University of Southampton

Faculty of Engineering and Physical Sciences

Electronics and Computer Science

Extending Autonomous Skill Discovery with Recurrent Neural Networks

by

Michael Kai Lawrence

September 2020

Supervisor: Dr Danesh Tarapore
Second Examiner: Professor Les Carr

A dissertation submitted in partial fulfilment of the degree of MSC

University of Southampton

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
ELECTRONICS AND COMPUTER SCIENCE

Master of Science

by Michael Kai Lawrence

"How do robots learn" This is a question that the fields of Bio-Inspired AI and Behaviour Based Robotics have tried to answer for a long time. Though there are many answers to the question, they all come with caveats. In Cully (2019), the author answers the specific problem of autonomous learning by proposing a universal algorithm for robotic skill discovery without user input. They do this by exploiting an autoencoder's ability to compress large scale sensory information into low dimensional space that can be explored using evolutionary algorithms. This report covers an extension to that work by adding on a recurrent neural network with the hopes that the system will train to account for time dependent information.

Acknowledgements

Academically I must first give my thanks to my supervisor Dr Danesh Tarapore who I will once again congratulate on becoming a parent. I also must say a big thanks to Dr David Bossens for his help and guidance over the latter half of this project. Finally thank you to Dr Antoine Cully for answering my questions about his research. In retrospect the questions I asked must have seemed simplistic but your answers acted as helpful insight. I greatly appreciate the help of all of you.

Thank you to my parents and to my friends for keeping me sane over the last few hectic months. For those who instead *drove me insane* (you know who you are) remember, I know where you live.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

Contents

List of Figures

Listings

Abbreviations:
ABR - Automatic Behavioural Repertoire
Adagrad - Adaptive Gradient Algorithm
AURORA - AUtonomous RObots that Realize their Abilities
AURORA-AE - AURORA with Autoencoder as dimension reduction algorithm
BD - Behavioural Descriptor
BR - Behavioural Repertoire
GRU - Gated Recurrent Unit
HBR - Hierarchical Behavioural Repertoire
KLC - Kullback-Leibler Coverage
LSA - Latent Semantic Analysis
LSTM - Long Short Term Memory
MAP-Elites - Multi-dimensional Archive of Phenotypic Elites
NSGC - Novelty Search with Global Competition
NSLC - Novelty Search with Local Competition
PCA - Principal Component Analysis
QD - Quality Diversity
ReLU - Rectified Linear Unit
RMSE - Root Mean Squared Error
RNN - Recurrent Neural Network
SVD - Singular Value Decomposition
VAE - Variational Autoencoder
WAE - Wasserstein Autoencoder

Chapter 1

Introduction

The realm of autonomous robotics has advanced and benefited due to the widespread application of machine and deep learning techniques. Many of these advancements are covered in texts such as Andrew (1999) and Floreano and Mattiussi (2008). However, how these techniques are applied is often quite different in comparison to other fields. This is due to a core difference between the "big-data" problems in most fields and the "micro-data" problems in robotics. In big-data problems, modern advancements usually focus on reducing the dimensionality of the problem or in creating more efficient algorithms for processing the large amount of data. On the other hand, "micro-data learning" as described in Chatzilygeroudis et al. (2018), refers to situations and problems where there is very little to no data and the aim of the learning algorithm is to efficiently learn from the handful of information that it can obtain. This is similar to data-efficient reinforcement learning, but instead of efficiency being a measure of quantity of data against complexity of task, micro-data learning refers to bounded time efficiency.

The author of Cully (2019) describes a variation of one such algorithm. More accurately a type of ABR algorithm, AURORA (AUtonomous RObots that Realize their Abilities) combines a Quality Diversity algorithm with Behavioural Repertoire techniques to create an algorithm that autonomously discovers new behaviours. AURORA takes the sensory information generated by a repertoire of controllers (the vector of values that controls the robot) and explores the reduced dimensional space. The performance of this has been found to be effective at generating a wide range of behaviours. A specific feature of AURORA is that it flattens the sensory information into a one-dimensional vector even though it records the data as a time series. Robotic behaviours are grounded in the real world where time is a factor and thus it is sensible to explore the variation in AURORA's effectiveness when it considers time dependent sensory information. This can be done by extending the autoencoder component of AURORA with a recurrent neural network.

Initially, the aim of this project was to build off of the original AURORA code and extend far more in the number of environments the algorithm was applied to as well as test the viability of using VAEs to generate the behavioural descriptors. However, whilst making progress in modifying the code an OS update (to Apple's Catalina) was installed on the environment the project had so far been programmed in. This meant it was no longer possible to use the version of Singularity to compile and run the original code. There were three alternatives: create a virtual Linux environment run Singularity in using Vagrant Box, use the Beta version of Singularity for MacOS, or re-implement the AURORA algorithm. The first alternative was limited by the fact that only certain older versions of Singularity are currently supported (the full list of supported version can be found here: `https://app.vagrantup.com/singularityware`). The second method was limited by the fact that the Beta version of Singularity 3.6 for MacOS (`https://sylabs.io/singularity-desktop-macos/#:~:text=Singularity%20itself%20%2D%20a%20native%20port,interface%20between%20Singularity%20and%20macOS`) either has significant flaws or, far more likely, is not currently suitable for running on Catalina. Hence, the only option left was to re-implement the algorithm. The decision to do so was made on the 31st July 2020 (the project start date was the 22nd June and the end date was the 17th September). Fortunately, it was possible to salvage the Tensorflow code from the original implementation used to create the autoencoder.

Chapter 2

Background

2.1 Evolutionary Algorithms

2.1.1 Basics

Classic evolutionary algorithms consist of:

- a population, a set of individuals
- a fitness function, a method for evaluating the fitness of the members of the population
- a selection process, a method that chooses which members of the population will reproduce
- a reproduction process, a method or methods for generating new members of the population

The aims of an evolutionary algorithm are to iteratively add fit individuals to the population, remove less fit individuals from the population, and generate new and hopefully fitter individuals by selecting fit parents that will generate offspring. The population acts as a container for all the individuals. The fitness function is a method that evaluates an individual and assigns them a value by some arbitrary metric, such as the distance from a preset target or a tendency to produce offspring. This assigned value is the fitness of the individual. In this report we focus on the other two components. As they relate to specific stages, we will refer to these components as the selection stage and the reproduction stage.

2.1.2 Selection

Many selection methods can be understood using the analogy of a roulette wheel. The most basic method of selection in evolutionary algorithms is random selection. We can imagine this as giving each member of the population a number, giving them an equal fraction of the wheel, and spinning the roulette wheel to see where it will land and who will get to reproduce. As with many basic methods it is flawed. For example, in nature not all individuals have the same likelihood to reproduce. Multiple methods exist to allow for fitness to be taken into account during selection. Some examples include tournament selection and fitness proportionate selection. The latter can be thought of as giving individuals with a greater value of fitness a greater proportion of the roulette wheel. This changes the probability of selection by making it biased towards fitter individuals of the population, which is indeed what we want from evolutionary algorithms.

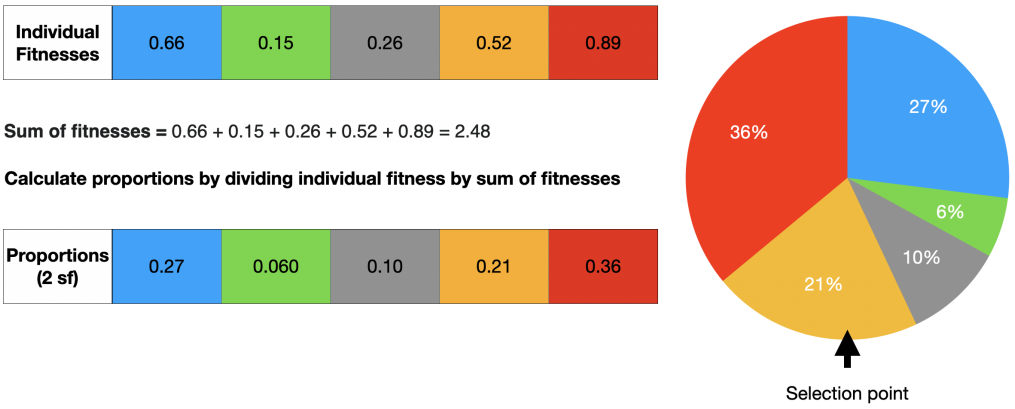

FIGURE 2.1: The method for Fitness Proportionate Selection. The figurative wheel is constructed from fitness of the individuals in the population.

In pure fitness proportionate selection if there is one individual that is significantly more fit than the rest of the population then the rest of the individuals may not get chosen at all, resulting in one parent siring all of the offspring. While this may not be a significant issue, it can result in stagnation if the population is too small and there is no mutation. This would eventually result in an upper limit to the fitness of the population that does not represent the true maximum possible fitness. To make the next generation more statistically representative compared to the previous, we must ensure that observed selection frequencies are similar to the expected frequencies. This is possible with stochastic universal sampling.

To illustrate the differences between the two methods we observe Figure 2.1. Notice that in fitness proportionate selection, when the wheel is constructed a selector is set in position. In truth a random number between one and zero is generated, and when the cumulative proportion is equal to or greater than this random number, the individual at

that index is chosen. In comparison, with stochastic universal sampling we have many selectors at different points. For example, we could space the selectors equally around the wheel. When the random number is generated an offset is introduced, and then the cumulative process repeats as before. The offset would be different for each selector.

But what do we do if there is no fitness function? Whilst it is possible to simply use random selection to choose any individual of the population with equal probability, this does not guarantee efficient search. Cully and Demiris (2017) considered such algorithms. The authors evaluated multiple existing alternative methods to random selection, before introducing novel alternatives and metrics that make the selection more biased. These included score proportionate selection, population-based selection and Pareto-based selection. Score proportionate selection is identical to fitness proportionate selection except for the metric used to calculate the proportions. Though the Search Method section details this further, simplistically it is a measure of an individual's ability to add to the population. This new arbitrary metric is called the "curiosity score". If the parent succeeds in generating new offspring that can be added to the population, the parent's curiosity score is increased. If the offspring cannot be added to the population then the parent's curiosity score is decreased. In modern evolutionary algorithms, where the aim has been to reduce user input as much as possible, metrics like the curiosity score are extremely beneficial. This is likely why curiosity score was incorporated into the later work of one of the authors (Cully (2019)).

2.1.3 Reproduction

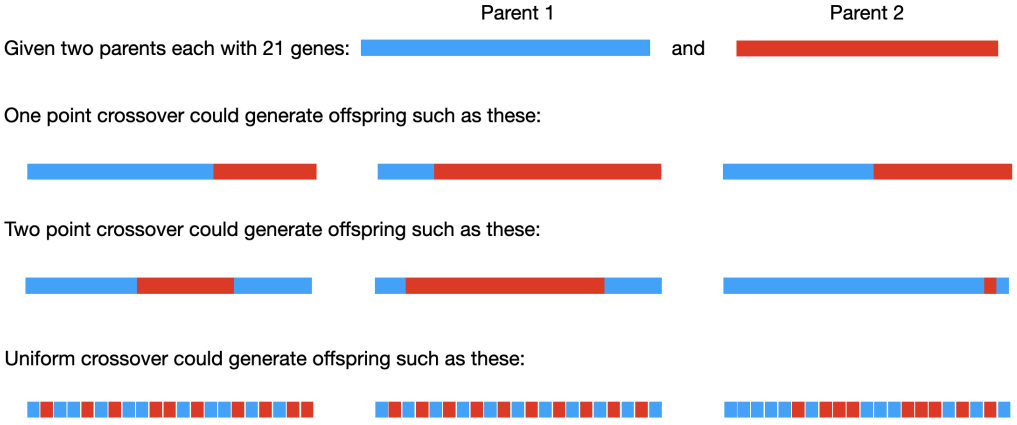

FIGURE 2.2: An example of some of the offspring that can be generated from two parents and different types of crossover.

There are multiple ways of reproducing from the chosen parent or parents in evolutionary algorithms. Overall they usually fit under one of two groups: crossover or mutation. Crossover consists of taking the genotypes of the parents and choosing one or the other's specific gene to be applied to the offspring. There are multiple variants such as one point

crossover (copies one parent up to a certain point after which it then copies from the other), two point crossover (copies one parent up to a certain point, then copies from the other parent up to a certain point, then goes back to copying from the first parent) and uniform crossover (each gene is chosen from either parent with equal probability). Figure 2.2 illustrates how crossover can create new offspring.

The result with crossover is that if fitter individuals are selected from the population to generate the offspring, and these parents have similar genes, then the offspring will maintain the minimum fitness of the parents. This is because if fit parents have similar genes it doesn't matter which parent the gene is chosen from, and the offspring will eventually converge towards the fittest solution (if one exists). This idea led to the building block hypothesis as covered and critiqued in Burjorjee (2008). In evolutionary algorithm terms, a building block is a low order schema with a short defining length with a fitness above the average. To help explain schema and defining length, observe Figure 2.3. The definition of schema is the set of genotypes that contain that subset of genes. The defining length is the maximum distance between defining symbols of that schema. The building block hypothesis is the idea that an evolutionary or genetic algorithm implementing crossover will identify and recombine these building blocks and converge towards a superior solution.

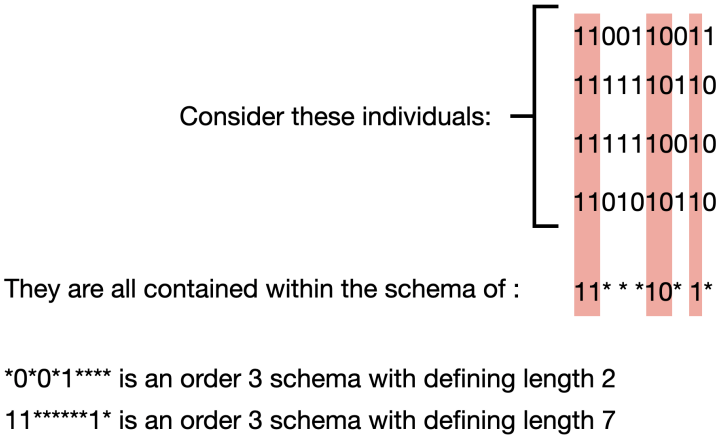

FIGURE 2.3: How to find the schema of a set of bit strings and examples of short order schema with differing defining lengths

The other major method of reproducing in evolutionary and genetic algorithms is mutation. In this project we focus primarily on mutation. Mutation consists of taking an individual from the population and changing the genes in its genotype with a set probability. For example, classic methods include flipping a bit of a individual composing of a bit-string, or choosing a random value in an upper and lower bound to set at the gene of the genotype of the new offspring. Modern methods can be far more complex such as with Gaussian and polynomial mutation. Deb and Deb (2014) goes into detail about how these two work and as polynomial mutation was used in this project it is covered in depth in the Search Method section.

2.2 Quality Diversity Optimisation

Taking inspiration from nature is a principle used across many technical fields. However, we find that while evolution can result in many fantastic developments such as flight in birds or intelligence in humans, we find that in computation the results are less ambitious and more modest. Previously it was believed that this was due how evolutionary algorithms were used. This is described as "objective optimisation" in Pugh et al. (2016), and it describes the aim of evolution to be working towards a singular goal. This is often coded into many evolutionary algorithms when the fitness is a measure of similarity to some arbitrary point. However, the rise of high performance GPUs corresponded to advances in machine learning techniques, and these provide more mathematically sound and computationally efficient methods of optimisation. Alternatively there exists the possibility of "intrinsically motivated learning" Barto et al. (2004). This is the scenario wherein an agent is not acting towards a defined goal. In these problems, as covered in Pugh et al. (2016), there is effectively no optimal function and the fitness only acts as a guiding hand. Hence, we find that the alternative and better use of evolution in computer science is for diversification. Such algorithms that do not look for targeted behaviour but instead search for high performance in a varied and diverse set of behaviours are defined as "Quality Diversity" (QD) algorithms.

2.2.1 Novelty Search

A well known basic QD algorithm within the field of evolutionary algorithms is Novelty Search from Lehman and Stanley (2010). Novelty search works by generating a new individual and then evaluating how different the this new individual is from the other members of the population. If the new individual is different or "novel" enough then it is added to the population. Though Novelty Search is often good at finding different behaviours, it does not take into account the fitness of the individuals and adds them regardless. To take fitness into account, Novelty Search was extended into algorithms that incorporate competition. The original extended algorithm was called Novelty Search with Global Competition (NSGC). This algorithm can be described as a two stage evolutionary algorithm that consists of a generation stage that employs Novelty Search, and a secondary fitness evaluation stage. This fitness evaluation stage evaluates the population as a whole according to a fitness function and only allows the fittest individuals in the whole population to remain. This was further extended into Novelty Search with Local Competition (NSLC) as defined in Lehman and Stanley (2011). NSLC differs from its global counterpart by not comparing members to the global greatest fitness, but rather localised fitness or fitness limited to a member's nearest neighbours. This can be done by creating clusters of similar members, and only allowing the fittest individuals to remain in the population. The idea behind this is that we can exploit Novelty Search's ability to find different behaviours while still getting high performance individuals.

2.2.2 MAP-Elites

An incredibly popular and well known QD algorithm is "multi-dimensional archive of phenotypic elites" or "MAP-Elites" from Mouret and Clune (2015). Technically not a optimisation algorithm, it is instead an "illumination algorithm" which is designed not to return a single "one size fits all" optimal result, but a variety of high performing results for each point of the feature space. For example, for a robot trying to get from point **A** to **B**, it would find the optimum for when a robot was small, fast and efficient, the optimum when the robot was big, slow and inefficient, and every combination given the number of features set beforehand. In articles such as Cully et al. (2015) it has been found that using MAP-Elites significantly improves performance.

procedure MAP-ELITES ALGORITHM (SIMPLE, DEFAULT VERSION)
  $(\mathcal{P} \leftarrow \emptyset, \mathcal{X} \leftarrow \emptyset)$ ▷ *Create an empty, N-dimensional map of elites: {solutions $\mathcal{X}$ and their performances $\mathcal{P}$}*
  for iter $= 1 \rightarrow I$ **do** ▷ *Repeat for I iterations.*
    if iter $< G$ **then** ▷ *Initialize by generating G random solutions*
      $\mathbf{x}' \leftarrow$ random_solution()
    else ▷ *All subsequent solutions are generated from elites in the map*
      $\mathbf{x} \leftarrow$ random_selection($\mathcal{X}$) ▷ *Randomly select an elite x from the map $\mathcal{X}$*
      $\mathbf{x}' \leftarrow$ random_variation($\mathbf{x}$) ▷ *Create x', a randomly modified copy of x (via mutation and/or crossover)*
    $\mathbf{b}' \leftarrow$feature_descriptor($\mathbf{x}'$) ▷ *Simulate the candidate solution x' and record its feature descriptor $\mathbf{b}'$*
    $p' \leftarrow$performance($\mathbf{x}'$) ▷ *Record the performance p' of x'*
    if $\mathcal{P}(\mathbf{b}') = \emptyset$ or $\mathcal{P}(\mathbf{b}') < p'$ **then** ▷ *If the appropriate cell is empty or its occupants's performance is $\leq p'$, then*
      $\mathcal{P}(\mathbf{b}') \leftarrow p'$ ▷ *store the performance of x' in the map of elites according to its feature descriptor $\mathbf{b}'$*
      $\mathcal{X}(\mathbf{b}') \leftarrow \mathbf{x}'$ ▷ *store the solution x' in the map of elites according to its feature descriptor $\mathbf{b}'$*
  return feature-performance map ($\mathcal{P}$ and $\mathcal{X}$)

FIGURE 2.4: MAP-Elites psuedocode from Mouret and Clune (2015)

Although both NSLC and MAP-Elites are proven and functional algorithms, superior or alternate versions have been developed such as in Cully and Demiris (2017) and Vassiliades et al. (2017). Cully and Demiris (2017) elaborates the mathematical detail and explains how both NSLC and MAP-Elites are applications of the same raw theory.

2.3 Behavioural Repertoires

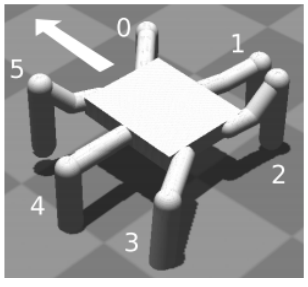

FIGURE 2.5: A simulated robot from Cully and Mouret (2013)

Describing Behavioural Repertoires (BR) and their various subsets is best done using an example. Observe the robot in Figure 2.5. This robot will have many motors operating each of its different legs. A BR in this case would be a "set of simple controllers that governs the motor commands sent to the robot" as defined in Cully and Demiris (2018). Here a controller would refer to the input necessary to control a specific motor. Each BR is characterised by a Behavioural Descriptor (BD), which describes the resultant behaviour of the robot from executing this controller. Simply put, the BD (sometimes referred to as Behavioural Characterisation Meyerson et al. (2016)) could be the description of

"forwards" and the set of motors required to move forwards is the corresponding BR. In reality BDs are not containers of literal words or concepts but rather a value or set of values that describe behaviour. These values could represent multiple different things such as the distance from preset points or how much energy was used, the only real limit is in how many features are hard-coded or in how many sensors the robot has access to. In this way we can think of BDs as representations of the total sensory information that the robot can observe. Although ideally low-level representations, due to the unconstrained limit on how much data BDs can contain, they often end up existing in high dimensional space. Modern solutions to this exist as covered in the Automatic Behavioural Repertories section.

2.3.1 Behavioural Repertoire Evolution

In Cully and Mouret (2013), the researchers describe a method wherein un-driven low level controllers (action commands that result in primitive behaviour like "step" or "turn right") were collated and learned to result in high level behaviour. Un-driven in this case means that there was no preset target that the robot was trying to achieve. They describe this method as BR-Evolution, and it optimises local performance and novel behaviours. While most learning in robotics evolves just one main controller, BR-Evolution evolves multiple simple controllers. Each simple controller matches to a specific behaviour. The algorithm then proceeds to use Novelty Search to discourage similar controllers in favour of diverse behaviours (MAP-Elites can also be applied to evolutionary BR methods Tarapore et al. (2016)). We can do this by comparing the BDs of each of these behaviours. If the Euclidean distance between BDs is small we can say that the values are similar and hence the behaviour is similar. However, as raised in Duarte et al. (2018) this only serves as generating locomotive behaviour without achieving an actual defined target. The same paper that criticises this also addresses the issue through their method of "Evolution of Repertoire-based Control" (EvoRBC), which is a two stage solution to a robot navigation problem. The first stage is effectively BR-Evolution to find the primitive locomotion methods. The second stage is what they define as an "arbitrator" that chooses which of these generated methods result in the desired behaviour as defined by the target.

2.3.2 Hierarchical Behavioural Repertoires

A subset of BRs, Hierarchical Behavioural Repertoires (HBRs) are an organised stack of multiple BRs, with higher levels of the hierarchy selecting lower level controllers, resulting in complex high level behaviour, as defined in Cully and Demiris (2018). Figure 2.6 is a figure from this paper as a representation of how HBRs work.

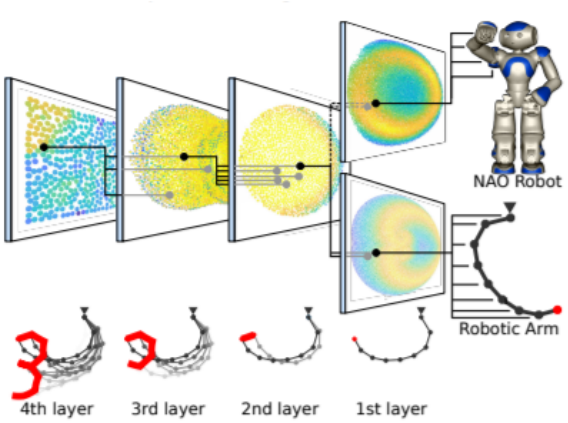

FIGURE 2.6: From Cully and Demiris (2018), a diagram of how Hierarchical Behavioural Repertoires work. From left to right controllers point to lower level controllers, with only the lowest level controlling the actual robot. This means that one high level controller can represent much more advanced behaviour than any low level controller.

Due to the fact that only the lowest level of this hierarchy interacts directly with the motors HBRs provide many benefits. These include scalability by decreasing the complexity of the optimisation problem and modularity by the interchangeable bottom layer being capable of being applied to different robots. A problem within HBRs is that combining various behaviours in sequence sometimes will not work due to the conditions of the environment (e.g. moving further right when at the rightmost point). Solving this issue by inputting information about the environment results in a massive increase in the size of the BR and in turn drastically increases the inefficiency of the computation. This issue is instead solved by treating the behaviour as probabilistic and hence using stochastic BDs. These are created by sampling behaviours and taking a mean. Not all the layers of a HBR will require stochastic BDs.

2.3.3 Automatic Behavioural Repertoires

Automatic Behavioural Repertoires (ABR) are behavioural repertoires that are not generated and defined by some user but are instead generated and then evaluated by an algorithm, such as the method in Meyerson et al. (2016). While ABR methods that use QD algorithms are certainly functional, a key issue they have is that originally they required a way to know which individuals were successful. This would imply that there is a pre-defined reward function and defeats the purpose of automating the process. The aim of ABRs is to remove user input and allow a robot to autonomously discover its own capabilities. In the recent methods that are employed to fix this issue we can discern two schools of thought. One is the exploration of multiple behavioural spaces so that each controller is not limited by one BD. The other is to use dimension reduction algorithms as in Vassiliades et al. (2017), in which the researchers implemented an algorithm to define BDs in high-dimensional feature space. This is done by using a customised

MAP-Elites algorithm called Centroidal Voronoi Tessellation (CVT). Effectively, this is a method of applying clustering algorithms like k-means in high feature space.

However ABRs currently lack an automated method for figuring out which type of behaviour space will improve the overall objective of the population. A possible solution to this is meta-evolution Bossens et al. (2020). In the field of machine learning meta-learning refers to the concept of "learning to learn", the idea that given a small dataset of training examples an algorithm should be able to create the learning method that is best suited to learn from this dataset. Meta-evolution follows suit with the core idea being that each of the individual members in the population of and evolutionary algorithm is a QD algorithm. So meta-evolution evolves a population of quality diversity algorithms.

ABRs are often integrated into the field of AI due to the nature of learning processes. This often means taht QD algorithms are combined with deep neural networks. In Cully and Demiris (2018), the researchers used unsupervised learning models to generate the BRs. Such a practice is completely hands free and must approach the pure evolutionary and exploratory method that the field is currently looking for.

2.3.4 AURORA

In this paper we build off of AURORA from Cully (2019). In the Automatic Behavioural Repertoires section we discussed two types of ABR algorithms. AURORA is of the latter type and uses a dimension reduction algorithm. AURORA consists of two main phases. *Phase 1:* A dimension reduction algorithm is trained on a data set of sensory information gleaned from randomly generated controllers, and gains a low dimensional representation of the data. *Phase 2:* QD iterations are applied and controllers are selected according to curiosity score proportionate selection. The new controllers are then added to the data set and Phase 1 repeats, now using the controllers that have been generated. After a given retraining phase not all the controllers will be added back to the population. This is because the same novelty rule that determined whether a new member could be added to the population during the QD phase is consistent. So after retraining each controller is given its new BD and the controllers are only added back to the population if the BDs can be considered novel.

AURORA has been applied to two tasks: a ballistic task where a ball is thrown and the controller controls the angle and force at launch; and an air hockey task where a puck is knocked around by a robotic arm with 4 degrees of freedom where the controller controls the arm. The results of both have been that the algorithm produces behavioural repertoires that represents a large range of behaviours similar to the range that would have been created given prior knowledge about the domain space.

2.4 Autoencoders

2.4.1 Introduction to Autoencoders

Autoencoders are algorithms designed to compress data usually utilising a neural network. They fit within the umbrella term of unsupervised learning as no user input is required to label the data before training Baldi (2012). Autoencoders have two main features. The first is that they are specific to the data that they have been trained on, unlike task specific compression. This means that, for example, an autoencoder trained on a dataset of images of boats will be bad at compressing an image of a cat. The second feature is the more beneficial one. There is no need to design a custom compression algorithm for each dataset or to know general details about the dataset as the autoencoder will learn this. This is increasingly beneficial with the variety of datasets that exist.

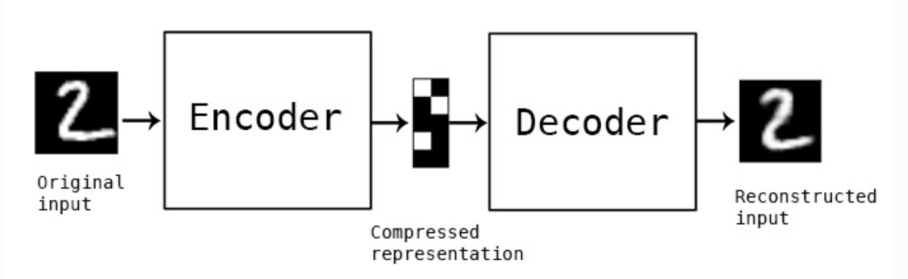

FIGURE 2.7: Simple view of autoencoder system from "Building Autoencoders in Keras" (`https://blog.keras.io/building-autoencoders-in-keras.html`)

Training an autoencoder requires three components: an encoder **enc** that will project the input data into latent space; a decoder **dec** that will reconstruct the data from latent space so that we get our output data; and finally a distance measure **dist** that we can use to calculate the reconstruction loss between the input and output. This distance will be our loss which we can use to calculate the gradients needed to train and improve the autoencoder's respective parts. So to train **enc** and **dec**given input value x:

$$\begin{aligned} &\text{Step 1)} && latent = \mathbf{enc}(x) \\ &\text{Step 2)} && y = \mathbf{dec}(latent) \\ &\text{Step 3)} && loss = \mathbf{dist}(x, y) \\ &\text{Step 4)} && \text{Train } \mathbf{enc} \text{ and } \mathbf{dec} \text{ using } loss \end{aligned} \tag{2.1}$$

A consensus in the field is that autoencoders are not practical compression algorithms in the real world. The main purpose of a compression algorithm is to be able to compress any type of data so autoencoders are not ideal for that purpose. In the case of linear autoencoders, where the autoencoder is designed as a very basic $\mathbf{enc}(x) = \mathbf{W}_{enc}x + \mathbf{b}_{enc}$ and $\mathbf{dec}(\mathbf{enc}(x)) = \mathbf{W}_{dec}\mathbf{enc}(x) + \mathbf{b}_{dec}$, it has even been found that they simply

learns how to map the data to the same feature space as Principal Component Analysis (PCA) Bourlard and Kamp (1988). Though this could be argued as good compression there are already closed form mathematical solutions for PCA such as Singular Value Decomposition (SVD). Autoencoders have, however, seen popular use in data denoising such as in Vincent et al. (2008). During training an autoencoder is introduced to a noisy version of the data input and the loss is calculated against the original noise-free clean data. The result is an autoencoder that can successfully denoise the input data. Convolutional autoencoders can extract high level features of a dataset, so when they are trained in this manner they excel at denoising data.

Although denoising autoencoders is one method to effectively regularise autoencoders, other methods exist such as sparse autoencoders. The method used by these autoencoders can be considered as similar to dropout as covered in Srivastava et al. (2014). When dropout is used during training of a neural network it means that a number of nodes may be masked and not considered so that any value that they could add to or take away from the final output is set to 0. Effectively they are "dropped out", hence the term. The result of this is that unlike in most neural networks where nodes can rely on the specialisation of their neighbours or even of other nodes in previous layers, the nodes must now try to compensate and hence generalise in order to decrease the loss. This results in regularising behaviour and noticeably less overfitting. Similarly, in sparse autoencoders only a small number of the hidden units are allowed to be active at any point. Due to the reasons illustrated, this produces more regularised autoencoders.

2.4.2 Modern Extensions to Autoencoders

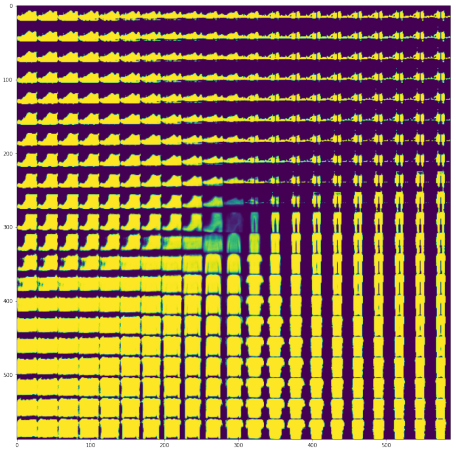

FIGURE 2.8: Images sampled from a Variational Autoencoder trained on the Fashion-MNIST dataset. This was generated by passing in values for log-sigma into the decoder function. The standard deviation was 2-dimensional and values that 21 different values from -4 to 4 were passed in for both dimensions.

There are many modern and growing extensions and novel approaches to autoencoders. One of them is Variational Autoencoders (VAE), originally introduced in Kingma and Welling (2013). VAEs, as covered in Doersch (2016), are very much similar to autoencoders and differ only in that the hidden layer consists of two vectors: one contains the means of the latent distributions and the other contains the log-sigma or the standard deviation of the latent distributions. The decoder then proceeds to take a sample from this latent distribution and

generates a reconstruction after whatever layers it contains. VAEs are effectively generative autoencoders. VAEs are trained using two different loss functions: the Kullback-Leibler Divergence between the current latent distribution and the distribution of the actual dataset and, as before, the reconstruction loss. An example of data being sampled from a VAE can be seen in Figure 2.8.

A very recent extension to autoencoders and another generative type is the Wasserstein Autoencoder (WAE) introduced in Tolstikhin et al. (2017). WAEs are very similar to VAEs in their construction and differ primarily in what type of loss they are attempting to minimise. Contrary to VAEs that attempt to minimise the Kullback-Leibler Divergence between the model distribution and the target distribution, WAEs minimise a modified form of the Wasserstein distance between these two distributions. Understanding the Wasserstein distance takes some time. We first need to define a "transportation policy" $\gamma(\mathbf{x}, \mathbf{y})$. This is the effective amount of probability mass that is transferred from $p(\mathbf{x})$ to $q(\mathbf{y})$:

$$p(\mathbf{x}) = \int \gamma(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{y} \qquad q(\mathbf{y}) = \int \gamma(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{x} \tag{2.2}$$

What this means is that if we imagine probability histograms as made of blocks, how much would you have to move to take the blocks of one histogram to fully recreate another histogram of a different probability distribution. So we want a policy $\gamma(\mathbf{x}, \mathbf{y})$ that minimises the amount of probability mass that needs to be moved. The less that needs to be moved the "closer" the two distributions are. We define a distance measure $d(\mathbf{x}, \mathbf{y})$. With this we can define the cost of a given transport policy, $\mathcal{C}(\gamma)$ as:

$$\mathcal{C}(\gamma) = \int \int d(\mathbf{x}, \mathbf{y}) \gamma(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{x} \mathrm{d}\mathbf{y} \tag{2.3}$$

$d(\mathbf{x}, \mathbf{y})$ can be any distance, such as Euclidean: $||\mathbf{x} - \mathbf{y}||$. We now imagine that there is a set of transport policies, which we call $\Gamma(p, q)$. So $\gamma \in \Gamma(p, q)$. With all this in mind, for a given $d(\mathbf{x}, \mathbf{y})$ we can finally define the Wasserstein distance $\mathcal{W}(p, q)$ as the minimum expected value of the distance across a set of policies:

$$\mathcal{W}(p, q) = \min_{\gamma \in \Gamma(p,q)} \mathbb{E}_{\gamma}[d(\mathbf{x}, \mathbf{y})] \tag{2.4}$$

2.5 Recurrent Neural Networks

2.5.1 Basic Theory

Recurrent Neural Networks (RNNs) stem from the idea that in a time series, historic data could effect future data. While standard neural networks could take in individual inputs of the time series to figure out what the output would be, they do not take into account the relationship between the data points. To account for previous data's effect

on current data we need some method to remember past inputs. RNNs address this by outputting a hidden state value or vector of values after each input into the network. It feeds this hidden state back into itself, hence the term "recurrent". This hidden state is a powerful tool that can capture how neighbours effect each other and has subsequently resulted in many real world applications such as text generation, language modelling and text summarisation. All of these are cases in which the order and sequence of the inputs is directly relevant to the next output. Figure 2.9 displays the inherent difference between standard neural networks and recurrent neural networks.

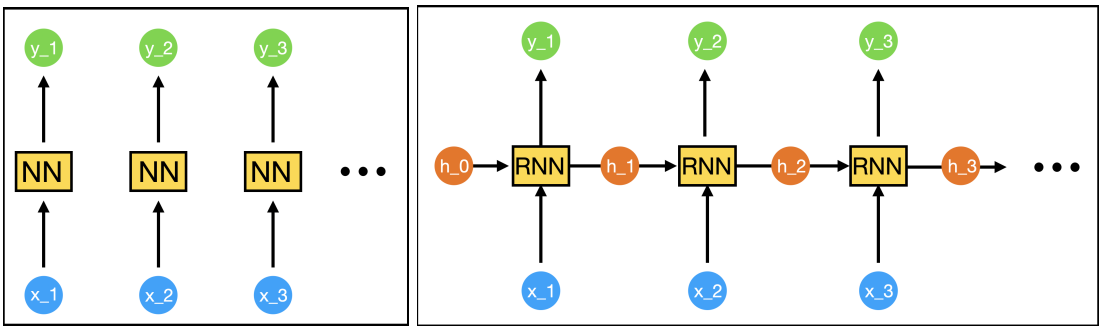

FIGURE 2.9: **Left:** A standard neural network taking in different data points as input and producing an output. **Right:** A RNN taking in a time series and outputting hidden states and outputs (h_0 is initialised to 0 as there is no data yet). Unlike a standard neural network it outputs a hidden state that is passed on to the next cell to aid in the prediction of the next output.

The simplest RNN are Elman networks introduced in Elman (1990)). Given a sequence of inputs $\mathbf{x}_1 ... \mathbf{x}_n$, the output for input $\mathbf{x}_t$ is calculated by:

$$\begin{aligned} \mathbf{h}_t &= \sigma_h(\mathbf{W}_i \mathbf{x}_t + \mathbf{b}_i + \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{b}_h) \\ \mathbf{y}_t &= \sigma_y(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y) \end{aligned} \tag{2.5}$$

Where $\mathbf{h}_{t-1}$ is the hidden state from the previous input, $\mathbf{W}_i$ and $\mathbf{b}_i$ are the weights and bias of the input, $\mathbf{W}_h$ and $\mathbf{b}_h$ are the weights and bias for the previous hidden state, $\mathbf{W}_y$ and $\mathbf{b}_y$ are the weights and bias of the new hidden state, and σ_h and σ_y are the activation functions of the hidden state and the output respectively. The weights are trained using a measure of reconstruction loss between the true outputs of the sequence against the predicted outputs. This reconstruction loss can range from being a direct Euclidean distance to a softmaxed cross entropy loss, as the problem itself will determine what loss will be optimal to calculate gradients off of.

2.5.2 Embeddings

An issue that needs to be considered is how to encode the actual inputs into the network. Directly inputting raw data will not always work as the data could be strings taken from a text or multidimensional such as coordinates. In these cases we must encode the data

so that the RNN can learn to predict the encoded version. This RNN output can later be decoded into a readable output. The simplest version of encoding is one hot encoding. This consists of creating a vector where all but one value is 0 and the odd one out is set to 1. The vector needs to be as long as there are different possible inputs into the network. Obviously one hot encoding is not efficient at all and not suitable for real world problems. The English language alone has more than 171000 words which would mean every input needs a vector longer than 171000, the contents of which nearly all will be 0. Another issue in one hot encoding is the idea of orthogonality, by which we mean that no input is considered to be similar to another. Firth, of Firth (1957) fame, was one of many authors who popularised the idea that words had similarities that could be physically measured. In doing so the idea of reducing the dimensional size of the encoding was combined with the idea that certain inputs could be considered similar to others.

Latent Semantic Analysis (LSA) introduced in Deerwester et al. (1990) is one such method. The initial hypothesis is that words that mean similar things will occur in similar text segments, raising the idea that a text segment has a specific sentiment. Certain words are ignored such as articles and conjugates (like "the" or "and") as they do not represent any specific sentiment. Alternatively we can use term frequency-inverse document frequency (TF-IDF) where the value of the relative importance of a given word is increased if it occurs frequently in a given text, but decreased if it is found to increase in all texts. By doing so it is not necessary to preprocess data and there can be less user input. This also allows for this method to be used in any verbal language with disjoint words. We can then take this frequency or relative importance matrix and use truncated SVD to de-noise the matrix by only considering the most significant features (see Figure **??** for a visualisation). This provides us with a much lower dimensional representation.

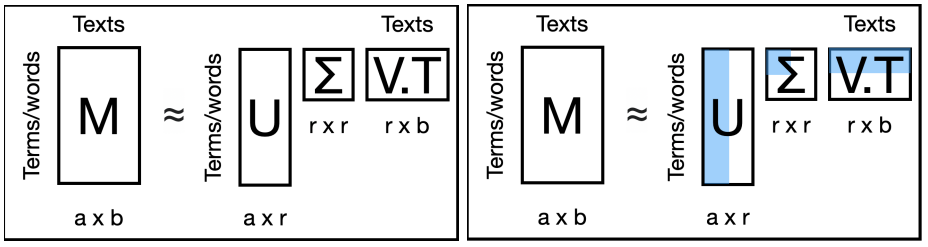

FIGURE 2.10: **Left:** Standard SVD given an input of terms and their frequency in a number of different texts. **Right:** Truncated SVD only considers the most significant features, and will reconstruct the matrix using matrices of smaller dimensions which have been highlighted in blue.

A popular method of dimension reduction that maintains some feature representation are *embeddings*. These are learned low-dimensional vector representations of the discrete input variables. For example, let us say we are given high dimensional data that has 50 features. The idea is that an embedding can take this data and meaningfully compress

it into a lower dimensional vector, for example of 3 features. This embedding would then be able to place individuals that could be considered similar close to each other in the embedding space. This makes embeddings ideal visualisation problems as can be seen in Figure 2.11, as well as make them useful as inputs into RNNs. Popular word embeddings include "word2vec" (introduced in Mikolov et al. (2013a) and extended in Mikolov et al. (2013b)) and "GLoVe" (introduced in Pennington et al. (2014)). These are not training methods but rather pretrained models that many people use.

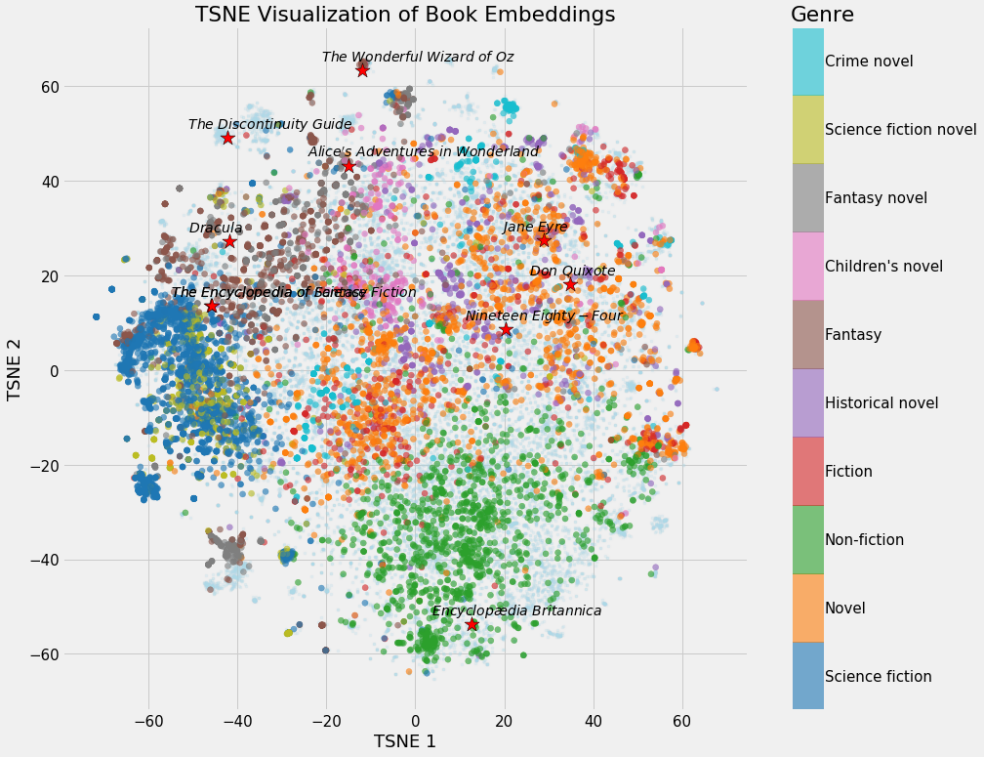

FIGURE 2.11: An example of how neural network embeddings can be used to visualise high dimensional data in low dimensional space while maintaining feature representation as generated in Koehrsen (2018)

2.5.3 LSTMs and GRUs

Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) are popular types of memory unit that are used in RNNs. LSTMs are capable of learning long term dependencies by learning when to forget and when to change memory. Like in standard RNNs we consider a single cell or network as taking in inputs of the new input in the sequence $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ and outputs $\mathbf{y}_t$. However, LSTMs take as input the previous LSTM cell's output and concatenates this previous output and the new input vector into one vector $\mathbf{z}_t = (\mathbf{x}_t, \mathbf{y}_{t-1})$. Using multiple different trainable weights and biases we then update the hidden state $\mathbf{h}_{t-1}$, which now referred to as the long term memory in contrast to $\mathbf{z}_t$ which is now referred to as the short term memory. A visual representation of a single LSTM cell can be seen in Figure 2.12

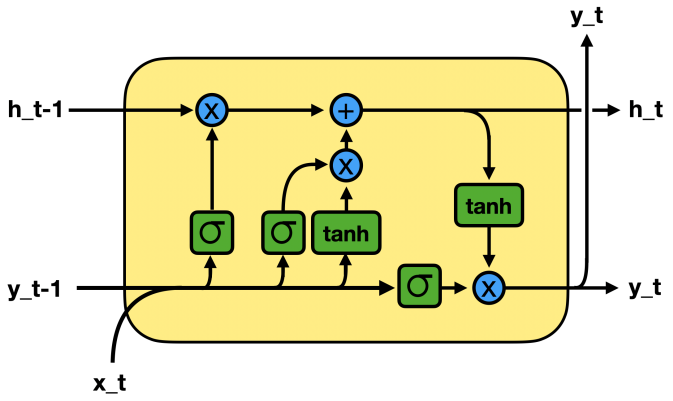

FIGURE 2.12: Visual representation of the calculations that occur in the LSTM. σ is sigmoid activation function and **tanh** is a hyperbolic tangent activation function.

A GRU is a variant of an LSTM that has two gates rather than three. The LSTM's gates function as input, output and forget gates whereas a GRU's gates function as reset and update gates. As covered in Chung et al. (2014), they are mathematically speaking effectively the same as LSTMs but are less computationally intensive. That being said there is no clear evidence of which performs better even though GRUs were developed more recently. There is some anecdotal evidence/rule of thumb within the community that GRUs are better for small datasets and LSTMs perform slightly better when used on large datasets. However what is meant by "large" and "small" is unclear and often varies depending on the dimensionality of each data point. Figure 2.13 displays the structure of a GRU and the following equations correspond to what data is being sent to where.

$$\begin{aligned} \mathbf{r}(t) &= \boldsymbol{\sigma}(\mathbf{W}_r \mathbf{z}_t + \mathbf{b}_r) & \mathbf{u}(t) &= \boldsymbol{\sigma}(\mathbf{W}_u \mathbf{z}_t + \mathbf{b}_u) \\ \mathbf{n}(t) &= \mathbf{tanh}(\mathbf{W}_n \mathbf{v}_t + \mathbf{b}_n) & \mathbf{y}_t &= (1 - \mathbf{u}(t)) \cdot \mathbf{y}_{t-1} + \mathbf{u}(t) \cdot \mathbf{n}(t) \end{aligned} \tag{2.6}$$

Where $\mathbf{v}_t = (\mathbf{x}_t, \mathbf{r}(t) \cdot \mathbf{y}_{t-1}$, the vector product of the concatenated input and reset gate output and the previous GRU cell's output

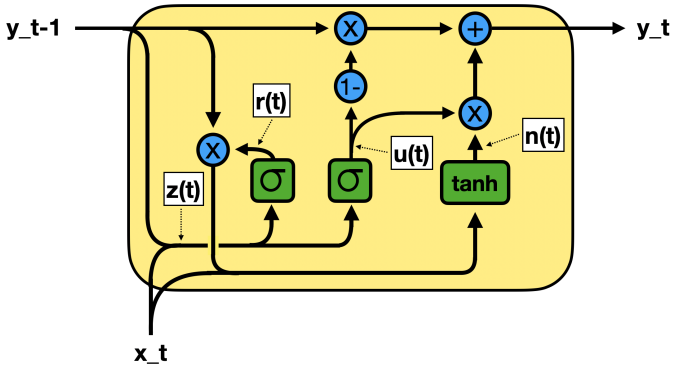

FIGURE 2.13: Visual representation of the calculations that occur in a GRU. As before, σ is sigmoid activation function and **tanh** is a hyperbolic tangent activation function.

Chapter 3

Method

3.1 Recreating AURORA

3.1.1 Search Algorithm

In this project we employed Novelty Search. It is easiest to explain how the algorithm works by going through examples. Let us begin with a population of five individuals. For the sake of simplicity, individuals within this population have two features. We can plot these individuals and observe their position in feature space, as seen in Figure 3.1.

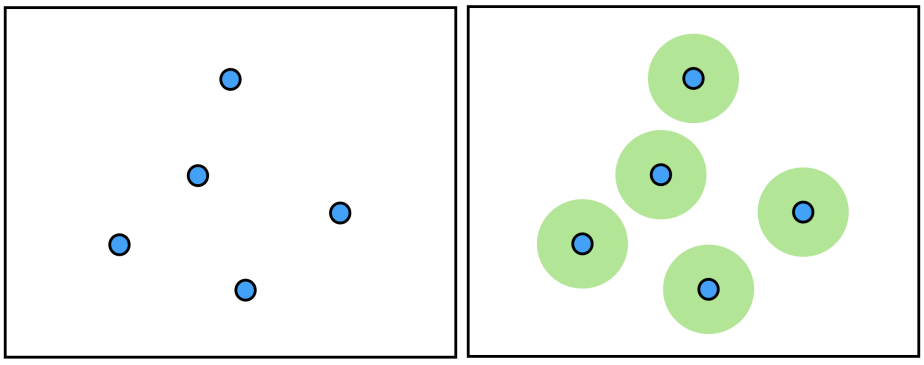

FIGURE 3.1: **Left:** Members of a population represented in feature space. **Right:** Same as left but the members of the population are now displaying their novelty.

Novelty Search has two key values known as "novelty" and the "novelty threshold". In two dimensional space we can think of novelty as the distance between two individuals. Novelty Search requires that the minimum novelty for any individual, i.e. the distance between and individual and its nearest neighbour, to be greater than or equal to the novelty threshold. The fields that are displayed on the right hand plot represent circle with a radius equal to the novelty threshold. If any new individual that is added to the population rests within one of these green areas, it cannot be considered novel. Let us add a new member to the initial population. As seen in left image of Figure 3.2.

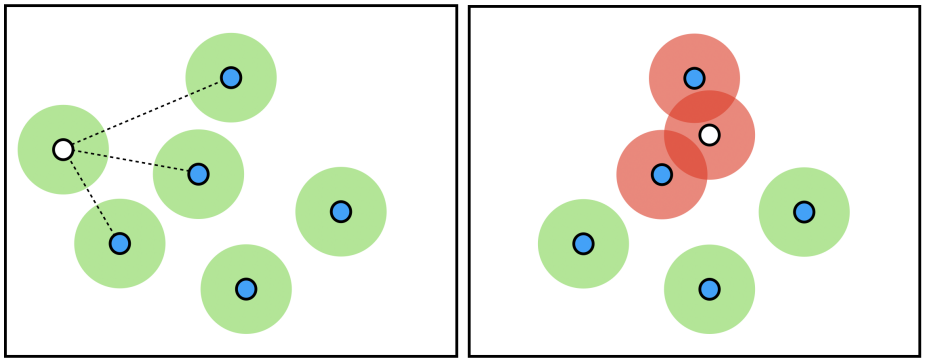

FIGURE 3.2: **Left:** A novel individual has been generated. It will be added to the population. **Right:** A individual with novelty below the threshold has been generated. It will not be added to the population

The new individual to the far left has a novelty greater than the novelty threshold. So it can be added to the population with no problem. Alternatively we can have an individual that is not novel, as can be seen in right image of Figure 3.2. This individual is not novel and is therefore not added to the population. What follows is the pseudo-code that would implement this algorithm.

LISTING 3.1: Novelty Search

```
1
2  def calculate_novelty(new_indiv, population, threshold):
3      nearest_neighbour = None
4      dist_from_n_n = 0.0
5      for member in population:
6          dist = np.linalg.norm(new_indiv - member)
7
8          # Is this the closer than the current nearest neighbour?
9          if dist < d_from_n_n:
10             dist_from_n_n = dist
11             nearest_neighbour = member
12
13     return novelty
```

Though the algorithm seems simplistic here there is a specific case that has not yet been considered known as "domination". Here domination is similar to the Pareto-Dominance definition, i.e. the principle by which individual A can be considered to be better than individual B in every metric such that individual A will always be chosen. An example of the process of domination can be seen in Figure 3.3.

In the case of strict Pareto dominance, the new individual and its nearest neighbour measure their novelty respective to their second nearest neighbours. If the new individual is more novel then it dominates its nearest neighbour, so the nearest neighbour is removed from the population and the new individual remains. In this project a modified version called "exclusive ϵ-dominance" is used, as defined in Cully and Demiris (2017). This is because in standard dominance it is required that all new individuals must be

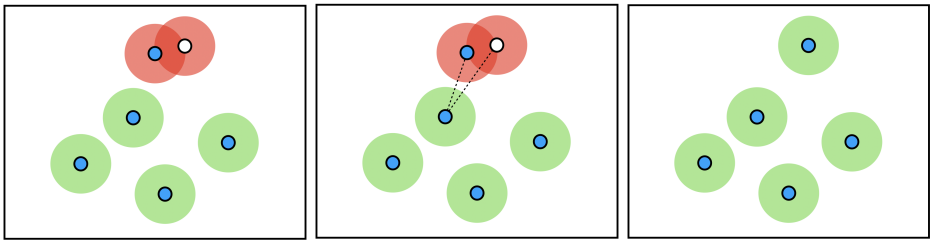

FIGURE 3.3: **Left:** An individual that is not novel is added to the population. **Middle:** The new individual and its nearest neighbour have their novelty evaluated. **Right:** The new individual has a greater novelty and replaces the nearest neighbour in the population.

better and more diverse, meaning many will not be added to the population and result in an overall non diverse population. Exclusive ϵ-dominance softens this by introducing three metrics that a new individual x_1 must meet in order to dominate an existing individual x_2:

$$\begin{aligned} &\textbf{1)} \quad \mathbf{N}(x_1) \geq (1-\epsilon) \times \mathbf{N}(x_2) \\ &\textbf{2)} \quad \mathbf{Q}(x_1) \geq (1-\epsilon) \times \mathbf{Q}(x_2) \\ &\textbf{3)} \quad (\mathbf{N}(x_1) - \mathbf{N}(x_2)) \times \mathbf{Q}(x_2) > -(\mathbf{Q}(x_1) - \mathbf{Q}(x_2)) \times \mathbf{N}(x_2) \end{aligned} \tag{3.1}$$

Where $\mathbf{N}$ calculates the novelty, $\mathbf{Q}$ calculates the fitness and ϵ is a defined preset value (in this project it was set at 0.1). So unlike standard ϵ-dominance as defined in Laumanns et al. (2002), the dominating individual is allowed to be worse than the one it may be replacing by one objective.

What follows is a simplified pseudo-code version of Novelty Search with domination as it was implemented in the project code.

LISTING 3.2: Novelty Search with Domination

```
def does_dominate(threshold, k_n_n, d_from_k_n_n, population):
    dominated_indiv = -1
    x_1_novelty = d_from_k_n_n[0]
    # If novelty threshold is greater than the nearest neighbour but less
     than the second nearest neighbour
    if (threshold > d_from_k_n_n[0]) and (threshold < d_from_k_n_n[1]):
        pop_without_x_2 = population.copy()
        del pop_without_x_2[k_n_n[0]]
        x_1_novelty = d_from_k_n_n[1]
        x_2_novelty, _ = calculate_novelty(population[k_n_n[0]].get_bd(),
     pop_without_x_2, threshold, False)

        # Find if exclusive epsilon dominance is met
        if x_1_novelty >= (1 - EPSILON) * x_2_novelty:
            dominated_indiv = k_n_n[0]
    return dominated_indiv, x_1_novelty

```

```
16
17 def calculate_novelty(new_indiv, population, threshold, check_dom):
18     k_n_n = []              # K nearest neighbours
19     d_from_k_n_n = []
20     for member in population:
21         dist = np.linalg.norm(new_indiv - member)
22         if len(d_from_k_n_n) < K_NEAREST_NEIGHBOURS:
23             d_from_k_n_n.append(dist)
24             k_n_n.append(member)
25         else:
26             # Is new novelty larger than the maximum?
27             if dist < max(d_from_k_n_n):
28                 d_from_k_n_n[max_index] = dist
29                 k_n_n[max_index] = member
30
31     novelty = d_from_k_n_n[0]
32     dominated_indiv = -1
33
34     if check_dom == True:
35         dominated_indiv, novelty = does_dominate(threshold, k_n_n,
       d_from_k_n_n, population)
36
37     return novelty, dominated_indiv
```

As a selector we can use the curiosity score introduced in Cully and Demiris (2017). This is the tendency of an individual to produce offspring, and unlike the fitness function it increases and decreases according to whether or not it produces viable offspring after selection. So if c is the curiosity score of a given parent:

$$c = \begin{cases} c + \text{REWARD} & \text{if offspring added to population} \\ c - \text{PENALTY} & \text{if offspring not added to population} \end{cases}$$

In this project the reward was set to 1.0 and the penalty was set to 0.5 in accordance with the framework set up in Cully and Demiris (2017). The roulette wheel is then constructed by calculating the total curiosity of the population and then dividing each member's curiosity by this sum to get the probability of selection for each individual. As the curiosity is initialised to 0 there needs to be a minor offset so that new individuals will be selected and the algorithm doesn't fail on initialisation. This offset is added to every individual's curiosity and taken into account when calculating the probabilities. In this project the offset was set to 0.01.

Finally for the search method we need a mutation algorithm. In this project we used polynomial mutation to the same experimental standards as the original work. The method for polynomial mutation is as follows: given a range of values $x_i^{(L)}$ to $x_i^{(U)}$ in which each gene of a genotype is limited to, for a given parent p the mutated offspring p' is:

$$p' = \begin{cases} p + \delta_L(p - x_i^{(L)}) & for \leq 0.5 \\ p + \delta_R(x_i^{(U)} - p) & for > 0.5 \end{cases}$$

Where:

$$\begin{aligned} \delta_L &= (2u)^{\frac{1}{1+\eta_m}} - 1 \\ \delta_R &= 1 - (2(1-u))^{\frac{1}{1+\eta_m}} \end{aligned} \tag{3.2}$$

Where η_m is a preset parameter. In Figure 3.4, taken from Deb and Deb (2014), we can see the probability density function for a given parent to produce mutated offspring.

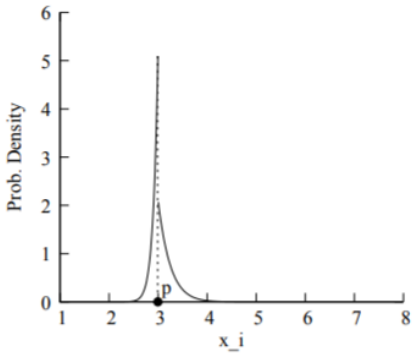

FIGURE 3.4: From Deb and Deb (2014), the probability density function for a given parent p to produce mutated offspring given bounds 1 to 8

3.1.2 Training the Autoencoder

Figures 3.5 and 3.6 display diagrams of the autoencoder design used for the ballistic task in AURORA. The encoder input vector of sensory information of size 1×100 is passed through a 2-dimensional convolutional layer, a max-pooling layer and two fully connected layers, both with sigmoid activation, before reaching latent space representation. The decoder then takes this latent space representation and passes it through two fully connected layers with sigmoid activation to extrapolate the information from the encoded representation, and then pass that output into two 2-dimensional deconvolutional layers before passing into a final fully connected layer, this time with identity activation.

The autoencoder was trained using the gradients calculated by the reconstruction loss between the network input and the output. The Adagrad optimizer from Duchi et al. (2011) was used and the learning rate was decayed exponentially. At each epoch of training the learning rate was set according to the equation:

$$\text{learning rate} = 0.1 \times 0.9^{\frac{\text{current epoch}}{\text{total epochs}}} \tag{3.3}$$

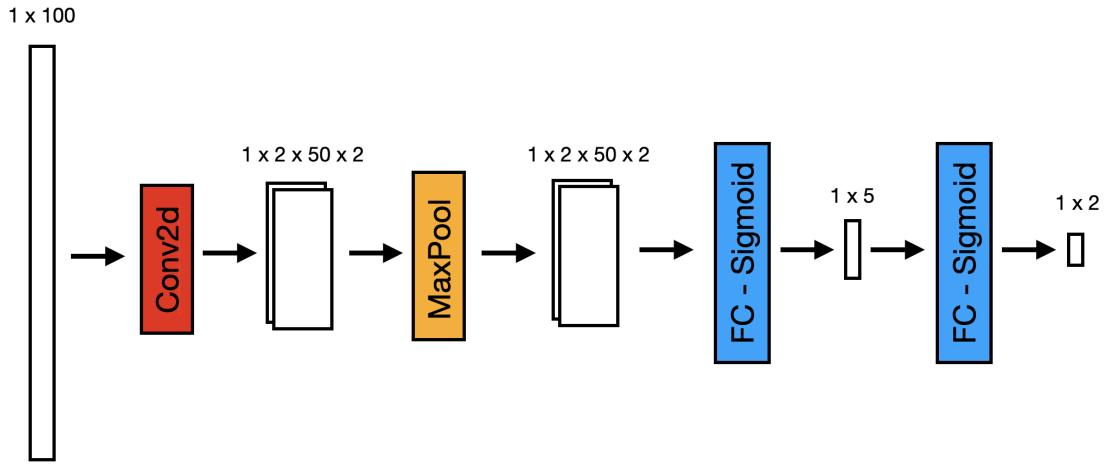

FIGURE 3.5: Design of the encoder half of AURORA's (Cully (2019)) autoencoder.

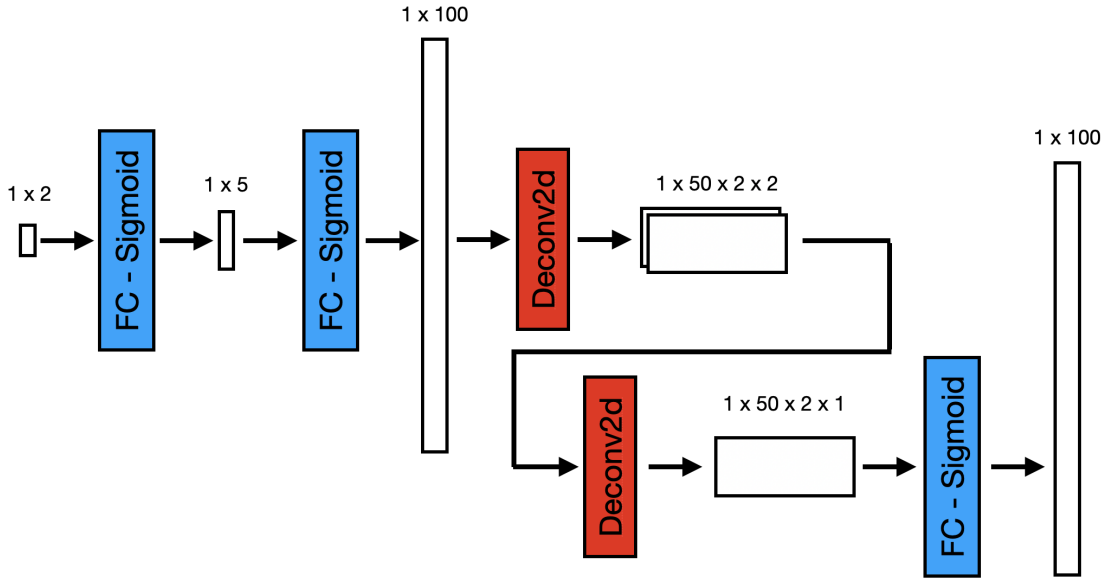

FIGURE 3.6: Design of the decoder half of AURORA's (Cully (2019)) autoencoder.

As always in machine learning we must consider the case of overfitting (when the model can accurately predict training data at the cost of making incorrect predictions based on validation data). Although this would likely not be an issue once the population or repertoire size was large, it is a problem at initialisation when it is small. Overfitting is avoided by first randomly splitting the data into training and validation sets at proportions of 75% and 25%. In the original AURORA paper (Cully (2019)) each training session went on for 25000 epochs. If the average validation loss over the last 500 epochs had increased or if the maximum epoch limit had been reached then training was terminated. The data would then be split into another random 75% and 25% split and the process was repeated. This process of splitting data and training was repeated 5 times for each training session. This means that training could theoretically run for 5 times the current epoch limit if the validation loss does not increase. To limit this if a

training session ends either due to reaching the epoch limit or an increase in validation loss, *and* the total number of epochs of training was greater than or equal to double the epoch limit, then training would end. During initial testing it was found that 25000 epochs was beyond over-zealous and took extensive computation time, even though the overwhelming majority of training was finished by the first 500 epochs. To both decrease computation time and make the system more efficient, the maximum number of training epochs was set to 500 and the validation loss was averaged over the last 50 epochs. This value was not arbitrarily chosen but was instead discerned after initial testing to see at how many epochs the model began to overfit. Some of the initial testing plots used to discern this are shown in Figure 3.7.

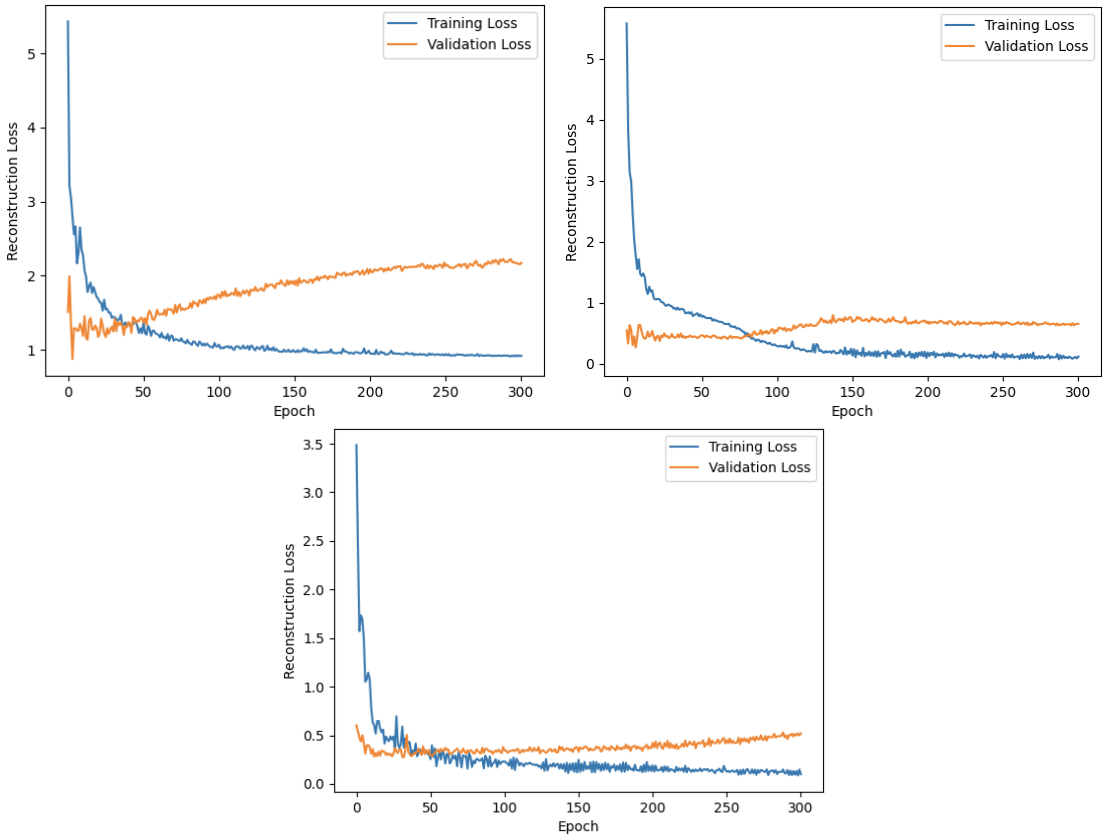

FIGURE 3.7: Some of the loss plots from initial testing. The population sizes were set to be small, at values of values from 80 to 200. The blue line is the training error and the orange line is the validation error. Clearly overfitting is occurring early on, thus the loss was averaged over the last 50 epochs.

The network was built using the original AURORA code's Tensorflow implementation. Tensorflow Version 1 was used in this project as it was in the original code. The trainable weights in this network use Xavier Initialisation introduced in Glorot and Bengio (2010). Xavier initialisation was an attempt to initialise the weights of a network to make the variance of the outputs of that specific layer to be equal to the variance of the inputs of that specific layer. When logistic sigmoid is used as the activation function it was found that this property was very useful until ReLU (first present in Hahnloser et al. (2000)

and implemented in Glorot et al. (2010)) and variations such as Leaky ReLU became the popular activation function.

After a training session a new population container was generated and each member of the current population was assigned their new behavioural descriptor. Then the members of the old population are only added to the new population if the new BDs are novel enough. A proportion of the population will be lost after these training periods, and combined with the new network the latent space will now be different. It is therefore necessary to modify the novelty threshold of the Novelty Search so that we can continue exploring the latent space efficiently. This is done by calculating the maximum distance between two points in the current latent space and dividing them by an arbitrary value such that the resultant threshold is small enough. This arbitrary value is set to the square root of 60000. The following code is the novelty calculator as implemented in the project code. It does the standard calculation of calculating the distance between any two points in two dimensional Euclidean space by using the hypotenuse. So if one point has coordinates (x_0, y_0) and the other point has coordinates (x_1, y_1), the distance is calculated as $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$. The code following code implements this with matrices so that it is computationally more efficient.

Listing 3.3: Novelty Threshold Calculator

```
def calculate_novelty_threshold(latent_space):

    rows, cols = latent_space.shape
    X = np.zeros((rows, cols))
    for i in range(rows):
        for j in range(cols):
            X[i][j] = latent_space[i][0][j]

    XX = np.zeros((rows, 1))

    for i in range(rows):
        sigma = 0
        for j in range(cols):
            sigma += X[i][j]**2
        XX[i][0] = sigma

    XY = (2*X) @ X.T

    dist = XX @ np.ones((1, rows))
    dist += np.ones((rows, 1)) @ XX.T
    dist -= XY

    maxdist = np.sqrt(np.max(dist))

    #  arbitrary value to have a specific "resolution"
    K = 60000

    new_novelty = maxdist/np.sqrt(K)
```

```
29      return new_novelty
```

In doing so it was realised that there was a way to make the standard novelty calculator more efficient.

LISTING 3.4: Computationally Efficient Novelty Search with Domination

```
1  def make_novelty_params(latent_space):
2
3      rows, cols = latent_space.shape
4      x_sq = np.zeros((rows, 1))
5      two_x = np.zeros((rows, 1))
6      y_sq = np.zeros((rows, 1))
7      two_y = np.zeros((rows, 1))
8
9      for i in range(rows):
10         x_sq[i] = latent_space[i][0]**2
11         two_x[i] = 2 * latent_space[i][0]
12         y_sq[i] = latent_space[i][1]**2
13         two_y[i] = 2 * latent_space[i][1]
14
15     return x_sq, two_x, y_sq, two_y
16
17 def does_dom(th_hold, k_nn, d_from_k_nn, x_sq, two_x, y_sq, two_y, pop):
18     dominated_indiv = -1
19     x_1_novelty = d_from_k_nn[0]
20     if (th_hold > d_from_k_nn[0]) and (th_hold < d_from_k_nn[1]):
21         x_1_novelty = d_from_k_nn[1]
22         x_2_novelty, _ = calculate_novelty(pop[k_nn[0]].get_bd(), th_hold
       , False, x_sq, two_x, y_sq, two_y, pop)
23         if x_1_novelty >= (1 - EPSILON) * x_2_novelty:
24             dominated_indiv = k_nn[0]
25
26     return dominated_indiv, x_1_novelty
27
28 def calculate_novelty(this_bd, th_hold, check_dom, x_sq, two_x, y_sq,
       two_y, pop):
29     two_x_new_x = two_x * this_bd[0]
30     two_y_new_y = two_y * this_bd[1]
31     size = len(x_squared)
32     new_x_sq = np.full((size, 1), this_bd[0]**2)
33     new_y_sq = np.full((size, 1), this_bd[1]**2)
34
35     sq_distances = x_sq - two_x_new_x + new_x_sq + y_sq - two_y_new_y +
       new_y_sq
36
37     min_dist = np.sqrt(np.min(sq_distances))
38     nn = np.argmin(sq_distances)
39
40     sq_distances[nn] = 99999999
41
```

```
42      second_min_dist = np.sqrt(np.min(sq_distances))
43      second_nn = np.argmin(sq_distances)
44
45      novelty = min_dist
46
47      dominated_indiv = -1
48
49      if check_dom:
50          dominated_indiv, novelty = does_dom(th_hold, [nn, second_nn], [
        min_dist, second_min_dist], x_sq, two_x, y_sq, two_y, pop)
51      else:
52          novelty = second_min_dist
53
54      return novelty, dominated_indiv
```

3.1.3 Evaluation

Evaluation of the implementation of AURORA could be done in two ways: comparison with the original results and using the same metrics as used in the original paper. To do this the latent space, "ground truth", repertoire size and reconstruction loss (root mean squared error) were recorded for each run. On top of this the Kullback-Leibler Coverage (KLC) score, used in the original paper and initially introduced in Péré et al. (2018), was recorded to evaluate each algorithm. The process to calculate the KLC score starts by first getting the "ground truth" representation of the current repertoire. The ground truth is a hand-coded referential behavioural descriptor used as a common metric to compare every single algorithm. In the case of the ballistic task it is the x, y coordinate of the highest point the figurative projectile reached. A normalised histogram (a histogram where all the elements add up to one) of distributions with 30 bins per dimension is then generated. Finally, we calculate Kullback-Leibler Divergence from each point in the histogram to a reference distribution's normalised histogram.

$$KLC = \mathcal{D}_{KL}[E\|A] = \sum_{i=1}^{30} E(i) \log \frac{E(i)}{A(i)} \tag{3.4}$$

Where E is a reference distribution and A is the distribution that we are comparing to it. If we state that the generated behavioural descriptors are the compared distribution A, we then need to consider a reference distribution E. We define this as the same as it was in the original AURORA paper and generate a ground truth distribution to compare to. This was created by uniformly sampling 300 values from the minimum to maximum values of each of the genes of the genotypes, and then only adding each controller in this repertoire back into the population if they were novel enough in the ground truth space (the novelty threshold was set to a consistent 0.01, the same as the initial novelty for AURORA-AE incremental method). Figure 3.8 shows what this ground truth distribution looks like.

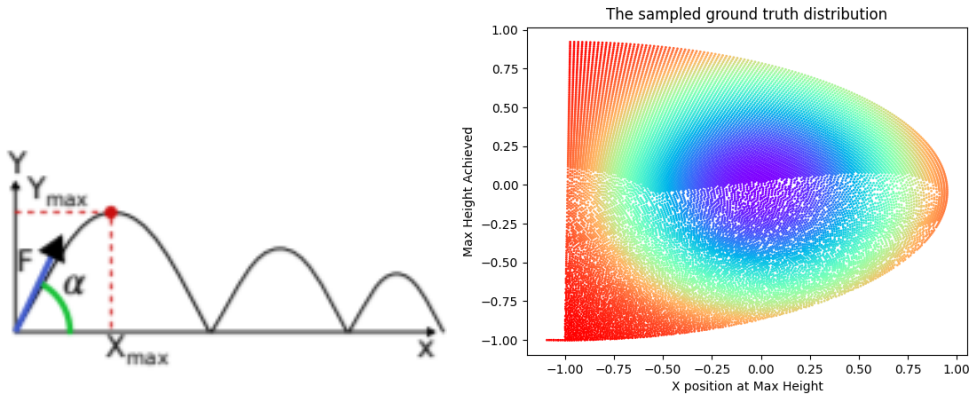

FIGURE 3.8: **Left:** from Cully (2019) Visualisation of what the ground truth is for the ballistic task. **Right:** The generated ground truth distribution that acts as the reference distribution for the KLC calculation

Finally, in order to test the implementation against itself the alternative version of AURORA-AE (AURORA with an autoencoder used as the dimension reduction algorithm) was also tested. The two versions of AURORA-AE are *incremental* and *pretrained*. Incremental is the version that has been described so far where the autoencoder is trained at specified intervals on the growing population. The other type, pretrained, consists of initialising a population with 10000 members (in contrast to the incremental version's initialisation size of 200 members) and proceeding to train the autoencoder once on this much larger dataset. The members of the population were generated in the same way that the ground truth was. But instead of 300 values uniformly sampled from the minimum to maximum, the pretrained version is trained on a distribution generated from 100 values uniformly sampled (hence why the population has 10000 members at training). The population is then cleared and reinitialised with 200 members and the network is never retrained. In the original paper the pretrained version performs differently than the incremental version, and is shown to have greater reconstruction loss, a worse Kullback-Leibler Coverage score and greater reconstruction loss.

3.1.4 Summary and User Guide

The pseudo-code summary for AURORA can be seen in Figure 3.9. The need to update the dimension reduction algorithm lessens as time goes on as less novel behaviours are recognised. When less novel behaviours are recognised quality becomes more important and the dimension reduction update becomes less necessary, so in Cully (2019) they manually set when the algorithm would update at a range of times measured by generation (decreasing in frequency). In the original paper and in this project the total number of generations was 5000 with 200 QD algorithm iterations per generation. The autoencoder was trained at generations 0, 50, 150, 350, 750, 1550 and 3150.

```
Function try_to_add(new_member, population):
    if new_member not novel then
        # Penalise curiosity score of parent of new member;
    else if new_member dominates then
        # Reward curiosity score of parent of new member;
        population[dominated_member] := new_member;
    else
        # Reward curiosity score of parent of new member;
        population.append(new_member);
    end
    return population;
# Initialise population with randomised controllers;
population := 200 × random_controller();
# Create the Autoencoder;
autoencoder = Make_AE();

for gen in NB_GENERATIONS do
    if gen is in RETRAIN_LIST then
        autoencoder := Train_Autoencoder(autoencoder, population);
        # Give all members of the population their new behavioural descriptors;
        population := assign_BDs(autoencoder, population);
        # Add novel members back to population;
        new_population := [];
        for member in population do
            new_population := try_to_add(member, new_population);
        end
        population := new_population;
    end
    for NB_QD_ITERATIONS do
        # Make curiosity score proportionate roulette wheel from population;
        wheel := make_wheel(population);
        parent := spin_wheel(wheel);
        child := mutate(parent);
        population := try_to_add(child, population);
    end
end
```

FIGURE 3.9: AURORA (Cully (2019)) algorithm pseudo-code for when the dimension reduction algorithm used is an autoencoder. This is the incremental version not the pretrained version.

For the sake of ease of use an argument parsing system was implemented into the control program. To run this algorithm you simply type in the command `$python3 control.py --version incremental`. Alternatively to run the pretrained version of AURORA-AE the command is `$python3 control.py --version pretrained`. In the following Extending with Recurrent Neural Networks section we cover an extension to this argument parser. The commands here remain unchanged as the default is set to deactivating this extension. For metric testing purposes the "hand-coded" and "genotype" algorithms were also recreated and implemented. These are effectively doing the Novelty Search method described in the Method section in different latent spaces not generated by an autoencoder. The hand-coded method uses as BD the literal ground truth, i.e. the x-y coordinate of the highest point achieved by that controller, and the genotype method uses the genotype used to control the robot as the BD. To run these the command line arguments are `$python3 control.py --version handcoded` and `$python3 control.py`

`--version genotype`.

3.2 Extending with Recurrent Neural Networks

The LSTM variants had the epoch limit set to 100 due to the intense computational load. This means that the pretrained version was trained for 100 epochs and the incremental version was trained using the same validation loss method explained in the Training the Autoencoder section, but instead of an epoch limit of 500 it was set to 100. The type of RNN chosen to extend the autoencoder were LSTMs. This is due to the fact that the datasets could get relatively large and, as stated in the Recurrent Neural Networks section, there is a rule of thumb to use LSTMs over GRUs when analysing large datasets. In that section we also stated that LSTMs concatenate the previous output and the new input vector into one vector $\mathbf{z}_t = (\mathbf{x}_t, \mathbf{y}_{t-1})$. The LSTM cell then generates multiple outputs according to the trainable weights and biases:

$$\begin{aligned} \mathbf{f}(t) &= \boldsymbol{\sigma}(\mathbf{W}_f \mathbf{z}_t + \mathbf{b}_f) & \mathbf{i}(t) &= \boldsymbol{\sigma}(\mathbf{W}_i \mathbf{z}_t + \mathbf{b}_i) \\ \mathbf{g}(t) &= \mathbf{tanh}(\mathbf{W}_g \mathbf{z}_t + \mathbf{b}_g) & \mathbf{o}(t) &= \boldsymbol{\sigma}(\mathbf{W}_o \mathbf{z}_t + \mathbf{b}_o) \end{aligned} \tag{3.5}$$

Where we refer to $\mathbf{f}$ as the forget gate, $\mathbf{i}$ as the input gate, $\mathbf{o}$ as the output gate and $\mathbf{g}$ as the gated memory. Using these we then update the hidden state $\mathbf{h}_{t-1}$ now referred to as the long term memory, in contrast to $\mathbf{z}_t$ which is now referred to as the short term memory.

$$\mathbf{h}_t = \mathbf{f}(t) \cdot \mathbf{h}_{t-1} + \mathbf{g}(t) \cdot \mathbf{i}(t) \tag{3.6}$$

Which we can then use to update our output:

$$\mathbf{y}_t = \mathbf{o}(t) \cdot \mathbf{tanh}(\mathbf{h}_t)) \tag{3.7}$$

We can now update Figure 2.12 with the position of where these functions are to give us a better understanding of how LSTMs work. This can be seen in Figure 3.10. In text prediction it is necessary to feed in either the full text, which could vary in length, or pass in sub-strings, which may not be long enough to represent sentiments. In AURORA the sensory information is a time series. Instead of flattening this immediately and passing it into the network, as is done in AURORA-AE, we instead encode the time series and pass that into the LSTM layer. All the encoded outputs of the LSTM layer are then decoded, flattened and passed into the autoencoder. Unlike standard LSTMs which are trained on the reconstruction loss between their output and the input, the LSTM was considered as a part of the autoencoder network, so the reconstruction loss it was trained on was the decoded output of the autoencoder.

An issue was the encoding method used. Embeddings would not really be applicable in this case as each data point is not high dimensional so there are no high level features that

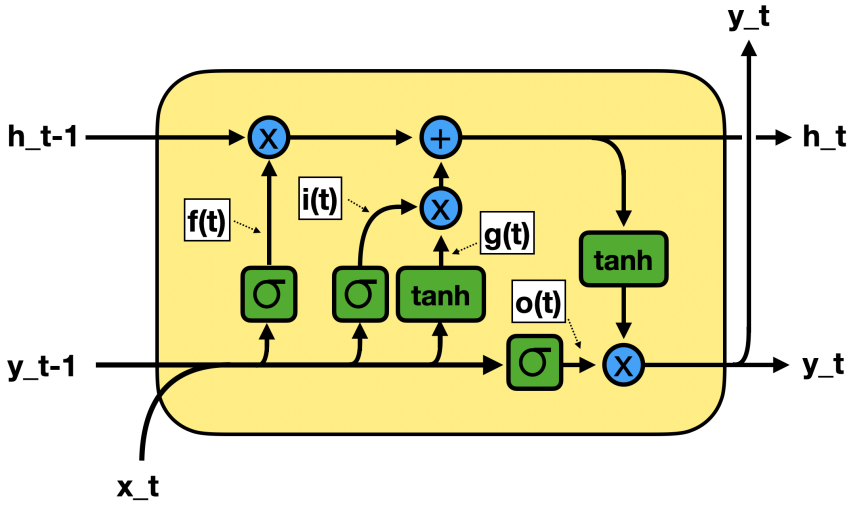

FIGURE 3.10: Visual representation of the calculations that occur in the LSTM with the locations of the output values.

can be extracted. Thus, though inefficient, one hot encoding needed to be used. This meant setting a dictionary size for the LSTM. This was done by rounding the values of the coordinates to a nearest determined value, encoding that output and passing it into the LSTM. The reverse was done to decode the outputs of the LSTM. Although initially the specificity was set high (to the nearest 0.0001), this resulted in low performance as each data point needed a massive vector. After testing, it was determined that values would be rounded to the nearest 0.0075. The number of hidden units for each LSTM cell was set to 128. The overall design of the extended autoencoder can be seen in figure 3.11.

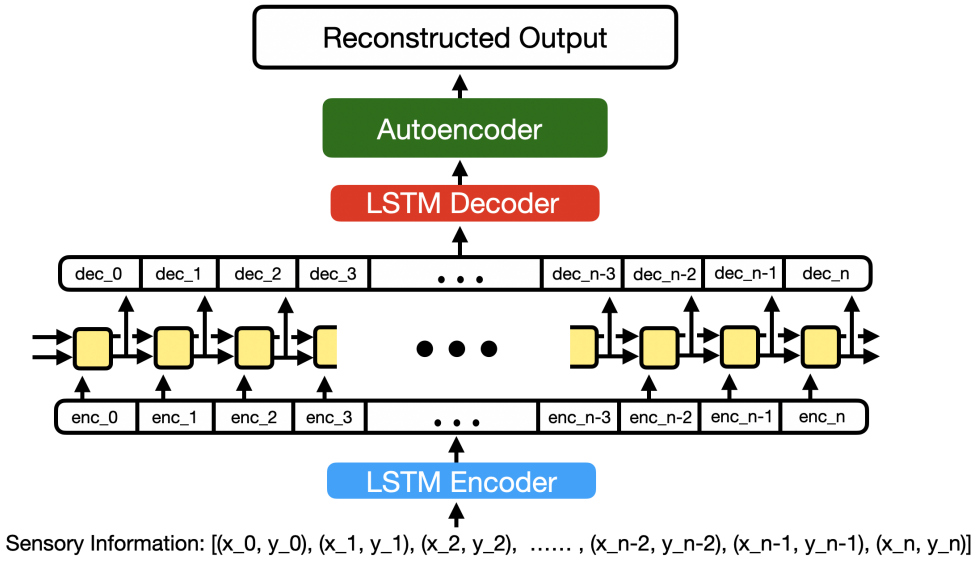

FIGURE 3.11: Design of the extended AURORA autoencoder with LSTM.

Chapter 4

Results

4.1 Recreating Original Results

We recall from earlier that in Figure 3.8 we displayed what the sampled ground truth distribution looks like. We consider this to be the best possible spread of different controllers with varying behaviour. You will notice that the axes both here and in every other ground truth plot have limits of -1 to 1. This is because the x, y coordinate is normalised according to the maximum possible values of x and y. This is done by taking the true ground truth coordinates, dividing them by 4000, and then subtracting each of them by 1.

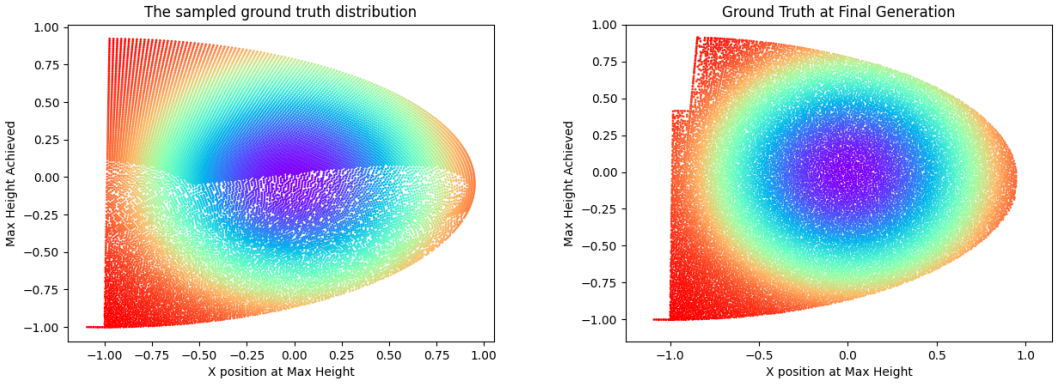

FIGURE 4.1: **Left:** The reference ground truth distribution used to calculate KLC score. **Right:** The ground truth distribution at the end of the final generation of the **hand-coded** method.

Figure 4.1 displays on the left the reference distribution first shown in 3.8. The right figure displays the repertoire of controllers at the end of the generations of the hand-coded method. To be absolutely clear the two plots here are not related to each other and were generated through different means. Each dot is the BD of a different controller. The colour for each controller is set by calculating how far away from the origin the BDs

are in ground truth space. This feature is helpful later when the latent space is no longer linear as the controller BDs in latent space will have the same colour as their ground truth equivalents in ground truth space. An example of this can be seen in Figure 4.2.

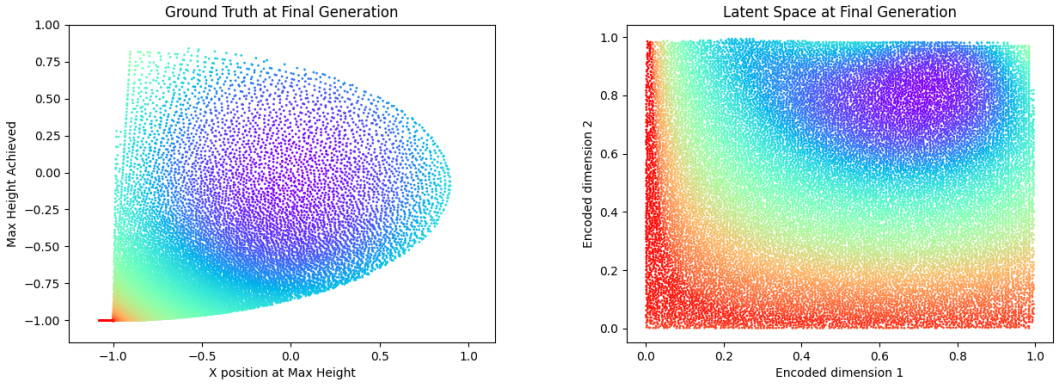

FIGURE 4.2: **Left:** The ground truth distribution at the end of the final generation of the **genotype** method. **Right:** The latent space distribution at the end of the final generation of the **genotype** method.

In Figure 4.2 the right image displays the latent space in which the QD algorithm searched for novel behaviours. The left image displays the corresponding BDs that these controllers generate in the ground truth space. Here we can clearly see that although the genotype method can fill the latent space very well there is an abundance of very similar behaviours that do not provide good results. An intuitive way to analyse these plots is to look at the amount of each colour in both plots. The more similar the amounts, the better the latent space.

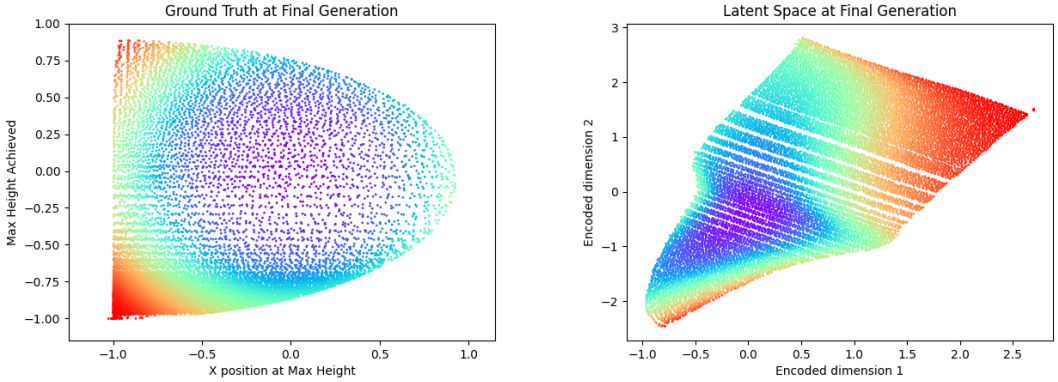

FIGURE 4.3: **Left:** The ground truth distribution at the end of the final generation of **AURORA-AE Pretrained** method. **Right:** The latent space distribution at the end of the final generation of **AURORA-AE Pretrained** method.

In Figure 4.3 we can see the result of running the pretrained version of AURORA-AE. As before the ground truth distribution is on the left and the latent space distribution is on the right. We can see how the density of controllers in the latent space inversely

translates to the density of controllers in the ground truth space. For example, in the latent space the darker blue and purple controllers are all densely packed together, whereas in ground truth space they are far apart. The density in the latent space would make it difficult to add a new controller to the population.

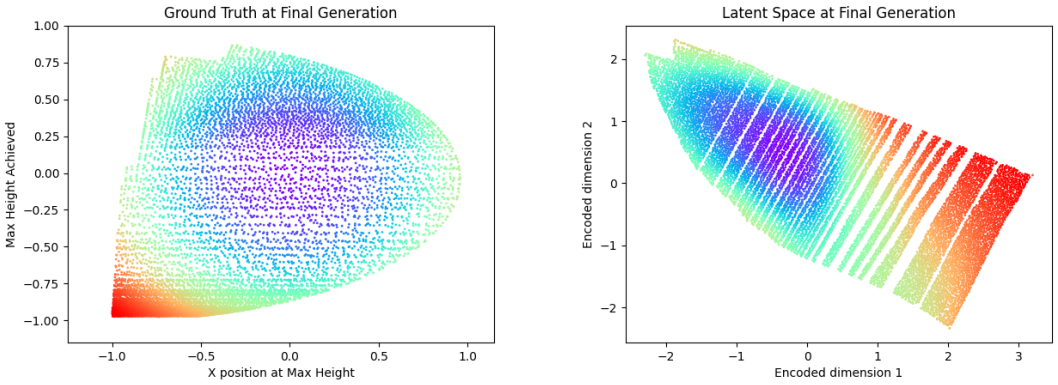

FIGURE 4.4: **Left:** The ground truth distribution at the end of the final generation of **AURORA-AE Incremental** method. **Right:** The latent space distribution at the end of the final generation of **AURORA-AE Incremental** method.

Figure 4.4 we can see the last of the original methods that were recreated. Specifically the incremental version of the AURORA-AE method. Looking at each of these plots individually is not an effective way of an analysis, and as covered in the Method we have already discussed which metrics we will use to judge the performance of each method. The first of these can be seen in Figure 4.5.

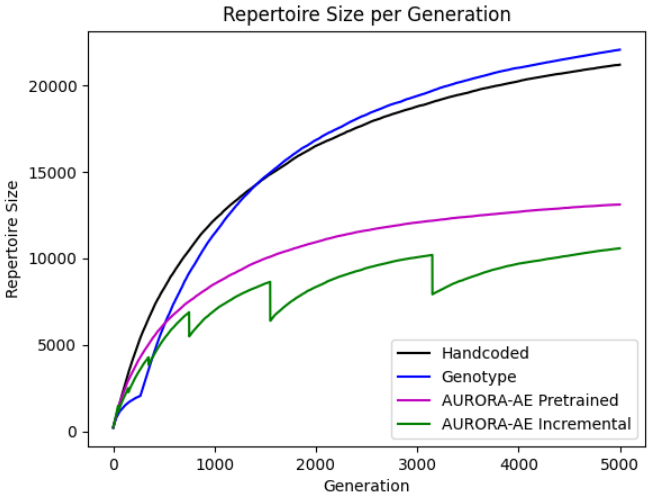

FIGURE 4.5: The repertoire size per generation of each of the implemented methods covered in the original paper.

This is the repertoire size per generation. Initially each of the algorithms should discover many novel behaviours and as they map out their respective latent spaces they should be plateau as less novel behaviours are discovered. The original paper displayed results that suggested that the pretrained version of AURORA-AE generates significantly less controllers than it's incremental equivalent. Though there were cases when the controller count was much less it was

often to do with the fact that the model had overfitted and had become incapable of managing out of sample data. It is likely that the minor error generated during the latter epochs of training in the original paper would have further trained the model and overfitted it. This would not be a problem in most models, but in the case of a QD algorithm searching for novel behaviours it is possible that the non-linear latent space was more difficult to explore because of this, and only the exact controllers that had initially trained the model would have distinct BDs. However, in the implemented version with less epochs for training (for the sake of computational efficiency) this was never observed, and although the pretrained version performed worse in other metrics it did not perform as badly as expected.

The second metric was the KLC score per generation. This was measured by creating by taking the ground truth distribution of the repertoire, creating a histograms along the x axis and the y axis, doing the same for the reference distribution and calculating the Kullback-Leibler Divergence of each point. These plots can be seen in Figure 4.6.

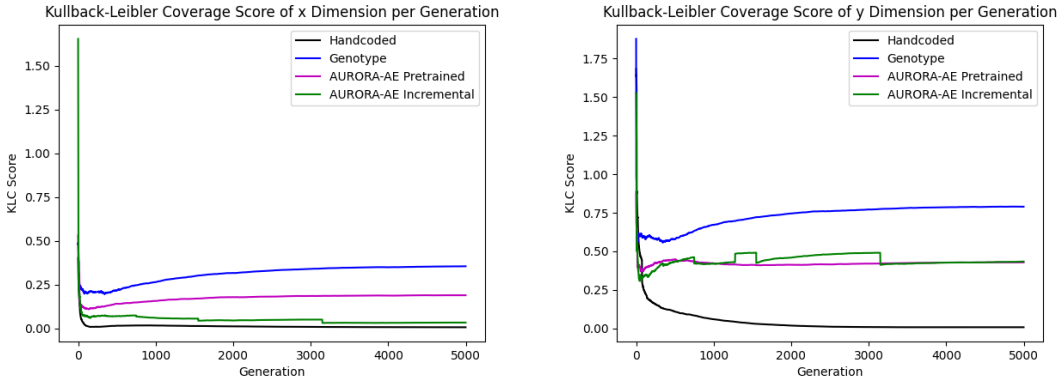

FIGURE 4.6: The KLC scores of the methods recreated from the original paper. **Left:** The KLC scores generated by comparing histograms of the x dimension. **Right:** The KLC scores generated by comparing histograms of the y dimension.

Though the KLC scores are distinct in the left hand plot that measures the KL-Divergence between the x-axis histograms this cannot be said for the right hand plot that measures the same but for the y-axis histograms. In order to get a view of the overall true KLC scores for the distributions the average per generation was calculated and plotted. This can be seen in Figure 4.7. The lower the value the more closely the distribution mirrors our reference distribution. We can see that the pretrained version is narrowly beaten by the incremental method, but on average this behaviour is too close to determine which method is superior.

Finally, Figure 4.8 displays some of the plots generated during the latent space exploration of the incremental version of AURORA-AE. The latent space will change shape when the autoencoder is retrained but overall the latent space is explored and diverse behaviours are found.

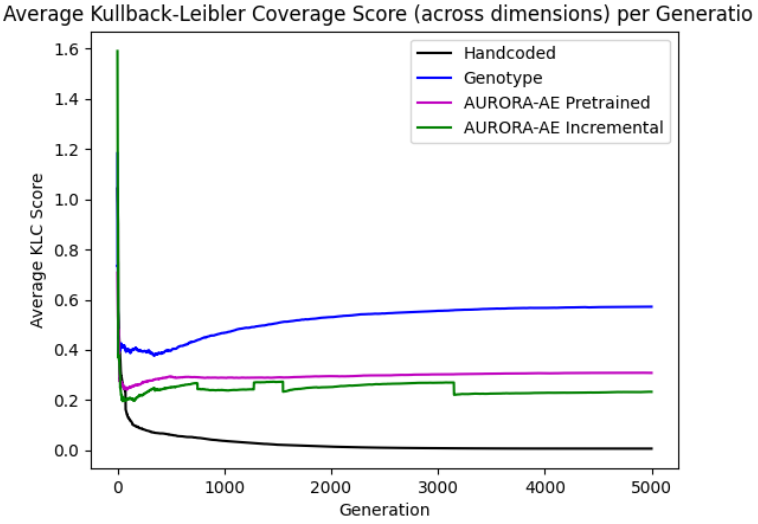

FIGURE 4.7: Average KLC scores per generation generated by averaging the KLC scores generated from comparing x and y dimensions.

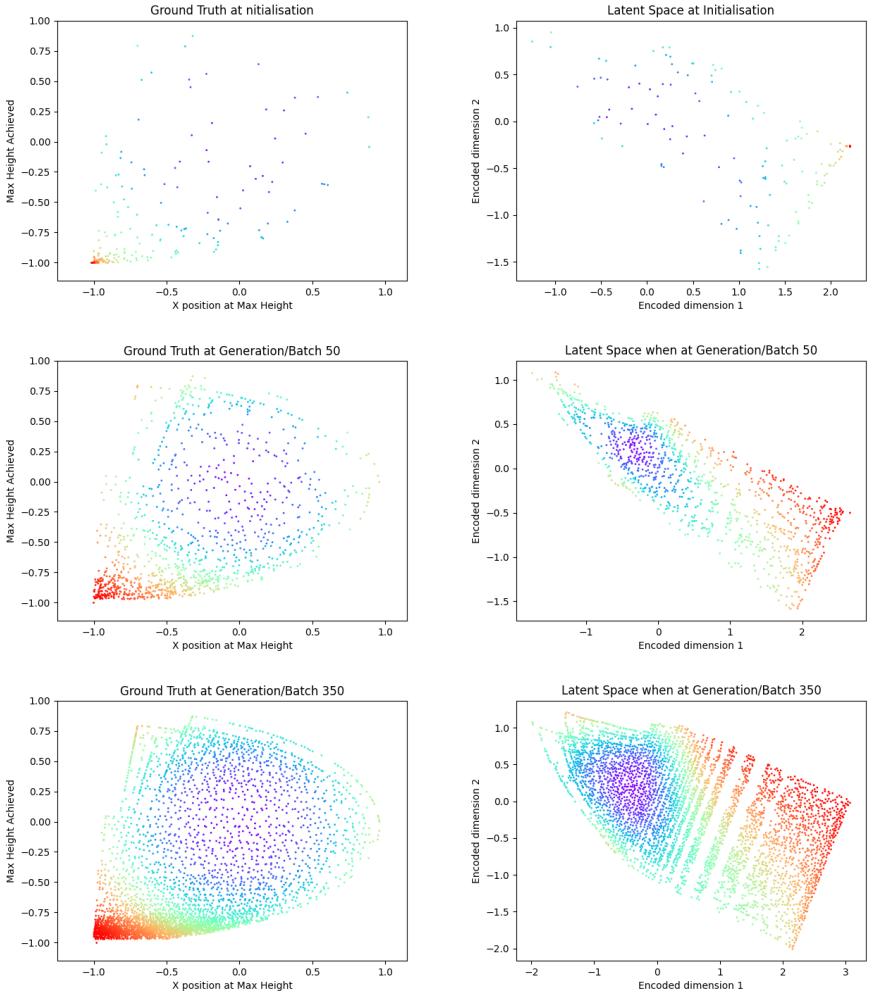

FIGURE 4.8: AURORA-AE incremental method, growing repertoire due to novelty search in the latent space and the corresponding growth in ground truth space. Taken at generations 0, 50 and 350.

4.2 Adding LSTM

The LSTM method was tested on both the pretrained and incremental versions. The result of running the pretrained method can be seen in Figure 4.9 Very clearly the

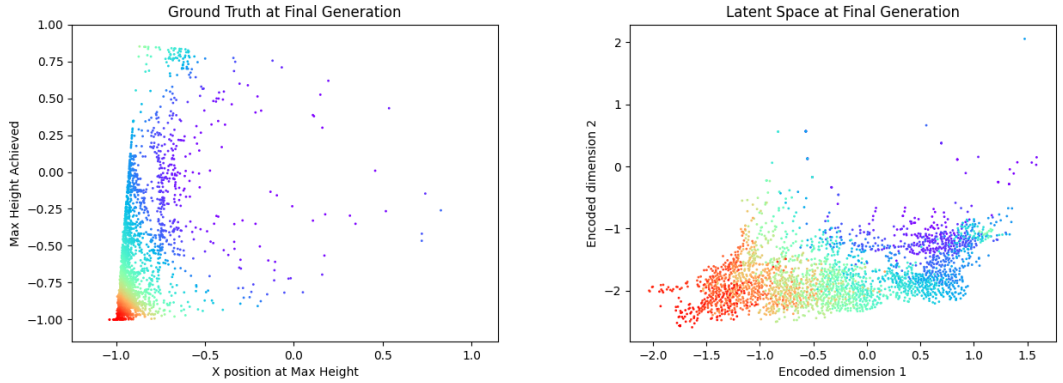

FIGURE 4.9: AURORA-AE-LSTM pretrained method, final ground truth and latent space. Repertoire size plateaued at around 2600 different controllers. Reflects the expected performance of AURORA-AE pretrained from the original paper.

exploration has not found a large variety of different controllers. However this may have been due to an undesirable initialisation of controllers making it hard to explore the possible options. Thus to test this an attempt was made to effectively map out the latent space. This was done by uniformly sampling the 150 values across the possible range of values for each gene and passing these values into the trained model and observe the resulting latent space. The result of attempting this can be seen in Figure 4.10.

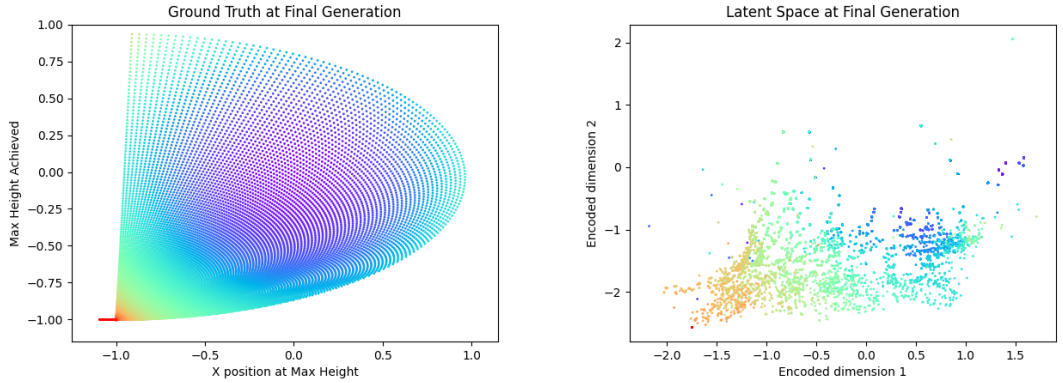

FIGURE 4.10: Attempt to fill the latent space of the AURORA-AE-LSTM pretrained method with a range of behaviours to illuminate effectiveness of behavioural descriptors.

Clearly, even with a diverse set of behaviours the latent space is incapable of mapping all of these out. When mathematically calculating how much of the latent space the controllers were able to cover it was found that, as could be guessed from initial observation, less area was covered by the diverse set of behaviours. This is surprising considering

how the pretrained method is trained on a diverse set of behaviours much like the one that we tried to fill it by.

In initial testing, a strange phenomenon was observed. Due to the lack of a better term we will refer to it here as the "donut effect". In some of the runs for the incremental version of the AURORA-AE-LSTM method it was found that an empty space was made around the center of the latent space, creating a sort of donut. Figure 4.11 displays examples of this effect.

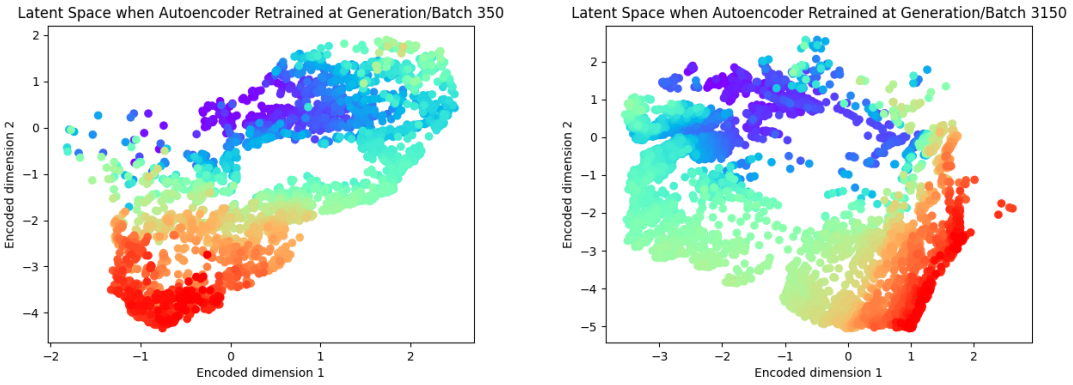

FIGURE 4.11: The "donut effect" in action. These two latent spaces were taken from two different AURORA-AE-LSTM incremental runs at early and late points in the run. Both display this effect of the empty space around the center.

It was theorised that this would be reserved space for either unobserved for impossible data. What is meant by impossible data here is that, due to the encoding method used, certain sequences of numbers will never be input into the LSTM layer. In the case of the ballistic task we can understand this by considering the ball that is being launched. The ball will always lose energy after its first bounce and will never reach a higher point than it achieves when first being thrown. Thus certain coordinates will never happen after other ones. But the network has no way of knowing this and may try to allow for them to exist in the latent space. To test this we took the trained

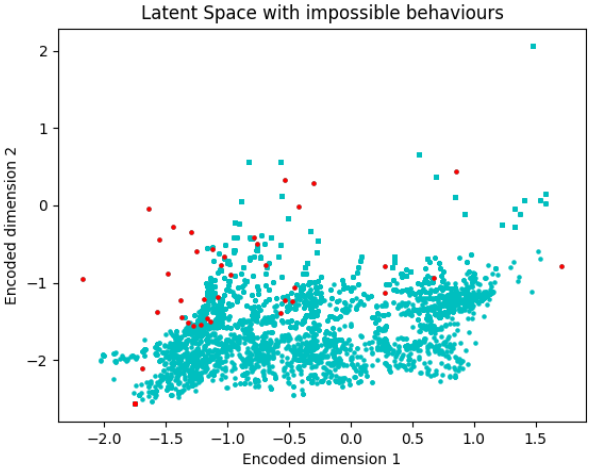

FIGURE 4.12: Observing where the impossible behaviours would be positioned in the latent space of the AURORA-AE-LSTM pretrained method. The cyan points are the data points that can be observed and the red points are those that can never be observed.

pretrained model that does display some of this donut effect, albeit on a smaller scale. This was done by taking some random trajectory image and shuffling the data points so that the coordinates were not in order. The result of this can be seen in Figure 4.12. Here the cyan points are the points that we forcefully input into the trained network, so they are the same as the points in right image in Figure 4.10. The red points are the shuffled trajectories.

The result appears initially unpromising until we consider that nearly all of the red points are positioned in un-mapped territory. That is to say that no controller has a BD in that point in the latent space. Yet this distribution of points is the same as shown in 4.10. This means that even though these points are impossible they exist much closer than they should logically be in the latent space. This means that the network is allowing for impossible data. Whether this is an advantage or not remains to be seen.

The last part of the extension was the incremental method that can be seen in Figure 4.13. Here we can see an example of the donut effect in the latent space as the points surround the blank space.

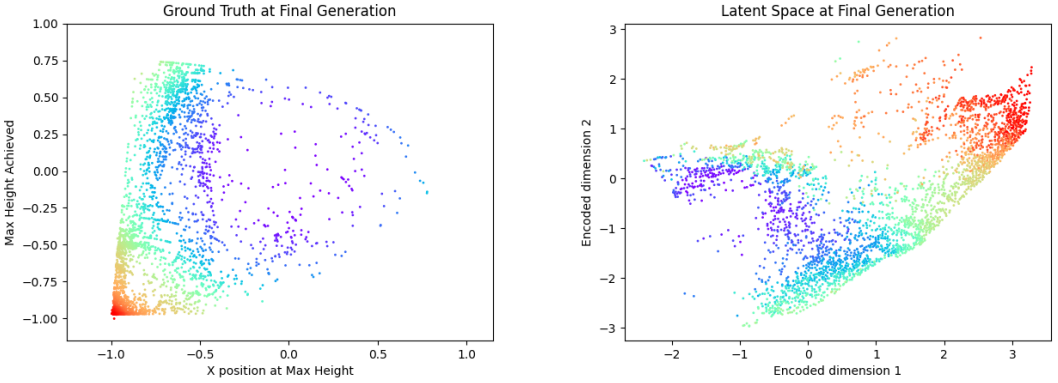

FIGURE 4.13: AURORA-AE-LSTM incremental method, final ground truth and latent space. Donut effect can be seen.

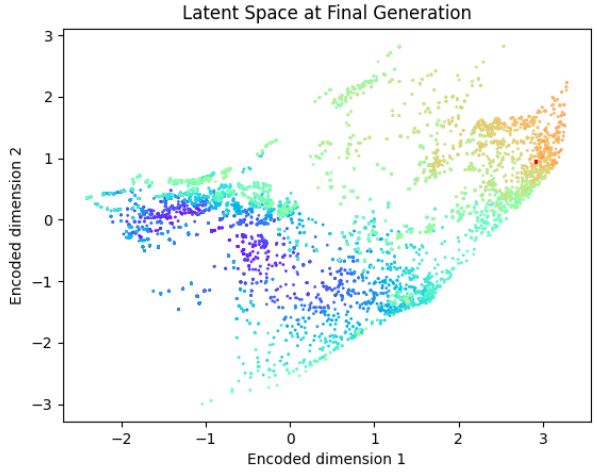

FIGURE 4.14: The latent space of AURORA-AE-LSTM incremental method filled with a wide range of repertoires.

The latent space was once again filled to observe as much as possible using the same distribution as shown in Figure 4.10. The result can be seen in Figure 4.14. As previously we note that less area than expected is covered and therefore the latent space itself is flawed. Using this latent space we also examined what happens when we take a controller's sensory information

and pass into the network both a shuffled and un-shuffled version. If the two resultant BDs are in distinct locations it would be confirmation of the network exploiting the time series aspect of the data. Figure 4.15 displays this. The cyan points are the mapped latent space. Some controllers have been chosen at random and have been assigned a different colour. This colour is assigned to both BDs, the shuffled and un-shuffled versions.

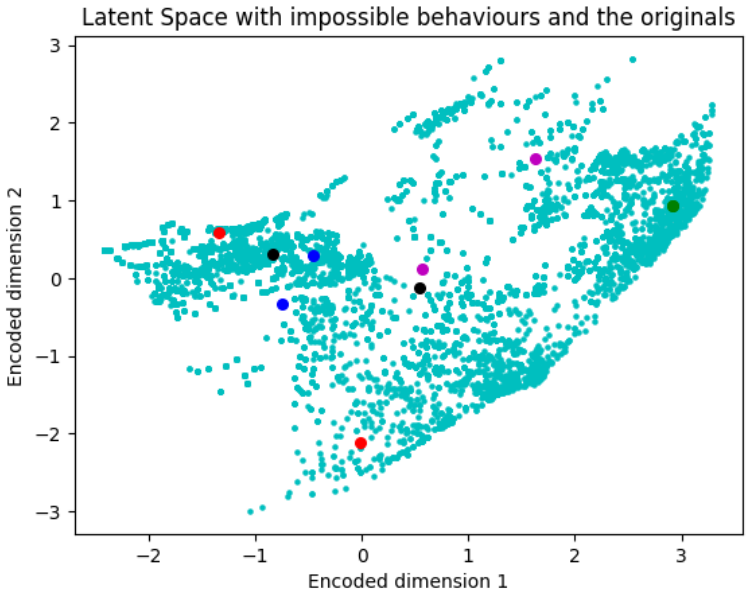

FIGURE 4.15: The filled latent space with some randomised trajectories fed into the network. Each colour that is not cyan, for example the two black dots, are assigned to a specific controller. One dot is the BD of the normal trajectory and the other is the BD of the shuffled trajectory

An immediate feature we can see is that there is only one green point. This could only be the case if either another point, itself included, overlaps the other green point completely. We can clearly see that both the randomised and true trajectories are fitting into the latent space easily. This is a negative as it shows that the time series aspect of the data is not in fact being exploited. If it was, there would be far more distinct behaviours from the randomised trajectories.

In order to fairly compare all the methods the standard AURORA-AE methods were run both with 100 epochs of training. As the hand-coded and genotype methods are not effected by the epoch limit they were not rerun. To truly examine whether or not the AURORA-AE-LSTM method's efficacy, the pretrained version was run with two variants. The first variant would run the pretrained version with more data. So instead of uniformly sampling 100 values per dimension, it would sample 170 values to create a dataset approximately 3 times the size. The second variant was run with the same training dataset as the original AURORA-AE-LSTM pretrained method, but was trained for double the number of epochs (so 200). For the methods that employ an autoencoder the average root mean squared error of sensory information reconstruction

per generation is also recorded and plotted. This RMSE plot can be seen in Figure 4.16 in the left hand image. This RMSE plot is hard to understand or analyse so on the right of this Figure we have taken a moving average across the last 50 generations. We can see that the error of both AURORA-AE-LSTM methods is significantly higher and they have much smaller repertoires. The repertoire size plot can both be seen in Figure 4.17.

FIGURE 4.16: The average RMSE per generation and the average of this error across the last 50 generations. Here the autoencoder algorithms were run for 100 epochs of training.

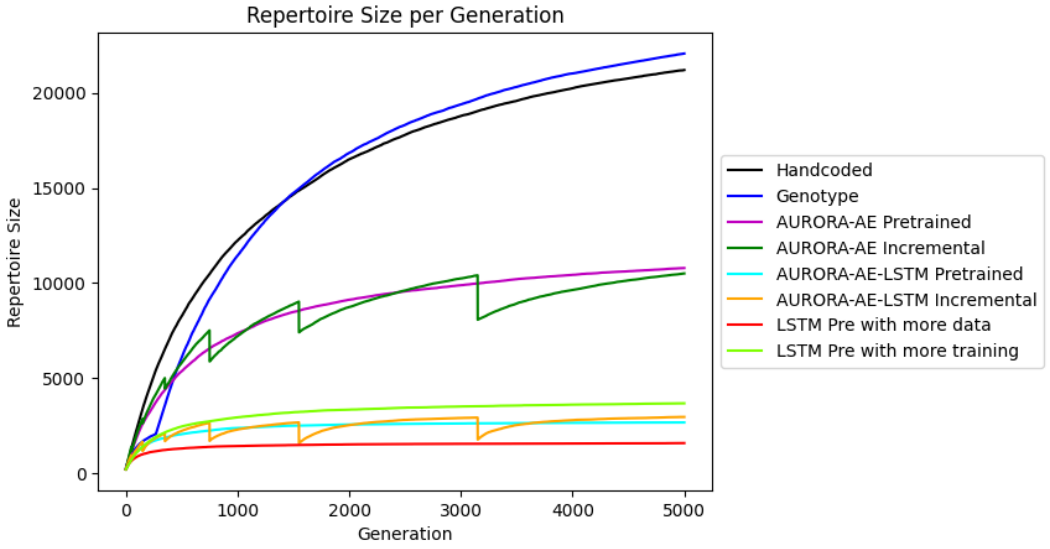

FIGURE 4.17: The repertoire sizes per generation when the autoencoder algorithms were run for 100 epochs of training.

The next metric to consider is the KLC score. KLC score plots and the average KLC score plot can be seen in Figure 4.18. It is hard to discern to discern meaning from these plots, so we implement a moving average across the last 50 generations (much like we did with the RMSE) to get a clearer image. This can be seen in Figure 4.19. From these plots we now have scientific confirmation that what we hypothesised when the ground truth of the LSTM methods was seen. Namely that AURORA-AE-LSTM does

not represent the ground truth well and has higher divergence than both of the standard AURORA-AE methods.

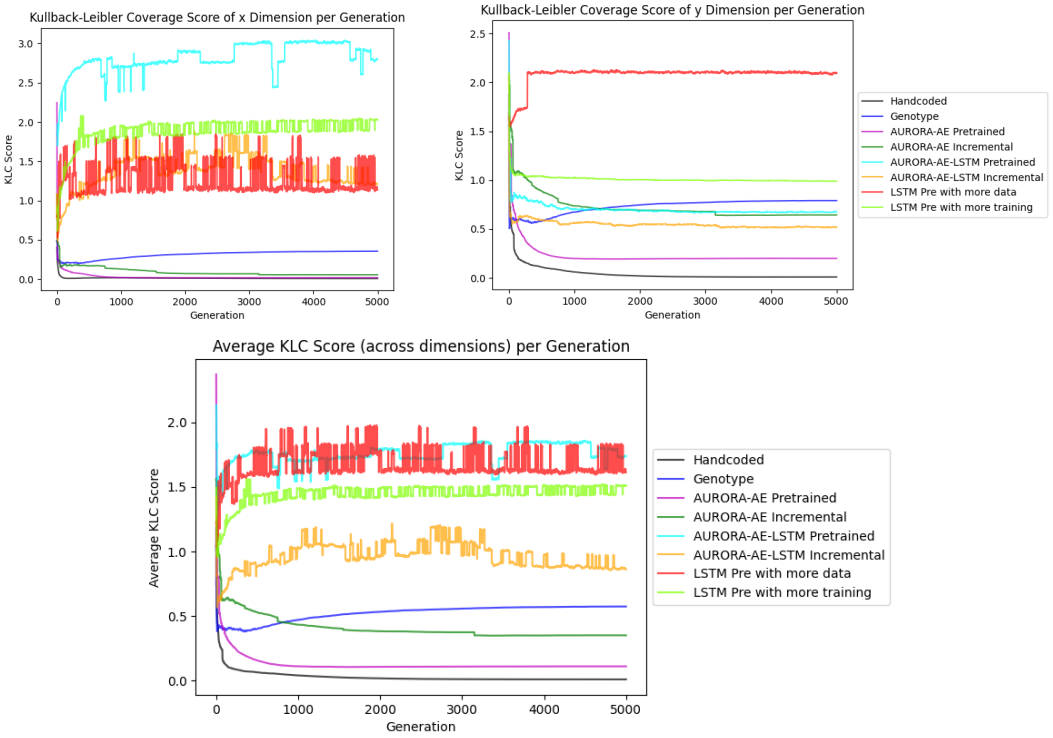

FIGURE 4.18: **Left:** The KLC scores per generation with respect to the x dimension. **Right:** The KLC scores per generation with respect to the y dimension. **Bottom:** The average KLC scores per generation across all dimensions. The epoch limit for the autoencoders was set to 100.

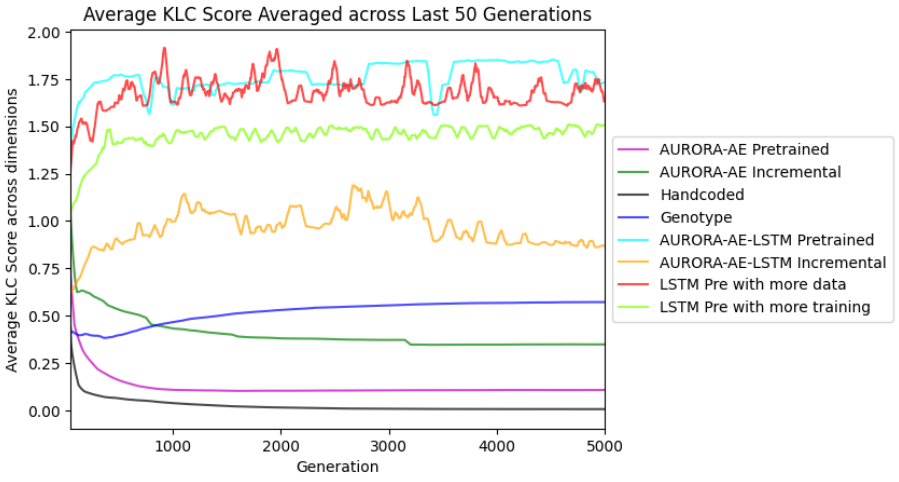

FIGURE 4.19: The average KLC scores per generation across all dimensions. The epoch limit for the autoencoders was set to 100.

Chapter 5

Discussion and Conclusions

From the various plots displayed in the Results section we can determine that the AURORA-AE-LSTM method implemented in the project performed worse than the original AURORA-Ae method, and from analysing the BDs in latent space when the trajectories were shuffled we can tell that the property we wanted to exploit in this project, the time series aspect, has not been fully explored. When testing whether this was due to the lack of a large dataset or training it was discovered that even with extended training times and larger datasets the performance did not improve. Interestingly it was consistently found that the original AURORA-AE's pretrained method performed admirably, often surpassing the performance of it's incremental counterpart. It is theorised that, due to the excessive number of epochs during training done in the original research, the model overfit to predict the exact distribution it had been provided with. This would not have been a problem if the latent space was linear, but it is likely that these specific points used to train the autoencoder became the only distinct points, and that many others would be grouped closer together in the latent space. Thus unless the exact point used to train the autoencoder was found it would not have a distinct BD. In this project, setting the number of epochs of training to be small has therefore proved itself an advantage. That being said, although the KLC divergence and repertoire size were better, the reconstruction error of the pretrained method was still larger than its incremental twin.

Previously it was considered whether or not VAEs would be a beneficial substitute to the standard autoencoder. It was thought that by sampling from the latent space means it would be possible to generate novel controllers. However, this would mean that it would be possible for a controller to be created that had no real world counterpart. On top of this, in the case of the ballistic task the sensory information was not a large enough matrix to warrant the use of a VAE.

The "donut effect" that was observed was found to not be reserving space for impossible sequences. If this is not the case then the only other logical alternative would be that

this is space that is reserved for input impossible to be entered into the autoencoder. This is because, due to the encoding method, only a dictionary of predetermined outputs are possible to be fed into the autoencoder. Whether this blank space is useful or not remains to be seen, as one could argue either that it makes the latent space less efficient as it is accounting for behaviours that can never and will never be observed, or it could be argued that this blank space makes it easier to add controllers to the repertoire improving the performance of the QD algorithm. In either case it would be necessary to adapt the encoding method used by the LSTM to allow for a wider range of output values.

Extending this of modifying the LSTM one could argue that there are other reasons why the method of encoding used in the LSTM could be the source of the implemented algorithm's poor performance. The current network is taking in one-hot encoded inputs that do not extract any feature information from the input data. Ironically, higher dimensional sensory information may perform better with AURORA-AE-LSTM by adding an embedding layer to encode this data. Furthermore, the principle notion of exploiting time series data could be achieved in other ways such as using network types that are known to be effective at analysing spatio-temporal data like Spiking Neural Networks (Ghosh-Dastidar and Adeli (2009)).

Overall, although the outputs of the AURORA-AE-LSTM methods were not as high performing as desired, there is still merit in attempting to exploit the time dependent information. Extensive further work is necessary to fully evaluate this, as well as the reasoning as to why the latent space in the AURORA-AE-LSTM methods result in the donut effect. What should be noted is that the incremental method generated this effect significantly more than the pretrained method, suggesting that it is to do with recognising that members in the population currently have distinct behaviours.

Bibliography

Alex M. Andrew. Behavior-based robotics by ronald c. arkin, with a foreword by michael arbib, intelligent robots and autonomous agents series, mit press, cambridge, mass., 1998, xiv+491 pp, isbn 0-262-01165-4 (£39.95; hbk). *Robotica*, 17(2):229–235, March 1999. ISSN 0263-5747.

Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. volume 27 of *Proceedings of Machine Learning Research*, pages 37–49, Bellevue, Washington, USA, 02 Jul 2012. JMLR Workshop and Conference Proceedings.

A. G. Barto, S. Singh, and N. Chentanez. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of International Conference on Developmental Learning (ICDL)*. MIT Press, Cambridge, MA, 2004.

David M. Bossens, Jean-Baptiste Mouret, and Danesh Tarapore. Learning behaviour-performance maps with meta-evolution. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO '20, page 49–57, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371285.

Herve Bourlard and Y Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59:291–4, 02 1988.

Keki Burjorjee. The fundamental problem with the building block hypothesis, 2008.

Konstantinos Chatzilygeroudis, Vassilis Vassiliades, Freek Stulp, Sylvain Calinon, and Jean-Baptiste Mouret. A survey on policy search algorithms for learning robot controllers in a handful of trials, 2018.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

Antoine Cully. Autonomous skill discovery with quality-diversity and unsupervised descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 81–89, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361118.

Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, May 2015. ISSN 1476-4687.

Antoine Cully and Yiannis Demiris. Quality and diversity optimization: A unifying modular framework, 2017.

Antoine Cully and Yiannis Demiris. Hierarchical behavioral repertoires with unsupervised descriptors. *Proceedings of the Genetic and Evolutionary Computation Conference*, Jul 2018.

Antoine Cully and Jean-Baptiste Mouret. Behavioral repertoire learning in robotics. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 175–182, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319638.

Kalyanmoy Deb and Debayan Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4:1–28, 02 2014.

Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

Carl Doersch. Tutorial on variational autoencoders, 2016.

Miguel Duarte, Jorge Gomes, Sancho Oliveira, and Anders Christensen. Evolution of repertoire-based control for robots with complex locomotor systems. *IEEE Transactions on Evolutionary Computation*, 22:314–328, 04 2018.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12: 2121–2159, 07 2011.

J. Elman. Finding structure in time. *Cogn. Sci.*, 14:179–211, 1990.

J. Firth. A synopsis of linguistic theory 1930-1955. In *Studies in Linguistic Analysis*. Philological Society, Oxford, 1957. reprinted in Palmer, F. (ed. 1968) Selected Papers of J. R. Firth, Longman, Harlow.

Dario Floreano and Claudio Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press, 2008. ISBN 0262062712.

Samanwoy Ghosh-Dastidar and Hojjat Adeli. Third generation neural networks: Spiking neural networks. In Wen Yu and Edgar N. Sanchez, editors, *Advances in Computational Intelligence*, pages 167–178, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03156-4.

Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

Xavier Glorot, Antoine Bordes, and Y. Bengio. Deep sparse rectifier neural networks. volume 15, 01 2010.

Richard Hahnloser, Rahul Sarpeshkar, Misha Mahowald, Rodney Douglas, and H. Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947–51, 07 2000.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

Will Koehrsen. wikipedia-data-science, 2018.

Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

Joel Lehman and Kenneth Stanley. Evolving a diversity of creatures through novelty search and local competition. pages 211–218, 01 2011.

Joel Lehman and K.O. Stanley. Revising the evolutionary computation abstraction: Minimal criteria novelty search. pages 103–110, 01 2010.

Elliot Meyerson, Joel Lehman, and Risto Miikkulainen. Learning behavior characterizations for novelty search. pages 149–156, 07 2016.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013a.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013b. Curran Associates Inc.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015.

Jeffrey Pennington, Richard Socher, and Christoper Manning. Glove: Global vectors for word representation. volume 14, pages 1532–1543, 01 2014.

Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016. ISSN 2296-9144.

Alexandre Péré, Sébastien Forestier, Olivier Sigaud, and Pierre-Yves Oudeyer. Unsupervised learning of goal spaces for intrinsically motivated goal exploration, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

Danesh Tarapore, Jeff Clune, Antoine Cully, and Jean-Baptiste Mouret. How do different encodings influence the performance of the map-elites algorithm? In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, page 173–180, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342063.

Ilya Tolstikhin, Olivier Bousquet, Sylvain Gelly, and Bernhard Schoelkopf. Wasserstein auto-encoders, 2017.

Vassilis Vassiliades, Konstantinos Chatzilygeroudis, and Jean-Baptiste Mouret. Using centroidal voronoi tessellations to scale up the multi-dimensional archive of phenotypic elites algorithm. *IEEE Transactions on Evolutionary Computation*, PP:1–1, 08 2017.

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*, ICML '08, page 1096–1103, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582054.