Evolutionary Reinforcement Learning for Vision-Based General Video Game Playing

by

Adam Tupper

under the supervision of

Dr. Kourosh Neshatian

A thesis submitted in partial fulfilment for the
degree of Master of Science

in the
Department of Computer Science and Software Engineering
College of Engineering
THE UNIVERSITY OF CANTERBURY

August 2020

THE UNIVERSITY OF CANTERBURY

Abstract

Department of Computer Science and Software Engineering
College of Engineering

Master of Science

by Adam Tupper

Over the past decade, video games have become increasingly utilised for research in artificial intelligence. Perhaps the most extensive use of video games has been as benchmark problems in the field of reinforcement learning. Part of the reason for this is because video games are designed to challenge humans, and as a result, developing methods capable of mastering them is considered a stepping stone to achieving human-level performance in real-world tasks. Of particular interest are vision-based general video game playing (GVGP) methods. These are methods that learn from pixel inputs and can be applied, without modification, across sets of games. One of the challenges in evolutionary computing is scaling up neuroevolution methods, which have proven effective at solving simpler reinforcement learning problems in the past, to tasks with high-dimensional input spaces, such as video games. This thesis proposes a novel method for vision-based GVGP that combines the representational learning power of deep neural networks and the policy learning benefits of neuroevolution. This is achieved by separating state representation and policy learning and applying neuroevolution only to the latter. The method, AutoEncoder-augmented NeuroEvolution of Augmented Topologies (AE-NEAT), uses a deep autoencoder to learn compact state representations that are used as input for policy networks evolved using NEAT. Experiments on a selection of Atari games showed that this approach can successfully evolve high-performing agents and scale neuroevolution methods that evolve both weights and topology to domains with high-dimensional inputs. Overall, the experiments and results demonstrate a proof-of-concept of this separated state representation and policy learning approach and show that hybrid deep learning and neuroevolution-based GVGP methods are a promising avenue for future research.

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Deputy Vice-Chancellor's Office
Postgraduate Research Office

Co-Authorship Form - Masters

This form is to accompany the submission of any thesis that contains research reported in co-authored work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from the extract comes:

The work conducted for and presented in Chapter 5.

Adam Tupper and Kourosh Neshatian. 2020. Analysing the Quality of Autoencoder-based State Representation Learning Techniques for Atari. [Unpublished Manuscript]

Please detail the nature and extent (%) of contribution by the candidate:

The research was conducted by Adam. He has written the paper. I provided supervisory guidance/feedback.

Certification by Co-authors:

If there is more than one co-author then a single co-author can sign on behalf of all

The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Masters candidate's contribution to this co-authored work
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text

Name: *Kourosh Neshatian* Signature: *Kourosh Neshatian* Date: *14 Aug 2020*

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Deputy Vice-Chancellor's Office
Postgraduate Research Office

Co-Authorship Form - Masters

This form is to accompany the submission of any thesis that contains research reported in co-authored work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from the extract comes:

The preliminary findings of the work conducted for and presented in Chapter 6.

Adam Tupper and Kourosh Neshatian. 2020. Evolving neural network agents to play atari games with compact state representations. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20). DOI: https://doi.org/10.1145/3377929.3390072

Please detail the nature and extent (%) of contribution by the candidate:

The research was conducted by Adam. He has written the paper. I provided supervisory guidance/feedback.

Certification by Co-authors:

If there is more than one co-author then a single co-author can sign on behalf of all
The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Masters candidate's contribution to this co-authored work
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text

Name: Kourosh Neshatian Signature: *Kourosh Neshatian* Date: *14 Aug 2020*

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Deputy Vice-Chancellor's Office
Postgraduate Research Office

Co-Authorship Form - Masters

This form is to accompany the submission of any thesis that contains research reported in co-authored work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from the extract comes:

The method proposed in Chapter 4 and the evaluations of the method presented in chapters 6-8.

Adam Tupper and Kourosh Neshatian. 2020. AE-NEAT: Autoencoder-Augmented Neuroevolution for Vision-Based Reinforcement Learning. [Unpublished Manuscript]

Please detail the nature and extent (%) of contribution by the candidate:
The research was conducted by Adam. He has written the paper. I provided supervisory guidance/feedback.

Certification by Co-authors:
If there is more than one co-author then a single co-author can sign on behalf of all
The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Masters candidate's contribution to this co-authored work
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text

Name: Kourosh Neshatian Signature: *Kourosh Neshatian* Date: *14 Aug 2020*

Preface

Publications

The preliminary findings for the experimental work presented in Chapter 6 were presented at the Genetic and Evolutionary Computation Conference 2020 (GECCO '20):

> Adam Tupper and Kourosh Neshatian. 2020. Evolving Neural Network Agents to Play Atari Games with Compact State Representations. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20).

We also have two further papers awaiting submission for publication at the time of completing this thesis. The first is a conference paper covering the findings of Chapter 5:

> Adam Tupper and Kourosh Neshatian. 2020. Analysing the Quality of Autoencoder-based State Representation Learning Techniques for Atari. Unpublished Manuscript.

and the second is a journal article covering the method proposed in Chapter 4 and the subsequent experimental evaluations presented in chapters 5-7:

> Adam Tupper and Kourosh Neshatian. 2020. AE-NEAT: Autoencoder-Augmented Neuroevolution for Vision-Based Reinforcement Learning. Unpublished Manuscript.

Funding

This work was supported by a UC Master's Scholarship that covered my tuition fees for duration of my thesis studies (a period of one year).

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Kourosh Neshatian, for your help and guidance throughout my research. I will be forever grateful for your willingness to engage in discussions, impart your wisdom, and provide advice. I have learnt much about the world of academic research from you and many skills that will no doubt be extremely useful in the future. Second, I would like to thank my family and friends for their support throughout the year. Your encouragement during the stressful times and the welcome distractions you have provided when you have sensed that I have needed a break have been immensely helpful. I would also like to thank my fellow students and the staff in the Computer Science and Software Engineering department for providing a friendly and supportive environment in which to conduct research. A special thank you must also be made to Pete, the department technical staff, and Francois from the UC Research Computing Cluster, who have been immensely helpful in helping me to acquire the compute resources that I needed to complete my research.

Contents

List of Figures

List of Tables

Abbreviations

AtariARI	**Atari A**nnotated **R**AM **I**nterface
AE	**A**uto**E**ncoder
AE-NEAT	**A**uto**E**ncoder-augmented **N**euro**E**volution of **A**ugmented **T**opologies
CNN	**C**onvolutional **N**eural **N**etwork
ERL	**E**volutionary **R**einforcement **L**earning
GVGP	**G**eneral **V**ideo **G**ame **P**laying
NEAT	**N**euro**E**volution of **A**ugmented **T**opologies
RL	**R**einforcement **L**earning
RNN	**R**ecurrent **N**eural **N**etwork
VAE	**V**ariational **A**uto**E**ncoder
WSSE	**W**eighted **S**um of **S**quared **E**rrors
β**-VAE**	Disentangled **V**ariational **A**uto**E**ncoder

Chapter 1

Introduction

The ability to learn from experience and through interaction with an environment is perhaps one of the first skills we think of when describing intelligent agents, such as ourselves. This trial-and-error style of learning is something that we humans apply to great success in learning many procedural and cognitive tasks, including learning how to walk, ride a bicycle, or play chess. For example, consider a child that is learning how to walk. They are acting in an environment, the real world, that they observe and that changes in response to their actions. They are trying to achieve some goal, usually in the beginning to move on their own from one parent to the other. Finally, they receive reward based on their progress towards completing the goal, in the form of encouragement and praise. These types of problems – where an agent must learn which actions to perform to maximise some reward – are sequential decision-making problems, commonly referred to as *reinforcement learning* (RL) problems.

Over the past decade, research on developing RL methods for artificial agents has quickly advanced from solving simple problems, using high-level features (observations) from the environment, to solving more complex problems in more complex environments. Thanks to the advancements in deep learning, the focus has quickly shifted to vision-based RL problems, where the agent must learn how to act appropriately based on images of the environment, rather than hand-crafted features. Whereas RL methods were once assessed on their ability to solve classical control problems, such as pole balancing (Geva and Sitte, 1993), we are now at the stage where methods are assessed on their ability to solve benchmarks that challenge human-level intelligence, such as video games.

Video games have become increasingly utilised tools for research in artificial intelligence, and perhaps the most extensive use has been as benchmark problems in the field of RL. Part of the reason for this is because video games are designed to challenge humans, and as a result, developing methods capable of mastering them is considered a stepping stone to achieving human-level performance in real-world tasks. In line with the desire to develop vision-based RL methods, of particular interest are general video game playing (GVGP) methods that learn from pixel inputs and can be applied across sets of games.

In the past, an effective method for solving simple RL problems was to use evolutionary algorithms to optimise a neural network, what is known as *neuroevolution*, that encodes an agent's strategy. Particularly effective were methods that optimise both the weights and topology (architecture) of neural networks (Stanley and Miikkulainen, 2002). However, while deep RL methods have allowed us to tackle harder RL problems, the large networks required to learn both state representation and policy using this approach limit the effectiveness and benefits of neuroevolution. This thesis explores a potential solution for scaling up topology and weight evolving neuroevolution methods to complex RL problems with high-dimensional inputs. Our specific focus is on vision-based GVGP. Our solution separates end-to-end learning into two components: state representation learning and policy learning. In the following section we discuss our motivation for pursuing this research, before then describing our aims and approach.

1.1 Motivation

Evolutionary reinforcement learning (ERL) methods have several properties which make them well suited to overcoming some of the difficulties faced by gradient-based deep RL methods. First, using neuroevolution we can evolve the topology of neural networks, something that cannot be optimised via gradient descent due to the lack of a differentiable loss function. Second, they allow us greater exploration of the policy space, and third, they do not suffer from problems that arise due to sparse rewards. We discuss each of these advantages below, as well as some key disadvantages faced by ERL methods that motivate our approach.

The ability to optimise both the weights and topology is one distinct advantage of neuroevolution that provides many benefits (Turner and Miller, 2013). Not only can

evolving the topology remove or reduce the effort spent on time-consuming architecture design, it also reduces human bias and can, as a result, lead to the discovery of more compact and creative architectures. Smaller networks are more memory efficient and are faster at inference time, which are particularly useful advantages for embedded RL applications, such as robotic control. They can also be trained more efficiently due to the reduced number of parameters that need to be optimised. The freedom afforded by optimising both the weights and topology also contributes to greater exploration of the policy space.

There are also several additional factors that contribute to the greater exploration of the policy space when using evolutionary algorithms. First, evolutionary algorithms evolve a population of solutions, which means that multiple strategies can be explored and optimised simultaneously. Second, they do not follow the gradient of the reward signal and therefore do not risk following it to locally optimal behaviour. When using gradient-based methods, it is common for the reward signal to at times mislead the agent and discourage them from learning behaviours that would eventually lead to an optimal policy (Such et al., 2017).

Finally, ERL methods are immune to the difficulties introduced when solving problems with sparse rewards. When feedback (reward) is given to the agent infrequently, after a long sequence of actions, the agent is faced with what is commonly referred to as the credit assignment problem (Sutton and Barto, 2018). The agent must identify which actions within the long sequence contributed most to receiving the reward. An example of such a situation is when the agent is only rewarded with a win or a loss at the end of a game. Gradient-based RL methods sometimes use shaped rewards that try to provide more guidance to alleviate the credit assignment problem, but this can often result in a policy that does not correspond to the problem we actually want the agent to solve (Such et al., 2017). In contrast, evolutionary methods search directly in the policy space and do not seek to learn the value of individual states or state/action pairs. The only thing that matters is the total accumulated reward at the end of the evaluation, and therefore the frequency with which the agent receives reward is inconsequential. Because of this approach, they avoid the credit assignment problem.

Despite the advantages described above, gradient-based deep RL methods are popular and successful in part because they are particularly good at optimising the large models

that are required for solving complex tasks with high-dimensional inputs. These large networks are able to both (a) extract features and learning good representations of the state of the environment, and (b) learn a policy that dictates how the agent should act. Scaling neuroevolution methods that evolve both the weights and topology to domains with high-dimensional input spaces is a key challenge in enabling evolutionary methods to compete with gradient-based methods in vision-based tasks. Existing methods suffer because there is a low limit on how fast evolution can proceed when optimising large networks. Only small changes to weights and topology can be made at a time to prevent breaking existing functionality between generations, yet it takes many such changes to realise a real difference in the performance of the task. Although several deep neuroevolution algorithms have been shown to be competitive with gradient-based approaches for some tasks (Salimans et al., 2017, Such et al., 2017), it is apparent that the increased size of the networks required for end-to-end learning limits their effectiveness.

The advantages and disadvantages discussed motivates our investigation into a separated state representation and policy learning approach. While the idea of separate state representation and policy learning is not in itself new, there is a lack of research around the generalisability of such methods, hence our focus on GVGP. There are also novel differences in the specific method we propose. Similar methods are discussed and compared in the literature review (Chapter 3).

1.2 Aim and Approach

This thesis proposes a novel method for vision-based GVGP that combines the representational learning power of deep neural networks and the policy learning benefits of neuroevolution. This is achieved by separating state representation and policy learning and applying neuroevolution only to the latter. Our method, AutoEncoder-augmented NeuroEvolution of Augmented Topologies (AE-NEAT), uses a deep autoencoder to learn compact state representations that are used as input for policy networks evolved using NEAT (Stanley and Miikkulainen, 2002).

Our overall aim is to provide a proof-of-concept that this separated state representation and policy learning approach can be used for GVGP. This goal is divided into three main objectives:

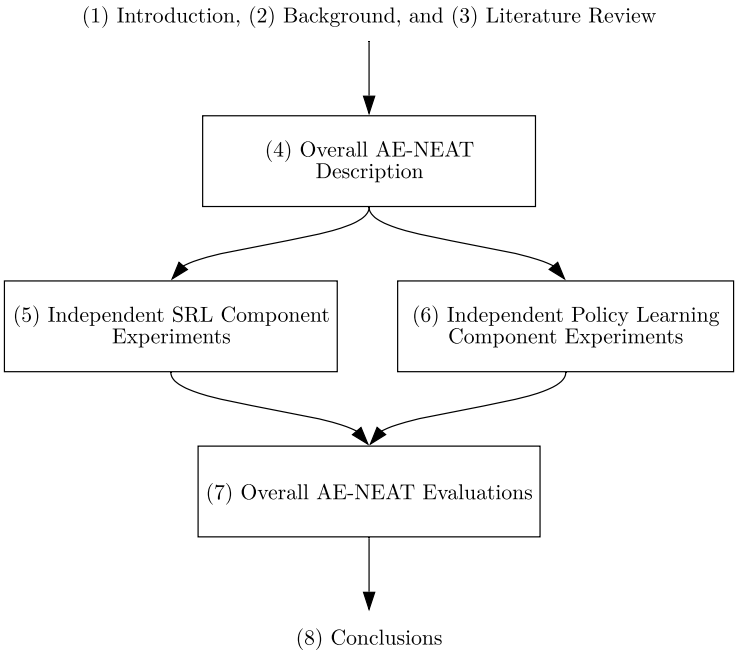


FIGURE 1.1: An overview of the relationship between chapters and the overall flow of the research and contributions of this thesis.

1. To identify an autoencoder-based state representation learning method for learning compressed state representations from images that generalises across multiple games.

2. To evaluate the applicability of using NEAT to evolve policy networks from compact state representations.

3. To develop a proof-of-concept that a separated state representation and policy learning RL method can scale neuroevolution to complex problems with high-dimensional inputs without the loss of generality.

1.3 Organisation

This thesis consists of eight chapters, including this chapter, the introduction.

Chapter 2 provides an overview of some of the fundamental components that this work builds on and utilises. The aim of this chapter is to provide some familiarity with key concepts, such as autoencoders and evolutionary algorithms, that underlie this work.

Chapter 3 discusses prior work in the area of evolutionary reinforcement learning, focusing on previous methods that have applied evolutionary-based methods to vision-based tasks. Here, we describe the differentiating factors of our work.

Chapter 4 introduces our proposed hybrid method of autoencoder-augmented evolutionary reinforcement learning, AE-NEAT, in greater detail. We also describe some of the high-level details common to all of our experiments and evaluations.

Chapters 5 and 6 describe the experiments performed to independently verify and evaluate the state representation and policy learning components used in AE-NEAT, respectively. Since these chapters focus on separate components and both feed into Chapter 7, they can be visualised as parallel investigations. This flow is depicted in Fig. 1.1.

Chapter 7 evaluates AE-NEAT (described in Chapter 4) as a whole, combining the knowledge acquired in chapters 5 and 6.

Chapter 8 summarises the work presented and contributions of this thesis. We draw conclusions based on our work and provide a discussion on potential avenues of investigation for future research.

Chapter 2

Background

The research presented in this thesis sits at the intersection of deep learning, evolutionary computing, and reinforcement learning. This chapter provides an overview of the key concepts and methods that underpin our work from each of these fields. While an effort has been made to arrange the sections as a progression of concepts, fully understanding the links between each concept may require multiple readings for readers unfamiliar with all three fields. In the following chapter, we discuss the literature directly related to vision-based evolutionary reinforcement learning.

2.1 Neural Networks

Neural networks (more formally known as *artificial neural networks*) are general function approximators that are loosely inspired by the structure of neurons and synapses in biological neural networks, such as the brain. They consist of neurons that are typically arranged as a sequence of interconnected layers. The first and last layers are labelled the input and output layers, respectively. The layers between these two outermost layers are referred to as hidden layers. An example of a neural network that represents a function with two inputs and a single output is illustrated in Fig 2.1. Neural networks with only forward connections between neurons are known as *feed-forward neural networks*.

The individual neurons within a neural network represent simple mathematical functions. Each neuron has one or more incoming connections (inputs), a bias, and an

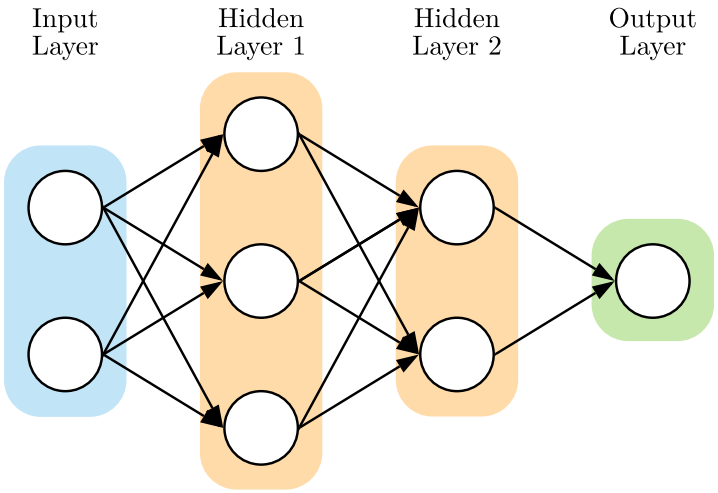


FIGURE 2.1: An example of a multilayered feed-forward neural network with two inputs, a single output, and two hidden layers.

output (or activation, as it is sometimes known). The output of each neuron a is a function of the weighted sum of the inputs $\boldsymbol{x}$ and the bias b:

$$a = g(b + \sum_{i=1}^{n} w_i x_i) \tag{2.1}$$

This is also depicted in Fig. 2.2. The weights $\boldsymbol{w}$ and bias are learnable parameters. The weights scale the inputs from the incoming connections and the bias is used to shift the output by a constant amount. The activation function g usually performs a simple nonlinear transformation, which allows the network to approximate arbitrary nonlinear functions. The *sigmoid* function is a commonly used activation function:

$$g(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Differentiable activation functions are used to enable the network to be trained using an approach called *back-propagation* (Rumelhart et al., 1986). Although a network with a sufficiently large single hidden layer is theoretically able to approximate any function (Hornik, 1991), multilayered *deep neural networks* are most commonly used in practice because they are easier to optimise.

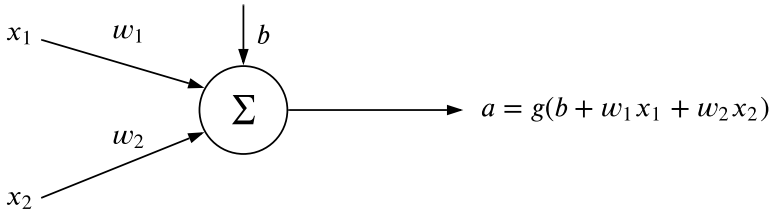


FIGURE 2.2: An example of how the output of a single neuron is calculated.

Neural networks are trained to minimise a loss function L that describes the error in the approximation of the true function f. An example of a loss function suitable for regression is the mean squared error (MSE) over a set of training examples:

$$L_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.3}$$

where n is the number of examples, and y_i and $\hat{y}_i$ are the true and predicted values for each example, respectively. The back-propagation algorithm uses gradient descent to optimise the learnable parameters $\boldsymbol{\theta}$ (weights and biases) of a neural network so that the loss is minimised. The algorithm iteratively calculates the partial derivatives $\frac{\delta L}{\delta \theta_j}$ of the loss function L with respect to each parameter θ_j. These partial derivatives tell us how quickly the loss changes as we change each parameter. With each iteration, the weights are updated by an amount proportional to $\frac{\delta L}{\delta \theta_j}$, as given by the equation from Rumelhart et al. (1986):

$$\theta_j \leftarrow \theta_j - \eta\frac{\delta L}{\delta \theta_j} \tag{2.4}$$

The magnitude of the weight updates is controlled by a hyperparameter η, the learning rate.

Back-propagation is an efficient training algorithm that has proven effective at training deep neural networks when combined with implementations that utilise graphics processing units (GPUs) for performing calculations. Before moving on, the next subsections briefly introduce two different extensions to neural networks: *recurrent neural networks* and *convolutional neural networks*.

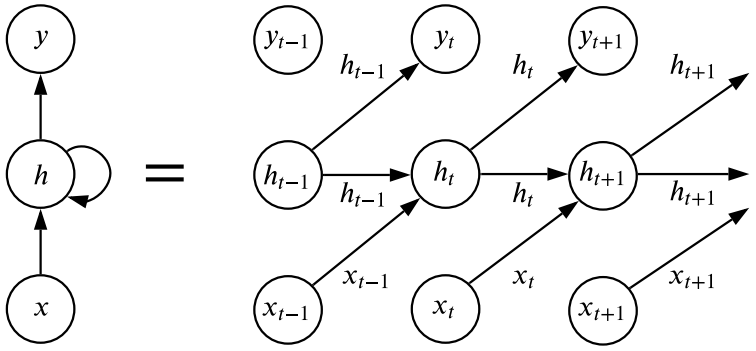


FIGURE 2.3: An example of the how the values of the inputs, hidden state, and outputs in a recurrent neural network are propagated over time.

2.1.1 Recurrent Neural Networks

The feed-forward neural networks described previously only contain connections that propagate the output of each node forwards. However, neural networks can also be designed that include backward or recurrent connections. Such networks are called recurrent neural networks (RNNs).

The advantage offered by RNNs is that nodes are able to consider values from previous time steps. As a result, they can utilise a form of memory and enables them to learn temporal functions and extract patterns from sequential data, such as text or video. In contrast to feed-forward networks, where the entire network from the inputs to the outputs is calculated in a single time step, with RNNs, there is a time delay. In an RNN, the output of each node is propagated forward one step at each time step. Figure 2.3 illustrates this behaviour.

RNNs are commonly trained using a modified version of the back-propagation algorithm, known as *back-propagation through time*. This training procedure "unfolds" the network, as shown in Fig. 2.3, and trains the larger resulting network. The details of this procedure are not covered because in this research RNNs are trained using evolutionary optimisation (covered in §2.3).

2.1.2 Convolutional Neural Networks

Another extension of neural networks is convolutional neural networks (CNNs) (LeCun et al., 1989). CNNs are used when the input data is arranged in a grid-like structure, such

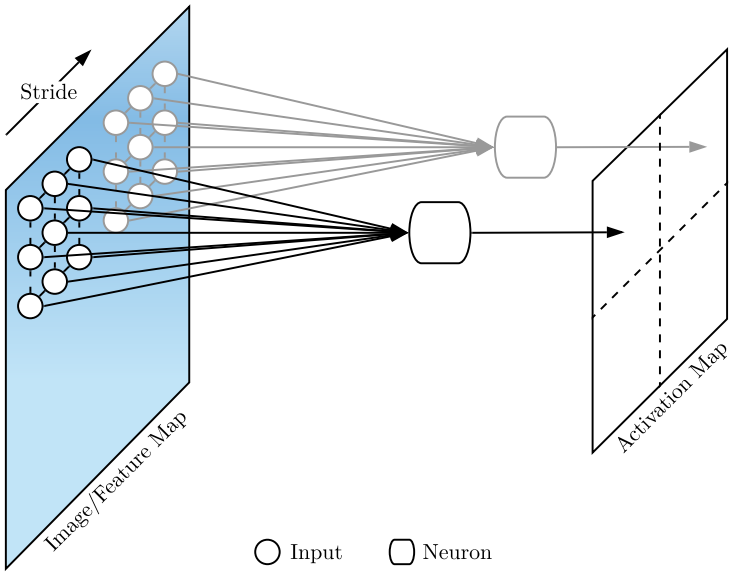


FIGURE 2.4: An example of how a single filter is implemented in a convolutional neural network.

as the pixel values of images. They are designed to take advantage of the assumed spatial relationship between the inputs to reduce the number of parameters in the network and make the processing of spatial data more efficient (Karpathy, 2015).

The backbone (the initial layers) of a CNN is typically composed of interleaved *convolution* and *pooling* layers. Convolution layers are used to extract spatial invariant features from the inputs, whereas pooling layers reduce the dimensionality. The final layer of the convolutional backbone is usually connected to a number of fully connected feed-forward and/or recurrent layers that process the extracted features for the desired task (e.g. classification).

To illustrate the benefits of using convolution layers over fully connected layers, imagine a single channel (e.g. grey scale) 100×100 pixel image. If we were to connect each pixel to a hidden layer with even just a single fully connected neuron, we would have $100 \times 100 = 10,000$ weights to train for this neuron alone. To avoid this, convolutional layers learn filters (otherwise known as kernels) that contain neurons that share weights and biases. Each neuron within a filter shares the same weights and bias, and simply applies the filter to a different part of the image. This has the effect of convolving the filter over the entire input. For example, a 3×3 filter applied to a single channel input has only 9 weights and 1 bias. As a result, each filter is used to identify the same

features, e.g. edges, in different parts of the image. An additional hyperparameter, the stride, determines how far the filter is shifted across the input. This idea is illustrated in Fig. 2.4. The resulting activation map created by a filter might be downsampled using a pooling layer, or serve as the input for a subsequent convolution layer. By stacking banks of filters (layers), we create deeper networks that can extract increasingly abstract features at each layer.

2.2 Representation Learning

An important component of our research is representation learning. This is the task of learning compact or compressed representations of data, such as images or text. In our case, we aim to learn compact representations of the game states of Atari games from the images that are ordinarily displayed on screen. To accomplish this, we focus on a particular class of representation learning techniques, *autoencoders*. In the following subsections, we provide an overview of the basic theory and implementation aspects of two different types of autoencoders that we investigate in our work: undercomplete and variational autoencoders.

2.2.1 Undercomplete Autoencoders

An autoencoder is a neural network that is trained to reproduce its inputs. However, rather than simply learning to memorise the training data, the aim is for the network to learn to extract useful features about the data. The simplest approach to achieve this is to introduce an *information bottleneck* in the network, a choke point that restricts the amount of information that can pass through the entire network. Autoencoders that rely on this bottleneck alone to force the network to learn which aspects of the data are important are known as *undercomplete autoencoders*.

An autoencoder consists of two halves; an encoder that learns a function $f(\boldsymbol{x})$ that maps an input $\boldsymbol{x}$ to a latent space encoding or compressed representation $\boldsymbol{z}$, and a decoder that learns a function $g(\boldsymbol{z})$ that maps an encoding $\boldsymbol{z}$ back to a reconstruction of the input $\hat{\boldsymbol{x}}$. As illustrated in Fig 2.5, we restrict the number of neurons in the encoding $\boldsymbol{z}$ to a value less than the dimensionality of the input. This causes the information bottleneck in the network.

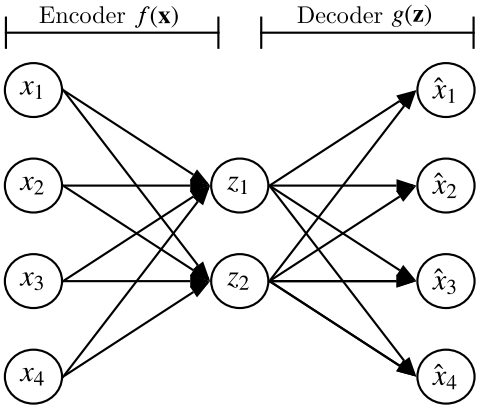


FIGURE 2.5: A simple undercomplete autoencoder. In the process of learning to reconstruction its inputs $\boldsymbol{x}$, it encodes the most important features of the data for achieving this in the encoding $\boldsymbol{z}$.

By forcing the network to prioritise what information is retained, the network is forced to learn only the features of the data that are most important for reconstruction. However, it is worth noting that the most important features for reconstruction may not always align with the most important features for policy learning. This is an issue that we tackle in Chapter 5.

Undercomplete autoencoders are trained solely to minimise the difference between the inputs $\boldsymbol{x}$ and the reconstructed outputs $\hat{\boldsymbol{x}}$. A commonly used loss function is the mean squared error, repeated below:

$$L_{MSE}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 \tag{2.5}$$

Because of the ability to learn nonlinear encoding and decoding functions, through the use of nonlinear neuron activation functions, undercomplete autoencoders are a more powerful nonlinear generalisation of Principal Component Analysis (PCA) (Goodfellow et al., 2016).

Although early autoencoders consisted of single-layer encoders and decoders (as shown in Fig. 2.5), deep autoencoders have been shown to offer better compression than shallow or linear autoencoders (Hinton and Salakhutdinov, 2006). Autoencoders have also been extended to use convolutional layers in place of fully connected layers, to reap the benefits of convolutional neural networks (§2.1.2). 2D convolutions allow the network to learn

spatial relationships in the data and require fewer parameters. This allows us to scale up to larger images. We use convolutional encoders and decoders in our work.

Although undercomplete autoencoders offer advantages other traditional representation learning techniques, it is still not guaranteed that they will perform well for our purpose. For this reason, we also investigate an extension to undercomplete autoencoders, *variational autoencoders*.

2.2.2 Variational Autoencoders

Variational autoencoders (Kingma and Welling, 2013) use a statistical approach for learning compact encodings. They assume that the training data is drawn from a distribution that can be parameterised by the latent variables $\boldsymbol{z}$. They attempt to learn a probability distribution for each latent variable, through a process called *variational inference*. This contrasts with undercomplete autoencoders that output a single value for each latent variable. When decoding, we sample from each distribution to generate a vector to serve as the input for the decoder. An advantage of this approach is that by learning a distribution for each latent variable, we force the encoder to learn a smooth, continuous latent space representation of the data, where similar observations should be located close to each other in the latent space. Our interest in variational autoencoders is that they may learn different encodings to undercomplete autoencoders that are better suited for policy learning.

2.2.2.1 Theory

Variational autoencoders formulate the problem of learning compressed representations as follows. We treat our training data $\boldsymbol{x}$ as a set of observations that are drawn from an underlying distribution that we can parameterise using a set of latent (hidden) variables $\boldsymbol{z}$. We can then describe the problem of learning the generating distribution, given our training data, using Bayes rule:

$$p(\boldsymbol{z}|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{p(\boldsymbol{x})} \tag{2.6}$$

However, we cannot compute the distribution of $p(\boldsymbol{z}|\boldsymbol{x})$ analytically, because to do so requires us to compute the distribution $p(\boldsymbol{x})$. Although $p(\boldsymbol{x})$ can be calculated via

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})dz \tag{2.7}$$

it is intractable do so, due to the size of the search space of $\boldsymbol{z}$.

This is where variational inference comes in. Instead of computing $p(\boldsymbol{z}|\boldsymbol{x})$ analytically, we try to approximate it using a distribution $q(\boldsymbol{z}|\boldsymbol{x})$ that we restrict to a family of Gaussian distributions, i.e. each latent variable z_i is modelled by a Gaussian distribution. We can use Kullback Leibler (KL) divergence (Kullback, 1997) to measure the distance between the approximation and the true distribution to find the optimal approximation $q^*(\boldsymbol{z}|\boldsymbol{x})$:

$$q^*(z|x) = \min_{z \in \boldsymbol{z}} KL(q(z|x)||p(z|x)) \tag{2.8}$$

Training the variational autoencoder then becomes an optimisation problem, in which we optimise a loss function consisting of the reconstruction error and the KL divergence:

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \sum_j KL(q_j(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})) \tag{2.9}$$

2.2.2.2 Implementation

Since variational autoencoders learn a distribution for each latent variable, the information bottleneck is implemented in a slightly different way to the undercomplete autoencoder shown in Fig 2.5. Instead, the bottleneck is implemented as shown in Fig 2.6.

The bottleneck is separated into two vectors: one that encodes the mean of each hidden variable, μ_i, and another that encodes the variance, σ_i^2. These two parameters allow us to encode a Gaussian distribution for each hidden variable. Unlike a true multivariate Gaussian, we make a simplifying assumption that each distribution is independent, which allows us to encode the variances in a vector.

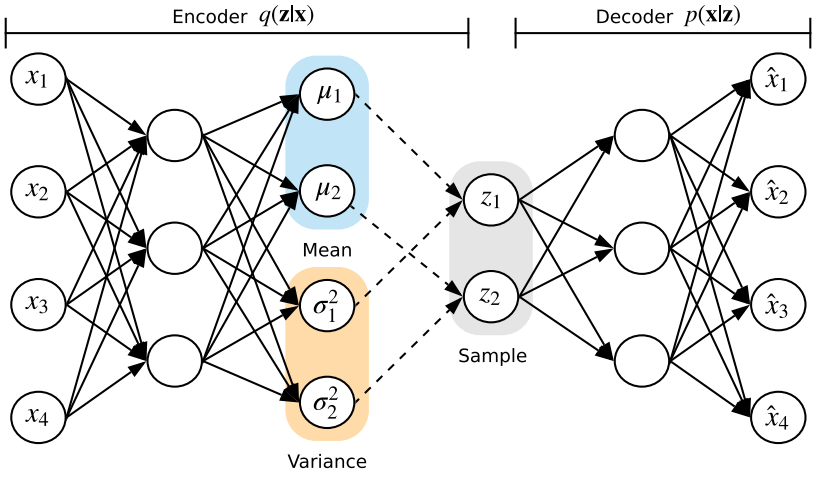


FIGURE 2.6: A simple variational autoencoder.

To ensure the network is trainable, variational autoencoders use what is referred to as the "reparameterisation trick" (Kingma and Welling, 2013). Since we must be able to compute the derivative of each parameter with respect to the loss to train the autoencoder using backpropagation, the sample from the distributions encoded by $\boldsymbol{\mu}$ and $\boldsymbol{\sigma^2}$ is drawn by drawing a value from a standard Gaussian distribution and shifting the mean and scaling variance by this amount.

2.2.2.3 Disentangled Variational Autoencoders

Finally, we also investigate the use of an extension to variational autoencoders, *disentangled variational autoencoders* (Higgins et al., 2017a). Disentangled variational encoders introduce a parameter $\beta > 1$ that assigns a higher weight to the KL divergence term in the loss function:

$$L(\boldsymbol{x}, \hat{\boldsymbol{x}}) + \beta \sum_j KL(q_j(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})) \tag{2.10}$$

Through greater penalisation of the difference between the distributions, we place a larger emphasis on enforcing our simplifying assumption that the distributions of each latent variable are uncorrelated and follow independent Gaussian distributions. This has the effect of ensuring that each latent variable encodes a different attribute in the data. Prior work on the use of disentangled variational autoencoders in reinforcement

learning environments has shown the benefits of state representations where each latent variable encodes a different property of the environment (Higgins et al., 2017b). This may simplify the encoding and make it easier for policy learning.

2.3 Evolutionary Algorithms

Evolutionary algorithms are a class of black box optimisation techniques inspired by Darwin's (1859) theory of evolution. They evolve a *population* of candidate solutions using operators based on biological evolution mechanisms including *mutation*, *crossover*, and *selection*. Candidate solutions are evolved iteratively in *generations*, within which each candidate's fitness is evaluated and used to assess their performance towards some goal. Therefore, the goal of an evolutionary algorithm becomes to evolve solutions which maximise fitness and thus performance. Individuals with the highest fitness are usually more likely to survive each generation and be used to create offspring to populate the next generation. Through this mechanism, the search is guided towards areas of the search space of higher fitness, though maintaining a population also allows for broad exploration and helps to prevent finding locally optimal solutions.

2.3.1 Classes of Evolutionary Algorithms

For brevity, we only introduce three main classes of evolutionary algorithms which have been utilised in the prior work on evolutionary reinforcement learning discussed in our literature review (Chapter 3). Evolutionary algorithms themselves are part of a family of wider biologically inspired algorithms under the umbrella of evolutionary computing, such as swarm intelligence algorithms. It is worth noting that while initially these classes of algorithms may have developed independently, there is now a substantial blur in how the terms and definitions are used today.

2.3.1.1 Genetic Programming

Genetic programming algorithms are methods for evolving programs or functions that were first published in the 1980s (Cramer, 1985). Initially, the population is seeded with randomly generated programs. Traditionally, programs are represented as syntax

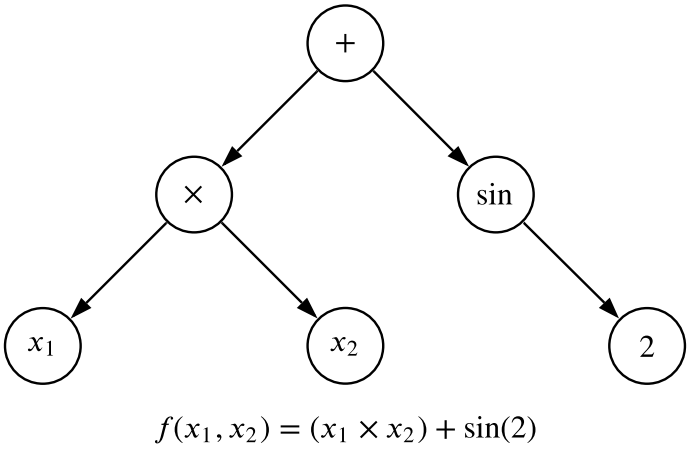


FIGURE 2.7: An example of a function represented by an abstract syntax tree.

trees that are evaluated recursively, as shown in 2.7. Using a syntax tree representation, mutation may take the form of changes to node operators or values, or additions to or deletions from the tree. Crossover operations might include switching subtrees between parents.

2.3.1.2 Evolution Strategies

Evolution strategies (ES) (Rechenberg, 1973) is an algorithm for real-valued function optimisation. Specifically, the algorithm seeks to optimise a vector of real values $\boldsymbol{\theta}$ that parameterise some function f. For example, these parameters could be the weights and biases of a neural network.

The ES algorithm starts with a random set of parameter values. In its simplest implementation, a population of N individuals is created by producing N random perturbations of $\boldsymbol{\theta}$. Each random perturbation is created by applying a small amount of Gaussian noise with a mean of zero and some standard deviation σ (De Jong, 2006) to each parameter, i.e.

$$\theta_i \leftarrow \theta_i + \epsilon_i \tag{2.11}$$

where $\epsilon_i \sim \mathcal{N}(0, \sigma)$. Each individual is then evaluated, and an updated parameter vector $\boldsymbol{\theta}'$ is defined as the weighted sum of the perturbed parameter vectors, where each

weight is proportional to the fitness of the individual. This process repeats, iteratively optimising the parameters of the function. Both vanilla ES and a popular extension, Covariance Matrix Adaption ES (CMA-ES) (Hansen and Ostermeier, 1996), are used in evolutionary reinforcement learning methods covered in the literature review.

2.3.1.3 Genetic Algorithms

Genetic algorithms developed as application-independent evolutionary algorithms, unlike evolution strategies, which was designed for real-valued function optimisation, and genetic programming, which was designed for evolving programs. The simplest form of genetic algorithms encodes genomes as a fixed-length binary string. Using this representation, mutations to parents can be performed as random bit flips of genes, and crossover operators defined as swapping substrings of each parent. Two simple crossover operators are 1-point and 2-point crossover. For 1-point crossover, a single point in the genome is chosen at random, and the substrings of each parent genome after this point are swapped. For 2-point crossover, two points in the genome are chosen, and the genes between these two points are swapped. The specifics of the genome encoding scheme depend on the application, i.e. the solution (or *phenotype*) that the genomes encode. The genome encoding also influences the mutation and crossover operators that are applied. In the next section, we describe a specific genetic algorithm for optimising neural networks.

2.4 Neuroevolution and the NeuroEvolution of Augmenting Topologies (NEAT) Algorithm

Neuroevolution is the term used to describe the use of evolutionary algorithms to optimise neural networks. In our work, we focus on the use of one neuroevolution method, NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002), that optimises both the weights and topology (i.e. architecture) of neural networks. It has been shown to be a powerful and popular algorithm that has been used for many applications. There have also been many extensions proposed for NEAT that expand the usefulness of the algorithm to other applications. These include HyperNEAT (Stanley et al., 2009), FS-NEAT (Whiteson et al., 2005), CoDeepNEAT (Miikkulainen et al.,

2019) and rtNEAT (Stanley et al., 2005), among many others. Since NEAT is the algorithm that we use to evolve policy networks for Atari games, it is important that we provide an overview of how NEAT operates.

NEAT is designed to evolve neural networks from minimal structures. The initial population consists of minimal networks that contain only input, bias, and output nodes. From this starting point, the networks are gradually optimised and complexified through mutation and crossover. By starting minimally, NEAT is always searching through fewer dimensions than other topology and weight evolving, and purely weight evolving (i.e. fixed-topology) algorithms, which offers a performance advantage. There are several important features of NEAT that enable the efficient evolution of solutions. These are discussed below. However, NEAT is a complex algorithm with too many components and details to discuss in detail here. For a full description of the NEAT algorithm, readers are directed to Stanley and Miikkulainen (2002).

2.4.1 Genome Encoding

NEAT uses a direct encoding scheme to encode the genomes. A direct encoding scheme directly encodes the parameters of the neural network. Genomes consist of node and connection genes. Each connection gene contains the input and output nodes for that connection, the connection weight, a Boolean value that indicates whether or not that connection is enabled, and an innovation number. The node genes specify the type of each node (input, bias, or output) and the activation function for each node.

NEAT encodes networks using variable length genomes that differ in length depending on the number of connections and nodes in each network. As the genomes grow, it becomes difficult to identify matching genes, which is crucial for encouraging productive offspring to be produced during crossover. Even in small networks, randomly crossing over genes results in large numbers of degenerate offspring and slows down the search. To alleviate this issue, each time a new connection is added it is associated with an *innovation number*. A global record of each mutation (identified by an innovation number) is kept so that when an identical mutation occurs down the line, it can be matched up accordingly.

2.4.2 Speciation

Motivated by the fact that networks need time to make use of new structural additions, NEAT speciates the population to protect topological innovations. During the reproduction phase of the algorithm, each species is allocated a certain number of offspring relative to the fitness of that species. This means that individuals only have to compete within their niche, rather than against others that have had more time to mature and be optimised. To accomplish this, genomes are grouped into species using a *genomic distance* function and a distance threshold. If the distance between two genomes is less than the threshold, they are considered members of the same species and compatible for crossover.

The genomic distance δ used in NEAT is specified in the equation below:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{2.12}$$

This is a function of the number of excess (E) and disjoint (D) connection genes and the average weight difference ($\overline{W}$) between the two genomes. The coefficients c_1, c_2, and c_3 are hyperparameters that specify the relative importance of each of these components in the distance between the genomes. The term N can be used to normalise the disjoint and excess gene terms by setting it equal to the size of the larger of the two genomes, but in practice it is often set to one.

2.4.3 Reproduction

The population of networks is evolved through a combination of mutation and crossover. After the fitness evaluations of networks in the current population have been performed, each species is assigned a number of offspring based on the fitness of its members. We explain the mutation and crossover operators used to produce the next generation of individuals below.

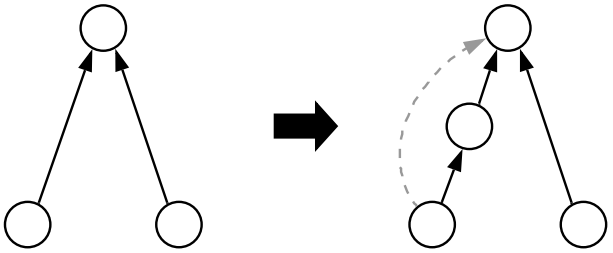


FIGURE 2.8: A visualisation of the *add node* mutation operator for NEAT

2.4.3.1 Mutations

To create offspring, NEAT performs structural and weight mutations. To evolve the topology of the networks there are two structural mutations that are performed in NEAT. The first mutation is the *add connection* mutation. This mutation selects two nodes randomly from the node genes and adds a connection between them. The second mutation is an *add node* mutation that randomly selects a connection gene to split and add a node between. For the *add node* mutation, the selected connection to be split is disabled, a node is added, and two new connections (one going into the new node and one going out of the new node) connecting the input and output nodes of the old connection via the new node are added. This process is shown in Fig. 2.8. The old disabled connection is shown dashed in grey. The new connection leading into the new node is assigned a weight of one, while the new connection leading out of the new node is assigned the weight of the disabled connection. This is done to minimise the disruption to the network.

As well as structural mutations, weight mutations are also performed. For each mutated offspring, the weights of the offspring are probabilistically perturbed. The range and type of the distribution the perturbations are drawn from is user defined. Each connection is also probabilistically replaced by a new value to occasionally introduce more severe mutations.

2.4.3.2 Crossover

The innovation numbers assigned to genes allows us to identify matching genes between genomes. Furthermore, the speciation of the population ensures that only similar

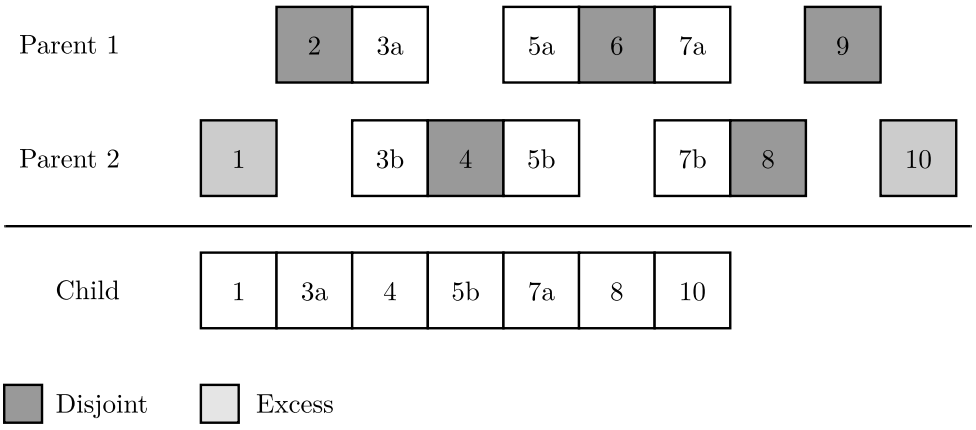


FIGURE 2.9: A visualisation of the crossover operation between two parent genomes. Each box represents a connection gene, labelled by innovation number. Parent 2 is fitter, and therefore the excess and disjoint connection genes are inherited from them. Genes with matching innovation numbers are inherited randomly from either parent.

genomes are mated together (though there is also a small probability that inter-species crossover is performed). The crossover operation between genomes is performed as follows. First, the two parents are aligned using the innovation numbers of the connection genes. Non-matching genes that lie within the range of the other parent are called disjoint genes. Non-matching genes that lie outside the range of the other parent are called excess genes. The offspring is created by traversing the connection genes of the fitter parent. Non-matching genes are inherited from the fitter parent, and matching genes are randomly chosen from either parent. An example of the crossover operation is shown in Fig. 2.9.

2.5 Deep Reinforcement Learning

Reinforcement learning (RL) is a formal framework for solving sequential decision-making problems (Francois-Lavet et al., 2018). Instead of learning from a set of labelled examples (as is the case for supervised learning), or learning patterns or relationships from unlabelled examples (as is the case for unsupervised learning), for RL, models, known as *agents*, learn through interaction with an environment. Deep RL is the combination of deep learning (i.e. deep neural networks) and reinforcement learning. This combination is useful for solving problems with high-dimensional input spaces, such as

vision-based tasks, because neural networks (and in particular convolutional neural networks) can learn to extract features, removing the need for manual feature construction. Prior to the development of deep RL techniques, designing features for reinforcement learning problems was a challenging issue and limited the general applicability of traditional RL methods.

2.5.1 Reinforcement Learning Basics

A traditional RL system consists of six elements (Sutton and Barto, 2018):

- *An agent* that is able to perform actions that cause some change in the environment. Whose goal it is to maximise a reward (see below) over the long term.
- *An environment* within which the agent is learning to solve the problem. This could be a simulation or the real world. The environment may have discrete or continuous action and state spaces that define the actions the agent may take and the state of the environment, respectively.
- *A policy* which defines the action the agent should take given the state of the environment. This encodes a strategy and dictates how the agent behaves.
- *A reward signal* that represents the goal of the problem and rewards the agent for progress towards the goal. In the context of video games, the reward signal is often some function of the agent's score. The reward signal may be dense, meaning that the agent receives a reward frequently (e.g. for every action) or sparse, meaning that the agent receives a reward only after many actions.
- *A value function* that estimates the long-term value of each state or state/action pair, i.e. it provides an estimate of the total reward that can be accumulated from each state in the future or an estimate of the total reward that can be accumulated by performing an action from each state.
- *A model* (optional) that models the environment and allows the agent to make predictions about how the environment will change given the state of the environment and actions performed by the agent. Agents that learn by learning a model of the environment are called *model-based* agents, whereas those that do not use a model are called *model-free* agents.

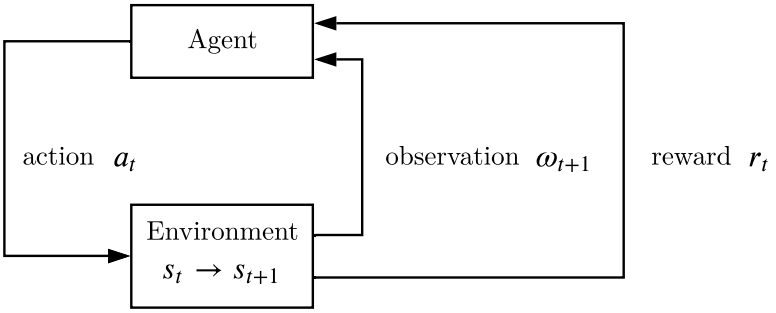


FIGURE 2.10: The basic agent interaction loop in reinforcement learning. Modified from Francois-Lavet et al. (2018).

The agent's interaction with the environment can be described in the form of an interaction loop, as shown in Fig. 2.10. Considering a problem with discrete action and state spaces, first of all, the agent performs an action for the current time step t from a set of discrete actions $a_t \in A$. This causes some change in the state of the environment from $s_t \in S$ to $s_{t+1} \in S$. The agent receives an observation from the environment $\omega_{t+1} \in \Omega$ that represents the updated state of the environment s_{t+1} and a reward $r_t \in \mathbb{R}$ that provides the agent feedback on their action a_t.

2.5.2 Value-Based Methods

Value-based RL methods aim to approximate the value function. One of the simplest and most popular algorithms is the Q-learning algorithm (Watkins, 1989). In its most basic form, this algorithm aims to learn a lookup table $Q(s, a)$ that contains the expected total reward that can be gained by performing action a at state s. Since the table contains an entry for every state-action pair, in this form, Q-learning does not scale well to problems with high-dimensional state and/or action spaces. To alleviate this issue, fitted Q-learning algorithms (Gordon, 1996), use a function approximator with parameters $\boldsymbol{\theta}$, such as a neural network, to approximate the value function Q:

$$Q(s, a; \boldsymbol{\theta}) \tag{2.13}$$

Arguably, the most popular value-based deep reinforcement learning method, which has spawned many variants, is the deep Q-network (DQN) algorithm (Mnih et al., 2015).

This was the first method to train agents that achieved superhuman performance in a number of Atari games from raw pixels (Mnih et al., 2015). DQN was used to approximate the value function using a convolutional neural network (§2.1.2). This network is trained to learn the expected total reward of each action, given the current state of the environment. The policy of the agent is derived from the value function as:

$$\pi(s_t) = \arg\max_{a_t \in A} Q(s_t, a_t; \boldsymbol{\theta}) \tag{2.14}$$

2.5.3 Policy Gradient Methods

Value-based methods derive the policy from an approximation of the value function (see Eq. 2.14). This causes problems when the action space is large or continuous (Sutton and Barto, 2018). To address this issue, policy gradient methods learn to approximate the policy function π directly, instead of learning a value function:

$$\pi(s, a; \boldsymbol{\theta}) \tag{2.15}$$

First proposed by Sutton et al. (2000), policy gradient methods use gradient ascent to optimise a policy network that maximises the expected reward. This approach has the advantage that it is effective in high-dimensional or continuous action spaces (Peters, 2010). However, policy gradient methods can be inefficient and slow to converge, and are also prone to converging to local optima (Peters, 2010). Some methods combine policy and value function learning, such as Actor-Critic methods (Mnih et al., 2016) to capture the best of both approaches.

2.5.4 Evolutionary Methods

Evolutionary reinforcement learning (ERL) methods are another type of policy-based methods that search for policies directly in the policy space. With ERL methods, policies are modified through either mutation or crossover (or both) and are evaluated for entire episodes at a time. The fitness of each agent is the total accumulated reward during the episode.

Evolutionary approaches to solving RL problems avoids several issues present in traditional reinforcement learning methods. Namely, they avoids the credit assignment problem and can offer better exploration of the search space. These factors were discussed in the motivations for our work (§1.1).

2.5.5 Video Games as Reinforcement Learning Benchmarks

The real world is an incredibly complex environment with significant inherent variation and randomness. If we consider the simple task of learning to pick up a cup, even in extremely controlled scenarios, there is still complexity introduced in the hardware, cameras, lighting conditions, and object and agent placement. To compare two methods fairly, one must exactly replicate the conditions of the previous evaluation, which can be extremely difficult. With simulations and games, many of these problems are eliminated or reduced.

Games provide us with simplified environments that are adaptable, consistent, and fully controllable. At the same time, they provide external validity in that they are designed to challenge humans. This is especially true for vision-based game playing, because vision is also an important input when operating in real-world environments, where access to ground truth information or an underlying state-of-the-world is not available. Games also provide us with unlimited training data and provide inbuilt mechanisms for assessing performance, such as scores. It is this combination of qualities that has seen games rise as a dominant tool in evaluating and comparing RL methods designed to produce intelligent behaviours.

One of the most popular video game RL evaluation platforms is the Arcade Learning Environment (ALE) (Bellemare et al., 2013). The ALE provides a standardised RL interface for more than 50 different Atari 2600 games, built on top of the Stella emulator[1]. Each game has an discrete action space of 18 actions, which represent all possible combinations of inputs on the Atari 2600 controller (a 9-directional joystick and a "Fire" button). Also offered is a reduced set of "legal" actions for each game that consists only of actions that actually register as inputs for the game. The ALE provides multiple different types of observations for agents, ranging from 160×210 pixel colour images of the game screen to the values of the 128 bytes of RAM for the console. The ALE

[1] https://stella-emu.github.io/index.html

can run games at up to 6000 frames per second, which allows for very fast evaluation and training, and it incorporates a stochastic frame skipping mechanism for introducing non-determinism into the games.

The OpenAI Gym (Brockman et al., 2016) is another RL evaluation platform for Python that provides a common interface for many different RL problems, including an interface for the ALE. Other video game playing RL benchmarks include the General Video Game AI (GVGAI) framework (Perez-Liebana et al., 2016).

Chapter 3

Literature Review

This chapter discusses prior work in the field of vision-based evolutionary reinforcement learning (ERL). Although not all of the methods we discuss are suitable for general video game playing (GVGP), they still provide motivation and inspiration for our research. As a whole, the discussions in this chapter establish the context around our work and summarise the current and recent history of vision-based ERL.

The prior work in vision-based ERL can be divided into two categories: purely evolutionary methods, and hybrid methods. Purely evolutionary methods attempt to solve problems using evolutionary optimisation techniques alone. Hybrid methods combine evolutionary optimisation methods with other machine learning methods. The hybrid methods are the closest to our own work. After summarising the prior work in these two categories, we outline how they motivate our investigations and describe the niche within which our work sits.

3.1 Purely Evolutionary Methods

The majority of the prior work around vision-based ERL uses purely evolutionary optimisation methods. The majority of these focus on neuroevolution, following the increasingly popular trend of using neural networks to solve reinforcement learning (RL) problems. Like traditional deep RL methods, most ERL methods focus on *end-to-end learning*: optimising a single neural network that is responsible for feature extraction, state representation learning, and policy learning. We start by discussing these methods,

followed by those that use a *separated learning* approach like our own. Separated learning methods divide the responsibility of state representation and policy learning between two separate networks. Finally, we discuss a completely different genetic programming approach, before moving on to hybrid learning methods.

3.1.1 End-to-End Neuroevolution

We described earlier that optimising the large networks required for feature extraction, state representation, and policy learning is problematic for evolutionary algorithms and limits their effectiveness. However, this has not prevented some success. Broadly, end-to-end neuroevolution methods that have been used for vision-based RL can be categorised by their use of *direct* or *indirect* genome encoding. Direct encoding methods encode the parameters of solutions (e.g. neural networks) directly, whereas indirect encoding methods encode the parameters of solutions indirectly to reduce the size of the genomes.

3.1.1.1 Direct Encoding Methods

Hausknecht et al. (2014) published some of the first work evaluating neuroevolution as a potential technique for general Atari game playing. They assessed the ability of three different direct encoding methods — conventional neuroevolution (a simple genetic algorithm that uses crossover and mutation to modify the network weights), covariance matrix adaptation evolution strategy (CMA-ES, Hansen and Ostermeier (1996)), and neuroevolution of augmenting topologies (NEAT, Stanley and Miikkulainen (2002)) — to evolve feed-forward policy networks for 61 different Atari games. Two of these methods, conventional neuroevolution and CMA-ES, evolve the weights of fixed topology networks, while NEAT evolves both the weights and topology of networks[1]. They also compared these methods against an indirect encoding method, HyperNEAT (Stanley et al., 2009). Their HyperNEAT results are discussed in the next section. For their vision-based experiments, they used the Arcade Learning Environment (ALE) (Bellemare et al., 2013). The algorithms were used to evolve networks that took as input downsampled $8 \times 21 \times 16$ pixel images, with one channel for each of the eight colours in the SECAM colour palette. They reported that none of the three methods were able to evolve solutions for any of the games using these inputs.

[1] NEAT is described in §2.4.

Despite the failure of conventional neuroevolution and CMA-ES to evolve solutions for Atari games from raw pixels. Salimans et al. (2017) and Such et al. (2017) later showed that similar methods, when operated on a larger scale (parallelised across 1,440 and 730 CPU cores, respectively), were indeed able to evolve solutions. Salimans et al. (2017) showed that evolution strategies (ES), described in §2.3.1.2, is an effective approach to optimising the weights of deep neural networks for hard reinforcement learning tasks. Using ES, they produced agents that matched the performance of agents trained using gradient-based optimisation on most Atari games tested. They found their implementation, OpenAI ES, to be data efficient, requiring only between 3-10 times as much data to match the performance of the Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016) algorithm on 23 of the 51 Atari games tested. They also found OpenAI ES to be qualitatively better at exploration on a non-vision-based 3D humanoid locomotion task using the MuJoCo physics simulator (Todorov et al., 2012), learning a variety of walking gaits not observed through training with gradient-based methods.

Shortly after Salimans et al. (2017) found ES to be effective at optimising deep neural networks to solve hard RL problems, Such et al. (2017) also showed that simple genetic algorithms are a competitive alternative to gradient-based techniques. They compared the performance of networks optimised using a genetic algorithm (Deep GA), to networks optimised through gradient-based methods on a selection of Atari games. Across the 13 games tested, networks evolved using Deep GA outperformed networks trained using DQN (Mnih et al., 2015), A3C and OpenAI ES on three games. Further, the training time for networks using Deep GA was drastically lower than both DQN and A3C, due to the ability to exploit parallelism when evaluating individuals in each generation. The networks evolved using Deep GA took between one and four hours to train, depending on the level of parallelism, compared to approximately four days and between seven and ten days for A3C and DQN respectively (Such et al., 2017). Deep GA and OpenAI ES have since been shown to also successfully evolve agents capable of driving in the Donkey Car Simulator as well (AbuZekry et al., 2019).

One of the difficulties in scaling neuroevolution to deep neural networks is that only small perturbations can be made to the network weights to ensure that existing functionality is not broken by the mutations. This limits the speed with which evolution can proceed, resulting in neuroevolution often being far less efficient than gradient-based training algorithms. Lehman et al. (2018), and Tymchenko and Antoshchuk (2019) both proposed

techniques to alleviate this issue and improve the effectiveness of weight mutations and crossover operations, respectively.

Lehman et al. (2018) coined the term "safe mutations" to describe mutations that are strong enough to allow evolution to progress, but not so strong as to break any existing functionality in the evolved networks. They proposed two types of safe mutation operators for perturbing the connection weights of neural networks, safe mutation through rescaling (SM-R) and safe mutation through gradients (SM-G), that scale weight perturbations relative to the effect that they have on the network's outputs. Both methods scale the magnitude of weight perturbations so that the divergence in the network's outputs (across a small sample of recent examples) between the parent and child remain within some threshold. This threshold replaces the usual mutation rate parameter. SM-R operates by optimising the magnitude of the perturbations using a simple line search, at the cost of a forward pass per example. SM-G uses gradient information, at the cost of an additional backward pass per example. For each weight, the magnitude of each output's gradient (with respect that weight) provides an estimate of the sensitivity of the output to that weight, which is used to scale the perturbation[2].

Lehman et al. (2018) evaluated variants of the SM-G operators on a vision-based maze navigation reinforcement learning task. The weights of a fixed network architecture with eight convolution layers, two LSTM layers, and an output layer (a total of 20,573 parameters) were evolved using mutation alone. They found that the use of safe mutation operators lead to significantly shorter times to find solutions and a significant increase in the proportion of runs in which solutions were found, compared to a control version that did not use safe mutations.

Tymchenko and Antoshchuk (2019) proposed a safe crossover operator that, as with the safe mutation operators proposed by Lehman et al. (2018), aims to reduce the likelihood of degenerate offspring being generated during reproduction. Their method, Layer Blend Crossover (LBC), linearly interpolates the weights of corresponding layers in the parent networks according to the fitness of the parents. The interpolation factor is drawn from a truncated normal distribution with a fixed variance, centred around the proportion of the total fitness that is earned by Parent 1. Tying the interpolation factor to the relative

[2]This is not to be confused with deep RL techniques that use gradient descent and calculate gradients with respect to reward signals

performance of the networks has the effect of the fitter parent influencing the weights of the child more.

In their evaluations using The Open Racing Car Simulator (TORCS)[3], Tymchenko and Antoshchuk (2019) use a simple genetic algorithm with LBC and the SM-R mutation operator described previously, to evolve the weights of a deep convolutional neural network. They found that their genetic algorithm was able to find good continuous control policies across a number of different tracks that were competitive against the policies trained using the Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2015). DDPG has been shown to perform very well across a wide range of reinforcement learning problems (Lillicrap et al., 2015).

3.1.1.2 Indirect Encoding Methods

Stanley et al. (2009) developed a method of neuroevolution called HyperNEAT. This method uses NEAT (Stanley and Miikkulainen, 2002) to evolve compositional pattern-producing networks (CPPNs) that indirectly encode the weights of larger neural networks. The evolved CPPNs are similar themselves to small neural networks, except for that they use different activation functions (e.g. a combination of Gaussian, sine, or linear functions) for the neurons. The composition of these basic functions allows them to generate spatial patterns in the network weights with symmetry (through the use of symmetric functions, such as Gaussians) and repetition (using periodic functions, such as sine). For a fixed neural network node layout, the CPPNs output a connection weight for every possible connection (including recurrent connections). Using HyperNEAT, Stanley et al. (2009) were able to evolve functional million-connection networks.

Hausknecht et al. (2014) evaluated the ability of HyperNEAT to evolve Atari playing agents that learn from raw pixel inputs (alongside the other algorithms mentioned earlier). Using the ALE, they evolved agents for 61 Atari games and found that HyperNEAT was able to evolve solutions that beat average expert human scores on three games: Bowling, Kung Fu Master and Video Pinball. Their evolved agents learnt from the downsampled $8 \times 21 \times 160$ pixel images encoded using the 8-colour SECAM colour palette described earlier.

[3] http://torcs.sourceforge.net/

Wavelet-Based Encoding (WBE) (van Steenkiste et al., 2016), is an alternative indirect encoding scheme for neuroevolution, inspired by wavelet-based lossy image compression methods. This encoding scheme reduces the search space by using wavelets (signal processing functions for decomposing signals into separate frequency components) to create a mapping between the gene space (wavelet coefficients) and the weight space. Instead of storing and directly evolving the weights of the networks in the genomes, the wavelet coefficients for the low-frequency components are stored and evolved. These describe the most important information in the weights. Lossy reconstructions of the network weights are obtained by convolving the inverse wavelet transforms over the weight matrices of the networks. As with HyperNEAT, this means that their method is able to preserve spatial relationships between the network weights.

van Steenkiste et al. (2016) evaluated WBE on a set of five Atari games (Atlantis, Gravitar, Phoenix, Seaquest, Space Invaders, and Q-Bert) using the ALE. The images from the ALE were converted to greyscale and downsampled to 105 $\times$ 80 pixels. They compared the performance of WBE against direct evolution in the weight space, and the scores reported by Hausknecht et al. (2014) using HyperNEAT. They used a fixed network architecture with a single 100 neuron, fully recurrent hidden layer and a fully connected output layer and evolved the wavelet coefficients using CoSyNE (Gomez et al., 2008). They reported that evolution using WBE outperformed direct evolution (using CoSyNE to directly evolve the weights of the network) in all six games, and HyperNEAT in five of the six games. Despite the ability of HyperNEAT and WBE to evolve high-performing Atari agents, neither of these methods evolve both the weights and topology of the policy networks. Therefore, they do not harness the full potential and benefits of using neuroevolution to evolve policy networks.

3.1.2 Separated Neuroevolution

Though most of the work exploring separated learning methods use neuroevolution for policy learning and other methods for state representation learning (and hence are discussed with the other hybrid learning methods), an outlier is the work by Koutník et al. (2014). They proposed a method in which the weights of two separate networks, a convolutional neural network for state representation learning and a smaller recurrent neural network for policy learning, are evolved independently. First, the weights of a small

CNN are evolved using images collected by manually exploring the environment. The weights are evolved using CoSyNE (Gomez et al., 2008), to maximise the variation in the feature vectors produced by the network. The fitness function used is the sum of the minimum and mean pairwise Euclidean distance between the produced feature vectors for the images. Following this, the weights of a small recurrent neural network controller are evolved, again using CoSyNE, that uses as input the feature vectors produced by the CNN instead of raw observations from the environment.

Koutník et al. (2014) evaluated their method on a driving task using the TORCS racing simulator. The inputs to the networks were 64×64 pixel images. They found that agents trained in the way described above were able to complete a lap of the test circuit and continue to drive without crashing. These results are encouraging for our research, because they provide evidence that for a *single* domain, sufficient state representations can be learnt (without input from the policy learner) that enable policy learning. One of the intentions of our research is to see if sufficient representations can be learned in an online setting (i.e. alongside policy learning) for multiple different domains (games).

3.1.3 Genetic Programming

The methods presented in the previous two subsections use neural networks approximate policy functions that decide which action to take at each time step. However, Kelly and Heywood (2017) proposed a fundamentally different, genetic programming framework for vision-based reinforcement learning called Tangled Program Graphs (TPG). This method evolves "teams" of programs that are connected to form a program graph that decides which action to take at each time step. An example structure of an evolved TPG policy is displayed in Fig. 3.1. Each team consists of a number of programs (a sequence of instructions composed of linear operations) that operate on the inputs or internal registers (memory). Each program outputs a real value, and is associated with an action the agent can perform or another team. The initial population of teams are initialised with a small number of programs that are each linked with an action. Links to other teams or different actions arise through mutations, leading to the emergence of multi-team graphs over time. At each time step, the program graph is traversed to choose the agent's action. For each team, the deployed program is the one with the highest output.

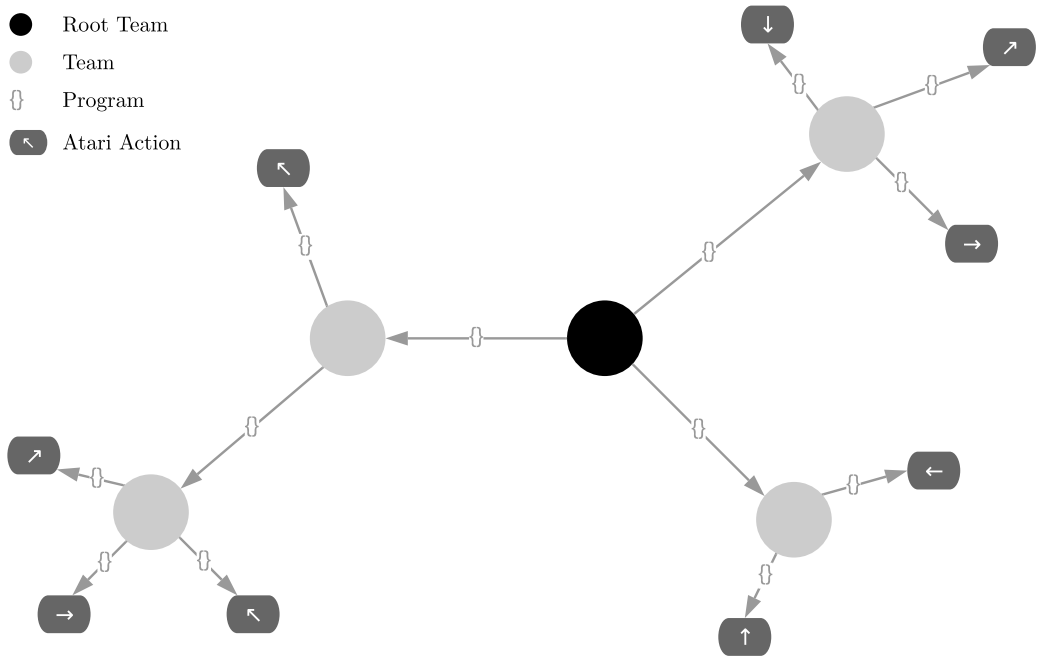


FIGURE 3.1: A simplified example of the evolved TPG policy drawn by Kelly and Heywood (2017).

TPG was evaluated on the 20 Atari games that DQN (Mnih et al., 2015) failed to train agents that surpass human-level performance. For their evaluations, the input frames were preprocessed and reduced to a 1344-dimension vector using a screen quantization procedure. This procedure took the frames provided by the Arcade Learning Environment (Bellemare et al., 2013), encoded in the 8-colour SECAM colour space, and subdivided them into a 42 × 32 grid. Each cell in the grid was described by a single byte, in which the bits encoded the presence of one of the eight colours in the cell. The final 1344-dimension vector consisted of the decimal values for each cell byte. This preprocessing is the most extensive of all the methods described, and is heavily reliant on certain properties of the Atari domain, severely limiting the generalisability of this method to other domains.

In their evaluations, Kelly and Heywood (2017) compared the performance of TPG performed against scores reported for DQN (Mnih et al., 2015) and HyperNEAT (Hausknecht et al., 2014). The TPG agents achieved higher scores in 14 of the 20 games used in their evaluations and surpassed expert human scores (Mnih et al., 2015) in seven of those. They also found that the evolved TPG solutions were far less complex than these competing neuroevolution methods. The average number of instructions executed per time

step for the evolved TPG agents ranged from 116 and 1036 (depending on the game), compared to the > 800,000 weight calculations performed by DQN and HyperNEAT (Kelly and Heywood, 2017). However, while the final polices were less complex, the evolutionary runs to evolve the TPG agents were very long, lasting a maximum of two weeks each.

3.2 Hybrid Methods

The methods presented thus far all use purely evolutionary approaches to solving RL problems. However, much like our proposed method, others have proposed methods that combine neuroevolution with other approaches to harness the respective benefits of both. We call these hybrid methods.

Most of the hybrid methods proposed are separated learning methods that separated state representation and policy learning. Before discussing these methods, however, we first describe an end-to-end learning method.

3.2.1 End-to-End Learning

An approach to incorporating gradient information into the evolutionary search was taken by Khadka and Tumer (2018). Their method uses a population of networks evolved using a simple evolutionary algorithm to generate diverse data to train an agent using the actor-critic method A3C (Mnih et al., 2016). The actor-critic network is trained in parallel to the evolutionary population, and periodically injected into the population to provide potential guidance. The idea behind this technique is that if the policy gradient trained network is better performing than most agents in the population at a given time, then it will be selected as a parent of the next generation, thus incorporating the gradient information into the evolutionary search. Compared against popular state-of-the-art pure policy gradient methods DDPG (Lillicrap et al., 2015) and PPO (Schulman et al., 2017), and a simple evolutionary algorithm similar to Deep GA, their method consistently outperformed each method, showcasing it's ability to capture the best of both worlds. Their method also took only around 3% longer to run on average than DDPG (Khadka and Tumer, 2018).

3.2.2 Separated Learning

To alleviate the difficulties of evolving networks with large input spaces, but still reap the benefits of evolutionary approaches, some researchers have explored compressing the input space automatically before evolving networks which learn instead from this compressed input space. This is the same as the method proposed by Koutník et al. (2014) (§3.1.2), except for that the methods discussed here use alternatives to evolutionary algorithms to optimise the compressors. This explicit separation of feature extraction and behaviour learning is what we refer to as a *compressor-controller* agent model.

As far as we are aware, the first occurrence of automatic input space compression was the work by Cuccu et al. (2011). They employed a vector quantisation algorithm to compress 15 × 30 px images (450 inputs) from a vision-based version of the mountain car task into a smaller feature vector of 8 inputs. This feature vector was in turn used as the input for a recurrent neural network (RNN) controller. By reducing the number of inputs, the controller is far smaller than it would otherwise need to be, and the weights were successfully evolved using Separable Natural Evolution Strategies (SNES) (Wierstra et al., 2014). Although more modern techniques, such as those discussed in §3.1.1.1, would likely be able to solve this task without compression, SNES was not.

Another technique which divides that task into low-dimensional representation learning and behaviour learning and addresses this problem is that of Alvernaz and Togelius (2017). They propose a compressor-controller architecture that uses an autoencoder to learn a compressed representation of game images and evolves controller networks using CMA-ES (Hansen and Ostermeier, 1996), with the compressed representation as their input. The shared experience of agents, i.e. the images they generate, is used to periodically refine the compressor and improve the compressed representation. The idea behind their work was that as the agents learn they will encounter new and varied game states, which can be used to further train the autoencoder, and that as the autoencoder improves the agents will be able to learn better behaviour from a better representation. This way, both the training of the agents and the learning of a compressed representation proceed in tandem. Their method was tested on a health-pack gathering task in the VizDoom environment (Kempka et al., 2016). Despite reducing the dimensionality of the inputs for the evolved behaviour generating networks, they did not evolve the topology of these networks, only the weights. They also did not experiment with the size of

the autoencoder bottleneck (the compressed representation). They used all images that were reproduced with higher than a certain mean-squared error (MSE) threshold in their training set for the autoencoder.

Despite success in being able to reduce the input space to evolve smaller controllers, little research has been conducted to utilise this to harness the benefits of evolving topology. As far as we are aware, the only work to have pursued this is that of Poulsen et al. (2017). They proposed a different approach for allowing smaller behaviour-generating networks to learn to play a first-person shooter game from raw pixel inputs. They trained a deep convolutional neural network in a supervised fashion using ground truth information from the game engine itself to learn a compressed feature vector representing the current game state. They experimented with learning both an angular representation of the angle between the agent and the target and a visual partitioning representation which specifies the position of the target within the image. The behaviour-generating networks were also trained on ground truth representations based on game engine data, but at evaluation time were evaluated using the learned representations. Poulsen et al. (2017) used NEAT (Stanley and Miikkulainen, 2002) to evolve the topology and weights of the behaviour-generating networks. Due to the requirement of game engine data to train the compression network in a supervised fashion, their technique would not generalise well to other simulation or game tasks where this data is unavailable, and for the same reason presents a problem when moving from games to tasks in the real world. Because of the use of game-specific knowledge, Poulsen et al.'s (2017) technique is not as applicable to the task of general video game playing (GVGP).

Finally, Ha and Schmidhuber (2018) developed a model-based reinforcement learning method that trains agents that consist of three components: a compressor, a world model, and a controller. To compress the observations from the environment, they use a variational autoencoder (VAE) to learn compact encodings. A Mixture Density Network combined with a recurrent neural network (MDN-RNN) (Graves, 2013) is then trained to predict the compact encodings produced by the VAE, given the encoding at the current time step, the action taken, and the RNN hidden state on a set of roll-outs. This MDN-RNN is the world model. Finally, having trained both the VAE and MDN-RNN, a linear controller network is evolved using CMA-ES that that takes as input the compact encoding produced by the VAE and the hidden state of the MDN-RNN.

In their evaluations, Ha and Schmidhuber (2018) evaluated their method in two OpenAI Gym (Brockman et al., 2016) environments, a top-down car racing simulator (CarRacing-v0) and a 3D VizDoom environment (DoomTakeCover-v0). They found that their agents were able to achieve scores that surpassed the threshold beyond which the environments are considered solved, outperforming traditional deep reinforcement learning methods, such as DQN (Mnih et al., 2015) and A3C (Mnih et al., 2016). In each of these environments, the agents learned from 64×64 pixel RGB images. For car racing, the VAE learned a latent representation of 32 dimensions, that was increased to 64 dimensions for the VizDoom task. For the car racing experiments, they also found that the agent was able to learn good policies without using the world model, but did not perform as well as the full agent.

3.3 Key Findings from Prior Work

The downside of the end-to-end learning approaches attempted in the past are that they are only able optimise the weights of fixed topology networks. Hausknecht et al. (2014) showed that NEAT, an commonly used algorithm that evolves both the weights and topology of networks, is unable to scale to vision-based reinforcement learning domains. The downside of the previous separated learning approaches by Cuccu et al. (2011) and Koutník et al. (2014), Poulsen et al. (2017), and Ha and Schmidhuber (2018) is that the compressor is trained offline before the evolution of the behaviour controlling network. This introduces the problem of requiring training data to be collected for this training ahead of time, either manually or by random agents. The problem with manual collection, as we discuss later in our state representation learning experiments (§5.2), is that it is infeasible to collect data for a large number of environments, hampering the general applicability of such methods. The problem with random agent collection is that in hard to explore games, random agents will not explore large areas of the state space, meaning that the models and encodings learnt may not generalise well to unseen, later game states and therefore hamper the ability of the agent to perform well upon reaching these conditions. Poulsen et al. (2017) showed that NEAT can be scaled to domains with high-dimensional inputs, but the caveat to their work is that the representations used to train the NEAT agents were both trained offline, and in a supervised manner, making their method unsuitable for GVGP as it requires the collection of labelled data

for each new environment. Finally, although Alvernaz and Togelius (2017) train the compressor and controller online simultaneously, they still evolve the weights of a fixed topology network, again, not harnessing the full benefits of neuroevolution.

Our method is different to those proposed in prior work in that it is the only method that combines (a) simultaneous compressor and controller training, similar to end-to-end learning approaches, (b) topological evolution for the controller, and (c) training of the compressor in an unsupervised manner. Furthermore, we investigate the relationship between the size of the learnt latent space encodings and the quality of the learnt encodings, something that none of the prior work has investigated. It is conceivable to think that the density of the input space, i.e. the possible variation in game images, and the nature of the game, e.g. 2D or 3D, may influence the required bottleneck size.

Based on our review of prior work on evolutionary vision-based reinforcement learning, we have identified the following key findings which motivate our research:

- Using raw pixels, the input space is too large to evolve both the weights and topology of neural network agents using current methods.
- Prior methods that compress the image before passing it to evolved networks show promise but do not yet compete with the state of the art, nor have they been thoroughly evaluated for GVGP.

Chapter 4

AutoEncoder-augmented NEAT

The previous chapters provide the motivation and justification for our pursuit of reinforcement learning (RL) method in which state representation and policy are learnt simultaneously by two separate networks. This chapter first and foremost describes our proposed method for achieving this, AutoEncoder-augmented NeuroEvolution of Augmented Topologies (AE-NEAT). We begin by outlining the design of the agents that are trained using AE-NEAT, before describing the details of state representation and policy components. After this, we describe the details of how agents are trained using AE-NEAT.

In addition to describing AE-NEAT, this chapter also introduces two important aspects of the evaluations that were performed to assess AE-NEAT, which are presented in chapters 5, 6, and 7. The first is the Atari Annotated RAM Interface (AtariARI) (Anand et al., 2019), which is used to independently evaluate both the state representation and policy learning components of AE-NEAT. The second is the common set of games that are used throughout the remainder of the thesis to evaluate both these individual components and the overall effectiveness of AE-NEAT.

4.1 Agent Design

The agents trained using AE-NEAT follow the same compressor-controller design utilised by other methods discussed in §3.1.2 and §3.2.2. For the agents trained using AE-NEAT, the compressor and controller are two separate neural networks. The interaction between

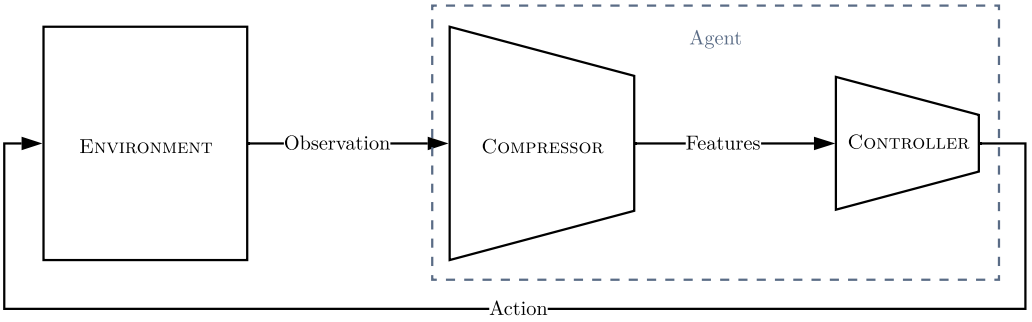


Figure 4.1: The design of our agents and the function of each component within the interaction loop with the environment.

the compressor and controller networks is as follows. At each time step, an observation, which is an image of the game screen, is provided by the environment. This image is first passed through the compressor, which creates a compact state representation from the image. This compact state representation is then fed to the controller, which decides the action taken by the agent. This loop continues until the end of the episode is reached. Fig. 4.1 shows a high-level representation of this design and the responsibility of each component within the interaction loop. The key difference between this interaction loop and the basic RL loop described in §2.5 is that agents perform a two step process to decide their actions.

There are a number of benefits gained through this separated agent design. First, it allows for great flexibility in terms of the methods we can apply for each. Second, it allows us to clearly evaluate their performance independently and as a whole. Third, it allows us to harness the benefits of gradient-based and evolutionary-based learning. The following sections provide a closer look at the compressor and controller components.

4.2 State Representation Learning

In the compressor-controller agent design described in the previous section, the compressor is responsible for learning a compact state representation of the environment. As well a generating a sufficiently compact set of features from which we are able to evolve a successful policy network, the other requirement of the encoder is that it must learn in an unsupervised manner. This is crucial for the development of a general video game playing (GVGP) agent, as we cannot manually handcraft state representations or

use game-specific knowledge for each game. To learn compact state representations, we use the encoder half of an autoencoder.

Autoencoders provide us a powerful unsupervised method for training a compressor that produces compact representations of the game state, by learning compact representations of the images that are ordinarily displayed on screen. As described in §2.2, autoencoders that rely on a sufficiently small bottleneck and are trained to minimise the difference between reconstructed and original images alone are known as undercomplete autoencoders. However, our primary concern is not with the quality of the reconstructed images, rather that the compressor learns to encode important properties of the data. From the perspective of policy learning, the important properties may not align with the important properties for reconstruction. Therefore, we also investigate the use of variational autoencoders (§2.2.2) that attempt to learn a distribution over the latent variables, as well as disentangled variational autoencoders (§2.2.2.3) that place greater emphasis on each of the latent variables encoding different properties. Our hope is that one of these autoencoder techniques is able to learn adequate state representations to enable policy learning. The exact type and architecture of autoencoder will be determined empirically via a comparison between different type and architecture combinations. These experiments are detailed in Chapter 5.

The observations from the environment that are passed to the compressor undergo very minimal preprocessing. The only preprocessing that is performed is converting the RGB images output by the environment into single-channel greyscale images. This differs from the vast majority of reinforcement learning methods assessed on Atari environments. Typically, most methods both convert the images output by the Arcade Learning Environment to greyscale and downsample them from their original 210×160 pixel dimensions to 64×64 pixels. Furthermore, most methods rely on frame stacking to circumvent the problem of *partial observability* (described in the following section). They pass not only the current frame but also the previous three frames to the agent at each time step. Our method does not require this modification, as the policy networks are able to evolve recurrent connections to deal with partial observability. Our choice to use full-size frames further strengthens the generalisability of our method, removing the need for the downsampling step specific to Atari.

4.3 Policy Learning

For policy learning, we harness the exploration qualities and architectural design benefits of topology and weight evolving neuroevolution algorithms. Specifically, we use the NeuroEvolution of Augmenting Topologies (NEAT) algorithm (§2.4). This algorithm evolves both the weights and topology of networks, starting from minimal structures consisting only of the input and output nodes. We use NEAT to evolve recurrent neural network controllers.

Due to the presence of moving objects, most Atari games, including Pong and Breakout, are partially observable Markov decision processes (POMDPs) when only information from the current frame is provided with each observation. This is because the speed and direction of moving objects cannot be determined from a single observation. To alleviate this problem, a common solution is to include information from the four most recent frames in each observation and train feed-forward networks (Mnih et al., 2015). However, this increases the dimensionality of the input space and does not solve the partial observability problem for games that require a memory of more than four time steps. Instead, we evolve recurrent neural networks (RNNs) to enable the development of good policies in partially observable environments.

We evolve RNNs that at each time step propagate the outputs of each neuron forward. This behaviour was described in §2.1.1. Compared to some RNN implementations, the initial outputs are not immediately influenced by the inputs, but they do not require there to be topological ordering of nodes or the specification of whether lateral connections are recurrent. For situations like ours, where the network topology is evolved, rather than hand-crafted, this reduces the constraints that need to be specified (allowing for potentially more creative solutions) and overhead in implementation. To evolve the controller, we use our own implementation of NEAT.

4.3.1 PyNEAT: A Python Implementation of NEAT

PyNEAT, our Python implementation of NEAT, conforms to the NEAT-Python interface (McIntyre et al., 2017), to allow for the reuse of existing reporting and logging modules. To parallelise the fitness evaluations of the population, our implementation uses Ray (Moritz et al., 2017).

TABLE 4.1: A comparison of the original version of NEAT compared to PyNEAT for the XOR problem.

	NEAT Original	PyNEAT
Mean # Generations	35.9 (18.0)	41.50 (18.7)
Mean # Hidden Nodes per Solution	2.7 (1.6)	3.8 (2.2)
Mean # Connections per Solution	10.6 (4.3)	11.82 (4.0)
Success Rate	97 %	95 %

4.3.1.1 Negative Fitness Values

To better fit the Atari domain, we modify NEAT to support negative fitness values. The original specification of NEAT does not allow for negative values, because positive values are required for fitness sharing. To avoid this issue, fitness sharing is implemented using an adjusted fitness value, specified as the difference between the individual's fitness and the lowest fitness in the population. This ensures positive values and preserves the fitness differences between individuals.

4.3.1.2 XOR Verification

To verify the quality of PyNEAT, we applied it to the XOR problem. The XOR problem requires the algorithm to evolve XOR ("exclusive OR") networks. Since these networks require hidden nodes, it is a good test to establish that our implementation can reliably grow and optimise additional network structures when they are required. We followed the same experimental design as Stanley and Miikkulainen (2002) to compare the performance of PyNEAT against their original implementation of NEAT in C++.

For each implementation, we performed 100 runs using the same hyperparameter values reported by Stanley and Miikkulainen (2002). Table 4.1 compares the performance of both implementations. These results show that PyNEAT performs comparably to their implementation, finding solutions on average in only a slightly longer number of generations and solutions that are on average only slightly more complex (for reference, the simplest possible XOR requires only one node). PyNEAT also had a success rate of only 2% less than the original implementation. These results provided confidence that PyNEAT is a sufficiently high-quality implementation that could be used for our experiments.

4.4 Training Procedure

The combination of reinforcement learning and evolutionary algorithms lends itself to being a very computationally expensive process. This is the single biggest limiting factor in our work. To alleviate this issue, we implemented a parallelised training algorithm for training such agents, depicted in Fig. 4.2, using the Python library Ray (Moritz et al., 2017) to utilise a cluster of nodes.

During training, the core of the algorithm is performed on the master node. Here, the population of policy networks is created, and then mutated between generations. Also, between generations, the autoencoder is trained on the latest images collected during the agents' exploration of the games. One of the limiting factors we experience is the sheer number of images generated during exploration. As we describe later on, a large population of agents is able to generate potentially tens of millions of frames. Within our cluster constraints (a collection of machines in the Department of Computer Science and Software Engineering computer labs) it is infeasible to collect, store and return these images to a central observation store from where they can be sampled by the master for training the autoencoder. Instead, sampling is performed by each worker on a per episode basis. First, the policy network is evaluated for an episode and the observations generated by the environment are store in memory. Once the episode is completed, these observations are sampled and only this sample is passed onto the centralised image store. This achieves two purposes. First, it controls the number of observations that are store, and second, limits the network traffic and reduces the overall evaluation time of each policy network by minimising excess data transfer.

Our image store is implemented as an in-memory Redis[1] database on the master node. This provides a fast, low-latency store that removes the need for observations to be saved and written to disk. It also solves the problem of collating observations from the distributed workers. The image/observation store is configure to store the latest N observations generated from the agents. The limiting factor for N is the amount of available memory on the master node. To train the autoencoder between generations, a sample of images is drawn from the image store, the autoencoder is refined, and then when the new population of network is evaluated, the updated encoder is sent with the policy network to be evaluated.

[1] https://redis.io/

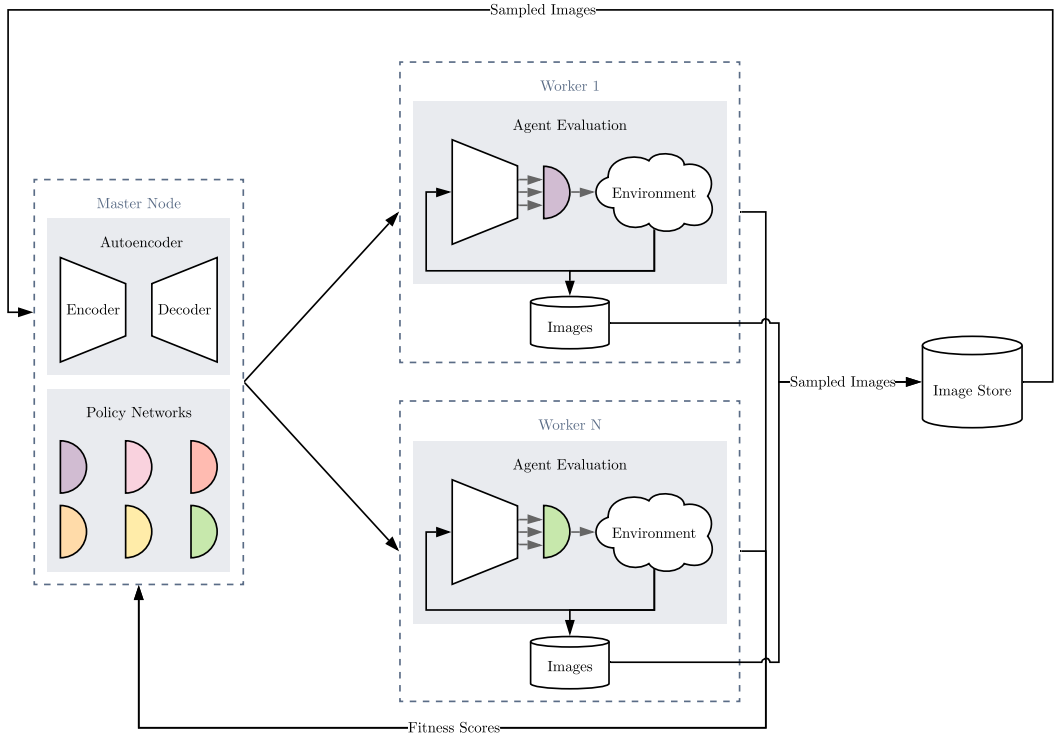


FIGURE 4.2: An overview of the the training process for our hybrid agents.

4.5 The Atari Annotated RAM Interface

The Atari Annotated RAM Interface (AtariARI) (Anand et al., 2019) provides RAM annotations for 22 of the 59 Atari 2600 games supported by the OpenAI Gym toolkit[2]. These annotations identify which of the 128 bytes of RAM store values related to information displayed on screen, such as the position of the player. These annotations were created by analysing either commented disassemblies or the source code (where available) for each game. Through a wrapper for the existing OpenAI Gym interfaces for each supported Atari game, the RAM values for each state variable are made available at each time step. For one of these games, Pong, the values provided at each time step are shown in Fig 4.3.

The interface was designed for assessing the performance of state representation learning methods that try to learn condensed representations of the game state from the images displayed on the screen. Such methods are assessed by training a classifier (referred to as a *probe*) for each state variable that predicts the value using the condensed representation

[2]At the time that this experimental work was conducted.

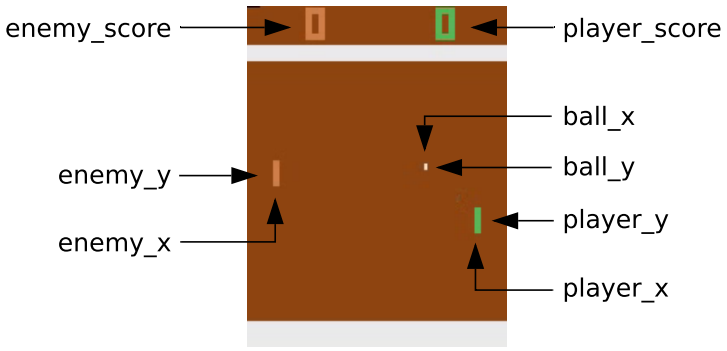


FIGURE 4.3: The RAM annotations provided by the Atari Annotated RAM Interface (Atari ARI) for the game of Pong.

as input. The performance of the classifiers give an indication of the quality of the encoding of each variable in the learned state representations.

In this work, we use the AtariARI for two purposes. First, we use it to evaluate the quality of representations learned by autoencoders, and propose several improvements on the original evaluation procedure. Specifically, we propose the use of regression over classification for appropriate state variables, and the use of non-linear probes. These improvements are proposed in §5.3. Second, we use the compact representations provided by the AtariARI as a baseline for policy learning using NEAT. By filtering out unimportant RAM values, the input space for each game is substantially reduced, removing excess noise that might otherwise make the task of policy learning more challenging. For instance, for Pong, the input space is reduced to only eight inputs, a reduction of 93%. To our knowledge, the AtariARI provides the most condensed, high-quality state representation for many of the supported Atari 2600 games. These uses of the AtariARI allow us to perform independent evaluations of each of the components in our hybrid learning method, and shine light on potential deficiencies.

4.6 Game Selection

Although the AtariARI provides gym wrappers for 22 different Atari games, inspection of the objectives of different games reveals inadequacies in some of the state representations provided. Due to this, we use only a subset of the supported games in our evaluations.

We categorise the supported games into three categories: *poor*, *fair* and *good*; based on the perceived quality and completeness of the state representations provided by the AtariARI. Representations that lack key information relevant to the objective of the game, that is so crucial that we do not reasonably expect an agent to be able to learn a good strategy from, are classified as *poor*. Representations that lack some information about the state of the game, but appear to include enough information to develop an adequate strategy from, are classified as *fair*. Finally, games for which the representation appears to include all information required to learn an optimal strategy, generally speaking information about all game play components displayed on screen, are classified as *good*. Each game is classified by examining game play and comparing the knowledge required to play the game against the information provided by the AtariARI.

One example of a *poor* game is Riverraid. The AtariARI representation is classified as *poor* for this game, because only information regarding the state of the player (e.g. x position, remaining fuel) is provided at each time step. The representation does not include any information about the state of enemies and obstacles, or the locations fuel tanks for refuelling. Because this information is not provided, we find it unreasonable to expect the agent to develop an adequate strategy, given that they must avoid the obstacles and refuel to survive. An example of a *fair* game is Video Pinball, which includes information on the location of the ball and position of the paddle, allowing the agent to keep the ball in play, but it does not include information on the targets the agent needs to aim for to score points. Finally, an example of a *good* representation is the representation provided for Pong, which includes information about all of the objects displayed in each frame. The state representation for Pong is illustrated in Fig 4.3.

Of the 22 supported games, eight games have *poor* representations (Hero, Montezuma's Revenge, Pitfall, Private Eye, Qbert, Riverraid, Venture and Yars Revenge), eight games have *fair* representations (Berzerk, Breakout, Demon Attack, Frostbite, Ms. Pacman, Seaquest, Space Invaders and Video Pinball) and six games have *good* representations (Asteroids, Bowling, Boxing, Freeway, Pong and Tennis). We use only the 14 games for which the AtariARI provides *fair* or *good* representations in our evaluations.

4.7 Summary

The most important aspects of our hybrid learning method are:

- State representation and policy learning are performed separately, but simultaneously. This separation allows neuroevolution methods that evolve both weights and topology to be applied to domains with large input spaces.
- State representations are learned using an autoencoder that is trained using observations gathered by the policy networks during evaluation.
- A population of policy networks are learned (evolved) using NEAT, that take as input the feature vector learned by the encoder half of the autoencoder.

Furthermore, we also identified games that the AtariARI may be insufficient for evaluating state representation learning methods, due to missing information that appears important for learning effective policies. In the next two chapters, we evaluate each of the state representation and policy learning components independently, before evaluating the performance of the entire method as a whole in Chapter 7.

Chapter 5

Learning Compact State Representations

The last chapter provided an overview of the compressor-controller design and training method for our general Atari playing agents. The agents consist of two separate components, a compressor and a controller, that are responsible for state representation and policy learning, respectively. The role of the compressor is crucial, because, by providing compact representations for observations, it allows us to scale topology and weight evolving neuroevolution algorithms to reinforcement learning problems with high-dimensional input spaces. Fig. 5.1 highlights the role of the compressor in an agent.

This chapter focuses on identifying and subsequently evaluating a suitable autoencoder model and training method that allows us to train a compressor in an unsupervised manner, without the need for domain specific knowledge, such as labelled observations. We start by defining a design space over a number of different types of autoencoders, network architectures, representation sizes, and loss functions. Following this, we evaluate each candidate in the design space by examining both their reconstruction quality and ability to encode the important state variables identified by the Atari Annotated RAM Interface (AtariARI). Our aim is to identify a method that balances the trade-off between representation quality and size. This is crucial for evolving effective game-playing policies because, although the policy learner requires access to rich information, the state representations produced by the compressor must also be compact enough to ensure that the search space is of a manageable size.

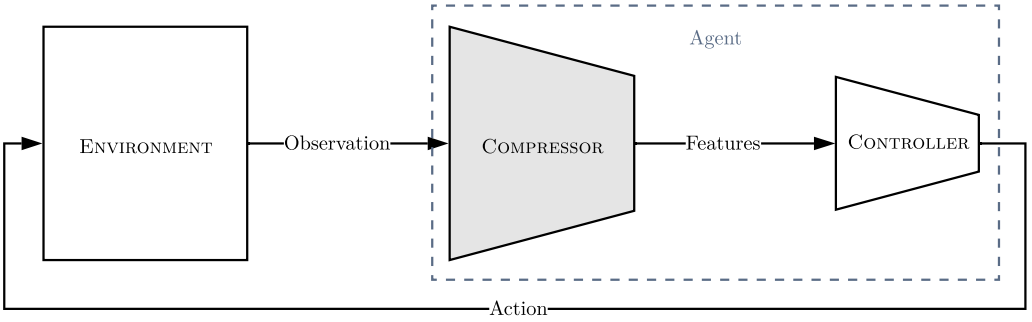


Figure 5.1: A recap of the role of the compressor in agents that use a compressor-controller design.

The first section of this chapter defines the design space of autoencoder configuration candidates that we evaluated in our search for a suitable state representation learning method. It also includes a description of a customised loss function we proposed for improving representation quality. The second part of this chapter details our procedure for identifying the best candidate from our design space to use moving forward. Finally, we describe the results of our evaluations and, based on these results, choose a candidate to use in our overall evaluation of AE-NEAT in Chapter 7.

5.1 Design Space

As we have previously discussed, autoencoders have been used, with great success, as a method of lossy compression. We define and evaluate a design space of autoencoder configurations because although their exists some prior work that have evaluated autoencoders for representation learning (Anand et al., 2019, Poulsen et al., 2017), there is a lack of knowledge about the relationship between representation size and other design factors.

In this section, we describe our autoencoder design space, that is, the search space over which we search for a suitable model to use moving forward. The following subsections describe the 3 autoencoder types, 2 architectures, 10 representation sizes, and 2 different reconstruction error measures we investigate in our design space. This gives a total of 120 models.

5.1.1 Autoencoder Types

We identified three different types of autoencoders that we thought held promise as state representation learners: undercomplete (AE), variational (VAE), and disentangled variational (β-VAE) autoencoders. §2.2 described the differences between each of these types. We evaluate each of these types of autoencoders as part of our design space.

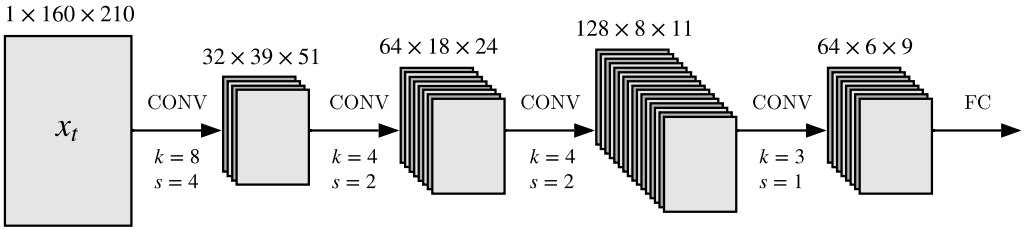
To recap, each of these types of autoencoders learns a different type of representation. For undercomplete autoencoders, there are no additional constraints on the representation learned, other than that it includes the information required to reconstruct the input. In comparison, variational autoencoders are trained to learn the generating distribution of the data. This results in a continuous encoding of the inputs, in which similar inputs are located close to each other in latent space. A further extension of this is disentangled variational autoencoders. These also try to learn the generating distribution for the data, but greater emphasis is placed on each variable of the representation encoding different, independent information.

5.1.2 Architectures

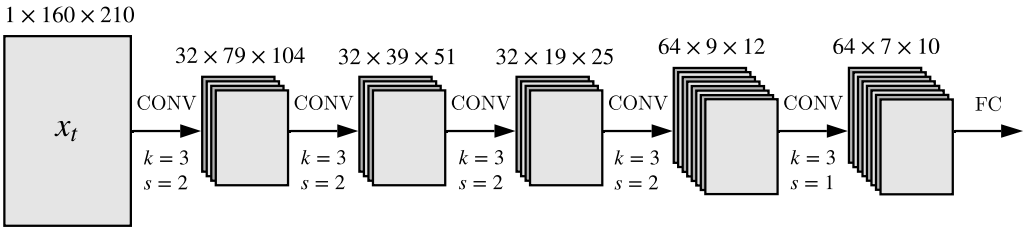
As part of our design space, we test two different encoder/decoder architectures, one based on the convolutional layers of the DQN network architecture published by Mnih et al. (2015) that matches the architecture used by Anand et al. (2019), and another inspired by the results of ResNet (He et al., 2015). These architectures are shown in Fig. 5.2. k and s denote the filter size (i.e. $k \times k$) and stride of the convolution operations in each layer, respectively.

Our motivation behind the use of the DQN-inspired architecture was the good performance of gradient-based reinforcement learning methods that use this architecture, such as DQN. We wanted to see how well the convolutional layers of this architecture perform when trained and assessed independently of the reinforcement learning aspect. The kernel sizes of this architecture match those of the original paper, while the number of filters for each layer matches those used by Anand et al. (2019) in their autoencoders based on this same architecture.

We also trained models that used an alternative architecture, the Small Kernel architecture, that consists solely of stacks of 3×3 convolution layers. The motivation behind



(A) The DQN-inspired encoder architecture.



(B) The Small Kernel encoder architecture.

FIGURE 5.2: The encoder architectures evaluated in our design space.

this architecture design is that we can achieve the same receptive field size as larger 5×5 or 7×7 convolution layers by instead stacking two or three 3×3 convolution layers respectively (Karpathy, 2015), while requiring fewer parameters. Furthermore, the deeper stacks of convolution layers contain more nonlinearities that allow for more expressive features to be extracted (Karpathy, 2015). By replacing the larger 8×8 and 4×4 kernel sizes in the DQN-inspired architecture, the resulting Small Kernel architecture contains less than half the number of parameters, 2,240 as opposed to 5,984.

Each of the architectures takes as input a single-channel, greyscale 160×210 pixel image. The DQN-inspired architecture consists of four convolutional layers, each convolving 32, 64, 128, and 64 filters, respectively. The first filters are created by convolving an 8×8 kernel, the second layers use 4×4 kernels, while the fourth used a kernel size of three. The exact dimensions of the resultant filters and kernels, the stride size, and the number of filters are detailed in Fig. 5.2. In comparison, the Small Kernel architecture has five layers, all of which use 3×3 kernels. The first three layers have 32 filters, while the last two have 64 filters. All models have a final fully connected layer that connects the flattened output of the final convolutional layer to a bottleneck of a particular size. Descriptions of the bottlenecks for undercomplete and variational autoencoders are

given in §2.2. The decoders of each model are the reverse of the compressors/encoders described.

5.1.3 Representation Size

One of the largest gaps in the current knowledge on representation learning is how representation size affects performance from a policy learning perspective. This is particularly relevant when considering evolutionary policy learning, where the size of the input space is essential. To see how representation size affects performance in representation learning for Atari games, we evaluate each architecture and autoencoder type using 10 different representation sizes, ranging from 10 to 100 dimensions.

Although others have experimented with one or multiple of these types of architectures and autoencoder types, to the best of our knowledge, none have evaluated performance over a set of games, crucial for general video game playing, and none have sufficiently explored the impact of representation size on the performance of each type. It is quite conceivable that the relative performance of each type of autoencoder is different at different representation sizes. Representation size is of greater importance to us because it directly impacts the ability to evolve policy learners.

5.1.4 Reconstruction Loss Measure

To avoid the need for labelled data that would introduce game-specific dependencies, the compressor is trained in an unsupervised fashion. As discussed previously, the compressor is simply the encoding half of an autoencoder which is trained to reconstruct gameplay frames. One issue that this approach raises is that since autoencoders are typically trained to minimise the overall reconstruction error, small features that are important for playing the game, but contribute relatively little to the overall reconstruction error, are lost in the reconstructions. This problem is particularly pronounced in some Atari games, where critical features for policy learning, such as the location of the ball for the game of Pong, are only a few pixels in diameter.

One solution is to take advantage of the fact that the images passed to the compressor come from a sequence, and thus it is possible to identify moving portions of the gameplay images by performing image differencing. By incorporating this information in the loss

function, we can penalise the compressor for the poor reconstruction of important objects without explicitly labelling them. This approach seems particularly well suited for Atari games, which often consist of a static or largely static background behind moving or changing objects. Small static details are likely to be well reconstructed because they are present in every frame.

The idea of using an image differencing-based loss function to focus on small, dynamic objects in Atari game frames was first postulated by Sandven (2016), but was not investigated until Nylend (2017). Our implementation is inspired by the implementation used by Nylend (2017), but differs in how the weighting of each pixel is calculated. Through the use of the AtariARI and a larger number of games in our evaluations, our experiments provide a more comprehensive evaluation of this idea.

5.1.4.1 Weighted Reconstruction Loss Formulation

First, we create a weight matrix $W_{m \times n}$ that matches the dimensions of the frames provided by the environment. For Atari games, these frames are of size 210×160 pixels, which are greyscale conversions of the raw RGB images provided by the emulator. The entries of W are given by:

$$w_{ij} = \begin{cases} \alpha & x_{ij,t} \neq x_{ij,t-d} \\ 1 & otherwise \end{cases} \tag{5.1}$$

where $x_{ij,t}$ is a pixel value in the current frame at time step t, and $x_{ij,t-d}$ is the value of the corresponding pixel in a previous time step $t - d$, d steps in the past. α is a configurable hyperparameter that acts as a multiplier for the pixel error at each position.

This weight matrix does not take into account the magnitude of the difference between $x_{ij,t}$ and $x_{ij,t-d}$, as the magnitude of the colour difference between the pixels is not indicative of the significance of the change, i.e. larger changes in pixel colour do not necessarily indicate more significant differences in the game state. The weight matrix can be utilised in any pixel-wise reconstruction error metric, a weighted equivalent of the sum of squared error loss function is given by:

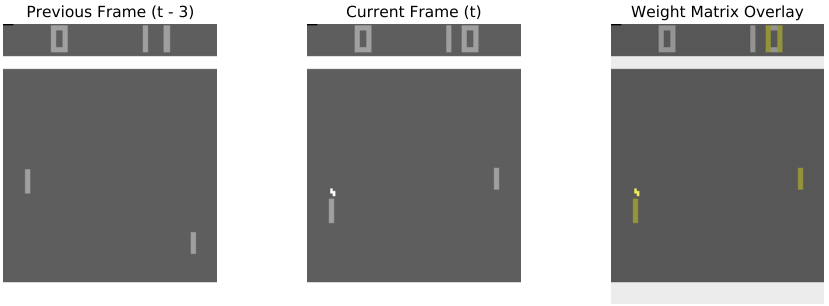


FIGURE 5.3: The resulting weighted loss overlay given a pair of current and previous Pong images.

$$WSSE = \sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} \times (x_{ij,t} - \hat{x}_{ij,t})^2 \tag{5.2}$$

In our experiments, we use empirically determined values of $\alpha = 4$ (i.e. dynamic pixels are four times as imported as static pixels) and $d = 3$. Our choice of $d > 1$ is because the positions of all objects do not necessarily update in every frame, and the larger delay captures more of the movement of moving objects.

5.1.4.2 Weighted Reconstruction Loss Examples

The effect of using our weighted reconstruction error is that the compressor is penalised more for poorly reconstructing regions of the image that objects have moved from and to. Fig. 5.3 shows the regions of the image that are more highly penalised in an example from the game Pong. Pong represents an ideal case for this loss function, as the background remains static during the game and all foreground objects move or change. Fig. 5.4 illustrates how the use of the loss function draws attention to the objects that are important for policy learning in all games.

5.2 Atari Gameplay Dataset

Evaluating each candidate in the design space (§5.1) requires a dataset of gameplay images for each game that can be used for training and evaluation. Two important considerations for this dataset are that it contains a sufficient volume of images for each

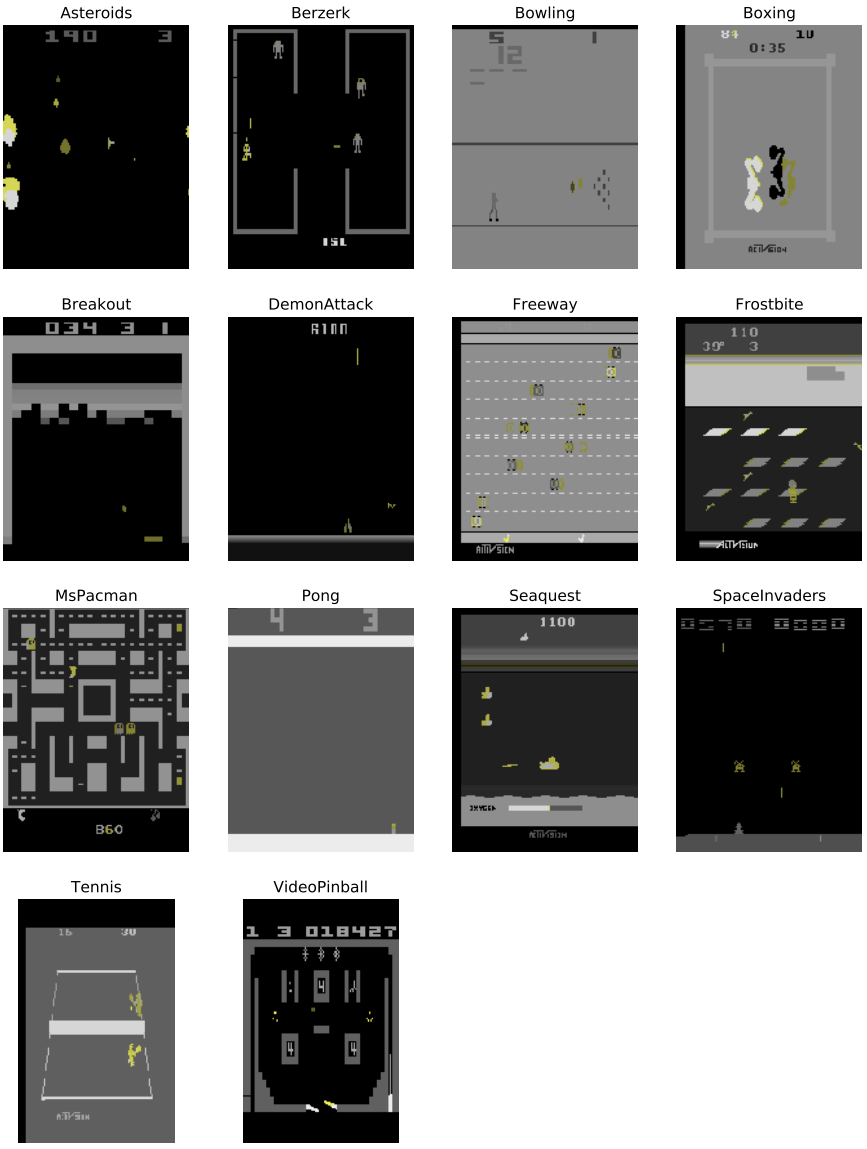


FIGURE 5.4: Weighted loss overlays for a randomly selected observation for each game used in our evaluations.

game to train deep neural networks, and that these images represent a diverse set of game states. Both requirements are important to ensure that the candidates are able (and assessed on their ability) to learn representations that generalise to many different game states. Assessing performance across games is important to ensure that the models

generalise well across games, as each game tends to look very different to each other and have different objects of different shapes and sizes of interest.

Datasets of Atari gameplay images have been collected by others in the past, for the purposes of pre-training deep RL models (Kimura, 2018, Sandven, 2016), state representation learning (Anand et al., 2019, Nylend, 2017, Sherburn, 2017, Wang et al., 2017), and other RL-related tasks (Carr et al., 2019, Tucker et al., 2018). However, these datasets are not publicly available, so we created our own. Before discussing the details of our dataset generation process and characteristics of the created dataset, we first discuss the considerations made about the process of collection.

5.2.1 Data Collection Considerations

Three possible methods for collecting gameplay images are to use a human player, a random agent, or a trained agent. The advantage of collection using a human player is that they are able to explore and collect images for a diverse set of game states by employing a variety of different strategies, provided that they can play the game competently. However, due to the time required to learn to play each game at a high enough level to reach later game states, and to collect enough images to train on, in the order of tens of thousands, it is infeasible to achieve this in reasonable time, particularly for a collection of games.

Using a random agent, which at each time step performs a random action until the end of the episode, to collect gameplay images alleviates the issue of being unable to collect enough images. Left for long enough, a random agent is able to collect an essentially unlimited number of images. However, what the random agent makes up for in quantity, it suffers in diversity. While a random agent can explore many game states, the limitation is that only states that lie close to the initial state and are easily reachable by performing a random sequence of actions will be visited. A compromise between these two exploration extremes is to use a trained agent.

As with the random agent, using a trained agent we would be able to collect an effectively unlimited number of images (even if the agent executed the same strategy during each episode, due to the stochasticity in the environments). However, since the agent is able to progress in the game, it would be able to collect a greater diversity of images that

represent states farther from the initial starting states of the game. The compromise compared to human player collection is that the trained agent may introduce more bias into the dataset by including only images for game states that are encountered when using a particular strategy. However, this trade-off is necessary to collect enough images for training and testing, and address the issues associated with random agent collection.

Based on the above considerations, collecting game state images using a trained agent is a good compromise between human player and random agent collection strategies. For these reasons, we used a trained agent to collect our data.

5.2.2 Data Collection Details

The images for our trained agent dataset were collected using trained Proximal Policy Optimisation (PPO) agents (Schulman et al., 2017). The implementation of PPO used was provided by the Stable Baselines reinforcement learning algorithm library (Hill et al., 2018). Each agent was trained using the same hyperparameters as in the original paper, and trained for the same 10 million time steps. The training curves and final performance for each agent are included in Appendix A.1. Using the agents, we collected 100,000 training, 10,000 validation, and 10,000 testing images for each game.

To briefly examine the differences in game states explored by random and trained agents, we compared the data collected by a random agent against the data collected by the trained agents. Since it is difficult to measure exploration directly, as a proxy we used the distribution of pairwise ℓ_1 distances between images to visualise the differences between the two datasets. While we observed differences between the distributions for all games, this is a particularly suitable and easily interpretable metric for Breakout, where exploration can be measured by the number of and different combinations of bricks that are broken by the agent. The pairwise difference distributions for Breakout are shown in Fig. 5.5. We can see that the trained agent visited states with many different combinations of broken bricks, leading to a much flatter pairwise difference distribution compared to the random agent. Since the random agent was never able to break more than a few of the bricks in the lowest levels, the only game states explored were those close to the starting state, where all the bricks are present. This corresponds to a very narrow difference distribution, where the differences in the collected images are mostly caused by differences in the player and ball positions.

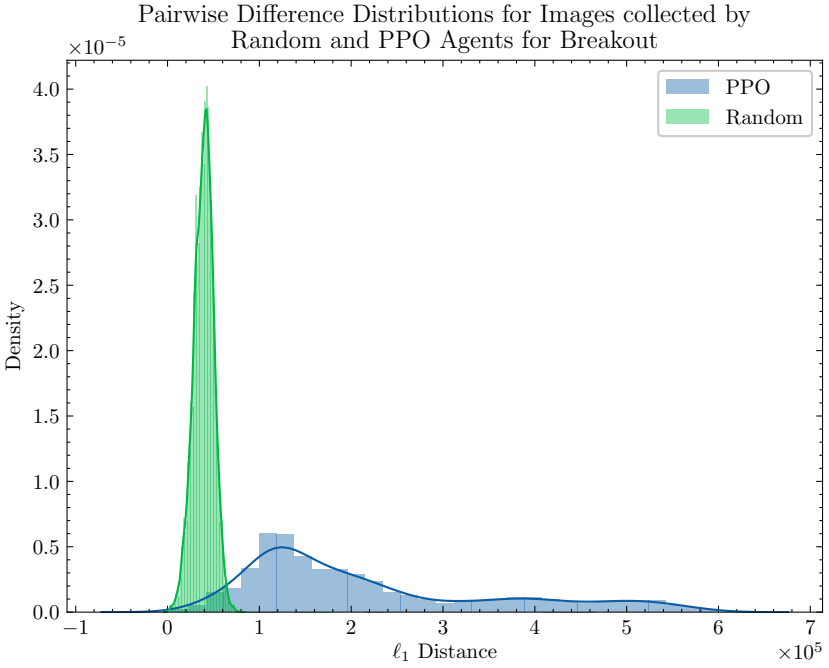


FIGURE 5.5: Pairwise distance distributions for images in the randomly collected and trained agent collected datasets for Breakout.

5.3 Evaluating Representation Quality from a Policy Learning Perspective

Autoencoders are trained to learn compact representations of the training data that contain the information required to reconstruct the input. As such, they are typically evaluated by assessing the quality of the reconstructions they produce. However, for state representation learning, we are primarily concerned with the quality of the encoding of important state variables, rather than the quality of the reconstructed images. Since the learning objective during training does not necessarily align with our goal, measuring reconstruction quality alone is insufficient for assessing the quality of the learned representations from a reinforcement learning standpoint. For example, minimising the reconstruction error does not ensure that the important state variables can be easily extracted from the latent space encoding of the input. Nor does it ensure that small, yet crucial details, such as the position of ball in Breakout, Pong, and Tennis, will be retained, since they contribute very little to the overall reconstruction error.

To assess the quality of the representations learned by the autoencoders in our design space from a reinforcement learning perspective, we used the Atari Annotated RAM

Interface (AtariARI) (Anand et al., 2019). This interface was described in §4.5. Before going into the details of how the state representation learning evaluations were performed, we first take a more in-depth look at the properties of good state representations.

5.3.1 Properties of Good State Representations

For efficient policy learning, state representations should encode the essential information about the environment and discard anything irrelevant (Lesort et al., 2018). In the context of video games, important information includes the positions of objects and the values of scores. Textures, colours, and background objects are often included in games for aesthetic reasons, but they are irrelevant aspects of the environment with respect to the objective of the game.

In the case of reinforcement learning methods that attempt to learn a value function, Böhmer et al. (2015) defined four properties that good state representations should adhere to:

- They should satisfy the Markov property, i.e. all the information required by the agent to decide on an action should be present in the current state.
- They should contain enough information to estimate the true value of the current state well enough to enable policy improvement.
- The must generalise to unseen states with similar features.
- They must be low-dimensional enough for efficient estimation of the value of the state.

However, for our method, only the last two properties are required, as our policy learning method is able to evolve memory and does not estimate a value function. Instead, the search takes place directly in the policy space. Furthermore, the desire for low-dimensional representations is not to enable efficient estimation of the value of each state, but rather to enable the topological evolution of policy networks.

5.3.2 Evaluating State Representations using the AtariARI

Anand et al. (2019) created the AtariARI to evaluate the state representations learned by different state representation learning methods. In particular, those learnt using deep neural networks. Their approach to this evaluation was to train the state representation learner, fix the weights, and then train smaller *probes* that try to predict the values of different state variables at each time step from the corresponding image from the environment. For each state variable, a *probe* (a single-layer linear classification network) was trained to correctly classify the target value for each state variable. Each state variable is stored in a single byte of RAM, making this a 256-way classification problem. To compare representation quality between games, which each have different state variables, they are divided into six different categories: agent localisation, other object localisation, small object localisation, score/clock/lives/display, and miscellaneous. We follow a similar evaluation method to the one that they propose, however, we propose the use of regression probes for evaluating the representation of localisation variables (i.e. variables that encode the positions of objects) and nonlinear probes. We explain these modifications before describing the evaluation procedure.

5.3.2.1 Regression Probes

Anand et al. (2019) formulate the task of predicting each state variable as separate classification problems, regardless of the nature of each variable. While this makes aggregating the results over all state variables for each game easier, it ignores the ordinal nature of many of the variables. For instance, those that store the positions of objects along the x or y axis of the screen. When framing the tasks of predicting the values of these variables as a classification problem, an off-by-one prediction is equally as bad as a prediction that is off by 100, despite the fact that when predicting discrete variables only being off by one is an excellent prediction. Especially when the range of each variable is [0, 255]. Given the importance of localising objects for learning to play the game, it is important that this information is evaluated as accurately as possible.

As alluded to before, one of the downsides of evaluating representations using classification and regression for different state variables is that it is harder to aggregate the

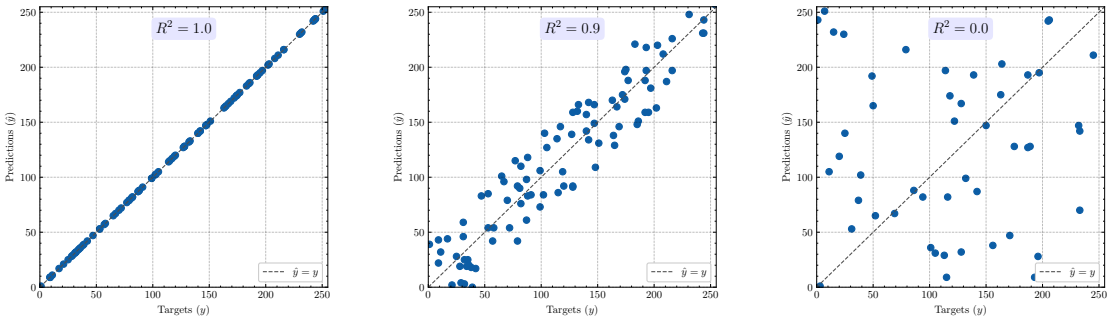
results for each game. Now, we need to report the overall performance across the categorical variables separately to the overall performance across the discrete variables. It also means that if categorical and discrete variables both exist within the categories defined by Anand et al. (2019), then we would have to break them up. Examining the state variables for each of the 14 games used in our evaluations (§4.6), we found that 78.8% of all variables are either discrete or ordinal, and thus better suited to evaluation using regression, rather than classification. Of these variables, 83.3% of them fall under the localisation category (61.4% of the total number of variables). The localisation categories are the only categories in which all the state variables for all games are better suited for regression, and therefore are the only categories that we evaluated using regression probes. Although some games have other variables, such as scores or clocks that would also be better evaluated using regression, the applicability of regression for such variables varies between games. For example, for games in which the scores achieved by the agent and/or opponent are low, such as Pong, these variables are stored in a single byte of RAM, which means that regression is suitable. However, for games where the agent can achieve a higher score than 255, as is common in endless-play games such as Video Pinball, the score is encoded over multiple bytes of RAM. Because the score is split between bytes, we cannot use regression to evaluate these variables.

Another issue we face with regression is the choice of evaluation metric. For classification, Anand et al. (2019) report the accuracy and $F1$ scores for each variable, as well as the average of these metrics over the variables within categories, and overall. This ability to aggregate the results of different variables, which have the same maximum range [0, 255], but different distributions of values and ranges in practice, is crucial for comparisons between games and categories. For regression, we require a similar metric that can (a) provide a good indication of predictive performance, and (b) be meaningfully averaged within and across categories. For this, we use the coefficient of determination, R^2, between the predicted and target values.

The R^2 value provides a good indication of model fit and predictive power of the regression probes trained for each state variable. It is defined as follows:

$$R^2 = 1 - \frac{\sum(y - \hat{y})^2}{\sum(y - \bar{y})^2} \tag{5.3}$$

where y is the target value, $\hat{y}$ the predicted value, and $\bar{y}$ the mean target value. An R^2 value of one means that the model (in this case our probe) is able to perfectly predict the target values, whereas a value of zero indicates that the model is unable to make good predictions. In this way, performance is described similarly to the classification probes using accuracy or $F1$ scores. In practice, the R^2 values can be below zero when the model is evaluated on unseen data, such as is the case for our evaluations, which use a test set. The interpretation here is the same as for a low positive value: the probe performance is very poor. Figures 5.6a to 5.6c show how we can visually interpret the prediction quality of regression probes using R^2 values.



(A) (B) (C)

FIGURE 5.6: An illustrative example of how we can visualise the R^2 values of regression probes.

5.3.2.2 Non-Linear Probes

Another change we made to the evaluation method was training nonlinear probes instead of linear ones. The rationale behind this decision is that the information on state variables may be compressed in a nonlinear manner that is not extractable using a linear classifier/regressor. This may be particularly true as the representation size decreases, and the compressors are forced to compress the state information into a smaller vector. In their initial work, Anand et al. (2019) used different methods that learnt 256-dimensional representations. These representations, although far smaller than raw images, are still very large from a neuroevolution perspective. We experimented with far smaller representations in the range of 10 to 100 dimensions. While nonlinear encoding introduces complexity, the policy networks evolved from these encoding methods are not limited to learning only linear functions, therefore we should allow the evaluation method to be flexible enough to account for this. Allowing a single hidden layer in the

probes, proportional to the representation size is a good compromise between allowing for too much nonlinearity and not assessing the true content of the representations.

One problem that introducing nonlinear probes with a hidden layer is the number of nodes that the hidden layer should contain. While there is no silver bullet for selecting the optimal number of hidden nodes, Heaton (2008) provides several rule-of-thumb methods commonly used in practice for determining the appropriate number of nodes:

- The number of hidden nodes should be between the size of the input layer and the size if the output layer.
- The number of hidden nodes should be two-thirds the size of the input layer, plus the size of the output layer.
- The number of hidden nodes should be less than twice the size of the input layer.

For simplicity, we set the number of hidden nodes in the hidden layer used by nonlinear probes equal to the size of the input layer, which falls roughly in line with the above recommendations. The nodes in the hidden layer used Rectified Linear Unit (ReLU) nonlinearities.

5.4 Experimental Procedure

This section describes the procedure followed for training and evaluating each of the candidates in the autoencoder design space defined in §5.1. It also describes the process we used for selecting the final model to use for our experiments assessing the performance of our hybrid reinforcement learning method, AE-NEAT. The order of the steps leading to the selection of our final compressor model is illustrated in Fig. 5.7.

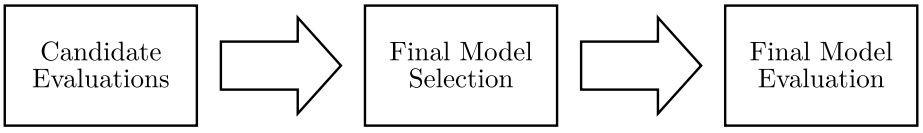


FIGURE 5.7: An overview of the steps leading to the selection of our final compressor model.

5.4.1 Candidate Training

Our design space included 120 different candidates (§5.1). We trained each of these candidates on a set of four games – Asteroids, Bowling, Ms Pacman, and Pong – resulting in 480 different models. This ensured that the total training time for all of the models remained feasible, and allowed us to check that the final model chosen from the 120 candidates generalised to the remaining 10 games. Asteroids, Bowling, Ms Pacman, and Pong were chosen as the subset of games because they are (a) very different in appearance, (b) contain a wide range of objects of different shapes and sizes, and (c) represent a range of complexity in terms of their game states.

The learning rate, maximum number of training epochs, dynamic pixel weighting multiplier $\alpha = 4$ (§5.1.4), and the KL divergence weighting $\beta = 4$ for the disentangled variational autoencoders (§2.2.2.3) were chosen through informal experimentation prior to our evaluation of the design space. Each model was trained using our Atari gameplay dataset (§5.2) for a maximum of 50 epochs. We used early stopping with a patience of 10 epochs to stop training early if performance plateaued. We used a learning rate of $1e^{-4}$ and a batch size of 64.

5.4.2 Candidate Evaluation

We trained the regression and classification probes following the same procedure outlined by Anand et al. (2019). For each game, we collected 45,000 unique frames and the values of the state variables provided by the AtariARI at the corresponding time step using the same trained PPO agents used to collect the images for training the Atari gameplay dataset (§5.2). These images were split into 35,000 training, 5,000 validation, and 10,000 test images. A probe was trained for each state variable for each game, using the Adam optimiser and a learning rate scheduler with an initial learning rate of $5e^{-4}$. Each time the validation plateaued for five epochs on the validation set, the learning rate was decreased by a factor of 0.2, to a minimum of $1e^{-5}$. Each probed was trained for a maximum of 100 epochs, but with early stopping if the validation loss plateaued for 15 epochs. Classification probes used cross-entropy loss, whereas regression probes used mean squared error loss. Only the candidate chosen as our final model was evaluated on the test set.

5.4.3 Final Model Selection from the Candidates

The task of identifying the "best" model from our pool of candidates was non-trivial. Maximising the quality of the encoding of each state variable across all the games while also minimising representation quality is a multi-objective optimisation problem with tens of objectives. Therefore, to narrow the search for a good model, we first selected a representation size based on the trends in AtariARI probe prediction quality observed for the different categories of state variables across the four games (Asteroids, Boxing, Ms Pacman, and Pong). After identifying a suitable representation size, we compared the average localisation and classification performance across all variables and all games for the remaining models. The final chosen compressor model was one that has good performance with respect to both of these categories.

5.5 Results

We start by analysing the performance of each candidate on a per-game basis for Asteroids, Boxing, Ms Pacman, and Pong. Following this, we take a holistic look at performance across all of these games to identify a candidate to move forward with in the next chapter. Finally, we evaluate the performance of the final compressor model across all 14 games of interest.

5.5.1 Candidate Evaluations

The first games we analyse are those with the simplest game states, Boxing and Pong, followed by the games with more complicated game states, Asteroids and Ms Pacman.

5.5.1.1 Boxing

For Boxing, the AtariARI provides access to seven state variables. These store the x and y positions of the player and opponent, their scores, and the value of the clock.

Localisation Performance The reconstructed images produced by each candidate show that all of them were able to consistently and accurately reproduce the position

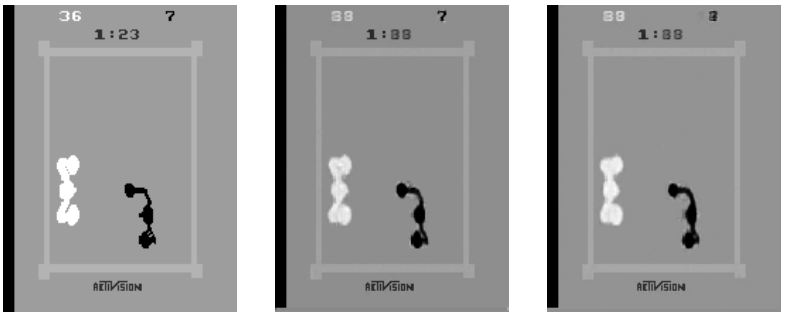
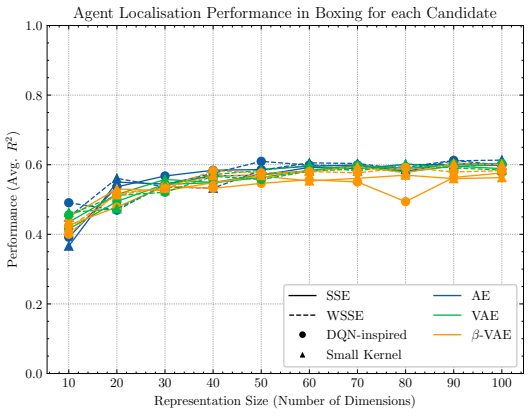


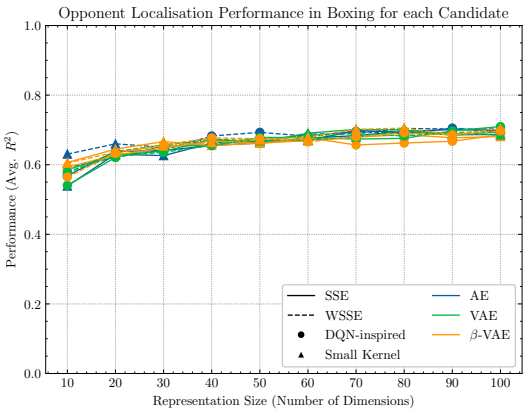
FIGURE 5.8: Original (left) and reconstructed images for the Small Kernel, undercomplete autoencoder using the standard mean squared error reconstruction loss and a representation size of 20 (middle) and 10 (right).

and shape of both the agent and the opponent, even candidates constrained to the lowest representation size of 10 dimensions. A comparison between the reconstruction quality for two (otherwise identical) candidates with representation sizes of 20 and 10 dimensions is shown in Fig. 5.8.

The consistent localisation performance of all candidates is reflected in the AtariARI evaluation results shown in figures 5.9a and 5.9b, respectively. These figures show that localisation performance was fairly consistent across all candidates, even as representation size decreased. However, given the quality of the reconstructions of the agent and opponent, it is surprising that the average of the R^2 values over the x and y positions were not higher. An interesting observation is that the drop in performance exhibited as the representation size dropped from 20 to 10 dimensions is not noticeable in the reconstructions (see Fig. 5.8).



(A) Average agent localisation performance over the x and y variables.



(B) Average opponent localisation performance over the x and y variables.

FIGURE 5.9: Object localisation performance in Boxing for each candidate.

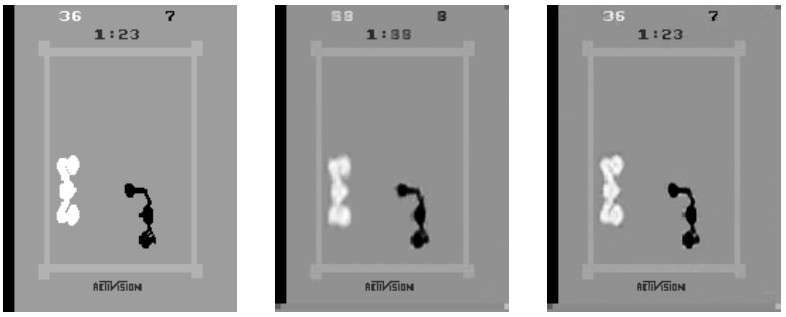


FIGURE 5.10: Original (left) and reconstructed images for the best variational (middle) and undercomplete (right) autoencoders with a representation size of 100 dimensions. Both models used the DQN-inspired architecture and the sum of squared errors (SSE) reconstruction loss.

Classification Performance The differences between the representations learned by different candidates are clear when measuring their ability to learn the values of the agent's and opponent's scores, and the value of the clock. Here, the biggest contributors to the differences in performance between candidates were autoencoder type (AE, VAE, or β-VAE) and representation size. In the reconstructions, variational autoencoder candidates (standard or disentangled) were consistently unable to clearly reproduce the score and clock values in the original images, regardless of representation size, reconstruction error measure, or architecture. In contrast, the undercomplete autoencoders were able to perfectly reconstruct these values with large representation sizes, although the reproduced values became blurrier as representation size decreased. This difference is highlighted in the reconstructions produced by the best undercomplete and variational autoencoders shown in Fig. 5.10. As with the variational models, the reconstruction error measure (SSE vs. WSSE) and architecture (DQN-inspired vs. Small Kernel) did not have a substantial or consistent impact on performance for undercomplete autoencoders.

These results were mirrored in the AtariARI evaluations, shown in Fig. 5.11. The variational autoencoders (both standard and disentangled) performed consistently poorly for all representation sizes. However, for the undercomplete autoencoders, we observed a gradual decrease in performance as representation size decreased. While it is difficult to identify differences in performance between variational and disentangled variational autoencoders using the reconstructions, the AtariARI evaluations show that standard variational models consistently outperformed their disentangled counterparts.

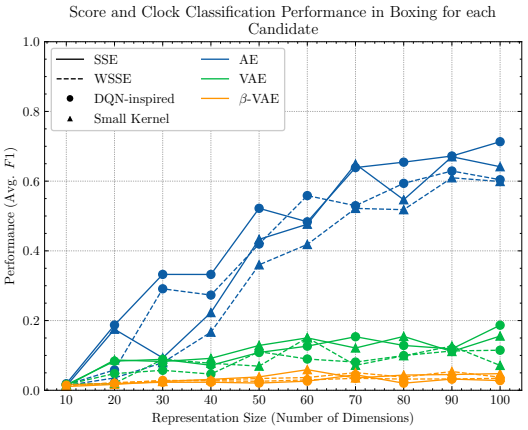


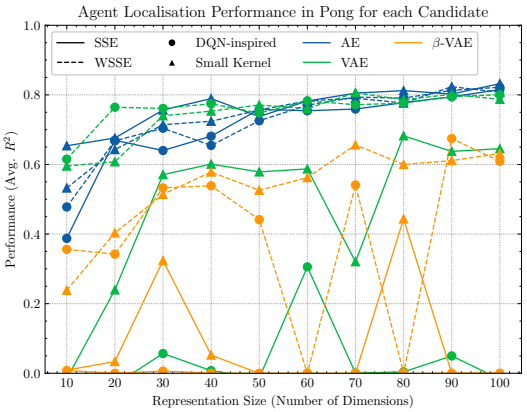
FIGURE 5.11: Average classification performance over the the player and opponent scores, and the clock value in Boxing for each candidate.

5.5.1.2 Pong

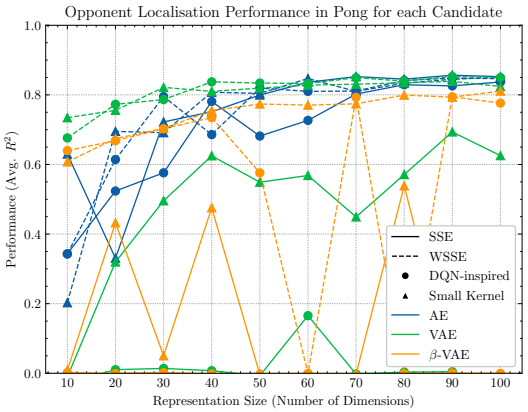
For Pong, the AtariARI exposes eight state variables. These are the x and y positions of the player's paddle, the opponent's paddle, and the ball; as well as the player's and opponent's scores.

Localisation Performance First inspecting the paddle localisation performance of the candidates, shown in figures 5.12a and 5.12b, it is clear that some candidates were very successful. The biggest contributors to differences in performance were autoencoder type and reconstruction error measure. For undercomplete (AE) candidates, the choice of reconstruction measure and architecture mattered very little. All undercomplete candidates performed similarly for each representation size and exhibited the same trends in performance as representation size decreased.

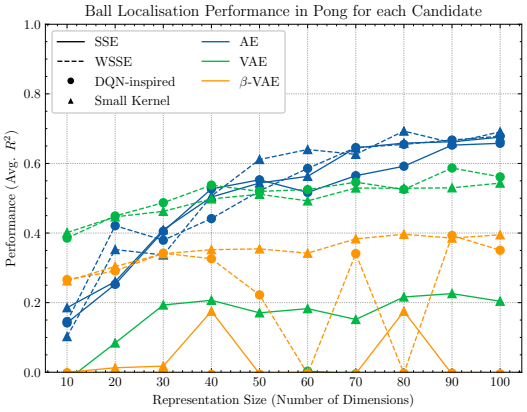
The most interesting differences occurred for the variational autoencoder candidates (both standard and disentangled). Starting with the standard variational candidates (VAE), both architecture and reconstruction error measure impacted performance. Without the use of the weighted reconstruction error (WSSE), the VAE candidates performed worse than the undercomplete candidates. Furthermore, the Small Kernel candidates were far higher performing across the spectrum of representation sizes. However, when WSSE was used, this difference was eroded, and the performance of all VAE candidates vastly improved. In fact, with large representation sizes, the performance matched that of the undercomplete autoencoders and exhibited less of a decay as the representation size decreased. Interestingly, paddle localisation performance was also improved through



(A) Average agent localisation performance. The average over x and y localisation.



(B) Average opponent localisation performance. The average over x and y localisation.

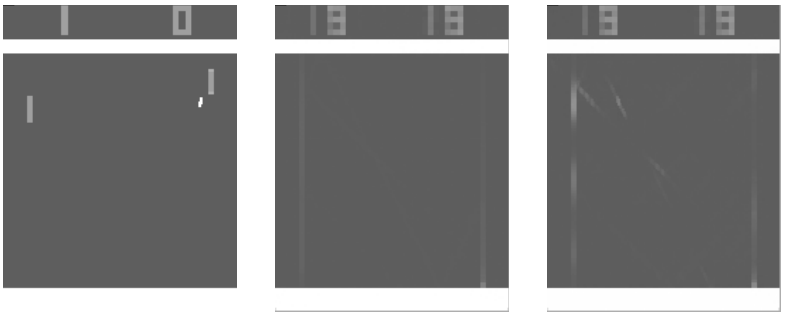


(C) Average ball localisation performance. The average over x and y localisation.

FIGURE 5.12: Object localisation performance in Pong for each candidate.

the use of the weighted reconstruction error measure for disentangled variational autoencoders, but this was less apparent in the reconstructions. Although the R^2 values for the agent and opponent paddles were around 0.6 and 0.75, respectively, when the weighted reconstruction error measure and the largest representation size of 100 were used (compared to around 0 when the unweighted reconstruction error measure was used), the reconstructions did not necessarily reflect this, as shown in Fig. 5.13. Particularly in the case of the opponent's paddle, although the AtariARI performance almost matched that of the best performing candidates, the reconstructions were far less accurate and there appeared to be far more uncertainty.

The trends in the localisation performance for the ball (Fig. 5.12c) as the representation size decreased were similar to that of the paddles, however, there were some differences. First of all, for larger representation sizes, there was a clearer delineation in



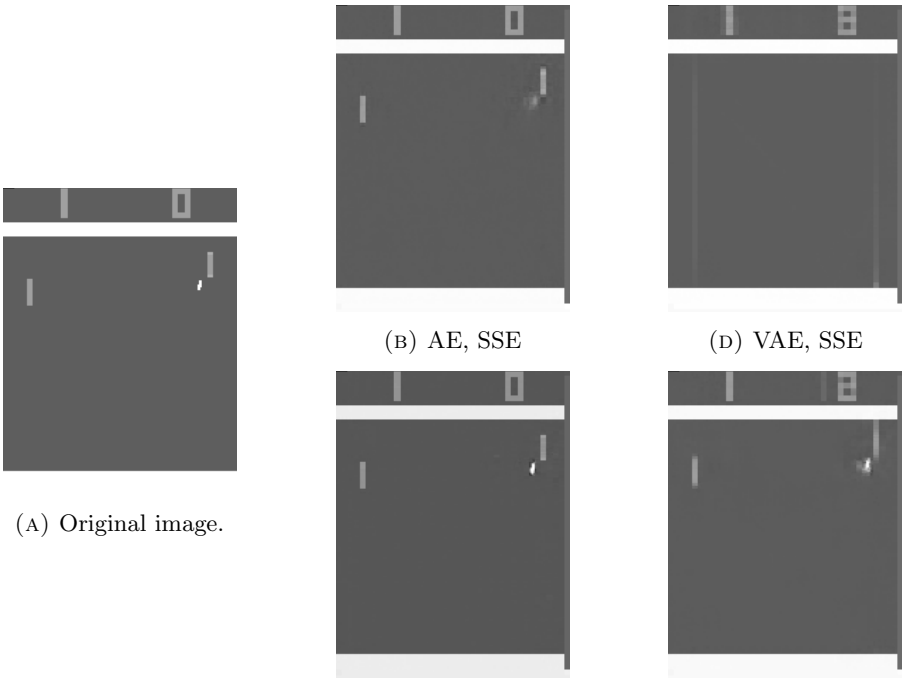
(A) Original. (B) Model using SSE (C) Model using WSSE

FIGURE 5.13: Disentangled variational autoencoder reconstructions for Pong using the Small Kernel architecture and a 100-dimensional representation.

performance between different groups of candidates. Performance for the undercomplete models was similar and the best, followed by variational models that used the weighted reconstruction error measure, disentangled variational models that used the weighted reconstruction error measure, the standard and disentangled variational models that used the unweighted reconstruction error measure. Once again, the Small Kernel architecture outperformed the DQN-inspired architecture in standard variational models that used the unweighted reconstruction error measure. This difference was eroded when the weighted reconstruction error measure was used.

At the lower end of the representation size spectrum, we observed a steep decline in performance for undercomplete models below a representation size of 40. Similar to the paddle localisation results, variational models that used the weighted reconstruction error measure were more tolerant to reduced representation size and as a result, despite performing worse for larger representation sizes, they surpassed their undercomplete counterparts at this point. Interestingly, despite the reconstructions of the ball remaining clear and accurate when the representation size was low, particularly when using the weighted reconstruction error measure, the AtariARI performance dropped drastically. This conflict is evident in Fig. 5.14.

One interesting difference between the results for Boxing and Pong is that the performance of the disentangled variational autoencoders was far more erratic for Pong, with large spikes and drops in performance between small changes to the representation size. It seems unlikely that representation size was the cause of this, and more likely indicates that these models are far more unstable to train than their standard variational and undercomplete counterparts.



(A) Original image.

(B) AE, SSE

(C) AE, WSSE

(D) VAE, SSE

(E) VAE, WSSE

FIGURE 5.14: A comparison between the image reconstruction quality for different Pong models that use the DQN-inspired architecture and a representation size of 30. (A) Shows the original image, (B) and (C) the AE candidates using SSE and WSSE respectively, and (D) and (E) the VAE candidates using SSE and WSSE respectively.

Classification Performance As witnessed for the Boxing results, many models showed a sharp decline in performance when the representation size dropped from 20 to 10. This was most apparent in their ability to learn the scores, as shown in Fig. 5.15. This is interesting because this drastic drop in performance was not accompanied by a noticeable drop in the reconstruction quality. This may indicate that rather than the information no longer being present, it may just have been far more compressed and more difficult to extract by the probe. In comparison, the decoder is far larger and more powerful and might therefore might have been able to. The difference in the clarity of the reproduced scores between AE and VAE models (illustrated in Fig. 5.14) was reflected in the AtariARI performance.

5.5.1.3 Asteroids

Asteroids has the largest number of state variables (41) of all the games. These include the x and y positions for each of the 15 asteroids, the player, and the player's two missiles. They also include variables for the directions of the player and the player's

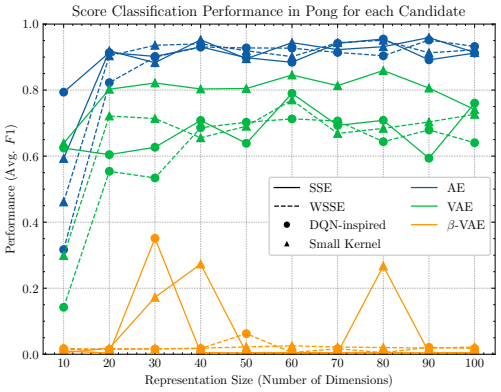
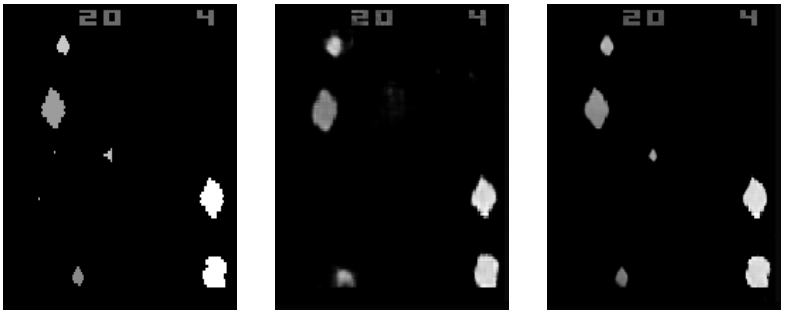


FIGURE 5.15: The average score classification performance for each candidate.

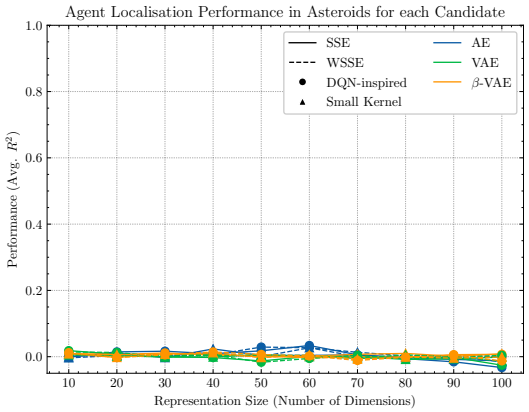
missiles, as well as two variables that encode the player's score, and a variable for the player's number of lives.

Localisation Performance At first glance, the reconstruction performance in asteroids seemed promising (see Fig. 5.16). All candidates appeared to consistently reproduce the positions of asteroids and the shape of large asteroids, even with representation sizes of just 10 dimensions. For reproducing small asteroids, it appears that the weighted reconstruction measure helped all types of models, though not all in the same way. All undercomplete autoencoders were good at reproducing the positions of the asteroids, however, for smaller representation sizes, WSSE improved the reconstruction of the shape of smaller asteroids. However, as representation size increased, the difference between WSSE and SSE models decreased. For variational models without WSSE, these were typically unable to reconstruct the positions or shapes of small asteroids. However, they were with the help of WSSE.

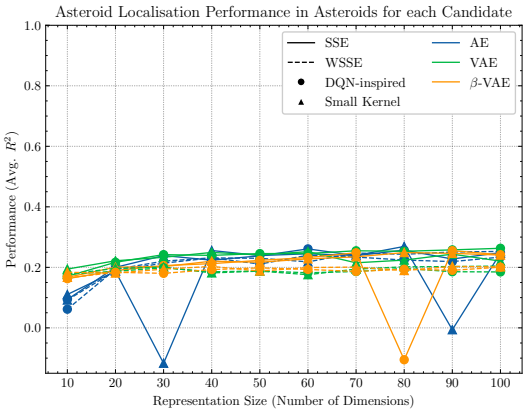


(A) Original. (B) $|Z| = 10$. (C) $|Z| = 100$.

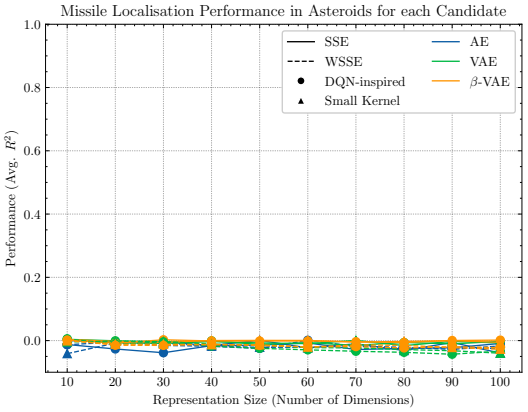
FIGURE 5.16: Typical Asteroids gameplay image reconstructions using an undercomplete autoencoder with the DQN-inspired architecture, the weighted sum of squared errors (WSSE) loss function, and different representation sizes.



(A) Average agent localisation performance. The average over x and y variables.



(B) Average asteroids localisation performance over the x and y variables for each asteroid.

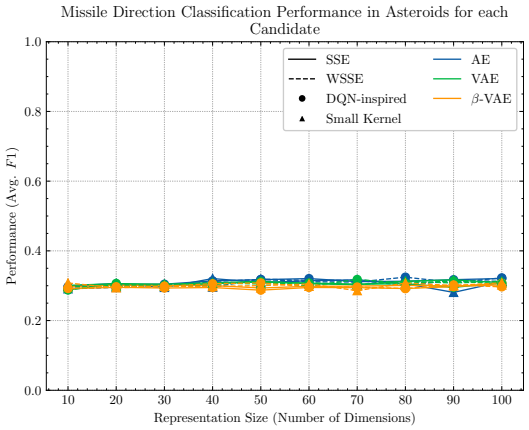


(C) Average missile localisation performance. The average over the x and y variables.

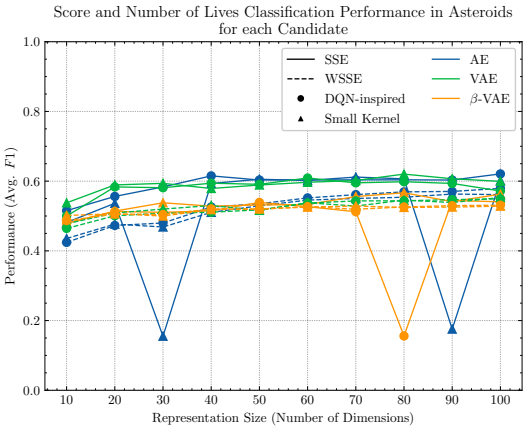
FIGURE 5.17: Localisation performance in Asteroids evaluated using the AtariARI for each candidate.

Looking at the agent, we saw a similar trend to that of small asteroids for undercomplete autoencoders. When the representation size ($|Z|$) was too small, none of the models were able to consistently reproduce the location of the agent. However, the use of WSSE made the agent appear in reconstructions earlier than models that used SSE, typically around $|Z| = 40$ as opposed to $|Z| = 50$. For variational models, only those with WSSE and representation sizes of approximately greater than 50 were able to reproduce the position of the agent. One thing that all models were uniformly poor at was the reconstruction of the agent's missiles, though this is not surprising given that they are only a few pixels wide, even smaller than the ball in Pong.

Interestingly, the AtariARI evaluation results for Asteroids, displayed in Fig. 5.17, painted a different picture. For the missile localisation, all models performed poorly, consistently achieving R^2 values very close to zero. However, interestingly, this was also



(A) Average missile direction classification performance.



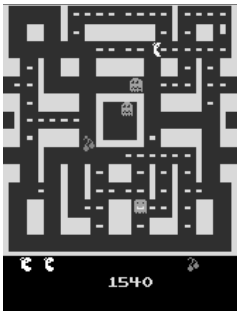
(B) Average classification performance over the score and number of lives variables.

FIGURE 5.18: Classification Performance in Asteroids evaluated using the AtariARI for each candidate.

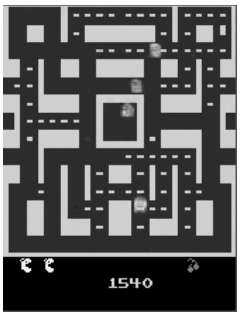
true for agent localisation, despite the position of agent being reconstructed well by some candidates, particularly when the representation size was large. It is also surprising that asteroid localisation was low for all candidates, around $R^2 = 0.2$ for all representation sizes.

Classification Performance All candidates were able to accurately reproduce the number of lives the agent has. Score reconstruction, however, was inconsistent. While all candidates were able to consistently reproduce the correct number of digits in the score, all models were inconsistent when it came to reproducing the values of the digits clearly. It appears that when the score was low, i.e. limited to two digits, all models were able to reproduce the values, but when the score was large they suffered. The AtariARI probing results showed no substantial drop in the classification performance for the score and number of lives as representation size decreased (see Fig. 5.18b).

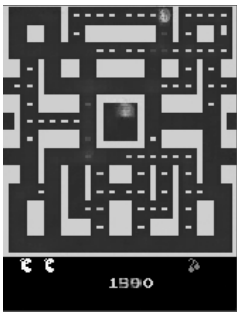
For all candidates, the missiles were never reconstructed in the reconstructions, meaning that the direction of the missiles along with their positions was impossible to see. However, the AtariARI probing results, displayed in Fig. 5.18a, indicated that despite this some information about the missile directions was still retained in the learned representations.



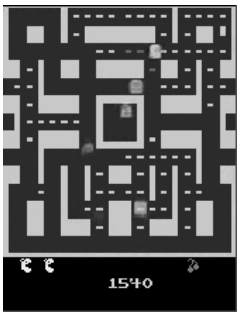
(A) Original image.



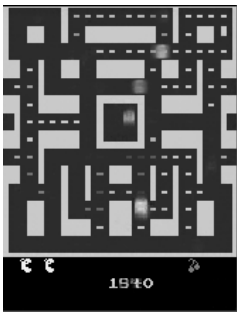
(B) AE, Small Kernel, SSE, $|Z| = 100$



(D) VAE, Small Kernel, SSE, $|Z| = 100$



(C) AE, Small Kernel, WSSE, $|Z| = 100$



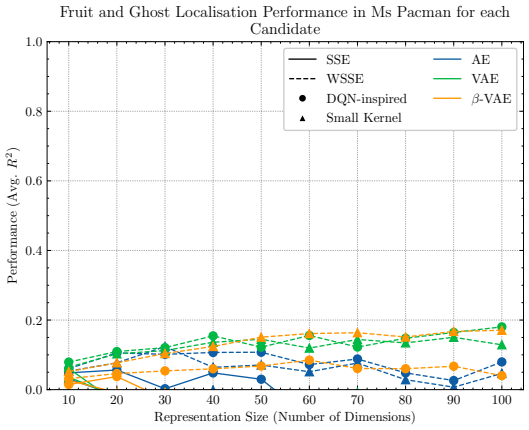
(E) VAE, Small Kernel, WSSE, $|Z| = 100$

FIGURE 5.19: Image reconstructions for various Ms Pacman models.

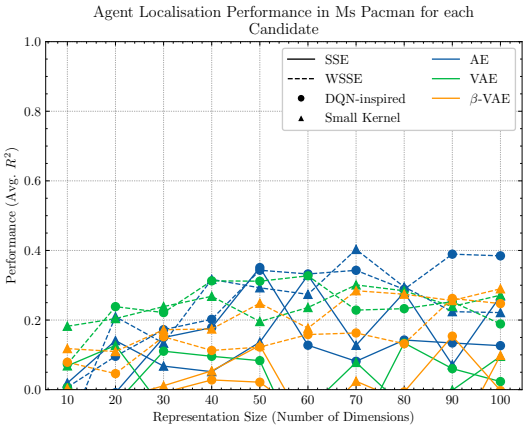
5.5.1.4 Ms Pacman

For Ms Pacman, the AtariARI exposes 17 state variables. 12 of these encode the x and y positions of the agent, four enemies, and the fruit. While the remainder encode the player's direction, score and number of lives; and the count of the number of ghosts and dots eaten. As with Asteroids, for Ms Pacman we observed similar overall results. While the reconstructions appeared to show that the representations had learned the values of important state variables, the AtariARI evaluations often showed otherwise.

Localisation Performance For undercomplete models, the fruit was only reproduced by models with large representation sizes, and only consistently in those with representation sizes of 90 or 100. Unlike the undercomplete autoencoders using the DQN-inspired architecture, the Small Kernel architecture models were unable to reproduce the fruit without using the WSSE loss. This is shown in Fig. 5.19. In comparison, none of the variational or disentangled variational models reproduced the position of the fruit (again, see Fig. 5.19).



(A) Average fruit and ghost localisation performance over the x and y variables for each.



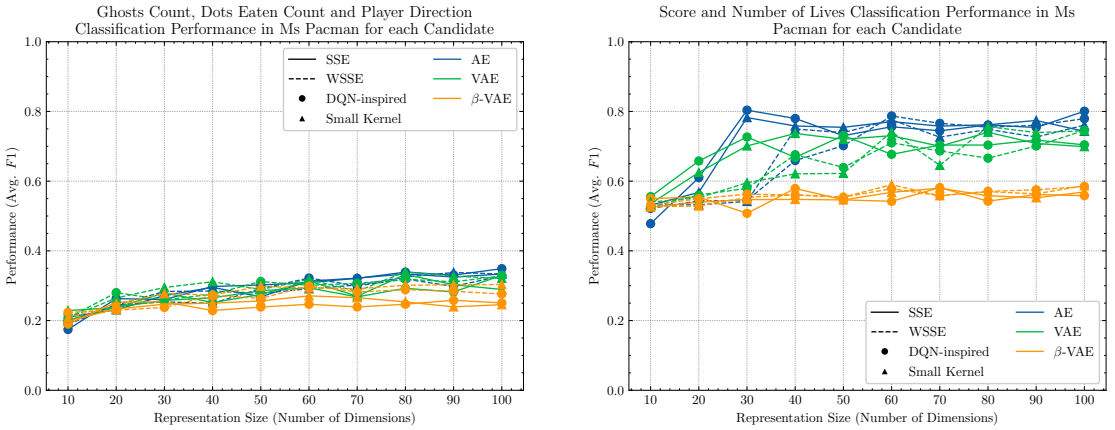
(B) Average agent localisation performance. The average over the x and y variables.

FIGURE 5.20: Localisation performance in Ms Pacman evaluated using the AtariARI for each candidate.

The position of the agent was consistently reproduced by WSSE undercomplete models for all representation sizes, but not for SSE undercomplete models with representation sizes less than around 40. Furthermore, as $|Z|$ increased, so too did the brightness of the spot where the agent was located. However, even at the largest representation sizes, the agent remained a blurry spot in many reconstructions, and it was difficult or impossible to visually ascertain the direction the agent was facing. For variational models, the location of the agent was only reproduced consistently by WSSE models with representation sizes larger than 10. For disentangled variational models, the agent's position was not consistently reproduced by any of the models.

Lastly, the undercomplete autoencoder models did the best job of reconstructing the ghosts, with blurry spots in their place for models with $|Z| >= 50$. Below this threshold, consistency was worse, but WSSE models appeared to be more consistent. However, we noticed that in some cases "phantom" ghosts that were not present in the original image were present in the reconstructed images. For disentangled variational models, the positions of the ghosts were rarely reproduced. For variational models, ghosts were reproduced only by WSSE models, with increasing accuracy as $|Z|$ increased.

Despite the consistent presence of the ghost in the reconstructions of some models, performance when evaluated using the AtariARI (Fig. 5.20a) was consistently poor. Whereas it appeared from the reconstructions that undercomplete autoencoders were the best at learning to extract the positions of the ghosts and the fruit, the AtariARI results show that the variational models consistently outperformed them for all representation



(A) Average over ghosts and dots eaten counts and player direction classification performance.

(B) Average classification performance over the score and number of lives variables.

FIGURE 5.21: Classification performance in Ms Pacman evaluated using the AtariARI for each candidate.

sizes. Finally, for agent localisation, none of the models performed particularly well, with a lot of noise present in the AtariARI probing results (Fig. 5.20b). Although, it appeared that the use of WSSE did improve performance in this regard. All models showed a slight decline as representation size decreased.

Classification Performance The best category was learning the score and lives. Here, all candidates performed far better than the other categories. As shown in Fig. 5.21b, undercomplete autoencoders (AEs) were consistently the best, followed by variational autoencoders (VAEs), and finally disentangled variational autoencoders (β-VAEs).

Looking at reconstructions alone, even at low representation sizes, all models did a fairly decent job of recreating the positions of the dots the agent must consume, with only a small number of missing or additional dots in most cases. Overall, it appears that undercomplete autoencoders produced fewer errors in this regard than variational autoencoders. Likewise, all models very consistently reproduced the correct number of remaining lives for the player. Despite good reconstructions of the remaining dots and ghosts (particularly for those models with larger representation sizes), all models were consistently poor at learning the player's direction, the dots eaten count, and ghosts count (shown in Fig. 5.21a). In this category, all models performed similarly but poorly.

Differences between candidates appeared when inspecting the reconstruction of the player's score. While all models consistently reproduced the correct number of digits, the clarity of the digits varied substantially. For undercomplete autoencoders, the

values were consistently and accurately reproduced, and discernible down to $|Z| = 20$ when using SSE. However, it appears that using WSSE hurts performance in this regard, as the values became indiscernible earlier, at around $|Z| = 30$. For disentangled variational models, none of them appear to have been able to clearly reproduce the values of digits in the score. For variational models, this also holds true except for one case: Small Kernel models that used WSSE and had large representation sizes, typically greater than or equal to 60. The AtariARI classification results are shown in Fig. 5.21.

5.5.1.5 Overall Performance

There was substantial difference between the performance of each candidate across the different state variable categories and games. There were also prominent differences in the relationships between representation size and quality for different combinations of model architecture, type and reconstruction loss measure. In some variable categories for specific games, we observed consistent performance across all the candidates, even for those with lower representation sizes (see figures 5.17a, 5.17c, and 5.18a). These results do not help us to distinguish between the candidates. In other state variable categories, we observed sharp drops or moderate declines in the representation quality only after a certain representation size threshold was passed (see figures 5.9a, 5.12a, and in particular 5.15). These categories were the most useful in identifying a good lower bound on the choice of representation size. Finally, in a few categories we observed a steady decrease in performance (of the best models) as representation size was decreased (see figures 5.11 and 5.12c). These categories required a judgement on how important the extra performance was worth as representation size increased. As a result of these different trends, there was no clear representation size that stood out as the best choice.

In the interest of balancing representation size and and quality, we opted for using a representation size of 40. In most categories, 40 represented a point before a drop in performance occurred, and in the case of ball localisation in Pong, before a steep drop in performance for undercomplete models. For the score and clock category, performance was substantially worse than for higher representation sizes, but considering that this category is assumed to be of lesser importance than localisation in the context of playing the game, this was a sacrifice we were willing to make.

TABLE 5.1: The average localisation and classification performance of each candidate with a representation size of 40 averaged over all categories of Asteroids, Boxing, Ms Pacman, and Pong. The ranks of the top four models are shown, and the final model shown in bold.

Type	Architecture	Reconstruction Loss	Localisation Performance (Avg. R^2)		Classification Performance (Avg. $F1$)	
AE	**DQN-Inspired**	**Unweighted**	**(4)**	**0.3732**	**(1)**	**0.5865**
AE	DQN-Inspired	Weighted		0.3718	(3)	0.5376
AE	Small Kernel	Unweighted		0.3649	(2)	0.5594
AE	Small Kernel	Weighted	(3)	0.3810	(4)	0.5247
VAE	DQN-Inspired	Unweighted		0.1915		0.4440
VAE	DQN-Inspired	Weighted	(1)	0.4019		0.4220
VAE	Small Kernel	Unweighted		0.2987		0.4775
VAE	Small Kernel	Weighted	(2)	0.3905		0.4229
β-VAE	DQN-inspired	Unweighted		0.1869		0.2283
β-VAE	DQN-inspired	Weighted		0.3327		0.2374
β-VAE	Small Kernel	Unweighted		0.2469		0.2979
β-VAE	Small Kernel	Weighted		0.3541		0.2371

To choose our final candidate with a representation size of 40, we calculated the overall localisation and classification scores for each of the remaining models. These are presented in Table 5.1.

Among the final candidates shown in Table 5.1, there is not a model that performs the best in both object localisation and the classification of other important state variables. The model with the highest ranking in both categories was the undercomplete autoencoder, with the DQN-inspired architecture, trained using the unweighted reconstruction loss function. This model ranked 1^{st} and 4^{th} in localisation and classification performance, respectively. This candidate was chosen as the final combination to be used as the representation learning for our evaluation of our hybrid reinforcement learning method, AE-NEAT.

5.5.2 Final Model Performance

The model that was selected for the evaluation of our hybrid reinforcement learning method, AE-NEAT, was the undercomplete autoencoder using the DQN-inspired architecture, a representation size of 40 dimensions, and the sum of squared errors loss function. Table 5.2 presents the AtariARI representation quality probing evaluations for each category of state variables for each game.

TABLE 5.2: The AtariARI probing results for each game for the final autoencoder selected from the design space.

	Localisation (Avg. R^2)			Classification (Avg. F1)	
Game	Small Object	Agent	Other	Score/Clock/Lives	Misc.
Asteroids	-0.03	-0.02	0.23	0.58	0.34
Berzerk	0.11	0.83	0.51	0.82	0.51
Bowling	0.86	0.96		0.91	0.98
Boxing		0.58	0.68	0.31	
Breakout	0.03	0.40		0.88	0.83
Demon Attack	0.03	-0.03	0.13	0.99	0.80
Freeway		0.35	0.95	0.22	
Frostbite		0.63	0.85	0.82	0.51
Ms Pacman		0.16	0.03	0.72	0.30
Pong	0.45	0.71	0.70	0.95	
Seaquest	0.36	0.69	0.35	0.88	0.75
Space Invaders	0.00	0.92	0.84	0.44	0.39
Tennis	0.27	0.81	0.82	0.64	
Video Pinball	-0.00	-0.00		0.23	

Similarly to the results observed on the smaller subset of games, the representation quality of our final model varied substantially between both categories and games. In many of the games, we observed high prediction quality in the classification state variable categories (Score/Clock/Lives and Miscellaneous). This is the case for Bowling, Breakout, Demon Attack, Pong, and Seaquest. However, in other games, for example Freeway and Video Pinball, we observed very poor performance. While the range of the scores achieved in Video Pinball (ranging from zero to tens of thousands) and the division of the score encoding between multiple variable might explain the poor performance for Video Pinball, it is not obvious why the classification performance of the score in Freeway was so low, given higher performance in other games.

For localisation performance, we observed even greater variation between categories and games. With the exception of Bowling, the model was were generally poor at localising the positions of small objects. For Asteroids, it is unlikely that not learning the positions of the player's missiles would not impact the ability to learn a good policy. However, for games where the agent must learn to avoid (e.g. enemy missiles in Berzerk and Demon Attack) or return (e.g. the ball in Breakout, Pong, Tennis or Video Pinball) small objects, this may make it very difficult or even impossible to learn good strategies[1]. It is interesting that small object localisation performance is so different between Breakout

[1]It is still conceivable that the agent may learn creative policies to alleviate these deficiencies, such as tracking the opponent instead of the ball in Pong.

($R^2 = 0.03$) and Pong ($R^2 = 0.45$), given that for both games this relates solely to the ability to predict the x and y locations of the ball, and in this regard they are visually very similar.

Despite the inconsistencies in performance across the categories and games, there are still sufficiently positive results to motivate the evaluation of this state representation learning approach within our hybrid reinforcement learning method. For example, based on the high quality potential of the representation learned for Bowling across all categories, we expect that this game will be one of the high performing games for AE-NEAT, assuming that the policy required to play the game is not too sophisticated.

5.6 Discussion

There are several interesting results and observations worthy of further discussion.

5.6.1 The Effectiveness of using a Weighted Reconstruction Error

The limitation of using unsupervised learning methods, such as autoencoders, to learn compact representations is that there is no guidance as to what parts of the image are useful for policy learning. This means that the compressors sometimes learn to encode irrelevant information, which may interfere with the learning of relevant information. What constitutes irrelevant information varies between games, which makes the task of developing techniques that reduce the focus on irrelevant information difficult. To maintain generalisability, the techniques used must not require game-specific information. A clear example of irrelevant information is the Activision logo at the bottom of the screen in Boxing (see Fig. 5.8), which is well reconstructed by all candidates.

In an attempt to focus the learning of the encoders on relevant aspects of the image, we tested the use of a weighted reconstruction error measure, WSSE, that penalised the autoencoders for not accurately reconstructing moving objects. However, the design space evaluations showed that the effectiveness of measure varied substantially between candidates. Although the use of WSSE tended to substantially improve the localisation performance for variational autoencoders, it had little to no effect on the representations learned by the undercomplete autoencoders, as assessed by the AtariARI probes. When

it came to learning the values of game state information such as scores, clock values, or the number of lives, WSSE tended to have a detrimental effect on performance across all models.

Further investigation is still required to fully clarify the conditions under which weighted reconstruction error measures may benefit or disadvantage performance. However, our results represent the most in depth analysis on this idea yet and have shown that improvements in the in the reconstructions do not necessarily lead to improvements in the encoding of important state variables.

5.6.2 Trade off Between Representation Size and Encoding Quality

Not unsurprisingly, the relationship between representation size and encoding quality varied quite substantially not only between games, but also between state variables within games. This variation meant that comparisons between the candidates in our design space were not straightforward, and explanations for why we observed differing relationships for different state variables and models are not universal or obvious. An improved evaluation framework similar to the AtariARI, but that provides object positions in pixels, and is not limited to storing solely returning RAM values may allow for a better assessment of state representation learners, and by extension the relationships between representation size and quality.

5.6.3 Evaluating Representation Quality using the AtariARI vs. Reconstructions

When proposing the AtariARI as a method for evaluating state representations, Anand et al. (2019) did not compare assessments using the AtariARI against measuring performance by inspecting reconstruction quality, the de facto standard method for evaluating unsupervised representation learning performance prior to the release of the AtariARI. Our experiments, in the most formal setting yet, investigated the relationship between representation size and quality. They highlighted some interesting differences in conclusions that might have been drawn had the candidates been evaluated using reconstruction quality instead.

A common occurrence in the results was good reconstruction quality not transferring to good performance in the evaluations performed using the AtariARI. One of the best examples of this was the quality of the reconstructions of the agent and opponent in Boxing, compared to the middling AtariARI performance. We observed that while all models were able to accurately reproduce the positions of the agent and opponent, the R^2 value for the regression performance topped out at approximately 0.6. One plausible reason for this is that the information, while clearly present in the compressed representations, was too highly compressed for the nonlinear probe to extract and process to a higher level. This would imply that autoencoders, be them undercomplete, variational, or disentangled variational, are unable to learn representations suitable for better performance.

On the other hand, the AtariARI also sheds greater light on situations where performance appears poor or indifferent between models in reconstructions, but clear differences are present in the AtariARI. For example, the differences in score/clock performance in Boxing appear equally poor between variational and disentangled variational autoencoders, and the better performance of variational autoencoders is only evident when looking at the AtariARI results.

The AtariARI was also able to help identify drops in the representation quality that would have gone otherwise unnoticed if considering reconstructions alone. Another commonplace characteristic of the results was sharp drops in performance only after the representation size was decreased past a breaking point. This is most evident in Pong, where for the evaluation of the models' abilities to encode the scores for undercomplete and variational autoencoders were very consistent, up until the representation size was reduced to 10 dimensions. Looking at the reconstructions alone, the drop in performance is unnoticeable, however, in the AtariARI results, the $F1$ score drops substantially. For the best models, the drop in $F1$ score for the best models as the representation size is decrease from 20 to 10 dimensions is as much as 50%.

Overall, as anticipated, the AtariARI evaluations provide a far more detailed breakdown of performance than reconstructions alone, but there are still unanswered questions that remain about the representation learning capabilities of autoencoders.

5.7 Summary

In this chapter, we defined and evaluated a design space of autoencoder models. Our aim was to identify a suitable model to use as the representation learner in our hybrid reinforcement learning method, AE-NEAT. We required a model that balanced representation size and quality. Through evaluations using the Atari Annotated RAM Interface (AtariARI), we identified an undercomplete autoencoder model with a representation size of 40 dimensions to use in our evaluations of AE-NEAT.

Overall, the results observed in our evaluation of our final selected candidate give us confidence that, in at least some of the games in our evaluation set, our selected autoencoder is able to sufficiently reduce the input space to a high quality compressed encoding that will enable policy learning using NEAT. In the next chapter we perform an independent evaluation of NEAT for policy learning from compact state representations. Then, in Chapter 7, we evaluate AE-NEAT as a whole.

Chapter 6

Policy Learning from Compact State Representations

Our proposed hybrid learning method, AutoEncoder-augmented NeuroEvolution of Augmenting Topologies (AE-NEAT), consists of two major components: a state representation learner, and a policy learner. The last chapter concentrated on identifying a suitable autoencoder model to serve as the state representation learner, followed by an evaluation of the most promising model in isolation from the policy learner. This chapter shifts the focus to the policy learning component of AE-NEAT. The role of the controller (the policy learning component) is highlighted in Fig. 6.1. In this chapter, we perform an independent evaluation of using the NeuroEvolution of Augmenting Topologies (NEAT) algorithm (Stanley and Miikkulainen, 2002) for evolving policy networks for playing Atari games, from hand-crafted, compact state representations.

The remainder of this chapter is divided into four sections. The first section describes the experimental procedure for the AtariARI policy learning experiments, including the setup of the Atari environments, our process of hyperparameter selection, and the metrics for evaluating and benchmarks for comparing agent performance. The second section describes the results of the experiments, followed by a discussion of these results. The third section provides a discussion of the results observed, while the closing section concludes by summarising the key findings of the experiments, that influence decisions made in Chapter 7.

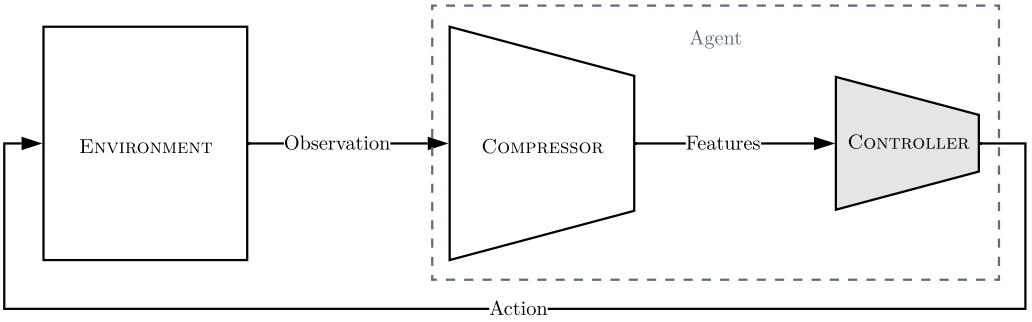


Figure 6.1: A recap of the role of the controller in agents that use a compressor-controller design.

6.1 Experimental Procedure

The experiments performed for this chapter assessed the plausibility of using NEAT as the algorithm for evolving policy networks in our hybrid learning method, before evaluating the method as a whole. This was assessed using the state representations provided by the Atari Annotated RAM Interface (AtariARI) (§4.5). These representations are both compact and of high quality, which enabled us to set expectations on performance for our later experiments. We followed a similar standardised experimental setup to other works that have trained Atari-playing agents using evolutionary (Hausknecht et al., 2014, Salimans et al., 2017, Such et al., 2017) and gradient-based (Mnih et al., 2015, van Hasselt et al., 2016) methods.

6.1.1 Environment Setup

The evolved agents were evaluated using the Atari environments provided by the OpenAI Gym (Brockman et al., 2016). These provide a high-level interface for the Arcade Learning Environment (Bellemare et al., 2013). For each environment, the respective AtariARI wrapper was used to provide a compact state representation at each time step. Further details on these state representations are provided in §4.5. For each game, we normalised the state variables provided by the AtariARI to values within the range [0, 1]. These values served as the inputs to the evolved neural network agents. Normalisation was achieved by dividing each state variable by 255. This is guaranteed to produce a value within the range [0, 1] because each state variable represents a byte of RAM, with a value within the range [0, 255].

To ensure that each episode of gameplay terminated, and that the agents could not succeed by memorising a specific sequence of actions, we used standard adaptations to the Atari environments. To ensure episode termination, we used a frame cap, that is, a limit on the length of each episode. We set this limit to 18,000 frames, which equates to five minutes of gameplay, the same limit used by Mnih et al. (2015). This allows for fair comparison against the expert human scores they reported for each game, collected under the same conditions. This frame cap applies to *game* frames, not *training* frames (a frame that the agent decides to take an action on). The number of training frames is fewer than 18,000 because we used stochastic frame skipping to introduce randomness into the environments. This repeats each action chosen by the agent for the next either two, three, or four frames. This is one of the standard practices for breaking the determinism of games, and is available in the OpenAI gym Atari environments. Other alternatives include using a random number of no-op (no action) starts (Mnih et al., 2015) or human starts (Nair et al., 2015).

6.1.2 Hyperparameter Selection

Given the number of hyperparameters, and the time required to perform evolutionary runs, a grid search or other method for formally searching for optimal hyperparameter values was infeasible. Therefore, our hyperparameters were initialised based on prior studies that have used evolutionary algorithms in the context of Atari-playing agents (Hausknecht et al., 2014, Peng et al., 2018, Salimans et al., 2017, Such et al., 2017). We empirically refined these values through informal experimentation on a subset of three games: Asteroids, Boxing and Pong. Our approach to identifying a promising set of hyperparameters was to look for those that a) showed improved performance over time, and b) controlled speciation. The single set of values chosen from these games was then used for our formal experiments across all games. This subset of games was chosen because they all have good representations, considerably different gameplay mechanics, and cover the spectrum of initial network sizes (due to the size of their input and output spaces). Considering games that require different network sizes is important to ensure that the speciation mechanism of NEAT adequately speciates the populations for each game.

TABLE 6.1: A summary of the experimental parameters used in our experiments.

Parameter	Value
Evaluation	
Max Frames per Episode	18,000
Action Set	Legal Actions
Min Action Set Size	3
Max Action Set Size	18
Min Input Set Size	6
Max Input Set Size	41
NEAT	
Generations	200
Population Size	130
Add Node Probability	0.03
Add Connection Probability	0.05
Mutate Weight Probability	0.8
Activation Function	Sigmoid

The initial population of networks were fully connected with no hidden nodes. The number of input and output nodes varied between games due to the size of the state representation provided by the AtariARI and the set of legal actions the agent is allowed to perform. All other hyperparameters were held constant across the games. All weights and biases were initialised using a uniform distribution with a range of [-3, 3]. Mutations for weights and biases were drawn from a uniform distribution with a range of [-0.05, 0.05]. We chose low mutation rates to ensure that existing functionality was not broken by mutations. A summary of important hyperparameter values are listed in Table 6.1. The full list and assignment of hyperparameters are included in Appendix B.1.

6.1.3 Evaluation Procedure

All games use a common assignment of hyperparameter values. A separate policy was evolved for each game. For each game, three evolutionary runs, each with a different random seed, were performed, each for 200 generations. During each generation, the fitness of each individual in the population was calculated as the mean cumulative reward received over three episodes of gameplay. The cumulative reward is analogous to the agent's score for the episode. An average over a number of episodes is required to obtain an accurate estimate of the fitness of each individual because of the introduced (§6.1.1), and sometimes inherent stochasticity, in the environments. We found three episodes to be a good balance between fitness estimation quality and maintaining feasible training

run times, given the available compute resources. An episode was terminated if the frame cap was reached, and the score for that episode was recorded as the cumulative reward at that point. To speed up the evaluations of the population at each generation, they were parallelised over 130 CPU cores. To select the final policy for each game, the best policy from each run was evaluated for 100 episodes. The policy with the highest average reward is reported in our results.

To assess the performance of the evolved solutions for each game, we compare them against the expert human scores published alongside DQN (Mnih et al., 2015). These scores are the mean score achieved over 20 episodes by a professional human games tester, after approximately two hours of practice playing each game. The only exception to this is for Berzerk, which was not included in their set of experiments. Instead, for Berzerk, we compare the agent's performance against the human score reported by van Hasselt et al. (2016). While this score was achieved under slightly different conditions (achieved from episodes starting with human starts), it provides a closer comparison than using world record scores. The reason we do not use records for the games is because they are not indicative of average expert human performance. We also compare the performance of our agents against the published results of NEAT using Hausknecht et al.'s (2014) object class representation. This provides a point of comparison against using NEAT to evolve agents using another high-level state representation. These scores were obtained using a slightly different evaluation procedure, only an average over five episodes of gameplay and using a frame cap of 50,000 frames. We also compare the performance of the human scores, and the NEAT AtariARI and NEAT Object Class agent scores against the scores obtained by a random agent. The scores for the random agents were collected using the same environment setup as for the NEAT AtariARI agents (§6.1.1). The random agents make decisions by uniformly sampling each action from the set of legal actions for each game.

6.1.4 Human-Normalised Scoring

For Boxing, Pong, and Tennis, the agent plays against a computer-controlled opponent. In these games, the total reward is defined as the player's score minus the opponent's score. Therefore, the agent can achieve a negative total reward. For all other games, the agent's score begins as zero, and points are accumulated during gameplay.

Because each game has a different scoring mechanism, it is difficult to compare agent performance between games directly. Therefore, to compare performance between games and against human performance, we report agent performance using *human-normalised* scores. These scores measure the percentage of the human score that is achieved by the agent and are calculated by

$$normalised\ score = \frac{agent\ score - min\ score}{human\ score - min\ score} \tag{6.1}$$

where *agent score* refers to the mean cumulative reward of the best performing solution over 100 episodes of gameplay, *min score* is the minimum total reward possible for the game, and *human score* is the expert human score.

6.2 Results

This section presents the results of the AtariARI policy learning experiments. First, we examine the performance of the evolved agents in comparison to expert human performance, agents evolved using a different state representation, and the performance of random agents. Following this, we investigate other characteristics of the results, including the network architectures and policies of the evolved agents.

6.2.1 Overall Performance

Fig. 6.2 shows the human-normalised performance of each of the best agents; as can be seen, performance varies substantially between games. The bar colour denotes the quality of the state representation provided by the AtariARI, while the dashed line indicates the threshold of expert human performance. Setting aside the performance of the Tennis A agent, which turns out to have exploited a loophole in the evaluation process, only the agents for Boxing and Bowling exceed expert human performance. For other games, the agents perform far worse. The Freeway and Frostbite agents come close to expert human performance, achieving 92% and 89% of the average expert human score on average respectively. Although the best performing games (Boxing, Bowling and Freeway) have *good* state representations, the high performing Frostbite agent illustrates

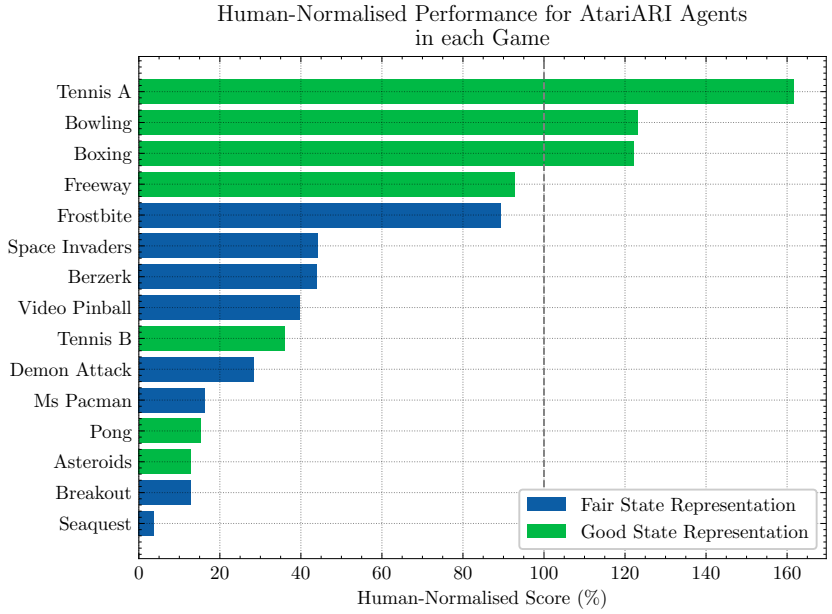


FIGURE 6.2: The human-normalised performance of the best agent for each game.

that for some games, good strategies can still be discovered with imperfect and missing information. Although the Tennis A agent also exceeds expert human performance, this is achieved by exploiting a loophole in the evaluation process. The true performance on Tennis, after addressing the loophole mentioned above, is recorded as Tennis B. The difference between Tennis A and Tennis B is explained in §6.2.3.

Table 6.2 lists the average scores of the best agents for each game and compares these against the average scores achieved using Hausknecht et al.'s (2014) hand-crafted object class representation, the performance of a random agent, and the expert human scores reported by Mnih et al. (2015). The standard deviations for the AtariARI and random agents are also reported. The variability in scores for the other agents were not reported in the respective papers. This table highlights a few unexpected outliers in terms of performance; in particular, our evolved agents for Pong and Breakout perform considerably worse than NEAT using the object class representation and human performance. In addition, the random agent exceeds both the AtariARI and expert human performance for Video Pinball. These are further investigated in the following sections.

The result for Video Pinball is an outlier, in that it was the only game for which the best evolved agent did not outperform a random agent. However, we found that the random

[1]Score from van Hasselt et al. (2016).

TABLE 6.2: Scores for NEAT using the AtariARI inputs, compared against NEAT using a hand-crafted object representation, a random agent and expert human performance.

Game	NEAT AtariARI	NEAT Obj. Class (Hausknecht et al., 2014)	Random	Human (Mnih et al., 2015)
Good Representations				
Asteroids	1693.9 (613.0)	4144.0	1013.7 (439.0)	**13157.0**
Bowling	190.8 (23.4)	**231.6**	24.3 (5.3)	154.8
Boxing	27.4 (17.0)	**92.8**	0.2 (4.4)	4.3
Freeway	27.5 (1.9)	**30.8**	0.0 (0.0)	29.6
Pong	-16.4 (2.5)	15.2	-20.3 (0.9)	**18.9**
Tennis A	0.4 (0.9)	**1.2**	-23.9 (0.3)	-8.9
Tennis B	-18.6 (1.1)	**1.2**	-23.9 (0.3)	-8.9
Fair Representations				
Berzerk	984.8 (175.3)	1202.0	162.0 (118.1)	**2237.5**[1]
Breakout	4.0 (5.4)	**43.6**	1.5 (1.3)	31.8
Demon Attack	964.7 (388.6)	**3464.0**	185.6 (139.2)	3401.0
Frostbite	3873.2 (1128.8)	1452.0	76.0 (40.7)	**4335.0**
Ms Pacman	2559.2 (872.2)	4902.0	220.3 (169.5)	**15693.0**
Seaquest	750.0 (38.1)	944.0	88.8 (64.3)	**20182.0**
Space Invaders	729.3 (91.9)	1481.0	157.4 (101.0)	**1652.0**
Video Pinball	6859.0 (5401.2)	**253986.0**	22768.1 (14386.9)	17298.0

agent also outperformed the expert human performance (on average). This result is discussed further in §6.3.4.

Despite the more compact state representations provided by the AtariARI, our agents typically perform worse than the NEAT object class agents; outperforming them in only one of the 14 games (Frostbite).

6.2.2 Fitness over Time

Figure 6.3 shows the fitness curves for the best run for each respective game. For each generation, the fitness of the best genome is shown in red, the population's mean fitness in blue, and one standard deviation around the mean is shaded. The plots highlight some key differences in the learning patterns between games. For some games, including Asteroids, Bowling, Boxing, Ms Pacman, Pong, and Space Invaders, we see promising evidence of progress being made over time. Another interesting observation is that for many of the games with only *fair* state representations (Berzerk, Breakout, Demon Attack, Seaquest, and Video Pinball) there is little to no improvements in the best solution found over time. This may indicate that the missing information for these games is indeed important for learning.

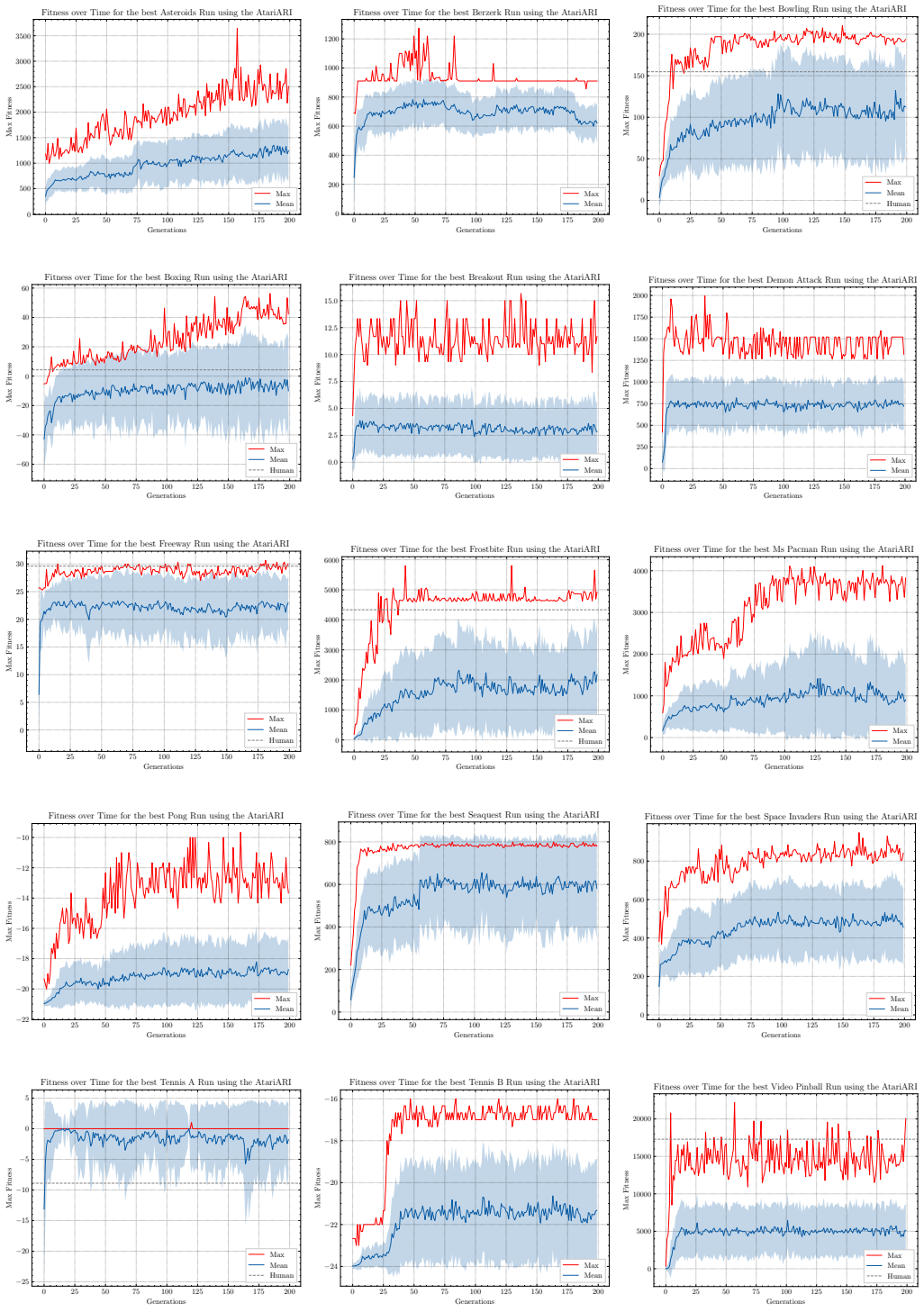


FIGURE 6.3: The fitness curves for the AtariARI agents in the best run for each game.

6.2.3 Loopholes and Local Optima

In several of the games, we observed the agents succumbing to interesting locally optimal behaviour. Two extreme cases of this are for Pong, due to a loophole in the game, and Tennis, due to the way we calculated fitness.

When playing Pong, the initial trajectory of the ball for each episode is randomly selected from a small set. However, for one of the trajectories, the agent can achieve a perfect score (21-0) in approximately 22% of episodes[2] by merely moving to one particular position. In this position, when the opponent returns the ball, it always rebounds back to the agent. This behaviour was discovered early on in one of the three Pong runs (in the 18th generation), and was never surpassed. Interestingly, this run performed much poorer than the other two runs after finding this local optima, which may indicate that after this misdirection, the population was unable to recover and discover a better strategy.

To play Tennis, the agent and the opponent take it in turns to serve each game within the episode. When it is the agents turn to serve, the agent must execute the FIRE action to begin. However, agents that learn to do this typically perform very poorly initially, as they cannot track and hit the ball each time it is returned. Therefore, these agents lose the majority of games and therefore, all sets. As a result, the agent accumulates total reward close to the minimum (-24). Because of the frame cap on the length of episodes (put in place to prevent endless play in games without a definitive end), the population of agents quickly converges on the strategy of taking no actions to stop the Tennis match from progressing and receive a total reward of zero.

To address the issue of the agent refusing to play, we changed the reward given to Tennis agents if they reach the frame cap. Since the frame cap should never be reached when playing Tennis, if the agent is attempting to play the game, we set the reward that the agent receives to the minimum possible reward (-24). This prevents the agent from exploiting the loophole and leads to objectively worse, but not misleading results. We report the results for the original and modified settings as Tennis A and B, respectively.

6.2.4 The Effect of Representation Quality and Network Size

When considering performance and representation quality, all of the agents that match or exceed human performance are provided *good* state representations. However, none of the other agents for games with *good* representations eclipse human performance. This suggests, unsurprisingly, that more than just representation quality influences performance.

[2] averaged over 30 samples of 100 episodes

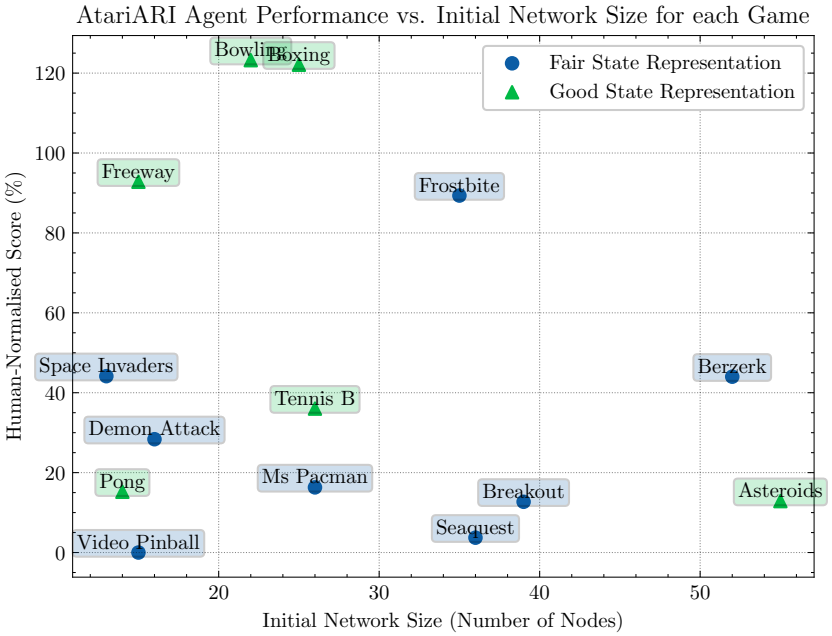


FIGURE 6.4: The relationship between performance, representation quality and initial network (input + action space) size.

One potential factor that might also influence the ability to evolve solutions is the combined size of the state representation and action space for each game. This can be considered a pseudo-measure for game difficulty, something that is hard to define precisely. Games with both large state representations and large action spaces require the agent to process more inputs and choose from many possible actions, making the task of choosing a good action more difficult.

Fig. 6.4 helps to illustrate and investigate the relationship between performance, representation quality and network size. From this, we can see that the highest performing agents were all in games that had relatively small combined input and output space sizes. Given this trend, there are several outliers, such as Pong and Tennis, with surprisingly low performance, when considering the small size of the input and output spaces, representation quality and the performance achieved with other methods.

6.2.5 Evolved Architectures

Inspecting the architectures of high-performing agents reveals some surprising simplicity. None of the solutions to any games evolved many hidden nodes (the maximum number of hidden nodes was 27 for Ms Pacman). Even the networks for high-performing solutions were very simple, with the best Boxing and Bowling agents having 18 and 13

nodes respectively. This may have been a consequence of the particular hyperparameter values chosen, but it shows that simplicity is not the sole explanation of poor performance. Architecture diagrams for each one of the best performing agents are included in Appendix B.2.

Though it is difficult to interpret neural networks, there are some structures we know to be useful that we can inspect the networks for, such as the presence of longer path lengths and recurrent connections for tracking moving objects. As mentioned earlier, a consequence of our time-delayed recurrent neural network implementation is that not only self-loops and backward connections create memory, but also different path lengths. Knowing that memory is required in order for the agents to track the movement of objects, we can inspect solutions for games in which tracking is important as an explanation for their poor performance. Despite requiring memory to learn good strategies, the solutions for Tennis or Breakout do not appear to have developed the structural innovations required to track the ball. In Breakout, the evolved architectures do not include a delayed pathway or recurrent connection to enable the agent to account for the y direction of the motion of the ball, and the agent for Tennis does not evolve connections or additional nodes that allow it to account for either the x or y direction of the ball. For the best Pong agent, there are delayed pathways between the x and y positions of the ball and the outputs. However, the agent does not appear to be utilising such structures to track the ball when it's behaviour is examined. The architecture diagram for the best Pong agent is shown in Fig. 6.5. The width of each connection is scaled by the magnitude of its weight.

6.3 Discussion

The variation in performance, behaviour, and architecture of evolved policies for different games leads to several interesting points worthy of discussion. In this section, we elaborate on each of these points before discussing avenues for further research.

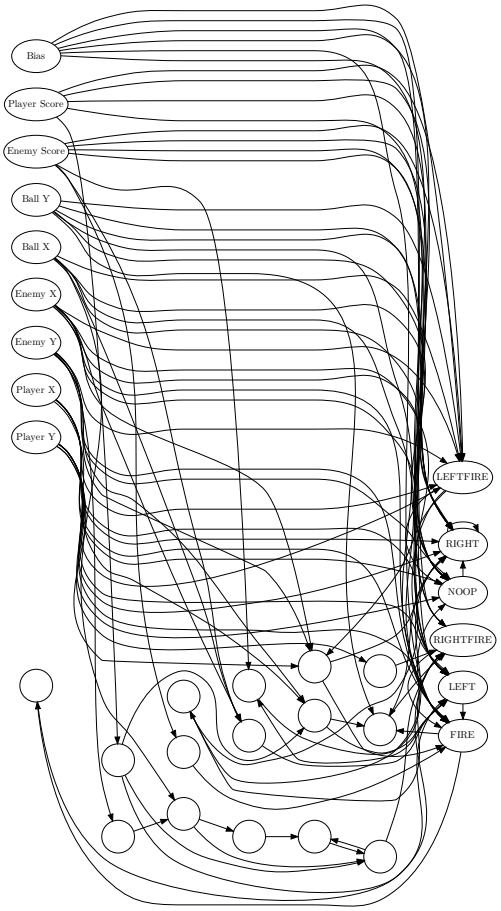


FIGURE 6.5: The evolved network architecture for the top performing Pong agent.

6.3.1 Performance Relative to the Hand-Crafted Object Representation

The AtariARI agents perform poorly compared to the agents trained by Hausknecht et al. (2014) using their hand-crafted object representation. With the exception of the Frostbite agent, the other agents all perform worse than their reported scores. There are a number of possible reasons for this.

One possible reason is that spatial information may be more explicit in object class representation than in the AtariARI representation. Though the object class representation is larger, each input value represents the presence of a particular object in a particular location. In comparison, the information in the RAM values is more densely encoded. In some cases, there is also an unknown mapping between the RAM values and the object locations on screen. Since the value of each byte is limited to the range [0, 255], the values in RAM may not directly match the values of higher level information. A

prime example of this is for games with large scores, and games with agents that move vertically. For games with large scores (e.g. Space Invaders, Video Pinball) the player's score is stored using several bytes (most likely each representing least significant to most significant bits of the larger number), therefore, in order for this score to be useful, it is likely that the evolved architecture must learn to combine these values into something meaningful.

For games that store locations, there may also be mappings between the RAM values and the locations of objects. The original authors of the AtariARI have since reported that not only are the mappings between RAM values and pixel positions inconsistent between games, they are also sometimes inconsistent within games. For example, the RAM values of some games increase in increments of 16 as the sprites move.

There is also the possibility that differences in performance are caused by differences in the implementation of NEAT, or in the chosen hyperparameters. While our hyperparameters match those for the ones they reported, some important hyperparameters, such as those for the strength of weight mutations are not reported.

One final observation is that the frame cap was higher for their evaluations (50,000 vs. 18,000 frames), and the number of episodes of gameplay the agents scores were averaged over is significantly less (5 vs. 100). While our frame cap was chosen to enable us to make primary comparisons between our agents and expert human scores. For quick-to-complete games, such as Pong, the extra frame cap is unlikely to make much of a difference, but for endless gameplay games, such as Space Invaders and Video Pinball, the longer episode time could contribute to the higher scores observed. Finally, given the variability in scores achieved by our agents, the low number of evaluations per game may misrepresent the true performance of their agents. For many of our agents, the fitness plots show far higher scores during evolution (e.g. Video Pinball) which are vastly reduced when the agent's score is averaged over a large number of episodes. Given the variation observed in even our high performing agents, it could be that the scores reported by Hausknecht et al. (2014) might be lower if averaged over more episodes. Since each of our agents is evaluated over 3 episodes each generation, our max fitness scores are more comparable to their reported scores.

6.3.2 Dealing with Partial Observability

Without incorporating information from multiple frames, many Atari games are partially observable Markov decision processes. Good examples of such games are Pong, Tennis and Breakout, as they require the agent to evolve memory to track the direction of movement and speed of a ball. As reported earlier, none of the agents for these games evolved the required structures to perform this tracking, which partially explains their poor performance. One potential reason for this is that the structures required for tracking the ball require many successful mutations and as such lie some distance from the initial conditions. If this is the case, devising speciation methods that better protect intermediate mutations that are initially detrimental to performance but are needed for more complex structures may be the key to unlocking better performance in games that require complex or long term memory. A difficulty in addressing this though is that increasing the strength of structural mutations may prevent the agents from learning entirely. This balance between mutation strength and progress is one of the key challenges for neuroevolution methods, and one of the reasons why they do not scale to evolving larger networks. An alternative method for increasing the likelihood of the required mutations occurring is to increase the population size. This allows for greater exploration while not negatively impacting performance.

6.3.3 Surprising Simplicity of Solutions

Perhaps the most interesting of our results is that for some games there exist surprisingly simple solutions. Often, human-designed neural networks are highly over-parameterised, and our results begin to lift the veil on the actual complexity required to encode effective policies for some games. The size of our evolved solutions, particularly those that are high-performing, stand in stark contrast to the sizes of networks trained using other techniques. For example, even after the convolutional feature extraction layers, the commonly used DQN architecture has a layer of 512 hidden nodes before the output layer. Our results show that given a sufficient state representation, learned or otherwise; some games only require comparatively tiny policy networks. These findings suggest that the possibility of finding smaller overall networks by separating state representation and policy learning may indeed be possible and is worthy of further investigation. We suspect the primary reasons for the discovery of these small, yet successful, architectures appears

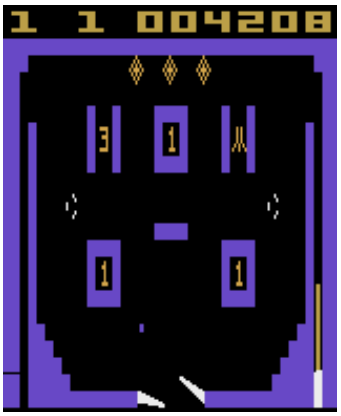


FIGURE 6.6: A gameplay image from Video Pinball.

to be that the solutions exist near the initial conditions and the games are simple to play, and that structural mutations must provide benefit to the agent to ensure that they are retained and slowly propagated throughout the population.

6.3.4 High Random Agent Performance in Video Pinball

For Video Pinball, the experiments found that on average the random agent outperformed both the AtariARI agent and the human player. To understand why randomly choosing each action is such a successful strategy (at least on average), it is important to understand how the game is played.

Video Pinball is designed to simulate an arcade pinball machine. The aim of the game is to accumulate as many points as possible. These points awarded in varying amounts for hitting different targets with the ball. The player is only required to hit the ball, by firing the left or right bumper, when it drops to the bottom of the screen. Failing to do so, causes the ball to fall out of play. Simply keeping the ball in play is sufficient for gaining points slowly, but actively aiming for high value targets is the ideal strategy. A screenshot of Video Pinball is shown in Figure 6.6.

One reason why the random agent performs so well is because there are a small number of actions to choose from, and, there are very few critical time steps (time steps when the agents choice of action is important). In fact, at the critical time steps, there is an approximately 44% chance that the agent will randomly choose the correct action[3]. While this strategy works well on average, as was shown in the results, it leads to very

[3]There is a total of nine legal actions the agent can choose from, but eight of these actions result in firing either the left or right bumper (four for each), and only one results in the agent taking no action.

high variability in episode scores. Though the variance in the scores achieved by the human player was not reported by Mnih et al. (2015), it is likely that although the average score is slightly lower, they are more consistent than the random agent.

The variability in episode scores also likely contributes to the failure of NEAT to find a good solution. This is because, particularly early on, the high variance makes it difficult to assess the true fitness of each individual. As a result, individuals that have developed better strategies on average, are likely assigned lower fitnesses than poorer agents because the fitness is decided only based on an average over three episodes. Although an average over three episodes is a good compromise between evaluation accuracy and compute time for most games, for Video Pinball, it would appear that it is a poor one. However, as is shown in Table 6.2, even when performing 100 evaluations the variance is still very high. This would make it very difficult for evolution to proceed, but, if it could pick a better direction early on, then you might expect the variance to decrease later as the agent actually learns a better strategy.

6.4 Summary

High-quality, compact state representations can make it easier to find solutions to complex reinforcement learning problems. They also open up the possibility of using neuroevolution to evolve elegant solutions. This chapter assessed the plausibility of using NEAT to evolve agents for a subset of Atari games, using the inputs provided by the AtariARI. The purpose of these experiments was to independently evaluate the ability to evolve solutions from hand-crafted state representations. In the following chapter, we combine the findings from this and the previous chapter and evaluate our combined state representation and policy learning method, which was described in Chapter 4.

Although the evolved policies only exceeded or were competitive with expert human performance in a handful of games – Boxing, Bowling, Freeway, and Frostbite – we were able to show that surprisingly simple and small neural networks could play these games effectively. Furthermore, we were able to identify potential reasons for the poor performance exhibited in some games. First, many of the games for which poor solutions were found were categorised as having only *fair* quality state representations provided by the AtariARI. This indicates that the missing information may be more important than

first thought for finding good solutions. Second, the encoding of the spatial information provided by the AtariARI may make the information difficult to process, adding to the challenge of evolving solutions for the games.

Overall, the results show that, when not required to learn compact state representations as well as policies, neuroevolution methods that optimise both weights and topology can find elegant solutions to complex reinforcement learning problems. This provides promising evidence that our hybrid learning method may be successful. In the following chapter, we combine the findings from this and the previous chapter and evaluate our combined state representation and policy learning method that was described in Chapter 4.

Chapter 7

Simultaneous State Representation and Policy Learning

The previous two chapters evaluated the state representation (compressor) and policy learning (controller) components of our hybrid learning method, AE-NEAT, in isolation. Having established that both of these components are able to function independently, in this chapter we evaluate our hybrid method as a whole. We begin by explaining the experimental setup for our AE-NEAT evaluations. Following this, we present the results of these evaluations. Finally, we discuss the results.

7.1 Experiments

The experiments presented in this chapter follow a similar design to the policy learning experiments that were conducted using the AtariARI, detailed in §6.1. As we describe the setup for the experiments in this chapter, we highlight any differences between the two experimental designs.

7.1.1 Environment Setup

In keeping with our other experiments, we again used the OpenAI Gym (Brockman et al., 2016) Atari environments to evaluate the agents. As with the environment setup for our independent policy learning experiments (§6.1.1), we used stochastic frame skipping to introduce randomness into the environments and a frame cap of 18,000 frames (equivalent to five minutes of game time). The only difference between the environments used for our independent and hybrid experiments is the observations that are provided to the agents. Instead of using the the state representations provided by the AtariARI, the agents are fed full-size (210×160 px), single-channel, greyscale images that the compressor must learn to compress.

7.1.2 Hyperparameter Selection

AE-NEAT has two sets of hyperparameters: those that relate to the evolution of the policy network using NEAT, and those that relate to the training of the autoencoder. The addition of the compression stage during fitness evaluations and the training of the autoencoder between generation substantially increases the time to perform each run. As a result, the difficultly of optimising hyperparameters remained and prohibited our ability to perform a more comprehensive search for good hyperparameters. Table 7.1 includes a summary of the hyperparameter values used in our experiments. The full list of hyperparameters and values are listed in Table C.1 in Appendix C.

7.1.2.1 NEAT Hyperparameters

Given the success of the experiments in Chapter 6, we used the same NEAT hyperparameters as in our AtariARI policy learning experiments (§6.1.2), with the exception of a large population size of 300 instead of 130. This difference was possible because of increased availability of compute resources at the time of performing our hybrid learning experiments. A larger population is beneficial because it allows for greater exploration of the policy space. To ensure that the chosen hyperparameter values remained viable despite the changes to the state representations fed to the policy networks, we checked them on the on the same subset of three games (Asteroids, Boxing, and Pong) that were used for hyperparameter optimisation in Chapter 6. The main purpose of this

TABLE 7.1: A Summary of the important hyperparameter values used for our AE-NEAT evaluations.

Parameter	Value
Evaluation	
Max Frames per Episode	18,000
Action Set	Legal Actions
Min Action Set Size	3
Max Action Set Size	18
Image Store Size	100,000
Image Sampling Method	Random
NEAT	
Generations	200
Population Size	300
Add Node Probability	0.03
Add Connection Probability	0.05
Mutate Weights Probability	0.8
Activation Function	Sigmoid
Autoencoder	
Latent Space Dimensions	40
Type	Undercomplete
Architecture	DQN-inspired
Loss Function	Sum of Squared Errors
Epochs per Generation	1
Images per Generation	20,000

was to check that these parameters still adequately controlled speciation with the larger networks and showed evidence of some performance improvement over time. Both observations provide an indication that the mutation settings are of appropriate strength.

7.1.2.2 Compressor Hyperparameters

Additional hyperparameters are required for the training of the compressor: the number of training epochs per generation, training sample size, and observation store size. The observation store is used to collect the gameplay images that are encountered by the different policy networks so that they can be used to train the compressor. For other hyperparameters, such as the learning rate and optimiser, we used the same configuration as for the state representation learning experiments. These are listed in §5.4.1. Between each generation, we refined the encoder by training for one epoch, on a random subset of 20,000 images from the observation store. We found this to be a good compromise between minimising the time between generations and ensuring that the encoder is

refined quickly enough at the beginning of each run. We set the image store size as 100,000. This is a compromise between storing all images generated by the agents and what can be stored in memory for quick access.

One of the biggest issued faced with image collection is the introduction of latency in the evaluation of individuals, caused by the need to save observations from the environment that result from the agents actions. Given a population size of 300, three episodes of gameplay for each evaluation, and a frame cap of 18,000 frames, the maximum number of images that can be generated is over 16 million. Storing all of the images generated in a centralised store for the autoencoder to use for training between generations is infeasible, due to the time to transfer each image over the network and save it to disk. Instead, the size of the observation store was set to a fixed limit (in our experiments 100,000 images). Each worker stores only a random sample of $N \times \frac{1}{P} \times \frac{1}{E}$ images for each episode of gameplay, where N is the size of the observation store, P is the population size, and E is the number of episodes that each individual in the population is evaluated for every generation. This sample is then sent to the centralised observation store, a fast in-memory Redis database[1], which stores the most recent N images. Random sampling is important for ensuring that observations from throughout the episodes are available for training the autoencoder, rather than those only for early or late game states.

7.1.3 Evaluation Procedure

Our evaluation procedure remains the same as for our AtariARI policy learning experiments, however, in addition to expert human scores, we perform additional comparisons against the best performing alternative evolutionary methods that have been used to evolve Atari agents from low-level pixel inputs.

We compare the performance of our hybrid method against OpenAI ES (Salimans et al., 2017), Deep GA (Such et al., 2017), and HyperNEAT (Hausknecht et al., 2014). We also compare our results against Agent57 (Badia et al., 2020), the highest performing general Atari game playing method of any type to date, and the first method to produce agents that exceed expert human performance in all 57 Atari games supported by the Arcade Learning Environment (Bellemare et al., 2013).

[1] redis.io

As emphasised earlier, the main contribution of this work is to show that AE-NEAT can successfully evolve game playing agents. However, we include comparisons against other reported results for evolutionary approaches to reinforcement learning to place our results in the context of alternative methods. In our comparisons against human performance and alternative methods, we compare the performance of the agents trained using AE-NEAT against the reported scores from the respective papers. There are several factors that prohibit more in-depth comparison, including statistical testing, against these benchmarks.

First, is the time taken to train both our agents and the other methods, given our limited compute resources. For our experiments, we performed three evolutionary runs per game, for a total of 200 generations. Depending on the game (which influences the episode length time), these runs typically took between eight to ten hours each to complete, distributed over our make-shift compute cluster using the undergraduate lab machines. This length of time limited the number of runs that could be performed for each game. For the other methods, our comparisons are limited due to our inability to train agents for ourselves due to time requirements for training these methods, the lack of implementations, or differences in the input spaces, discussed in the following paragraph. For instance, the results for Deep GA (Such et al., 2017) and OpenAI ES (Salimans et al., 2017) were obtained using clusters of 720 cores, and 1,440 cores respectively[2].

Second, one should note that the reported results for each of Deep GA, OpenAI ES, and HyperNEAT are collected using different input spaces. OpenAI ES and Deep GA use frame-stacked, greyscale, downsampled 84×84 pixel images that is commonly used for gradient-based approaches, and HyperNEAT uses downsampled 16×21 pixel images and the reduced SECAM (eight colour palette). The experiments performed by OpenAI ES and Deep GA used frame-stacked images, because they were evolving the weights of feed-forward networks that required frames from previous time steps to alleviate the partially observable Markov decision process (POMDP) nature of many games. Despite the use of four frames stacked together, the input space used for these methods was still smaller than ours (28,224 dimensions vs. 33,600 dimensions). The input space for HyperNEAT was substantially smaller than both, at 336 dimensions. Our work uses full-size images from the environments (160×210 pixels) to limit pre-processing performed

[2]Deep GA was also evaluated using "modern" desktop, with 48 CPU cores and 4 GPUs.

for the agents. We also use the AtariARI for policy learning and state representation learning methods. The AtariARI only supports full-size images.

A lack of suitable implementations[3] of the methods used in our comparisons and limited compute resources (a common issue in reinforcement learning research) compounded to make it infeasible for a fair statistical comparison between the methods. However, once again, our main contribution is to show that the method presented is capable of training general video game playing agents, not that our method performs significantly better than these with respect to overall performance, training time, or some other performance measure.

7.2 Results

This section presents the results of the AE-NEAT hybrid learning experiments. First, we examine the overall performance of the trained agents. We compare their performance on each game against human, random agent, and state-of-the-art baselines. We also compare the performance against other evolutionary reinforcement learning methods that also learn from raw pixels. Following this, we investigate reasons for the differences in performance between games.

7.2.1 Agent Performance

Fig. 7.1 compares the human-normalised performance of the agents for each game. 0% represents the minimum possible score obtainable the game, while 100% represents average expert human performance (as reported by Mnih et al. (2015)). As shown, the best performing agent, relative to human performance is discovered for Bowling, followed by Boxing and Video Pinball. The agents for Bowling, Boxing, and Video Pinball all exceed expert human performance, achieving 131.4%, 116.4%, and 109.4% of the expert human scores on average respectively. Additionally, the best agent for Freeway achieves 87.1% of the expert human score on average. The results for these games, and the remaining games, are similar to the performance observed evolving agents using the AtariARI inputs (§6.2.1). The biggest difference between the two types of agents was

[3]Implementations that were (a) available, and (b) able to be modified (within our time constraints) to make them deployable on our custom compute cluster.

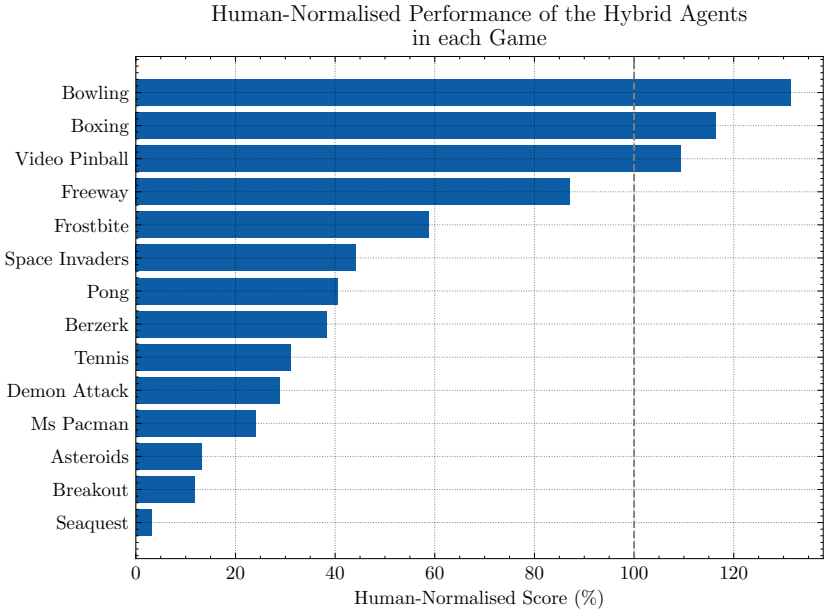


FIGURE 7.1: The human-normalised performance of the best hybrid learning agent for each game.

for Video Pinball. Whereas previously, the best performing Video Pinball agent using the AtariARI achieved on average only 39.7% of the average expert human score, the best AE-NEAT agent surpassed average expert human performance, achieving on average 109.4% of the average expert human score.

Table 7.2 compares the performance of the AE-NEAT agents against other neuroevolution methods that also learn from raw pixels. The left-hand side of the table lists the average scores achieved by random agents, an expert human player, and Agent57 agents (the current state-of-the-art general Atari game playing method) to provide context around the scores of the neuroevolution methods. The right-hand side of the table lists the average scores achieved by our AE-NEAT agents, and for comparison, the average scores reported for competing neuroevolution methods where available. For the random and AE-NEAT agents we also report one standard deviation in the scores to indicate the variability. The results for Agent57 are the only others to do this. As is shown, the best AE-NEAT agent outperforms the other neuroevolution agents in three of the 14 games: Asteroids, Bowling, and Ms Pacman. OpenAI ES posts the best scores among evolutionary methods in seven games, Deep GA in one game, and HyperNEAT in three games. An interesting observation is all of the ERL methods *and* the expert human performed worse than or similarly to the random agent in Video Pinball. Possible explanations for this were discussed in §6.3.4. While AE-NEAT proves to be a competitive

TABLE 7.2: Scores for our hybrid learning method (AE-NEAT), compared against other evolutionary methods that learn from raw pixel inputs, a random agent, expert human performance, and Agent57 (the current state-of-the-art general Atari game playing agent).

Game	Random	Human [1]	Agent57 [2]	AE-NEAT	OpenAI ES [3]	Deep GA [4]	HyperNEAT [5]
Asteroids	1013.7 (439.0)	13157.0	150854.6 (16116.7)	**1739.0** (596.1)	1562.0	1661.0	1694.0
Berzerk	162.0 (118.1)	2237.5	61507.8 (26539.5)	855.7 (233.5)	686.0		**1394.0**
Bowling	24.3 (5.3)	154.8	251.2 (13.22)	**203.4** (17.7)	30.0		135.8
Boxing	0.2 (4.4)	4.3	100.0 (0.0)	21.4 (12.2)	**49.8**		16.4
Breakout	1.5 (1.3)	31.8	790.4 (60.0)	3.8 (4.7)	**9.5**		2.8
Demon Attack	185.6 (139.2)	3401.0	143161.44 (220.3)	981.4 (541.5)	1166.5		**3590.0**
Freeway	0.0 (0.0)	29.6	32.59 (0.7)	25.8 (1.7)	**31.0**		29.0
Frostbite	76.0 (40.7)	4335.0	541280.9 (17485.8)	2546.5 (514.8)	370.0	**4536.0**	2260.0
Ms Pacman	220.3 (169.5)	15693.0	63994.4 (6652.2)	**3761.1** (1580.3)			3408.0
Pong	-20.3 (0.9)	18.9	20.7 (0.5)	-8.7 (13.9)	**21.0**		-17.4
Seaquest	88.8 (64.3)	20182.0	999997.6 (1.4)	630.2 (126.8)	**1390.0**	798.0	716.0
Space Invaders	157.4 (101.0)	1652.0	48680.9 (5894.0)	729.9 (189.3)	678.5		**1251.0**
Tennis	-23.9 (0.3)	-8.9	23.8 (0.1)	-19.3 (3.4)	**4.5**		0.0
Video Pinball	22768.1 (14386.9)	17298.0	992340.7 (12867.9)	18927.6 (13978.2)	**22834.8**		0.0

[1] Mnih et al. (2015) [2] Badia et al. (2020) [3] Salimans et al. (2017) [4] Such et al. (2017) [5] Hausknecht et al. (2014)

alternative to other evolutionary methods, all tend to fall some way behind the scores reported for Agent57, the current state-of-the-art, indicating that much progress is still to be made for evolutionary methods to be competitive with the best policy gradient methods.

7.2.2 Fitness over Time

The fitness curves for each game in Fig. 7.2 show the progress of the policy network populations over time for the run that produces the top performing agent. Each plot displays the maximum fitness, average population fitness, and one standard deviation around the average population fitness. Where appropriate, the threshold of expert human performance is also displayed. Examining the fitness curves for the Bowling, Boxing, and Video Pinball agents that surpass expert human performance, we see that they all do so quite quickly. Agents that exceed human performance over an average of three episodes are first in Generation 47 for Bowling, Generation 2 for Boxing, and Generation 15 for Video Pinball. While the agents for Freeway never surpass expert human performance, we see that even in the initial population a solution with a fitness of 23.6 points is found, which equates to 86.5% of the expert human score. A particularly interesting result is that the top two performing Pong agents, found in generations 53 and 57, both exceed expert human performance over three episodes. The sudden and extreme jump in maximum fitness around Generation 40, and the consistent maximum fitness of zero points (a draw) shortly after, indicate that these scores are outliers and

the population became stuck in a local optimum. The evolved policies that produce such wildly fluctuating and then consistent scores are explored in §7.2.4.

Across all the games, the trends in performance for the runs can be separated into four categories: little/no improvement over time, improvement followed by stagnation, an uptick in fitness towards the generation limit, and consistent improvement. For most of the games, we observe that performance tends to improve over time before stagnating. This suggests that the populations reach local optima that are difficult to escape. However, for Frostbite, Ms Pacman, and Space Invaders, the fitness begins trending upwards again towards the end of the runs. This indicates that better solutions might be found if the runs were allowed to continue for longer. Similarly, the runs for Boxing and Video Pinball show consistent improvement over time, again indicating that better solutions might be found had the generation limit been higher. The runs for Breakout and Demon Attack, whose agents perform among the worst relative to human performance, there appears to be little to no improvement over time. This indicates that it is difficult to escape the local optimum surrounding the initial conditions. Both the performance of the autoencoders and the choice of hyperparameters may contribute to these trends. The remainder of this section focuses on investigating possible explanations for the differences in performance between games.

7.2.3 Compressor Performance

We begin our investigations into the reasons for the differences in performance between games by examining the quality of the compressors trained during the best training runs for each game. Although we report the results for only the run that produces the best agents for each game, the quality of the final compressors is similar across all three runs for each game. Table 7.3 shows the localisation and classification results of the final compressors for the best run for each game. There are some notable differences compared to the compressors of the same design trained independently using the observations collected by a trained PPO agent (§5.5.2).

In the majority of cases (39 of the 53) the scores for the performance of the compressors from the final agents across each state variable category were comparable, within 0.05 of the reported average R^2 and $F1$ values, to the performance reported in the independent evaluations (§5.5.2). In four cases, the compressors learnt better than the independently

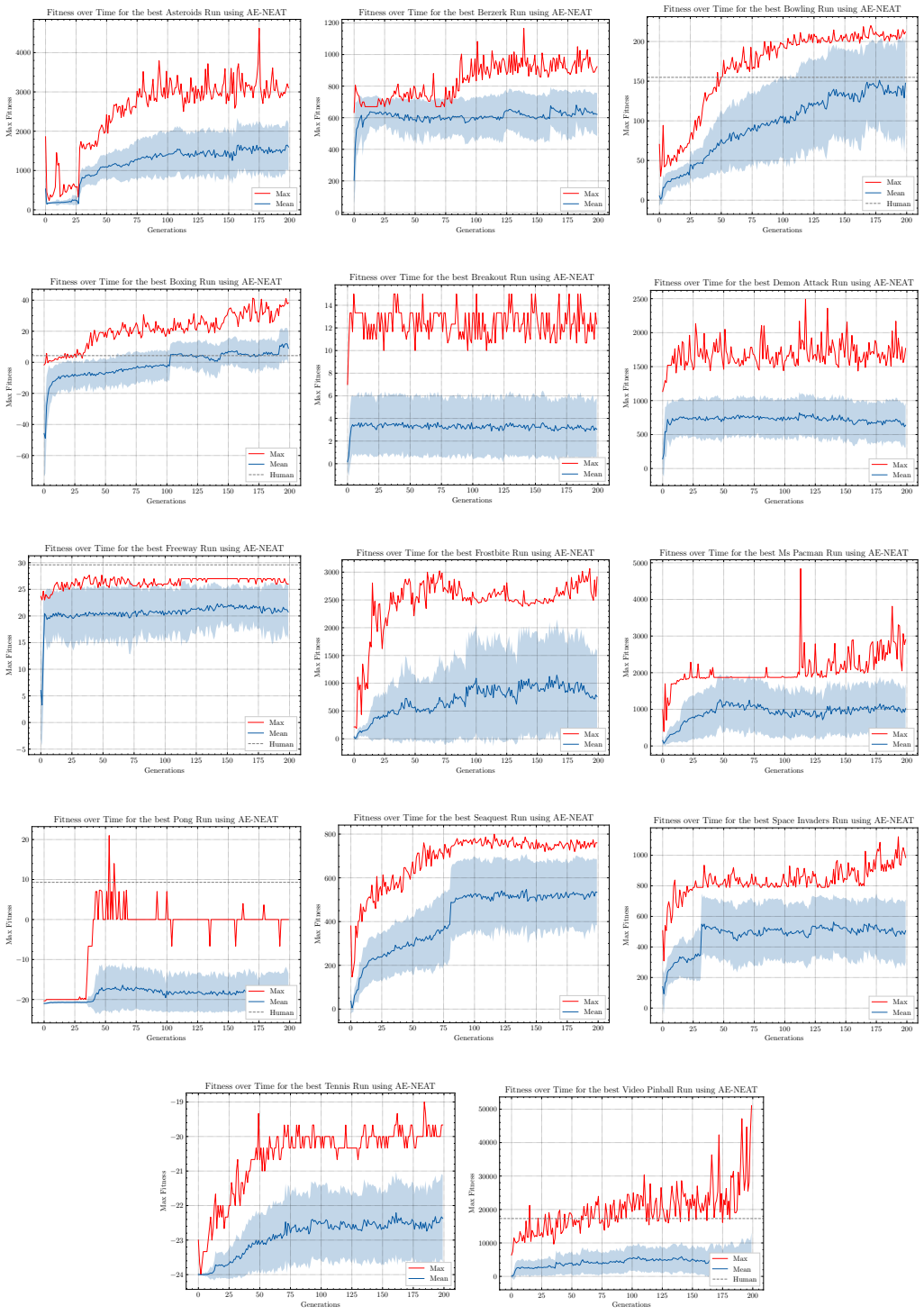


FIGURE 7.2: The fitness curves of the population for the best run of each game.

trained compressors. For learning the score and clock variables for Boxing, the online trained compressor had a higher average $F1$ score (0.45 compared to 0.31). In Breakout, the online trained compressor had a higher average R^2 value for agent localisation (0.56 compared to 0.40). For Berzerk, the online trained compressor had a higher average R^2 value for small object localisation (0.17 vs. 0.11). Finally, for Tennis, the online trained

TABLE 7.3: The localisation and classification performance in each category for the final compressors of the best run for each game.

	Localisation (Avg. R^2)			Classification (Avg. F1)	
Game	Small Object	Agent	Other	Score/Clock/Lives	Misc.
Asteroids	0.00	0.00	0.18	0.56	0.32
Berzerk	0.17	0.70	0.52	0.76	0.51
Bowling	0.59	0.93		0.86	0.97
Boxing		0.58	0.67	0.45	
Breakout	0.05	0.56		0.58	0.65
Demon Attack	0.02	-0.02	0.06	0.98	0.60
Freeway		0.39	0.74	0.12	
Frostbite		0.60	0.84	0.78	0.51
Ms Pacman		0.21	0.06	0.73	0.30
Pong	0.33	0.67	0.69	0.66	
Seaquest	0.32	0.65	0.27	0.82	0.70
Space Invaders	0.00	0.65	0.85	0.45	0.39
Tennis	0.28	0.77	0.80	0.73	
Video Pinball	0.00	-0.02		0.11	

compressor had a higher average $F1$ score for classifying the agent and opponent scores (0.73 compared to 0.64).

Of equal interest are the 16 instances where the online compressors scores were worse than the equivalent compressors trained during the independent evaluations. For Bowling (0.59 vs. 0.86) and Pong (0.33 vs. 0.45), the online trained compressors performed worse in for encoding the locations of the ball, captured in the small object localisation category. For agent localisation, the online trained compressors performed worse in Berzerk (0.70 vs. 0.83) and Space Invaders (0.65 vs. 0.92). For other object localisation, the online compressors performed worse in Demon Attack (0.06 vs. 0.13), Freeway (0.74 vs. 0.95), and Seaquest (0.27 vs. 0.35). Moving on to classification, the online trained compressors all recorded worse average $F1$ scores in the score/clock/lives category for Berzerk (0.76 vs. 0.82), Breakout (0.58 vs. 0.88), Freeway (0.12 vs. 0.22), Pong (0.66 vs. 0.95), Seaquest (0.82 vs. 0.88), and Video Pinball (0.11 vs. 0.23). Finally, for the miscellaneous category the online trained compressors were worse for Breakout (0.65 vs. 0.83) and Demon Attack (0.60 vs. 0.80).

Fig 7.3 examines the relationship between the quality of the compressor and the quality of policy for each hybrid agent. The x and y axes measure the overall classification and regression performance, respectively, of the probes trained using the state representations

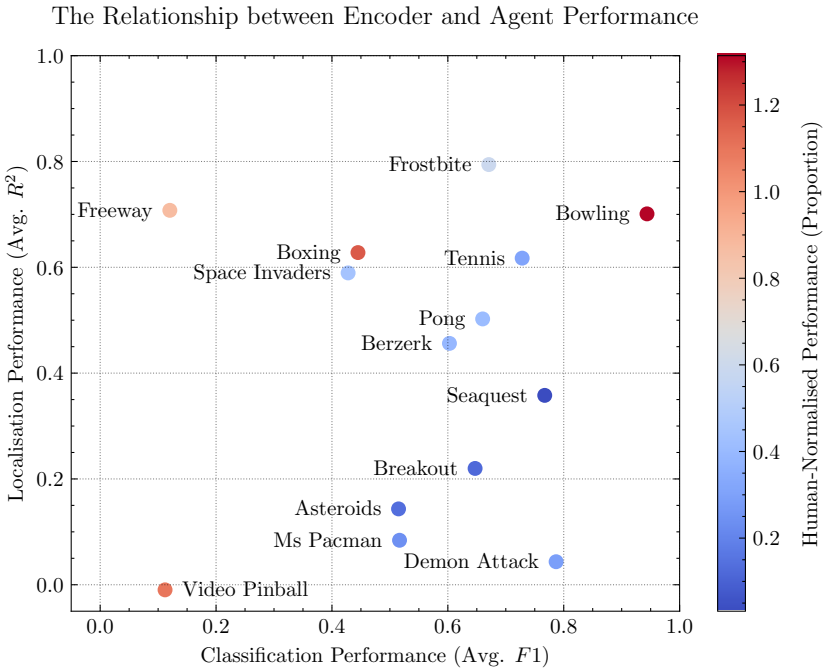


FIGURE 7.3: An illustration of the relationship between compressor/encoder and policy performance for the best AE-NEAT agents for each game.

generated by the compressors. The colour of each point indicates the average human-normalised performance of the particular agent. Although we know from our policy learning experiments using the AtariARI that having good information alone is not sufficient for finding a good policy (§6.2.1), Fig. 7.3 is still useful for investigating the results. For example, it illustrates that for Freeway and Video Pinball, knowledge about the score (the only classification state variable in both cases[4]) are not important for learning good policies. Additionally, it shows that knowledge about the positions of the ball and bumpers are not important for human-beating performance in Video Pinball. For Video Pinball, this is unsurprising as even a random strategy is able to beat a human strategy on average. Fig. 7.3 also highlights the importance of being able to accurately localise objects compared to being able to classify other state variables, such as a the number of lives, score, and the status of objects (e.g. the count of the number of igloo blocks formed in Frostbite). With the exception of the Video Pinball agent, whose who high human-normalised performance despite the poor compressor has been explained, the common factor between the other highest performing agents – for Freeway, Boxing, Frostbite, and Bowling – all have high localisation performance in common, despite substantially different classification performance.

[4]For Video Pinball the score is encoded in two separate bytes of RAM.

7.2.4 Evolved Polices

As mentioned earlier (§7.2.2), the fitness curve for Pong alludes to some interesting polices developed by the agents throughout the training run. In particular, the best agent, which is found in Generation 53, achieves a maximum possible score of 21 points across all three episodes during the fitness evaluation, but shortly after the population converged on a strategy that consistently yielded an average score of zero points from Generation 68. Roll-outs of the agent confirm that the best agent exploits the loophole in the game first observed for one of the runs of the AtariARI policy learning experiments (§6.2.3), learning to move to a position that guarantees it is able to always beat the opponent for a particular starting trajectory. However, unlike that run, the population does not converge on this local optimum once discovered. Instead, the population converged on a different strategy the consistently results in a draw. For this strategy, the agents discovered a position where the ball is infinitely bounced between the agent and the opponent, resulting in neither scoring a point before the frame cap is reached. This behaviour is similar to the agent refusing to serve the ball in Tennis. However, this strategy is not always successful, due to the stochasiticity introduced into the agents actions. The population is unable to move on from this local optima, as developing policies that we know to be more popular in the long run, such as those that track the ball and/or the opponent perform much poorer at this early stage of development.

7.2.5 Evolved Architectures

Compared to the policy learning experiments conducted using the AtariARI, it is more difficult to interpret the evolved policy networks of the hybrid agents. Aside from the fact that the policy networks of the hybrid agents are much larger (due to the 40 inputs from the compressors), due to the nature of the representation learned by the autoencoders, we do not know which dimensions encode information pertinent to different state variables (e.g. object positions, scores, etc.). Therefore, evidence of the structures required for combating partial observability is harder to find. Table 7.4 summarises some of the characteristics of the evolved policy networks for the best AE-NEAT agents. Understandably, the solutions discovered later in the evolutionary runs include more hidden nodes than those found earlier. This is because solutions start out minimally (with only input, bias and output nodes) and evolve hidden nodes over time. A prime

example of this are the best solutions for Breakout and Video Pinball. The best solution for Breakout was discovered during the 5th generation and contains only a single hidden node, whereas the best solution for Video Pinball was found during the 200th generation and contains 25 hidden nodes. Solutions found early on in the evolutionary process tend to have evolved fewer hidden nodes than solutions found later.

There are two reasons that solutions are found early on. First, good solutions lie close to the initial conditions and only require simple network architectures to learn good strategies. In this case, additional structural mutations and weight optimisations offer little to no improvement. An example of such a game is Freeway. As described in §7.2.1, the best Freeway agent achieves 90% of expert human performance with a very simple network architecture with only five hidden nodes. Second, better polices lie far from the initial conditions and many structural mutations are required before a better policy is discovered. This appears to be the case for Breakout, where the best solution is found during the 5th generation, yet this solution is only able to accrue on average 3.8 points. Very poor in comparison to the 31.8 points accrued on average by the expert human player. The poor performance in Breakout is likely a result of the fact that policy networks need to reach some critical complexity before they are able to utilise their structure for better performance. Although structural innovations are protected by the NEAT speciation policy, if these networks perform poorly in comparison to simpler networks and require many mutations to take advantage of their more complex structure, they will not be allocated enough offspring to adequately search for more complex structures. This exploration vs. exploitation trade-off is discussed in the following section (§7.3.3).

7.3 Discussion

Our analysis of the agents produced by our experiments yielded several interesting outcomes worthy of discussion. In this section we address these points. We delay our discussion of potential avenues of future work until the next chapter.

7.3.1 Minimum Policy Network Complexity

One of the observations from our results is that for some of the poorest performing games, such as Breakout and Demon Attack, there is a lack of improvement in fitness

TABLE 7.4: The generation discovered, and the number of (used) hidden nodes and connections for the policy networks of the best AE-NEAT agents for each game.

Game	Generation	Hidden Nodes	Connections
Asteroids	175	23	620
Berzerk	139	21	770
Bowling	171	23	270
Boxing	170	19	756
Breakout	5	1	166
Demon Attack	117	10	257
Freeway	39	5	131
Frostbite	193	32	801
Ms Pacman	113	16	398
Pong	53	3	244
Seaquest	116	9	749
Space Invaders	193	19	273
Tennis	184	32	781
Video Pinball	199	25	411

from the outset. This suggests that the initial conditions, i.e. minimal networks without any hidden nodes, lie in a flat region of the policy space, far from any architecture that is able to yield better fitness. In other words, many mutations are required before the positive effects of successful mutations are realised by rewards of higher fitness. This make the incremental process of evolving increasingly better policy networks extremely difficult, as all individuals in the population are awarded similar fitnesses. This means that the search for weights and topologies is essentially random and unguided. One way to address this is to increase the strength and frequency of structural mutations, however, make these too strong and not only is it likely that for other games the mutations will be too powerful to enable the progress observed searching those policy spaces, but when better solutions are found, they will be very slowly refined as the strength of mutations will frequently overshoot them. The severity of this problem is unique to neuroevolution methods that evolve both weights and topologies, compared to those that evolve or optimise through gradient descent the weights of fix topology networks that are often designed on the safe side with more capacity than is needed. One potential solution that might be worthy of investigation is seeding the initial populations with different starter architectures to help overcome the initial conditions, while ensuring that the mutations are not too powerful to prevent learning. Alternatively, the strength of mutations could be modified and reduced or increased throughout evolution depending on the progress of the population. NEAT already includes a mechanism to replace weights, but rarely

applying strong structural mutations could also be of benefit.

7.3.2 Compressor Performance

One potential explanation for the difference in performance between the compressors trained offline using the data collected by trained Proximal Policy Optimisation (PPO) agents, and the compressors trained online using the data collected during the training of the NEAT agents is the difference in gameplay performance between these two types of agents. The training, validation, and test data that was collected for training the probes, was generated using the data collected for PPO agents, and may not be representative of the game states encountered by the hybrid agents. This difference between the data used for training the compressors may bias the results. For example, in Breakout, the PPO agent was able to learn a very good policy for playing the game that far exceeded expert human performance. As result the observations collected by this agent have higher scores and more of the bricks destroyed. This is likely the cause of the lower classification scores achieved by the online trained compressor, that was never given the opportunity to train on these game states. This explanation seems to hold, as the object localisation performance is similar between the two models, and the agent localisation performance is in fact much higher for the online trained compressor. However, this argument also holds for Pong and Video Pinball, where the PPO agents produced much higher scores than the AE-NEAT agents, but does not hold for Demon Attack, or Boxing, where the compressors from the AE-NEAT agents performed similarly and outperformed the compressors trained on PPO collected observations, despite the PPO agents receiving vastly higher scores.

Compressor performance is clearly a limiting factor in the performance of some of the AE-NEAT agents, particularly in the cases where the ability of the compressors to encode positional information of different objects. This is one area of improvement discussed in Chapter 8.

7.3.3 Balancing Exploration and Exploitation

The evolved solutions for some games attest to the fact that evolutionary algorithms are not immune to becoming stuck in local optima. This was a phenomena we observed

in both the policy learning using the AtariARI, and with AE-NEAT. The most obvious examples are the loopholes exploited by the evolved solutions for Tennis A in our AtariARI experiments, and Pong in both the AtariARI and AE-NEAT experiments. In these cases, the local optima are particularly problematic because they are discovered early, before the agents are able to develop what we know to be more promising strategies, such as tracking the ball or opponent. Despite speciation, in the short term, alternative strategies are not rewarded highly enough to be preserved; highlighting the need for a better balance between exploration and exploitation. It is possible that including elements of novelty search or intrinsic motivation may help to achieve this balance. It is unclear how other researchers avoided the loopholes introduced by the frame cap, though we suspect that they took a similar approach to us. The reported score of zero points for the HyperNEAT agent reported by Hausknecht et al. (2014) leads us to believe that this loophole may have gone unnoticed.

7.4 Summary

In this chapter, we performed experiments to evaluate our hybrid learning method, AE-NEAT, which was detailed in Chapter 4. This was after having evaluated the state representation and policy learning components of our method in isolation, in chapters 5 and 6 respectively. Overall, we found that through the separation of state representation and policy learning, we were indeed able to scale NEAT, a neuroevolution method that evolves both the weights and topology of neural networks to video game domains with high-dimensional, raw pixel inputs. Furthermore, we found agents that were able to outperform alternative neuroevolution methods on several games, proving that our proposed approach is worth further investigation in the future. Despite this, both our method and other neuroevolution methods lag far behind the state-of-the-art gradient-based deep reinforcement learning general Atari game playing method. In the following chapter, we round up the contributions we have made throughout our experimental process to reach this point, and propose avenues of future work for closing the gap on gradient-based deep reinforcement learning methods.

Chapter 8

Conclusions

This research aimed to investigate the plausibility of using a separated state representation and policy learning method to scale topology and weight evolving neuroevolution to vision-based general video game playing (GVGP). In the following sections, we discuss the contributions, implications, and limitations of our research, before presenting avenues for future work.

8.1 Contributions

In this section, we summarise the contributions of our work. We begin by discussing the most significant contribution related to the main research aim, before discussing secondary contributions that we have also made.

8.1.1 Scaling Topology and Weight Evolving Neuroevolution to Vision-Based GVGP

The main contribution of this research was showing that, through the separation of state representation and policy learning, topology and weight evolving neuroevolution methods can be used to train vision-based GVGP agents. We proposed a method, Autoencoder-Augmented NEAT (AE-NEAT), that we showed was able to train agents that exceeded expert human performance in a number of Atari games, and performed

competitively against the reported results of other state-of-the-art evolutionary reinforcement learning (ERL) methods.

As discussed in our review of the literature (Chapter 3), prior work had shown that NEAT, was unable to learn policies from raw pixels (Hausknecht et al., 2014) and had only been shown to be successful at evolving policies for a single vision-based aim-and-shoot task when combined with a compression network that was trained in a *supervised* manner *before* evolving policies. Our method, AE-NEAT, combines NEAT with a compression network that is trained in an *unsupervised* manner *simultaneously* alongside the policy networks. This removes the need for pre-training and labelled data, enhancing the general applicability to different tasks and domains.

Our findings are significant for two reasons. First and foremost, our method opens up the ability to harness additional benefits of ERL that motivated our research and were discussed in Chapter 1. These benefits include greater exploration of the policy space, and the ability to learn given only sparse rewards. Second, vision-based GVGP benchmarks, such as the Atari games used in our experiments, are commonly used to assess the ability of algorithms to find solutions to complex reinforcement learning problems with high-dimensional inputs. Success on this benchmark provides an indication that AE-NEAT may be useful for solving tasks with similar properties.

8.1.2 Deeper Insights into the Quality of Representations Learned by Autoencoders

In Chapter 5, we investigated promising autoencoder-based representation learning method to use for training compressor networks. We compared not only the quality of representations learned by different types of autoencoders (undercomplete, variational, and disentangled variational), but also the effect of representation size on the quality of representations. Our results provide insight into the relationship between different types of models, representation quality (from a policy learning perspective) and representation size.

We also provide the most comprehensive evaluation to date on the use of weighted reconstruction errors for focusing the representations learned by autoencoders on important features for reinforcement learning. Prior to our work, work by Nylend (2017) indicated

that the use of weighted reconstruction errors could improve the quality of the representations learned by autoencoders (from a policy learning perspective) by focusing the reconstruction on dynamic portions of the images. However, the only evidence to support this was that the quality of the reconstructions improved, in particular the presence of small objects in the reconstructions. However, in our analysis, we investigated the impact of using weighted reconstruction error on different types of autoencoders and for different representation sizes. Our method for evaluating the impact of weighted reconstruction errors was more sophisticated and showed that despite visible improvements in the reconstructions, the use of a weighted reconstruction error measure only led to improvements under certain conditions.

8.1.3 A New Method for Evaluating Learned State Representations using the AtariARI

In our work, we extended the representation evaluation method proposed by Anand et al. (2019) alongside the release of the AtariARI, introducing non-linear, and regression probes to enhance evaluations. These probes reveal insights into the quality of learned state representations that could not have been detected previously. Our method incorporates regression evaluations for appropriate variables, and uses a normalised metric to retain the ability to compare performance between state variables. It also maintains conformance with the existing categorisations of variables defined by Anand et al. (2019). Using our method, the performance of each model can be summarised down to two values, the overall localisation and classification performance. Previously, models were summarised by a single overall classification performance value, however, separating categorical and discrete variables allows for a better and more interpretable evaluation of these respective categories. We have open-sourced this extended evaluation method (github.com/adamtupper/atari-representation-learning) so that it can be used by other researchers.

8.1.4 PyNEAT: A New Implementation of NEAT in Python

The existing implementation of NEAT written in Python (McIntyre et al., 2017) is substantially different from the original implementation of NEAT by Stanley and Miikkulainen (2002). For our research, we developed our own implementation of NEAT,

PyNEAT, that we experimentally verified to ensure its effectiveness. This is, to the best of our knowledge, the closest available implementation of NEAT in Python to the original implementation. Having an original implementation of NEAT in Python is useful for incorporating NEAT into existing methodologies and pipelines with other machine learning libraries that are also available for Python (e.g. PyTorch, TensorFlow, and Scikit-learn) and also for performing benchmarking and comparisons. NEAT is renowned as a complex and challenging algorithm to implement, and so we have also open sourced PyNEAT for the benefit of other researchers and practitioners here: github.com/adamtupper/pyneat.

8.1.5 The Simplicity of Solutions for Complex RL Problems

Our policy learning experiments using the representations provided by the AtariARI (Chapter 6) and learned state representations (Chapter 7) demonstrated the simplicity of high performing solutions that exist for some Atari games, given compact state representations. As far as we are aware, this is the first work to discuss and present the evolved architectures for the solutions to games. These architectures shed more light on the complexity of different games and the complexity of solutions required to solve them. This is an important benefit of evolving solutions from minimal structures. Outside of this work, discussions on the complexity of games are centred around the complexity of different algorithms that are able to find solutions. Our results provide insight along a different axis, the complexity of the networks required to solve them.

8.2 Limitations

The gap between ERL and gradient-based RL methods While our method performs well when compared with other ERL methods, both fall short of the current state-of-the-art deep RL methods. The best deep RL GVGP method, Agent57 (Badia et al., 2020), is able to train agents that surpass expert human perform in all Atari games. Despite this, our method adds to the toolbox of methods that can be used to solve RL problems, and might serve as a base upon which further improvement can be made to eventually challenge deep RL methods. We discuss avenues for closing this gap and further improving our method in §8.3.

Testing on a wider selection of games We conducted our experiments using Atari games as the benchmark for evaluating and comparing our results. While this benchmark is widely used, it does not guarantee that our method will generalise to other video game domains or other domains with high-dimensional input spaces. However, in contrast to nearly all other RL methods, evolutionary or otherwise, in our experiments we performed minimal preprocessing on the images fed to the agents (only grey scale conversions). This lack of reliance on the properties of the images from Atari domain gives confidence that our method might generalise well to other domains.

Proof of concept We believe that our methodology was effective in demonstrating that our method is able to train vision-based GVGP agents, however, our results represent a proof of concept that this approach can work and holds promise. The significant time and resource constraints prohibited us from performing large number of runs in our evaluations and from conducting extensive hyperparameter tuning. As a result, our results do not represent the limit of the performance that can be achieved using this method, and instead serve as a proof of concept that this approach can work, even if only shown to do so for a small selection of games.

8.3 Future Work

We conclude this thesis by briefly discussing several potential avenues for future research. The modular nature of the compressor-controller agent design, and our proposed method for training these components, offers ample flexibility for integrating improvements. Furthermore, the independent evaluations of each component conducted in chapters 5 and 6 help guide our recommendations for the areas where future effort should be invested.

8.3.1 State Representation Learning (Compressor) Improvements

We used an autoencoder for training the compressor networks for our agents. However, the modularity of the compressor-controller agent design and the flexibility of our proposed training method allows for any unsupervised representation learning method to be substituted in the autoencoders place. For example, the SpatioTemporal DeepInfoMax

(ST-DIM) method proposed by Anand et al. (2019). Furthermore, our results showed clear inconsistencies in the ability of the different autoencoder-based compression methods we tested to extract and encode important features required for policy learning, both between and within games (§5.5.2). Future research is needed to investigate alternative methods for state representation learning, that might learn representations better suited for policy learning, yet maintain the ability to learn in an unsupervised manner.

8.3.2 Policy Learning (Controller) Improvements

NEAT (Stanley and Miikkulainen, 2002) is a powerful topology and weight evolving neuroevolution algorithm that is widely used. However, a number of extensions have been proposed (Nodine, 2010, Stanley et al., 2009, Whiteson et al., 2005) that could be assessed in future work, to see if they yield improved performance. An alternative avenue of investigation is whether or not policy learning can be improved through the inclusion of some form of neuromodulation.

In biological brains, the properties of neurons and the synapses (connections) between them can be altered by other neurons, or chemicals released by other neurons, through a process called neuromodulation (Katz and Calin-Jageman, 2009). Neuromodulatory actions can cause temporary excitation or inhibition, and can also cause lasting changes in the excitability of neurons and the strength of synapses (Katz and Calin-Jageman, 2009). This process allows for localised learning.

Modulatory neurons, proposed by Soltoggio et al. (2008), aim to emulate the process of neuromodulation in artificial neural networks by altering the learning rate of the network at the connection level. This localised regulation of plasticity allows for the selective activation of learning in specific parts of the network in response to different inputs. In reinforcement learning, this corresponds to allowing selective learning in specific parts of the network in response to specific changes in the environment (Ellefsen et al., 2015).

For simple reinforcement learning tasks, such as maze navigation (Soltoggio et al., 2008) and decision making (Ellefsen et al., 2015), in changing environments, the use of modulatory neurons has successfully mitigated catastrophic forgetting and enabled the evolution of agents that are robust to changes in the environment and reward. The inclusion of modulatory neurons in the controller could improve performance in games where the

environment changes over time, such as those with different levels or those in which the entire environment is not visible to the agent at once (e.g. Berzerk).

8.3.3 Alternative Sampling Methods for Selecting the Observations used to Train the Compressor

How the observations used to train the compressor are selected is important because this is the mechanism through which the controllers influence the features that are learned. We used random sampling in our experiments to ensure that (a) agents with different strategies contributed to the training set, and (b) observations from throughout the evaluation episodes were included. However, further investigation is required to see whether or not more sophisticated sampling methods could be used to reduce training times or produce higher performing agents. For instance, Alvernaz and Togelius (2017) used a reconstruction error threshold to select the observations that were included in the training set, but it is not known whether the extra computation (reconstructing each image and calculating the reconstruction error) at each time step during the fitness evaluations is worthwhile.

8.3.4 Multi-Task Learning

A natural extension of GVGP is to train agents that can learn and remember how to play multiple games. A final avenue of future work is to investigate how our method can be extended to achieve this goal. This is worthy of exploration because there are several benefits offered by topology and weight evolving neuroevolution algorithms that make them particularly well suited for multi-task problems.

First, population-based evolutionary algorithms are inherently suited to multi-task learning problems because they evolve a population of solutions. This is advantageous because multi-objective optimisation problems, such as multi-task learning, naturally give rise to sets of Pareto-optimal solutions (Deb, 2011).

Second, weight and topology evolving neuroevolution algorithms allow us to evolve modular network architectures that consist of sub-networks that can be reused or perform specific functions. This has already been shown to improve performance in simple multi-task learning problems (Ellefsen et al., 2015). The gradient-based deep reinforcement

learning algorithms that currently dominate GVGP cannot achieve this because they are unable to optimise modularity.

Finally, the ability to evolve the topology of networks also allows extra capacity to be added to the networks as required to help learn multiple tasks. Current gradient descent-based methods resort to very large, highly overparameterised networks to avoid this issue (Fernando et al., 2017, Rusu et al., 2016), but this approach does not scale well with the number of tasks.

Appendix A

Additional State Representation Learning Details

This appendix includes additional details related to the state representation learning experiments and results presented in Chapter 5.

A.1 Proximal Policy Optimisation (PPO) Agents for Gameplay Image Collection

Figure A.1 displays the training curves for each PPO agent trained for collecting gameplay images. Each agent was trained for 10 million timesteps, using the same hyperparameters reported alongside the algorithm in the original paper (Schulman et al., 2017).

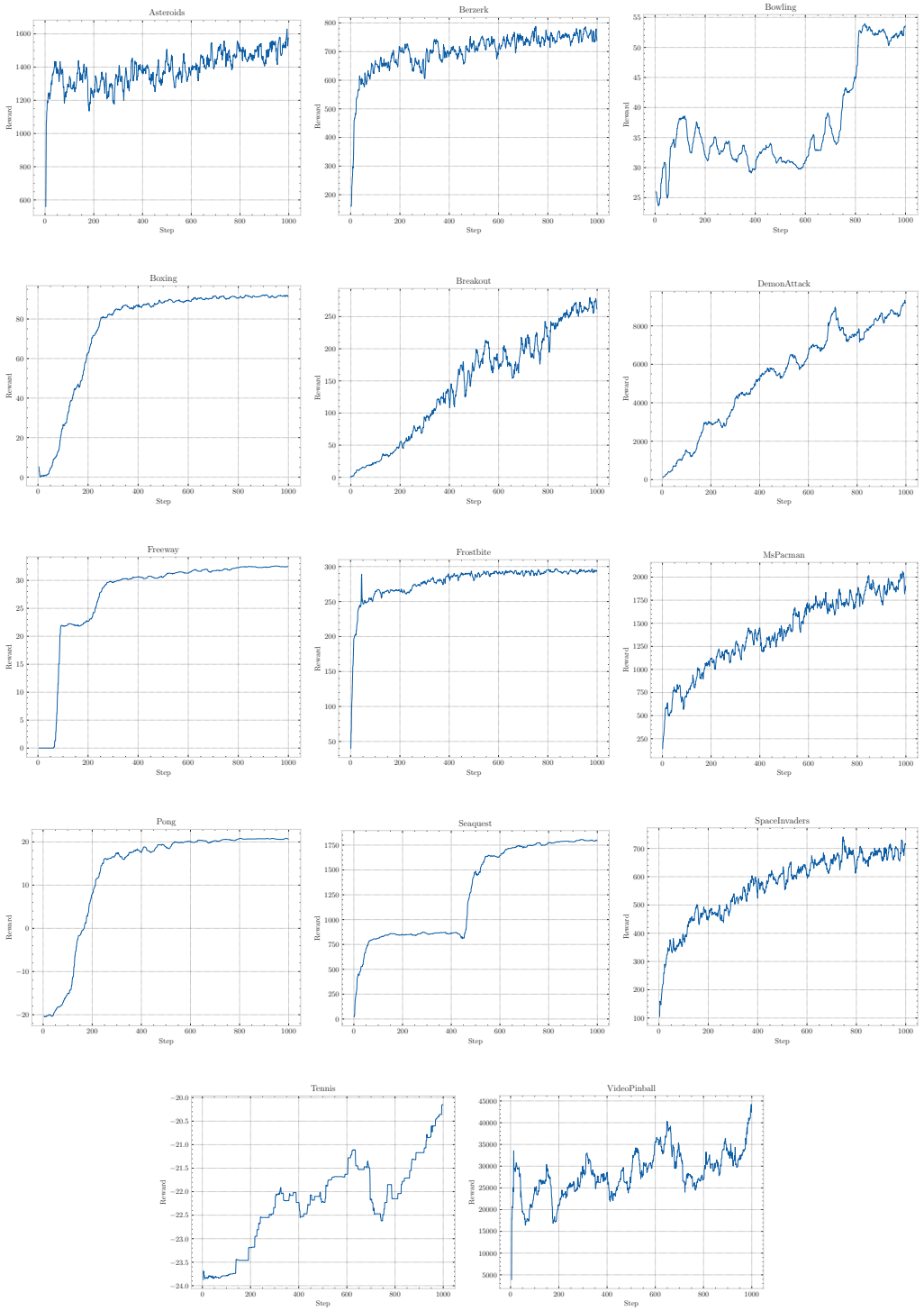


FIGURE A.1: Reward over time for each PPO agent on each game.

Appendix B

Additional AtariARI Policy Learning Details

This appendix includes additional details related to the policy learning experiments and results presented in Chapter 6.

B.1 AtariARI Policy Learning Experiments

Table B.1 provides a full list of the hyperparameters used to evolve policy networks using PyNEAT. These experiments used the state representations provided by the AtariARI as the input to the networks.

TABLE B.1: Common hyperparameters for NEAT for each game for the AtariARI policy learning experiments.

Hyperparameter	Value
Generations	200
Population Size	130
Add Node Probability	0.03
Add Connection Probability	0.05
Mutate Weights Probability	0.8
Weight Range	[-30, 30]
Weight Perturb Power	0.05
Weight Init Power	3.0
Weight Replace Probability	0.1
Compatibility Distance Disjoint/Excess Gene Coefficient	1.0
Compatibility Distance Weight Difference Coefficient	0.4
Compatibility Distance Threshold	3.0
Normalise Gene Distance	False
Species Fitness Function	max
Max Stagnation	15
Species Elitism	2
Mutate Only Probability	0.25
Average Crossover Probability	0.4
Crossover Only Probability	0.2
Inter-Species Crossover Probability	0.001
Number of Elites	1
Elitism Threshold	5
Survival Threshold	0.2
Gene Disable Probability	0.75
Initial Connection Probability	1.0
Feed-Forward	False
Activation Function	Sigmoid

B.2 Evolved AtariARI Agent Architectures

The following figures show the evolved architecture for the best AtariARI agents for each game. The width of the connections is proportional to the magnitude of the connection weight. On the left-hand side are the inputs to the network (the state variables provided by the the AtariARI) and on the right-hand side are the outputs of the network. Each output represents a legal action the agent can perform.

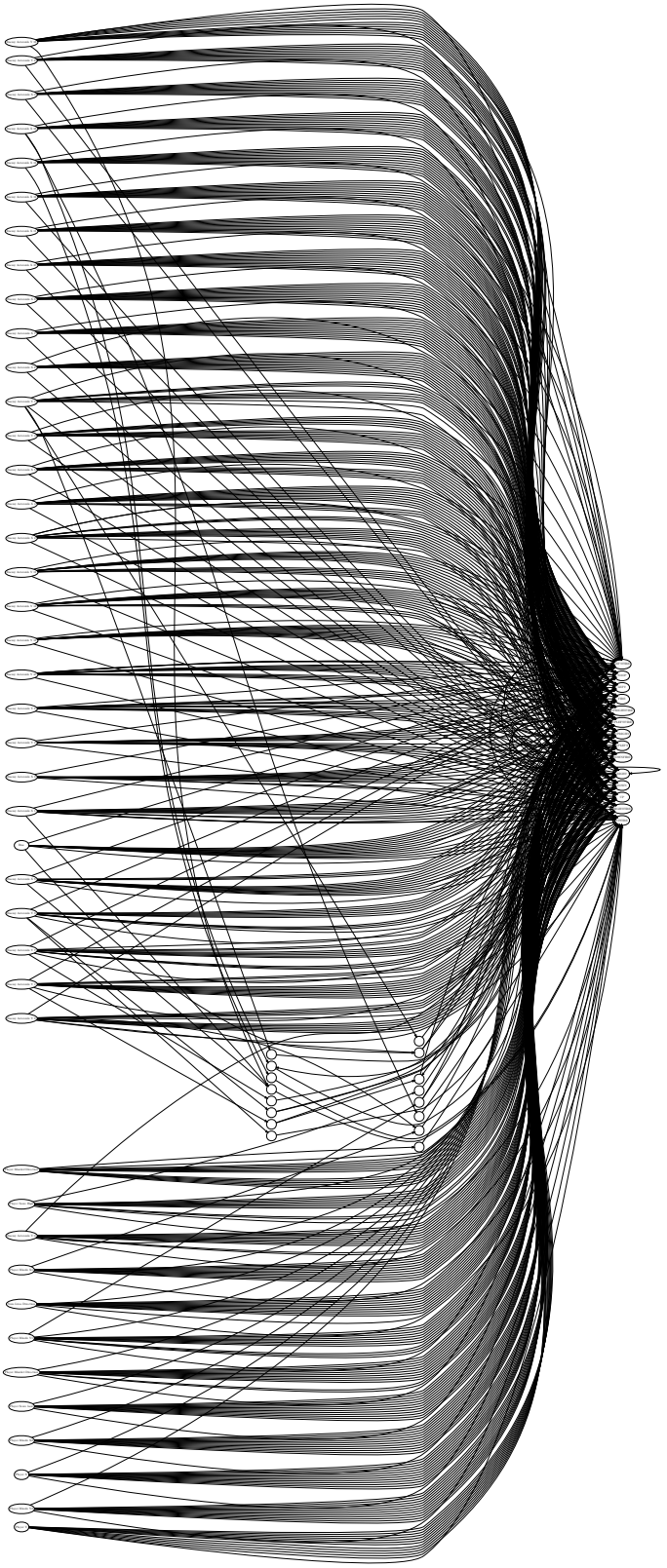


FIGURE B.1: The evolved network architecture for the top performing Asteroids agent.

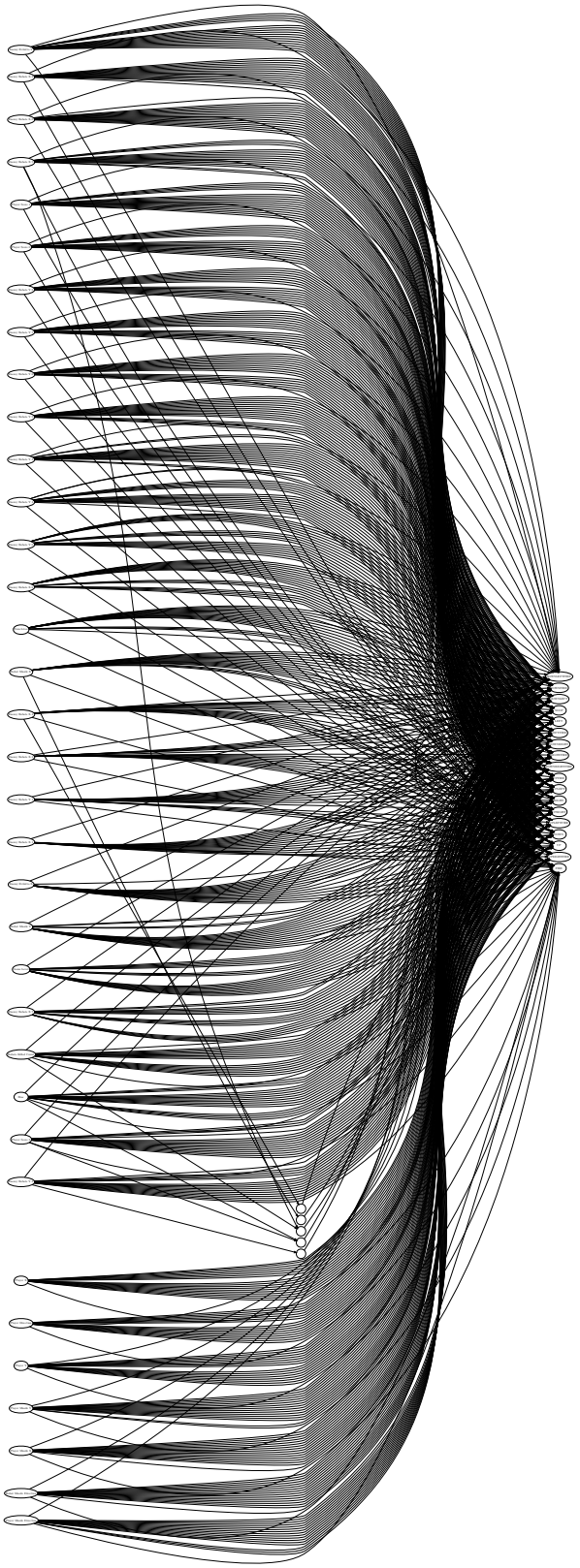


FIGURE B.2: The evolved network architecture for the top performing Berzerk agent.

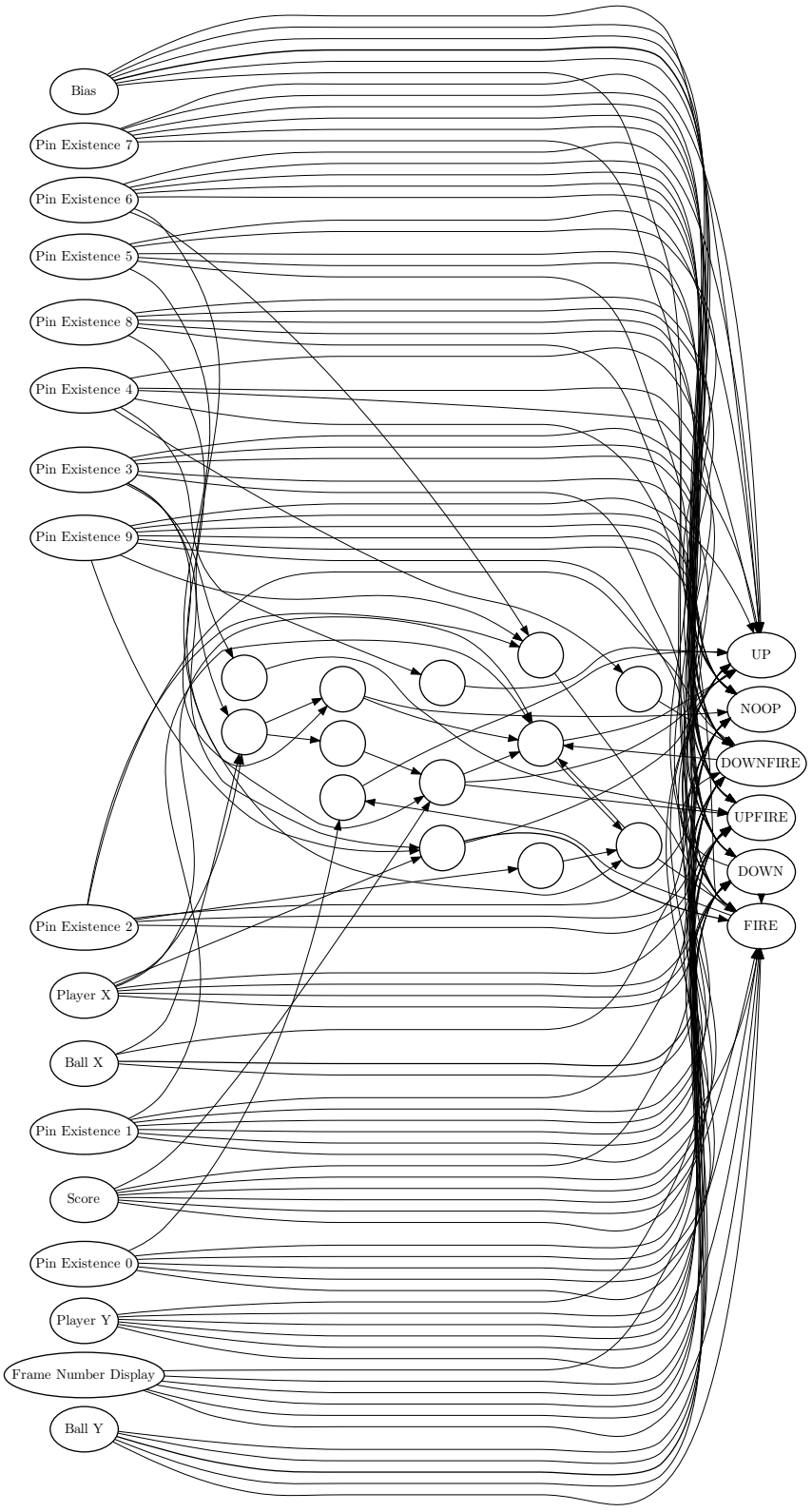


FIGURE B.3: The evolved network architecture for the top performing Bowling agent.

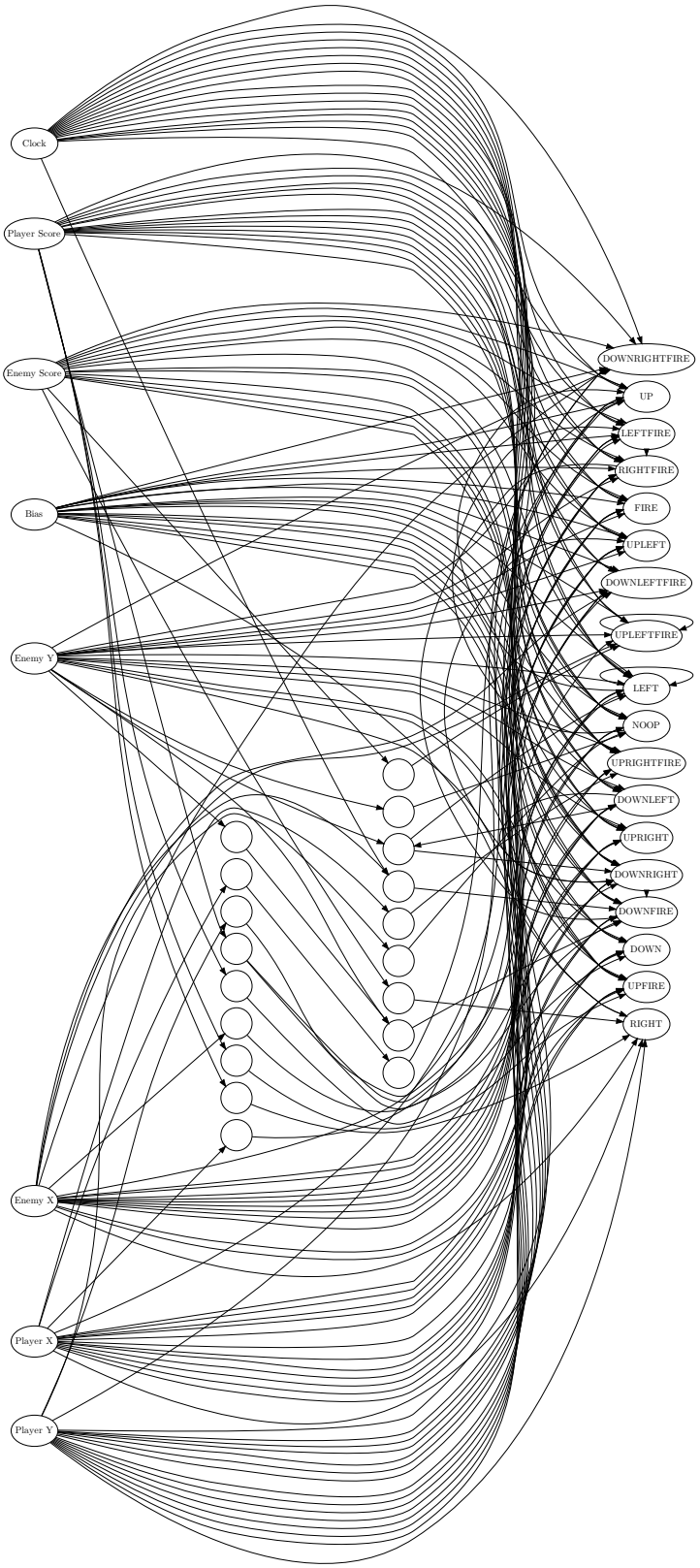


FIGURE B.4: The evolved network architecture for the top performing Boxing agent.

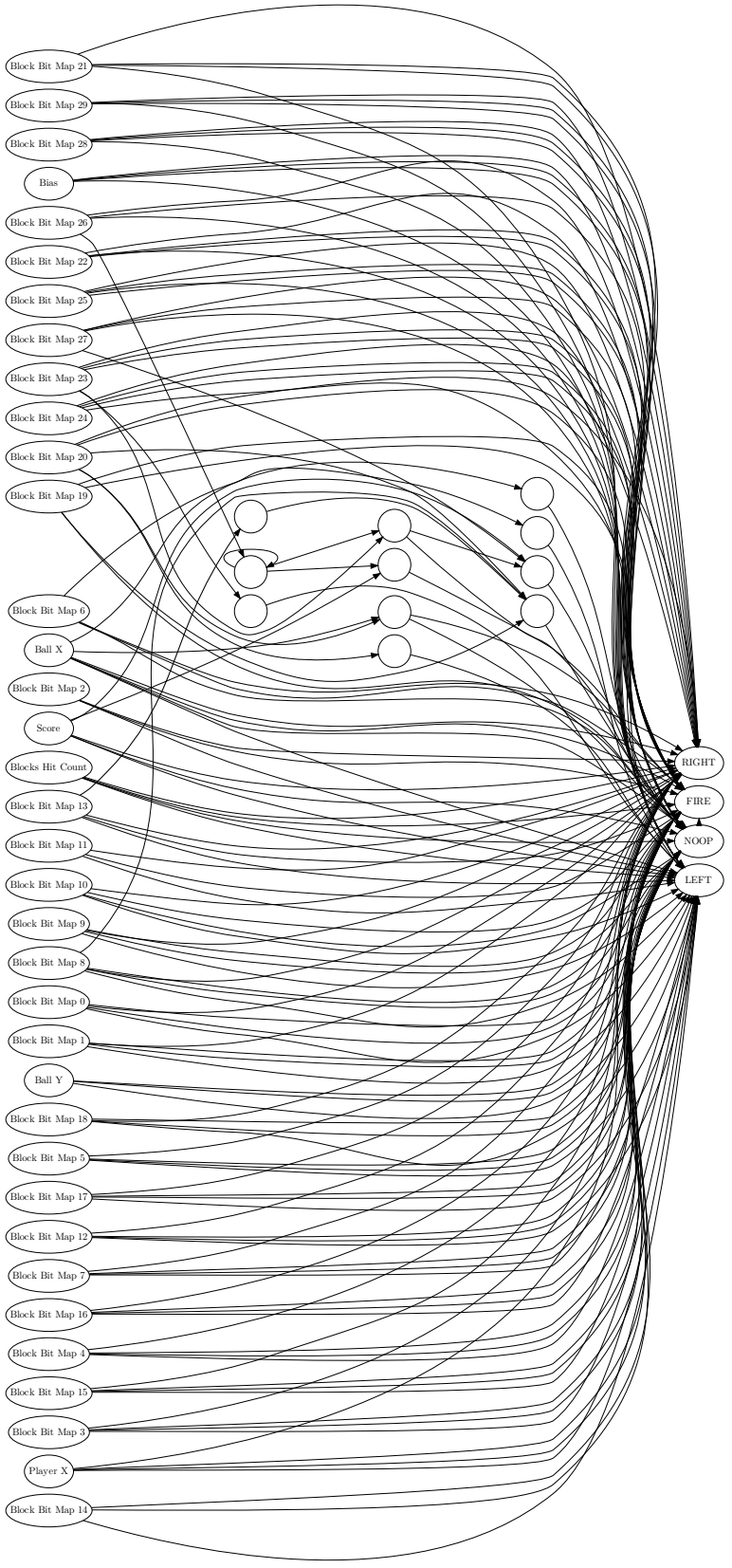


FIGURE B.5: The evolved network architecture for the top performing Breakout agent.

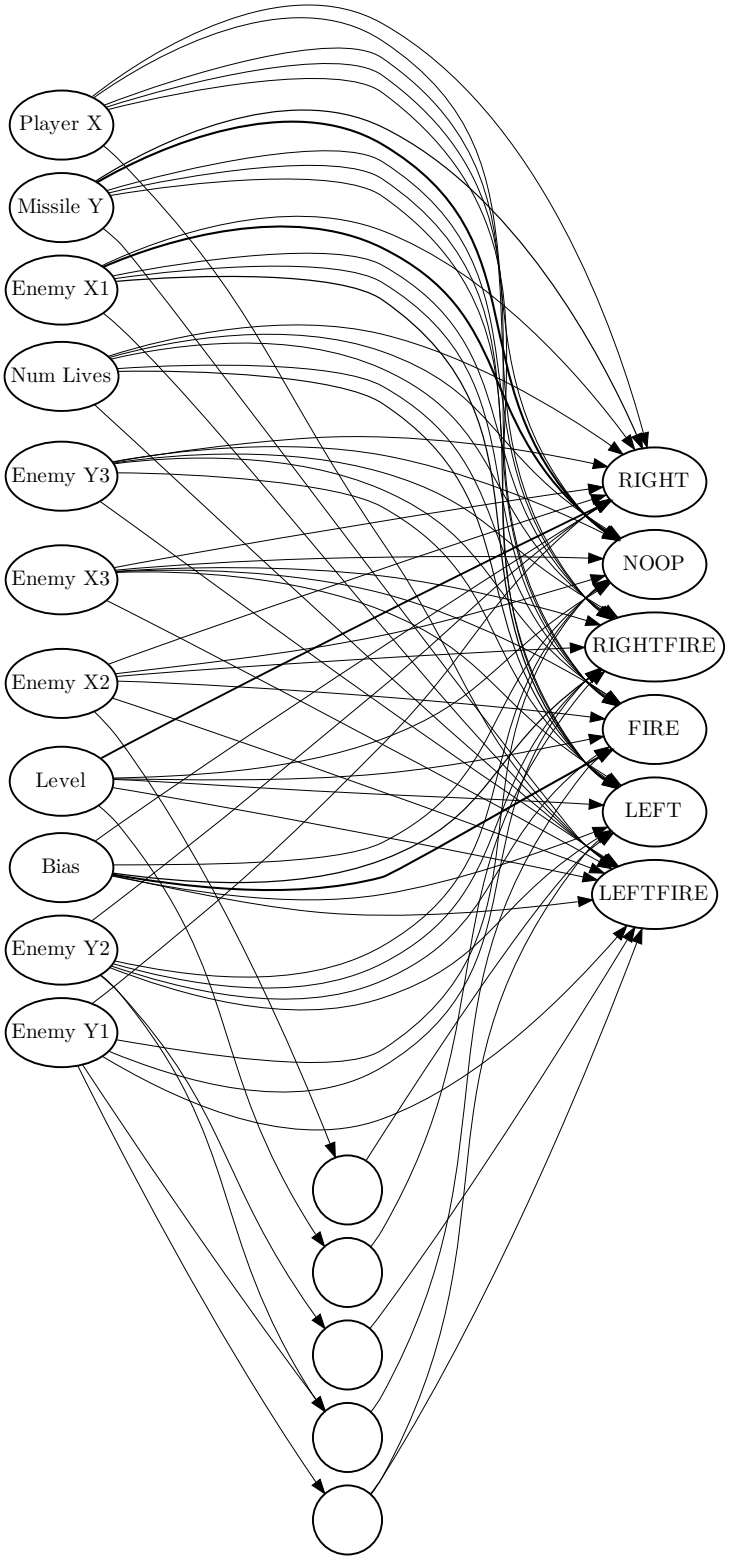


FIGURE B.6: The evolved network architecture for the top performing Demon Attack agent.

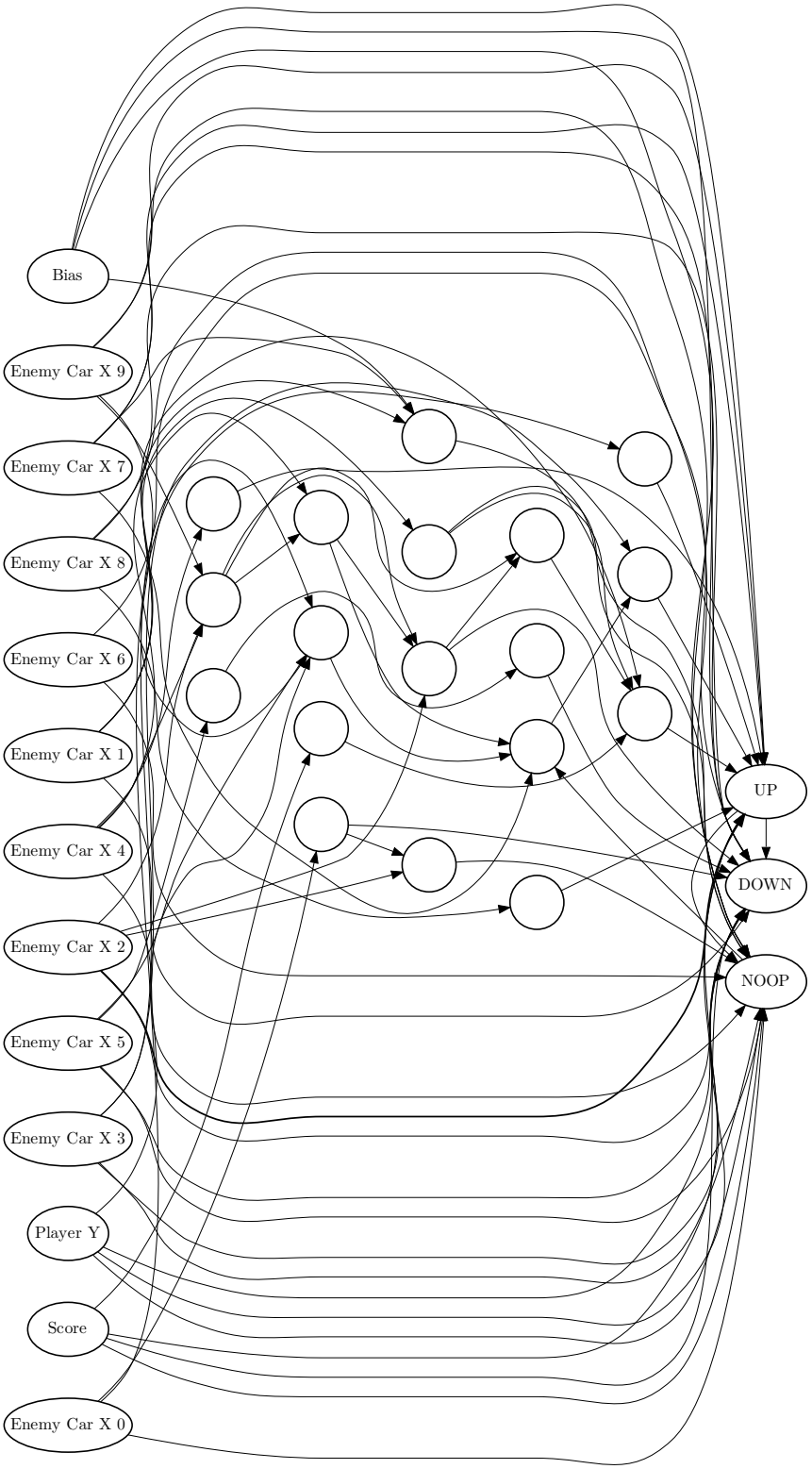


FIGURE B.7: The evolved network architecture for the top performing Freeway agent.

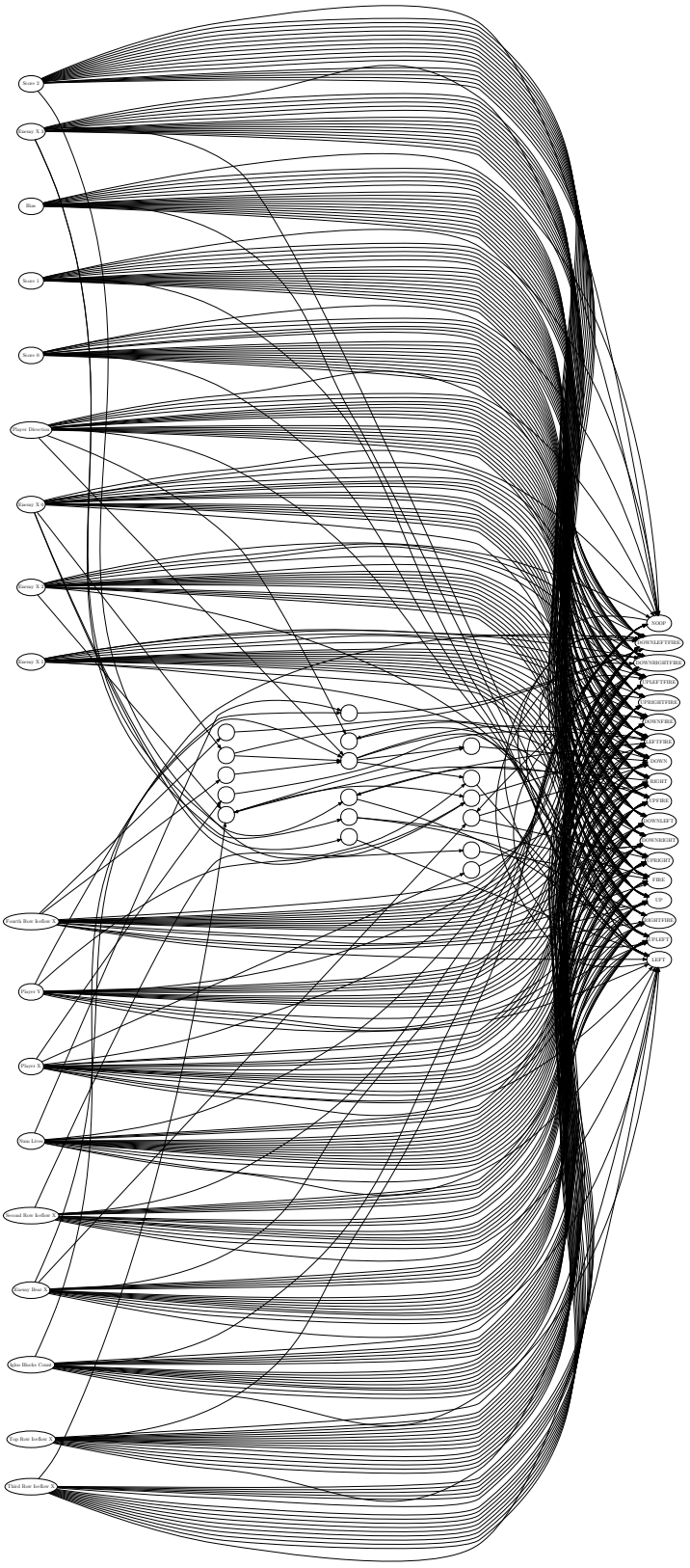


FIGURE B.8: The evolved network architecture for the top performing Frostbite agent.

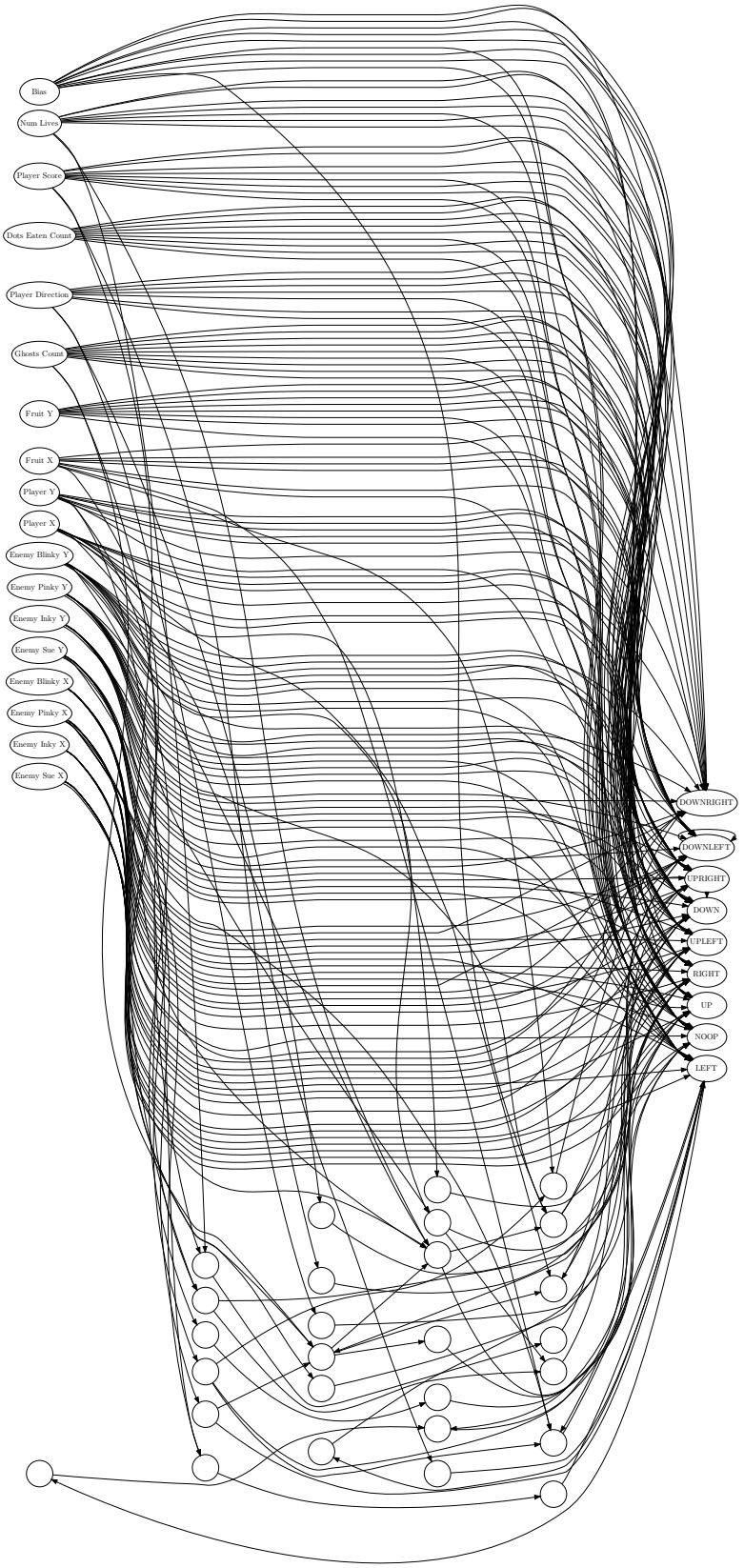


FIGURE B.9: The evolved network architecture for the top performing Ms Pacman agent.

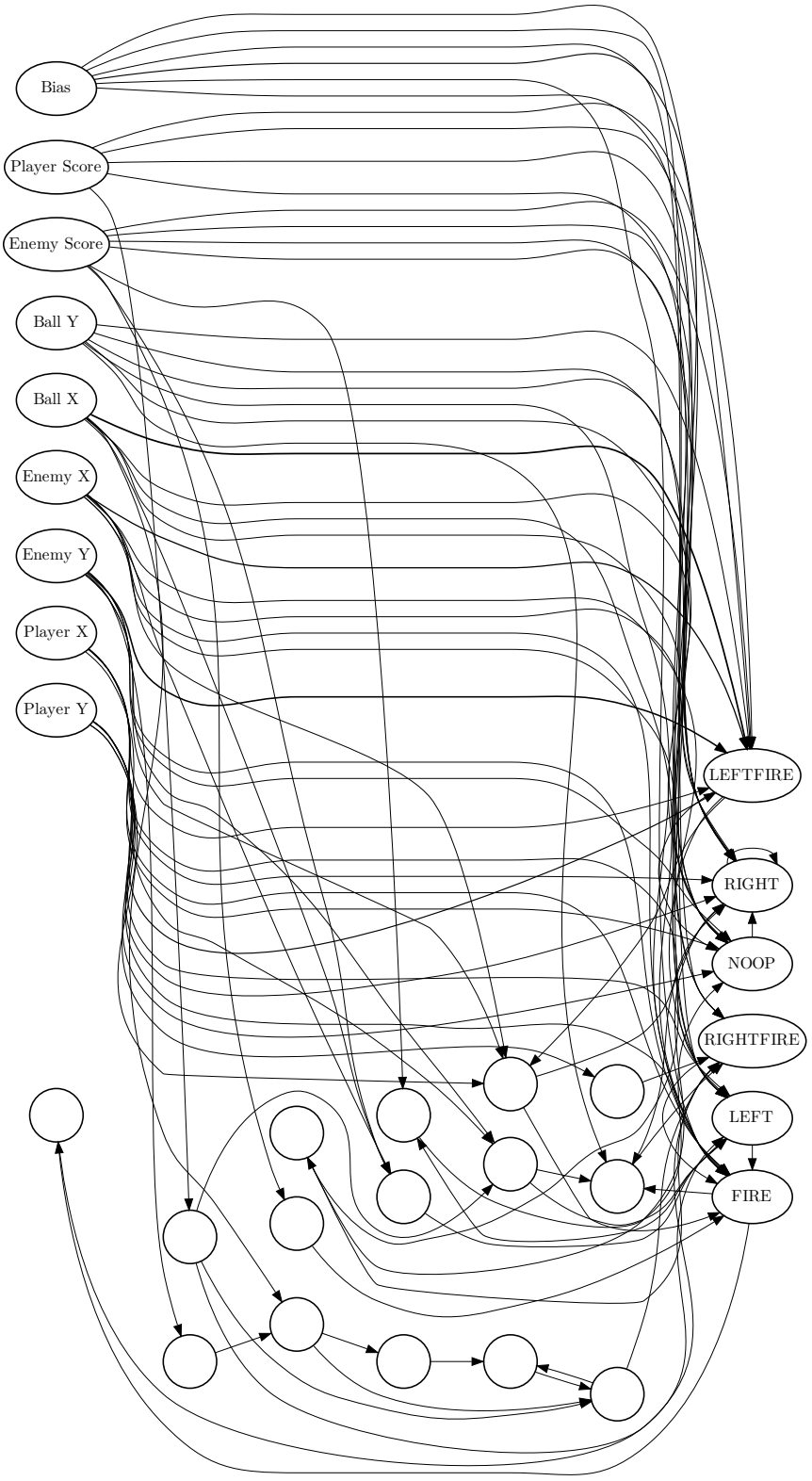


FIGURE B.10: The evolved network architecture for the top performing Pong agent.

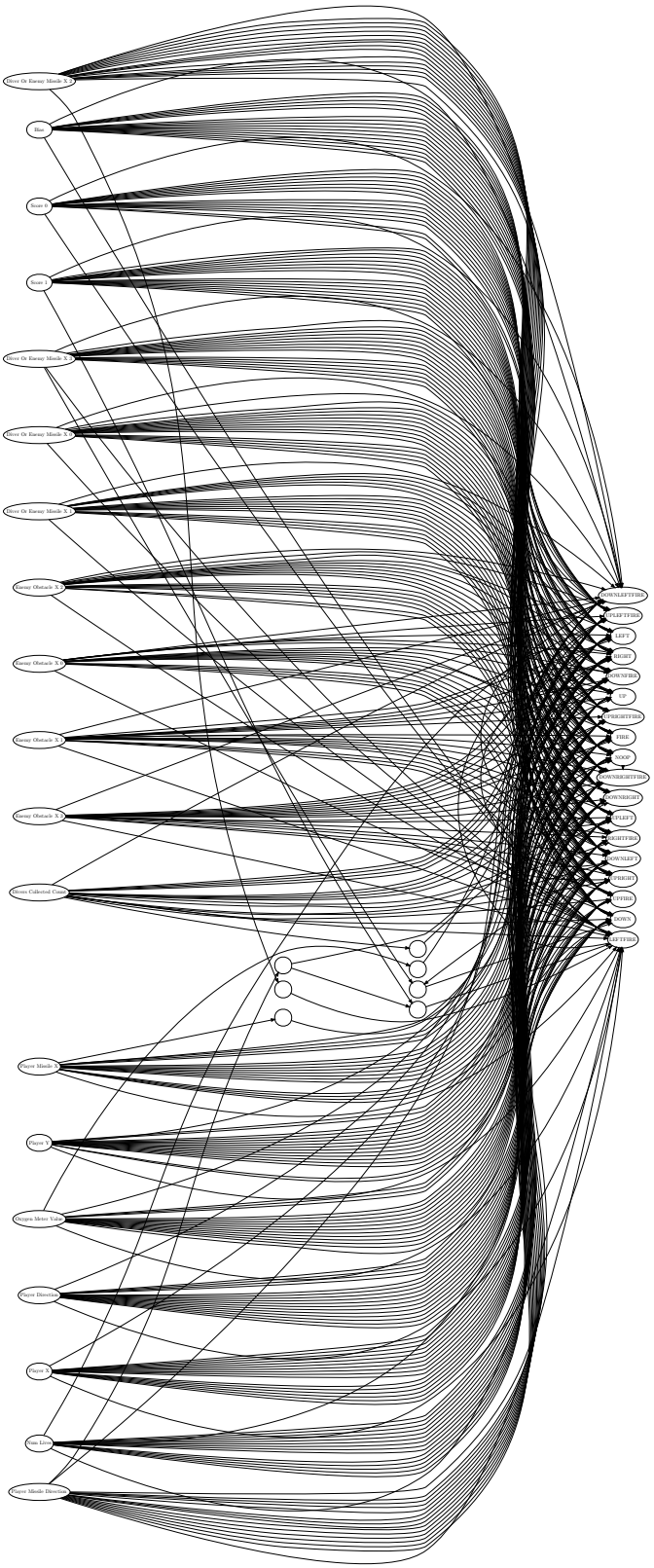


FIGURE B.11: The evolved network architecture for the top performing Seaquest agent.

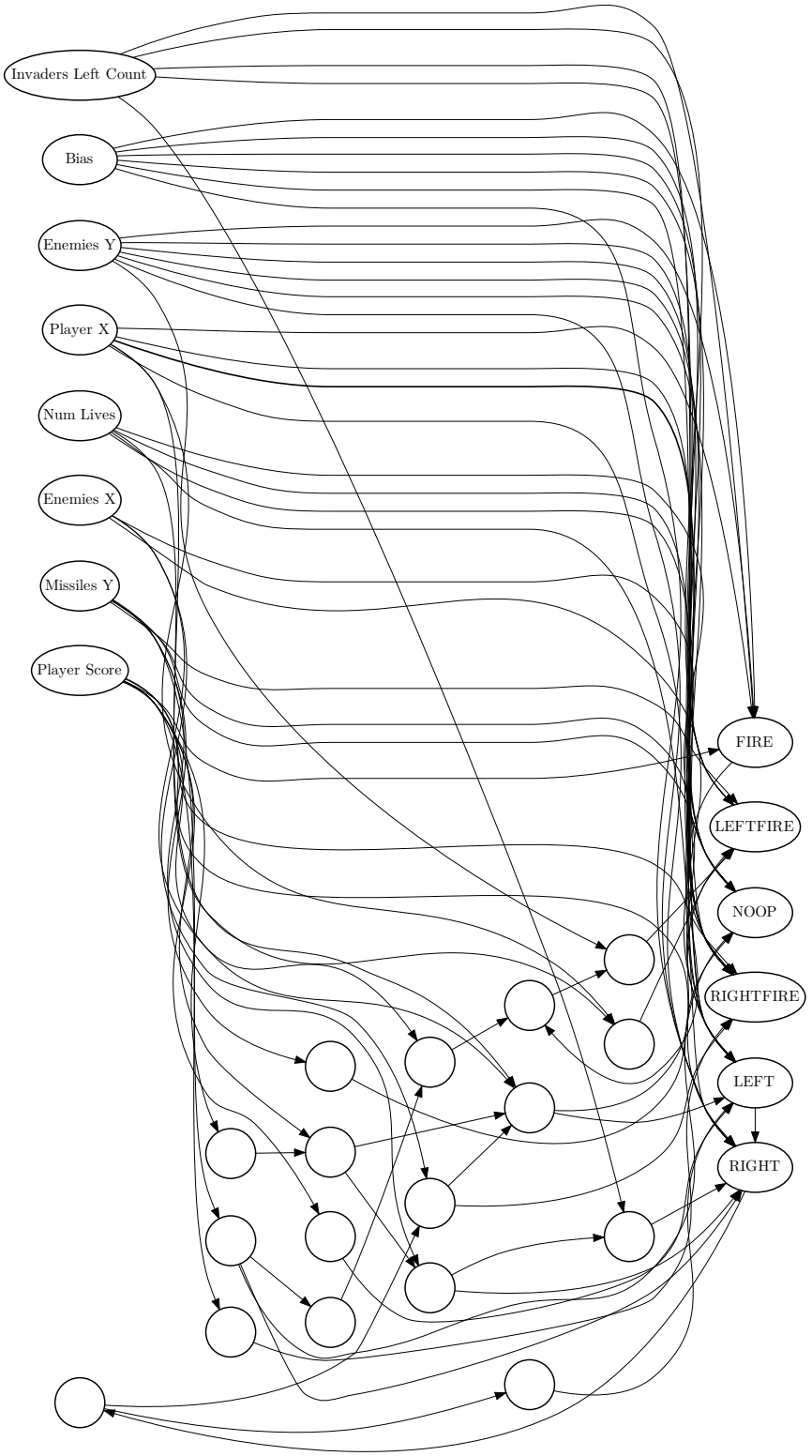


FIGURE B.12: The evolved network architecture for the top performing Space Invaders agent.

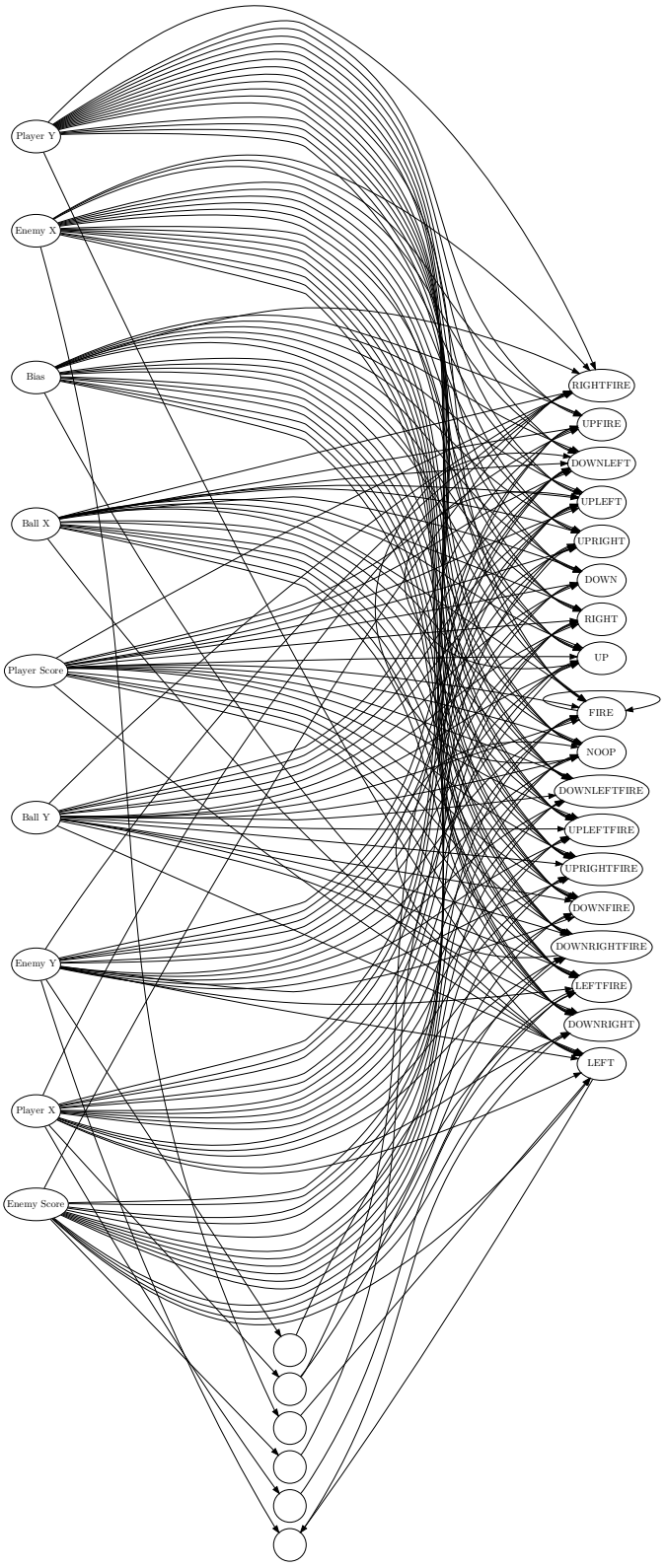


FIGURE B.13: The evolved network architecture for the top performing Tennis agent (using the Tennis B condition).

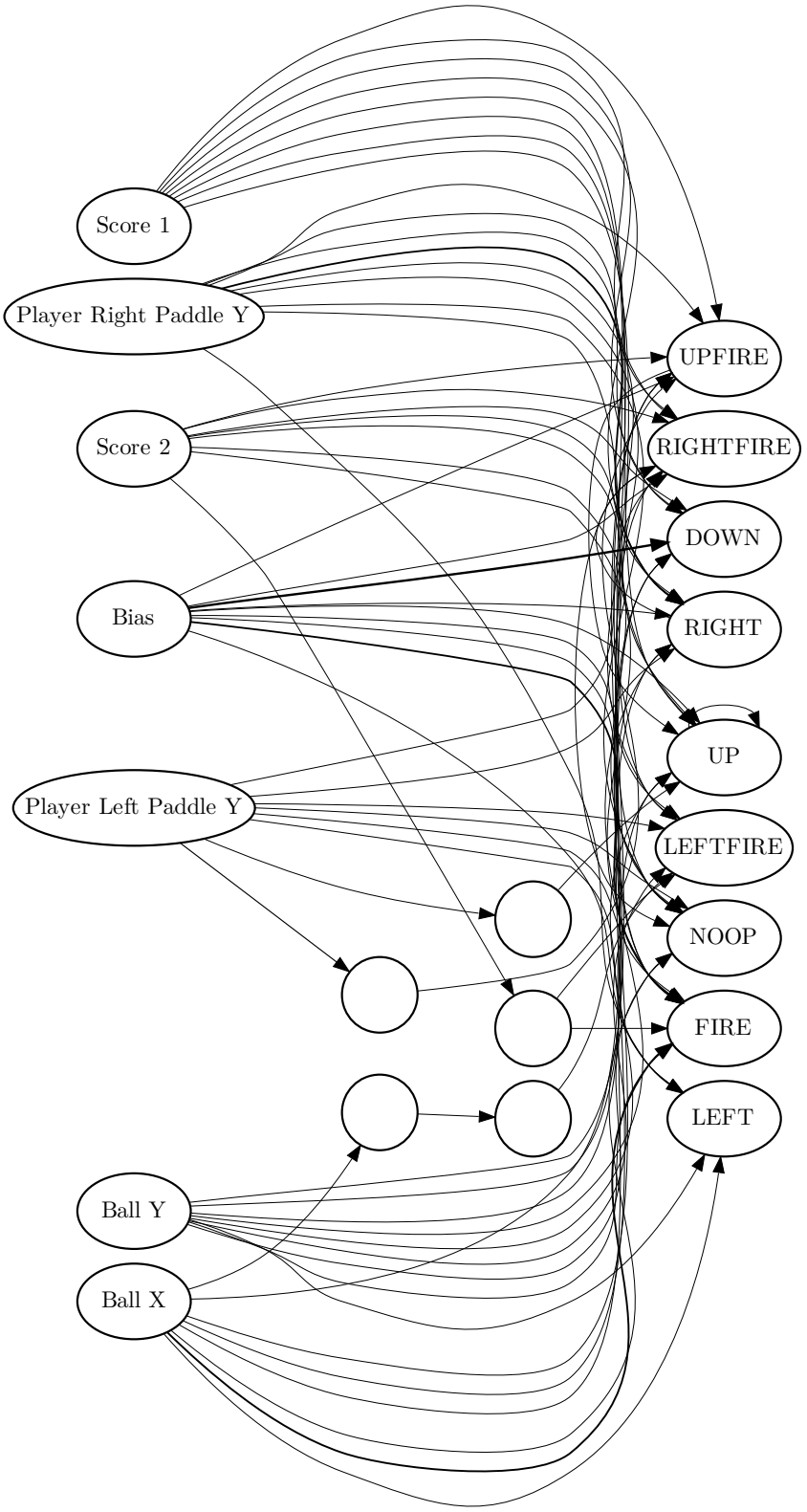


FIGURE B.14: The evolved network architecture for the top performing Video Pinball agent.

Appendix C

Additional Details Related to the AE-NEAT Evaluations

This appendix includes additional details related to the AE-NEAT evaluations presented in Chapter 7.

C.1 Hyperparameters used for the AE-NEAT Evaluations

Table C.1 lists the full set of hyperparameter values used for the AE-NEAT experiments.

TABLE C.1: Common hyperparameters for each game for the AE-NEAT experiments.

Hyperparameter	Value
NEAT	
Generations	200
Population Size	300
Add Node Probability	0.03
Add Connection Probability	0.05
Mutate Weights Probability	0.8
Weight Range	[-30, 30]
Weight Perturb Power	0.05
Weight Init Power	3.0
Weight Replace Probability	0.1
Compatibility Distance Disjoint/Excess Gene Coefficient	1.0
Compatibility Distance Weight Difference Coefficient	0.4
Compatibility Distance Threshold	3.0
Normalise Gene Distance	False
Species Fitness Function	max
Max Stagnation	15
Species Elitism	2
Mutate Only Probability	0.25
Average Crossover Probability	0.4
Crossover Only Probability	0.2
Inter-Species Crossover Probability	0.001
Number of Elites	1
Elitism Threshold	5
Survival Threshold	0.2
Gene Disable Probability	0.75
Initial Connection Probability	1.0
Feed-Forward	False
Activation Function	Sigmoid
Autoencoder	
Learning Rate	$5e^{-4}$
Optimiser	Adam
Training Epochs per Generation	1
Training Sample Size	20,000
Observation Store Size	100,000
Representation Size	40
Architecture	Nature
Type	Undercomplete
Loss Function	Sum of Squared Errors

Bibliography

Ahmed AbuZekry, Ibrahim Sobh, El-Sayed Hemayed, Mayada Hadhoud, and Magda Fayek. Comparative Study of Neuro-Evolution Algorithms in Reinforcement Learning for Self-Driving Cars. In *Proceedings of The 6th International Conference on Innovation in Science and Technology*, pages 6–19, London, United Kingdom, July 2019. Diamond Scientific Publishing.

Samuel Alvernaz and Julian Togelius. Autoencoder-augmented neuroevolution for visual doom playing. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, August 2017. ISBN 2325-4289.

Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R Devon Hjelm. Unsupervised State Representation Learning in Atari. In *Advances in Neural Information Processing Systems 32*, pages 8766–8779, Vancouver, BC, Canada, December 2019. Curran Associates, Inc.

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the Atari Human Benchmark. *arXiv*, abs/2003.13350, March 2020.

Marc Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

Wendelin Böhmer, Jost Tobias Springenberg, Joschka Boedecker, Martin Riedmiller, and Klaus Obermayer. Autonomous Learning of State Representations for Control: An Emerging Field Aims to Autonomously Learn State Representations for Reinforcement Learning Agents from Their Real-World Sensor Observations. *KI - Künstliche Intelligenz*, 29(4):353–362, November 2015. ISSN 1610-1987. doi: 10.1007/s13218-015-0356-1.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv*, abs/1606.01540, June 2016.

Thomas Carr, Maria Chli, and George Vogiatzis. Domain adaptation for reinforcement learning on the atari. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, pages 1859–1861, Richland, SC, May 2019. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-1-4503-6309-9.

Nichael Lynn Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187. L. Erlbaum Associates Inc., July 1985. ISBN 0-8058-0426-9.

Giuseppe Cuccu, Matthew Luciw, Jürgen Schmidhuber, and Faustino Gomez. Intrinsically motivated neuroevolution for vision-based reinforcement learning. In *2011 IEEE International Conference on Development and Learning (ICDL)*, volume 2, pages 1–7, August 2011. ISBN 2161-9476. doi: 10.1109/DEVLRN.2011.6037324.

Charles Darwin. *On the Origin of Species by Means of Natural Selection, or Preservation of Favoured Races in the Struggle for Life*. John Murray, 1859.

Kenneth De Jong. *Evolutionary Computation: A Unified Approach*. MIT press, 2006.

Kalyanmoy Deb. Multi-objective Optimisation Using Evolutionary Algorithms: An Introduction. In *Multi-Objective Evolutionary Optimisation for Product Design and Manufacturing*, pages 3–34. Springer London, London, 2011. ISBN 978-0-85729-652-8. doi: 10.1007/978-0-85729-652-8_1.

Kai Olav Ellefsen, Jean-Baptiste Mouret, and Jeff Clune. Neural Modularity Helps Organisms Evolve to Learn New Skills without Forgetting Old Skills. *PLOS Computational Biology*, 11(4):e1004128, April 2015. doi: 10.1371/journal.pcbi.1004128.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei Rusu, Alexander Pritzel, and Daan Wierstra. PathNet: Evolution Channels Gradient Descent in Super Neural Networks. *arXiv*, abs/1701.08734, January 2017.

Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *arXiv*, abs/1811.12560, November 2018. doi: 10.1561/2200000071.

Shlomo Geva and Joaquin Sitte. A cartpole experiment benchmark for trainable controllers. *IEEE Control Systems Magazine*, 13(5):40–51, October 1993. ISSN 1941-000X.

Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, May 2008.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Geoffrey Gordon. Stable fitted reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pages 1052–1058. MIT Press, December 1996.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv*, abs/1308.0850, August 2013.

David Ha and Jürgen Schmidhuber. Recurrent World Models Facilitate Policy Evolution. In *Advances in Neural Information Processing Systems 31*, pages 2450–2462. Curran Associates, Inc., December 2018.

Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, May 1996. doi: 10.1109/ICEC.1996.542381.

Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, December 2014.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *arXiv*, abs/1502.01852, December 2015.

Jeff Heaton. *Introduction to Neural Networks for Java, 2nd Edition*. Heaton Research, Inc., second edition, 2008. ISBN 1-60439-008-5.

Irina Higgins, Loïc Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. Beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017a.

Irina Higgins, Arka Pal, Andrei A. Rusu, Loïc Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. DARLA: Improving Zero-Shot Transfer in Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 1480–1490, 2017b.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable Baselines. *GitHub repository*, 2018.

Geoffrey Hinton and Ruslan Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006. ISSN 00368075, 10959203.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, January 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90009-T.

Andrej Karpathy. Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/convolutional-networks/, 2015.

Paul Katz and Robert Calin-Jageman. Neuromodulation. In *Encyclopedia of Neuroscience*, pages 497–503. Academic Press, Oxford, January 2009. ISBN 978-0-08-045046-9. doi: 10.1016/B978-008045046-9.01964-1.

Stephen Kelly and Malcolm Heywood. Emergent Tangled Graph Representations for Atari Game Playing Agents. In *Genetic Programming*, pages 64–79. Springer International Publishing, 2017. ISBN 978-3-319-55696-3.

Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. ViZDoom: A doom-based AI research platform for visual reinforcement learning. *arXiv*, abs/1605.02097, May 2016.

Shauharda Khadka and Kagan Tumer. Evolution-Guided Policy Gradient in Reinforcement Learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 1196–1208, Red Hook, NY, USA, 2018. Curran Associates Inc. doi: 10.5555/3326943.3327053.

Daiki Kimura. DAQN: Deep Auto-Encoder and Q-Network. *arXiv*, abs/1806.00630, June 2018.

Diederik Kingma and Max Welling. Auto-encoding variational bayes. *arXiv*, abs/1312.6114, December 2013.

Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez. Evolving Deep Unsupervised Convolutional Networks for Vision-based Reinforcement Learning. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 541–548, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2662-9.

Solomon Kullback. *Information Theory and Statistics*. Courier Corporation, 1997. ISBN 0-486-69684-7.

Yan LeCun, Bernhard Boser, John Denker, Donnie Henderson, Richard Howard, Wayne Hubbard, and Lawrence Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, December 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.4.541.

Joel Lehman, Jay Chen, Jeff Clune, and Kenneth Stanley. Safe Mutations for Deep and Recurrent Neural Networks Through Output Gradients. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 117–124, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5618-3. doi: 10.1145/3205455.3205473.

Timothée Lesort, Natalia Díaz-Rodríguez, Jean-Franǫis Goudou, and David Filliat. State representation learning for control: An overview. *Neural Networks*, 108:379–392, December 2018. ISSN 0893-6080. doi: 10.1016/j.neunet.2018.07.006.

Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv*, abs/1509.02971, September 2015.

Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. NEAT-Python, 2017.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving Deep Neural Networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Academic Press, January 2019. ISBN 978-0-12-815480-9. doi: 10.1016/B978-0-12-815480-9.00015-3.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529, February 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Int. Conf. Mach. Learn., ICML*, volume 4, pages 2850–2869. International Machine Learning Society (IMLS), 2016. ISBN 9781510829008 (ISBN).

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *arXiv*, abs/1712.05889, December 2017.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, and Stig Petersen. Massively Parallel Methods for Deep Reinforcement Learning. *ICML Deep Learning Workshop*, July 2015.

Timothy Nodine. Speciation in NEAT. Undergraduate honors thesis, Department of Computer Science, The University of Texas at Austin, 2010.

Mikkel Nylend. *Data Efficient Deep Reinforcement Learning through Model-Based Intrinsic Motivation*. MSc Thesis, NTNU, 2017.

Yiming Peng, Gang Chen, Harman Singh, and Mengjie Zhang. NEAT for large-scale reinforcement learning through evolutionary feature learning and policy gradient search. In *GECCO '18: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 490–497. ACM, July 2018. ISBN 978-1-4503-5618-3. doi: 10.1145/3205455.3205536.

Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Simon Lucas, and Tom Schaul. General video game AI: Competition, challenges, and opportunities. In *AAAI'16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 4335–4337, February 2016.

Jan Peters. Policy gradient methods. *Scholarpedia*, 5:3698, 2010.

Andrea Poulsen, Mark Thorhauge, Mikkel Funch, and Sebastian Risi. DLNE: A hybridization of deep learning and neuroevolution for visual control. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 256–263, August 2017. ISBN 2325-4289. doi: 10.1109/CIG.2017.8080444.

Ingo Rechenberg. Evolution Strategy: Optimization of Technical systems by means of biological evolution. *Fromman-Holzboog, Stuttgart*, 104:15–16, 1973.

David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986. ISSN 1476-4687. doi: 10.1038/323533a0.

Andrei Rusu, Neil Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv*, abs/1606.04671, June 2016.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv*, abs/1703.03864, March 2017.

Torstein Sandven. *Visual Pretraining for Deep Q-Learning.* MSc Thesis, NTNU, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv*, abs/1707.06347, July 2017.

Dane Sherburn. *On the Feasibility of Using Fully-Convolutional Variational Autoencoders to Advance Deep Symbolic Reinforcement Learning.* MEng Thesis, Imperial College London, 2017.

Andrea Soltoggio, John Bullinaria, Claudio Mattiussi, Peter Dürr, and Dario Floreano. Evolutionary Advantages of Neuromodulated Plasticity in Dynamic, Reward- based Scenarios. In *Proceedings of the 11th International Conference on Artificial Life (Alife XI)*, pages 569–576. MIT Press, 2008.

Kenneth Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, June 2002. ISSN 1063-6560.

Kenneth Stanley, Bobby Bryant, and Risto Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6): 653–668, December 2005.

Kenneth Stanley, David D'Ambrosio, and Jason Gauci. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life*, 15(2):185–212, January 2009. ISSN 1064-5462. doi: 10.1162/artl.2009.15.2.15202.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth Stanley, and Jeff Clune. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv*, abs/1712.06567, December 2017.

Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* MIT press, second edition, 2018. ISBN 0-262-03924-9.

Richard Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, December 2000.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, October 2012. ISBN 2153-0858.

Aaron Tucker, Adam Gleave, and Stuart Russell. Inverse reinforcement learning for video games. In *Proceedings of the Workshop on Deep Reinforcement Learning at NeurIPS*, December 2018.

Andrew Turner and Julian Miller. The Importance of Topology Evolution in NeuroEvolution: A Case Study Using Cartesian Genetic Programming of Artificial Neural Networks. In *Research and Development in Intelligent Systems XXX*, pages 213–226. Springer International Publishing, 2013. ISBN 978-3-319-02621-3.

Borys Tymchenko and Svitlana Antoshchuk. Race from Pixels: Evolving Neural Network Controller for Vision-Based Car Driving. In *Proceedings of the XVIII International Conference on Data Science and Intelligent Analysis of Information*, pages 20–29. Springer International Publishing, June 2019. ISBN 978-3-319-97885-7.

Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100, Phoenix, Arizona, 2016. AAAI Press.

Sjoerd van Steenkiste, Jan Koutník, Kurt Driessens, and Jürgen Schmidhuber. A Wavelet-based Encoding for Neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 517–524, New York, NY, USA, July 2016. ACM. ISBN 978-1-4503-4206-3. doi: 10.1145/2908812.2908905.

Elias Wang, Atli Kosson, and Tong Mu. Deep Action Conditional Neural Network for Frame Prediction in Atari Games. Technical report, Stanford, 2017.

Christopher Watkins. *Learning from Delayed Rewards*. PhD Thesis, King's College, Cambridge, January 1989.

Shimon Whiteson, Peter Stone, Kenneth Stanley, Risto Miikkulainen, and Nate Kohl. Automatic Feature Selection in Neuroevolution. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, pages 1225–1232, Washington DC, USA, July 2005. Association for Computing Machinery. ISBN 1-59593-010-8. doi: 10.1145/1068009.1068210.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(27):949–980, 2014.