6-3-2019 2:00 PM

Algebraic Neural Architecture Representation, Evolutionary Neural Architecture Search, and Novelty Search in Deep Reinforcement Learning

Ethan C. Jackson
The University of Western Ontario

Supervisor
Daley, Mark
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

Abstract

Evolutionary algorithms have recently re-emerged as powerful tools for machine learning and artificial intelligence, especially when combined with advances in deep learning developed over the last decade. In contrast to the use of fixed architectures and rigid learning algorithms, we leveraged the open-endedness of evolutionary algorithms to make both theoretical and methodological contributions to deep reinforcement learning. This thesis explores and develops two major areas at the intersection of evolutionary algorithms and deep reinforcement learning: generative network architectures and behaviour-based optimization. Over three distinct contributions, both theoretical and experimental methods were applied to deliver a novel mathematical framework and experimental method for generative, modular neural network architecture search for reinforcement learning, and a generalized formulation of a behaviour-based optimization framework for reinforcement learning called *novelty search*. Experimental results indicate that both alternative, behaviour-based optimization and neural architecture search can each be used to improve learning in the popular Atari 2600 benchmark compared to DQN — a popular gradient-based method. These results are in-line with related work demonstrating that strictly gradient-free methods are competitive with gradient-based reinforcement learning. These contributions, together with other successful combinations of evolutionary algorithms and deep learning, demonstrate that alternative architectures and learning algorithms to those conventionally used in deep learning should be seriously investigated in an effort to drive progress in artificial intelligence.

Keywords: Artificial neural networks, deep learning, reinforcement learning, algebraic methods, genetic algorithms, novelty search, neural architecture search

Lay Summary

Artificial neural networks (ANNs) have become popular tools for implementing many kinds of machine learning and artificially intelligent systems. While popular, there are many outstanding questions about how ANNs should be structured, and how they should be trained. Of particular interest is the branch of machine learning called reinforcement learning, which focuses on training artificial agents to perform complex, sequential tasks, like playing video games or navigating a maze. In this thesis, three contributions to research at the intersection of ANNs and reinforcement learning are presented. First, a mathematical language that generalizes multiple contemporary ways of describing neural network organization, second, an evolutionary algorithm that uses this mathematical language to help define an algorithm for machine learning with ANNs in which the network's architecture can be modified during training by the algorithm, and third, a related algorithm that experiments with an alternative method to training ANNs for reinforcement learning called novelty search, which promotes behavioural diversity over greedy reward seeking behaviour. Experimental results indicate that evolutionary algorithms, a form of random search guided by evolutionary principles of selection pressure, are competitive alternatives to conventional deep learning algorithms such as error back propagation. Results also show that architectural mutability the ability for network architectures to change automatically during training can dramatically improve learning performance in games over contemporary methods.

Co-Authorship Statement

Chapter 2 was published in the proceedings of the 2017 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology with Mark Daley, James Hughes, and Michael Winter as co-authors. Co-authors were responsible for helping to frame the narrative of the paper as well as examples. Michael Winter in particular contributed to the verification of algebraic methods used in the paper. I was responsible for the large majority of research and writing.

A condensed version of Chapter 3 has been accepted for publication for at the 2019 Genetic and Evolutionary Computation Conference with Mark Daley as a co-author, who provided supervisory advice and direction for the paper. I was responsible for all other research and writing.

Chapter 4 will be submitted for review to a suitable venue later in 2019 with Mark Daley as a co-author, who provided supervisory advice and direction. I was responsible for all other research and writing.

Acknowledgements

I would like to acknowledge the following people and institutions for their part in supporting me throughout my doctoral studies: Dr. Mark Daley, who challenged and enabled me to have the most productive learning experience of my life; The Vector Institute, for supporting me financially and for providing immensely valuable research community connections and computing resources; Dr. James Hughes, for being the friend, colleague, and collaborator who helped me navigate the ups and downs of life as a graduate student; Dr. Jim Staples, for providing guidance and for facilitating interdisciplinary research opportunities; Dr. Jody Culham, for providing incredibly valuable mentorship in teaching and science; My parents, Susan and Frank Hawkins, and Joe Jackson, for unwaveringly supporting me in all aspects of life; Andrew Herring, for helping me to not get completely absorbed by academic life; and my wife Kate for being the most supportive, loving partner I could possibly ask for.

Epigraph

"It was the best of times, it was the BLURST of times?!"

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Artificial neural networks (ANNs) form much of the bedrock of contemporary machine learning. Inspired by computing in the brain, ANNs embody the connectionist approach to cognitive modelling [14]. In contrast to symbolic models of cognition or computation, connectionism does not assume that processes must be expressible using structured symbolic expressions involving explicit operations for storage and retrieval of information. Instead, connectionism models computation using a network of relatively simple computational units that, in isolation of each other, do not necessarily exhibit similar properties as the whole system. Furthermore, memory is an emergent feature of a network's state and parameters, as opposed to being operationally encoded or making use of ad hoc data structures.

The theoretical and practical merits of connectionist and symbolic artificial intelligence were being compared and debated by the late 1980's and early 1990's [32]. Where symbolic systems were being successfully applied to develop highly-structured models including expert systems, neural networks were were often seen as uninterpretable models that learned representations rather than solutions [17]. Despite this being the view of many computational and cognitive scientists, early successes especially in computer vision [24] motivated research on the theory and applications of ANNs continued alongside an explosion of computing resources. This enabled researchers to explore the effectiveness of larger or *deeper* neural architectures.

Deep neural networks consisting of many structured layers of artificial neurons are now being used to solve very difficult problems in a wide range of application areas. The architecture of a deep learning model can be hand-designed to reflect high-level domain knowledge, and this aspect has helped deep learning to become a dominant method for machine learning. By 2012, deep convolutional neural networks (CNNs) had emerged as the new state-of-the-art in image classification [22]. The convolutional layers of CNNs are designed to exploit the geometry of visual input spaces. Successes in computer vision using CNNs attracted the attention of many researchers, and has sparked progress in other areas of computer vision, audio, and natural language processing [23].

In reinforcement learning, characterizable as the application of machine learning to control or decision problems in which artificial agents learn behaviours or policies as a function of environmental observations, deep learning has also had great success. In 2015, Mnih et al. showed that deep convolutional neural networks could be used for reinforcement learning (RL) in an approach called deep Q-learning [29]. The policies yielded by deep Q-learning were some of the first to reach human-level control in the Atari 2600 benchmark [3] when learned directly

from pixels, as opposed to hand-crafted features.

Prior to the emergence of deep Q-learning, neuroevolutionary algorithms such as Neuroevolution of Augmenting Topologies (NEAT) [41] and Hypercube NEAT (HyperNEAT) [39] had also been successfully applied to complex RL problems including the Atari 2600 benchmark [16]. In general, a neuroevolutionary algorithm is the application of an evolutionary algorithm to learning the structure or parameterization of an ANN. Neuroevolutionary algorithms that learn both the topology and weights of an ANN are called topology and weight evolving artificial neural networks (TWEANNs). Recently, researchers have experimented with combinations of tensor-based deep learning and neuroevolutionary algorithms.

For reinforcement learning, Such et al. introduced a method for learning the weights of a deep neural network architecture using a very simple genetic algorithm (GA) [42]. This showed that, given a fixed network architecture, gradient-based learning can be completely substituted by a gradient-free approach and yield often improved performance. Other work has focused on the architecture, rather than parameterizations. In recent work from Google Brain, an evolutionary algorithm was used to evolve the architecture of a deep neural network for image classification that achieves state-of-the-art performance in image classification [33]. The method was also successfully applied in RL contexts. The re-emergence of neuroevolution, in conjunction with deep learning, as a competitive machine learning framework presents many challenges and opportunities.

First, we consider the problem of representation — how an instance of a neural network will be represented as an individual in the evolutionary search, or how as a point in a search space. The most popular contemporary deep learning tools (TensorFlow [1], Keras [8], PyTorch [31], etc.) use graphs of computational units (usually layers) to represent neural networks. Users design a neural network architecture by specifying how various network layers are to be interconnected. Neuroevolutionary methods typically use a different approach, especially when they are used to automatically learn topologies or architectures. NEAT, for example, represents neural networks at the level of individual neurons and connections. HyperNEAT additionally introduces an indirect encoding for ANN topologies using an abstraction called Compositional Pattern Producing Networks (CPPNs) [38], thus representing ANNs differently than in deep learning tools or NEAT. To do research at the intersection of deep learning and neuroevolution, we needed a descriptive language for ANN architectures that generalized the representations used in each of these contexts. This problem is described further in Section 1.1 and a framework designed in response is presented in Chapter 3.

Second, we consider the usefulness of some of the loosened constraints that are enabled by evolutionary frameworks, including non-differentiable objective functions. In RL, researchers are typically interested in developing methods that produce high-quality policies in a given context. In control benchmarks and games, a high-quality policy can be characterized as one that successfully completes a task, or performs well, across many environments. In an Atari game, these environments could be different stages of the same game, the same stage with different initial conditions such as avatar, enemy, or game item positions, or even entirely different games. For benchmarking in such games, policy quality is typically reported using game scores. Similarly, RL methods typically, but do not necessarily, use game score as the reward source. Deep Q-learning optimizes neural network weights by following policy gradients computed in terms of this reward [28], [29]. Evolutionary algorithms can be used to optimize for reward by defining selection pressure or fitness in terms of it. Regardless of which approach

to learning is taken, naïve reward-based optimization often leads to degenerate policies due to reward sparsity or *deceptive local optima*. Novelty Search is an evolutionary framework for addressing these problems and was first applied to deep reinforcement learning in [42]. The application of Novelty Search to RL is described further in Section 1.2 and a generalized application of it to deep neuroevolution for RL is presented in Chapter 4.

Third, we develop a more deeply connected combination of deep learning and neuroevolution in a new approach to TWEANNs. In 2002, Stanley and Miikkulainen introduced NEAT as a state-of-the-art neuroevolution method [41] and successfully applied it to contemporary RL problems [40] such as pole-balancing. NEAT, its many derivatives, and other methods that it inspired, have since been applied to more complex RL problems, such as platform games [43] and control problems related to robotics [39]. More recently, neural architecture search (NAS) has gained popularity in the deep learning community as an alternative to the hand-design of deep learning architectures.

In [30], Negrinho and Gordon introduced *DeepArchitect* — a method for describing the search space of neural network architectures and hyperparameterizations using trees. A number of algorithms including random search, Monte Carlo tree search, and sequential optimizers can be used to construct neural network instances. DEvol [10] is another approach to NAS that uses a simple genetic algorithm to learn sequential models in Keras [8]. More recently, Google Brain developed and applied another neural architecture search method based on an evolutionary algorithm [33]. This method searches a space of constrained architecture-inducing parameters. As of February 2019, this method produces the state-of-the-art ImageNet classification model (top-1 and top-5).

Each of these approaches works at the same level of abstraction: deep learning layers or tensors. In an effort to investigate the effectiveness of NAS using mixed abstractions in an evolutionary algorithm, Chapter 5 introduces a method for mixed neuroevolution that combines deep learning layers and generative graph-based modules for arbitrary neural connectivity.

The remainder of this chapter provides further context and motivation.

1.1 Neural Network Model Description

The research reported in this thesis began with an investigation of general mathematical models suitable for modelling both artificial and biological neural networks. Graph and network theoretic methods have been applied to create quantitative and qualitative measures for high level features of biological neural networks. For example, in [36] Sporns identified small world properties in graph models of primate neural connectivity, and also wrote about the emergence of self-similarity or fractal-like patterns in neural connectivity models. These are characterized by regular patterns occurring at different levels of organization within the brain. The study of network models of the brain has been coined *connectomics*, and a review of graph theoretical analytical methods for studying brain networks is given in [4].

With the rising popularity of studying brain connectivity using mathematical models, it is unsurprising that a variety of software systems for simulating neural network dynamics have been developed. These include NEURON [5], NEST [13], and Brian [15]. Such tools can be used to study how neural architecture and other parameters affect the behaviour of the simulated system as a whole. High-level interfaces for experimenting with various neural simulators have

been developed, including PyNN [9]. Network architectures can be specified for these tools 'by hand' using a neural adjacency representation, or by using other tools for generating larger, more complex networks.

One of the most cited such tools is called Connection Set Algebra (CSA) [11]. CSA is a mathematical framework and software implementation that provides a small set of primitive objects and operations that can be combined to form expressions to generate large connectivity patterns for neural networks. Though certainly useful in some applications, CSA has several flaws. CSA is not truly a formal algebraic framework. Its objects and operations are similar to those described in existing mathematical frameworks, but the connection is not explicitly made nor is it exploited. It also lacks the ability to specify arbitrary connectivity patterns at higher levels of organization. As a result, CSA lacks important operations, is not easily extensible, and does not provide any formal framework for reasoning. This makes CSA unsuitable for use as a general mathematical framework for describing neural network architectures.

1.1.1 Neural Networks and Evolutionary Algorithms

The lack of a standard, general model description language for biological neural network architectures is similarly mirrored in the artificial neural networks community. Though currently dominant, tensors and deep learning layers provide a single level of abstraction for ANN organization. Other popular approaches for connectionist or network-based learning including NEAT [41], HyperNEAT [39], and Cartesian genetic programming [27] represent (neural) architectures each using different levels of abstraction. NEAT operates at the level of individual connections and neurons, while HyperNEAT is much more powerful. HyperNEAT uses a hypercube-based indirect encoding to represent *compositional pattern producing networks* (CPPNs). The motivation for CPPNs stems from the natural relationship between genotype and phenotype, which is necessarily developmental. Stanley cites the relatively small number of genes in the human genome, roughly 30,000, compared to the trillions of connections in the human brain as evidence. CPPNs can be used to define the geometry of the input space for a particular problem. For example, a CPPN can be used to define retinotopy for visual problem spaces by specifying geometric parameters for neuron placement.

CPPNs represent only one possible indirect, developmental encoding for neuroevolution. A recursive description language for modular neural networks was proposed by Jung and Reggia in 2004 [19]. Their descriptive language allows a network to be defined in terms of subnetworks, and the language is purpose-built for manipulation using evolutionary algorithms given its grammatical properties.

1.1.2 Neuromorphic Computing

Model description languages are also used in neuromorphic computing. IBM's TrueNorth architecture, for example, is implemented using interconnected chips of artificial spiking neurons [6]. Programs for TrueNorth are written using the Corelet programming language [2] and are compiled to TrueNorth's hardware-level model description language. One of TrueNorth's strengths is that it is designed to support highly modular and nested programs, while still being implemented using a connectionist model of computation at the hardware level. Other ap-

proaches to neuromorphic computing such as SpiNNaker [20] again use different description languages for expressing neural connectivity.

1.1.3 Generalized Connectionist Models

Considering that, underlying each of these frameworks or systems is a connectionist interpretation of computation, it is perhaps surprising that little work has been done to develop theoretical or practical tools for enabling direct comparison or conversion between them. From a theoretical point of view first, this presents an opportunity to identify or develop a common, generalized abstraction.

In response, we developed a mathematically sound framework as an alternative to CSA based on relation and matrix algebraic methods that has several potential applications. Though they are not used to encode computations directly, the framework enables the description of connectionist architectures using symbolic expressions. In summary, we exploited the abstract connection between relations and matrix algebras [21] to define a very simple framework in which complex, modular network architectures can be described using algebraic expressions. To show that our framework has possible applications for artificial neural networks, we showed that network models written as expressions in the algebraic framework can be translated to concrete models in two popular frameworks for artificial neural computation, namely HyperNEAT [38] and TensorFlow [37]. HyperNEAT is a framework and implementation for neuroevolution, while TensorFlow is a high-level tool for designing and training deep learning architectures. The introductory paper about this framework is presented as Chapter 3.

1.2 Open-Endedness in Machine Learning

Deep learning combines powerful machine learning models and algorithms that can be applied in a wide variety of contexts. In most cases, deep neural networks are used in optimization contexts with a clear objective function. For example, minimize classification error or cross-entropy loss of images, or, maximize the similarity between outputs and reconstructed, compressed inputs. In reinforcement learning, or specifically in Q-learning [46], agent actions are similarly framed as an optimization problem. In games, the reward signal used to characterize the optimization objective is usually provided by the game itself (e.g. score, number of wins, levels cleared, etc.)

Deep Q-learning [28], [29] combines Q-learning with a deep convolutional neural network architecture (DQN) and a policy gradient algorithm. It applies a reward-seeking optimization objective to learning agent policies directly from pixels. This aspect of DQN is in contrast to many prior methods for RL in complex environments, such as games. To be effectively used in RL, evolutionary methods such as NEAT [41] and Cartesian genetic programming [27] require hand-constructed features to be used as inputs rather than raw pixels. This is largely due to their relative unlikeliness to capture important input space regularites that convolutional network layers explicitly assume. HyperNEAT [39] is an evolutionary algorithm that enables input space geometry to be specified and has been successfully applied in game benchmarks [16].

1.2.1 Evolutionary Algorithms

More recently, evolutionary algorithms have seen a resurgence in RL. In [42], Such et al. introduced a very simple, highly scalable genetic algorithm (GA) for learning DQN network weights as an alternative to gradient-based algorithms. Evolutionary algorithms have also been successfully applied to learn tensor-based deep learning architectures [10], [35], [33].

The idea to simulate evolutionary principles using computers can be attributed at least as far back as 1950, when Turing famously proposed the *imitation game* [44]. Evolutionary computation has since become a fruitful method for combinatorial optimization and machine learning with many successes. Genetic algorithms, in particular, have re-emerged as powerful tools when combined with deep learning [42].

Before applying a GA, it is important to consider how a problem's search space will be defined, and how a point in that space can be used. In the case of a deep neural network, the obvious search space might be the space of all possible parameterization vectors. Then, a point in the search space would be one such vector. In practice, this approach is unwieldy for large neural networks, and much more compact, indirect encodings have been developed [42]. In a GA, a point in the search space is represented as an individual in a population. The objective of a GA would then be to evolve a population of individuals until a desirable or useful one is found. In a simple approach, the GA will first generate a population of candidate solutions by randomly generating points in the search space. Next, the algorithm will evaluate the fitness of each individual. This is usually the most computationally intensive step and is entirely problem-specific. Then, using some selection criteria, selected members of the population may be modified (mutated), combined with other individuals (reproduce), or promoted directly to the next generation (elitism). The algorithm continues until some stoppage criteria are met, and the best individuals at each generation are usually stored. In RL, fitness is usually defined as a function of reward, such as the game score in an Atari 2600 game.

1.2.2 Novelty Search

With each of the aforementioned learning methods, a similar approach to reward optimization is usually taken. With DQN, network weights are updated by following the direction of the reward gradient, and in evolutionary frameworks, selection pressure or fitness is usually, but not always, defined in terms of reward. One of the most important differences between these methods is that the standard DQN algorithm requires the objective function to be differentiable in order to define weight updates in terms of gradients. Evolutionary algorithms, on the other hand, can in theory use any computable function to provide fitness scores. As such, evolutionary algorithms can be used as an experimental framework for non-differentiable objective functions in RL. This provides a level of open-endedness in RL policy search that may not be attainable using gradients alone.

A well-known example of this in the evolutionary algorithms community is *novelty search* [25]. Rather than using the reward signal directly, evolutionary fitness is defined in terms of an agent's behaviour. The algorithm then selects agents whose behaviours were most different from previously recorded behaviours, according to a behavioural distance function, and allows them to reproduce. Novelty search has been successfully applied to RL [42], but not in a game-independent context.

Ideas related to novelty search have spawned other successes in RL. Uber AI Labs recently introduced Go-Explore [12] — an exploration-focused algorithm that uses archived environment observations to help train agents to explore their environments more successfully than many previous methods. Go-Explore produced the current state-of-the-art policy for *Montezuma's Revenge* — a game for which conventional reward-seeking learning algorithms generally fail [29], [12].

In Chapter 4, the concept of novelty search in RL is further discussed, and we present a generalized formulation of novelty search that can be applied in diverse RL problems. It introduces the use of string edit metric distances to compute the novelty of agent behaviours in deep RL.

1.3 Modular Neural Networks

Until quite recently, machine learning in domains such as computer vision, natural language processing, speech synthesis, and games has been dominated by deep learning. Artificial neural networks tuned by gradient descent or related strategies have dominated previous approaches — see [23], [26], [45], and [34] respectively. These successes were either due to or in spite of the networks' architectures being hand-crafted by machine learning and domain experts. Evolutionary algorithms have recently re-emerged as viable tools that can complement deep learning in several ways — with neural architecture search (NAS) at the forefront of these efforts.

1.3.1 Neural Architecture Search

Various strategies for NAS or automated architecture design have been explored, and an increasing number of tools and papers on this subject are being released and published. For example, DeepArchitect [30] uses a Monte Carlo tree search-based algorithm to search for optimal architectures, while DEvol [10] uses a very basic genetic algorithm to search for optimal linear arrangements of deep learning layers in Keras [8], a very high-level deep learning library for Python. Recently, work by Google Brain has shown that evolutionary algorithms can be used to automatically design state-of-the-art image classification model architectures [33].

1.3.2 Modular Deep Learning

An essay entitled *The Future of Deep Learning* [7] by François Chollet, author of Keras, asserts that future progress in machine learning will depend heavily on automated model construction. Models will be much more like conventional programs, and will consist of a mix of algorithmic and *geometric* modules such as convolutional layers. Informally, Chollet is hinting that the future of machine learned programs must be preceded by the development of a framework in which trainable deep learning models can be combined with other computational modules, algorithms, and data structures. Chollet claims also that capability for abstraction will come from the use and reuse of computational modules in learned programs. This is an open-ended challenge for researchers in computer science, and it is important to encourage diverse efforts.

This should include those that draw inspiration from the recent discoveries about the organization of biological neural networks.

Many of the building blocks needed to address this challenge already exist. TrueNorth [6] implements a modular, hierarchical programming language for its neuromorphic hardware. The design of this language could inspire the organization of tensors or other deep learning modules. Other modular approaches to neural network description have been proposed in the context of evolutionary algorithms, including Jung and Reggia's recursive language introduced in [19] and Stanley et al.'s compositional pattern producing networks used in HyperNEAT. In deep learning, there are abundantly many, highly-extensible modular frameworks for describing modular neural networks including TensorFlow [1], Keras [8], PyTorch [31], Caffe [18], and more. And due to progress in the development of a generalization for the connectionist models underlying each of these, we can interpret part of the challenge as an invitation to combine these building blocks in meaningful ways.

In response, Chapter 5 introduces a framework for evolutionary deep RL that combines recent advances in deep neuroevolution, deep Q-learning, and *architectural mutability*. Using PyTorch [31], we implemented a framework in which deep learning layers, such as convolutional layers, can be combined with architecturally mutable layers. These mutable layers enable a NEAT-like [41] algorithm to gradually learn both the topology and weights of individual neural network layers in a PyTorch module. A proof of concept is provided using four Atari 2600 games, and results indicate that architectural mutability can dramatically improve learning in certain cases.

Bibliography

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] AMIR, A., DATTA, P., RISK, W. P., CASSIDY, A. S., KUSNITZ, J. A., ESSER, S. K., ANDREOPOULOS, A., WONG, T. M., FLICKNER, M., ALVAREZ-ICAZA, R., AND OTHERS. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on* (2013), IEEE, pp. 1–10.

[3] BROCKMAN, G., CHEUNG, V., PETTERSSON, L., SCHNEIDER, J., SCHULMAN, J., TANG, J., AND ZAREMBA, W. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[4] BULLMORE, E., AND SPORNS, O. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature reviews. Neuroscience 10*, 3 (2009), 186.

[5] CARNEVALE, N. T., AND HINES, M. L. *The NEURON book*. Cambridge University Press, 2006.

[6] CASSIDY, A. S., MEROLLA, P., ARTHUR, J. V., ESSER, S. K., JACKSON, B., ALVAREZ-ICAZA, R., DATTA, P., SAWADA, J., WONG, T. M., FELDMAN, V., AND MODHA, D. Cognitive computing

building block: A versatile and efficient digital neuron model for neurosynaptic cores. *The 2013 International Joint Conference on Neural Networks (IJCNN)* (2013).

[7] Chollet, F. The future of deep learning.

[8] Chollet, F. Keras, 2017.

[9] Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. PyNN: a common interface for neuronal network simulators. *Frontiers in neuroinformatics 2* (2008).

[10] Davison, J. Genetic convnet architecture search with keras, 2017.

[11] Djurfeldt, M. The Connection-set Algebra—A Novel Formalism for the Representation of Connectivity Structure in Neuronal Network Models. *Neuroinformatics 10(3)* (2012).

[12] Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. Go-explore: a new approach for hard-exploration problems. *arXiv:1901.10995* (2019).

[13] Eppler, J., Morrison, A., Diesmann, M., Plesser, H.-E., and Gewaltig, M.-O. Parallel and Distributed Simulation of Large Biological Neural Networks with {NEST}. In *Computational Neuroscience Meeting CNS*06, S48, Edingburgh, UK* (2006).

[14] Fahlman, S. E., and Hinton, G. E. Connectionist architectures for artificial intelligence. *Computer;(United States) 20*, 1 (1987).

[15] Goodman, D. F. M., and Brette, R. The brian simulator. *Frontiers in neuroscience 3*, 2 (2009), 192.

[16] Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation* (2012), ACM, pp. 217–224.

[17] Hinton, G. E. Preface to the special issue on connectionist symbol processing. *Artificial Intelligence 46*, 1-2 (1990), 1–4.

[18] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), ACM, pp. 675–678.

[19] Jung, J.-Y., and Reggia, J. A. A Descriptive Encoding Language for Evolving Modular Neural Networks. In *GECCO* (2004).

[20] Khan, M. M., Lester, D. R., Plana, L. A., Rast, A., Jin, X., Painkras, E., and Furber, S. B. SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)* (jun 2008), pp. 2849–2856.

[21] KILLINGBECK, D., TEIXEIRA, M. S., AND WINTER, M. Relations among Matrices over a Semiring. *Relational and Algebraic Methods in Computer Science (RAMiCS 15)* (2015), Killingbeck, D., Santos Teixeira, M., Winter, M.

[22] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[23] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[24] LECUN, Y., BOSER, B., DENKER, J. S., HENDERSON, D., HOWARD, R. E., HUBBARD, W., AND JACKEL, L. D. Backpropagation applied to handwritten zip code recognition. *Neural computation 1*, 4 (1989), 541–551.

[25] LEHMAN, J., AND STANLEY, K. O. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE* (2008), pp. 329–336.

[26] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[27] MILLER, J. F. Cartesian genetic programming. *Cartesian Genetic Programming* (2011), 17–34.

[28] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[29] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., AND OTHERS. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[30] NEGRINHO, R., AND GORDON, G. DeepArchitect: Automatically Designing and Training Deep Architectures. *arXiv preprint arXiv:1704.08792* (2017).

[31] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[32] PINKER, S., AND MEHLER, J. *Connections and symbols*, vol. 28. Mit Press, 1988.

[33] REAL, E., AGGARWAL, A., HUANG, Y., AND LE, Q. V. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548* (2018).

[34] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., AND OTHERS. Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[35] SO, D. R., LIANG, C., AND LE, Q. V. The evolved transformer. *arXiv preprint arXiv:1901.11117* (2019).

[36] Sporns, O. Small-world connectivity, motif composition, and complexity of fractal neuronal connections. *Biosystems 85*, 1 (2006), 55–64.

[37] Staats, K., Pantridge, E., Cavaglia, M., Milovanov, I., and Aniyan, A. TensorFlow enabled genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2017), ACM, pp. 1872–1879.

[38] Stanley, K. O. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines 8*, 2 (jun 2007), 131–162.

[39] Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life 15*, 2 (2009), 185–212.

[40] Stanley, K. O., and Miikkulainen, R. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation* (2002), Morgan Kaufmann Publishers Inc., pp. 569–577.

[41] Stanley, K. O., and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation 10*, 2 (2002), 99–127.

[42] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv preprint arXiv:1712.06567* (2017).

[43] Togelius, J., Karakovskiy, S., Koutník, J., and Schmidhuber, J. Super mario evolution. In *2009 IEEE symposium on computational intelligence and games* (2009), IEEE, pp. 156–161.

[44] Turing, A. M. Computing machinery and intelligence. *Mind 59*, 236 (1950), 433–460.

[45] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. WaveNet: A Generative Model for Raw Audio. In *Arxiv* (2016).

[46] Watkins, C. J., and Dayan, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.

Chapter 2

Background

This thesis makes use of several concepts, tools, and prior research that warrant additional background.

2.1 Category Theory

Category theory [5] is an abstraction for mathematical structures and associative, compositional transformation operations between them. Category theory is commonly explained as generalizing the relationship between functions and sets, such that different kinds of mathematical structures and operations can be studied under a common framework. In practice, category theory can be used to understand, and to leverage, similarity between different mathematical structures and operations. In this thesis, we leverage the formalized connection between the categories of relations and matrices for the purpose of applying relation algebraic methods to a much more general class of matrices. In other words, the category theoretic connection between relations and matrices allows us to apply our understanding of relations, their associated operations, and reasoning methods, directly to more general matrices — such as the adjacency matrices often used to describe neural network connectivity.

Formally, a category is a mathematical construction that abstractly describes objects and morphisms between objects.

Definition A category C is

1. A collection of objects Ob_C,
2. A collection of morphisms $C[A, B]$, for every pair of objects A and B,
3. An associative, binary composition operation ; which maps morphisms f in $C[A, B]$ and g in $C[B, C]$ to a morphism $f; g$ in $C[A, C]$,
4. An identity morphism denoted by $\mathbb{I}_A$ for all objects A. For all f in $C[A, B]$ and g in $C[B, A]$ we have that $\mathbb{I}_A; f = f$ and $g; \mathbb{I}_A = g$.

In this thesis, we are interested exclusively in the categories **Rel** and **Mat** — the categories of relations and matrices, respectively.

For **Rel**, the objects of the category are sets, and the morphisms are binary relations between sets. Finite relations are very commonly interpreted using matrices. Given particular enumerations of elements in a source set A and a target set B, a matrix with elements belonging to a Boolean structure, such as the Boolean semiring, can be used to denote the presence or absence of a pair $(a \in A, b \in B)$ in a relation.

For **Mat**, the objects of the category are the natural numbers $\mathbb{N}$, and the morphisms are a collection of indexed elements denoting the matrix entries. Note that, at this level of abstraction, we do not specify a structure to which the matrix elements must belong. Practically, however, matrix elements typically belong to a field such as the real numbers $\mathbb{R}$, or to more constrained structures such as the Boolean semiring.

In Chapter 3, the relationship between the categories **Rel** and **Mat** is further explained and leveraged so that relation algebraic operations can be applied to arbitrary matrices satisfying certain constraints. In particular, this work enables relation algebraic methods to be applied to neural network adjacency matrices.

2.2 Artificial Neural Networks

ANNs are ubiquitous in machine learning. They are typically complex networks of biologically-inspired computational units called neurons that are trained with respect to some dataset and objective. ANNs have also been used to learn behaviours given an environment or to learn using underspecified or no objectives [17]. The behaviour of neurons, network representation, and learning algorithms vary greatly between frameworks. Rather than giving a complete history of the field, the rest of this section gives an overview of specific frameworks and methods that are most relevant to this thesis.

2.2.1 Deep Learning and Tensors

The successes of *deep learning* can be best summarized by a 2015 Nature review paper of the same name [10]. Authors LeCun, Bengio, and Hinton give an overview of successful applications of ANNs that feature increasing numbers of internal layers, rather than the previously typical one or two hidden layers. In particular, the effectiveness of convolutional layers for recognizing image features and of recurrent layers for learning sequentially or temporally dependent features is discussed. At the time of publication, backpropagation-based training algorithms for deep convolutional neural networks were delivering state of the art performance in image recognition, classification, and description. In sequentially dependent applications such as speech recognition and word prediction, recurrent neural networks are hailed for delivering state-of-the-art performance.

More recently, significant progress in deep learning has come from the development of new ANN architectures. For example, Google DeepMind's WaveNet [21] is a deep learning model for raw audio generation that is capable of producing near human quality speech audio. The model's architecture is summarized by a small graph of interconnected deep learning layers, and can easily be implemented using high-level model design tools, such as TensorFlow [15], PyTorch [14], or Keras [3].

Each of these tools implements a model description languages based on tensors. This allows a computational model to be defined using multidimensional arrays and compatible operations. This kind of representation is highly compatible with vector-based encodings typically used in machine learning, and with strategies for accelerated parallel linear algebraic computation. Each of these tools can exploit multiple such strategies.

The flexibility that these and similar tools provide is perhaps more interesting than any single neural network architecture. They are simple enough to use that non-experts can design ANN models featuring state-of-the-art layer types and training algorithms very easily. They are also extensible and provide a viable avenue for automated architecture generation — such as with Monte-Carlo tree search [13] or with genetic algorithms [4].

Altogether, tensor-based frameworks provide an invaluable interface for deep learning that abstracts away implementation details about training and optimization algorithms as well as basic linear algebra subroutines.

2.3 Genetic Algorithms

The idea to simulate evolutionary principles using computers can be attributed at least as far back as 1950, when Turing famously proposed the *imitation game* [20]. Evolutionary computation has since become a fruitful method for combinatorial optimization and machine learning with many successes.

A genetic algorithm (GA) is a randomized, population-based algorithm that applies evolutionary principles to explore a search space. Typically, a search space describes the set of all possible or valid solutions to a problem, and in the most simple approaches, each individual in a GA population encodes a single solution. Such simple GAs are often used for combinatorial optimization problems. In this framework, a GA finds solutions by applying selection pressure and genetic operations to traverse subspaces of the search space. Selection pressure, or fitness, is usually defined in terms of an objective function — often to find a globally optimal solution to the given problem. Genetic operations are used to modify the genetic representation of individuals, usually through mutation or recombination, so that new areas of the search space may be evaluated. Together, selection pressure and genetic operations determine how the GA will explore the search space while exploiting the evaluated quality of previously considered solutions.

Genetic algorithms and related methods in the broader field of evolutionary computation have also been applied to machine learning with artificial neural networks.

2.4 Evolutionary Neural Networks

The use of genetic algorithms and, in particular, genetic programming (GP) for evolving neural networks began in the early 1990s with Koza and Rice [8]. GP is an application of genetic algorithms to a search space consisting of valid, executable computer programs. Koza and Rice demonstrated that a population of LISP-like S-expressions could be evolved to generate both the architecture and connection weights of a simple ANN implementing a 1-bit adder. In 1999, Yao showed that contemporary *neuroevolutionary* algorithms that evolved both architecture

and connection weights simultaneously outperformed other strategies [23]. GAs and GP have both contributed significantly to progress in evolutionary machine learning using ANNs. In the early 2000s, this strategy was developed further, and importantly, indirect encodings were introduced.

NEAT and HyperNEAT

Neuroevolution of augmenting topologies (NEAT) [18] and Hypercube-based NEAT (HyperNEAT) [16] each represent significant contributions to neuroevolutionary methods developed by K. O. Stanley. The NEAT algorithm is based on a refined evolutionary strategy that implements *speciation* — a means to encourage diversity in the population. It uses a direct encoding or mapping between genotype and phenotype and prioritizes smaller, simpler network architectures before applying *complexification* to consider larger, more complex networks. HyperNEAT uses a hypercube-based indirect encoding to represent *compositional pattern producing networks* (CPPNs). The motivation for CPPNs stems from the natural relationship between genotype and phenotype, which is necessarily developmental. Stanley cites the relatively small number of genes in the human genome, roughly 30,000, compared to the trillions of connections in the human brain as evidence. CPPNs can be used to define the geometry of the input space for a particular problem. For example, a CPPN can be used to define retinotopy for visual problem spaces by specifying geometric parameters for neuron placement.

2.4.1 Descriptive Encodings for Neural Networks

The inspiration for exploring the effectiveness of indirect representational encodings for ANNs is clear. As Stanley points out, the number of connections in the human brain exceeds the number of genes in the human genome by a factor of roughly 10^8 [16]. While many ANN models and frameworks use an adjacency-based representation (e.g. NEAT [18], CGPANN [7]), many indirect encodings have been researched. For example, HyperNEAT uses the aforementioned hypercube based encoding, and TensorFlow, PyTorch, and Keras allow entire layers to be generated using a very small set of parameters.

In each of the previously mentioned frameworks, network descriptions are limited to connections between computational units at a fixed level of organization or abstraction. They are either organized at the level of individual neurons (NEAT, CGPANN) or at the level of a collection of neurons (e.g. HyperNEAT, TensorFlow, PyTorch, Keras).

In response, recursively-defined neural network modules have been proposed and implemented. A recursive description language for modular neural networks was proposed by Jung and Reggia in 2004 [6]. Their descriptive language allows a network to be defined in terms of subnetworks, and the language is well suited for manipulation using genetic programming. Another example is IBM's Corelet programming language for the TrueNorth architecture [1]. In this framework, networks of spiking neurons may be arbitrarily built up as modules of modules. Both of these descriptive languages provide a strong basis for further research on indirect, modular encodings for ANNs. Chapter 3 aims to capitalize on this progress with the help of algebraic structures.

2.5 Reinforcement Learning

Reinforcement learning (RL) is the application of machine learning to developing *policies* that *agents* apply in response to environment *observations* such that some objective function is optimized. Q-learning [22] is framework for reinforcement learning problems in which the objective is to learn optimal state/action mappings (Q function) over a Markovian decision process. The idealized optimal Q function would yield a policy that, given the current state (observation) and a set of possible actions, the optimal action with respect to the current and all subsequent states is known.

2.5.1 Reinforcement Learning for Games

Video games are commonly used as RL benchmarks as they are a good fit for the learning framework. Observations, agents, and actions are easy to characterize in this context. Observations are most commonly some representation of the visual aspect of the game, such as the colour intensities of pixels. Agents represent the entity or entities playing the game, who make successive observations of the environment, and according to a policy, select actions. An important feature of RL problems is that agent actions affect subsequent observations of the environment. For example, an action may cause an avatar to change positions in the environment. As such, there is temporal dependency between actions and observations.

The goal of RL in the context of a game is to learn a policy that agents can apply to be *successful*. Measures of success are typically the same used for human players: scoring the most points, reaching an end goal, defeating all enemies, etc. For RL to be applied to games, the objective function is usually characterized in terms of reward, such that the choice of action maximizes potential reward. For optimal policies to be learned, algorithms must be able to consider the result of every possible state/action pair, that in all but the simplest games is combinatorially explosive. For this reason, the Q function is commonly approximated.

2.5.2 Deep Q-Learning

In 2013, Mnih et al. introduced a state-of-the-art model and method for Q-Learning called DQN for *deep Q-Learning* [11]. DQN uses a deep convolutional neural network [9] as a model for approximating the Q function in RL benchmarks. Using gradient-based optimization, the DQN model learns to associate a stack of 4 consecutive game observations with predicted state/action pairs. By learning to predict not only actions, but also the effects of actions on subsequent environment states, DQN was one of the first RL models and methods to reach human-level performance in the majority of games in the popular Atari 2600 benchmark, which is part of the OpenAI Gym benchmark collection [12; 2].

2.5.3 Highly Scalable Deep Neuroevolution

In 2017, Such et al. introduced a stricly gradient-free method for learning effective parameterizations of the DQN model for deep RL [19]. The method is a very simple GA that exploits a compact, indirect encoding to enable unprecendented scalability for deep neuroevolutionary algorithms. Instead of encoding the network weights directly, each individual in the GA is a list

of pseudo-random number generator seeds that is decoded into a network instance. This significantly reduces communication bottlenecks, since thousands, rather than millions, of values are used to encode each individual.

Chapters 4 and 5 of this thesis make use of Such et al.'s method for highly scalable deep neuroevolution as both a baseline method for comparison, and as an extensible platform for new methods.

2.6 Summary

The state-of-the-art for ANNs is currently dominated by deep learning models, and is increasingly being advanced by automated search and optimization strategies. Field experts such as Chollet have commented that future progress in machine learning and artificial intelligence using ANNs will rely partly on modularity and the ability to integrate arbitrary computational units. Some descriptive languages for modular neural networks already exist and, with continued development, will be well-suited for next-generation automated architectural search. Furthermore, model description languages for computational neuroscience provide leverageable examples of mathematical tools that have been useful for generating biologically plausible neural networks.

Bibliography

[1] Amir, A., Datta, P., Risk, W. P., Cassidy, A. S., Kusnitz, J. A., Esser, S. K., Andreopoulos, A., Wong, T. M., Flickner, M., Alvarez-Icaza, R., and Others. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on* (2013), IEEE, pp. 1–10.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[3] Chollet, F. Keras, 2017.

[4] Davison, J. Genetic convnet architecture search with keras, 2017.

[5] Freyd, P., and Scedrov, A. *Categories, Allegories*. North-Holland, 1990.

[6] Jung, J.-Y., and Reggia, J. A. A Descriptive Encoding Language for Evolving Modular Neural Networks. In *GECCO* (2004).

[7] Khan, M. M., Ahmad, A. M., Khan, G. M., and Miller, J. F. Fast learning neural networks using Cartesian genetic programming. *Neurocomputing 121* (2013), 274–289.

[8] Koza, J. R., and Rice, J. P. Genetic generation of both the weights and architecture for a neural network. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on* (1991), vol. 2, IEEE, pp. 397–404.

[9] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[10] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[11] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[12] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., AND OTHERS. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[13] NEGRINHO, R., AND GORDON, G. DeepArchitect: Automatically Designing and Training Deep Architectures. *arXiv preprint arXiv:1704.08792* (2017).

[14] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[15] STAATS, K., PANTRIDGE, E., CAVAGLIA, M., MILOVANOV, I., AND ANIYAN, A. TensorFlow enabled genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2017), ACM, pp. 1872–1879.

[16] STANLEY, K. O. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines 8*, 2 (jun 2007), 131–162.

[17] STANLEY, K. O., AND LEHMAN, J. *Why Greatness Cannot Be Planned*. Springer, 2015.

[18] STANLEY, K. O., AND MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. *Evolutionary computation 10*, 2 (2002), 99–127.

[19] SUCH, F. P., MADHAVAN, V., CONTI, E., LEHMAN, J., STANLEY, K. O., AND CLUNE, J. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv preprint arXiv:1712.06567* (2017).

[20] TURING, A. M. Computing machinery and intelligence. *Mind 59*, 236 (1950), 433–460.

[21] VAN DEN OORD, A., DIELEMAN, S., ZEN, H., SIMONYAN, K., VINYALS, O., GRAVES, A., KALCHBRENNER, N., SENIOR, A., AND KAVUKCUOGLU, K. WaveNet: A Generative Model for Raw Audio. In *Arxiv* (2016).

[22] WATKINS, C. J., AND DAYAN, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.

[23] YAO, X. Evolving artificial neural networks. *Proceedings of the IEEE 87*, 9 (1999), 1423–1447.

Chapter 3

An Algebraic Generalization for Graph and Tensor-Based Neural Networks

3.1 Introduction

Algebraic methods provide symbolic representations and simple reasoning tools to a variety of scientific fields. In computer science, algebraic frameworks have been successfully applied in many application domains including computer algebra, high performance computing optimization, computer-assisted reasoning, and more recently in artificial neural networks. In the latter, modern frameworks such as TensorFlow[1] and PyTorch [11] use tensors as important abstractions for making deep learning more accessible. Tensors can be interpreted as multidimensional arrays, and are highly compatible with computational linear algebra. Researchers in computational neuroscience have also appealed to algebraic methods to find representational frameworks for neural networks, for example with Connection Set Algebra (CSA)[3].

More formal applications of algebraic methods in neural network modelling include a description of a classical neural network algebra as a subalgebra of image algebra [12], and a recent compositional framework based on activation functions over an arbitrary ring that formally defines some important operations for both network architecture and computation [2].

While respecting the contributions each of these projects have made to their respective fields, there is still no common formal framework for symbolically representing neural networks or their underlying generative processes [9]. CSA attempts to deliver a formal representational framework for biological neural networks, but it falls short by failing to take advantage of its similarity to existing mathematical frameworks. This makes it difficult to extend using existing mathematics. While the relationship between image algebra and neural networks is interesting and useful, it lacks the generality of a less constrained algebraic formalism. And while we agree that a general, ring-based, compositional framework is well-suited for modelling and computing with neural networks, we suggest that extensibility and compatibility with existing tools are necessary features for adoption.

In response, we worked to find an algebraic generalization for graph and tensor-based representations first for artificial neural networks. In such a framework, one should be able to define the connectivity of a neural network algebraically, either abstractly as an algebraic expression with place-holder terms or concretely using fully defined connectivity matrices. All instances

should be easily portable to any graph or tensor-based format used for machine learning.

Specifically, the focus of this chapter is on exploiting the known relationship between linear and relational algebras to deliver a set of very useful operations for building networks by composition. In particular, we make heavy use of objects called relational sums to define operations for combining matrices of different sizes using a typed notation. In the next section, we give specific details about the relationship between relations, linear algebra, and matrices that are useful in this work. It should be noted that the known relationship between relations and matrices is much more general than what is presented here, but the focus of this work is on practical tools rather than abstraction.

The approach we present could be used as a model for further generalization. With that, the scope of this work is decidedly narrow. We are not making any assertions about the merits of the machine learning approaches used in the systems mentioned, nor are we making claims about the effectiveness of transfer learning. We are providing an example of how algebraic methods can be used to generalize different notations and systems. As results, this will lead to the development of useful, generic tools, and provide a new perspective for studying neural networks as mathematical objects that other work has so far been unable to provide.

3.2 Mathematical Preliminaries

The purpose of this section is to demonstrate that 1) classical relations can be interpreted as the subset of matrices over $\mathbb{R}$ with entries from $\{0, 1\}$, 2) with few additional steps, all elementary relational operations can be defined using elementary matrix operations, and 3) the category theoretic interpretations of relations and matrices both have common *biproducts*, which allow certain relation operations to be applied directly to arbitrary matrices. Altogether, these mathematical preliminaries provide a convenient framework in which neural network connectivity can be described using matrices and manipulated using straightforward linear and relation algebraic operations. Note that the category theoretic aspects of this chapter are presented for completeness and rigour, and that their application can be understood intuitively via the examples in Section 3.3.

3.2.1 Matrix Notation

We use the following notation for matrices and operations. An n by m matrix is denoted by A_{nm} and indexing on A is denoted by $\mathrm{A}[i, j]$ where $1 \leq i \leq n$ and $1 \leq j \leq m$. For arbitrary matrices $\mathrm{A}_{nm}, \mathrm{B}_{nm}$, and C_{mp} we denote the transpose of A by $\mathrm{A}^\intercal$, the matrix product of A and C by $\mathrm{A} \cdot \mathrm{C}$, the component-wise sum of A and B by $\mathrm{A} + \mathrm{B}$, the component-wise difference of A and B by $\mathrm{A} - \mathrm{B}$, the component-wise product of A and B by $\mathrm{A} * \mathrm{B}$, and the scalar product of a scalar α and A by $\alpha * \mathrm{A}$.

3.2.2 Relations

We now give a formal definition of (classical) relations and explain the interpretation of finite relations using matrices.

Definition Formally, a relation R between a source set A and a target set B is a subset of ordered pairs from $A \times B$, i.e. $R \subseteq A \times B$. For $x \in A$ and $y \in B$ we say that x is in relation to y via R if and only if $(x, y) \in R$.

We denote a relation R with source A and target B by $R : A \to B$ or $R_{A \to B}$. Finite relations can be interpreted by $\{0, 1\}$-valued matrices. These are called relational matrices, or just relations, henceforth. In this interpretation, the source elements are enumerated along the rows and the target elements along the columns. We use A_i to denote the i^{th} element of A according to the same enumeration. The matrix interpretation $\mathrm{R}_{|A||B|}$ of R is defined by $\mathrm{R}[i, j] = 1 \iff (a_i, b_j) \in R$ and $\mathrm{R}[i, j] = 0 \iff (a_i, b_j) \notin R$. Since we are only concerned with finite relations, every relation $R : A \to B$ can be interpreted as a matrix $\mathrm{R}_{|A||B|}$, though we use relational notation wherever possible.

3.2.3 Relational Operations

Next we define a set of operations allowing us to interpret any matrix over $\mathbb{R}$ as a relation. To do this, we must first define element-wise operations mapping arbitrary elements of $\mathbb{R}$ to the relational coefficients $\{0, 1\}$.

Definition For all $x \in \mathbb{R}$, the flattening operation is defined by

$$x' := \begin{cases} 1 & \iff x \neq 0 \\ 0 & \iff x = 0. \end{cases}$$

Other maps from real-valued to relational coefficients can be defined, but the flattening operation is fundamental among those operations due to properties of elementary semirings used to generalize relations [8]. Next we need element-wise Boolean operations.

Definition For all $x, y \in \mathbb{R}$ we define join, meet, and negation respectively by

$$x \sqcup y := (x' + y') - (x' * y') \qquad x \sqcap y := (x * y)' \qquad \neg x := 1 - x'$$

With these we can now define fundamental relations and operations. Note that the relational operations are defined for all matrices. In other words, the relational operations are matrix operations that always produce $\{0, 1\}$-valued matrices and satisfy the axioms of a relation algebra. For more details we refer again to [8].

Definition Given a source set A, target set B, and matrices $R_{|A||B|}$, $S_{|A||B|}$, and $T_{|B||C|}$ we define the following operations and relations

$$(R \cup S)_{A \to B}[i, j] := R[i, j] \sqcup S[i, j] \qquad \text{(union)}$$

$$(R \cap S)_{A \to B}[i, j] := R[i, j] \sqcap S[i, j] \qquad \text{(intersection)}$$

$$(R; T)_{A \to C}[i, k] := \bigsqcup_{j \in B} (R[i, j] * T[j, k]) \qquad \text{(composition)}$$

$$\mathbb{I}_{A \to A} := \mathrm{Id}_{|A||A|} \qquad \text{(identity relation)}$$

$$R^{\smile}_{B\to A} := R^{\intercal}; \mathbb{I}_{A\to A} \qquad \text{(converse)}$$

$$\bar{R}_{A\to B}[i,j] := \neg(R[i,j]) \qquad \text{(complement)}$$

$$(R - S)[i,j] := R[i,j] * \neg S[i,j] \qquad \text{(difference)}$$

$$\perp\!\!\!\perp_{A\to B}[i,j] := 0 \qquad \text{(null relation)}$$

$$\top\!\!\!\top_{A\to B}[i,j] := 1 \qquad \text{(universal relation)}$$

In the definition of converse, note that composition with the identity relation effectively applies the flattening operation so that a relation, as opposed to a matrix, is produced. A detailed account of the algebraic properties satisfied by these operations can be found in [13], a standard text on relational mathematics. An abundance of additional operations compatible with this framework can also be found therein. Altogether, the set of operations presented in this section forms the mathematical foundation that allows us to work with relations as a special kind of matrix.

3.2.4 Relational Sums

With the elementary relational operations now defined, we now look to operations for building algebraic descriptions of networks by composition. Of particular importance is the ability to, based on type, keep track the set to which a node in such a network belongs. Luckily, such operations already exist and have been extensively studied in category theory. For more details, see [5].

Let **Mat** denote the category of all matrices, and let **Rel** denote the subclass of relations among those matrices. Since **Mat** is an Abelian category it has biproducts. For matrices, biproducts are defined using special matrices ι, κ, π and ρ together with matrix multiplication and component-wise addition. Biproducts in **Mat** are defined as follows.

Definition Let A and B be sets. Then biproducts in ***Mat*** consist of a set $A \oplus B$ together with matrices $\iota_{|A||A\oplus B|}$, $\kappa_{|B||A\oplus B|}$, $\pi_{|A\oplus B||A|}$ and $\rho_{|A\oplus B||B|}$ such that

$$\iota \cdot \pi = \mathrm{Id}_{|A|} \quad \kappa \cdot \rho = \mathrm{Id}_{|B|} \quad \kappa \cdot \pi = 0_{|B||A|}$$
$$\pi \cdot \iota + \rho \cdot \kappa = \mathrm{Id}_{|A\oplus B|} \quad \iota \cdot \rho = 0_{|A||B|}$$

Due to the inclusion and properties of the relational operations converse, intersection, and union, the subclass **Rel** forms a distributive allegory [5]. In **Rel**, the relational sum can be defined in terms of ι and κ together with the relational operations. It was then shown in [8] that the relational sum is also a biproduct in **Mat**. As a result, the relational sum can be applied directly to combine arbitrary matrices or relations. Notice that the definitions for matrix biproducts and relational sums are very similar but use different operations.

Definition Let A and B be sets. Then the relational sum of A and B is a set $A \oplus B$ together with injection relations $\iota : A \to A \oplus B$ and $\kappa : B \to A \oplus B$ such that

$$\iota; \iota^{\smile} = \mathbb{I}_A \qquad \kappa; \kappa^{\smile} = \mathbb{I}_B$$
$$\kappa; \iota^{\smile} = \perp\!\!\!\perp_{B\to A} \qquad \iota^{\smile}; \iota \cup \kappa^{\smile}; \kappa = \mathbb{I}_{A\oplus B}$$

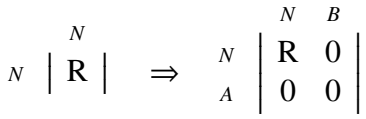

Figure 3.1: Matrix interpretations of R (left) and $iTL(\mathrm{R}, A, B)$ (right) where R is an $|N|$ by $|N|$ matrix and A, B are finite sets.

The correspondence between the matrix and relational versions of biproducts effectively provides us with operations that can be used build neural adjacency matrices by composed expressions that include both arbitrary matrices and special relational matrices. Informally, these operations are very useful for building algebraic expressions for networks with nodes belonging to different sets. To make it even easier to build such expressions, we denote $\iota^\smile$ by π and $\kappa^\smile$ by ρ, and we define four additional convenience functions as follows.

Definition Let A, B, and N be sets, and R be an $|N|$ by $|N|$ matrix. The injection functions iTL, iTR, iBL, iBR are defined by

$$
\begin{aligned}
iTL(R, A, B)_{|N\oplus A||N\oplus B|} &:= \pi_{N\oplus A\to N} \cdot R \cdot \iota_{N\to N\oplus B} \\
iTR(R, A, B)_{|N\oplus A||B\oplus N|} &:= \pi_{N\oplus A\to N} \cdot R \cdot \kappa_{N\to B\oplus N} \\
iBL(R, A, B)_{|A\oplus N||N\oplus B|} &:= \rho_{A\oplus N\to N} \cdot R \cdot \iota_{N\to N\oplus B} \\
iBR(R, A, B)_{|A\oplus N||B\oplus N|} &:= \rho_{A\oplus N\to N} \cdot R \cdot \kappa_{N\to B\oplus N}
\end{aligned}
$$

Informally, these injection functions enable 'gluing matrices together' in such a way that the intended network connections can be established or retained. For example, $iTL(\mathrm{R}, A, B)$ can be read as: 'inject R into the top-left of a matrix with $|A|$ rows and $|B|$ columns added'. This is visualized by Figure 3.1. Note that if A or B are the empty set, the operation behaves as expected and adds zero rows or columns, respectively.

Altogether, the core algebraic framework presented in this chapter is an extension of linear algebra that includes relational matrices, relational operations, and relational sums. And since each of these can be defined strictly in terms of linear algebraic operations, the framework can be implemented as an extension of any software package for linear algebra. For the remainder of this chapter, we assert that relations and real-valued matrices may be used interchangeably in expressions, and apply this in situations where relational notation is more convenient.

3.3 Extended Algebraic Operations

In this section we introduce three new algebraic operations as extensions of the basic framework outlined in Section 3.2. For each operation, we give an overview with example and an algebraic formulation with properties. The details in this section are meant to demonstrate how new operations can be designed using matrices, defined as algebraic expressions, and how important properties can be proven using algebraic reasoning.

a)

$$In := \begin{array}{c|cc} & H_1 & H_2 \\ \hline A_1 & 1 & 0 \\ A_2 & 0 & 1 \\ A_3 & 1 & 1 \end{array} \qquad Hid := \begin{array}{c|cc} & H_1 & H_2 \\ \hline H_1 & 1 & 1 \\ H_2 & 1 & 1 \end{array}$$

$$Out := \begin{array}{c|c} & B_1 \\ \hline H_1 & 1 \\ H_2 & 1 \end{array}$$

b)

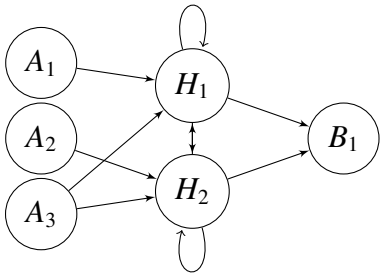

Figure 3.2: a) Relations $In : A \to H$, $Hid : H \to H$, and $Out : H \to B$ defined by matrices. b) A graph visualizing the connected relations.

3.3.1 Connect

The first extension is an operation to connect three subnetworks. Though we generalize *connect* later with the *Total Network Matrix*, this serves as an example of how simple operations can be designed using an algebraic approach. To keep the presentation and discussion simple, we will only use relational notation in this section. Note that real-valued matrices could be used interchangeably.

This simple operation takes three relations $In : A \to H$, $Hid : H \to H$, and $Out : H \to B$ and produces a relation $A \oplus H \oplus B \to A \oplus H \oplus B$ such that all connectivity is preserved. The main application for this operation is to modify input or output connectivity without affecting the hidden layers.

Example

Suppose we have three relations $In : A \to H$, $Hid : H \to H$, and $Out : H \to B$ defined by the matrices in Figure 3.2a. A graph interpretation of their connection is shown in Figure 3.2b.

Before defining the algebraic formulation of this operation, consider the intended result in matrix form, as visualized in Figure 3.3. We can define the *connect* operation by observing the positions of *In*, *Hid*, and *Out* in a connectivity matrix $A \oplus H \oplus B \to A \oplus H \oplus B$.

Algebraic Formulation

The algebraic formulation of the *connect* function is given by Definition 3.3.1.

	A_1	A_2	A_3	H_1	H_2	B_1
A_1				1	0	
A_2				0	1	
A_3				1	1	
H_1				1	1	1
H_2				1	1	1
B_1						

Figure 3.3: The result of applying *connect* to relations *In*, *Hid*, and *Out*. The position of coefficients in this matrix identify the source and target sets to which they originally belonged.

Definition Let A, B, and H be finite sets and $In : A \to H$, $Hid : H \to H$, and $Out : H \to B$ be relations. The connect operation is defined by:

$$\begin{aligned} &connect(In, Hid, Out) := \\ &\quad iTL(iTR(In, H, A) \ + \ iBR(Hid, A, A), B, B) + \\ &\quad iBR(iTR(Out, B, A \oplus H), A, \emptyset) \end{aligned}$$

It is easy to check that the type of relation produced by *connect* is indeed $A \oplus H \oplus B \to A \oplus H \oplus B$. Preservation of connectivity follows from a more general property (Lemma 3.3.2) and so its discussion is omitted here.

3.3.2 Substitution

The substitution operation (*subst*) enables the substitution of one connection by an arbitrary connectivity pattern. In connectomics, self-similar or fractal-like organization has been observed at multiple levels of neural organization. For example, the structure of cortical columns in primate brains is reported to be highly regular and self-similar ([14], [15]). Like *connect*, *subst* is defined using relational notation, though real-valued matrices could be used interchangeably.

Example

Suppose we have a very small network with just three nodes, and that this network describes the high-level organization of a neural network. We can use *subst* to replace individual connections by more detailed connectivity patterns. Let $Net : N \to N$ and $S : C \to C$ be relations where $N = \{N_1, N_2, N_3\}$ and $C = \{C_1, \ldots, C_5\}$ as defined by the graphs in Figure 3.4a and 3.4b, respectively.

In this example, the result of using *subst* to replace each connection in *Net* by the pattern in S is visualized by Figure 3.4c. Since the number of nodes in the network is increasing, the corresponding relation must be expanded by this operation. Where *Net* is a relation $N \to N$, the network in Figure 3.4c can be interpreted as a relation $N \oplus S \oplus S \oplus S \to N \oplus S \oplus S \oplus S$.

To see how new nodes and connections are added by the substitution operation, consider first the expansion via substitution from $Net : N \to N$ to $NetS : N \oplus S \to N \oplus S$ as visualized by Figure 3.5.

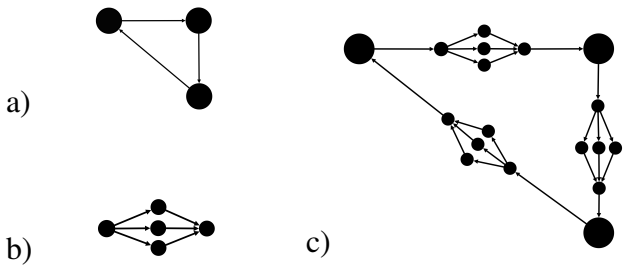

Figure 3.4: Substitution of connections by a repeating pattern. a) A high-level network *Net*. b) A substitution connectivity pattern S. c) The result of substituting each connection in *Net* by S. Notice that *Net* is necessarily a *graph minor* of this graph.

	N_1	N_2	N_3	C_1	C_2	C_3	C_4	C_5
N_1		~~1~~		$\underline{1}$				
N_2			1					
N_3	1							
C_1					1	1	1	
C_2								
C_3								1
C_4								1
C_5		$\underline{1}$						1

Figure 3.5: Replacing the connection (N_1, N_2) in *Net* by the relation S. The crossed out entry indicates removal; underlined entries indicate new connections between nodes in N and C. Each quadrant in the matrix represents connectivity between different sets, as indicated by the row and column labels.

The relation *NetS* can be interpreted as a matrix with four submatrices, each describing a different part of the corresponding network. In the top left are the remaining connections from the original network *Net*. In the top right is a relation $N \to C$ describing new connections between the original nodes in N and the newly added nodes in C. In the bottom left is a similar relation $C \to N$ describing connections between the newly added nodes in C and the original nodes in N. Finally in the bottom right is the relation $S : C \to C$ describing the substituted connectivity pattern.

Algebraic Formulation

The general algebraic formulation of the substitution operation is given by Definition 3.3.2.

Definition Let N and C be finite sets, $Net : N \to N$ and $S : C \to C$ be relations, and $(N_x, N_y) \in Net$. The substitution operation is defined by:

$$\begin{aligned} &subst(Net, S, (N_x, N_y)) := \\ &\quad iTL(Net - \{(N_x, N_y)\}, C, C) \cup \\ &\quad iBR(S, N, N) \cup \{(N_x, C_1), (C_n, N_y)\} \end{aligned}$$

The usefulness of this algebraic operation depends on its correctness. As an example, we may wish to prove that this operation preserves connectivity after substitutions. Interpreted as graphs, the substitution of a connection (N_x, N_y) by an arbitrary connectivity pattern should always preserve the existence of a path between N_x and N_y in the resulting graph. In strictly relational terms, we can express this more formally.

Lemma 3.3.1 *Let N and C be finite sets, $Net : N \to N$ and $S : C \to C$ be relations, and (N_x, N_y) be an element in Net. If $(C_1, C_n) \in S^+$ then $(N_x, N_y) \in NetS^+$, the transitive closure of $NetS$, where $NetS = subst(Net, S, (N_x, N_y))$.*

Proof $(C_1, C_n) \in S^+$ implies that $(C_1, C_n) \in NetS^+$, since $S^+ \subseteq iBR(S, N, N)^+ \subseteq NetS^+$. Then since $\{(N_x, C_1), (C_n, N_y)\} \subseteq NetS$ (by union), we have that $\{(N_x, C_1), (C_n, N_y)\} \subseteq NetS^+$. Finally since $\{(N_x, C_1), (C_1, C_n), (C_n, N_y)\} \subseteq NetS^+$, we have that $(N_x, N_y) \in NetS^+$ by transitivity.

Informally, if the first and last elements of C are transitively connected in the relation S, then the substitution operation with S applied to any connection $(N_x, N_y) \in Net$ will result in (N_x, N_y) being included in the transitive closure of the resulting relation. In other words, this operation preserves connectivity in networks.

3.3.3 Total Network Matrix

A currently popular approach for working with artificial neural networks involves 'hand crafting' network architectures such as convolutional neural networks. The *Total Network Matrix (TNM)* operation generalizes the *connect* operation and allows any combination of connectivity matrices over finite, non-intersecting sets of neurons to be combined into a single matrix. For this operation we proceed first with the algebraic formulation. A detailed example of its use is given throughout Section 3.5.

Algebraic Formulation

The algebraic formulation of *TNM* requires a map between source and target sets and a corresponding connectivity matrix. Given $S = \{S_1, S_2, \ldots S_n\}$, a set of sets, let $S^\oplus$ denote the set $S_1 \oplus S_2 \oplus \cdots \oplus S_n$.

Definition Let $S = \{S_1, S_2, \ldots, S_n\}$ be a finite set where each $S_i \in S$ is a finite set of neurons such that for all $S_x, S_y \in S, S_x \cap S_y = \emptyset$. The enumeration of elements in S is assumed to be $S_1, S_2, \ldots, S_{n-1}, S_n$ consistently. Then let $\mathcal{M} : (S_x, S_y) \in S \times S \Rightarrow M_{|S_x||S_y|}$ be map between pairs of sets of neurons and a corresponding connectivity matrix. Then the Total Network Matrix is defined by:

$$\mathrm{TNM}(S, \mathcal{M}) := \sum_{(S_x, S_y) \in S \times S} \rho \cdot \pi \cdot \mathcal{M}((S_x, S_y)) \cdot \iota \cdot \kappa$$

$$\rho : S^\oplus \to S_x \oplus S_{x+1} \oplus \cdots \oplus S_n$$

$$\pi : S_x \oplus S_{x+1} \oplus \cdots \oplus S_n \to S_x$$

$$\iota : S_y \to S_y \oplus S_{y+1} \oplus \cdots \oplus S_n$$

$$\kappa : S_y \oplus S_{y+1} \oplus \cdots \oplus S_n \to S^\oplus$$

It is easy to check that the *TNM* operation always produces a connectivity matrix with the correct dimensions. This is determined by the source and target sets used in the relations ρ, π, ι, and κ, which are fixed. Similar to *subst*, it is perhaps most important to prove that this operation also preserves connectivity.

Using the relational interpretation of matrices makes such a proof more straightforward and no less general because of the correspondence between relational and matrix biproducts in the case of matrices over $\mathbb{R}$. Informally, we need to show that pairs of neurons that are connected according to a connectivity matrix are still connected after *TNM* is applied to a map containing that matrix. A formal proof is given in Lemma 3.3.2.

Lemma 3.3.2 *For all* $(S_x, S_y) \in S \times S$ *and* $\mathcal{M} : (S_x, S_y) \Rightarrow \mathrm{M}_{|S_x||S_y|}$, $(S_{x_a}, S_{y_b}) \in \mathcal{M}(S_x, S_y); \mathbb{I}_{S_y} \longrightarrow (S_{x_a}, S_{y_b}) \in \mathrm{TNM}(S, \mathcal{M}); \mathbb{I}_{S^\oplus}$.

Proof For any $R : X \to Y$, $R \subseteq (\rho; \pi; R; \iota; \kappa); \mathbb{I}$ with $\rho_{A\oplus B\oplus C \to B\oplus C}$, $\pi_{B\oplus C \to X}$, $\iota_{Y \to B\oplus C}$, $\kappa_{B\oplus C \to A\oplus B\oplus C}$, and $\mathbb{I}_{A\oplus B\oplus C}$. This follows from the definition of relational sums. Then because $\rho; \pi; R; \iota; \kappa = \rho \cdot \pi \cdot R \cdot \iota \cdot \kappa; \mathbb{I}$ we have that $R \subseteq \bigcup_{R_x \in \mathcal{R}} (\rho \cdot \pi \cdot R_x \cdot \iota \cdot \kappa; \mathbb{I})$ where R_x denotes an arbitrary relation from a set of relations to which R also belongs, denoted by $\mathcal{R}$. Then because of the abstract correspondence between union and sum for relations and matrices, respectively, we have that $R \subseteq (\sum_{R_x \in \mathcal{R}} \rho \cdot \pi \cdot R_x \cdot \iota \cdot \kappa); \mathbb{I}$. Finally, since $(u, v) \in R \longrightarrow (u, v) \in (\sum_{R_x \in \mathcal{R}} \rho \cdot \pi \cdot R_x \cdot \iota \cdot \kappa); \mathbb{I}$ by the definition of relational inclusion, we can conclude that indeed $(S_{x_a}, S_{y_b}) \in \mathcal{M}(S_x, S_y); \mathbb{I}_{S_y} \longrightarrow (S_{x_a}, S_{y_b}) \in \mathit{TNM}(S, \mathcal{M}); \mathbb{I}_{S^\oplus}$.

In other words, it follows from the definition of relational sums that no connections are lost when these operations are applied in additive expressions using union or sum.

In summary, this section describes the exploitation of relational sums to define useful new operations that are fully compatible with linear algebra and with existing proof techniques for sets, relations, and matrices. As a result, the framework and operations presented here are demonstrated to be extensible, flexible, and formal. All of these points are in sharp contrast to similar work, such as CSA.

3.4 Implementation

The version of our framework presented in this chapter is fully compatible with linear algebra, by design. As a result, we can exploit any existing software for linear algebra to serve as the foundation for an implementation. In an effort to make this framework widely accessible, we used the Python packages NumPy[10] and SciPy[7] to provide such a foundation. From there we implemented the relational sums, injection functions, and each of the extended operations introduced in Section 3.3 as sparse matrix operations.

The Python implementations of operations follow almost directly from their algebraic counterparts. Compare for example the Python listing in Figure 3.6 to Definition 3.3.3. Implementations of all operations presented in this chapter are included in a digital appendix found at **`https://github.com/ethancjackson/SparseNALG-0.1/`**. Collectively, the Python implementation is called SparseNALG for *"Sparse Neural Algebra"*.

```
def TNM(S,M):
    S_relsum = np.sum(S.values())
    matrix = csc_matrix((S_relsum, S_relsum))
    for (src, trg) in M:
        x_idx = list(S).index(src)
        y_idx = list(S).index(trg)
        x_extra = np.sum(S.values()[x_idx + 1:])
        y_extra = np.sum(S.values()[y_idx + 1:])
        R = M[(src, trg)]
        x = S[src]
        y = S[trg]
        r = rho(S_relsum, x + x_extra)
        p = pi(x + x_extra, x)
        i = iota(y, y + y_extra)
        k = kappa(y + y_extra, S_relsum)
        matrix += r*p*R*i*k
    return matrix
```

Figure 3.6: Python implementation of the *TNM* operation. Scipy's sparse matrices (csc_matrix) are used to improve efficiency. Rather than considering each $(S_x, S_y) \in S \times S$, only those pairs present in the map M implemented as a dictionary are considered. All other pairs are assumed to be zero matrices.

3.5 Applications

One of the main motivations for formalizing the algebra of neural networks is to identify and exploit the commonalities between existing representations. Frameworks such as TensorFlow or Theano use tensor-based expressions to evaluate a neural network given inputs. This representation is highly optimized for linear algebraic computation, but is not directly compatible with many other neural network representations. HyperNEAT for example, uses graph-based representations for networks. In computational neuroscience, there exists no dominant representation for neural networks, and none of the popular representations are compatible with frameworks developed for the CI community.

To demonstrate the flexibility of the approach taken in our work, the following example shows how neural network connectivity can be defined algebraically, and translated to two common representations used in computational intelligence and machine learning. We 'hand-crafted' a neural network, algebraically, and translated it to both a HyperNEAT network genome and to a tensor expression.

3.5.1 Constructing a Connectivity Matrix

In this example we are designing a simple network inspired by the structure of a feed-forward convolutional neural network. Since this example is being used only to define connectivity, we use relations to describe connectivity between sets of neurons. Note again, though, that matrices of real-valued weights can be used interchangeably.

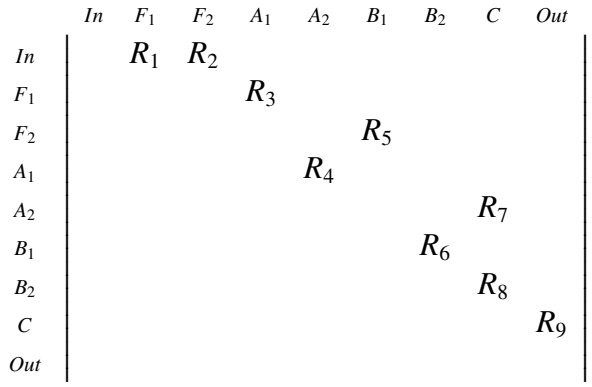

Figure 3.7: A matrix interpretation of the connectivity between pairs of sets in $S \times S$. With no relations on or below the diagonal of this matrix, we can assert that all instances of such a network will be feed-forward.

The first step is to declare a set of finite, non-intersecting sets of neurons and connectivity relations between them. Suppose we have the following set of 9 sets: $S = \{In, F_1, F_2, A_1, A_2, B_1, B_2, C, Out\}$. Abstractly, the number of neurons in each set is not important. Neural connectivity may be defined as a subset of connectivity matrices among the 81 possible pairs of sets in $S \times S$. In this example, we define only 9 such matrices as relations with the following types:

$$R_1 : In \to F_1 \qquad R_2 : In \to F_2 \qquad R_3 : F_1 \to A_1$$
$$R_4 : A_1 \to A_2 \qquad R_5 : F_2 \to B_1 \qquad R_6 : B_1 \to B_2$$
$$R_7 : A_2 \to C \qquad R_8 : B_2 \to C \qquad R_9 : C \to Out$$

The abstract connectivity between these sets is visualized by Figure 3.7. Instances of networks can now be specified by defining 1) the number of neurons in each set, and 2) concrete relations for each of $R_1, \ldots, R_9$. The remaining relations are assumed to be $\emptyset$.

Assuming instances of $R_1 \ldots R_9$ as defined in the digital appendix and omitted here, Figure 3.8 visualizes the application of *TNM* to build a single connectivity matrix *TotalNet* over all neurons. Notice that *TotalNet* is a concrete connectivity matrix that matches the abstract pattern visualized by Figure 3.7. Figure 3.9 visualizes *TotalNet* as a graph. As an algebraically constructed connectivity matrix, *TotalNet* can be manipulated arbitrarily by the operations presented in this chapter and by other linear or relation algebraic operations. For example, we may apply the *subst* operation to substitute each connection in *TotalNet* by an arbitrary connectivity pattern to obtain a more complex network. An example of this is visualized by Figure 3.10.

3.5.2 Connectivity Matrix as a HyperNEAT Genome

A HyperNEAT genome is an XML-formatted list of nodes and edges that defines artificial neural network connectivity for an individual in the underlying evolutionary algorithm. Though a genome can be seeded into an experiment, HyperNEAT does not provide any tools for assisting with manual network construction or importing other formats. The algebraic framework

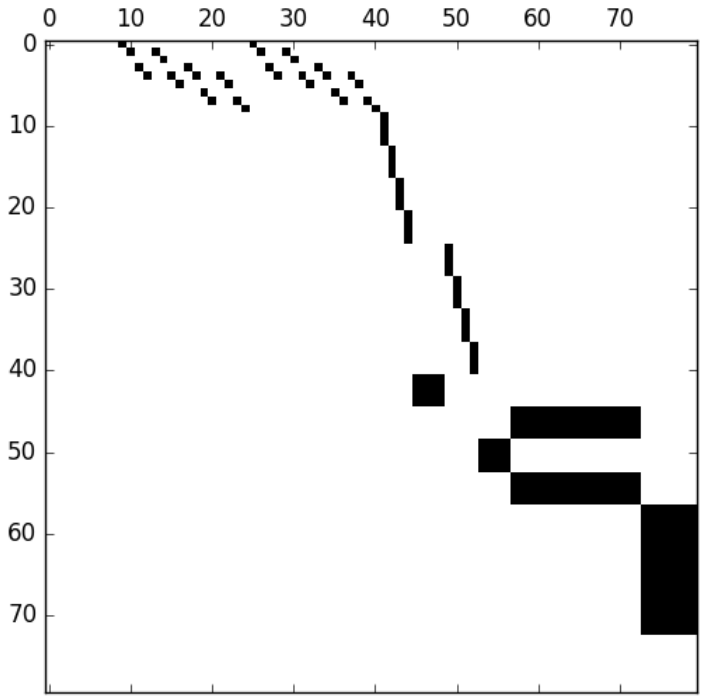

Figure 3.8: Matrix visualization of *TotalNet* — the result of *TNM* applied to S and a map $\mathcal{M} : S \times S \Longrightarrow \{R_1 \ldots R_9\}$ using the Python implementation.

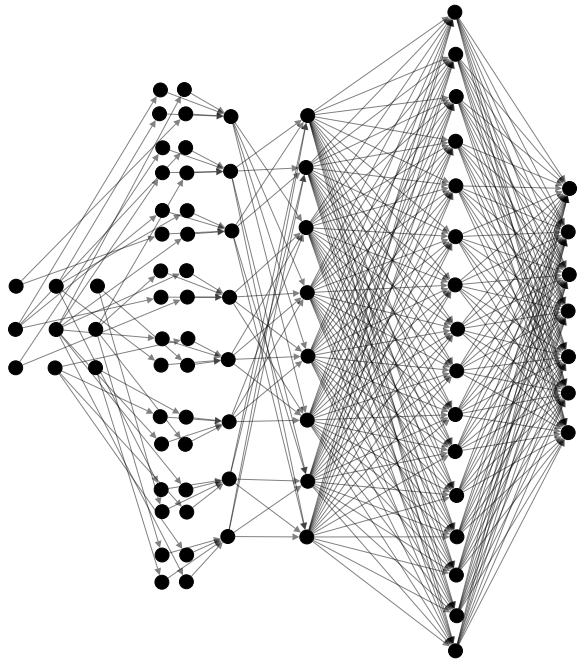

Figure 3.9: A graph visualization of *TotalNet*.

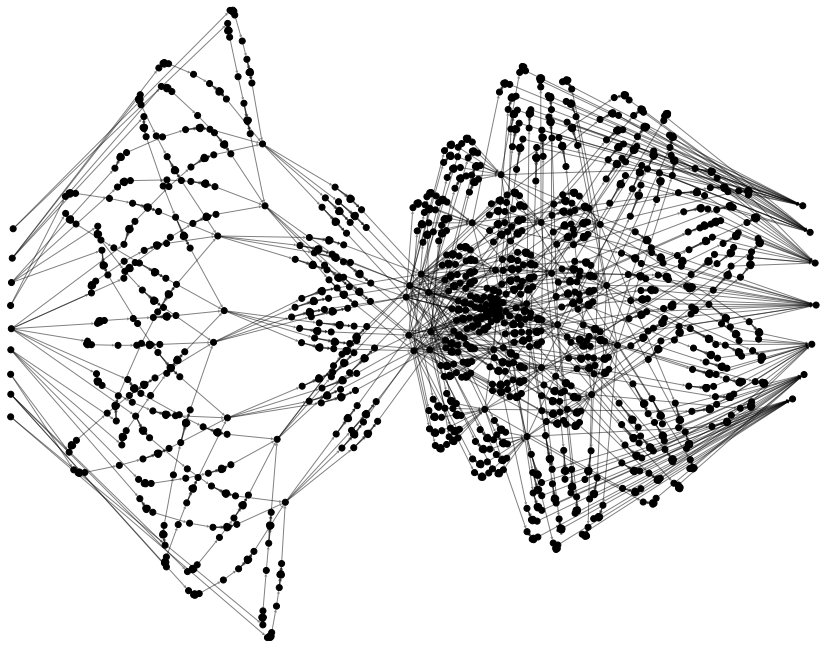

Figure 3.10: Example of *subst* being applied to substitute all connections in *TotalNet* with the connectivity pattern defined by Figure 3.4b.

presented in this chapter fills a gap by providing a set of tools that can be used to 'hand-craft' a HyperNEAT genome that could not easily be done otherwise.

Consider *TotalNet* as presented in the last section. Since this connectivity matrix denotes a network over a single set of neurons, it can easily be translated into a graph or by extension a HyperNEAT genome. The digital appendix includes a Python script to do this. The HyperNEAT interpretation of *TotalNet* is visualized by Figure 3.11.

3.5.3 Connectivity Matrix as a Tensor Operation

In tensor-based systems, neural network connectivity and evaluation strategy are defined simultaneously by tensor expressions. Since any matrix in linear algebra can be represented as a 2D tensor, little work needs to be done to translate an algebraic expression in our framework to a tensor operation. The neural network described by *TotalNet* can be translated to a tensor operation by reorganizing and evaluating its non-zero submatrices $R_1 \dots R_9$ as follows:

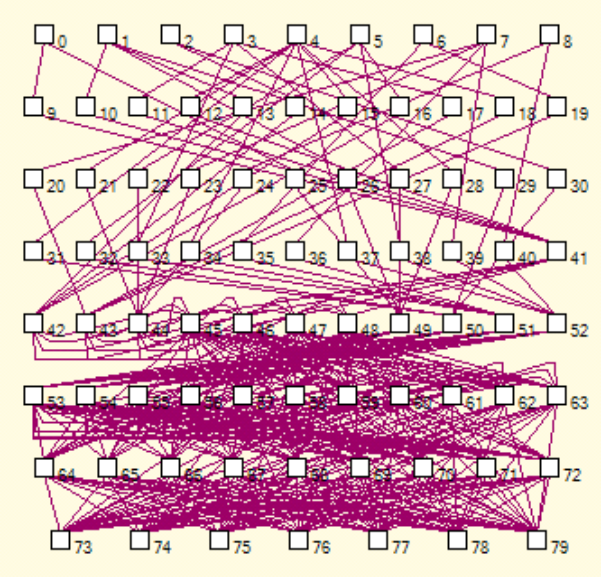

Figure 3.11: *TotalNet* exported as a HyperNEAT genome and visualized as a seed network for HyperSharpNEAT — a C# implementation of HyperNEAT.

$$eval(R_1 \dots R_9) : In \to Out := T_1 \cdot T_2 \cdot T_3 \cdot T_4 \cdot T_5$$

$$\begin{aligned}
T_1 &= R_1 \cdot \iota_{F_1 \to F_1 \oplus F_2} + R_2 \cdot \kappa_{F_2 \to F_1 \oplus F_2} \\
T_2 &= \pi_{F_1 \oplus F_2 \to F_1} \cdot R_3 \cdot \iota_{A_1 \to A_1 \oplus B_1} \\
&\quad + \rho_{F_1 \oplus F_2 \to F_2} \cdot R_5 \cdot \kappa_{B_1 \to A_1 \oplus B_1} \\
T_3 &= \pi_{A_1 \oplus B_1 \to A_1} \cdot R_4 \cdot \iota_{A_2 \to A_2 \oplus B_2} \\
&\quad + \rho_{A_1 \oplus B_1 \to B_1} \cdot R_6 \cdot \kappa_{B_2 \to A_2 \oplus B_2} \\
T_4 &= \pi_{A_2 \oplus B_2 \to A_2} \cdot R_7 + \rho_{A_2 \oplus B_2 \to B_2} \cdot R_8 \\
T_5 &= R_9
\end{aligned}$$

Assuming the above instances, the expression $T_1 \cdot T_2 \cdot T_3 \cdot T_4 \cdot T_5$ is an algebraic expression that can be interpreted without modification as a 2D tensor expression. Consistent with each example in this section, the sizes of the sets and connectivity matrices involved are arbitrary. As a result, this tensor expression is valid for any instance of the network described by Figure 3.7.

3.6 Conclusions and Future Work

The applications discussed in Section 3.5 serve as a proof of concept that a framework combining relation and linear algebraic operations provides a generalization for graph and tensor based neural networks. The main products of this work are the algebraic framework and an example of its implementation as *SparseNALG*. The approach used in this chapter serves as an example of how applied algebraic methods can contribute significantly to formalization efforts and to the development of practical tools for the CI and computational neuroscience commu-

nities. This work has two major avenues for future work, the first being more formal and the second being practical.

The matrices in this chapter are always assumed to contain elements from the field $\mathbb{R}$ with relations being interpreted as a subset of those matrices. We did this to keep the presentation relatively simple and to ensure all objects and operations are compatible with linear algebra for implementations.

However, the connection between matrices and relations is in fact much more general. Instead of using matrices over $\mathbb{R}$, which we used to denote either Boolean-valued or weighted connectivity between neurons, a much more general *semiring* can be used to provide matrix elements and elementary operations. These could be, for example, complex functions that model biological neural dynamics. Conveniently, none of the operations presented in this chapter need to be redefined to build connectivity matrices that use semiring elements instead of real numbers. In future work we will show this explicitly and provide an algebraic framework that supports complex neuron models.

For computational neuroscientists, this will provide a formal algebraic framework for modelling both network connectivity and neural dynamics. Such a formal language for describing and reasoning about biological neural networks does not currently exist. For the artificial intelligence community, this will provide not only a formal language and reasoning tools, but also software that will enable computational algebraic methods to be applied to neural networks with complex neural dynamics. Practical benefits include the type of optimization made possible by basic linear algebra subroutine (BLAS) systems such as those used by Spiral [4].

The framework presented in this chapter has two immediately possible applications of interest to both computational neuroscientists and the artificial intelligence community. First, this framework immediately enables the transfer of neural networks between systems. For example, a TensorFlow neural network can now be easily converted to a HyperNEAT genome. In future work, we will develop a set of software tools to seamlessly translate neural network models between any popular formats based on graphs or tensors, including CSA, NeuroML[6], HyperNEAT, TensorFlow, and PyTorch. Finally, the symbolic representation this framework provides for neural networks seems well-suited for symbolic evolutionary computation. Future work will explore the effectiveness of applying genetic programming to evolve algebraic expressions for neural networks in machine learning experiments.

Bibliography

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] CURTO, C., ITSKOV, V., VELIZ-CUBA, A., AND YOUNGS, N. The Neural Ring: An Algebraic Tool for Analyzing the Intrinsic Structure of Neural Codes. *Bulletin of Mathematical Biology 75*, 9 (2013), 1571–1611.

[3] DJURFELDT, M. The Connection-set Algebra—A Novel Formalism for the Representation of Connectivity Structure in Neuronal Network Models. *Neuroinformatics 10(3)* (2012).

[4] Franchetti, F., Low, T. M., Popovici, D. T., Veras, R. M., Spampinato, D. G., Johnson, J. R., Püschel, M., Hoe, J. C., and Moura, J. M. Spiral: Extreme performance portability. *Proceedings of the IEEE 106*, 11 (2018), 1935–1968.

[5] Freyd, P., and Scedrov, A. *Categories, Allegories*. North-Holland, 1990.

[6] Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., and Billings, G. O. NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. *PLoS Computational Biology 6(6)* (2010).

[7] Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–.

[8] Killingbeck, D., Teixeira, M. S., and Winter, M. Relations among Matrices over a Semiring. *Relational and Algebraic Methods in Computer Science (RAMiCS 15)* (2015), Killingbeck, D., Santos Teixeira, M., Winter, M.

[9] Menezes, T., and Roth, C. Symbolic regression of generative network models. *Scientific reports 4* (2014).

[10] Oliphant, T. E. *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[11] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[12] Ritter, G. X., Li, D., and Wilson, J. N. Image algebra and its relationship to neural networks. In *1989 Orlando Symposium* (1989), International Society for Optics and Photonics, pp. 90–101.

[13] Schmidt, G. *Relational Mathematics*. Encyplopedia of Mathematics and Its Applications, 2011.

[14] Sporns, O. Small-world connectivity, motif composition, and complexity of fractal neuronal connections. *Biosystems 85*, 1 (2006), 55–64.

[15] Sporns, O. Contributions and challenges for network models in cognitive neuroscience. *Nature neuroscience 17*, 5 (2014), 652–660.

Chapter 4

Novelty Search for Deep Reinforcement Learning Policy Network Weights by Action Sequence Edit Metric Distance

4.1 Introduction

Reinforcement learning (RL) [26] problems often feature sparse or infrequently accessible reward, and *deceptive local optima* that impose difficult challenges to many learning algorithms. These can be characterized as environment features that cause short-term reward seeking behaviour to learned much more easily than long-term, higher value reward. Algorithms that optimize strictly for reward often produce degenerate policies that cause agents to under-explore their environments or under-develop strategies for increasing reward. Deceptive local optima have proved to be equally challenging for both gradient-based RL algorithms, including DQN [16], and gradient-free algorithms including genetic algorithms (GAs) [25].

Deceptive local optima in reinforcement learning have long been studied by the evolutionary algorithms community — with concepts including *novelty search* being introduced in response [12]. The deep RL community has responded with similar ideas and tools, but in purely gradient-based learning frameworks. A good example is given by recent work from Google Brain and DeepMind that promotes *episodic curiosity* in deep RL benchmarks [21]. These methods were both designed to address deceptive local optima by substituting or supplementing reward signal with some measure of behavioural novelty. In practice, an agent's behaviour has usually been defined in terms of its environment. Behaviour is often quantified using information contained in environment observations. For example, agents that reach new locations [25], or that reach the same location using an alternate route[21], can be rewarded for their novel behaviour.

In this chapter, we investigate whether agent behaviour can be quantified more generally and leveraged more directly. We investigate the following question: "*Can the history of actions performed by agents be used to promote innovative behaviour in benchmark RL problems?*' Towards answering this, we implemented two novel methods for incorporating behavioural history in an evolutionary algorithm designed to effectively train deep RL networks. The base algorithm is an approximate replication of Such et al.'s genetic algorithm (GA) for learning

DQN network [16] weights. This is a very simple yet effective gradient-free approach for learning DQN policies that are competitive with those produced by Deep Q-learning [25].

Both methods are GA extensions based on Lehman and Stanley's *novelty search* [12] — an evolutionary algorithm designed to avoid deceptive local optima by defining selection pressure in terms of behaviour instead of conventional optimization criteria such as reward signal. Novelty search has been shown to be an effective tool for promoting innovation in RL [25]. In this chapter, we introduce the use of Levenshtein distance [13] — a form of *string edit metric distance* — as the behavioural distance function in a novelty search for deep RL network weights.

The first method (Method I) is an implementation of novelty search in which, during training, the reward signal is completely substituted by a novelty score based on the Levenshtein distance between sequences of game actions. In a novelty search, *behaviour characteristics* are stored in an *archive* for a randomly-selected subset of individuals in each generation. We define the behaviour characteristic as the sequence of actions performed by an agent during the training episode. Selection pressure is then determined by computing the *behavioural distance* between individuals in the current population and those in the archive — which we define as Levenshtein distance.

The second method (Method II) is not a novelty search, but rather a modification to the Base GA that incorporates elements of novelty search to avoid population convergence to locally-optimal behaviours. The modified algorithm detects slowing learning progress as measured using game scores in validation episodes. When validation scores are non-increasing for a fixed number of episodes, the population is regenerated by sampling the archive for individuals whose behaviours were most novel compared to the current population — a concept related to restarting and recentering in evolutionary algorithms [9].

Using two sets of experiments, we evaluated each method's effectiveness for learning RL policies for four Atari 2600 games, namely Assault, Asteroids, MsPacman, and Space Invaders. We found that while Method I is less effective than the Base GA for learning high-scoring policies, it returns policies that are behaviourally distinct. For example, we observed greater uses of obstacles or greater agent lifespans in some games. Method II was more effective than Method I for learning high-scoring policies. In two out of four games, it produced better-scoring policies than the Base GA, and in one out of four, it produced better-scoring policies than the original DQN learning method.

Importantly, and in contrast to previous uses of novelty search for deep RL, the behaviour characteristic and behavioural distance function used here *do not require or use environment-specific knowledge*. While such a requirement is not inherently a hindrance, it is convenient to have tools that work in more general contexts. Compared to related methods that use memories of observations (usually environment observations) to return to previous states [4] or to re-experience or re-visit under-explored areas [21], archives of action sequences are relatively compact, easy to store, and efficient to compare. As such, the methods presented in this chapter can either be used as stand-alone frameworks, or as extensions to existing methods that use environmental memory to improve learning.

In the next section, we give an overview of the Base GA and architecture, the Atari benchmark problem, and our experimental setup. In Section 4.3 we provide a full definition of novelty search and details of our implementation based on action sequences and Levenshtein distance (Method I). In Section 4.4 we provide further details for Method II. Section 4.5 de-

scribes experiments and results, and is followed by discussion in Section 4.6.

4.2 Highly-Scalable Genetic Algorithms for Deep Reinforcement Learning

The conventional objective in RL is to produce an optimal *policy* — a function that maps states to actions such that *reward*, or gain in reward, is optimized. The methods introduced in this chapter are extensions of a replicated state-of-the-art GA for learning deep RL policy network weights introduced by Such et al. in [25].

4.2.1 DQN Architecture and Preprocessing

A RL policy network is an instance of a neural network that implements a RL policy. A Q network uses a neural network to approximate a Q function. For comparability to related work that uses evolutionary algorithms [25], we used the DQN neural network architecture [16] to implement a policy network in all experiments. This means that the role of the neural network is simply to map an environment observation to an action. Like the original DQN architecture, this network consists of three convolutional layers with 32, 64, and 64 filters, respectively, followed by one dense layer with 512 units. The convolutional filter sizes are 8×8, 4×4, and 3×3, respectively. The strides are 4, 2, and 1, respectively. All network weights are initialized using Glorot normal initialization. All network layer outputs use rectified linear unit (ReLU) activation. All game observations (frames) are downsampled to $84 \times 84 \times 4$ arrays. The third dimension reflects separate intensity channels for red, green, blue, and luminosity. Consecutive game observations are summed to rectify sprite flickering.

4.2.2 Seed-Based Genetic Algorithm

Perhaps surprisingly, very simple genetic algorithms have been shown to be competitive with Deep Q-learning for learning DQN architecture parameterizations [25]. In their paper, Such et al. introduced an efficient seed-based encoding that enables very large network parameterizations to be indirectly encoded by a list of deterministic pseudo-random number generator (PRNG) seeds. This, in contrast to a direct encoding, scales with the number of evolutionary generations (typically thousands) rather than the number of network connections (typically millions or more). This encoding enables GAs to work at unprecedented scales for tuning neural network weights. It thus enables, more generally, a new wave of exploration for evolutionary algorithms and deep learning.

For the present work, we implemented a GA and encoding approximately as described in [25] using Keras [3], a high-level interface for Tensorflow [1], and NumPy [19]. An individual in the GA's population is encoded by a list of seeds for both Keras' and NumPy's deterministic PRNGs. The first seed is used to initialize network weights. Subsequent seeds are used to produce additive mutation noise. A constant scaling factor (mutation power) is used to scale down the intensity of noise added per generation.

A network parameterization is thus defined by:

$$\Theta^n = \Theta^{n-1} + \sigma\epsilon(\tau_n) \tag{4.1}$$

$$\Theta^0 = \phi(\tau_0) \tag{4.2}$$

where Θ^n denotes network weights at generation n, τ denotes the encoding of Θ^n as a list of seeds, ϕ denotes a seeded, deterministic initialization function, $\epsilon(\tau_n) \sim \mathcal{N}(0, 1)$ denotes a seeded, deterministic, normally-distributed PRNG seeded with τ_n and σ denotes a constant scaling factor (mutation power).

As in its introductory paper, the GA does not implement crossover, and mutation simply appends a randomly-generated seed to an individual's list τ. The GA performs *truncated selection* — a process whereby the top T individuals are selected as reproduction candidates (parents) for the next generation. From these T parents, the next generation's population is uniformly, randomly sampled with replacement, and mutated.

The GA also implements a form of *elitism* — a commonly used tactic to ensure that the best performing individual is preserved in the next generation without mutation. A separate set of *validation episodes* is used to help determine the elite individual during training. This has the effect of adding secondary selection pressure for generalizability and helps to reduce overfitting. More details are given in Section 4.5.

It is important to note that this encoding imposes network reconstruction costs that would not be needed using a direct encoding. The compact representation, though, enables a high degree of scalability that would not be practical using a direct encoding. Algorithm descriptions and source code for the Base GA, Method I, and Method II are provided in the Appendix and Digital Appendix, respectively. For further details on the Base GA, refer to [25].

4.2.3 Atari 2600 Benchmark

The Atari 2600 Benchmark is provided as part of OpenAIGym [2] — an open-source platform for experimenting with a variety of reinforcement learning problems. Work by Mnih et al. [15] introduced a novel method and architecture for learning to play games directly from pixels — a challenge that remains difficult [8]. Though many enhancements and extensions have been developed for DQN, no single learning method has emerged as being generally dominant [8], [4].

The games included in the Atari 2600 benchmark provide a diverse set of control problems. In particular, the games vary greatly in both gameplay and logic. In MsPacman, for example, part of the challenge comes from the fact that the rules for success change once MsPacman consumes a pill. To achieve a high score, the player or agent must shift strategies from escape to pursuit. This is quite different from Breakout for example — a game in which the optimal paddle position can be computed as a function of consecutive ball observations. The variety of problems provided by this benchmark makes it an interesting set to study.

Before designing experiments, it is important to ask whether the chosen methods are plausibly capable of learning high quality policies. In games like MsPacman, is it reasonable to expect that a strictly feed-forward network architecture like DQN should be capable of producing high-quality policies? Though we do not investigate this question in the experiments presented in this chapter, we comment on it in Section 4.6.

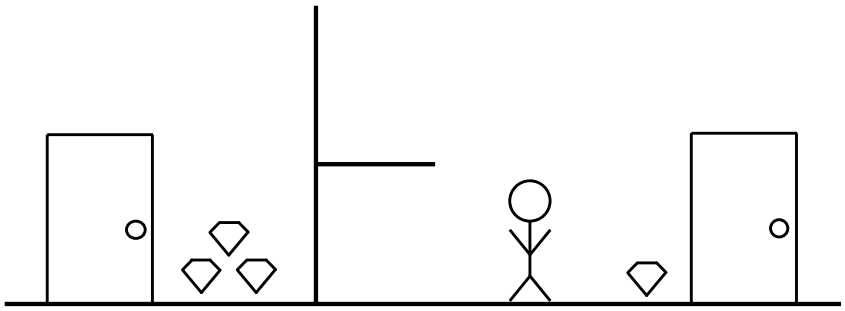

Figure 4.1: Example of a simple game stage with a deceptive local optimum. Assuming the goal is for the player to earn points by collecting as many diamonds as possible before using a door to exit the stage, a globally suboptimal policy may never learn to scale the wall to the player's left and collect three additional diamonds.

4.2.4 Experimental Setup

The Base GA and encoding for our experiments is an approximate replication of the GA and encoding introduced by Such et al. in [25]. All code was written in Python and uses Keras and TensorFlow for network evaluation. All experiments were run on a CPU-only 32-core Microsoft Azure cloud-based virtual machine (Standard F32s_v2). The code is scalable to any number of threads and could be adapted to run on a distributed system. A single run of a Method II experiment (see Table 4.1) required roughly 120 wall-hours of compute time using this system.

4.3 Novelty Search Over Action Sequences

Reinforcement learning problems often feature deceptive local optima or sparse reward signals. For example, consider a simple platform game in which the player navigates the environment to collect rewards. Environmental obstacles, such as walls and stacked platforms, increase gameplay complexity and introduce latent optimization criteria. A simple example of such a game is visualized by Figure 4.1.

To overcome such challenges, agents may need to develop behavioural or strategic innovations that are not exhibited by any agent in the initial population. While it is possible for innovations to appear strictly as a result of mutations using the Base GA, these innovations are only promoted to the next generation if they immediately yield a positive return in terms of reward signal. Introduced by Lehman and Stanley in [12], novelty search addresses environmental challenges in RL by redefining optimization criteria in terms of behavioural innovation. In the context of evolutionary algorithms, including GAs, a pure novelty search defines fitness in terms of novelty rather than reward. Novelty search requires the following additional components over a typical genetic algorithm: 1) a behaviour characteristic, 2) a behavioural distance function, and 3) an archive of stored behaviour characteristics.

4.3.1 Behaviour Characteristic

The behaviour characteristic of a policy π, denoted by $BC(\pi)$, is a description of some behavioural aspect of π with respect to its environment. For example, the behaviour characteristic of a policy for controlling an agent in a 2D navigation environment could be the coordinate pair of the agent's location at the end of an episode. The behavioural distance between two behaviour characteristics is the output of a suitable distance metric function d applied to two behaviour characteristics $BC(\pi_i)$ and $BC(\pi_j)$. For example, assuming that $BC(\pi)$ maps a policy π to the final resting coordinates of an agent in 2D space, the behavioural distance function d could be Euclidean distance in $\mathbb{R}^2$. Continuing with this example, an archive would consist of a randomly-selected subset of final resting coordinates reached by agents throughout training.

In previous work, both behaviour characteristics and behavioural distance functions were assumed to be domain-specific: they would not usually generalize to other environments. In this chapter, we introduce a generalized formulation of novelty that applies to any game in the Atari 2600 benchmark, and that generalizes to many more control problems.

We define the behaviour characteristic of a policy to be the sequence of discrete actions performed by an agent in response to consecutive environment observations. These action sequences are encoded as strings of length F, where F is the maximum number of frames available during training. Characters are either elements of a game's *action space* (distinct symbols that encode a button press) or the character x, which is reserved to encode a *death action* or non-consumed frame.

4.3.2 Behavioural Distance Function

We define the behavioural distance function as an approximation of the Levenshtein distance [13] between action sequences encoded by strings. Note that other string edit distance metrics, such as Hamming distance [7], or distributional distance metrics, such as Kullback-Leibler divergence [11] could also be used as behavioural distance functions. We chose to base our behavioural distance function on Levenshtein distance because it captures temporal relationships between action sequences that the other metrics do not.

For example, two action sequences encoded by *x12345* and *12345x* are much closer in Levenshtein space (two edits: one deletion and one insertion) than in Hamming space (six edits: one substitution at each position). The Kullback-Leibler divergence between the distribution of actions in these two strings is zero since each action occurs exactly once, thus failing to discriminate the two policies by their statistics.

The additional descriptive power of Levenshtein distance comes with higher computational costs. The time complexity of computing the Levenshtein distance between two strings of length n is $O(n^2)$. For large enough n, Levenshtein distance computations will impose a bottleneck on learning – a problem we encountered in preliminary experiments.

To remedy this, we simply restrict the size of n by splitting action sequences into fixed-length segments and compute the cumulative Levenshtein distance between corresponding segments. All experiments reported in this chapter use $n = 500$ for computing segmented Levenshtein distance. While some information is lost using this approach, the practical reduction in runtime necessitates the choice. The behavioural distance function d which computes segmented Levenshtein distance is defined by Equation 4.3:

$$d(A, B) = \sum_{s=0}^{S-1} L(A_{sn\cdots sn+n-1}, B_{sn\cdots sn+n-1}) \tag{4.3}$$

where A and B are two action sequences encoded by strings, S is the number of segments, n is the length of each segment, and L computes the Levenshtein distance between two strings. The number of segments n is determined by computing $\lceil F/s \rceil$, where F is the number of characters in A and B, equal to the maximum number of frames available during training. In experiments, L is computed using the Python package python-Levenshtein [6].

4.3.3 Hybrid Algorithm

In a pure novelty search, fitness in the GA would be defined entirely by novelty scores. The experiments reported in this chapter for Method I use a hybrid algorithm in which, like for a pure novelty search, selection pressure during training is solely determined using novelty scores. To identify the generation elite, however, we use the validation game score instead of a novelty score. This is due to the episodic nature of our chosen behaviour characteristic. Action sequences archived during training are specific to the training episode. To be consistent with other experiments using novelty search, we avoided the introduction of validation-specific archives for additional episodes. And so while novelty is the dominant component of selection pressure, we make this distinction clear to differentiate it from a pure novelty search. Experiments using Method I are discussed in Section 4.5.1.

4.4 Novelty-Based Population Resampling in Genetic Algorithms

Reward sparsity is highly variable between RL problems. The Atari 2600 game Montezuma's Revenge, for example, is a complex platform game that requires significant exploration, puzzle-solving, and other strategies to complete. Until very recently, it has proved challenging to develop high-performing policies for this game without human-generated playthrough examples. A new method called *Go-Explore* was recently introduced as the state-of-the-art for producing Montezuma's Revenge policies [4]. Though it is not based on evolutionary algorithms or the DQN architecture, Go-Explore borrows ideas from novelty search – namely the use of an archive to store and recall states over the course of policy search.

Motivated by this result, we designed Method II as an extension to the Base GA that adds features inspired by Go-Explore. In particular, we designed experiments to test whether an archive of action-sequences recorded throughout evolution could be effectively used for promoting innovation. Over the course of evolution, a randomly selected subset of individuals together with their action sequences are archived. This archive gradually collects individuals that could potentially lead to better policies than those that were selected for reproduction. Since the Base GA's selection pressure is based entirely on game score, it is still susceptible to converge around locally optimal policies and to discard innovations that do not yield immediate returns.

Hyperparameter	Method I	Method II
Population Size (N)	100 + 1	1,000 + 1
Generations	500	1000
Truncation Size (T)	20	20
Mutation Power (σ)	0.002	0.002
Archive Probability	0.1	0.01
Max Frames Per Episode (F)	20,000	20,000
Training Episodes	1	1
Validation Episodes	5	30
Improvement Generations (*IG*)		10

Table 4.1: Hyperparameters for Method I and Method II experiments. Note that the Improvement Generations hyperparameter is only used in Method II experiments, and that baseline results do not use archiving. Population sizes are incremented to account for elites.

Since novelty scores are not computed to determine primary selection pressure, Method II is not a novelty search. Instead, novelty scores are only computed when the algorithm detects that policy generalizability has stagnated over some number of generations. In such cases, the algorithm generates a new population by sampling the archive for individuals whose behaviour characteristics are most distant from the current population. These sampled individuals are used as parents for the next generation and the GA proceeds otherwise identically as the Base GA.

As expected given DQN's prior ineffectiveness for learning MONTEZUMA'S REVENGE, both Methods I and II were also unsuccessful in preliminary experiments. As a result we excluded it from main experiments, which are discussed in the next Section.

4.5 Experiments

All experiments use the same four games: ASSAULT, ASTEROIDS, MSPACMAN, and SPACE INVADERS. These games were chosen because they each feature gameplay that falls into one of two categories: games with one- or two-dimensional navigation. ASSAULT and SPACE INVADERS both allow the player or agent to move an avatar across a one-dimensional axis at the bottom of the game screen, while ASTEROIDS and MSPACMAN allow a much greater range of exploration. Experiments using these four games also provide new results for the Base GA's effectiveness for learning to play Atari using the DQN architecture.

In all experiments, we provide a baseline result using our replication of the GA described by Such et al. in [25]. The purpose of this baseline is to provide a replicated benchmark for using GAs to learn DQN architecture weights. While we acknowledge that many modified versions of the DQN architecture have been developed [8], we use the original architecture to ensure comparability to a wide variety of existing results, thereby controlling for differences between algorithms rather than architectures. Video comparisons of the Base GA and Methods I and II are included in the Digital Appendix.

	Mean			St. Dev.		
Game	Base GA	Method I	DQN	Base GA	Method I	DQN
ASSAULT	812	488	**3359**	228	158	775
ASTEROIDS	1321	736	**1629**	503	426	542
MSPACMAN	2325	1437	2311	351	527	525
SPACE INVADERS	500	474	**1976**	303	195	893

Table 4.2: Comparison of Base GA and Method I testing results over 30 episodes not used in training or validation. Means and standard deviations are measured in game score units. Bolded means denote significantly better testing performance ($p < 0.05$ in a two-tailed t-test). The Base GA outperforms Method I in all but one game.

4.5.1 Method I

Method I was designed to test the merits of using novelty search over agent action sequences in the Atari 2600 benchmark. This method substitutes reward signal with a measure of behavioural novelty as the selection pressure in an evolutionary search for DQN architecture weights. For comparability with existing gradient-based [16] and gradient-free [25] methods, we evaluated Method I against the Base GA. Due to compute time constraints, these experiments were run at a smaller scale than for Method II. Hyperparameters are summarized by Table 4.1.

Method I training progress is visualized by Figure 4.2 and testing evaluation of Method I policies is summarized by Table 4.2. Overall, Method I does not produce policies that score better than either DQN or the Base GA. On the other hand, it is interesting to evaluate the behaviours of policies generated by (almost) completely ignoring the reward signal during training.

Results for Method I experiments suggest that novelty search indeed creates selection pressure for innovation. For example in SPACE INVADERS, we observed more regular uses of obstacles by agents trained using Method I than the Base GA. And in MSPACMAN, we observed that agents trained using Method I tended to explore more paths than their Base GA-trained counterparts. In two out of four games (ASSAULT and SPACE INVADERS), agents trained by Method I had significantly longer lifespans than those trained by the Base GA (see Table 4.3). This could either be due to the behavioural distance function's sensitivity to differences in lifespan, or to defensive innovations that increase agent lifespan.

To determine whether Levenshtein distance is effectively different than lifespan as a behavioural distance function, we conducted an additional small-scale experiment using MSPACMAN (Method I-L). We observed that lifespan, measured by counting the number of frames an agent survives in its environment, is not an equivalent behavioural distance function to Levenshtein distance. See Table 4.4 for hyperparameters and Figure 4.3 for results.

A problem with this approach is that by continually selecting for innovation, there may be insufficient evolutionary time for innovations to be optimized. Method II attempts to remedy this by integrating secondary selection pressure for novelty into an otherwise standard search for reward-optimizing policies.

	Mean		St. Dev.	
Game	Base GA	Method I	Base GA	Method I
ASSAULT	3538	**5242**	995	1998
ASTEROIDS	1263	1223	545	668
MSPACMAN	**1112**	931	137	142
SPACE INVADERS	1264	**1552**	369	285

Table 4.3: Comparison of Base GA and Method I lifespans over 30 episodes not used in training or validation. Means and standard deviations are shown in numbers of frames over which agents survived. Bolded means denote significantly longer lifespans ($p < 0.05$ in a two-tailed t-test). Method I produced agents with significantly longer mean lifespans in testing in ASSAULT and SPACE INVADERS.

Hyperparameter	Method I-L
Population Size (N)	100 + 1
Generations	500
Truncation Size (T)	10
Mutation Power (σ)	0.004
Archive Probability	0.02
Max Frames Per Episode (F)	2,500
Training Episodes	2

Table 4.4: Hyperparameters for experiment on Method I-L. Validation episodes were not used in this experiment – elites determined using highest game score in training over 2 episodes.

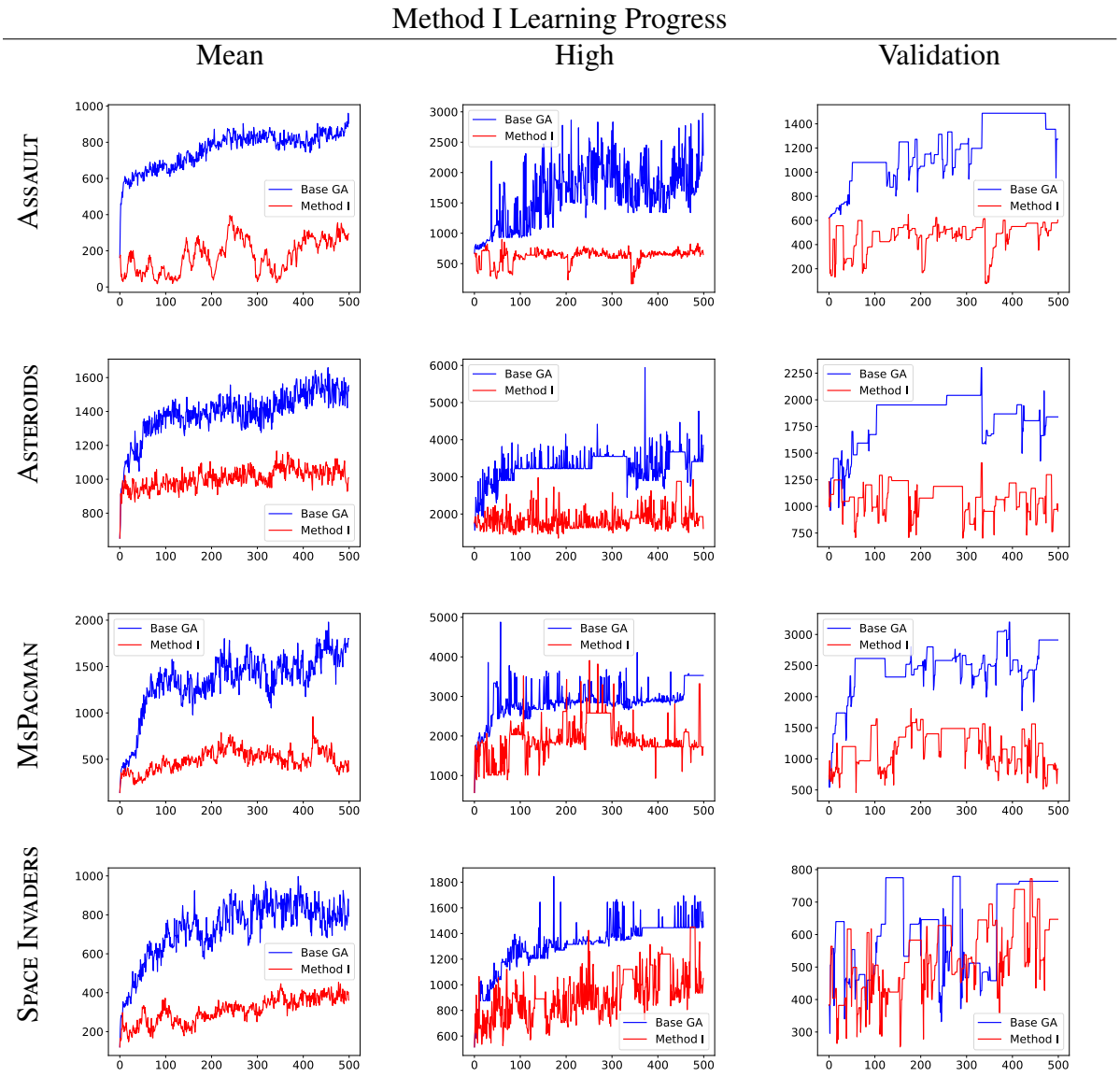

Figure 4.2: Base GA and Method I learning progress.

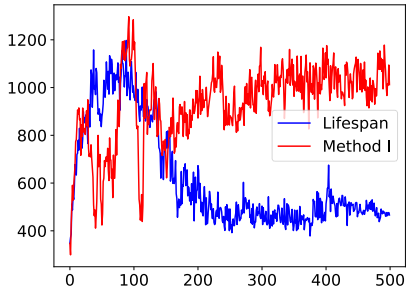

Figure 4.3: Population mean game score over generations during training on MSPACMAN. Mean scores diverge after generation 160. Levenshtein distance (Method I) and lifespan are thus not equivalent behavioural distance functions.

4.5.2 Method II

Method II was designed to help the GA avoid *stagnation* or premature convergence to locally optimal solutions. This method adds two components to the Base GA: 1) stagnation detection, and 2) population resampling. Stagnation is detected by examining the trend of validation scores. In the Base GA, validation episodes are used solely to identify the elite individual of a population. In Method II, learning progress is declared to be stagnant when validation scores are non-increasing over 10 episodes. This is reflected by the hyperparameter *Improvement Generations* (*IG*) in Table 4.1. Population resampling is achieved by sampling $2 * T$ individuals from the archive to be the next generation's parents. For Method II, novelty scores are used to select archived individuals whose policies were most different from the current population, according to the behavioural distance metric.

As a baseline, we tested whether novelty-based population resampling is better than sampling random individuals from the archive for learning MSPACMAN policies. Using the same evaluation criteria and hyperparameters as for Method II (see Table 4.1) we found that, for MSPACMAN, novelty-based population resampling is significantly better than random archive sampling. This result is summarized by Table 4.5 and motivated further evaluation of the method applied to other games.

We then evaluated Method 2 by comparing it to the Base GA. These experiments were run using similar hyperparameters to previous related work [25] — (see Table 4.1). Method II training progress is visualized by Figure 4.4 and testing evaluation of Method II is summarized by Table 4.6. In testing, Method II yielded improved results over the Base GA in two out of four games and no significant change in two out of four games. We also compared Method II testing scores to those reported in [16] for Deep Q-learning — see Table 4.7. Method II outperforms DQN methods in one game, and is outperformed by DQN methods in two games. These mixed results are consistent with previous work [25].

	Mean		St. Dev.	
Game	Random	Method II	Random	Method II
MSPACMAN	3377	**3790**	661	322

Table 4.5: Comparison of Method II (novelty-based population resampling) to random population-resampling over 30 episodes not used in training or validation. In MSPACMAN, Method II yielded better mean game scores in testing than random population resampling with $p < 0.05$ in a two-tailed t-test.

	Mean		St. Dev.	
Game	Base GA	Method II	Base GA	Method II
ASSAULT	1219	1007	676	413
ASTEROIDS	1263	1476	590	640
MSPACMAN	3385	**3700**	633	209
SPACE INVADERS	615	**1211**	323	244

Table 4.6: Comparison of Base GA and Method II testing results over 30 episodes not used in training or validation. Means and standard deviations are measured in game score units. Bolded means denote significantly better testing performance ($p < 0.05$ in a two-tailed t-test). Method II improves learning in 2 out of 4 games over the Base GA.

	Mean		St. Dev.	
Game	DQN	Method II	DQN	Method II
ASSAULT	**3359**	1007	775	413
ASTEROIDS	1629	1476	542	640
MSPACMAN	2311	**3700**	525	209
SPACE INVADERS	**1976**	1211	893	244

Table 4.7: Comparison of DQN and Method II using testing scores over 30 randomly-seeded episodes reported in [16]. Means and standard deviations are measured in game score units. Means and standard deviations are measured in game score units. Bolded means denote significantly better testing performance ($p < 0.05$ in a two-tailed t-test). Method II outperforms DQN in one game, performs similarly to DQN in one game, and is outperformed by DQN in two games. These mixed results are consistent with previous comparisons between gradient-based and gradient-free learning methods [25].

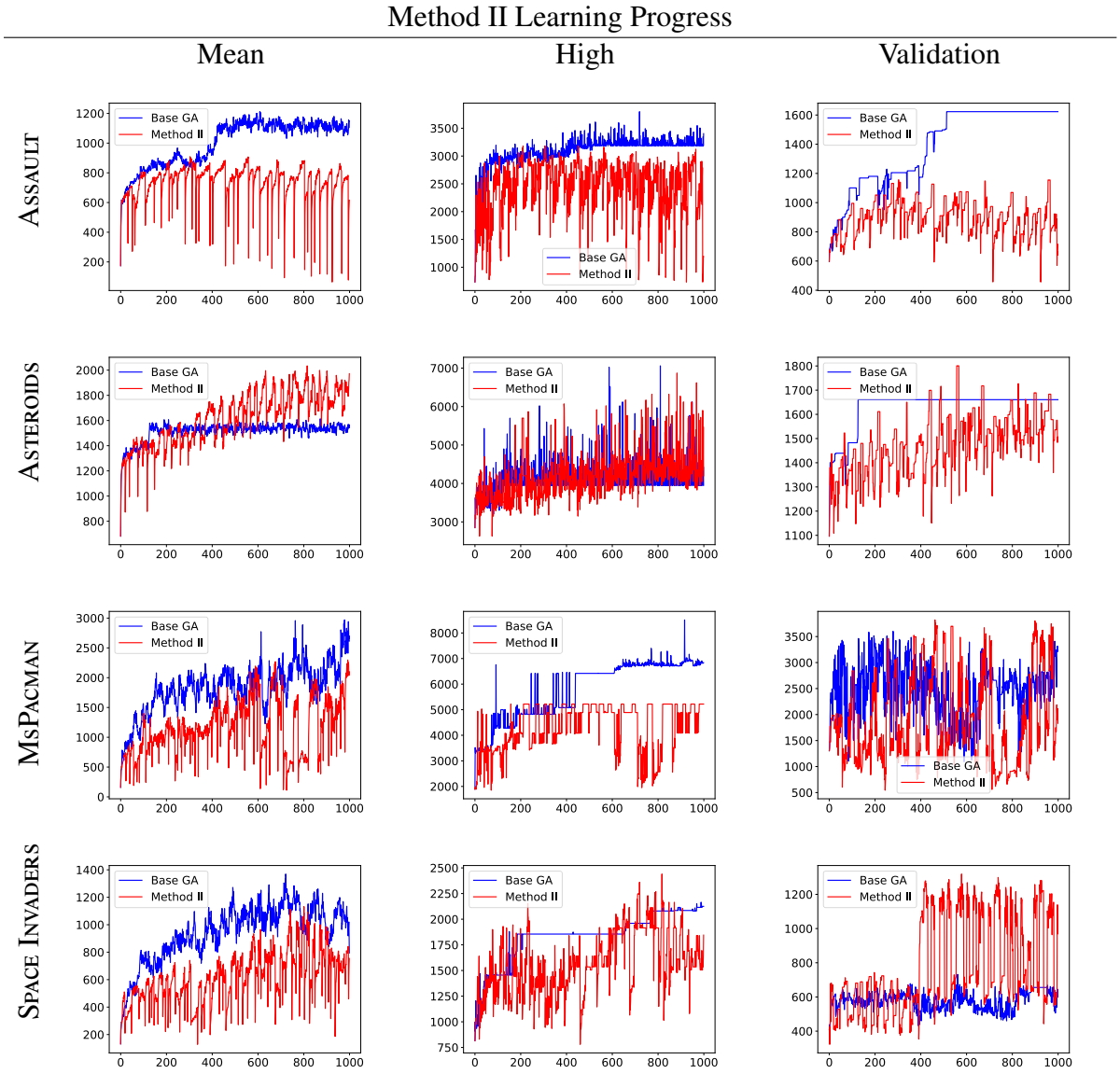

Figure 4.4: Base GA and Method II learning progress. Mean denotes population mean game score over generations in training, high denotes score of top-performing individual over generations in training, and validation denotes the mean score of the best-generalizing individual to 30 differently-seeded environments. In each generation, the best individual in validation is designated as the elite. In 3 out of 4 games, validation scores reach a higher maximum. Whereas the Base GA seemingly failed to escape a local optima, Method II was particularly effective for improving performance in SPACE INVADERS.

4.6 Discussion

The results presented in this chapter support recent work showing that GAs are effective at training deep neural networks for RL. We took advantage of this to explore whether the behaviour of agents could be effectively used as selection pressure in an evolutionary search for RL policies. While our implementation of novelty search based on Levenshtein distance was not as effective as the Base GA, we found that it produced potentially useful and informative policies. In particular, we found that novelty search over Levenshtein distances is not equivalent to a longevity search, and that the policies it produces may be more defensive than than those produced by typical reward optimization.

The combination of reward signal and novelty scores in Method II resulted in a net improvement in testing scores over the Base GA in the four games tested. During SPACE INVADERS training it is particularly evident that, while the Base GA was showing signs of stagnation or convergence in validation performance, Method II effectively reoriented the search.

Method II yielded improved policies for MsPacman over both the Base GA and DQN method. On closer inspection, however, it is clear that all of the compared policies suffer from limited complexity. In no cases did we observe a successful strategy shift from escape to pursuit upon consumption of a pill. The lack of this emergent behaviour in any of the results we considered, in addition to sub-human performance in the current state-of-the-art based on a modified DQN architecture [8], leads us to suspect that the DQN architecture combined with reward-signal optimization is not well-suited for effectively learning situational policies or *discrete mode switching*. In response, we think that emerging methods such as Differentiable Inductive Logic Programming [5], a learning framework that enables logical rules to be inferred from large-scale data using neural networks, and a new wave of automated network architecture construction algorithms could be especially useful.

More broadly, the production and storage of policies with varying behaviours, including defensiveness, could have many applications in real-world control problems. In autonomous transport, for example, it could be desirable to evaluate potential policies with a wide range of behaviours in order to select the safest. Methods based on novelty search, like the ones introduced in this chapter, could be used to purposefully learn diverse strategies for achieving the same goals. This concept has recently been shown to be effective in learning frameworks based on a wide variety of methods — see [4] and [21], each of which use environment observations to help instil novelty, in addition to [17] and [20]. Methods that already implement observationally-based storage and comparison methods could benefit from the relatively low-cost inclusion of action sequences and string edit metric distances to diversify learned policies.

4.7 Future Work

The evolutionary algorithms community has developed and applied many methods for evolving network architectures and related structures. NEAT [24] and HyperNEAT [23] are very popular methods for simultaneously evolving network architectures and weights, and Cartesian Genetic Programming [14] is a related method that uses more general basis functions than are typically used in neural networks. All of these methods have been successfully applied in RL problems.

In future work, we will extend the methods detailed in this chapter to include automated

network architecture search. A method inspired by NEAT and that uses a compact algebraic approach to modular network representation [10] is currently in development. Given the scale at which Such et al.'s method enables GAs to train deep neural networks, we are optimistic that both existing and forthcoming methods for topology- and weight- evolving neural networks (TWEANNs) will be effective tools for solving increasingly complex problems in RL.

We are particularly eager to develop tools that combine the open-endedness of evolutionary algorithms with the reliability and robustness of functional modules, which could range from simple logical operators to convolutional network layers and beyond. Methods for searching the complex search space of deep neural network architectures and hyperparameters have recently been developed for gradient-based learning [18]. And though similar methods like HyperNEAT are certainly able to learn high-quality RL policies [22], we think a method that combines recent advances in both evolutionary algorithms and gradient-based deep reinforcement learning could be even more effective.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[3] Chollet, F. Keras, 2017.

[4] Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. Go-explore: a new approach for hard-exploration problems. *arXiv:1901.10995* (2019).

[5] Evans, R., and Grefenstette, E. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research 61* (2018), 1–64.

[6] Haapala, A. python-levenshtein, 2018.

[7] Hamming, R. W. Error detecting and error correcting codes. *Bell System technical journal 29*, 2 (1950), 147–160.

[8] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[9] Hughes, J., Houghten, S., and Ashlock, D. Recentering, reanchoring & restarting an evolutionary algorithm. In *Nature and Biologically Inspired Computing (NaBIC), 2013 World Congress on* (2013), IEEE, pp. 76–83.

[10] Jackson, E. C., Hughes, J. A., Daley, M., and Winter, M. An algebraic generalization for graph and tensor-based neural networks. In *Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), 2017 IEEE Conference on* (2017), IEEE, pp. 1–8.

[11] Kullback, S., and Leibler, R. A. On information and sufficiency. *The annals of mathematical statistics 22*, 1 (1951), 79–86.

[12] Lehman, J., and Stanley, K. O. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation 19*, 2 (2011), 189–223.

[13] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (1966), vol. 10-8, pp. 707–710.

[14] Miller, J. F. Cartesian genetic programming. *Cartesian Genetic Programming* (2011), 17–34.

[15] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[16] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., and Others. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[17] Mouret, J.-B., and Clune, J. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).

[18] Negrinho, R., and Gordon, G. DeepArchitect: Automatically Designing and Training Deep Architectures. *arXiv preprint arXiv:1704.08792* (2017).

[19] Oliphant, T. E. *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[20] Pugh, J. K., Soros, L. B., and Stanley, K. O. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI 3* (2016), 40.

[21] Savinov, N., Raichuk, A., Marinier, R., Vincent, D., Pollefeys, M., Lillicrap, T., and Gelly, S. Episodic curiosity through reachability. *arXiv preprint arXiv:1810.02274* (2018).

[22] Shenton, C. Atari 2600 leaderboard, 2018.

[23] Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life 15*, 2 (2009), 185–212.

[24] Stanley, K. O., and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation 10*, 2 (2002), 99–127.

[25] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv preprint arXiv:1712.06567* (2017).

[26] Sutton, R. S., and Barto, A. G. *Introduction to reinforcement learning*, vol. 135. MIT press Cambridge, 1998.

Chapter 5

Generative, Mutable Network Architectures for Deep Reinforcement Learning via Genetic Algorithms

5.1 Introduction

A recent surge of progress has been made in reinforcement learning (RL) largely due to the application of deep learning. Now-popular methods including DQN [12] and its many variants [3] have broken and maintain records in many benchmarks [15]. The hallmark feature of DQN is, perhaps, its ability to learn RL policies *directly from pixels* using convolutional neural network layers [9]. In recent work by Uber AI Labs, it was shown that a strictly gradient-free approach — a very simple but highly scalable genetic algorithm (GA) — can be used to effectively learn the weights of deep RL policy networks including DQN. This perhaps-surprising result prompts us to revisit other evolutionary approaches to RL including topology and weight evolving neural networks (TWEANNs).

TWEANNs have a long history of successful application in RL. Two of the most popular methods are neuroevolution of augmenting topologies (NEAT) [19] and hypercube NEAT (HyperNEAT) [17]. In general, a TWEANN is an evolutionary framework in which neural network architectures and their parameterizations are simultaneously evolved. In NEAT, architectures are initially small, and networks gradually become larger and more complex over evolutionary time. In RL problems with relatively small observation spaces — the spaces of inputs made available to agents — NEAT has been very successful [18; 22]. In most of these cases, observations are some *transformation* of the raw input space. For a video game, the raw input space typically consists of the observable pixels for each game frame [11], [1]. An input space transformation applies preprocessing steps to reduce the dimensionality and augment the information content of the raw inputs. For example, object detection, classification, and tracking could be used to summarize attributes of on-screen objects as feature vectors to be used as inputs instead of raw pixels. When such transformations are applied, NEAT has been more successful [22], [18]. Learning directly from raw pixels, on the other hand, had remained very challenging for RL researchers until the emergence of convolutional neural networks and, in particular, DQN [12].

Given that NEAT has been shown to be highly effective for learning RL policy networks from transformed features, and that DQN effectively learns such transformations via convolutional network layers, we think an exploration of methods that combines these approaches is well-warranted. In this chapter, we introduce a novel framework for RL that combines tensor-based deep learning and sparse matrix-based TWEANNs that is enabled by recent developments in highly-scalable GAs.

We developed this framework as a simple extension of the highly-scalable GA introduced by Such et al. in [20] that adds evolvable, generative network modules to densely-connected layers. The generative network modules are inspired by the objects and operations of SparseNALG — a highly general algebraic framework for representing and manipulating neural network architectures introduced in [6]. Using this approach, and in contrast to existing TWEANN frameworks like NEAT or HyperNEAT, a network's architecture can be seamlessly defined as a mix of layers with immutable and mutable architectures in any programming environment that supports both tensors and sparse matrices.

To evaluate the effectiveness of this approach, we implemented an instance of the framework that primarily uses PyTorch [14], NumPy [13], and Scipy [7] – all commonly-used Python packages in the deep learning and RL communities. We used series of experiments to explore the effects of enabling mutability in post-convolutional network layers for four games in the Atari 2600 benchmark [11] — a set of problems that remain challenging to most RL methods. The games used in this work are ASSAULT, ASTEROIDS, MSPACMAN, and SPACE INVADERS.

In short, we found that by enabling mutability in the last densely connected layer of the DQN network, learning performance was significantly better in three out of four games tested.

In Section 5.2 we give a brief overview of SparseNALG (for Sparse Neural Algebra) as an algebraic framework. Then in Section 5.3, we explain the limitations of employing SparseNALG directly and motivate the use of a procedural interpretation instead. In Section 5.4 we give an overview of Such et al.'s highly-scalable GA for deep RL on which our extension is based, and in Section 5.5 we describe the extension itself. Section 5.6 gives details about experiments and results using our implemented framework, Section 5.7 discusses the results, and directions for future work are discussed in Section 5.8.

5.2 Overview of SparseNALG

SparseNALG was developed in response to the lack of a standard, general framework for representing neural network architectures. It was inspired by Connection Set Algebra (CSA) [2], a mathematical framework for expressing connectivity structures in computational neuroscience, but provides an algebraic foundation that is much more general and extensible. Using SparseNALG, a large neural network architecture can be built using substitution operations that replace existing connections with arbitrary patterns. An example of such a substitution is visualized by Figure 5.1.

The objects and operations developed for SparseNALG are very useful for designing and reasoning about operations for manipulating the adjacency matrices of neural networks. However, we found that it would be difficult to apply the algebra directly in neuroevolution.

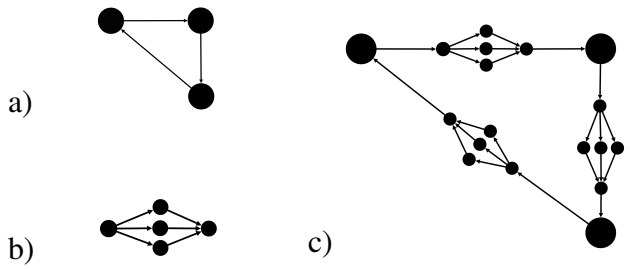

Figure 5.1: Application of a substitution operation in SparseNALG. a) A network architecture A. b) A connectivity pattern P. c) The result of substituting each connection in A by P. These substitutions can be performed using a combination of injection and projection operations on the adjacency matrix of A, which grows in size as new neurons are inserted.

5.3 Limitations of SparseNALG for Neuroevolution

SparseNALG is a formal algebra. Valid expressions must be composed of terms that satisfy the basic rules of the algebra. Neuroevolutionary algorithms often work by simultaneously searching the spaces of network topologies and weights. As such, an individual (genotype) in the evolutionary algorithm must encode all information needed to generate the corresponding phenotype. For TWEANNs, the genotype must then contain information that can be decoded to produce both the topology and weights of an artificial neural network. The genotype must also be mutable using genetic operations so that the search space can be explored.

Network topologies represented as algebraic expressions are compact, but not easily mutable. For example, consider the following simple expression:

$$subst(A, src, trg, P)$$

where A is the adjacency matrix of a network architecture, src and trg are source and target sets of neurons in the network represented by A, and P is a connectivity pattern that will be injected into A.

Such an expression could certainly be used as part of the genotype, but this would introduce added complexity, especially for genetic operators. A mutation operation, for example, would need to be an operation that modifies an algebraic expression. This would introduce added requirements of parsing the expression, representing it as an expression tree or graph, and selecting a *safe* operation and its operands. This approach would drastically increase the computational complexity of mutating the genotype because it would require the phenotype (the network) to be instantiated in order to check the validity or safety of mutations.

Many evolutionary frameworks for genetic programming (GP) [8] are implemented this way. For example, Hughes and Daley's framework for symbolic regression via GP mutates arithmetic expressions very similarly [4]. The key difference is that while the genotypes in [4] directly encode relatively small arithmetic expressions, the neural network topologies indirectly encoded by expressions in SparseNALG could be arbitrarily large. This motivates the use of genetic operations which explicitly do not require the phenotype to be instantiated for validation.

5.4 Highly Scalable Genetic Algorithms for Deep Reinforcement Learning

In [20], Such et al. introduced a compact, seed-based encoding for neural network weights that is highly scalable to parallel computing platforms. Instead of encoding network connection weights directly, the genome consists of a list of deterministic pseudo random number generator (PRNG) seeds. These seeds are used to generate vectors of mutation noise that are additively applied to initial network weights during network reconstruction. As such, genotype size scales with the number of generations instead of the number of network nodes or connections and greatly reduces communication overhead in a distributed implementation.

The compactness of this representation enables GAs to be applied to deep learning at very large scales. In [20] a large-scale GA was used to train DQN network weights in roughly 4 hours (wall-time) on a 720-core distributed system. The DQN policies resulting from this training were competitive with those learned using gradient-based algorithms that required several days of computing time. These results enable and prompt a serious investigation of the effectiveness of evolutionary algorithms applied at deep learning scales.

The GAs used in [20] and [5] were used to learn only the weights of the DQN architecture. In contrast, NEAT is a neuroevolutionary algorithm that also learns network topologies. It gradually searches the space of increasingly complex or augmenting topologies between set numbers of inputs and outputs via a process called *complexification*.

Using the GA and encoding introduced in [20] and objects and operations from SparseNALG, we can seamlessly combine architectural mutability inspired by NEAT with tensor-based deep learning. We call this framework EvoAlgNN for Evolving Algebraic Neural Networks.

5.5 EvoAlgNN

EvoAlgNN is an evolutionary framework based on SparseNALG, NEAT, and Such et al.'s GA for deep RL. It combines the seed-based encoding for neural network weights introduced in [20] with operations that enable arbitrary architecture mutability. EvoAlgNN is implemented in Python using PyTorch, NumPy, and SciPy — making it easily extensible and integrable with tensor-based deep learning, as is demonstrated in Section 5.6. The remainder of this section desribes the framework and implementation.

5.5.1 Mutable Subnetworks

The core object of EvoAlgNN is the *mutable subnetwork*. Formally, all mutable subnetworks are interpreted abstractly as relational matrices over the set $Inputs \oplus Outputs \oplus Hidden$ where $Inputs$ and $Outputs$ are non-empty, disjoint, ordered sets of neuron labels. The operator $\oplus$ is the *relational sum* — similar to the union of ordered sets, except that certain injection and projection properties must hold for the resulting objects. These properties are explained in detail in Chapter 3 and enable us to consistently organize adjacency matrices. Using this approach, any set of neurons can be interpreted as the relational sum of two smaller sets. This enables, for example, the set $Hidden$ to be interpreted as the disjoint union of two smaller sets Hid_1 and

Hid_2 and so on. Furthermore, this ensures that neural network inputs and outputs are consistently interpretable between reconstructed instances. Figure 5.2 visualizes an adjacency matrix over the relational sum of two sets.

	a_1	a_2	b_1	b_2
a_1	w_{a_1,a_1}	w_{a_1,a_2}	w_{a_1,b_1}	w_{a_1,b_2}
a_2	w_{a_2,a_1}	w_{a_2,a_2}	w_{a_2,b_1}	w_{a_2,b_2}
b_1	w_{b_1,a_1}	w_{b_1,a_2}	w_{b_1,b_1}	w_{b_1,b_2}
b_2	w_{b_2,a_1}	w_{b_2,a_2}	w_{b_2,b_1}	w_{b_2,b_2}

Figure 5.2: Adjacency matrix over the set $A \oplus B$, where $A = \{a_1, b_1\}$ and $B = \{b_1, b_2\}$ are disjoint, ordered sets. The relative order of elements in A and B are preserved by the adjacency matrix indices.

PyTorch does not currently support native matrix multiplication between sparse matrices. As such, mutable subnetworks are implemented in EvoAlgNN using SciPy sparse matrices. When a mutable subnetwork is reconstructed and ready to be evaluated, we refer to it as a *mutable layer* of a PyTorch model.

5.5.2 Operations for Network Mutability

EvoAlgNN uses operations taken from SparseNALG to incrementally build neural network architectures. In [20], Such et al. introduced the use of a list of seeds for a deterministic PRNG to encode the weights of a deep neural network. In this work, we use the same encoding to additionally and simultaneously encode the architectural operations, together with their parameters, which are applied to mutable network layers.

In this work, we investigate the effectiveness of enabling single-connection mutability and modular mutability in two separate experiments. During each generation of the evolutionary search, between 0 and 10 architectural operations may be applied to mutable layers.

Single-connection mutability implements a NEAT-like approach to architecture modification in which all substitutions replace single connections by two connections, which are connected via one new hidden neuron. Single-connection mutability is visualized by Figure 5.3.

Modular mutability enables more flexible and larger scale substitutions. Connectivity patterns available for substitution are called *primitives*. Primitives must be implemented as instances of a mutable subnetwork. This enables us to use a unified interpretation for mutable subnetworks at any level of organization.

In our experiments, we replaced the last layer of the DQN network with a mutable subnetwork. This network layer has 512 inputs and a variable number of outputs depending on which game is used. Using a relational sum-based interpretation, such a layer is initially encoded by an adjacency matrix over the set $Inputs \oplus Outputs \oplus Hidden$, where $|Inputs| = 512$, $|Outputs| = \#$ Game Actions, and $Hidden = \emptyset$. As substitution operations are applied, the $Hidden$ set grows, and new connections may be established between any neurons over the entire subnetwork. But because all primitives are also mutable subnetworks, the set $Hidden$ can be interpreted as the relational sum of all neurons added to the layer via substitutions. In other words, projection operations can always be applied to retrieve or identify neurons that belong

to the same primitive instance. More practically, this enables the adjacency matrix of an entire neural network layer to be stored as a single sparse matrix that supports arbitrary levels of mutability and organization. And crucially, this means that mutable network layers can be reconstructed using a sequence of operations applied to a single mutable subnetwork.

5.5.3 Network Reconstruction

The seed-based encoding used in EvoAlgNN requires network architectures and weights to be reconstructed before evaluation. The costs associated with this are proportional to the number of connections in the network instance at each of the generations elapsed in the evolutionary algorithm. In this work, network reconstruction is performed in 3 phases. First, the architectures of mutable network layers are reconstructed. For each seed in an individual's list, the seed is used to 1) determine whether architectural operations will be applied, 2) determine the number of operations to apply, 3) select operations to apply, and 4) select values for each operation's parameters. Next, the list of seeds is used to apply additive mutation noise to the reconstructed mutable layers. Lastly, the weights of immutable network layers (e.g. PyTorch convolutional layers) are reconstructed. The second and third steps both use the same approach used in [5] to apply additive mutation noise.

The architectural operations used in the first phase are directly taken from SparseNALG as described in Chapter 3. They consist of operations to create new neurons, establish connections between existing neurons, and substitute connections by patterns. Though the current implementation supports substitution by arbitrary patterns, the experiments in this work use single-connection substitution and one form of modular substitution. This is meant to provide a baseline result for enabling NEAT-like mutability in DQN network layers. A small-scale experiment using modular mutability is provided as a proof of concept.

Algorithm 1 Mutable Subnetwork Reconstruction

Input: Individual I, mutation power σ, complexification probability P_C, operations per complexification O_C, network initialization function *initNetwork*, deterministic PRNGs {*randomUniform*, *randomInteger*, *randomNormal*}
mutableSubnetwork ← *initNetwork(I.inputs, I.outputs, I.connectionRate, I.initialSeed)*
Seed all PRNGs with *I.initialSeed*
for *seed* in *I.generationSeeds* **do**
  if *randomUniform*() < P_C **then**
    for *increment* in [0 . . . *randomInteger*(O_C)] **do**
      Seed all PRNGs with *I.initialSeed* + *increment*
      op ← **Select** architectural operation using *randomInteger*
      params ← **Select** valid parameters for *op* using *randomInteger*
      Apply *op(params)* to *mutableSubnetwork*
for *seed* in *I.generationSeeds* **do**
  Seed all PRNGs with *seed*
  noise ← **Generate** noise vector scaled by σ using *randomNormal*
  mutableSubnetwork.weights ← *mutableSubnetwork.weights* + *noise*

5.5.4 Scalability

In contrast to other TWEANNs, EvoAlgNN genotypes scale in size with the number of generations as opposed to with the number of connections in the network. The compact, generative encoding enables very high scalability without imposing architectural immutability.

5.6 Experiments

To investigate the potential effectiveness of using combinations of immutable and mutable network layers in evolutionary searches for RL policy networks, we conducted experiments using four games from the Atari 2600 benchmark [11]. Using three variations of DQN learning [12], we applied EvoAlgNN to ASSAULT, Asteroids, MSPACMAN, and SPACE INVADERS. The first variation (Control) is a control condition using the DQN architecture, unmodified. The second (Single-Connection Mutability) replaces the last layer of the DQN architecture with a mutable EvoAlgNN layer in which only single-connection or NEAT-like architectural changes are possible with each generation. The third variation (Modular Mutability) extends this by enabling one additional substitution operation and one additional primitive or connectivity pattern.

5.6.1 Baseline and Architecture

All variations use an approximate replication of the GA introduced in [20] (Base GA). The Control variation in this work uses an unmodified version of the Base GA. This GA does not implement crossover and uses elitism to protect highly-generalizable individuals. The architecture in the Control experiment is an entirely immutable replication of the DQN architecture. It consists of three convolutional layers with 32, 64, and 64 filters, respectively, followed by one dense layer with 512 hidden units. The convolutional filter sizes are 8×8, 4×4, and 3×3, respectively. The strides are 4, 2, and 1, respectively. Network weights were initialized using Glorot normal initialization. Rectified linear unit (ReLU) activation is applied to all neuron outputs during evaluation. All game frames are downsampled to $84 \times 84 \times 4$ arrays. The third dimension encodes separate intensity channels for red, green, blue, and luminosity. Consecutive game observations are summed to address sprite flickering. Unlike the DQN method, no frame stacking is performed here, and the network must learn temporal dependency by incorporating recurrent connections.

5.6.2 Single-Connection Mutability

For the Single-Connection Mutability variation, we substituted the last layer of the DQN architecture with a mutable layer. Within this mutable layer, three architectural operations and one primitive are enabled. The architectural operations are 1) *create neuron*, 2) *create connection*, and 3) *substitute and fully connect*. The single primitive enabled consists of one input neuron, one internal or hidden neuron, and one output neuron. Together, these operations and primitive enable architectural mutations that are very similar to those in NEAT. Single-Connection substitution is visualized by Figure 5.3.

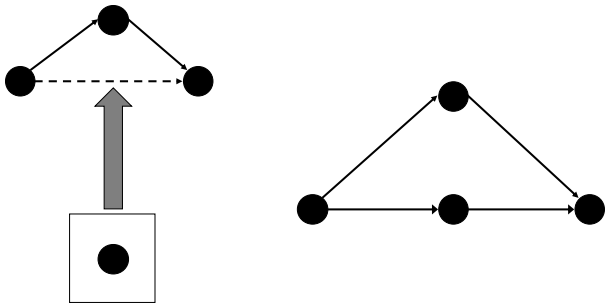

Figure 5.3: Single-Connection substitutions enabled by defining a single operation (*Substitute and fully connect*) and a single primitive (boxed). After this operation is applied, any of the four connections in the resulting mutable subnetwork are eligible for subsequent substitution.

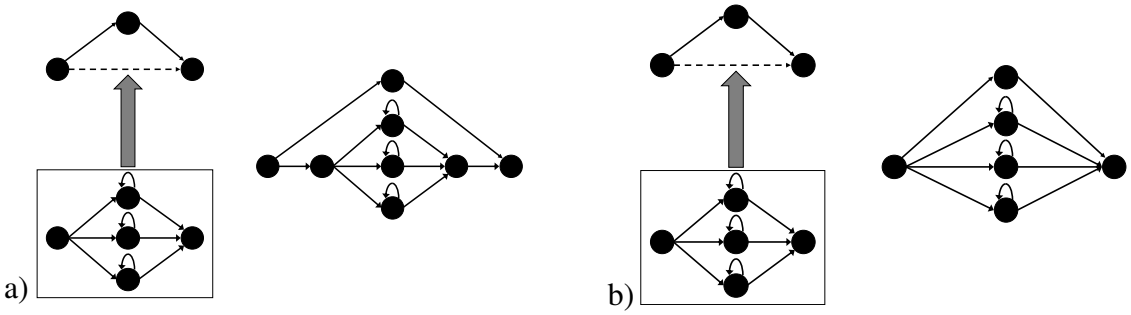

Figure 5.4: Substitution operations and the substitution primitive (boxed) implemented for the Modular Mutability experiment. *Substitute and fully connect* (a) and *direct substitution* (b) are used to replace a connection (dashed) with a primitive in two different ways. *Substitute and fully connect* connects the source node of the outgoing connection to all inputs of the substitution primitive and connects the target note of the outgoing connection to all outputs of the substitution primitive. *Direct substitution* replaces the source and target nodes of the outgoing connection with the input and output nodes of the substitution primitive, respectively. These operations and primitive were designed to demonstrate the flexibility that EvoAlgNN provides for defining architectural modification operations and substitution primitives.

In an effort to control for under- or over-parameterization compared to the Control variation, each run in the Single-Connection Mutability variation initializes the mutable subnetwork layer with 500 fewer connections than in the Control variation. This is because, with the selected hyperparameters, the expected number of connections that will be added by architectural operations is 250. As such, it is unlikely that any observed differences in performance will be due to over-parameterizing the experimental condition.

5.6.3 Modular Mutability

For the Modular Mutability variation, we enabled one additional architectural operation (*direct substitution*) and added one primitive featuring one input, one output, and three hidden neurons. The substitutions enabled by this variation are visualized by Figure 5.4. Otherwise, the experimental set-up and all hyperparameters are identical to the other two variants.

Hyperparameter	Value
Population Size (N)	100 + 1
Generations	500
Truncation Size (T)	20
Mutation Power (σ)	0.002
Complexification Probability	0.1
Operations Per Complexification	1 – 10
Max Frames Per Episode (F)	20,000
Training Episodes	1
Validation Episodes	30

Table 5.1: GA hyperparameters used in all experiments.

5.6.4 Experimental Set-up

We used a Microsoft Azure virtual machine with 64 cores and 128 GB of memory (Standard F64s_v2) for experiments. Experiments use OpenAIGym's implementation of the Atari 2600 benchmark. The DQN architecture and mutable subnetworks are implemented using a combination of modules from PyTorch, NumPy, and SciPy. Source code is available as part of the Digital Appendix.

5.6.5 Hyperparameters

The hyperparameters used in all experiments are summarized by Table 5.1. The relatively small population size and number of generations were chosen due to resource constraints. Despite this, we obtained policies that are competitive with related work. In future work, we will perform these experiments at larger scales.

Compared to the experiments in [5] and [20], this work introduces two new hyperparameters: *Complexification Probability* and *Operations Per Complexification*. The former determines, for each generation at network reconstruction, whether architectural operations will be applied. The latter then determines a range of possible numbers of operations to be applied at each such step. This range is uniformly sampled using the same generational seeds used in all other aspects of network reconstruction, thus ensuring deterministic, consistent network reconstruction.

5.6.6 Results

Experiments were designed to test whether architectural mutability could lead to improved learning performance using GAs over a static architecture. For comparability to other work [12], [20], [5], the baseline experiment uses the DQN architecture as introduced in [12] and a very simple and highly scalable GA introduced in [20]. The first experimental variation, Single-Connection Mutability, enables NEAT-like [19] mutability in the last, densely connected layer of the DQN network while the second variation, Modular Mutability, further enables a type

Game	Base GA	S-C	Modular	DQN
ASSAULT	530 (±160)	623 (±175)	774 (±190)†	**3359 (±775)**
ASTEROIDS	1146 (±609)	**2470 (±1045)**†	1493 (±533)	1629 (±542)
MSPACMAN	2750 (±889)	1821 (±734)	**3225 (±668)**†	2311 (±525)
SPACE INVADERS	964 (±226)	924 (±356)	738 (±302)	**1976 (±893)**

Table 5.2: Comparison of all experimental variation testing results over 30 episodes not used in training or validation. S-C denotes Single-Connection Mutability and Modular denoted Modular mutability. DQN results from 30 independent testing episodes are also reported directly from [12]. Means and standard deviations (shown in parentheses) are measured in game score units. Using two-tailed t-tests, experimental variations with mean testing scores higher than the Base GA with $p < 0.05$ are denoted by †. The best overall method when DQN is also considered is bolded. In two out of four games (ASTEROIDS and MSPACMAN), architectural mutability leads to better testing performance than the Base GA and the DQN method.

of modular mutability via more complex substitutions. Training progress for all experimental variations is visualized by Figure 5.5.

GAs with architectural mutability enabled achieved the highest testing scores in two out of four games (ASTEROIDS and MSPACMAN) when compared to both the Base GA and the DQN method. In three out of four games (ASSAULT, ASTEROIDS, and MSPACMAN) at least one of the methods with architectural mutability enabled achieved significantly better performance than the Base GA. In the fourth game, SPACE INVADERS, Single-Connection Mutability did not lead to significantly worse results than the Base GA. These results are summarized by Table 5.2.

Particularly in ASTEROIDS, Single-Connection Mutability led to impressive results, besting DQN and the other GAs in testing. The highest training score achieved by the population during training was 17450 points — higher than the top score of 13157 achieved by a professional (human) games tester as reported in [12] as a baseline. Though this performance did not fully generalize to validation or testing episodes, the GA run still resulted in the best testing performance among the compared methods.

Altogether, these results suggest that architectural mutability can significantly improve deep reinforcement learning policy network training using GAs.

5.7 Discussion

Architectural mutability in the last layer of the DQN network appears to dramatically improve learning over the Base GA and DQN ASTEROIDS. Interestingly, the number of connections that can possibly be added in the Single-Connection Mutability variation is directly proportional to the number of generations used in the GA. This is orders of magnitude smaller than the number of connections otherwise found in the densely connected layers of the DQN architecture. One possible explanation for this disproportionate effect is that recurrent connections may be established by architectural operations. Because the number of inputs (512) is much greater than the number of outputs in the last layer of the DQN architecture (e.g. 9), it is much more likely that architectural operations add connections between layer inputs, rather than between inputs

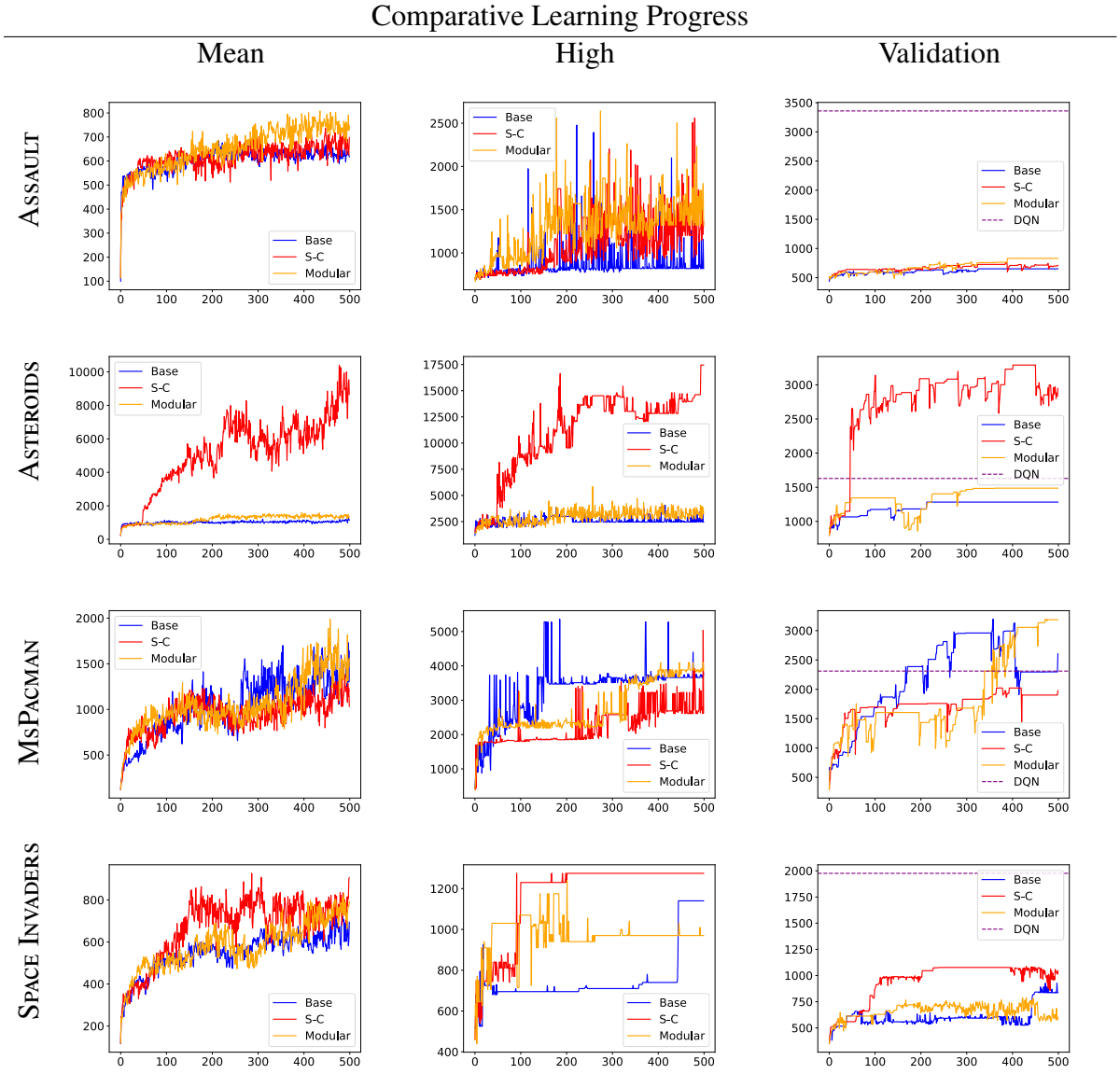

Figure 5.5: Comparative learning progress for the Base GA, Single-Connection Mutability, and Modular Mutability experiments. Mean denotes population mean game score over generations in training, high denotes score of top-performing individual over generations in training, and validation denotes the mean score of the best-generalizing individual to 30 differently-seeded environments. In each generation, the best individual in validation is designated as the elite. DQN testing results provided in [12] are shown against validation results as a dashed line. In two out of four games (ASTEROIDS and MSPACMAN, methods with architectural mutability enabled achieve higher scores than DQN in validation. This is in spite of the using a relatively small population (N=100+1). Single-Connection Mutability yielded a very large performance increase in ASTEROIDS over other methods.

and outputs. Recurrent connectivity, even at small scales, introduces temporal dependency that does not exist in DQN.

The scale of our experiments was small compared to other work that applies GAs to train deep neural networks. For example, in [20] and [5], population sizes were ten-times the size and the algorithm ran for twice as long. Since the number of architectural operations is directly proportional to the number of generations used in the GA, the number of possible architectures considered by the GA was also relatively small. This influenced our decision to only experiment with architectural mutability in the last densely connected layer of the DQN network. In MSPACMAN, a game with 9 possible actions, the last layer has $512 \times 9 = 4608$ connections. Over 500 generations, the expected number of connections added by the Single-Connection Mutability variant is 250. The results demonstrate that this relatively small number of modified connections can significantly change the behaviour and quality of policies.

As we increase the number of possible operations and primitives, so increases the size of the search space for the GA. Due to the scalability of the GA, however, it would be possible to run experiments with much larger search spaces as long as sufficient computing resources are available.

For the Modular Mutability variant, we designed a single primitive that featured three recurrent connections. There was otherwise no basis for the selection of this primitive. We also did not control for the larger number of connections it could evolve to have over the Base GA. Despite featuring possibly many more parameters, policies resulting from Modular Mutability performed best in only one game — MSPACMAN.

We cannot understate how valuable it is to have well-documented, extensible tools for working with tensors, including Tensorflow, Keras, PyTorch, and others. We found that PyTorch, in particular, provides a very convenient and intuitive interface for tensors without making restrictive assumptions about how they should be used. As demonstrated by our implementation and with the right mathematical interpretation, it is very straightforward to re-purpose PyTorch for gradient-free deep RL via a combination of architecturally immutable tensors and mutable sparse layers.

Sparse matrices are a crucial component for enabling architectural flexibility in deep learning. As we explore the use of larger, more complex neural network architectures, it is increasingly important that sparse matrix operations are efficient. The PyTorch community is actively developing improved support for sparse matrices with input from its users. High-performance computing projects like GraphBLAS are also developing increasingly accessible frameworks and interfaces for very general sparse matrices as a matter of priority.

In summary, we have presented a unified framework that combines conventionally opposed RL methodologies: deep learning and GAs. This combination is enabled by a solid mathematical interpretation of network architectures using a relation algebraic approach and a very pragmatic approach to highly-scalable neuroevolution using GAs.

5.8 Future Work

Our results show that architectural mutability can improve GA-based learning. This supports our initial hypothesis that TWEANNs could be combined with static convolutional layers to improve learning. This prompts us to consider whether higher degrees of architectural mutabil-

ity could further improve learning, or whether GAs and architectural mutability could be used to improve policy networks trained using other methods, including DQN and its many variants [3]. To better understand what kinds of architectural mutations may be most useful, we will use a variety of tools to quantify the relationships between network architectures. Velez and Clune introduced a method to identify functional modules within neural networks using a regression model [23], and we think this could be applied to identify useful primitives.

In future work, we will extract pre-trained convolutional layers from DQN and similar policy gradient networks. As a network module, convolutional network layers trained on one or possibly many games should learn to identify visual features that are relevant to achieving high quality policies that are reachable using gradient-based methods. We could then experiment with using GAs to learn the architecture and weights of post-convolutional network layers using the framework presented in this work. Going forward, we expect that combinations of gradient-based and gradient-free learning will contribute to driving progress in RL and other areas of machine learning. Evidence of likely success is demonstrated by recent results showing that evolutionary methods can be used to improve neural language translation models via evolutionary architecture search [16].

Looking ahead, we will also generalize this framework so that arbitrary functional modules can be used instead of only graph-like primitives. Existing methods including Cartesian Genetic Programming [10] offer insight into the effectiveness of using networks of interconnected functions for solving a variety of problems. Genetic Programming [8] has also been successfully applied to the design of convolutional neural networks [21]. We hypothesize that a framework that enables architectures to be manipulated at multiple levels of abstraction (e.g. single connections, network modules, functions, aggregate modules) could be both effective in machine learning applications and yield insight about the organization of more naturally-plausible neural networks.

Finally, the framework for architectural mutability introduced in this work could be applied in other contexts than RL. In future work, we will experiment with combinations of GA-based architectural mutability and gradient-based deep learning layers in a variety of contexts.

Bibliography

[1] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[2] Djurfeldt, M. The Connection-set Algebra—A Novel Formalism for the Representation of Connectivity Structure in Neuronal Network Models. *Neuroinformatics 10(3)* (2012).

[3] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[4] Hughes, J. A., and Daley, M. Finding nonlinear relationships in fmri time series with symbolic regression. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion* (2016), ACM, pp. 101–102.

[5] JACKSON, E. C., AND DALEY, M. Novelty search for deep reinforcement learning policy network weights by action sequence edit metric distance. *arXiv:1902.03142* (2019).

[6] JACKSON, E. C., HUGHES, J. A., DALEY, M., AND WINTER, M. An algebraic generalization for graph and tensor-based neural networks. In *2017 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)* (aug 2017), pp. 1–8.

[7] JONES, E., OLIPHANT, T., PETERSON, P., ET AL. SciPy: Open source scientific tools for Python, 2001–.

[8] KOZA, J. R. *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992.

[9] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[10] MILLER, J. F. Cartesian genetic programming. *Cartesian Genetic Programming* (2011), 17–34.

[11] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[12] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., AND OTHERS. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[13] OLIPHANT, T. E. *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[14] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch. In *NIPS-W* (2017).

[15] SHENTON, C. Atari 2600 leaderboard, 2018.

[16] SO, D. R., LIANG, C., AND LE, Q. V. The evolved transformer. *arXiv preprint arXiv:1901.11117* (2019).

[17] STANLEY, K. O., D'AMBROSIO, D. B., AND GAUCI, J. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life 15*, 2 (2009), 185–212.

[18] STANLEY, K. O., AND MIIKKULAINEN, R. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation* (2002), Morgan Kaufmann Publishers Inc., pp. 569–577.

[19] STANLEY, K. O., AND MIIKKULAINEN, R. Evolving neural networks through augmenting topologies. *Evolutionary computation 10*, 2 (2002), 99–127.

[20] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv preprint arXiv:1712.06567* (2017).

[21] Suganuma, M., Shirakawa, S., and Nagao, T. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2017), ACM, pp. 497–504.

[22] Togelius, J., Karakovskiy, S., Koutník, J., and Schmidhuber, J. Super mario evolution. In *2009 IEEE symposium on computational intelligence and games* (2009), IEEE, pp. 156–161.

[23] Velez, R., and Clune, J. Identifying core functional networks and functional modules within artificial neural networks via subsets regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (2016), ACM, pp. 181–188.

Chapter 6

General Discussion and Conclusions

The work presented in this thesis provides useful ideas and tools that can be used to explore open-ended, modular learning with artificial neural networks and neuroevolutionary algorithms. First, by providing a mathematical generalization for the expression of neural network architectures, we have set the stage for more principled exploitation of the connectionist relationship between them in different contexts. We then put this into practice via the development of a modular neuroevolution framework that supports substitution operations using arbitrary connectivity patterns. Finally, in a related effort to exploit the open-endedness that evolutionary algorithms can provide when combined with deep learning, we introduced an instance of novelty search that is generally applicable to any game or control problem with a discrete action space.

Each of these works provides a useful theoretical or practical result that can be extended or adapted to a variety of ends. Over the remainder of this chapter, we discuss this results in the context of past, present, and future endeavours in AI.

6.1 Interpretability and Modularity

Three decades later, we are still debating the same questions of interpretability that Hinton elucidated in the late 1980's and early 1990's [5]. How important is the interpretability of connectionist systems? Have we made much progress towards the interpretability of ANNs? I argue that as our understanding of effective modular network architectures has improved, we have at least gained better understanding into the reliability of networks, and also their functional organization.

In vision science, some computational neuroscientists have applied artificial convolutional models to the discovery and validation of biological models for early visual processing [3] and object recognition in the human brain [13]. These works add to the biological plausibility of contemporary ANN organization, at least for computer vision. By working together, computational scientists and neuroscientists will continue to gain insight into the relationship between artificial and biological computation or cognition.

The modularity of deep learning architectures has certainly helped to enable this line of work. The ability to design neural networks using modular components that are directly inspired by natural computation is a powerful feature of deep learning. What else might we

discover by identifying and relating functional modules in other contexts than vision? And what tools could we use to further our understanding of the relationship between biological and artificial neural networks?

Neural connectivity and broader functional connectivity has been studied in the brain from several perspectives. Sporns' work on graph- and network-theoretic analysis of neural connectivity patterns in the brain [14] could be applied to the same connectivity objects as Velez and Clune's methods for the identification of general functional modules in ANNs [16]. Neuromorphic computing projects such as SpiNNaker [6] or TrueNorth [1] could be used to define biologically plausible constraints at the hardware level, and may help us to change the way we think about machine learning algorithms.

More pragmatically, I think that our understanding of both biological and artificial neural network function will be driven by simulation and applications, respectively. The role of convolutions, for example, in ANNs is much better understood today than when they were beginning to be explored. As network modules, convolutions have been applied successfully in computer vision [7], reinforcement learning [9], natural language processing [8], and corresponding generative applications. By exploring the many uses of these modules in different application areas and under the constraints of different learning algorithms and complementing architectures, we have gained a much better understanding of their general function. Still, almost any non-trivial deep learning architecture succeeds in applications by learning successive non-linear transformations of inputs, thus creating representational spaces that are difficult to interpret — at least with respect to the raw inputs.

But is this really a problem? In situations where highly informative features are already available, for example responses to an expert-designed survey, deep learning is not likely to be a researcher or data scientist's first approach to predictive modelling. In such cases, interpretability with respect to the raw input space would likely be very important: we typically want to know exactly how the raw features contribute to classification decisions, for example. Conversely, when working with very granular input spaces such as those consisting of raw pixels or audio samples, we should not expect such decisions to be interpretable with respect to individual, isolated inputs. Instead, we gain much more information about the relationship between a model and its outputs by understanding its intermediary transformations of raw data into other representational spaces.

For the visual domain, we can easily visualize convolutional filters as images in an effort to interpret their function. More generally, the geometry of representational spaces can provide useful insight into neural network function. For example with word embeddings, representational spaces can be interpreted by considering the distances between words or groups of words. For images, representations of images after convolutional transformations have been applied can be interpreted for similarity or dissimilarity using a variety of metrics, thus describing the relationship between images with respect to learned transformations.

For reinforcement learning, we did not directly interpret the outputs of convolutionally transformed game frames. Instead, we hypothesized that the resulting representational space could be exploited by a neuroevolutionary algorithm to learn effective policy-producing network parameterizations. This is not fundamentally different from how we should expect DQN to work. Its convolutional layers are intended to *learn to see*, and the densely-connected layers that follow are intended to *learn to play* — by mapping points in a transformed representational space to actions such that the reward gradient is maximized.

By understanding the established, general function of convolutions as modules applied to visual input spaces, we were able to focus our research efforts on methodologies for *learning to play*. As we observed in Chapter 5, the application of a NEAT-like neuroevolutionary algorithm to an input space induced by convolutional transformations led to impressive RL policies, especially in ASTEROIDS. From this work, we gain evidence that the densely connected layers of the DQN architecture, in combination with reward gradient-based optimization, does not optimally leverage the transformed representational space induced by its convolutional layers. To conclude that this were the case, we should, in future work, fix the architecture and weights for convolutional layers using the DQN method, and use another method such as EvoAlgNN to find a better post-convolutional model than the one used in DQN. With new evidence that effectively random algorithms including GAs often result in better parameterizations of the same network than gradient-based optimization, we should be open-minded to re-characterizing neural search spaces and the methods we use to traverse them. In particular, until we have as clear an understanding of behavioural learning as we do for visual feature transformation, we should embrace methods that enable modular architecture search.

6.2 Open-Endedness in RL

In RL, we can plainly see that in some cases there are benefits to abandoning conventional objectives. Novelty search and related methods to promote behavioural diversity can help to overcome the problems of sparse reward or deceptive local optima. One of the major benefits of evolutionary algorithms are their open-endedness with respect to their objective functions. It is possible that through experimentation with different notions of selection pressure or fitness, we could discover new ways to train artificial agents in complex environments. Just as biological organisms did not develop flight via an optimization process promoting it directly, DQN and other gradient-based methods seem unable to learn truly complex behaviours in RL by optimizing for a single reward signal. See DQNs performance on MONTEZUMAS REVENGE, for example [9].

In contrast, relatively straightforward gradients work well for training neural networks in many domains. Convolutional network layers learn effective transformations of their input spaces by optimizing for compactness, mutual information, reproducibility, or other easily expressible, differentiable objective functions. The current dominance of deep convolutional neural networks in the visual domain is staggering and inspires confidence that gradient-based convolutional filters are highly effective in computer vision.

The structures and algorithms that enable deep neural networks to develop visual acuity, however, are not directly as powerful for learning complex behaviours. Again, *learning to see* could rely on a different process than *learning to play*. From our own experiences as humans, we know that we continue to learn new behaviours long after our visual acuity has effectively completed development, suggesting that different processes, or at least different levels of plasticity, are involved.

With this in mind, the success of DQN in the Atari 2600 benchmark may be due more to the relative logical simplicity of the games, and dually, to the close connection between successful gameplay strategies and computable visual features. In BREAKOUT, for example, an optimal policy can be computed in terms of very few environment observations, and these are directly

provided by the transformations of convolutional layers. For more complex games, it could be beneficial to learn visual processing and behaviour separately and with more of an open-ended approach in mind.

In future work, we will combine gradient-based and gradient-free learning in a multi-phase learning process. A convolutional autoencoder could be used, for example, to learn representations of the raw, visual input space. Alternatively, a block of convolutional modules pre-trained by following locally-optimal gradients could be generated. Then, separately, deep neuroevolution or other algorithms (e.g. Go-Explore) could be applied to learn a recurrent mapping between compressed visual representations and actions. At this stage, we point to the work in this thesis and the work on which it is based to support the claim that a series of two densely connected, feed-forward network layers trained by reward gradient optimization is most likely not capable of learning truly generalizable RL policies.

Though neuroevolutionary algorithms are not being proposed as a singular solution to this problem, they do enable researchers to consider less constrained ways of traversing neural search spaces. Novelty search is just one example of how a not-necessarily-differentiable objective function can be used to learn useful policies. And with methods like EvoAlgNN and related evolutionary neural architecture search more broadly being actively developed, we now have the beginnings of tools in place for discovering effective neural architectures for increasingly complex problems.

6.3 Conclusions

Behaviour-influenced learning and neural architecture (NAS) search have shown to be very promising in RL [4; 11; 15]. Chapters 4 and 5 further demonstrate that evolutionary algorithms (EAs) can be used in conjunction with more conventional methods including tensor-based deep learning. Chapter 5, in particular, shows that evolutionary NAS can occur at multiple levels of organization or modularity.

With all of this progress, we are far from understanding a general theory of learning with artificial neural networks. It is clear that while reward gradients lead to optimal policies in some contexts, current RL algorithms do not encompass a complete theory of environmentally-derived learning. In other words, it is clear that the reward sources and algorithms used in contemporary benchmarks alone do not constitute a satisfying theory of learning, especially for complex behaviours. We can plainly see this in cases where reward-seeking optimization fails to overcome either sparse reward or deceptive local optima. The Atari 2600 game Montezuma's Revenge features both, and conventional RL methods, including neuroevolution, completely fail to learn effective policies for this game.

EAs might be used to directly or indirectly improve this situation. As explored in Chapter 4, EAs enable the use of nearly arbitrary, non-differentiable objective functions to provide fitness measures. This kind of open-endedness enables alternative formulations of reward in RL contexts. Novelty search and related algorithms like episodic curiosity [11] and Go-Explore [2] have helped to demonstrate that RL policies can often be improved by incorporating agent behaviour into optimization criteria. More generally, by loosening the constraints on the kinds of optimization criteria that can be used, we allow ourselves more flexibility and creativity in designing algorithms. Go-Explore, in particular, is a great example of this.

It is exciting to see that the application of evolutionary algorithms to the design of deep learning architectures is becoming more popular — with research being published by world-class research groups [10; 12]. In future work, we will continue to develop tools that enable generative, modular neural networks to be combined with other deep learning objects. In a similar spirit as the computational neuroscience community's embrace of state-of-the-art methods in AI [13; 3], we will work to integrate developing knowledge of brain organization into highly modular, generative frameworks. Considering Sporns' consistent, mathematically-derived observations of self-similar connectivity patterns at multiple levels of organization in the human brain, we think it will be particularly important to support modularity at multiple levels of organization in AI as well. In light of the success of convolutional modules in computer vision, we are optimistic that the architectures and objectives necessary to reach the same levels of accomplishment in behaviour learning might be discovered in part by evolutionary frameworks.

Bibliography

[1] CASSIDY, A. S., MEROLLA, P., ARTHUR, J. V., ESSER, S. K., JACKSON, B., ALVAREZ-ICAZA, R., DATTA, P., SAWADA, J., WONG, T. M., FELDMAN, V., AND MODHA, D. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. *The 2013 International Joint Conference on Neural Networks (IJCNN)* (2013).

[2] ECOFFET, A., HUIZINGA, J., LEHMAN, J., STANLEY, K. O., AND CLUNE, J. Go-explore: a new approach for hard-exploration problems. *arXiv:1901.10995* (2019).

[3] EICKENBERG, M., GRAMFORT, A., VAROQUAUX, G., AND THIRION, B. Seeing it all: Convolutional network layers map the function of the human visual system. *NeuroImage 152* (2017), 184–194.

[4] HAUSKNECHT, M., KHANDELWAL, P., MIIKKULAINEN, R., AND STONE, P. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation* (2012), ACM, pp. 217–224.

[5] HINTON, G. E. Preface to the special issue on connectionist symbol processing. *Artificial Intelligence 46*, 1-2 (1990), 1–4.

[6] KHAN, M. M., LESTER, D. R., PLANA, L. A., RAST, A., JIN, X., PAINKRAS, E., AND FURBER, S. B. SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)* (jun 2008), pp. 2849–2856.

[7] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[8] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[9] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., and Others. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[10] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548* (2018).

[11] Savinov, N., Raichuk, A., Marinier, R., Vincent, D., Pollefeys, M., Lillicrap, T., and Gelly, S. Episodic curiosity through reachability. *arXiv preprint arXiv:1810.02274* (2018).

[12] So, D. R., Liang, C., and Le, Q. V. The evolved transformer. *arXiv preprint arXiv:1901.11117* (2019).

[13] Spoerer, C. J., McClure, P., and Kriegeskorte, N. Recurrent convolutional neural networks: a better model of biological object recognition. *Frontiers in psychology 8* (2017), 1551.

[14] Sporns, O. Small-world connectivity, motif composition, and complexity of fractal neuronal connections. *Biosystems 85*, 1 (2006), 55–64.

[15] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv preprint arXiv:1712.06567* (2017).

[16] Velez, R., and Clune, J. Identifying core functional networks and functional modules within artificial neural networks via subsets regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (2016), ACM, pp. 181–188.

Appendix A

Genetic Algorithms Pseudocode

Algorithm 2 Base GA

Input: mutation function ψ, population size N, number of generations G, truncation size T, individual initializer ϕ, individual decoder γ, fitness function F, training episodes E_t, validation episodes E_v, deterministic uniform PRNG U.

population $\leftarrow$ []
for $i = 1, 2, \ldots, N$ **do**
    Append $\phi(U(0, 2^{32} - 1))$ to *population*
for $g = 1, 2, \ldots, G$ **do**
    policies $\leftarrow$ map(γ, *population*)
    trainingResults $\leftarrow F(E_t$, *policies*)
    Sort *trainingResults* by game score
    eliteCandidates $\leftarrow$ 10 best in *trainingResults*
    validationResults $\leftarrow F(E_v$, *eliteCandidates*)
    Sort *validationResults* by game score
    elite $\leftarrow$ 1 best in *validationResults*
    Save *elite* to disk
    parents $\leftarrow T$ best in *trainingResults*
    if $g < G - 1$ **then**
        newPopulation $\leftarrow$ [*elite*]
        for $p = 1, 2, \ldots N - 1$ **do**
            parent $\leftarrow$ *parents*$[U(0, T - 1)]$
            Append ψ(*parent*) to *newPopulation*
        population $\leftarrow$ *newPopulation*

Algorithm 3 Method I - Novelty Search

Input: mutation function ψ, population size N, number of generations G, truncation size T, individual initializer ϕ, individual decoder γ, fitness function F, training episodes E_t, validation episodes E_v, deterministic uniform PRNG U, archive insertion probability p, novelty function η.

population $\leftarrow$ []
for $i = 1, 2, \ldots, N$ **do**
    Append $\phi(U(0, 2^{32} - 1))$ to *population*
$A \leftarrow$ []
for $g = 1, 2, \ldots, G$ **do**
    policies $\leftarrow$ map(γ, *population*)
    trainingResults $\leftarrow F(E_t,$ *policies*)
    for (*ind*, *gameScore*, *BC*) in *trainingResults* **do**
        Append *BC* to A with probability p
    nScores $\leftarrow$ map($\eta(A)$, *trainingResults*)
    Sort *trainingResults* by novelty score
    eliteCandidates $\leftarrow$ 10 best in *trainingResults*
    validationResults $\leftarrow F(E_v,$ *eliteCandidates*)
    Sort *validationResults* by game score
    elite $\leftarrow$ 1 best in *validationResults*
    Save *elite* to disk
    parents $\leftarrow T$ most novel in *trainingResults*
    if $g < G - 1$ **then**
        newPopulation $\leftarrow$ [*elite*]
        for $p = 1, 2, \ldots N - 1$ **do**
            parent $\leftarrow$ *parents*$[U(0, T - 1)]$
            Append ψ(*parent*) to *newPopulation*
        population $\leftarrow$ *newPopulation*

Algorithm 4 Method II - Stagnation Detection and Population Resampling

Input: mutation function ψ, population size N, number of generations G, truncation size T, individual initializer ϕ, individual decoder γ, fitness function F, training episodes E_t, validation episodes E_v, deterministic uniform PRNG U, archive insertion probability p, novelty function η, number of improvement generations IG

population $\leftarrow$ []
for $i = 1, 2, \ldots, N$ **do**
    Append $\phi(U(0, 2^{32} - 1))$ to *population*
vScores $\leftarrow$ []
for $g = 1, 2, \ldots, G$ **do**
    policies $\leftarrow$ map(γ, *population*)
    trainingResults $\leftarrow F(E_t$, *policies*)
    for (*ind*, *gameScore*, *BC*) in *trainingResults* **do**
        Append *BC* to *A* with probability *p*
    Sort *trainingResults* by game score
    eliteCandidates $\leftarrow$ 10 best in *trainingResults*
    validationResults $\leftarrow F(E_v$, *eliteCandidates*)
    Sort *validationResults* by game score
    elite $\leftarrow$ 1 best in *validationResults*
    Save *elite* to disk
    Append *elite* validation score to *vScores*
    parents $\leftarrow T$ best in *trainingResults*
    if *vScores.length* $\geq IG$ **then**
        progress $\leftarrow$ []
        for $i = g - IG + 1, g - IG + 2, ...g$ **do**
            Append *vScores*[i] - *vScores*[$g - IG$] to *progress*
        if $\forall x$ in *progress*, $x <= 0$ **then**
            noveltyResults $\leftarrow$ map(η(*trainingResults*), *A*)
            Sort *noveltyResults* by novelty score
            parents $\leftarrow T$ most novel in *noveltyResults*
            vScores $\leftarrow$ []
    if $g < G - 1$ **then**
        newPopulation $\leftarrow$ [*elite*]
        for $p = 1, 2, \ldots N - 1$ **do**
            parent $\leftarrow$ *parents*[$U(0, T - 1)$]
            Append ψ(*parent*) to *newPopulation*
        population $\leftarrow$ *newPopulation*

Curriculum Vitae

Name:	Ethan Jackson
Post-Secondary Education and Degrees:	Honours BSc Computer Science and Mathematics Brock University 2008 – 2012
	MSc Computer Science Brock University 2012 – 2014
	PhD Computer Science (*candidate*) University of Western Ontario 2014 – Present
Honours and Awards:	Ontario Graduate Scholarship 2014, 2015, 2016, 2017
	Vector Institute Postgraduate Affiliate Fellowship 2018, 2019
Related Work Experience:	Teaching and Research Assistant (Computer Science) University of Western Ontario 2014 – Present
	Limited Duties Faculty Lecturer University of Western Ontario 2016 – 2018
	Graduate Research Assistant (Neuroscience) University of Western Ontario 2017 – Present

Publications:

2019 | **Jackson, E.C.** and Daley, M. "*Novelty Search for Deep Reinforcement Learning Policy Network Weights by Action Sequence Edit Metric Distance*", 2019 Genetic and Evolutionary Computation Conference (GECCO 2019, accepted)

2018 | **Jackson, E.C.**, Hughes, J.A., Daley, M. "*On the Generalizability of Linear and Non-Linear Region of Interest-Based Multivariate Regression Models for fMRI Data"*, IEEE International Conference on Computational Intelligence in Bioinformatics and Computational Biology, Special Session on Neuroinformatics (IEEE CIBCB 2018), at St. Louis, USA.

MacCannell, A.D.V., **Jackson, E.C.**, Mathers, K.E. and Staples, J.F. "*An Improved Method for Detecting Torpor Entrance and Arousal in a Mammalian Hibernator Using Heart Rate Data*", Journal of Experimental Biology, jeb-174508.

2017 | Hughes, J.A., **Jackson, E.C.**, Daley, M. "*Modelling Intracranial Pressure with Noninvasive Physiological Measures*", IEEE International Conference on Computational Intelligence in Bioinformatics and Computational Biology (IEEE CIBCB 2017), at Manchester, England.

Jackson, E.C., Hughes, J.A., Daley, M., Winter, M. "*An Algebraic Generalization for Graph and Tensor-Based Neural Networks*", IEEE International Conference on Computational Intelligence in Bioinformatics and Computational Biology (IEEE CIBCB 2017), at Manchester, England.

2016 | Winter, M., **Jackson, E.C.** "*Categories of Relations for Variable-Basis Fuzziness*", Fuzzy Sets and Systems, Volume 298 Issue C, Pages 222–237.

2014 | **Jackson, E.C.** "*L-Fuzzy Relations in Coq*", M.Sc. Thesis, Brock University.

Winter, M., **Jackson, E.C.**, Fujiwara, Y. "*Type-2 Fuzzy Controllers in Arrow Categories*", Relation and Algebraic Methods in Computer Science (RAMiCS 2014), at Marienstatt, Germany.