does not require access to internal simulation states $(S_{t-\tau},\ldots,S_t) \in \mathbb{S}$. As stated in Section 2.3, the input for our approach is given instead by the feature $f\left(S_{t-\tau},\ldots,S_t\right)$ extracted from the simulation states. In our approach, every $I(S_t) \in \mathbb{R}^k$ is an image (i.e., pixel matrix) obtained through screen capture (e.g., screen-shots from simulations/video

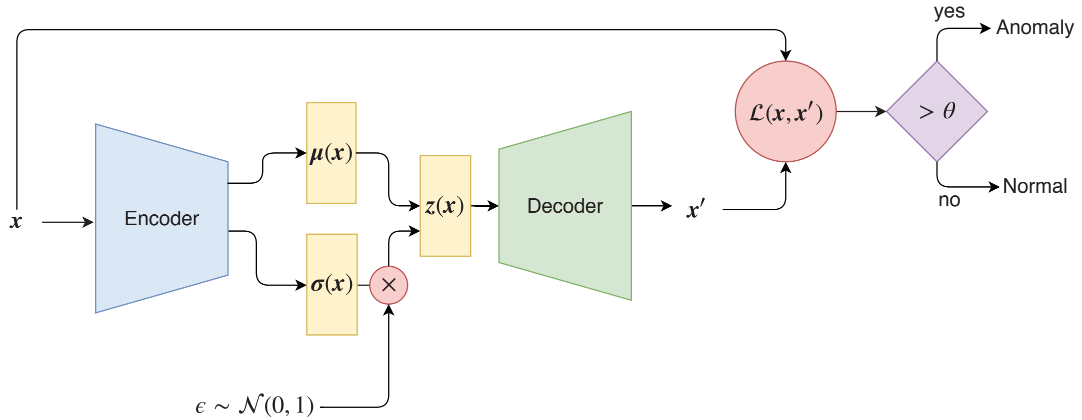

Figure 4.9: The flow of information in anomaly detection via a variational autoencoder.

games, video frames from surveillance cameras). An image $I(S_t)$ collected in a time step t is the visualization of the environment state S_t. The input features $f(S_{t-\tau}, \ldots, S_t)$ used in our approach is a weighted average of images:

$$f(S_{t-\tau}, ..., S_t) = \sum_{i=0}^{\tau} \gamma^i \cdot I(S_{t-i}) \tag{4.12}$$

where $0 < \gamma < 1$ is a decay coefficient. Figure 4.10 shows a sample gray scale image representing $I(S_t)$ (left) and the corresponding $f(S_{t-\tau}, \ldots, S_t)$ (right) for $\tau = 4$. Upon performing this step, the anomaly detection must classify $f(S_{t-\tau}, \ldots, S_t)$ as normal or anomalous.

4.4 Summary

For each simulation state S_t we collect its visualization I_t. We then add time information to our input by computing the frame average $\boldsymbol{x} = f(S_{t-\tau}, \ldots, S_t)$. A VAE is then trained only on normal image frames $\boldsymbol{x}$ to minimize the training loss (4.11). After training, an additional set composed of normal and surrogate anomaly (i.e., pixel-shuffled) images are used to choose a threshold θ that maximizes the detection accuracy. The trained VAE, in

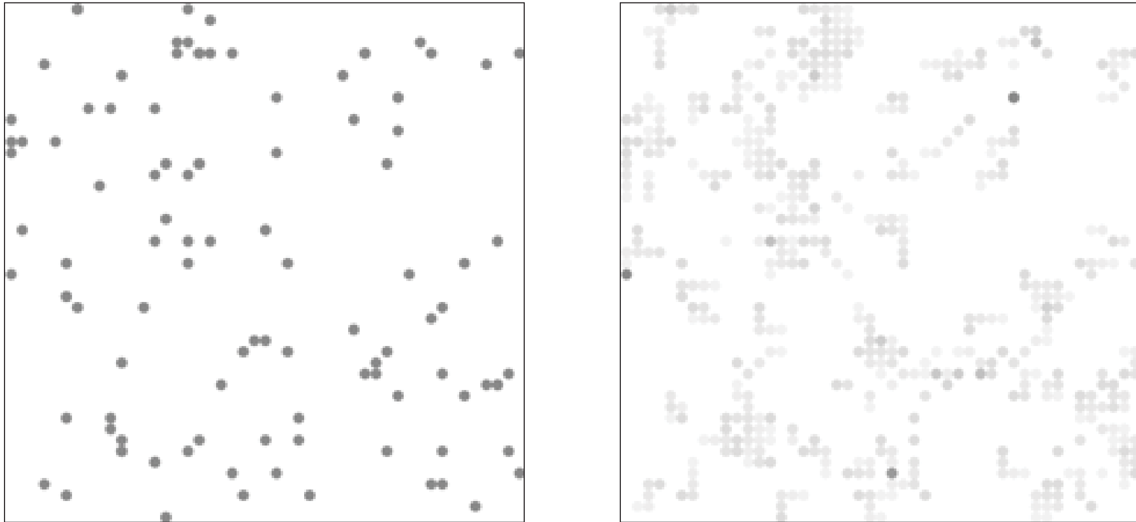

Figure 4.10: A image representing the environment state $I(S_t)$ on time step t (left) and its frame averaging $f\,(S_{t-\tau}, \ldots, S_t)$ considering the past $\tau = 4$ frames (right). Gray dots represent agents within the environment.

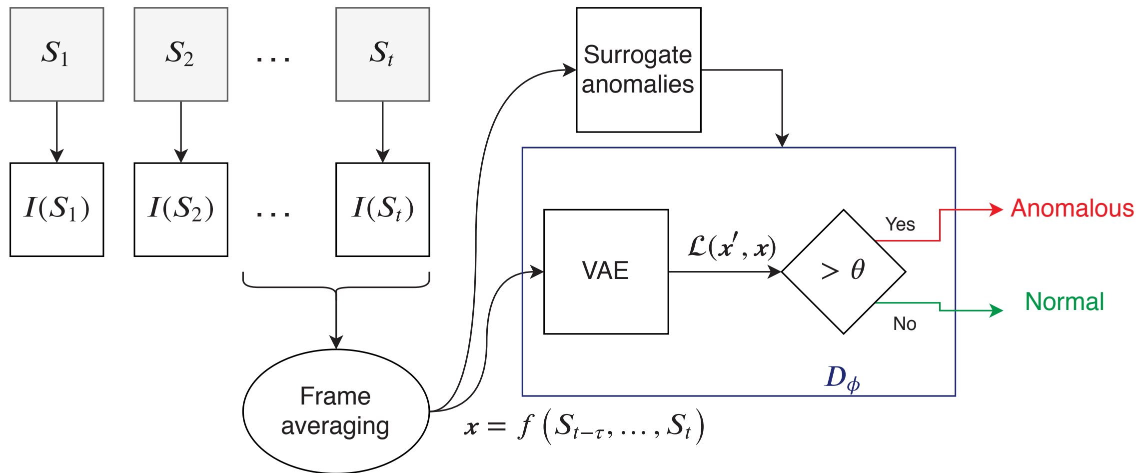

Figure 4.11: Our proposed approach takes unlabelled image visualization from simulated multi-agent environments and used than to train a VAE. We also tune a threshold by selecting the model with best accuracy when detecting surrogate anomalies. We perform frame averaging in the images to add temporal information.

conjunction with the chosen θ, forms the anomaly detector used in the empirical evaluation in Chapter 5 and 6.

4.5 The hypothesis

The hypothesis we are investigating empirically is whether using a VAE trained on unlabelled data is a viable approach to detecting novel/anomalous behaviours not seen (or not labelled as such) during training.

Chapter 5

Empirical Evaluation and Results

Performance of our approach was evaluated by computing the detection accuracy in a predator/prey A-life environment which allows emergence of novel behaviours through genetic evolution (Section 5.1). We compared our VAE against supervised learning: GoogLeNet (Szegedy et al., 2015). Two experiments were performed: one in which the training data contained only normal images (Section 5.1.5), and another in which the training data had a mixture of normal and anomalous images (Section 5.1.6). Some material in this chapter was adapted from previously published work (Bulitko et al., 2017; Soares et al., 2018; Soares and Bulitko, 2019).

Algorithm 1: `Predator/prey A-life environment`

input : Grid world $C^{k \times k}$; time steps T; initial population sizes $n_{\text{init}}^{\text{R}}, n_{\text{init}}^{\text{W}}$; initial grass per cell $n_{\text{init}}^{\text{g}}$; grass growth $g_{\text{grow}}^{\text{init}}$; max ages $a_{\max}$; reproduction age a_{repr}; reproduction energy e_{repr}; min energy $e_{\min}$; max grass $g_{\max}$; growth multiplier $g_{\text{grow}}^{\text{mult}}$; mutation rate δ_{rate}

output: Set I of image frames

1. $G^{k \times k} \leftarrow$ initial grass matrix with $n_{\text{init}}^{\text{g}}$
2. $R \leftarrow$ initial population with $n_{\text{init}}^{\text{R}}$ rabbits
3. $W \leftarrow$ initial population with $n_{\text{init}}^{\text{W}}$ wolves
4. $I \leftarrow \emptyset$; $t \leftarrow 0$; $g_{\text{grow}} \leftarrow g_{\text{grow}}^{\text{init}}$
5. **while** $t \leq T \wedge R \neq \emptyset \wedge W \neq \emptyset$ **do**
6.   **for** $\forall A \in R \cup W$ **do**
7. $N \leftarrow$ set of cells within A sight radius
8. $U(A, n) \leftarrow$ ultility of cell n in respect to agent $A, \forall n \in N$
9.   move A towards argmax$(U(a, n))$
10.   **if** $A \in R$ **then**
11. A eats grass in cell of A;
12.   **else**
13. A eats random rabbit in cell of A;
14.   **end**
15. $a \leftarrow A$'s age $+1$
16. $e \leftarrow A$'s energy
17.   age of $A \leftarrow a$
18.   **if** $a > a_{max} \vee e < e_{min}$ **then**
19.   **if** $A \in R$ **then**
20. $R \leftarrow R \setminus \{A\}$
21.   **else**
22. $W \leftarrow W \setminus \{A\}$
23.   **end**
24.   **end**
25.   **if** $a > a_{repr} \wedge e > e_{repr}$ **then**
26. $O \leftarrow$ mutated A according to δ_{rate}
27.   energy of $O \leftarrow e/2$
28.   energy of $A \leftarrow e/2$
29.   **if** $A \in R$ **then**
30. $R \leftarrow R \cup \{O\}$
31.   **else**
32. $W \leftarrow W \cup \{O\}$
33.   **end**
34.   **end**
35.   **end**
36. $g_{\text{grow}} \leftarrow g_{\text{grow}} * g_{\text{grow}}^{\text{mult}}$
37. $G[x, y] \leftarrow \max\left(G[x, y] + g_{\text{grow}}, g_{\max}\right), \forall (x, y) \in G^{k \times k}$
38. $I \leftarrow I \cup \{\text{screenshot at time } t\}$
39. $t \leftarrow t + 1$
40. **end**
41. **return** I

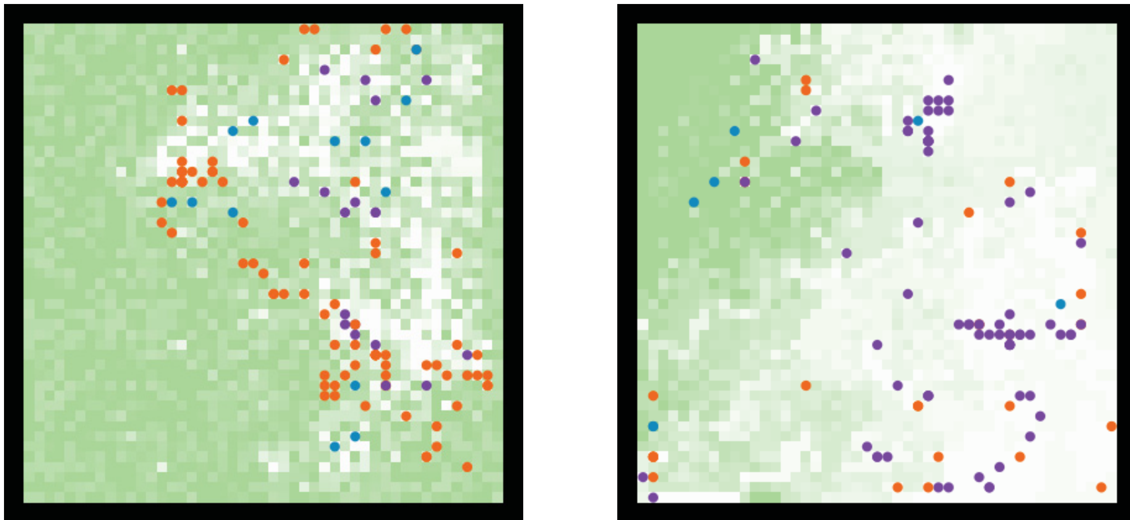

Figure 5.1: Our predator-prey A-life environment. The grass is shown via shades of green. Orange, purple and blue circles represent grass-liking rabbits, grass-disliking rabbit and wolves respectively. In the left image the majority of the rabbit population are grass-liking rabbits (i.e., the normal behaviour). In the right image the majority is comprised of the grass-disliking rabbits (i.e., an anomalous behaviour).

5.1 A-life Simulation

The A-life environment is a two-dimensional grid that supports a simple predator-prey evolution. The prey agents ("rabbits") eat a re-growing resource ("grass"). The predator agents ("wolves") eat rabbits. An agent's behaviour policy is determined by a set of weights encoded in its genes. On each time step, an agent computes the utility of each grid cell within its sight radius. The utility is a sum of cell contents (e.g., the amount of grass or the number of wolves in the cell) weighted by the agent's genetic weights (e.g., its affinity to grass). Agents that have a positive affinity to another component of the environment (i.e., rabbits, wolves, and grass) are said to like it; those with negative affinity, dislike it. The agent then moves towards the cell with the highest utility. Agents spend energy by taking actions or by simply existing in the simulation and replenish it by eating. They die when their energy drops below a minimum. A sufficiently old agent with enough energy will produce an offspring that inherits the parent's genes perturbed with random noise

(mutation). A high-level pseudocode is listed as Algorithm 41. All control parameters are found in Appendix A.

On a typical evolution run rabbits evolve to like grass and dislike wolves while wolves evolve to like rabbits. Anything substantially different is considered an anomaly. For instance, the leader-follower anomaly is an emergence of a large number of long-living yet grass-disliking rabbits (Soares et al., 2018). It can happen when two types of rabbits evolve: those who dislike grass but like other rabbits and those who like grass. Since a rabbit automatically eats grass in its current grid cell, grass-disliking rabbits can survive by liking and thus following grass-liking rabbits to grass-rich areas of the environment. While a small number of such follower rabbits are found on many evolutions runs (Figure 5.1, left), an emergence of follower rabbits in a large proportion happens rarely (i.e., less than 2% of the images have at least twice as many followers than leaders) and thus constitutes an anomaly (Figure 5.1, right).

5.1.1 Input Data

To satisfy the requirements in Section 2, our behaviour detector takes a stream of visualization frames produced by the A-life environment implemented in Netlogo (Wilensky, 1999). Although our A-life environment allows humans to visualize the rabbits' genes (purple and orange, Figure 5.1), we removed such genetic information and visualized all agents as grey dots only (red, green and blue colours all set to $\frac{141}{255}$). To make the input sparser, we also removed the grass visualization (Figure 5.2, left). Then to capture temporal patterns, we applied a moving average with an exponential decay to the frames (Figure 5.2, right) as explained in Section 4.3. A similar automated process can be applied to a screen-capture stream from a commercial video game.

5.1.2 Behaviour Classes

We use the A-life environment described in Section 5.1 with the image-generation process described in Section 5.1.1. The empirical evaluation will focus on detection of emergent

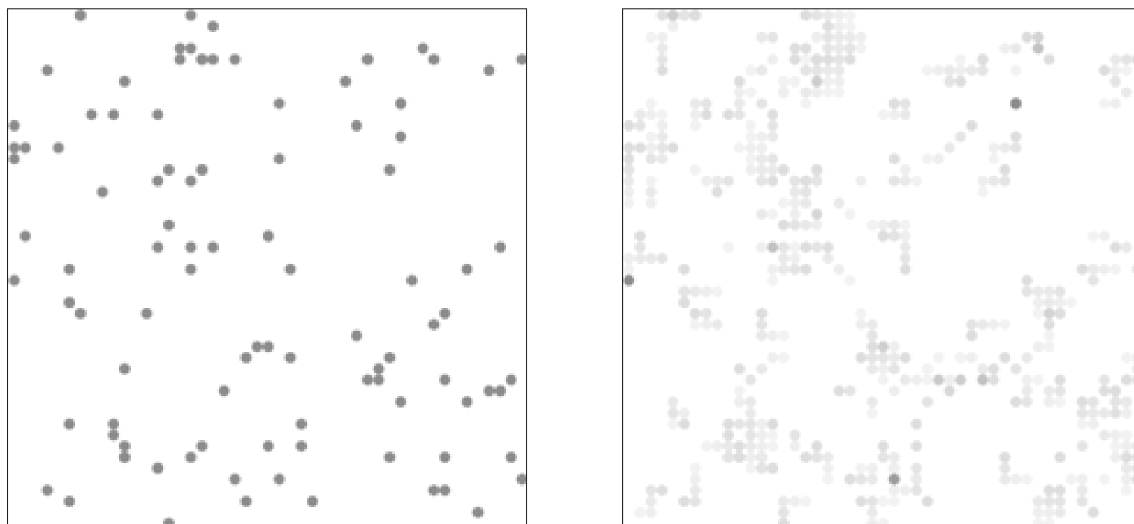

Figure 5.2: A single visualization frame from our predator-prey A-life environment with the resource and agent gene visualizations removed (left). A weighted moving average of the frame and its four predecessors captures movements of the agents (right).

anomalous behaviours in the rabbit population. We define three population-level behaviours in our A-life as follows. *Leaders* (B_{L}) behaviour occurs in a state of the simulation with at least twice as many leader rabbits as there are follower rabbits, which is the most common state and is considered normal behaviour. *Followers* (B_{F}) is a state of the simulation in which there are at least twice as many followers as there are leaders. This phenomenon happens much less frequently and thus constitutes an anomaly. Finally, *random* (B_{R}) is another type of anomaly where at least half of the rabbits in the environment have random behaviour policies.[1] Figure 5.3 shows representative images from the three behaviours.

Using A-life parameters from our prior work (Soares et al., 2018), detailed in Appendix A, we conducted 600 evolutions runs. On each run labelled each collected image into B_{L}, B_{F}, and B_{R} behaviours and stored only such images, discarding all other images.[2] Since we had access to the agent's genes in this case, we labelled images in which at least twice as many

[1] For our experiments, we needed two different types of anomalous behaviours, yet only one anomaly had been observed to date in the A-life environment. So we created the second anomaly artificially by injecting hand-built random-moving rabbits into the A-life environment.

[2] The greyscale time-averaged images were natively recorded at 420×420 pixels and then down-sampled to 224×224 pixels to fit the input dimension of our baseline classifier (GoogLeNet).

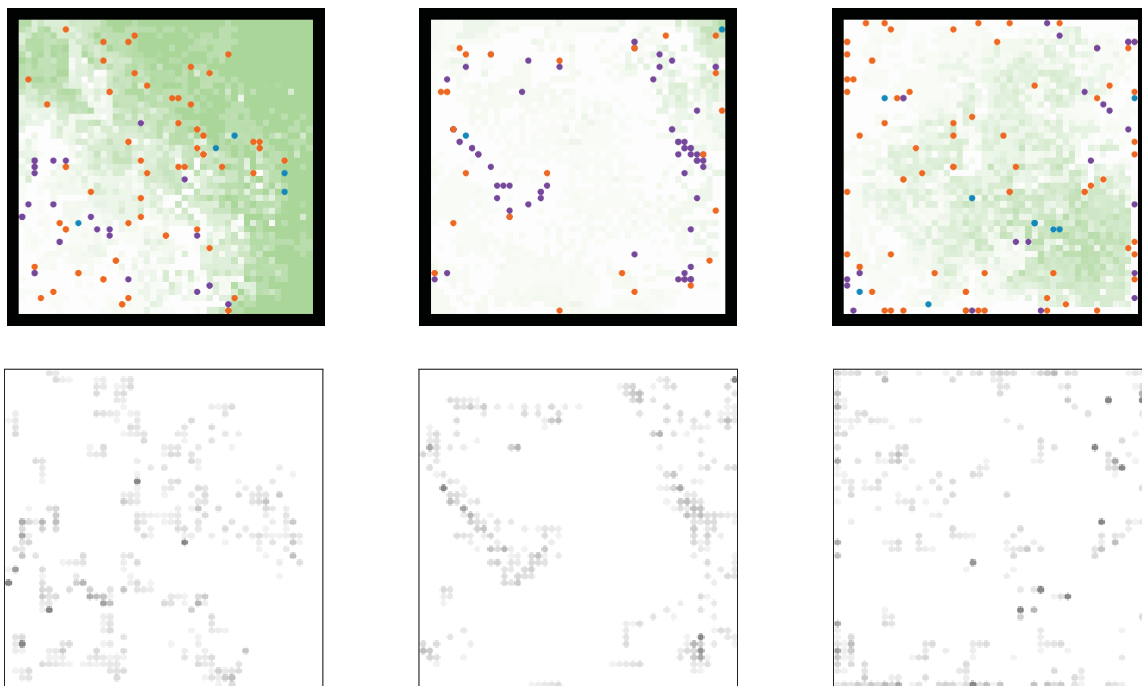

Figure 5.3: Top images are A-life visualization frames. They are converted to input images for the autoencoder and shown in the bottom. The three columns show representatives of classes B_{L}, B_{F}, and B_{R} correspondingly.

rabbits had positive affinity torwards grass as B_{L}. Images with twice as many rabbits with negative affinity towards grass were labelled as B_{F}. Images with at least 50% of the rabbit population containing random-moving rabbits – determined by a specific gene for random movement – were labelled as B_{R}. This process led to 1078946 images for the B_{L} class, 7941 for the B_{F} class, and 20095 images correspondent to neither B_{L} nor B_{F}. Table 5.1 shows the frequency of behaviours in the collected dataset, averaged over 600 runs.[3] After that, we balanced the dataset such that each class had the same number of images; thus, we obtained the sets $B_{\mathrm{L,pure}}$ and $B_{\mathrm{F,pure}}$, each one with 7941 images. These datasets are considered pure since each class has only samples that were correctly labelled (i.e., the $B_{\mathrm{L,pure}}$ has only B_{L} images, and $B_{\mathrm{F,pure}}$ has only B_{F} images).[4] In the beginning of the simulation, agents have

[3] The mean frequencies do not add up to 100% due to the variance of class distribution between runs. However, frequencies add up to 100% on each independent simulation run.

[4] We balanced the dataset for a better evaluation of the model. By performing this step, the test set will have approximately an equal share of images for each class, so that the classification accuracy is not inflated due to unbalanced data.

Table 5.1: Mean frequencies of behaviours in our A-life environment.

B_ψ	**Frequency (%)**
B_F	1.4 ± 9.7
B_L	56.3 ± 48.5
$\neg B_\mathrm{F} \wedge \neg B_\mathrm{L}$	2.3 ± 10.5

Table 5.2: Number of runs used on each data split.

Split	**Dataset**	**Number of runs**
complete dataset	$(B_\mathrm{L,pure}, B_\mathrm{F,pure})$	(600, 600)
complete dataset	$B_\mathrm{R,pure}$	2000
train	$\left(B^\mathrm{train}_\mathrm{L,pure}, B^\mathrm{train}_\mathrm{F,pure}\right)$	(254, 254)
train	$B^\mathrm{train}_\mathrm{L,pure}$	254
validation	$\left(B^\mathrm{val}_\mathrm{L,pure}, B^\mathrm{val}_\mathrm{F,pr}\right)$	(84, 84)
validation	$B^\mathrm{val}_\mathrm{L,pure}$	84
θ selection	$\left(B^\theta_\mathrm{L,pure}, B^\theta_\mathrm{S,pr}\right)$	(112, 112)
test	$\left(B^\mathrm{test}_\mathrm{L,pure}, B^\mathrm{test}_\mathrm{F,pure}\right)$	(150, 150)
test	$\left(B^\mathrm{test}_\mathrm{L,pure}, B^\mathrm{test}_\mathrm{L,pure}\right)$	(150, 500)

random affinities and thus move randomly. This situation can create a disconnect between image labels (B_L, B_F, B_R) which are based on rabbits' genes and the image content which depicts rabbits' actual behaviour.

In order to create independent data sets for training, testing, and validation, the original dataset was split at the level of evolution runs. The superscript *train*, *val*, θ, and *test* designate the corresponding use of the data split for training, validation, threshold selection, and testing, respectively. For example, $B^\theta_\mathrm{S,pure}$ correspond to the pure dataset composed of surrogate anomalies used for threshold selection. The split was performed in such a way that approximately 25% of the runs were used for testing ($B^\mathrm{test}_\mathrm{L,pure}$ and $B^\mathrm{test}_\mathrm{F,pure}$), 14% were used for validation ($B^\mathrm{val}_\mathrm{L,pure}$ and $B^\mathrm{val}_\mathrm{F,pure}$), and 43% of the runs were used for training ($B^\mathrm{train}_\mathrm{L,pure}$ and $B^\mathrm{train}_\mathrm{F,pure}$). The others 18% of the runs containing B_L behaviours were used to create surrogate anomalies $B_\mathrm{S,pure}$. This split ($B^\theta_\mathrm{L,pure}$, $B^\theta_\mathrm{S,pure}$) was used to compute the threshold θ for our VAE. Table 5.2 shows the number of runs per split.

Finally, we ran 2000 additional evolutions where, with probability $\eta = 50\%$, we replaced each rabbit in the starting population as well as each rabbit born during the evolution with a randomly moving rabbit[5]. This modification is in Algorithm 45. These 1020 runs yielded 21505 B_{R} images from which 7491 where selected to form the set $B_{\mathrm{R,pure}}$. These images were used only for testing according to the split in Table 5.2.[6]

5.1.3 VAE Implementation

Our VAE implementation used convolutional, fully connected, batch-normalization, max-pooling and transposed convolutional layers (Figure 5.4). It was coded and trained in TensorFlow (Abadi et al., 2015). Intuitively, we mainly used convolutional and transposed convolution layers to capture behaviour space-invariant patterns present in the image (e.g., clusters), to help with classification of population-level behaviour as discussed in previous work (Soares et al., 2018). Finally, the max-pooling layer was added to help with the dimensionality reduction: the code should be small enough to represent only normal patterns while conveying information about the underlying data distribution. All layers have rectified linear units (ReLu) activation which adds non-linearity to the topology. While we hand-tunned the architecture to the data at hand, we suspect that a similar architecture would work for other environments. Furthermore, the dimensions of $\boldsymbol{\mu}(\boldsymbol{x})$, $\boldsymbol{\sigma}(\boldsymbol{x})$, and $\boldsymbol{z}(\boldsymbol{x})$ were chosen as 500 through empirical process. We tested a different topology in a dataset collected from the same A-life simulation, but performing independent evolution runs from the one used in the experiments reported in this thesis. We selected the topology with higher detection accuracy in that independent dataset. Future work will investigate generating a VAE architecture automatically, such as via neuroevolution (Stanley and Miikkulainen, 2002). Details of the implementation are in Appendix B.

[5]Note that η is the probability of hatching a rabbit with hand-scripted random movement. For any $\eta \in [0, 100]$, B_{R} is defined as having at least half of the rabbit population in time t with such rabbits.

[6]We had to perform 1020 runs instead of only 600 to obtained the $B^{\mathrm{test}}_{\mathrm{R,pr}}$ dataset because the random movement of the led to a faster extinction time (i.e., fewer images per run).

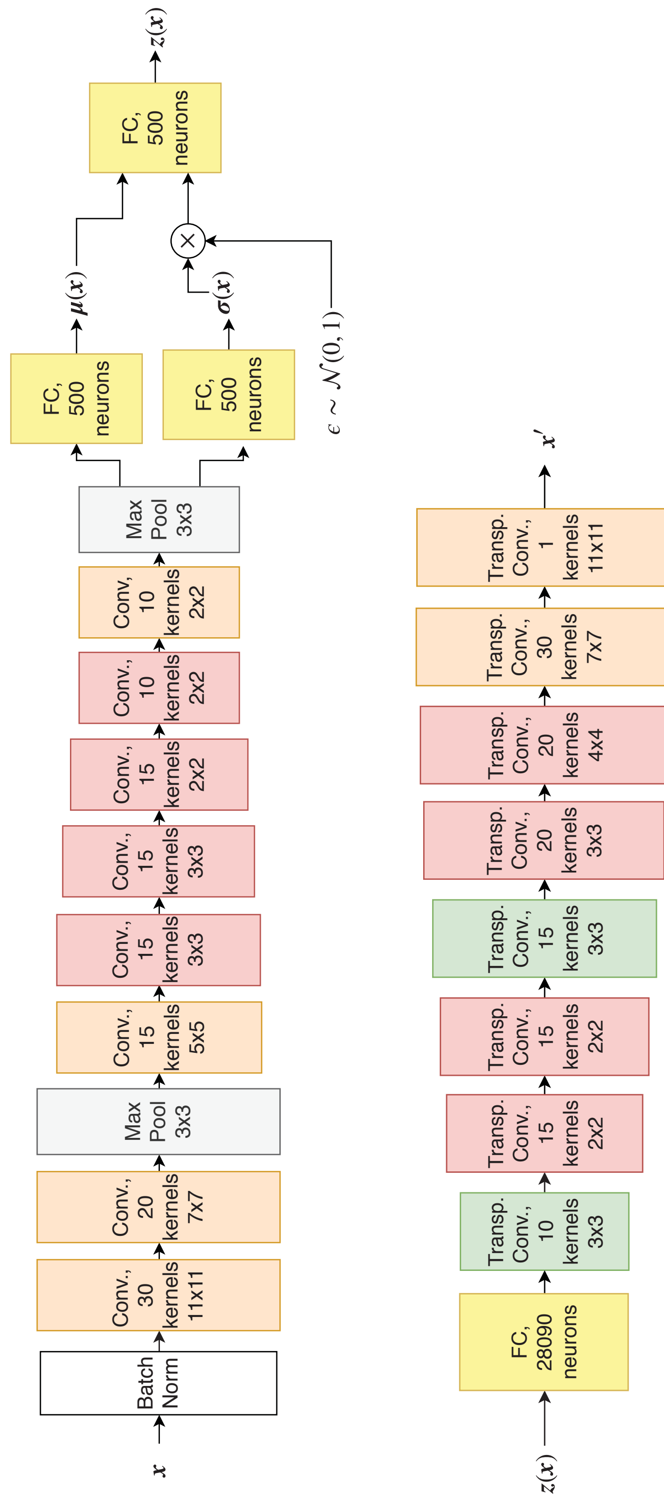

Figure 5.4: Our VAE: the encoder (top) and the decoder (bottom). The encoder consists of a batch-normalization layer, 8 convolutional layers, 2 max-pooling layers and 3 fully connected layers. The decoder has 1 fully connected layer and 8 transposed convolutional layers. Orange rectangles have the stride 1×1 and padding, red rectangles have the same stride but no padding, green rectangles have the stride 2×2 and no padding. Max-pooling layers have the stride 2×2 and all layers have ReLu activation units.

5.1.4 Multiple Trials and Majority Class Accuracy

For every experiment, we run 10 independent trials. On each trial, a new neural network was randomly initialized, trained, and tested. Although the dataset split follows Table 5.2, each split was performed per trial by randomly selecting evolution runs. We then averaged accuracy of the networks over the trials.

Additionally, since the data splits were performed at the level of evolution runs, the test set could have a different number of images for each class. Therefore, for better analysis of the network's performance, we also compute the majority class accuracy (MCA):

Definition 5.1.1. Majority Class Accuracy is the best possible accuracy given by a classifier that always outputs the same class. It corresponds to the frequency of samples belonging to the majority class.

5.1.5 Experiment 1: Pure Training Data

In this experiment, we assumed that a training data set consists entirely of images depicting the normal behaviour (i.e., the class B_{L}). Our VAE was initialized with weights and biases drawn from $\mathcal{N}(0, 0.1)$. It was then trained on the image set $B^{\mathrm{train}}_{\mathrm{L,pure}}$. The learning rate followed a stepwise cooling schedule with an initial value of 0.0001 and the decay of 0.8 on each epoch. We also used $\lambda = 0.0001$ as the parameter for the L1 regularization. The training had three stopping criteria: (i) a maximum of 100 epochs, (ii) the UP test (Prechelt, 1998) with strip of $k = 5$ epochs and $s = 3$ consecutive strips, (iii) the absolute difference between the validation error of two consecutive batches is not higher than 0.0001 in a total of 5 batches. The validation error was computed on the data set $B^{\mathrm{val}}_{\mathrm{L,pure}}$. Adam optimizer (Kingma and Ba, 2014) with the batch size of 50 images was used. Once the training was completed, we selected the threshold θ using $\left(B^{\theta}_{\mathrm{L,pure}}, B^{\theta}_{\mathrm{S,pure}}\right)$. This threshold was then used to detect anomalies in the test set. After that we evaluated the VAE on $\left(B^{\mathrm{test}}_{\mathrm{L,pure}}, B^{\mathrm{test}}_{\mathrm{F,pure}}\right)$ and on $\left(B^{\mathrm{test}}_{\mathrm{L,pure}}, B^{\mathrm{test}}_{\mathrm{R,pure}}\right)$. The resulting test accuracy is presented in Table 5.3 and discussed below.

To put the test accuracy of our VAE in a context, we also trained GoogLeNet (Szegedy et al., 2015) to classify $\left(B^{\text{test}}_{\text{L,pure}}, B^{\text{test}}_{\text{F,pure}}\right)$ and $\left(B^{\text{test}}_{\text{L,pure}}, B^{\text{test}}_{\text{R,pure}}\right)$. GoogLeNet requires both classes to be present and labelled in the training data. Thus, while our VAE was trained only on $B^{\text{train}}_{\text{L,pure}}$, GoogLeNet was trained on $\left(B^{\text{train}}_{\text{L,pure}}, B^{\text{train}}_{\text{F,pure}}\right)$. We used a version of GoogLeNet included in the Machine Learning toolbox with MATLAB R2018a. This network was trained with Adam optimizer from the MATLAB toolbox using the same learning schedule as we used for the VAE and the batch size of 50 images. The training stopped either when 100 epochs where reached or when the validation error computed every 20 batches on $\left(B^{\text{val}}_{\text{L,pure}}, B^{\text{val}}_{\text{F,pure}}\right)$ did not improve over the previously smallest validation error three times in a row.

Additionally, since GoogLeNet included with MATLAB had 1000 neurons in its output layer, and we have only two classes, we replaced the output layer with a new layer of two neurons. To compensate for the lack of ImageNet-based pre-training for the weights leading to the new layer, we increased the layer's learning rate ten folds.[7]

[7] A version of GoogLeNet initialized with random weights on all layers yielded similar results.

Algorithm 2: `Predator/prey A-life environment with injection`

input : Grid world $C^{k\times k}$; time steps T; initial population sizes $n^{\mathrm{R}}_{\mathrm{init}}, n^{\mathrm{W}}_{\mathrm{init}}$; initial grass per cell $n^{\mathrm{g}}_{\mathrm{init}}$; grass growth $g^{\mathrm{init}}_{\mathrm{grow}}$; max ages a_{max}; reproduction age a_{repr}; reproduction energy e_{repr}; min energy e_{min}; max grass g_{max}; growth multiplier $g^{\mathrm{mult}}_{\mathrm{grow}}$; mutation rate δ_{rate}; probability η

output: Set I of image frames

1 $G^{k\times k} \leftarrow$ initial grass matrix with $n^{\mathrm{g}}_{\mathrm{init}}$
2 $R \leftarrow$ initial population with $n^{\mathrm{R}}_{\mathrm{init}}$ rabbits (random with probability η)
3 $W \leftarrow$ initial population with $n^{\mathrm{W}}_{\mathrm{init}}$ wolves
4 $I \leftarrow \emptyset$; $t \leftarrow 0$; $g_{\mathrm{grow}} \leftarrow g^{\mathrm{init}}_{\mathrm{grow}}$
5 **while** $t \leq T \wedge R \neq \emptyset \wedge W \neq \emptyset$ **do**
6   **for** $\forall A \in R \cup W$ **do**
7         $N \leftarrow$ set of cells within A sight radius
8         $U(A, n) \leftarrow$ ultility of cell n in respect to agent $A, \forall n \in N$
9   **if** $A \in R \wedge A$ *is random* **then**
10   move A randomly
11   **else**
12   move A torwards $\mathrm{argmax}(U(a, n))$
13   **end**
14   **if** $A \in R$ **then**
15   A eats grass in cell of A;
16   **else**
17   A eats random rabbit in position of A;
18   **end**
19         $a \leftarrow A$'s age $+1$
20         $e \leftarrow A$'s energy
21   age of $A \leftarrow a$
22   **if** $a > a_{max} \vee e < e_{min}$ **then**
23   **if** $A \in R$ **then**
24                 $R \leftarrow R \setminus \{A\}$
25   **else**
26                 $W \leftarrow W \setminus \{A\}$
27   **end**
28   **end**
29   **if** $a > a_{repr} \wedge e > e_{repr}$ **then**
30             $O \leftarrow$ mutated A according to δ_{rate}
31   energy of $O \leftarrow e/2$
32   energy of $A \leftarrow e/2$
33   **if** $A \in R$ **then**
34                 $R \leftarrow R \cup \{O\}$; with probability η makes O random
35   **else**
36                 $W \leftarrow W \cup \{O\}$
37   **end**
38   **end**
39   **end**
40     $g_{\mathrm{grow}} \leftarrow g_{\mathrm{grow}} * g^{\mathrm{mult}}_{\mathrm{grow}}$
41     $G[x, y] \leftarrow \max\left(G[x, y] + g_{\mathrm{grow}}, g_{\mathrm{max}}\right), \forall (x, y) \in G^{k\times k}$
42     $I \leftarrow I \cup \{\text{screenshot at time } t\}$
43     $t \leftarrow t + 1$
44 **end**
45 **return** I

Table 5.3: Accuracy of behaviour classifiers on pure data. B_{R} was collected with $\eta = 50\%$. The top half of the table has the results for the detection of B_{F} anomaly, while the bottom half has the results for the detection of B_{R} anomaly.

Train set	**θ-select set**	**Validation set**	**Classifier**	$\left(B^{\text{test}}_{\text{L,pure}}, B^{\text{test}}_{\text{F,pure}}\right)$
$\left(B^{\text{train}}_{\text{L,pure}}, B^{\text{train}}_{\text{F,pure}}\right)$		$\left(B^{\text{val}}_{\text{L,pure}}, B^{\text{val}}_{\text{F,pure}}\right)$	GoogLeNet	$\mathbf{95.4 \pm 3.7\%}$
$B^{\text{train}}_{\text{L,pure}}$	$\left(B^{\theta}_{\text{L,pure}}, B^{\theta}_{\text{S,pure}}\right)$	$B^{\text{val}}_{\text{L,pure}}$	VAE	$77.4 \pm 5.1\%$
			MCA	$58.2 \pm 8.1\%$

Train set	**θ-select set**	**Validation set**	**Classifier**	$\left(B^{\text{test}}_{\text{L,pure}}, B^{\text{test}}_{\text{R,pure}}\right)$
$\left(B^{\text{train}}_{\text{L,pure}}, B^{\text{train}}_{\text{F,pure}}\right)$		$\left(B^{\text{val}}_{\text{L,pure}}, B^{\text{val}}_{\text{F,pure}}\right)$	GoogLeNet	$66.4 \pm 9.0\%$
$B^{\text{train}}_{\text{L,pure}}$	$\left(B^{\theta}_{\text{L,pure}}, B^{\theta}_{\text{S,pure}}\right)$	$B^{\text{val}}_{\text{L,pure}}$	VAE	$\mathbf{89.8 \pm 3.1\%}$
			MCA	$53.2 \pm 1.8\%$

Results. The means and the standard deviations of test accuracy are listed in Table 5.3. They were measured over ten trials. Each trial differed in the initialization of the VAE's weights and biases, GoogLeNet weights for the last layer, and the randomization of the training process using Adam optimizer. Additionally, the dataset was randomly split into training, validation, threshold selection, and test on each trial.

Our VAE never had access to anomalous data during training; however, GoogLeNet had access to the B_{F} anomaly during training, but it did not had access to the B_{R} anomaly during training. When labelled anomalies B_{F} were present in the training set, GoogLeNet outperformed our VAE (95.4% vs 77.4%); however, when no labelled anomalies B_{R} were present in the training set, GoogleNet was outperformed by our VAE (66.4% vs 88.9%). Our VAE had access only to normal data during training, yet it still obtained a good detection accuracy for both B_{F} (77.4%) and B_{R} (88.9%).

Although our VAE outperformed GoogLeNet when detecting random anomaly B_{R} with $\eta = 50\%$, this was not the case for every η. When $\eta = 40\%$, GoogLeNet surprisingly achieved a mean accuracy of 84.1% when detecting B_{R} even though no samples from this class was

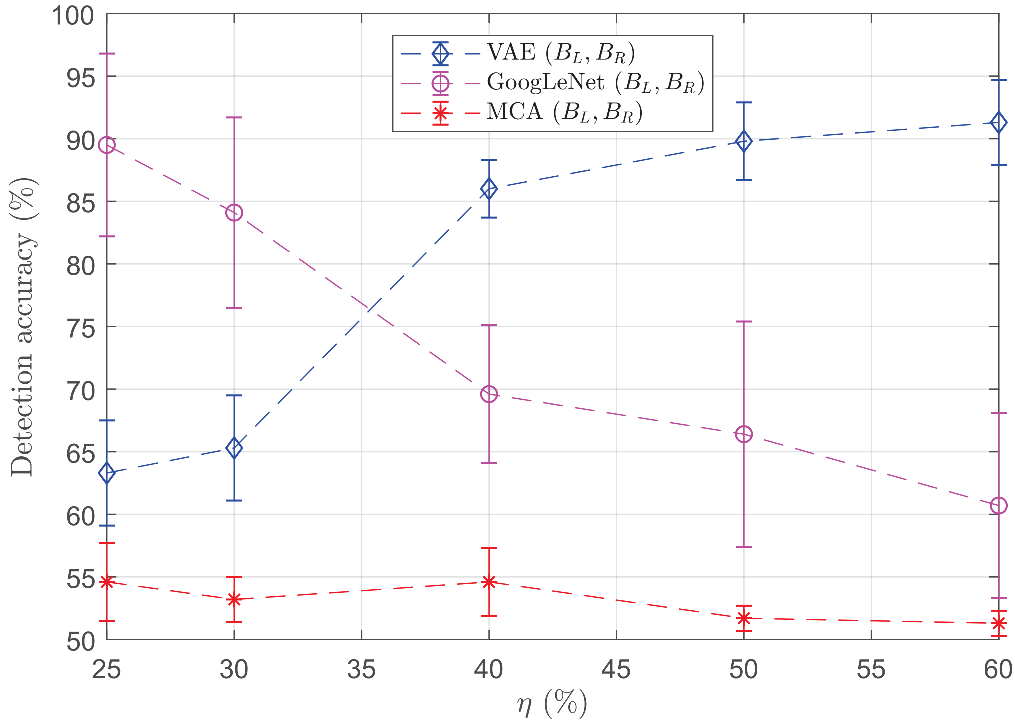

Figure 5.5: VAE's and GoogLeNet detection accuracy variation according to the probability η of injecting random agents. The error bars show standard deviation.

provided during training time. We suspect that the probability η of injecting rabbits may affect the evolution in different ways. For some values of η, the features learned when training GoogLeNet on the $(B_\mathrm{L}, B_\mathrm{F})$ may be sufficient to also distinguish B_L from B_R. To better understand how the detection accuracy is affected by the probability η of injecting random moving agents in the environment, we built the plot in Figure 5.5. Starting with $\eta = 40\%$ GoogLeNet is outperformed by the VAE. We can see that both models are affected by similarities between normal and anomalous patterns. GoogLeNet's performance for small η might be explained by the random rabbit's lack of adaptation. Since random rabbits are not evolved, they tend to die faster. Hence, for $\eta < 40\%$, the number of random moving agents born in each time step may not be enough to compensate for their mortality rate, leading to a lower impact of their population in the evolution as a whole. Thus, for small η, B_R images may be similar enough to B_F that GoogLeNet achieved accuracy above 90%.

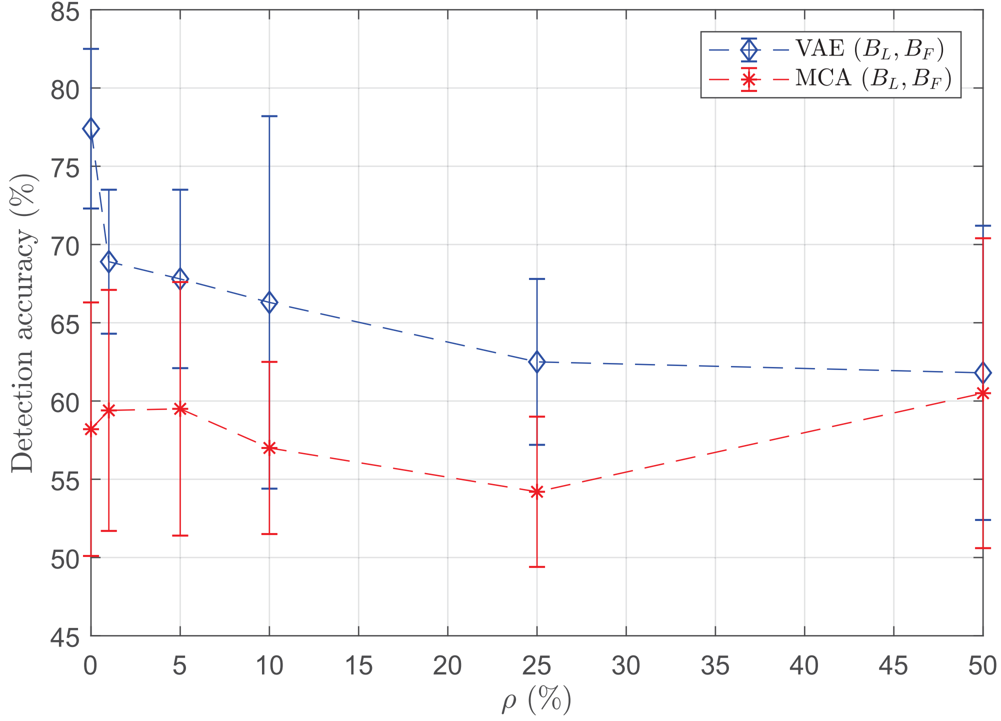

Figure 5.6: Effects of pollution degree ρ on VAE test accuracy.

5.1.6 Experiment 2: Polluted Training Data

In the previous experiment, we trained the VAE purely on normal behaviours of A-life agents (i.e., class B_{L}). Having such a pure training set is not always feasible as, during a typical evolution run, a small number of images representing anomalous behaviours may be recorded in the training set. In this section, we study the degradation of VAE test accuracy when it is trained on polluted image sets. We vary the degree of pollution by injecting different numbers of anomalous B_{F} images into the training data set.

We formed polluted sets by combining classes B_{F} and B_{L} images in proportion p and $1-p$ where $p \in \{0, 0.01, 0.05, 0.1, 0.25, 0.5, 0.75\}$. Thus the new image set $B^{\mathrm{train}}_{\mathrm{L,polluted}}$ was a mixture of $B^{\mathrm{train}}_{\mathrm{L,pr}}$ and $B^{\mathrm{train}}_{\mathrm{F,pure}}$, $B^{\mathrm{val}}_{\mathrm{L,polluted}}$ was a mixture of $B^{\mathrm{val}}_{\mathrm{L,pure}}$ and $B^{\mathrm{val}}_{\mathrm{F,pure}}$ and $B^{\theta}_{\mathrm{L,pollutes}}$ was a mixture of $B^{\theta}_{\mathrm{L,pure}}$ and $B^{\theta}_{\mathrm{F,pure}}$. After that $B^{\theta}_{\mathrm{S,polluted}}$ was created by permuting image pixels in $B^{\theta}_{\mathrm{L,polluted}}$.

We then trained the VAE on $B^{\text{train}}_{\text{L,polluted}}$ with $B^{\text{val}}_{\text{L,polluted}}$ used for validation. The threshold θ was computed on $\left(B^{\theta}_{\text{L,polluted}}, B^{\theta}_{\text{S,polluted}}\right)$. Finally, the VAE was tested on $\left(B^{\text{test}}_{\text{L,pure}}, B^{\text{test}}_{\text{F,pure}}\right)$ and $\left(B^{\text{test}}_{\text{L,pure}}, B^{\text{test}}_{\text{R,pure}}\right)$.

Results. Figure 5.6 shows VAE's mean test accuracy as a function of p. For instance, with 1% of anomalous images mixed in with the normal images, the VAE test accuracy is 68.9% for the B_{F} anomaly, compared to 77.4 for the pure case.

Chapter 6

Anomalous Behaviour Detection in a Video Game

In this chapter, we present experiments performed with the commercial game *Project Hastur* (Polymorphic Games, 2019). In this tower-defence game, procedurally generated NPCs gradually evolve according to the players' strategy and playstyle. We applied our VAE to the detection of emergent anomalous behaviours. Figure 6.1 shows a screen capture of the game, where the genetically evolved enemies (i.e., Proteans) can be seen up close. More details about how the data were collected can be found in Appendix C. Parts of this chapter were adapted from our previous work (Soares et al., 2018).

6.1 *Project Hastur*

Project Hastur is a strategy game in which the players can position different types of turrets to defend the main base tower from waves of enemy NPCs (i.e., Protean Swarms). The gameplay is divided into generations until the Proteans destroy all the player's towers or a target number of generations is reached. At the beginning of every generation, the player has some time of safety (i.e., no enemies are spawned), after which a new generation/wave of Proteans is spawned. The player has a certain amount of biomatter (i.e., in-game money)

Figure 6.1: Screen capture taken from *Project Hastur*.

that can be used to buy new turrets or towers, upgrade turrets, and restore any damage done to turrets or towers. Biomatter is obtained by killing Proteans and surviving a generation. The enemies can attack the towers, turrets, and the civilian. Upon attacking and killing the later, Proteans can reproduce, generating children depending on their size (i.e., smaller enemies have more children than larger enemies). Figure 6.2 illustrates *Project Hastur*'s gameplay.

Each Protean has a genome that determines it individual traits (e.g., sight range, resistances, speed, attack rate and damage,), morphology (e.g., body size, limb size, shape, colour), special abilities (jumping and swimming), and behaviour (attack preferences). Although some of these characteristics (i.e., phenotypes) are observable by the player, the genome itself is not. After each generation, new Proteans are generated through genetic evolution following two fitness functions. Half of the new population is created as offspring from parents selected according to the damage taken from turrets. The other half of the new population is selected from parents according to the parent's proximity to the tower:

Figure 6.2: *Project Hastur*: the main and base towers are the white/gray structures surrounded by turrets (purple, blue and red). Proteans are the spider-like NPCs, while civilians are the human-like NPCs.

Proteans that survive long enough to get near to a tower have a higher chance to reproduce. Additionally, Proteans can reproduce by consuming another type of NPC: civilians. Upon this attack and consumption, Proteans can generate one or more offspring according to their size (i.e., smaller ones generate more offspring than big ones). This reproduction leads to a secondary evolution path in which Proteans may adapt to attack civilians.

Finally, the game can be played on two modes: campaign or experiment. In the first mode, the game is played in different maps, while the player must survive a target number of generations. The experiment mode is the one we used to collect the data for our experiments. This game mode provides a controlled environment in which we can manually set parameters of the gameplay (e.g., number of generations, upgrades) and from the evolution (e.g., mutation rate, cross-rate).

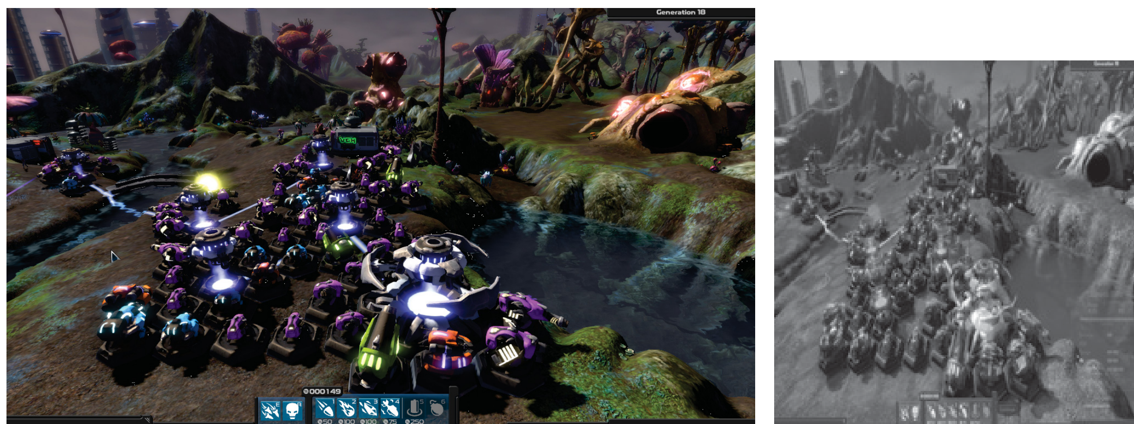

Figure 6.3: A original image frame taken from *Project Hastur* (left) and the corresponding features as seen by our VAE (right).

6.2 Collecting Data

Following the fitness functions mentioned above, the enemy population may evolve several types of behaviours depending on the player's playstyle. To reduce variance between game runs, we fixed the playstyle while playing in the experiment mode whose hyperparameters for the genetic algorithm were also fixed. Details of the playstyle and the hyperparameters can be found in Appendix C.1. In order to collect data, we used an external screen capture program to collect images from the game at every second. These colour images were then converted to grayscale. After that the features $f\left(S_{t-\tau}, \ldots, S_t\right)$ were created by frame averaging (Section 4.3) and then resized from 1920×1080 to 224×224 pixels to use the same VAE topology as in our A-life experiments. Figure 6.3 shows an original screen capture (left) and the resulting image as seen by the VAE (right).

A total of 20 runs were performed, each one played from generation 0 to 20, which gave a total of 45125 images. Since manually labelling the images was time-consuming (Section 6.3), we reduced the dataset to approximately 13.3%, randomly labelling 300 images from each run. Therefore, the final dataset had in total of 6000 images. More details about the parameters used in the experiment mode can be found in Appendix C.2. Due to time constraints and lack of human resources, we only had a single person to play the game and

Table 6.1: Frequencies for each observed behaviour B_ψ in the *Project Hastur* data.

Target class	B_ψ	Frequency (%)
B_N	$\neg B_\text{biped} \land \neg B_\text{swim} \land \neg B_\text{jump}$	$84.2 \pm 8.5\%$
B_B	B_biped	$7.9 \pm 7.8\%$
$B_\text{S,J}$	$B_\text{swim} \lor B_\text{jump}$	$7.9 \pm 4.4\%$

label the data. This limitation led to only 20 game runs worthy of data which took about 40 hours to collect. The process of labelling the 6000 selected images took about 500 hours, giving the estimation of 3761 hours for the complete dataset (i.e., 45125 images). Future work may crowd-source the task to collect more data and to reduce the error in the labelling processing (i.e., use voting to decide on the labels).

6.3 Labeling the Data

- $B_\text{B} = B_\text{biped}$ if a Protean capable of walking on two limbs was observed;
- $B_\text{S,J} = B_\text{swim} \lor B_\text{jump}$ if a Protean capable of jumping the rivers or swimming was observed;
- $B_\text{N} = \neg B_\text{biped} \land \neg B_\text{swim} \land \neg B_\text{jump}$ if none of the previous behaviours was observed

In order to evaluate how our model performs on *Project Hastur* data, we had to label the test images as normal or anomalous manually. Our normal data was composed of samples from B_N, while our two types of anomalies are given by B_B and $B_\text{S,J}$. As shown by Table 6.1, B_B and $B_\text{S,J}$ are rare, thus are considered anomalous. Figure 6.4, 6.5, and 6.6 show images from the B_N, B_B, and $B_\text{S,J}$ respectively; Figure 6.7 show the correspondent $f\left(S_{t-\tau}, \ldots, S_t\right)$ as seem by our VAE.

The dataset was then balanced by randomly selecting images such that all classes have the same number of samples. Due to the rarity of B_B and $B_\text{S,J}$, each dataset ended up with only 477 images after balancing. Finally, for each trial, a new split of runs into training (B_N^train and B_B^train), validation (B_N^val and B_B^val), threshold θ selection (B_N^θ), and test (B_N^test,

Figure 6.4: Normal B_{L} image (i.e., no bipedal Proteans nor NPCs capable of swimming or jumping).

$B_{\mathrm{B}}^{\mathrm{test}}$, and $B_{\mathrm{S,J}}^{\mathrm{test}}$) was made similarly to Section 5.1.2. Additionally, the set B_{N}^{θ} was used to create the surrogate anomalies for the set B_{S}^{θ}. The number of game runs used in each set is shown by Table 6.2. Since our VAE does not require a balanced dataset for learning, we trained two VAEs: $\mathrm{VAE_u}$ trained on an unbalanced dataset but tested on a balanced dataset, and $\mathrm{VAE_b}$ trained and tested on a balanced dataset.

6.4 Training VAE

A total of 20 game runs were performed, from which nine independent game runs were used during training, three used for threshold selection, three used for validation (i.e., early stop), and five for testing. We performed 10 independent trials (i.e., training and testing an individual VAE) using the same hyperparameters as in Section 5.1.5. Our VAE was trained on $B_{\mathrm{N}}^{\mathrm{train}}$ and evaluate with $(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{B}}^{\mathrm{test}})$ and $\left(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{S,J}}^{\mathrm{test}}\right)$. We used $B_{\mathrm{N}}^{\mathrm{val}}$ to compute the validation required in one of the stopping criteria (Section 5.1.5). Finally we automatically

Figure 6.5: Image from class B_{B} (i.e., bipedal Proteans). Red circle marks the presence of bipedal agents.

tuned the VAE's threshold on $\left(B_{\mathrm{N}}^{\theta}, B_{\mathrm{S}}^{\theta}\right)$. We also tested GoogLeNet on $(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{B}}^{\mathrm{test}})$ and $\left(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{S,J}}^{\mathrm{test}}\right)$. For both test sets, GoogLeNet was trained with the same hyperparameters as in the experiments in Section 5.1.5 using $(B_{\mathrm{N}}^{\mathrm{train}}, B_{\mathrm{B}}^{\mathrm{train}})$ as the training set and $\left(B_{\mathrm{N}}^{\mathrm{val}}, B_{\mathrm{B}}^{\mathrm{val}}\right)$ as the validation set.

6.4.1 Results and Discussion

Table 6.3 shows the mean accuracy for our VAEs and for GoogLeNet. GoogLeNet had a slightly higher mean than the VAEs for both cases, $(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{B}}^{\mathrm{test}})$ and $\left(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{S,J}}^{\mathrm{test}}\right)$. Both VAEs and GoogLeNet obtained accuracy similar to MCA: 60.4% when detecting B_{B} and 54.8% when detecting $B_{\mathrm{S,J}}$. This result means that both GoogLeNet and VAEs failed to distinguish between normal and anomalous data. We believe that the image complexity contributed to the failure. Figures 6.8, 6.9, and 6.10 show representatives of the B_{N}, B_{B}, and $B_{\mathrm{S,J}}$, respectively. Although the agents and behaviours were clearly visible in the colour images used in the labelling process, they can be barely seen in the downsampled grayscale

Figure 6.6: Image from class $B_{\mathrm{S,J}}$ (i.e., Proteans capable of jumping or swimming). Red circle marks the presence of an agent jumping.

images. For this reason, even a human observer has difficulties distinguishing between the classes. We believe that the data preprocessing (downsampling and exponential decay averaging) discards too much information.

Figure 6.11 gives an insight into the VAE's performance. We gave an image from our A-life environment (Figure 6.11, left) to the $\mathrm{VAE_u}$ trained on B_{N} data. The reconstructed image resembled an image from *Project Hastur* (Figure 6.11, centre) suggesting that the VAE mostly memorized information about the background map and the main tower. Thus the VAE might be discarding Proteans as noise in the image, which would make it nearly impossible to recognize behaviours patterns of the Proteans.

Figure 6.7: Features $f(S_{t-\tau}, \ldots, S_t)$ for classes B_{L}, B_{B}, and $B_{\mathrm{S,J}}$ respectively.

Table 6.2: Number of game runs used in each data split.

Split	Balance	**Dataset**	**Number of runs**
complete dataset		$(B_{\mathrm{N}}, B_{\mathrm{B}}, B_{\mathrm{B,J}})$	(20, 20, 20)
train	balanced	$(B_{\mathrm{N}}^{\mathrm{train}}, B_{\mathrm{B}}^{\mathrm{train}})$	(9, 9)
train	balanced	$B_{\mathrm{N}}^{\mathrm{train}}$	9
train	unbalanced	$B_{\mathrm{N+}}^{\mathrm{train}}$	9
validation	balanced	$\left(B_{\mathrm{N}}^{\mathrm{val}}, B_{\mathrm{B}}^{\mathrm{val}}\right)$	(3, 3)
validation	balanced	$B_{\mathrm{N}}^{\mathrm{val}}$	3
validation	unbalanced	$B_{\mathrm{N+}}^{\mathrm{val}}$	3
θ selection	balanced	$\left(B_{\mathrm{N}}^{\theta}, B_{\mathrm{S}}^{\theta}\right)$	(3, 3)
θ selection	unbalanced	$\left(B_{\mathrm{N+}}^{\theta}, B_{\mathrm{S+}}^{\theta}\right)$	(3, 3)
test	balanced	$(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{B}}^{\mathrm{test}})$	(5, 5)
test	balanced	$\left(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{S,J}}^{\mathrm{test}}\right)$	(5, 5)

Table 6.3: Accuracy of behaviour classifiers on *Project Hastur* data.

Training set	**θ-selection set**	**Validation set**	**Classifier**	$(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{B}}^{\mathrm{test}})$	$\left(B_{\mathrm{N}}^{\mathrm{test}}, B_{\mathrm{S,J}}^{\mathrm{test}}\right)$
$(B_{\mathrm{N}}^{\mathrm{train}}, B_{\mathrm{B}}^{\mathrm{train}})$		$\left(B_{\mathrm{N}}^{\mathrm{val}}, B_{\mathrm{B}}^{\mathrm{val}}\right)$	GoogLeNet	$60.0 \pm 9.1\%$	**$66.7 \pm 5.8\%$**
$B_{\mathrm{N}}^{\mathrm{train}}$	$\left(B_{\mathrm{N}}^{\theta}, B_{\mathrm{S}}^{\theta}\right)$	$B_{\mathrm{N}}^{\mathrm{val}}$	$\mathrm{VAE_b}$	$52.7 \pm 10.4\%$	$54.1 \pm 6.7\%$
$B_{\mathrm{N+}}^{\mathrm{train}}$	$\left(B_{\mathrm{N+}}^{\theta}, B_{\mathrm{S+}}^{\theta}\right)$	$B_{\mathrm{N+}}^{\mathrm{val}}$	$\mathrm{VAE_u}$	$53.9 \pm 11.7\%$	53.2 ± 6.7
			MCA	**$60.4 \pm 6.6\%$**	54.8 ± 5.5

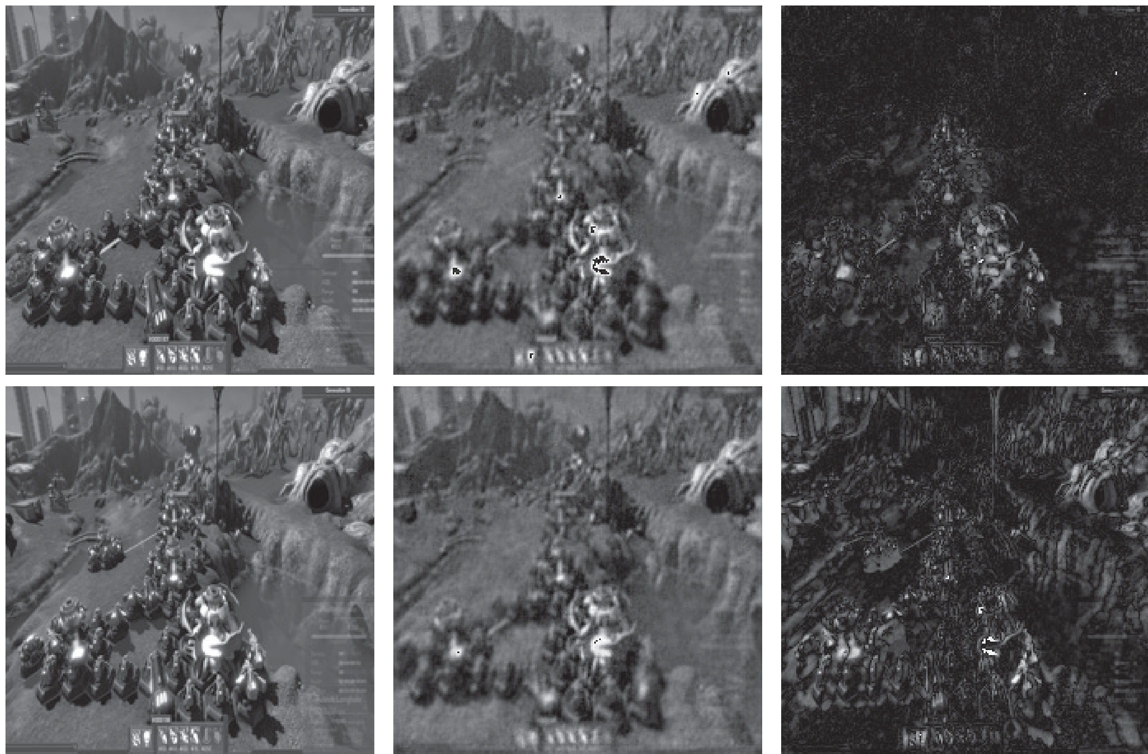

Figure 6.8: The top row shows an image from B_{N} correctly by the $\mathrm{VAE}_{\mathrm{u}}$. The bottom row shows a B_{N} incorrectly classified by the $\mathrm{VAE}_{\mathrm{u}}$. The left column shows the original images, the center column has the images reconstructed by our $\mathrm{VAE}_{\mathrm{u}}$, and the right column has the residual image between original and reconstruction.

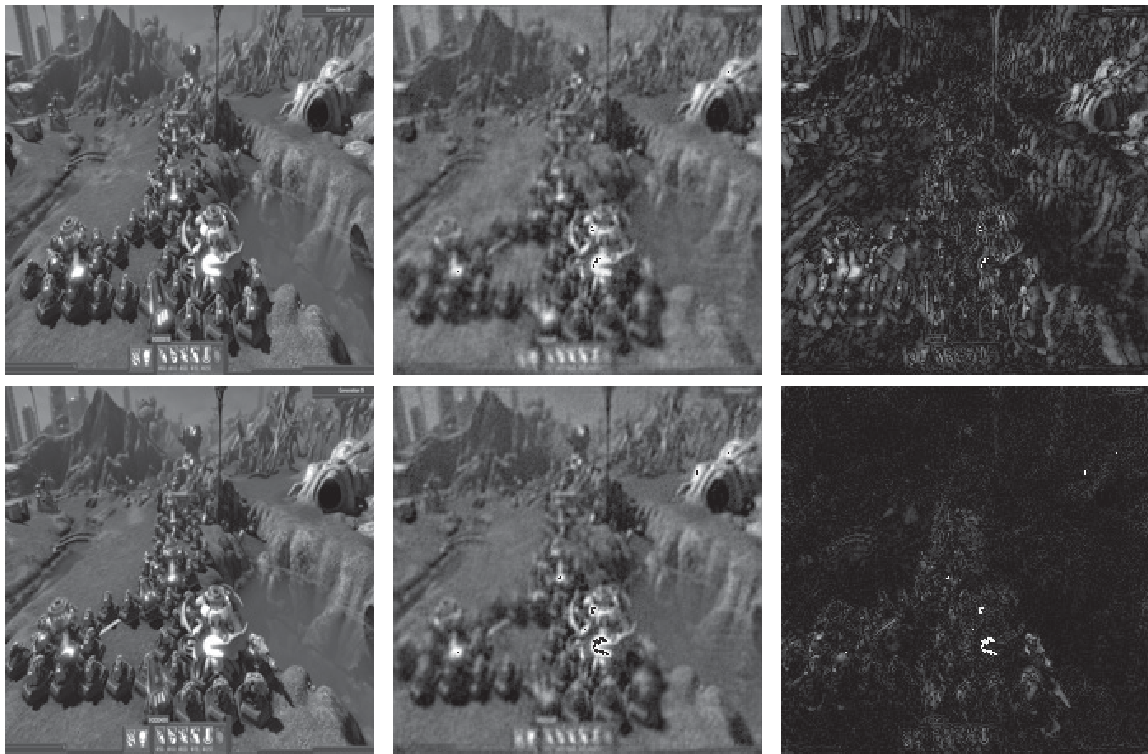

Figure 6.9: The top row shows an image from B_{B} correctly by the $\mathrm{VAE}_{\mathrm{u}}$. The bottom row shows a B_{B} incorrectly classified by the $\mathrm{VAE}_{\mathrm{u}}$. The left column shows the original images, the center column has the images reconstructed by our $\mathrm{VAE}_{\mathrm{u}}$, and the right column has the residual image between original and reconstruction.

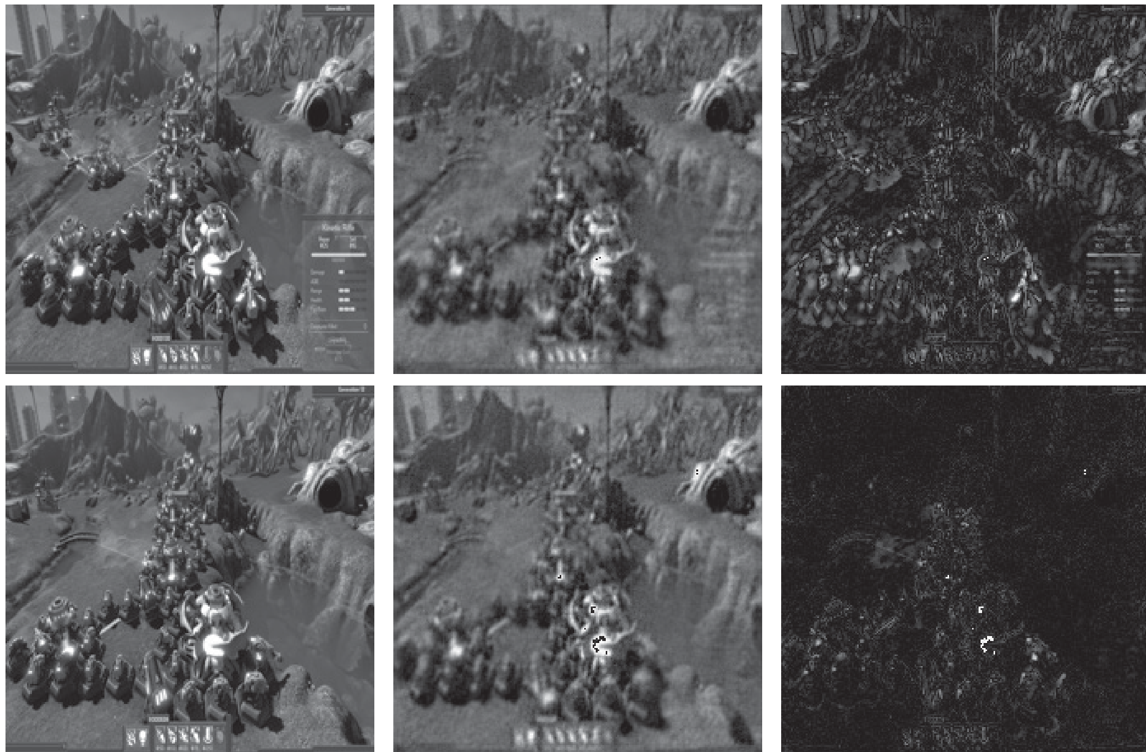

Figure 6.10: The top row shows an image from $B_{S,J}$ correctly by the VAE_u. The bottom row shows a $B_{S,J}$ incorrectly classified by the VAE_u. The left column shows the original images, the center column has the images reconstructed by our VAE_u, and the right column has the residual image between original and reconstruction.

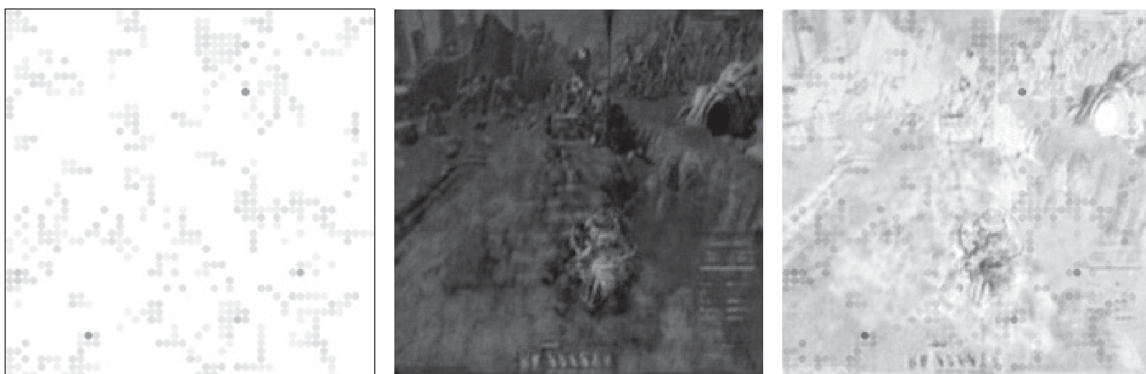

Figure 6.11: A frame average image from our A-life simulation (left), the correspondent reconstruction given by a VAE_u trained on *Project Hastur* data (middle), and the residual image between them (right).

Chapter 7

Future Work

In this chapter, we present a summary of the shortcomings in our work and the future research directions to solve them. We also discuss methods to collect and label data, human error associated with it, and possible solutions.

7.1 Future Research Directions

Although our VAE was capable of achieving an accuracy above 80% when detecting some anomalies (e.g., B_{R} for $\eta \geq 40\%$) in A-life environments, it failed to rise above MCA in *Project Hastur*. The unsatisfactory accuracy might be due to a combination of factors: human mistakes in the labelling processing, low-resolution images, and sub-optimal network topology. The use of NEAT (Stanley and Miikkulainen, 2002) may help improve the network's topology for higher resolution images. Data from *Project Hastur* or any commercial game also tends to have information irrelevant to the detection of behaviours such as the Heads Up Display and a fixed background/map. The use of NEAT may also help to find topologies capable of dealing with this extra information.

Additionally, the data for *Project Hastur* was collected and labelled manually. Unlike our A-life environment, when the B_{F} and B_{R} were algorithmically detected, the behaviours of *Project Hastur* were inferred through human observation. Although our model does not

Figure 7.1: *Project Hastur*'s default camera angle.

require labels during training, the evaluation may suffer from errors in target labels. The process was also time-consuming; we estimated about 3761 hours to label all the collected images. Future work may crowd-source this task.

The VAEs topology tuned for our two-dimensional A-life environment may not have enough capacity to detect behaviours in a three-dimensional game environment (*Project Hastur*) with several camera angles. We fixed the camera angle between all game runs by using *Project Hastur*'s default camera angle (Figure 7.1), which makes it harder to visualize distant agents. Further work will consider using neuroevolution to search for a new topology better suited to complex environments. Also in future work, a stack of simulation frames may be given as input instead of frame averaging we used.

Finally, we used images as inputs for our detector since they are readily available features in the simulations used in this thesis. However, this may not always be the case. For example, some multi-agent simulation may not have a visualization for the agents and environments; instead, the readily available features may assume other forms such as audio or electrical

signals, mathematical functions, and others. As long as we can represent the features as a vector $\boldsymbol{x} \in \mathbb{R}^k$, then the VAE can still be used for anomaly detection, as proposed in this chapter. It is important to note, however, that the topology presented in Section 5.1.3 expects an input with a fixed size of 224×224. Hence, an input with a different dimension from ours will lead to a number of parameters (i.e., weights and biases) different from our VAE. For future work, we may explore the use of neuroevolution to optimize a VAE topology for the problem in hand independently of the input size. We also believe that the approach here proposed might be applied to different anomaly detection tasks (i.e., equipment failures, intrusion detection); however, in this thesis we focused only on the detection of anomalous emergent collective behaviours in multi-agent simulations. Future work may explore new input data and tasks/environments.

Chapter 8

Conclusion

In this thesis, we proposed the use of a deep variational autoencoder (VAE) to detect emergent anomalous behaviours in populations AI of agents. We also presented a specific VAE topology and evaluated it empirically in two different settings: a predator/prey A-life environment and a commercial video game. Our approach does not require labelled data during training, being also robust to a mixture of normal and anomalous training data. When evaluated in our A-life environment, the autoencoder displayed good accuracy. This performance may be further improved through the use of neuroevolution to tune the parameters of our model. The results for the commercial game, *Project Hastur*, were inconclusive as both the autoencoder and GoogLeNet failed to detect anomalous behaviours. We believe that the challenge of the task is due to the image preprocessing which discarded too much information.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/.

David Ackley and Michael Littman. Interactions Between Learning and Evolution. *Artificial life II*, 10:487–509, 1991.

Muhammad Aurangzeb Ahmad, Brian Keegan, Jaideep Srivastava, Dmitri Williams, and Noshir Contractor. Mining for Gold Farmers: Automatic Detection of Deviant Players in MMOGs. In *Proceedings of 2009 International Conference on Computational Science and Engineering*, volume 4, pages 340–345, 2009.

Andrea Asperti. Sparsity in variational autoencoders. *arXiv preprint arXiv:1812.07238*, 2018.

Vadim Bulitko, Shelby Carleton, Delia Cormier, Devon Sigurdson, and John Simpson. Towards Positively Surprising Non-Player Characters in Video Games. In *Proceedings of the Experimental AI in Games (EXAG) Workshop at the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 34–40, 2017.

Chih-Chun Chen, Sylvia B Nagl, and Christopher D Clack. Specifying, Detecting and Analysing Emergent Behaviours in Multi-Level Agent-Based Simulations. In *Proceedings of the 2007 Summer Computer Simulation Conference*, pages 969–976. Society for Computer Simulation International, 2007.

Creature Labs. Creatures, 1996.

Creature Labs. Creatures 2, 1998.

Creature Labs. Creatures 3, 1999.

Creatures Wikia. Tortured Norns, 2004. URL https://creatures.fandom.com/wiki/Tortured_Norns.

Daybreak Game Company. EverQuest II, 2004.

Alexis Drogoul and Jacques Ferber. Multi-agent simulation as a tool for modeling societies: Application to social differentiation in ant colonies. In Cristiano Castelfranchi and Eric Werner, editors, *proceedings of Artificial Social Systems*, pages 2–23, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Solomon Kullback and Richard A Leibler. On Information and Sufficiency. *The ANNALS of Mathematical Statistics*, 22(1):79–86, 1951.

Chen Change Loy, Tao Xiang, and Shaogang Gong. Detecting and discriminating behavioural anomalies. *Pattern Recognition*, 44(1):117–132, 2011.

Ramin Mehran, Alexis Oyama, and Mubarak Shah. Abnormal Crowd Behavior Detection using Social Force Model. In *Proceedings of 2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 935–942, 2009.

Polymorphic Games. Darwin's Demons, 2017.

Polymorphic Games. Project Hastur, 2019.

Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

Eftychios Protopapadakis, Athanasios Voulodimos, Anastasios Doulamis, Nikolaos Doulamis, Dimitrios Dres, and Matthaios Bimpas. Stacked Autoencoders for Outlier Detection in Over-the-Horizon Radar Signals. *Computational Intelligence and Neuroscience*, 2017.

Lewis M. Killian Richard Pallardy Marco Sampaolo Ralph H. Turner, Neil J. Smelser. Collective behaviour, 2018. URL https://www.britannica.com/science/collective-behaviour.

Manassés Ribeiro, André Eugênio Lazzaretti, and Heitor Silvério Lopes. A study of deep convolutional auto-encoders for anomaly detection in videos. *Pattern Recognition Letters*, 105:13–22, 2018.

Jaqcqueline Ronson. How evolution can make video game minions into big bads: Natural selection is about to make the leap into digital worlds, 2016. URL https://www.inverse.com/article/22874-video-game-natural-selection-evolution-polymorphic-games-darwin-game-dynamics.

James Owen Ryan, Adam Summerville, Michael Mateas, and Noah Wardrip-Fruin. Toward Characters Who Observe, Tell, Misremember, and Lie. In *Proceedings of Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

Everton Schumacker Soares and Vadim Bulitko. Deep Variational Autoencoders for NPC Behaviour Classification. In *Proceedings of 2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2019.

Everton Schumacker Soares, Vadim Bulitko, Kacy Doucet, Morgan Cselinacz, Terence Soule, Samantha Heck, and Landon Wright. Learning to Recognize A-Life Behaviours. In *Proceedings of Advances in Cognitive Systems (ACS) Poster Collection*, 2018.

Terence Soule, Samantha Heck, Thomas E Haynes, Nicholas Wood, and Barrie D Robison. Darwin's Demons: Does Evolution Improve the Game? In *Proceedings of the European Conference on the Applications of Evolutionary Computation*, pages 435–451. Springer, 2017.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Penny Sweetser. *Emergence in games*. Cengage Learning, 2008.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1096–1103, 2008.

Uri Wilensky. Netlogo, 1999.

Ye Zhao, Brad Lehman, Roy Ball, Jerry Mosesian, and Jean-François de Palma. Outlier Detection Rules for Fault Detection in Solar Photovoltaic Arrays. In *Proceedings of 2013 Twenty-Eighth Annual IEEE Applied Power Electronics Conference and Exposition (APEC)*, 2013.

Appendix A

A-life Environment: Details

This appendix presents in details the control parameters used in our predator/prey A-life simulation.

A.1 A-life Environment: Parameters

Each agent in our A-life environment has an age, energy, cognition cost, and seven genes:

- $r_{\text{sight}} \in \mathbb{N}$ is the sight radius used to determine the number of cells observable by each agent;
- $\alpha_{\text{grass}} \in \mathbb{R}$ is the affinity to grass;
- $\alpha_{\text{rabbit}} \in \mathbb{R}$ is the affinity to rabbits;
- $\alpha_{\text{wolf}} \in \mathbb{R}$ is affinity to wolves;
- $\alpha_{\alpha_{\text{grass}}} \in \mathbb{R}$ is the affinity towards others agents affinity to grass;
- $\alpha_{\alpha_{\text{rabbit}}} \in \mathbb{R}$ is the affinity towards others agents affinity to rabbit;
- $\alpha_{\alpha_{\text{wolf}}} \in \mathbb{R}$ is the affinity towards others agents affinity to wolf.

Rabbits with $\alpha_{\text{grass}} > 0$ are said to like grass (i.e., leaders). Grass-disliking rabbits (i.e., followers) have $\alpha_{\text{grass}} < 0$. Additionally, the simulation is controlled by the hyperameters:

- $T \in \mathbb{N}$ is the maximum number of discrete time steps;
- $e^{W}_{gain}, e^{R}_{gain}$ are the wolf's and rabbit's food gain, respectively;
- $e^{W}_{repr}, e^{R}_{repr}$ are the minimum energy required to enable reproduction of wolves and rabbits, respectively;
- $a^{W}_{repr}, a^{R}_{repr}$ are the minimum age required to enable reproduction of wolves and rabbits;
- $m^{W}_{speed}, m^{R}_{speed}$ are the speed of wolves and rabbits, respectively;
- $n^{W}_{init}, n^{R}_{init}$ are the initial number of wolves and rabbits, respectively;
- n^{W}_{max}, n^{R}_{max} are the maximum size for the population of wolves and rabbits, respectively;
- a^{W}_{max}, a^{R}_{max} are the maximum age allowed for wolves and rabbits, respectively;
- e^{W}_{min}, e^{R}_{min} are the minimum energy necessary for survival (wolves and rabbits, respectively);
- e^{W}_{max}, e^{R}_{max} are the wolf's and rabbit's maximum energy, respectively;
- $e^{W}_{move}, e^{R}_{move}$ are the energy cost necessary for the movement of wolves and rabbits, respectively;
- e^{W}_{cog}, e^{R}_{cog} are the initial cognitive energy cost for wolves and rabbits, respectively;
- $e^{W}_{tick}, e^{R}_{tick}$ are the wolves' and rabbit's energy cost to keep existing in the system, respectively;
- $e^{W}_{init}; e^{R}_{init}$ are the initial energy of wolves and rabbits respectively;
- e^{mult}_{cog} is the cognitive energy cost multiplier;
- n^{G}_{init} is the initial minimum amount of grass per cell;
- g^{init}_{grow} is the initial grass growth amount per cell;

- $g_{\text{grow}}^{\text{mult}}$ is the grass growth multiplier;
- g_{max} is the maximum grass amount per cell;
- δ_{rate} is the mutation rate;
- r_{sight}^{init} is the initial maximum sight radius;
- k is the dimensions of grid world;

Table A.1 shows the actual values for these parameters used in our experiments.

Table A.1: Parameters for our A-life environemnt.

Parameter	**Value**	**Parameter**	**Value**	**Parameter**	**Value**
T	10000	$n_{\text{max}}^{\text{R}}$	100	$e_{\text{tick}}^{\text{R}}$	1
$e_{\text{gain}}^{\text{W}}$	44	$a_{\text{max}}^{\text{W}}$	1000000	$e_{\text{cog}}^{\text{mult}}$	0.0001
$e_{\text{gain}}^{\text{R}}$	83	$a_{\text{max}}^{\text{R}}$	1000000	$n_{\text{init}}^{\text{G}}$	166
$e_{\text{repr}}^{\text{W}}$	100	$e_{\text{min}}^{\text{W}}$	5	$g_{\text{grow}}^{\text{init}}$	0.934
$e_{\text{repr}}^{\text{R}}$	94	$e_{\text{min}}^{\text{R}}$	5	$e_{\text{init}}^{\text{W}}$	52.5
$a_{\text{repr}}^{\text{W}}$	63	$e_{\text{max}}^{\text{W}}$	100	$e_{\text{init}}^{\text{R}}$	52.5
$a_{\text{repr}}^{\text{R}}$	25	$e_{\text{max}}^{\text{R}}$	100	$g_{\text{grow}}^{\text{mult}}$	0.99957
$m_{\text{speed}}^{\text{W}}$	2.7	$e_{\text{move}}^{\text{W}}$	0.001	g_{max}	332
$m_{\text{speed}}^{\text{R}}$	2.2	$e_{\text{move}}^{\text{R}}$	0.001	δ_{rate}	1.3103
$n_{\text{init}}^{\text{W}}$	24	$e_{\text{cog}}^{\text{W}}$	0.01	r_{sight}^{init}	19
$n_{\text{init}}^{\text{R}}$	21	$e_{\text{cog}}^{\text{R}}$	0.01	k	50
$n_{\text{max}}^{\text{W}}$	100	$e_{\text{tick}}^{\text{W}}$	1		

Appendix B

VAE Implementation

This appendix contain relevant Python code used to train our VAE. The code can be seen in Listing B.1, B.2, B.3, B.4, and B.5.

```
1 """"
2     Variation Autoencoder model used for anomaly detection in A-life
3 
4     NOTE: This is an abstract class that serves as base for a VAE topology,
    it
5     should not be instantiated by itself. This is done in order to
    facilitate the
6     creation of new topoogies
7 
8     Created by: Everton Schumackers Soares on June 07, 2019
9     Based on: https://arxiv.org/pdf/1606.05908.pdf
10 
11 
    ---------------------------------------------------------------------------
12 
13     EXAMPLE: A new class should inherite this one as follows
14 
15         class Autoencoder(AbstractVAE):
16             def __init__(self, metainfo):
```

```
17                super(Autoencoder, self).__init__(metainfo)
18                self.__createInitNetwork__(metainfo.code_layer_size)
19                return
20
21            def __createEncoder__(self, x, builder):
22                x = slim.batch_norm(x, activation_fn=None)
23                builder.addConvLayer(name='encoder_1', num_filters=30,
    kernel_size=[11, 11])
24                return builder.buildFullyConvNet(debug=True)
25
26            def __createDecoder__(self, x, builder):
27                builder.addDeconvLayer(name='decoder_1', num_filters=1,
    kernel_size=[11, 11])
28                return builder.buildFullyConvNet(x, start_on='decoder_8',
    debug=True)
29
30
    ----------------------------------------------------------------------------
31 """
32
33 import tensorflow as tf
34 import numpy as np
35 import tensorflow.contrib.distributions as tds
36 import logging
37 import dill
38 import os
39
40 from abc import ABC, abstractmethod
41 from build_network_utils import FullyConnectedNetBuilder as FNNBuilder
42 from build_network_utils import ConvNetworkBuilder as CNNBuilder
43
44 class AbstractVAE(ABC):
45     #===================
```

```
46      # Public Properties
47      #====================
48
49      input_layer = None # placeholder tensor for input
50      input_width = None
51      input_height = None
52      input_channels = None # number of channels for the input
53      reguralizer_scaler = 0.0001 # scaler for L1 regularization
54      encoder = None # encoder layer
55      decoder = None # decoder layer
56      code = None
57      loss_layer = None # autoencoder's loss layer
58      train_operation = None
59      mu = 0
60      std = 0.1
61      z_mean_layer = None
62      z_stdev_layer = None
63      KL_weight = 1 # weight for the KL divergence term in the loss function
64      threshold = None # threshol for anomaly detection
65
66      #==============
67      # Constructor
68      #==============
69
70      def __init__(self, metainfo):
71          self.input_width = metainfo.target_width
72          self.input_height = metainfo.target_height
73          self.input_channels = metainfo.target_channels
74          self.KL_weight = 1
75
76          self.input_layer = tf.placeholder(tf.float32,
77            [None, metainfo.target_height, metainfo.target_width, metainfo.
    target_channels],
78            name='input_layer')
```

```
79          return
80
81      #=================
82      # Private Methods
83      #=================
84
85      def __createEncoder__(self, x, builder):
86          """
87              Abstract method to be defined by the child model
88
89              This functions must declare the topology of the VAE's encoder
90
91              Args:
92                  x := input tensor (tensor given as output by the shared
    decoder layers)
93                  builder := build_network_utils used to builde the encoder
94          """
95          pass
96
97      def __createDecoder__(self, x, builder):
98          """
99              Abstract method to be defined by the child model
100
101             This functions must declare the topology of the VAE's decoder
102
103             Args:
104                 x := input tensor (tensor given as output by the shared
    decoder layers)
105                 builder := build_network_utils used to builde the decoder
106         """
107         pass
108
109     def __createCodeLayer__(self, x, size_z):
110         """
```

```
111             Create code layer responsible for the variational portion of the
112             autoencoder.
113
114             Args:
115                 X := input tensor (tensor given as output by the shared
    decoder layers)
116                 size_z := (int) dimension of the latent variable z
117         """
118
119         input_shape = tf.shape(x)
120         builder = FNNBuilder(mu=self.mu, std=self.std)
121
122         logging.info('Code layer size: {}\n'.format(size_z))
123
124         # Code layer that parametrize P(z | X), represented by mu(x) and
    sigma(X)
125         self.z_mean_layer = builder.buildFullyConnectedLayers(
126             name='z_mean',
127             x=x,
128             sizes=[size_z],
129             activations={0:'relu'},
130         )
131
132         self.z_stdev_layer = builder.buildFullyConnectedLayers(
133             name='z_stdev',
134             x=x,
135             sizes=[size_z],
136             activations={0:'relu'},
137         )
138
139         # Samples a latent variable z from a normal distribution
    parametrized
140         # by mu(X) and sigma(X) using reparametrization
141         input_shape = tf.shape(x)
```

```
142         batch_size = input_shape[0]
143         z_sample = self.sampleOperation(batch_size, size_z)
144 
145         code = builder.buildFullyConnectedLayers(
146           name='code',
147           x=z_sample,
148           sizes=[size_z, int(np.prod(x.get_shape()[1:]))],
149           activations={0:'relu', 1:'relu'},
150             input_is_conv=False
151         )
152         return code
153 
154     def __createInitNetwork__(self, size_z):
155         builder = CNNBuilder(self.input_layer)
156 
157         # Creates layers
158         self.encoder = self.__createEncoder__(self.input_layer, builder)
159         self.code = self.__createCodeLayer__(self.encoder, size_z)
160         code = tf.reshape(self.code, tf.shape(self.encoder))
161         self.decoder = self.__createDecoder__(code, builder)
162 
163         # Creates Loss
164         self.loss_layer = tf.reduce_mean(tf.squared_difference(self.decoder,
     self.input_layer),
165           name='autoencoder_loss_layer')
166         self.loss_layer = tf.cast(self.loss_layer, tf.float64)
167 
168         # Creates prediction layer
169         self.prediction_layer = tf.cast(self.decoder, tf.uint8, name='
    prediction_layer')
170         return
171 
172     #=================
173     # Public Methods
```

```
174     #=================
175 
176     def regularizer(self):
177         l1 = sum(tf.reduce_sum(tf.square(var)) for var in tf.
    trainable_variables() if '_W' in var.name)
178         return self.reguralizer_scaler * tf.cast(l1, tf.float64)
179 
180     def KLLoss(self):
181         # Creates normal distribution varaible
182         normal_dist = tds.Normal(loc=0., scale=1.)
183 
184         # Creates N(mu(X), sigma(X))
185         z_dist = tds.Normal(loc=self.z_mean_layer, scale=self.z_stdev_layer)
186 
187         kl = tds.kl_divergence(z_dist, normal_dist)
188         kl_cast = tf.cast(
189             tf.reduce_mean(kl),
190             dtype = tf.float64,
191         )
192         return kl_cast
193 
194     def sampleOperation(self, batch_size, n):
195         """ Samples a latent variable z through reparametrization: it
    samples
196         a noise variable z from the normal distribution N(0, I) and compute
197         z = mu(X) + sigma(X)*epsilon
198 
199          Args:
200             batch_size := size of minibatch
201             n := dimension of latent variable z
202         """
203         shape = tf.stack([batch_size, n])
204 
205         epsilon = tf.random_normal(shape, 0, 1, dtype=tf.float32)
```

```
206         epsilon = tf.stop_gradient(epsilon)
207         z = self.z_mean_layer + (self.z_stdev_layer * epsilon)
208         return z
209 
210     def createTrainOperation(self, intial_learn_rate):
211         self.learning_rate = tf.placeholder(tf.float32, shape=[])
212         if self.train_operation is None:
213             train_loss = tf.reduce_mean(self.loss_layer + self.KL_weight *
    self.KLLoss() + self.regularizer())
214 
215             optimizer = tf.train.AdamOptimizer(self.learning_rate)
216             scope = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
217             grads_and_vars = optimizer.compute_gradients(train_loss, scope)
218             self.train_operation = optimizer.apply_gradients(grads_and_vars)
219         return
220 
221     def optimize(self, x, _, learn_rate):
222         sess = tf.get_default_session()
223         feed_dict = {
224             self.input_layer : x,
225             self.learning_rate : learn_rate
226         }
227         return sess.run(self.train_operation, feed_dict=feed_dict)
228 
229     def perform(self, x, _):
230         sess = tf.get_default_session()
231         feed_dict = {
232             self.input_layer : x,
233         }
234         return (sess.run(self.loss_layer, feed_dict=feed_dict), None)
235 
236     def computeErrors(self, x, _):
237         sess = tf.get_default_session()
238         feed_dict = {
```

```
239             self.input_layer : x,
240         }
241         return sess.run(self.loss_layer, feed_dict=feed_dict)
242 
243     def predict(self, x):
244         sess = tf.get_default_session()
245         feed_dict = {self.input_layer : x}
246         return sess.run(self.prediction_layer, feed_dict)
247 
248     def increase_KL_weight(self):
249         self.KL_weight = min(1.0, self.KL_weight+0.0001)
250 
251     def save(self, path):
252         """
253             Save model information into a file with name model_info.pickle.
254             The information consist of a dictionary with the target
    atributes.
255             The basic autoencoder only saves the anomaly detection
    threshold, but
256             a dictionary is still saved, since child models can save more
    info.
257 
258             Args:
259                 path: path for the target folder where mode_info.pickle will
     be saved
260         """
261 
262         model_info = {'threshold': self.threshold}
263         filename = os.path.join(path, 'model_info.pickle')
264 
265         with open(filename, 'wb') as model_file:
266             dill.dump(model_info, model_file, -1)
267         logging.info('Model info saved to: %s\n' % filename)
268 
```

```
269     def load(self, path):
270         """
271             Load model information from a file with name model_info.pickle.
272             The information consist of a dictionary with the target
    atributes.
273             The basice autoencoder only loads the anomaly detection
    threshold, but
274             a dictionary is laoded saved, since child models can save more
    info.
275 
276             Args:
277                 path: path for the target folder from where mode_info.pickle
     will
278                     be loaded
279         """
280 
281         filename = os.path.join(path, 'model_info.pickle')
282 
283         with open(filename, 'rb') as model_file:
284             model_info = dill.load(model_file, -1)
285 
286         self.threshold = model_info.get('threshold', None)
287         logging.info('Model info loaded from: %s\n' % filename)
```

Listing B.1: Abstract VAE class

```
1 """
2     This class has the basic deep learning functions necessary to train,
    evaluate,
3     save, load, and measure results in tensorflow independent of the type of
     Model
4     being used. As long as it follows the right interface illustrated in
5     abstract_variational_autoencoder.py
6 
7     Created by Everton Schumacker Soares
```

```
8     Updated on 07 June, 2019 by Everton Schumacker Soares
9  """
10
11 import tensorflow as tf
12 import numpy as np
13 import datetime
14 import time
15 import os.path
16 import matplotlib.pyplot as plt
17 import anomalyDetection as ad
18 import scipy.misc
19 import logging
20 import dill
21
22 from sklearn.utils import shuffle
23 from dataset_utils import make_pipeline_for_tfrecord as make_pipeline
24 from PerformanceMeasure import PerformanceMeasure
25
26 # Class containing all operations for deep learning in TensorFlow
27 class DlOperations:
28     metainfo = None
29     verboseFreq = None
30     verbose = None
31     config = None
32     session = None
33     saver = None
34
35     def __init__(self, metainfo, verbose=True, verboseFreq=100):
36         self.metainfo = metainfo
37         self.verbose = verbose
38         self.verboseFreq = verboseFreq
39
40         if metainfo.use_gpu:
41             config = tf.ConfigProto()
```

```
42            config.gpu_options.allow_growth = True
43            self.config = config
44        else:
45            config = tf.ConfigProto(device_count = {'GPU': 0})
46            self.config = config
47
48    # Resets current existin g graph
49    def reset(self):
50        tf.reset_default_graph()
51        return
52
53    def saveModel(self, path, model):
54        filename = os.path.join(path, 'model.ckpt')
55
56        if not os.path.exists(os.path.dirname(filename)):
57            os.makedirs(os.path.dirname(filename))
58
59        if self.saver is None:
60            self.saver = tf.train.Saver()
61
62        self.saver.save(self.session, filename)
63        model.save(path)
64
65    def loadModel(self, path, model):
66        filename = os.path.join(path, 'model.ckpt')
67
68        if self.saver is None:
69            self.saver = tf.train.Saver()
70
71        self.saver.restore(self.session, filename)
72        model.load(path) # Load trained model into the object model
73
74    # Need to be called before training or using the model
75    def startSession(self):
```

```
76          if not self.session is None:
77              self.closeSession()
78
79          sess = tf.InteractiveSession(config=self.config)
80          self.session = sess
81          return sess
82
83      def closeSession(self):
84          self.session.close()
85          self.session = None
86          return
87
88      # Creates the pipeline when using tfrecords
89      def getBatchIteratorWithTfRecords(self, batchSize, epochs,
    tfrecord_filenames, shuffle):
90          filenames, iterator = make_pipeline(batchSize, epochs, shuffle)
91          self.session.run(iterator.initializer, feed_dict={filenames:
    tfrecord_filenames})
92          x_batch, y_batch = iterator.get_next()
93          return x_batch, y_batch
94
95      # Helper function to compute number of samples in the dataset
96      def computeNumSamplesInTfRecord(self, tfrecord_filenames, shuffle=False)
    :
97          x_batch, _ = self.getBatchIteratorWithTfRecords(1, 1,
    tfrecord_filenames, shuffle=shuffle)
98          num_samples = 0
99
100         while True:
101             try:
102                 self.session.run(x_batch)
103                 num_samples += 1
104             except tf.errors.OutOfRangeError:
105                 break
```

```
106         return num_samples
107 
108     def trainNetworkWithTfRecordsOpt(self, model, num_tr_samples=None,
    num_val_samples=None):
109         """ Train network using tfrecords as inputs. The training is stoped
    once
110             the validation error platos or when the max number of epochs is
    reached.
111             A stepwise cooling schedule of the learning rate can be applied
    by setting
112             the scaler 0 < alpha < 1.
113 
114             NOTE: In this case self.epochs should be a large value, just to
115             prevent the training to run indefinetly, self.batchSize should
    be
116             the largest values supported by the GPU, and self.learningRate
117             should be initialy high.
118 
119             Argments:
120                 model := class containing tf computation graph to be train
121                 num_tr_samples := number of samples in the train set. If
    None,
122                     the number is computed; otherwise, fixed value is used
123                 num_val_samples := number of samples in the validation set.
    If
124                     None, the number is computed; otherwise, fixed value is
    used
125         """
126 
127         # Initialize variables
128         metainfo = self.metainfo
129         self.saver = tf.train.Saver()
130         start_time = time.time()
131         total_time = 0
```

```
132        model.createTrainOperation(metainfo.learn_rate)
133        curr_epoch = 1
134        global_step = 0
135        samples_in_epoch = 0
136        UP = np.array([False] * metainfo.s_size) # UP stop criteria
137        val_losses = np.zeros(metainfo.s_size * metainfo.k_size) #
    validation over s strips of size k
138        delta_val_losses = np.array([np.Inf] * metainfo. k_size)
139        learn_rate = metainfo.learn_rate
140
141        # KL cost anealling starts with weight 0 for the KL diverge loss
    term
142        if metainfo.KL_anelling:
143            model.KL_weight = 0
144
145        # Compute size of train set
146        if num_tr_samples is None:
147            num_tr_samples = self.computeNumSamplesInTfRecord(metainfo.
    train_files)
148
149        # Compute size of validation set
150        if num_val_samples is None:
151            num_val_samples = self.computeNumSamplesInTfRecord(metainfo.
    val_files)
152
153        # Creates batch iterator for train set
154        x_train_batch, y_train_batch = self.getBatchIteratorWithTfRecords(
155            metainfo.batch_size, metainfo.epochs, metainfo.train_files,
    shuffle=True)
156
157        # Creates batch iterator for validation set
158        x_val_batch, y_val_batch = self.getBatchIteratorWithTfRecords(
159            1, metainfo.epochs, metainfo.val_files,shuffle=True)
160
```

```
161         # Initialize tf computational graph
162         self.session.run(tf.global_variables_initializer())
163 
164         # Continue while the validation loss didn't go up for s strips, nor
    the
165         # validation platos (difference between epochs is always less than
    epsilon)
166         while (not all(UP)) and (not all(delta_val_losses < metainfo.epsilon
    )):
167             try:
168                 # Get next batch
169                 x_train, y_train = self.session.run([x_train_batch,
    y_train_batch])
170 
171                 # Optimize model with current batch
172                 model.optimize(x_train, y_train, learn_rate)
173 
174                 # Time passed so far since the beggining
175                 total_time = time.time() - start_time
176 
177                 if self.verbose and global_step % self.verboseFreq == 0:
178                     loss, acc = model.perform(x_train, y_train)
179 
180                 if samples_in_epoch < num_tr_samples:
181                     samples_in_epoch += len(x_train)
182                 else:
183                     # Cool learn rate
184                     if curr_epoch % metainfo.lr_freq == 0:
185                         learn_rate = learn_rate * metainfo.lr_decay
186 
187                     # compute validation error for previous epoch
188                     loss, _ = self.computeMeanValidationPerform(
189                         model,
190                         x_val_batch,
```

```
191                         y_val_batch,
192                         num_val_samples,
193                         1
194                     )
195 
196                     # Compute UP criteria for early stop
197                     val_losses[:-1] = val_losses[1:]
198                     val_losses[-1] = loss
199                     delta_val_losses = np.abs(np.subtract(val_losses[-
    metainfo.k_size:], val_losses[-metainfo.k_size-1:-1]))
200 
201                     for j in range(0, metainfo.s_size):
202                         UP[j] = val_losses[(j+1)*metainfo.k_size - 1] >
    val_losses[(j+1)*metainfo.k_size - metainfo.k_size]
203 
204                     # new epoch
205                     samples_in_epoch = 0
206                     curr_epoch += 1
207 
208                 global_step += 1
209 
210                 if metainfo.KL_anelling:
211                     model.increase_KL_weight()
212             except tf.errors.OutOfRangeError:
213                 break
214         return
215 
216     def computeMeanValidationPerform(self, model, x_batch, y_batch,
    num_samples, batch_size):
217         """ Computes the mean performance (loss and accuracy) on the
    validation set
218 
219             Arguments:
220                 model := class containing the tf computational graph
```

```
221                 x_batch := batch iterator for the samples in validation
222                 y_batch := batch iterator for the labels in validation
223                 num_samples := size of validation set
224                 batch_size := batch size for the validation set
225         """
226         total_loss = 0
227         total_acc = 0
228 
229         for i in range(0, num_samples, batch_size):
230             x_val, y_val = self.session.run([x_batch, y_batch])
231             loss, acc = model.perform(x_val, y_val)
232             total_loss = total_loss + (batch_size * loss)
233 
234             if acc is not None:
235                 total_acc = total_acc + (batch_size * acc)
236 
237         mean_loss = total_loss/num_samples
238 
239         mean_acc = None
240         if total_acc is not None:
241             mean_acc = total_acc/num_samples
242 
243         return (mean_loss, mean_acc)
244 
245     # Run trained model and returnes the accuracy, and majority class
    accuracy
246     def evaluateNetworkWithTfRecord(self, model, tfrecord_filenames, shuffle
    =True):
247         total_rights = 0
248         num_samples = 0
249         count_classes = {}
250 
251         x_batch, y_batch = self.getBatchIteratorWithTfRecords(1, 1,
    tfrecord_filenames, shuffle)
```

```
252
253         while True:
254             try:
255                 x_test, y_test = self.session.run([x_batch, y_batch])
256                 _, right = model.perform(x_test, y_test)
257                 total_rights += right
258                 num_samples += 1
259                 key = y_test[0]
260                 count_classes[key] = count_classes.get(key, 0) + 1
261             except tf.errors.OutOfRangeError:
262                 break
263
264         mc = np.max(np.array([x for x in count_classes.values()]))
265         return total_rights / num_samples, mc/num_samples
266
267     # computes reconstruction errors
268     def computeErrors(self, model, tfrecord_filenames, shuffle=False,
    log_file=None):
269         errors_for_labels = {}
270         index = 0
271
272         x_batch, y_batch = self.getBatchIteratorWithTfRecords(1, 1,
    tfrecord_filenames, shuffle)
273
274         log = None
275         if log_file is not None:
276             log = open(log_file, 'w')
277
278         while True:
279             try:
280                 x, y = self.session.run([x_batch, y_batch])
281                 error = model.computeErrors(x, y)
282                 errors_for_labels[y[0]] = errors_for_labels.get(y[0], [])
283                 errors_for_labels[y[0]].append(error)
```

```
284
285                 if log is not None:
286                     log.write('image_{}_{}: MSE = {}\n'.format(y[0], index,
    error))
287                 index += 1
288             except tf.errors.OutOfRangeError:
289                 break
290
291         if log is not None:
292             log.close()
293         return errors_for_labels
294
295     def drawImage(self, I, filename, w, h, gray):
296         if I.shape[2] == 1:
297             I = np.resize(I, (h, w))
298         scipy.misc.imsave(filename, I)
299         return
300
301     # Save the autoencoder's reconstructed images
302     def reconstrucImages(self, model, tfrecord_filenames, folder, gray=False
    , shuffle=False):
303         x_batch, y_batch = self.getBatchIteratorWithTfRecords(1, 1,
    tfrecord_filenames, shuffle)
304         index = 0
305
306         if not os.path.exists(folder):
307             os.makedirs(folder)
308
309         while True:
310             try:
311                 x, y = self.session.run([x_batch, y_batch])
312                 image = model.predict(x)
313
314                 if model.threshold is None:
```

```
315                     filename = '{}/image_{}_{}.png'.format(folder, y[0],
    index)
316                 else:
317                     loss = model.computeErrors(x, y)
318                     is_anom = ad.isAnomally(loss, model.threshold)
319                     filename = '{}/image_{}_{}_anom_{}.png'.format(folder, y
    [0], index, is_anom)
320                 self.drawImage(image[0], filename, self.metainfo.
    target_width, self.metainfo.target_height, gray)
321                 index += 1
322             except tf.errors.OutOfRangeError:
323                 break
324
325     def tfrecordToImages(self, tfrecord_filenames, folder, gray=False,
    shuffle=False):
326         x_batch, y_batch = self.getBatchIteratorWithTfRecords(1, 1,
    tfrecord_filenames, shuffle)
327         index = 0
328
329         if not os.path.exists(folder):
330             os.makedirs(folder)
331
332         while True:
333             try:
334                 x, y = self.session.run([x_batch, y_batch])
335                 filename = '{}/image_{}_{}.png'.format(folder, y[0], index)
336                 self.drawImage(x[0], filename, self.metainfo.target_width,
    self.metainfo.target_height, gray)
337                 index += 1
338             except tf.errors.OutOfRangeError:
339                 break
```

Listing B.2: Deep Learning Operations

```
1 def thesholdIQR(points, scaler=1.5):
```

```
2     """
3     Computes the threshold for outilers given the known data points using
     the
4     interquartile rule
5
6     points: series of known and normal points
7     returns: lower_threshold such that anything bellow it is an anomally,
8     upper_threshold such that anything above it is an anomaly
9     """
10
11    if scaler is None:
12      return 0, 0
13
14    Q1 = np.percentile(points, 25) # computes 25% percentile
15    Q3 = np.percentile(points, 75) # computes 75% percentile
16    IQR = Q3 - Q1 # interquartile rule
17    lower_threshold = Q1 - scaler*IQR
18    upper_threshold = Q3 + scaler*IQR
19    return lower_threshold, upper_threshold
20
```

Listing B.3: Interquartile rule

```
1   def compute_thresshold(operations, model, max_scaler=10):
2     """ Compute best threshold given the reconstruction error (anomaly
     scores)
3
4       Note: We are not considering the outiliers with abnormally low values,
      because
5       this implies a smaller reconstruction error for the anomalies, which
     contradicts
6       our assumption for the autoencoders
7     """
8
9     metainfo = operations.metainfo
```

```
10     best_thd = 0
11     best_measure = PerformanceMeasure()
12     best_rank = -1
13 
14     train_anom_score = operations.computeErrors(model, metainfo.train_files)
15     val_anom_score = operations.computeErrors(model, metainfo.val_files)
16 
17     for scaler in np.arange(0, max_scaler+1, 0.5):
18       _, upper_threshold = ad.thesholdIQR(train_anom_score.get(ad.Label.
    NORMAL.value, []), scaler=scaler)
19 
20       measure = performDetectionWithThreshold(val_anom_score.get(ad.Label.
    NORMAL.value, []),
21       val_anom_score.get(ad.Label.SYNTHETIC_ANOMALY.value, []),
    upper_threshold)
22 
23       if measure.accuracy > best_measure.accuracy:
24         best_measure = measure
25         best_thd = upper_threshold
26     return best_thd, best_measure
```

Listing B.4: Compute Threshold

```
1 """"
2     Variation Autoencoder model used for anomaly detection in A-life in the
    paper
3     publishe on IEEE COG 2019. This is the refactored
    variational_autoencoder2.py
4 
5     Created by: Everton Schumackers Soares
6     Based on: https://arxiv.org/pdf/1606.05908.pdf
7 """
8 
9 import tensorflow as tf
10 import numpy as np
```

```
11 import tensorflow.contrib.slim as slim
12 import tensorflow.contrib.distributions as tds
13
14 from build_network_utils import FullyConnectedNetBuilder as FNNBuilder
15 from build_network_utils import ConvNetworkBuilder as CNNBuilder
16 from abstract_variational_autoencoder import AbstractVAE
17
18 class Autoencoder(AbstractVAE):
19     #==============
20     # Constructor
21     #==============
22
23     def __init__(self, metainfo):
24         super(Autoencoder, self).__init__(metainfo)
25         self.__createInitNetwork__(metainfo.code_layer_size)
26         return
27
28
29     #=================
30     # Private Methods
31     #=================
32
33     def __createEncoder__(self, x, builder):
34         """
35             Overload abstract method from AbstractVAE
36
37             This function declares the topology of the VAE's encoder
38
39             Args:
40                 x := input tensor (tensor given as output by the shared
    decoder layers)
41                 builder := build_network_utils used to builde the encoder
42         """
43
```

```
44         x = slim.batch_norm(x, activation_fn=None)
45         builder.addConvLayer(name='encoder_1', num_filters=30, kernel_size
    =[11, 11])
46         builder.addConvLayer(name='encoder_2', num_filters=20, kernel_size
    =[7, 7])
47         builder.addMaxPoolLayer(ksize=[3, 3], strides=(2, 2))
48         builder.addConvLayer(name='encoder_3', num_filters=15, kernel_size
    =[5, 5])
49         builder.addConvLayer(name='encoder_4', num_filters=15, kernel_size
    =[3, 3], padd='VALID')
50         builder.addConvLayer(name='encoder_5', num_filters=15, kernel_size
    =[3, 3], padd='VALID')
51         builder.addConvLayer(name='encoder_6', num_filters=15, kernel_size
    =[2, 2], padd='VALID')
52         builder.addConvLayer(name='encoder_7', num_filters=10, kernel_size
    =[2, 2], padd='VALID')
53         builder.addConvLayer(name='encoder_8', num_filters=10, kernel_size
    =[2, 2])
54         builder.addMaxPoolLayer(ksize=[3, 3], strides=(2, 2))
55         return builder.buildFullyConvNet(debug=True)
56 
57     def __createDecoder__(self, x, builder):
58         """
59             Overload abstract method from AbstractVAE
60 
61             This function declares the topology of the VAE's encoder
62 
63             Args:
64                 x := input tensor (tensor given as output by the shared
    decoder layers)
65                 builder := build_network_utils used to builde the encoder
66         """
67 
```

```
68        builder.addDeconvLayer(name='decoder_8', num_filters=10, kernel_size
    =[3, 3], padd='VALID', strides=[2,2])
69        builder.addDeconvLayer(name='decoder_7', num_filters=15, kernel_size
    =[2, 2], padd='VALID')
70        builder.addDeconvLayer(name='decoder_6', num_filters=15, kernel_size
    =[2, 2], padd='VALID')
71        builder.addDeconvLayer(name='decoder_5', num_filters=15, kernel_size
    =[3, 3], strides=[2, 2], padd='VALID')
72        builder.addDeconvLayer(name='decoder_4', num_filters=20, kernel_size
    =[3, 3], padd='VALID')
73        builder.addDeconvLayer(name='decoder_3', num_filters=20, kernel_size
    =[4, 4], padd='VALID')
74        builder.addDeconvLayer(name='decoder_2', num_filters=30, kernel_size
    =[7, 7])
75        builder.addDeconvLayer(name='decoder_1', num_filters=1, kernel_size
    =[11, 11])
76        return builder.buildFullyConvNet(x, start_on='decoder_8', debug=True
    )
```

Listing B.5: VAE topology

Appendix C

Project Hastur: Details

This appendix details the playstyle (Appendix C.1) and parameters (Appendix C.2) used when collecting data from *Project Hastur*.

C.1 Playstyle

Project Hastur's enemy population (i.e., Proteans) may evolve several types of behaviours depending on the player's playstyle. To reduce variance we attempt to fix our playstyle as follows:

1. towers can only be purchased and placed in the map during the safety period (i.e., first seconds of every generation when no Protean spawns);
2. during the rest of the generation, the player can only upgrade or restore the existing towers;
3. each upgraded and non-upgraded tower must be placed around the same position on every run.

Our playstyle also follows additional rules for placing the energy matter converters, since they determine which regions of the map can be used to place new turrets and converters. They can only be placed in specific regions following the order:

1. the game starts on generation 0 with the main tower only (Figure C.1);
2. at the beginning generation 0 until end of generation 1, an energy matter converter must be placed on top of the main base (Figure C.2);
3. from generation 2 to 4, a new energy matter converter must be placed to the left of the main base (Figure C.3);
4. from generation 5 to 10, an energy matter converter is placed in the top-center of the screen to protect the civilians' base (Figure C.4);
5. from generation 11 to 16, a new energy matter converter should be placed to protect the bridge (Figure C.5);
6. around generation 17 to 20, the last energy matter converter should be placed to protect the other civilians' base in the top-left of the screen (Figure C.6);

Additionally, at the begging of each generation, the player can also place turrets in the allowed spaces. As already mentioned, the player can upgrade or heal turrets at any time as long as they have enough biomatter.

C.2 Experiment Mode

The data used in our experiments was collected using *Project Hastur*'s experiment mode (Figure C.7). The parameters used in this mode are listed bellow. The description of the parameters was transcribed from the game interface (Polymorphic Games, 2019):

- Map is the map of the experiment;
- Passive Upgrages : Kinectic are the upgrades for Kinectic turrets;
- Passive Upgrages : Fire are the upgrades for Fire turrets;
- Passive Upgrages : Ice are the upgrades for Ice turrets;

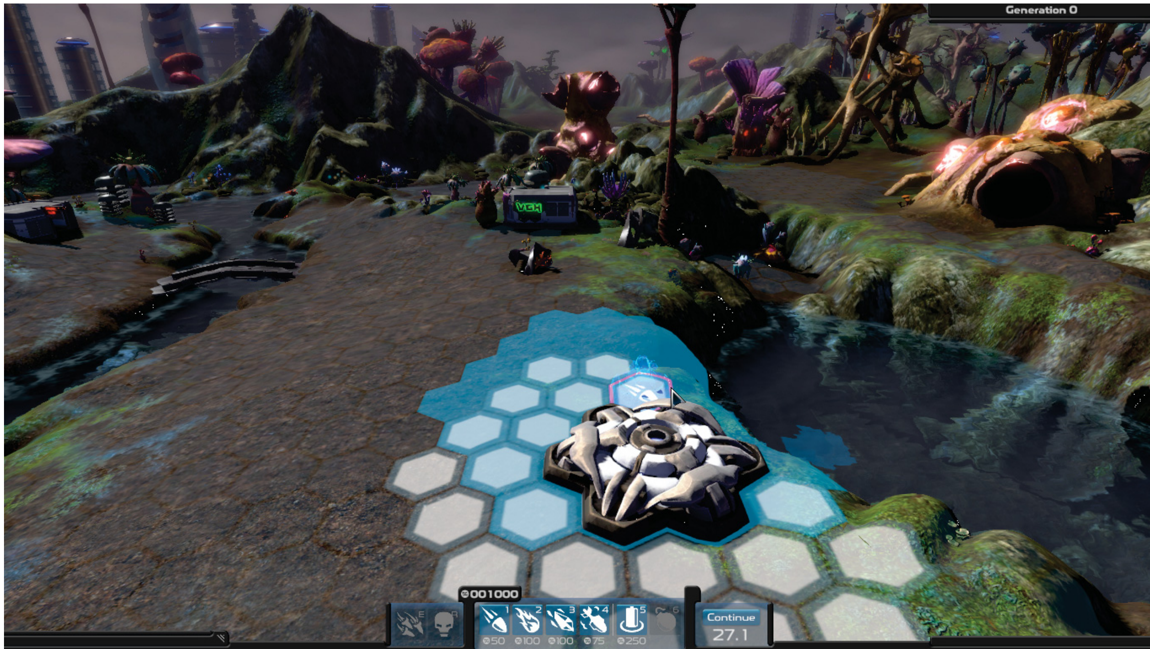

Figure C.1: *Project Hastur*: beginning of experiment mode, before the player place any turret or tower.

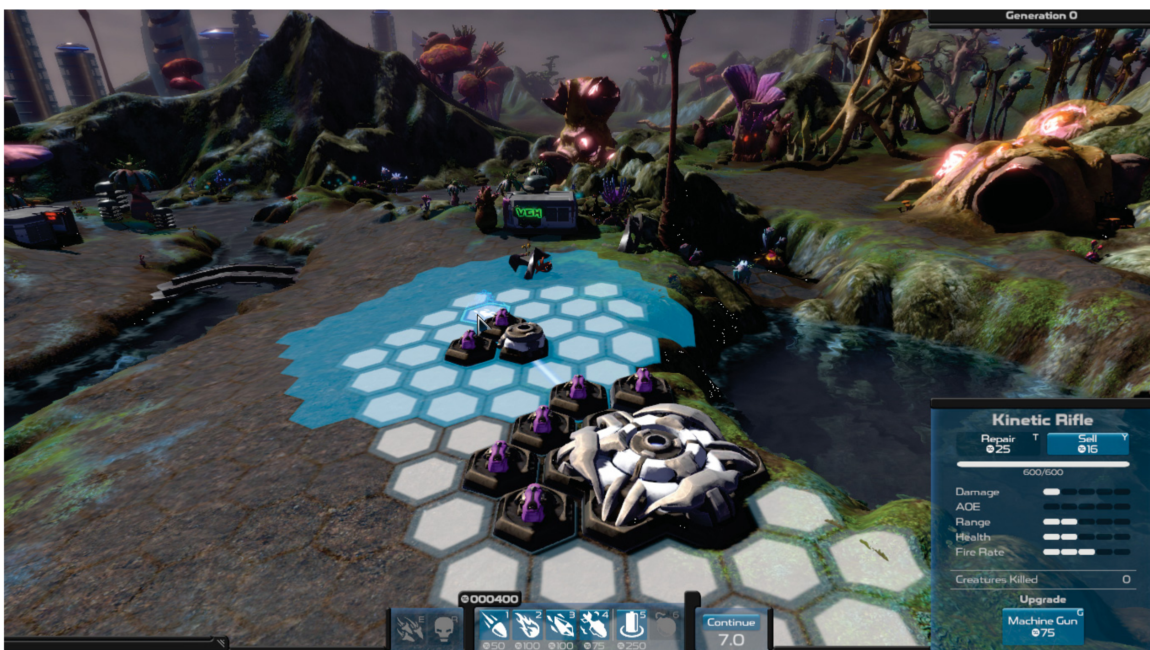

Figure C.2: *Project Hastur*: turret and tower configuration on Generation 0.

Figure C.3: *Project Hastur*: turret and tower configuration on Generation 2.

Figure C.4: *Project Hastur*: turret and tower configuration on Generation 6.

Figure C.5: *Project Hastur*: turret and tower configuration on Generation 11.

Figure C.6: *Project Hastur*: turret and tower configuration on Generation 17.

- Passive Upgrages : Mortar are the upgrades for Mortar turrets;
- Passive Upgrages : Airstrike are the upgrades for Airstrike;
- Passive Upgrages : Robot are the upgrades for Robot;
- Use Subpopulations is "When subpopulations are used, Proteans only reproduce with Proteans from the same burrow. Otherwise, any Protean in the population can reproduce with any other Protean. Either way, both fitness functions can be used.";
- Use Tower Fitness is "Creatures are selected based on how much damage they did to towers. If you use two fitness functions, half of the population will be selected using each. If you use subpopulations,half of each burrow will be selected using each. If there are no fitness functions, there will be no selection and parents will be chosen at random.";
- Use Base Fitness is "Creatures are selected based on getting close to the base and damaging it. If you use two fitness functions, half of the population will be selected using each. If you use subpopulations,half of each burrow will be selected using each. If there are no fitness functions, there will be no selection and parents will be chosen at random.";
- Population Size is "The number of Proteans spawned per generation.";
- Tournament Size is "The size of the selection tournament, calculated as a proportion of the population size.";
- Mutation Size is "The standard deviation on the Gaussian curve mutations are pulled from.";
- Mutation Rate is "Percentage change that any given gene will be mutated.";
- Starting Genetic Range is "The range the genes on first generation of creatures fall in. This is symmetrical. If you set a value of 10, te range will be -10 to 10. Will not be used if load genomes is turned on";

- Environmental Noise - Swimming is “Noise added to the genetic value for swimming. A higher value means that creatures are more likely yo develop swimming more quickly.”;
- Environmental Noise - Jumping is “Noise added to the genetic value for jumping. A higher value means that creatures are more likely yo develop jumping more quickly.”;
- Invincible is “Turrets and EMCs are invulnerable. Proteans can attack them, but they will never die.”;
- Reset Experiment at a Generation is “Immediately restarts when a certain generation is reached. Best used with invincibility turned on.”;
- Generation to Reset At is “Only available if "Reset Experiment at a Generation" is checked. The experiment will restart at generation 0 when the specific generation is reached.”;
- Restart on Loss is “Instead of bringing up the game over menu when you lose, the game will immediatly start over on generation 0 with a new population.”;
- Use People is “Specifies whether the people on the map should be turned off.”;
- Starting Biomatter is “Specifies the amount of biomatter you should have at the start of the game.”;
- Load Genome is “Load a saved population. The initial population in you experiment will be made of creatures chosen randomly from the saved population and mutated. Refresh the list if you externally change the available genomes through the "Manage Genomes" button.”;
- Load Turret Configuration is “Loads a saved turret configuration. Clicking "Create Turret Configuration" bellow will take precedence and the selected turret configuration will not be loaded. Refresh the list if you externally change the available configurations through the "Manage Turret Configurations" button.”;

- Manage Genomes is "Open the folder on your computer where genomes are kept to allow you to rename and delete them.";
- Manage Turret Configurations is "Open the folder on your computer where configurations are kept to allow you to rename and delete them.";
- Create Turret Configuration is "Loads the map without any creatures to allow you to place turrets. Press the "Save Configuration" button and the map will load with your saved turrets and the experiment will begin.".

The actual values used in our experiments are in Table C.1.

Figure C.7: Parameters for *Project Hastur* experiment mode.

Table C.1: Values for the parameters on *Project Hastur* experiment mode.

Parameter	Value
Map	"Crater Mountain"
Passive Upgrages : Kinectic	0 (basic turret)
Passive Upgrages : Fire	0 (basic turret)
Passive Upgrages : Ice	0 (basic turret)
Passive Upgrages : Mortar	0 (basic turret)
Passive Upgrages : Airstrike	0 (basic turret)
Passive Upgrages : Robot	0 (basic turret)
Use Subpopulations	true
Use Tower Fitness	true
Use Base Fitness	true
Population Size	100
Tournament Size	0.2
Mutation Size	0.6
Mutation Rate	0.4
Starting Genetic Range	0.1
Environmental Noise - Swimming	0.3
Environmental Noise - Jumping	0.3
Invincible	false
Reset Experiment at a Generation	true
Generation to Reset At	20
Restart on Loss	true
Use People	true
Starting Biomatter	1000
Load Genomes	None
Load Turret Configuration	None
Manage Genomes	not clicked
Manage Turret Configurations	not clicked
Create Turret Configuration	not clicked