Evolving principes of artificial neural design

Dennis G. Wilson

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Dennis WILSON

Le 4 mars 2019

Évolution des principes de la conception des réseaux de neurones artificiels

---

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Hervé LUGA et Sylvain CUSSAT-BLANC

Jury

M. Marc Schoenauer, Rapporteur
M. Keith Downing, Rapporteur
Mme Una-May O'Reilly, Examinatrice
Mme Sophie Pautot, Examinatrice
Mme Anna Esparcia-Alcázar, Examinatrice
Mme Emma Hart, Examinatrice
M. Hervé LUGA, Directeur de thèse
M. Sylvain Cussat-Blanc, Co-directeur de thèse

Evolving Principles of Artificial Neural Design

Dennis G. Wilson

February 28, 2019

Abstract

The biological brain is an ensemble of individual components which have evolved over millions of years. Neurons and other cells interact in a complex network from which intelligence emerges. Many of the neural designs found in the biological brain have been used in computational models to power artificial intelligence, with modern deep neural networks spurring a revolution in computer vision, machine translation, natural language processing, and many more domains.

However, artificial neural networks are based on only a small subset of biological functionality of the brain, and often focus on global, homogeneous changes to a system that is complex and locally heterogeneous. In this work, we examine the biological brain, from single neurons to networks capable of learning. We examine individually the neural cell, the formation of connections between cells, and how a network learns over time. For each component, we use artificial evolution to find the principles of neural design that are optimized for artificial neural networks. We then propose a functional model of the brain which can be used to further study select components of the brain, with all functions designed for automatic optimization such as evolution.

Our goal, ultimately, is to improve the performance of artificial neural networks through inspiration from modern neuroscience. However, through evaluating the biological brain in the context of an artificial agent, we hope to also provide models of the brain which can serve biologists.

Résumé

Le cerveau biologique est composé d'un ensemble d'éléments qui évoluent depuis des millions d'années. Les neurones et autres cellules forment un réseau complexe d'interactions duquel émerge l'intelligence. Bon nombre de concepts neuronaux provenant de létude du cerveau biologique ont été utilisés dans des modèles informatiques pour développer les algorithmes dintelligence artificielle. C'est particulièrement le cas des réseaux neuronaux profonds modernes qui révolutionnent actuellement de nombreux domaines de recherche en informatique tel que la vision par ordinateur, la traduction automatique, le traitement du langage naturel et bien d'autres.

Cependant, les réseaux de neurones artificiels ne sont basés que sur un petit sous-ensemble de fonctionnalités biologiques du cerveau. Ils se concentrent souvent sur les fonctions globales, homogènes et à un système complexe et localement hétérogène. Dans cette thèse, nous avons d'examiner le cerveau biologique, des neurones simples aux réseaux capables d'apprendre. Nous avons examiné individuellement la cellule neuronale, la formation des connexions entre les cellules et comment un réseau apprend au fil du temps. Pour chaque composant, nous avons utilisé l'évolution artificielle pour trouver les principes de conception neuronale qui nous avons optimisés pour les réseaux neuronaux artificiels. Nous proposons aussi un modèle fonctionnel du cerveau qui peut être utilisé pour étudier plus en profondeur certains composants du cerveau, incluant toutes les fonctions conçues pour l'optimisation automatique telles que l'évolution.

Notre objectif est d'améliorer la performance des réseaux de neurones artificiels par les moyens inspirés des neurosciences modernes. Cependant, en évaluant les effets biologiques dans le contexte d'un agent virtuel, nous espérons également fournir des modèles de cerveau utiles aux biologistes.

Acknowledgements

A thesis can sometimes appear a solitary endeavor and is certainly a reflection of the author's interests, methods, and understandings. In truth, this thesis has been anything but solitary, with numerous actors influencing not only the work presented in this thesis, but also myself, and my own interests, methods, and understandings. I want to acknowledge a select few, although many others remain unacknowledged but greatly appreciated.

My advisors, Sylvain Cussat-Blanc and Hervé Luga, have shaped, supported, and challenged every idea in this document, working with me tirelessly to guide my sometimes circuitous exploration of interests. They have also supported and challenged me as a person, guiding my growth and change these past three years. It isn't simple moving to a new continent and adapting to a new university, culture, and bureaucracy, and I've only made it to this final stage of my thesis thanks to their extensive and comprehensive support.

Along the way, I was fortunate to gain another advisor in all but name, Julian Miller. He has been a source of insight in our collaborations, and his passion for researching interesting topics, irrespective of their current difficulty or popularity, has inspired me and encouraged my own research directions.

The jury of this thesis have all also influenced it and me in various ways. Keith Downing's book, *Intelligence Emerging*, set the direction for much of this thesis and encouraged my interest in artificial life. My first experience with live neurons was in Sophie Pautot's lab, where I learned how much of a mystery neurons still are. The works of Marc Schoenauer, Emma Hart, and Anna Esparcia have all inspired and informed me, and a motivation to see and maybe impress them has pushed a number of the GECCO articles in this thesis through to completion. Finally, none of this would have happened without Una-May O'Reilly, who took in a somewhat lost sophomore, showed me the marvels of bio-inspired computing, and encouraged me to pursue a PhD in this field.

To all of the above, I express my deep gratitude for their impact on this thesis, whether direct or indirect, and on me. I can only hope to someday impact the research and life of another as they have mine.

Contents

1 Introduction **11**

1.1 The brain as a model . . . 14
1.2 Evolving emergent intelligence . . . 15
1.3 Organization of the thesis . . . 17

2 Background **19**

2.1 Neural cell function . . . 21
2.1.1 Biological neural models . . . 22
2.1.2 Activation functions . . . 24
2.1.3 Other cell behavior in the brain . . . 26
2.2 Neural connectivity . . . 26
2.3 Learning in neural networks . . . 29
2.3.1 Spike Timing Dependent Plasticity . . . 30
2.3.2 Gradient Descent and Backpropagation . . . 32
2.4 Evolutionary computation . . . 33
2.4.1 Evolutionary strategies . . . 34
2.4.2 Genetic Algorithms . . . 34
2.4.3 Genetic Programming . . . 36
2.5 Evolving artificial neural networks . . . 36
2.6 Objectives of the thesis . . . 38

3 Evolving controllers **41**

3.1 Artificial Gene Regulatory Networks . . . 42
3.1.1 AGRN applications . . . 43
3.1.2 AGRN overview . . . 45
3.1.3 AGRN dynamics . . . 47
3.1.4 AGRN experiments . . . 50
3.1.5 AGRN results . . . 52
3.2 Cartesian Genetic Programming . . . 56

3.2.1 CGP representation . . . 59
3.2.2 Playing games with CGP . . . 60
3.2.3 Positional Cartesian Genetic Programming . . . 65
3.2.4 Genetic operators for CGP . . . 66
3.2.5 CGP experiments . . . 68
3.2.6 CGP method comparison . . . 69
3.2.7 CGP parameter study . . . 71
3.3 Conclusion . . . 74

4 Evolving neural cell function **77**
4.1 Spiking neural activation functions . . . 78
4.2 Neural network model . . . 81
4.3 Experiment . . . 83
4.3.1 Clustering tasks . . . 83
4.3.2 Network . . . 84
4.3.3 Training . . . 84
4.3.4 Evolution . . . 85
4.4 Results . . . 86
4.5 Conclusion . . . 87

5 Evolving developmental neural connectivity **91**
5.1 Biological axon development . . . 93
5.2 Axon guidance model . . . 97
5.2.1 Cellular models . . . 97
5.2.2 Environment initialization . . . 99
5.2.3 Environment update . . . 99
5.2.4 Model configuration and evolution . . . 101
5.3 Eye-specific patterning . . . 102
5.3.1 Visual system environment . . . 102
5.3.2 Evolutionary results . . . 104
5.4 Robot coverage . . . 108
5.5 Conclusion . . . 111

6 Evolving learning methods **113**
6.1 Reward-Modulated Spike-Timing Dependent Plasticity . . . 117
6.1.1 Neuron and learning models . . . 118
6.1.2 Neuromodulation reward model . . . 119
6.1.3 Instrumental conditioning . . . 121
6.1.4 Aquatic Locomotion Problem . . . 123

6.1.5 Evolution of neuromodulation method . . . 125
6.1.6 Evolution results . . . 126
6.1.7 Summary of Reward-Modulated STDP . . . 128
6.2 Neuromodulation of learning parameters in deep neural networks . . . 128
6.2.1 AGRN neuromodulation model . . . 130
6.2.2 Evolution of the neuromodulatory agent . . . 132
6.2.3 Comparison of neuromodulation to standard optimization . . . 135
6.2.4 Generalization of the neuromodulatory agent . . . 136
6.2.5 Neuromodulation behavior . . . 138
6.2.6 Summary of neuromodulation of learning parameters in deep ANNs 139
6.3 Conclusion . . . 141

7 Discussion and conclusion **143**
7.1 A framework for developmental neuroevolution . . . 146
7.2 Evolving to learn for data classification . . . 149
7.3 The evolution of learning . . . 150
7.4 Conclusion . . . 153

Chapter 1

Introduction

Chlamydomonas reinhardtii is a type of single-celled freshwater green alga. This simple organism uses two flagella to propel itself through water, guided by photosensory, mechanosensory, and chemosensory cues. Membrane receptors located in the cell body detect sensory input, which leads to membrane depolarization and moves the cilia located on the cell body. This sensor-actuator system allows *C. reinhardtii* to navigate through ever-changing environments. The mechanism responsible for sensing the environment and sending information to other processes in the cell is the transient receptor potential (TRP) channel. These ion channels in *C. reinhardtii* share features with the TRP channels used by mammals in the sensory system [Ari+15]. This suggests that the TRP channel gating characteristics, used in the human brain, evolved early in the history of eukaryotes.

Single-celled organisms have also been shown to be capable of learning and having memory. This has mostly been demonstrated in ciliates, organisms which move using small hair-like cilia on the outside of the cell. In [Woo88], Wood showed that repetitive mechanical stimulation of *Stentor* leads to contraction becoming more and more common in the cell. The contraction response following the first input of mechanism stimulation disappeared after this input was gone, but the memory of the event was intact. When inputs of the same type were given, the contraction response was facilitated. *Paramecium*, a well-known freshwater ciliate, has been shown to be capable of learning to associate between light and electrical stimulations [AMJ06].

In unicellular organisms, sensors and actuators are part of the same cell, while in multicellular neuronal organisms the two components reside in different, sometimes very distant, cells. Nevertheless, in both cases the molecular machinery underlying the learning phenomena are basically the same [GJ09]. The cellular mechanisms that serve as building blocks of the brain were created by evolution long before the first organism with a brain, even before the first mammals. The cells of the brain, neurons, oligodendrocytes, astrocytes, and others, have evolutionary precursors and use mechanisms shared by other

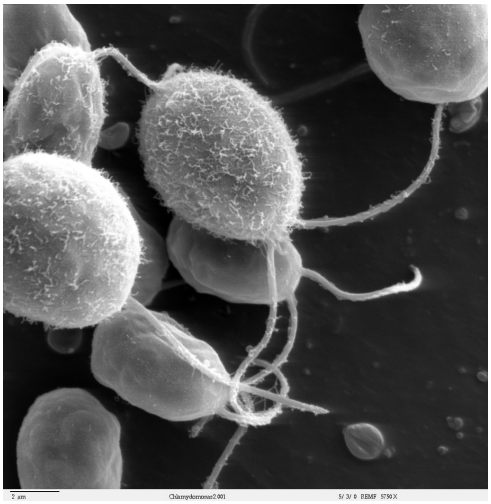

Figure 1.1: Scanning electron microscope image of *Chlamydomonas reinhardtii*, a unicellular flagellate used as a model system in molecular genetics work and flagellar motility studies.

cells and other organisms. Their form in the mammal and human brain is the result of a long evolutionary process of refinement, even before they were composed into nervous systems.

In neural organisms, those with a nervous system, neurons act as the medium of information transportation from sensors to actuators, as well as the regulators of biological processes. In *C. elegans*, a microscopic worm, the nervous system is used to control the movement of the worm and for triggering digestive processes, and uses information from the digestive system to influence the worm's movement, deciding whether it is roaming to find new food or grazing in a specific spot. The *C. elegans* nervous system is shown in Figure 1.2. While the acts of this nervous system are far from meeting modern definitions of intelligence, the brain of the *C. elegans* resembles our own in many ways [IB18]. Evolution has selected neural designs for the worm suited for its needs and environment by shaping the worm's genes, which in turn decide how the worm's neurons develop, act, connect, and learn.

The neural designs created by evolution are as remarkable as they are a mystery. A bee with only 10^6 neurons can learn to break camouflage, navigate a maze via symbolic cues, and perform cognitive tests that were thought, until recently, impossible except for in monkeys, humans, dolphins, and pigeons [SL15]. To do this, the relatively small number of cells must be used efficiently, and the neural designs found in biology use a variety of information passing mechanisms to ensure the best use of these cells.

Neurons primarily communicate with electrical signals, by sending electrical spikes down long, branching axons similar to insulated wires. This rapid information transfer allows for information to travel through the brain at 1 mm per ms [SL15]. Chemical signals, the release of neurotransmitters by specific cells, are used when information can be

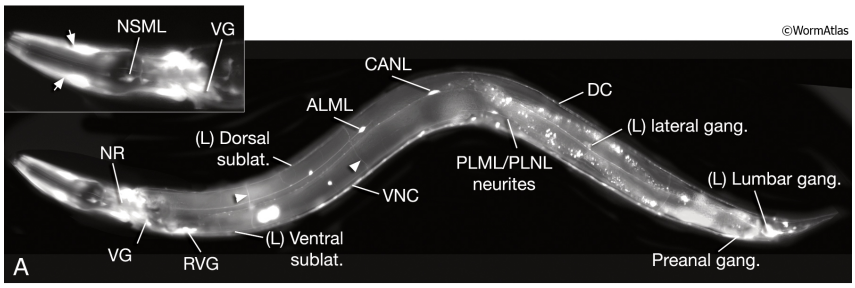

Figure 1.2: The *C. elegans* nervous system. The nerve ring (NR) hosts the largest collection of neurons, including the The largest collection of neurons is around the nerve ring (NR) within several head ganglia, including the retrovesicular ganglion (RVG) and ventral ganglion (VG). Full image and more information on *C. elegans* are available at www.wormatlas.org

passed locally or slowly, at around 1μm per ms. However, chemical signaling is much less expensive in terms of energy and requires less organization; multiple cells can be reached at the same time by using a diffusive signal. The design of using chemical signaling for some communication and wired signaling for other is one which has been optimized over the course of evolution. In *C. elegans*, chemical signals are the primary form of communication, while in humans, myelinated (insulated) wires traverse the nervous system and throughout the brain, with chemical signals distributing reward information and activating motor responses.

Evolution has found these designs through millions of years of trial and error. From mechanisms that allowed *C. reinhardtii* to sense the direction of light and *C. elegans* to search for food, complex brains have arisen. Some of these elements have persisted from the simplest organisms throughout all intelligent ones, such as the ion channels employed by the *C. reinhardtii*. Light-sensing cells in the eye, on the other hand, were found by evolution multiple times in distinct cases. The octopus and the mammal diverged 500 million years ago in evolution, and the octopus brain is very different from the mammal brain, with a processing center in each arm as opposed to one central brain. However, the two organisms respond to seratonin, a neurotrasmitter, in similar ways [ED18]. Some principles of neural design are seemingly inherent; they are concepts that are so beneficial that they are found multiple times by evolution.

In this work, we aim to understand these aspects of neural design as they apply to Artificial Neural Networks (ANNs). Just as these designs were found by evolution, we use artificial evolution to discover neural designs fit for computational tasks. In simulation, many of the physical principles encountered by biological evolution aren't present. Diffusive communication, done with a chemical in biology, is more expensive in simulation than wired communication, not less. Information can be passed between artificial neurons instantaneously, so the speed of information travel is not a consideration. The evolution of neural design for artificial neurons must therefore follow different principles than that of

biological evolution. However, we use biological neurons, from the simple networks found in *C. elegans* to the complex ones in the human brain, as inspiration for our artificial models. In the next section, we will briefly describe the history of ANNs inspired by the biological brain.

1.1 The brain as a model

The earliest model of the brain comes from the McCulloch-Pitts neuron model, developed in 1943 [**mcculloch_logical_1943-1**]. The motivation for the development of this model was in understanding the biological brain. Recent biological experiments had allowed for the collection of neural activity data, which was then used to create the mathematical model in a foundational moment for the field now known as computational neuroscience. However, the idea of simulating the biological brain soon caught the interest of those in the nascent field of computation. In 1954, an ANN capable of learning was simulated. This network used Hebbian learning, where correlated activity between any two neurons in a network reinforced the connection between them [FC54]; this concept had only recently been discovered in biology. Soon after, the perceptron model was created [Ros58], an abstract model of the neuron based on average spiking activity. This model was very popular in the new field of artificial intelligence for almost a decade, until various shortcomings led researchers to other models. The most well-known of these is the inability of the perceptron to calculate the exclusive-or function, described by Minsky in 1969 [MP69].

Almost a decade later, artificial intelligence research once again focused on ANNs. The backpropagation algorithm, from [Wer74], allowed for layers of perceptrons to be trained, fixing some of the earlier problems with the model. Schmidhuber showed that the same Hebbian learning principles used 40 years earlier could improve ANN learning by pre-training individual layers before using backpropagation [Sch92]. However, the prohibitive costs of training these multi-layer perceptrons discouraged their use. Other methods dominated the field of artificial intelligence, such as expert systems of conditional statements, based on the mathematics branch of logic.

After another two decades, ANNs were shown to solve problems in image recognition, greatly outperforming other methods of the time [Kri+12]. These deep neural networks somewhat resembled the perceptron models of the past, in that individual neurons simulated average activation and used a single synaptic connection to express the average connection with another neuron. However, their organization in layers gave them a considerable advantage over previous models. Some layers fixed weights in kernels, creating a convolutional function that added shift invariance during training, an idea inspired by image processing algorithms. Other layers condensed information by taking the maxi-

mum of different neural activations, or the average. This was an early work in the now ubiquitous field of deep learning [LBH15].

Meanwhile, the field of neuroscience has made enormous progress in understanding the brain. The Hebbian learning process was defined more specifically, and the Hebbian process of synaptic efficacy change, Spike-Timing Dependent Plasticity, was discovered [CD08a]. Other cells besides neurons were shown to play important roles in the brain; astrocytes, a type of glial cell, were shown to induce neurogenesis, the creation of new neurons [SSG02]. Developmental processes such as axon guidance were traced from the molecular level to the cellular to understand their underlying mechanisms [Chi06]. Other computational neural models were created such as the Hodgkin-Huxley model, developed in 1952 [HH52], and the Hindmarsh-Rose model in 1984 [HR84].

Some of these ideas have been integrated into modern artificial neural networks. The neurons in most deep neural networks employ a rectified linear unit function, which has a basis in biology [Hah+00]. One of the inspirations for organizing neurons into layers came from the organization of neurons in cortical columns in the visual system of the brain [LBH15]. However, other ideas from modern neuroscience have yet to be explored. Almost all ANNs use exclusively neural cells, where other types of cells have been shown to play an important role in neural function [Por+11]. ANNs generally use a fixed architecture, where connections between neurons are defined at initialization and don't change, but topological changes have been shown to play an important role in learning [MW17].

The biological brain and the nervous system can inform many of the choices made in artificial neural design. The mechanisms which make biological brains incredibly complex are the same mechanisms responsible for making them efficient computing machines. The biological brain uses 100,000 times less space and energy than a computer, and can perform feats not yet possible with artificial intelligence [SL15]. Bees can learn to do tasks only recently accomplished by deep reinforcement learning, and artificial cognition remains a distant goal. To improve ANNs, and to gain a better understanding of the biological brain, it is worth exploring the details of biological neural design.

In this work, we focus on the integration of recent neuroscience into artificial models. We explore axon guidance, glial cell function, and dopamine signaling in the context of ANNs. Some of these components, i.e. axon guidance, are not yet fully understood in biology. This work can therefore serve both purposes, to increase biological understanding and to improve ANNs, although our focus is on using ANNs for computational tasks.

1.2 Evolving emergent intelligence

In choosing the focus of study and the design of models in this work, certain guiding principles were followed. The use of modern neuroscience is one such principle. The

other main principle is a focus on emergence, the formation of global patterns from solely local interactions. In the brain, no single neuron is responsible for intelligence, nor does individual synaptic behavior represent an idea or thought. Instead, the intelligence of the brain is the result of the interaction between neurons; it emerges from the complex web of neural and synaptic interaction.

Emergent behavior has many examples in biology besides the brain. Ants collectively find optimal paths between their colony and a food source by communicating via chemical traces on the ground. Termites construct nests with specialized structures and intricate tunnels by collectively following the same set of simple rules [Bon+99]. These organisms have served as inspiration for artificial intelligence, notably in the field of swarm intelligence. In this work, we focus on the emergent behavior of neural systems.

In [Dow15], Downing provides a common framework for emergent artificial intelligence:

- Individual solutions to a problem (building an intelligent system) are represented in full. Each solution is an agent.
- Each agent has a genotype and a phenotype
- Each agent is exposed to an environment, and their performance within it is assessed.
- Phenotypes have adaptive abilities that come into play during their lifetime within the environment. This may improve their evaluation, particularly in environments that change frequently, throwing many surprises at the phenotypes.
- Phenotypic evaluations affect the probabilities that their agents become focal points (or forgotten designs) of the overall search process
- New agents are created by combining and modifying the genotypes of existing agents, then producing phenotypes from the new genotypes.

This framework can be seen as a guide to process used in the following chapters. Agents are evolved, and specific attention is given to the genotype representations to improve evolution. These agents act in environments to control the behavior or interactions of neurons, and the performance of the agents influences their evolutionary success.

To focus on emergence, we design agents for local interaction. These individuals inform the rules of local interactions in a neural network. The neural network is then evaluated for its capacity to learn, which is the evolutionary fitness then assigned back to the individual agent. The genotype of the agents can be seen as a part of the genome of neural design, deciding how a specific mechanism of the neural network functions.

1.3 Organization of the thesis

In this work, we use evolution to apply biologically inspired concepts to artificial neural design. To study the entire design of the neural network, we consider the different mechanisms of neural network functions in three separate components:

- cell function, how individual neurons behave
- neural connectivity, how neurons form connections
- learning, how neurons and connections change over time

We focus on the evolution of each of these components individually, and then consider them all together in a final model. This allows us to understand the importance of each part of a neural network and the challenges presented by the design of each component.

In the next chapter, we present an overview of each neural component, starting from their biological basis and exploring existing work to model them. We consider the full range of abstraction, from biological models that attempt to simulate reality to artificial models only vaguely inspired by these components. Each neural component is presented separately, so some models, such as deep learning, are presented in different contexts for the different components.

We conclude chapter 2 with an overview of artificial evolution. Following Downing's framework, the individuals in our evolutionary algorithms are agents. Unlike the evolution of a direct solution to a problem, the evolution of agents or their controllers involves specific techniques and challenges. We explore the evolution of agent controllers in chapter 3. Specifically, in this work we use two mediums for agent evolution, Artificial Gene Regulatory Networks (AGRN) and Cartesian Genetic Programming (CGP), which are presented in detail in chapter 3.

The behavior of individual neurons is examined in chapter 4. We explore existing models of spiking neurons and propose a new design goal for these models: increased fitness during learning. We evolve new spiking neural functions for this goal on a data clustering task. The learning method and neural architecture are fixed to understand how the neural behavior alone impacts learning.

Next, a study in neural connectivity is presented in chapter 5. We construct a model of axon guidance with evolved agents controlling axon behavior and the diffusion of chemical guidance cues throughout the environment. The agents are evolved to create topology with specific traits, such as high symmetry, and to perform in artificial tasks. We demonstrate that the axons rely on neural activity for guidance, a principle only recently discovered in biology.

The final component, learning, is covered in chapter 6. We focus on neuromodulation, the process by which normal synaptic change is enhanced or inhibited to encourage certain

behavior. To cover this vast topic, we present two separate studies. In the first, a dopamine signal is modeled to impact the Hebbian change in the neural networks of virtual creatures learning to swim. In the second, learning in deep neural networks is augmented by local changes to learning rate and synaptic weight update made by neuromodulatory agents placed throughout the network.

In chapter 7, these three neural components are brought together. A complete model of a neural network is presented, where each part of the network is controlled by evolved agents. Each agent is responsible for a function of the network, like the individual components presented here. The genes of these agents are represented as separate chromosomes of the evolved agent, allowing for specific study of certain functions or evolution of the entire agent.

The objective of this work is to discover, through evolution, the important principles of artificial neural design. In the biological brain, the same design principles are used across different components. For example, redundancy of information is used to combat the noise of stochastic biological processes at the level of individual synapses but also at the level of brain organization. By studying artificial neural design in different neural components, we can gain insight into the design principles that are general across all parts of an ANN, as well as those which are important for each component individually.

Chapter 2

Background

The brain has long been a source of inspiration for computational intelligence. In 1948, Turing developed B-type machines based on neurons [Tur09], and in 1954, Farley and Clark simulated a network based on the recently founded Hebbian theory [Heb+49], [FC54]. Perceptrons popularized the use of artificial neural networks for computational tasks as early as 1958 [Ros58].

Since then ANNs have risen and fallen in use as artificial controllers. Modern ANNs have recently demonstrated human-level ability, and in some cases, super-human ability, in image recognition [Kri+12], game playing [Mni+15], text translation, [Fir+17], speech recognition [Dah+12], and more [DY+14]. This is largely due to the advent of deep learning, a machine learning field which optimizes deep neural networks for specific tasks [LBH15]. Deep learning is the state of the art for a large amount of research using artificial neural networks, and in this thesis we use deep learning to study neuromodulation (section 6.2).

However, for the most part, the models in this work are based on spiking neural networks, which have only recently been used in ways that resemble deep learning [Khe+16], but which have also demonstrated impressive ability for application [Moz+18a], [LDP16]. The presentation of deep learning concepts in this chapter is therefore minimal; we instead focus on how biological neurons have influenced a variety of artificial neural network models, including but not limited to deep learning.

The neuron is the base unit of computation in the biological brain. A schematic of the neuron is shown in Figure 2.1, displaying the different components of the neuron. The cell body of the neuron, the soma, extends multiple projections, the morphology of which depends on the specific neuron type. In the development of the neuron, one of these projections becomes an axon, which serves as output for the neuron. All other projections become dendrites, which provide input to the neurons. The dendrites contain multiple connection sites, where synapses can form when another neural projection connects to the

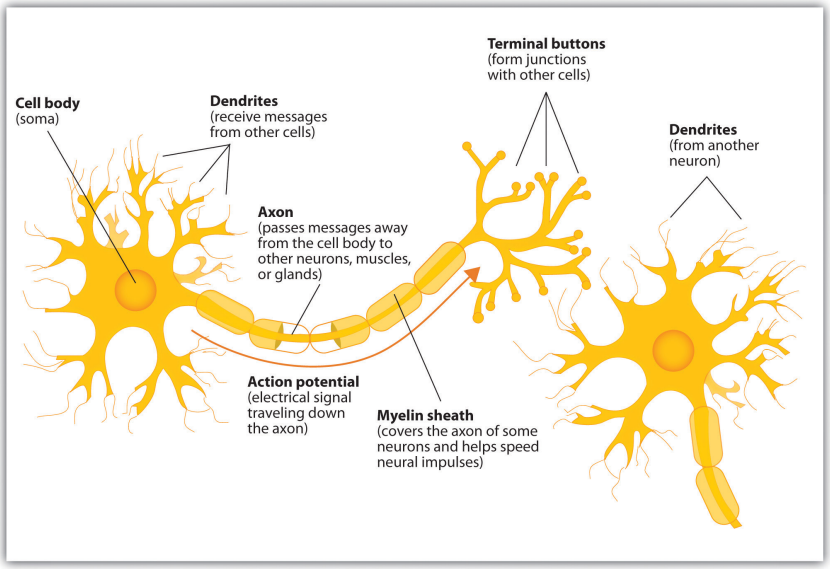

Figure 2.1: A schematic of the neuron and its components, from [SW15]. From the soma, the cell body, the neuron extends dendrites to receive input from other cells and an axon to deliver output. When the neuron activates, an electrical signal travels down the axon, which, for some neurons, is insulated by a myelin sheath.

dendrite. Neurons are excitable cells; they hold an electrical charge, which they discharge as an action potential, sending a signal to connected cells across synapses. In this process, the sending cell is referred to as the pre-synaptic cell and the receiving the post-synaptic.

Through this process of neural connection, activation, and communication, cognition is formed. The process by which individual neural functionality leads to cognition is a topic of active research and is far from understood [Men12], [SL15]. An example of realistic neurons is given in Figure 2.2, which shows pyramidal neurons from different sections of the cortex. This type of neuron is common in the pre-frontal cortex, where it is understood to play an important role in cognition. Recent research has shown, using theory from deep learning, that pyramidal neurons could inform credit assignment in the brain due to its specific structure, determining which neurons and signals are responsible for positive or negative actions [GLR17]. In the design of neural networks inspired by biology, it is important to consider the numerous mysteries which remain in understanding the biological brain.

In this chapter, we present neural networks in three stages. In section 2.1, we explore the function of the neuron; what the neural cell does and how it is modeled, both in biological models and artificial neural networks. Then in section 2.2 we present the means by which neurons communicate and are organized in a network, describing the architecture design process common for modern neural networks. Finally, in section 2.3, we discuss biological learning mechanisms, their translation to computational models, and

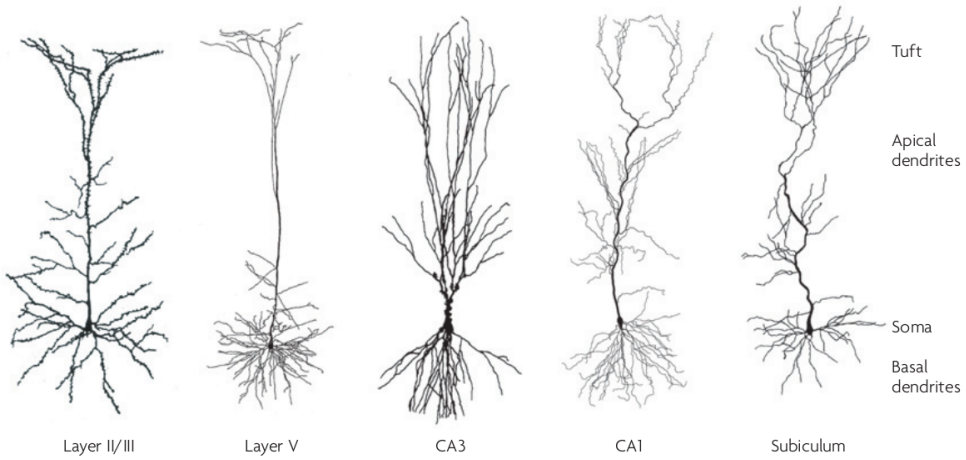

Figure 2.2: Biological neurons, specifically pyramidal neurons, from different areas of the cortex [Spr08]. Pyramidal neurons are the most numerous excitable cell in mammalian cortex structures and are considered important for advanced cognitive functionality.

mechanisms for learning in artificial neural networks. Through this exploration in three stages, we explain how a single biological neuron communicates with other neurons in a complex network, eventually leading to learning and cognition, and how artificial neural networks are designed towards the same goal.

Following the overview of biological and artificial neural network models, we present an overview of evolutionary computation in section 2.4, as evolutionary algorithms are the medium used throughout this work to control neural components. Finally, in section 2.5, we review other works which involve the evolution of artificial neural networks or neural design.

2.1 Neural cell function

The goal of neural cell functions is to model the activity of the cell body, mostly concerning the current flow across the cell membrane and the movement of ions across ion channels. The simplest models consider the neuron as a single electrical component with a potential, which either rises and falls in spiking models, or is expressed as an average rate in perceptron-based models. More complex models represent ion channels, such as sodium (Na) or potassium (K), which give rise to ionic current [HH52]. Spiking models generally focus on the action potential of the neuron, which is the event when a neuron spikes or "fires", its membrane potential rapidly rising and then falling. In most artificial models, this occurs when the membrane potential, often represented as V, surpasses a specific threshold, V_{thresh}. The neuron then sends a binary signal to downstream connected neurons. In this section we focus on models of the relationship between input

neural membrane currents and output membrane voltages. While other biological models of the neuron exist, the models which have inspired artificial neurons are all based on this relationship.

2.1.1 Biological neural models

One of the most well-known neuron models is the Hodgkin-Huxley model, which defines the relationship between ion currents crossing the cell membrane and the membrane voltage [HH52]. In this detailed model, different components of the cell are individually modeled as electrical element. The membrane potential of the neuron is represented as V_m. The membrane's lipid bilayer is represented as a capacitance, C_m. Voltage-gated ion channels are represented by electrical conductances, g_n for each channel n. Leak channels, which set the negative membrane potential of the neuron, are represented by linear conductances, g_L. These components are used to calculate different current flows: the current flowing through the lipid bilayer, I_c, and the current through an ion channel, I_i, which are calculated as:

$$I_C = C_m \frac{dV_m}{dt} \tag{2.1}$$

$$I_i = g_n(V_m - V_i) \tag{2.2}$$

where V_i is the reversal potential of the given ion channel. Through experimentation, Hodgkin and Huxley developed a set of differential equations, using dimensionless variables n, m, and h, to express potassium channel activation, sodium channel activation, and sodium channel inactivation, respectively:

$$\frac{dn}{dt} = \alpha_n(V_m)(1-n) - \beta_n(V_m)n \tag{2.3}$$

$$\frac{dm}{dt} = \alpha_m(V_m)(1-m) - \beta_m(V_m)m \tag{2.4}$$

$$\frac{dh}{dt} = \alpha_h(V_m)(1-h) - \beta_h(V_m)h \tag{2.5}$$

These quantities are then used to express the different ion channel activation in the current equations, leading to a final expression of the total current passing through the membrane of a cell with both sodium and potassium channels:

$$I = C_m \frac{dV_m}{dt} + g_K n^4(V_m - V_K) + g_{Na} m^3 h(V_m - V_{Na}) + g_L(V_m - V_L) \tag{2.6}$$

where g_K and g_{Na} are potassium and sodium conductance, V_K and V_{Na} are the potassium and sodium reversal potentials, and g_L and V_L are the leak conductance and the leak reversal potential.

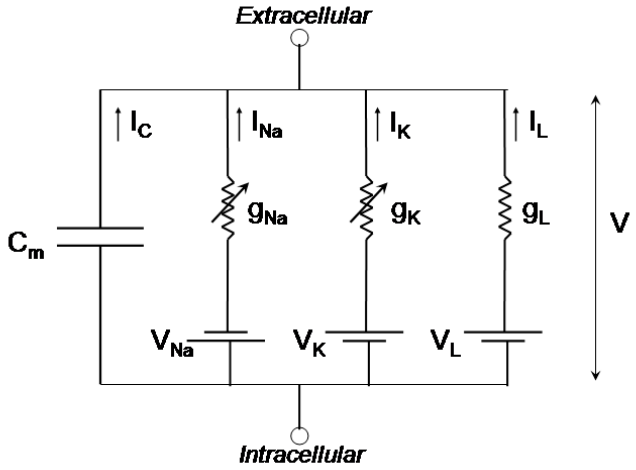

Figure 2.3: The Hodgkin-Huxley model of a neuron membrane, represented as an electrical circuit, from [Ski06].

A common way to consider neural models is as electrical circuits, and the corresponding circuit for the Hodgkin-Huxley model is presented in Figure 2.3. As is apparent in this representation, the Hodgkin-Huxley model is separable into the four principle currents, the current through the lipid bilayer, I_c, the current through potassium and sodium ion channels, I_K and I_{Na}, and the leak current, I_L, which represents passive properties of the cell. Conductance-based neuron models use these separate components to express cells of various conditions and at different levels of complexity, demonstrating one of the features of the Hodgkin-Huxley model. Selected components can be focused on independently due to their separation in the model, allowing for different neuron types and settings to be modeled. For instance, potassium ion channels may be ignored in certain cell types, while in others, the leak current may be negligible. The Hodgkin-Huxley model offers a good degree of flexibility while also simulating detailed cellular components.

For artificial computational purposes, however, the Hodgkin-Huxley model is incredibly costly while offering a level of detail which is often unnecessary. For the simulation of large networks of artificial neurons, especially in their study for performing artificial tasks, individual ion channel currents can be aggregated, or simply ignored, to simulate the electrical activation of the cell as a whole. One of the simplest and oldest models, the Integrate and Fire (IF) model [Lap07], can be expressed as using only the membrane potential current, I_c, in the Hodgkin-Huxley model:

$$I = C_m \frac{dV_m}{dt} \tag{2.7}$$

However, this simple model does not display the variety of behavior exhibited by

biological neurons and Hodgkin-Huxley simulations, for example, a series of spikes in rapid succession, known as bursting. A variety of extensions to this model have been proposed, such as the IF-or-Burst model [Smi+00], which explicitly includes bursting. Neural activity data has given rise to other models. The Izhikevich model was created by fitting the coefficients of a differential equation to the dynamics of a cortical neuron [Izh03]. In general, these models are referred to as spiking neural models, which are examined in more detail in chapter 4, where the different models are compared based on their biological accuracy, computational cost, and flexibility to extension.

2.1.2 Activation functions

Another approach to modeling neuron function is to model the average spiking activity of a neuron over a time window, referred to as the neuron's activation. These activation functions allow for computing with detailed continuous signals, as opposed to binary spiking events. Furthermore, most activation functions are continuously differentiable; as will be shown in section 2.3, this is an important feature for learning in artificial neural networks. Spiking neural models are non-differentiable due to the spike event and only recently have differentiable learning methods been used directly with spiking neural models [LDP16].

One of the earliest activation functions was the logistic function. The derivative of this function is easy to calculate, simplifying learning calculations. Neurons that use a logistic function are usually called "sigmoid neurons" and are still commonly used. The activation function and its derivative are:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

$$f'(x) = f(x)(1 - f(x)) \tag{2.9}$$

In these equations, x represents the sum of all synaptic input to the neuron, averaged over time. This value can be positive or negative; inhibitory neural inputs are represented as negative synaptic input. The output, $f(x)$ represents the average activation of the neuron over time. For sigmoid neurons, the range of the output is $(0, 1)$ and can be easily understood to represent neural activation; this is often interpreted as the average spiking rate of the neuron and can correspond to biological values. The firing rates of visual cortex neurons in anaesthetized cats was found in to be around 3.96 Hz and 18Hz in awake macaque monkeys [Bad+97]. The output of the logistic function could, for example, be interpreted as firing rates in MHz, or as the firing rate normalized by a biological limit [DA01].

Biological realism is not always a goal of activation function design, however. Another popular activation function is the hyperbolic tangent function, again for the simplicity of the derivative calculation:

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.10}$$

$$f'(x) = 1 - f(x)^2 \tag{2.11}$$

This function has a range of $(-1, 1)$, which indicates negative values of average neural activation. This has little biological basis, as frequencies cannot be negative, but the hyperbolic tangent function is usually preferred over the sigmoid function for optimization with artificial neural networks. This is due to the fact that, at $x = 0$, the hyperbolic tangent has a steady state, which aids optimization [LeC+98], [GB10]. In the design and choice of activation functions, the main priority is increasing neural network optimization.

Usage of modern ANNs has therefore mostly converged to a single activation function, due to its effectiveness for optimization. The recitified linear unit (ReLU) simply outputs x if $x > 0$, and outputs 0 otherwise:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{2.12}$$

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.13}$$

Despite having a non-continuous derivative, this function has been shown to increase ANN optimization [Hah+00]. It has also been argued that, compared to the hyperbolic tangent function, this is a more biologically realistic neuron model, as the output can again be considered as a spiking frequency [LDP16]. However, its popularity is due to the speed of computation, the increased optimization performance when using this function, and the use of numerical optimization which mitigates the problem of the non-continuous derivative.

In all of these models, we have considered how the sum x of all synaptic input is translated to an output, i.e. $f(x = \sum \vec{x})$. However, other integration methods of the synaptic inputs $\vec{x}$ are possible. Neurons which collect statistical information about their synaptic inputs are referred to as "pooling" neurons and are an important part of deep neural networks architectures [LBH15]. Max pooling, where $f(x) = \max(\vec{x})$ is a common choice, as is mean pooling ($f(x) = \frac{\sum_i^N \vec{x}_i}{N}$), although to a lesser degree due to optimization improvement when using max pooling neurons [SMB10]. While this artificial neuron type

is not based on any biological neuron, the brain often uses similar statistical information such as recent average firing activity from multiple synapses to inform activation or learning in other neurons [SL15].

2.1.3 Other cell behavior in the brain

In the brain, synaptic communication is not the only form of communication, and neurons express activity beyond the action potential. Synaptic communication is referred to as wire transmission, as the synapses resemble electrical wires, but a second form of communication, volume transmission, is also important [Agn+06]. Cells, including neurons, communicate by excreting chemical signals, which can travel and change in the extracellular space and be processed by receiving cells [Agn+10]. Dopaminergic neurons are one such example, which release the reward chemical dopamine at specific points throughout the brain [Hos+11]. While volume transmission in cells in general has been studied in computational models, it has rarely been considered for artificial neurons. [FB92], for example, uses volume transmission in artificial neurons to guide network development. There has been little continuation of this study in modern deep neural networks.

Neurons are also far from the only cells in the brain. Other cells play an important role in cognitive function. Glial cells create the structure around neurons, provide nutrients and oxygen to them, insulate them, protect them from pathogens, and remove dead neurons [JM80]. Astrocytes, a type of glia also called astroglia, have been shown to induce neurogenesis, the creation of new neurons, and to regulate synaptic activity, both processes which are foundational to learning and cognition [SSG02], [PNA09]. However, there has been little integration of non-neuronal cells in ANNs. In [Por+11], artificial astrocytes regulate high-frequency spiking activity in a neuron-glia network and improve learning on classification tasks.

In this thesis, we consider other cell types and other forms of cell behavior, although we focus on neuron activation and wire transmission as the base of our models. While the study of other cell behavior in the brain is necessary for ANNs, there is already an abundance of topics to study for ANNs based on cell models of neuron activation and wire transmission, namely in how neurons are connected and how they learn, which will be described in the next sections.

2.2 Neural connectivity

In the brain, neurons create connections with other neurons by projecting their axon throughout the brain. The axon is led by a growth cone, which senses extracellular chemicals and is guided by these cues, neural activity, and contact with other neurons

to eventually find a target, where it forms a synapse. Electrical signals from the neuron can then pass across the synapse to the downstream or postsynaptic neuron. Through following a set of rules, encoded in the neuron's genes, these axon growth cones create neural architectures responsible for learning and cognition, and understanding these rules is a large area of neuroscience. The axon guidance process is covered in more detail in chapter 5.

In artificial neural networks, neurons connect through synapses, which, as in biology, are directional: a presynaptic neuron sends a signal to the postsynaptic neuron. Unlike biology, these synapses are most often represented as a single scalar value, the synaptic weight, which represents the average synaptic efficacy from the presynaptic neuron to the postsynaptic. The presynaptic output is multiplied by the weight upon transmission to the postsynaptic neuron. The design of the structure of an ANN, being the number and type of cells in the network, how these cells are connected, and how the synaptic weights are used, constitutes one of the largest challenges of using an ANN.

The vast majority of modern artificial neural networks have static network topologies which are designed by human experts. These neural architectures are engineered to perform specific tasks, such as image classification or text translation. Some of the motivation for architectures remains biological; for instance, image classification networks take inspiration from the columns of neurons in the cortex [Ser+07]. The fly olfactory circuit inspired the neural architecture in [DSN17]. Most architectures, however, are designed based on machine learning principles, or domain-specific principles, such as in image processing tasks.

In designing neural networks, neurons are usually organized in layers: an input layer, one or more hidden or intermediary layers, and an output layer. In general, synapses connect the input layer to the first intermediary layer, each intermediary layer to the next intermediary layer, and the last intermediary layer to the output layer. One of the advances of modern deep learning is the idea that, instead of having a single or small number of intermediary layers of neurons (a "shallow" architecture), multiple layers should be stacked to encourage data processing and compression (a "deep" architecture), which has been facilitated by developments in GPU computation and optimization methods [LBH15].

Modern neural networks involve multiple different types of layers. In Figure 2.4, two types are shown, a fully-connected layer and a convolutional layer. In a fully-connected layer, each neuron from the first layer connects to each neuron in the second layer with an individual synapse. In a convolutional layer, neurons receive input from a subset of the neurons in the previous layer, where the subset is shifted over the previous layer to create a moving window. The synaptic weights of these layers are shared: the first synapse of the window, shown in Figure 2.4 in brown, has the same weight as the other two brown

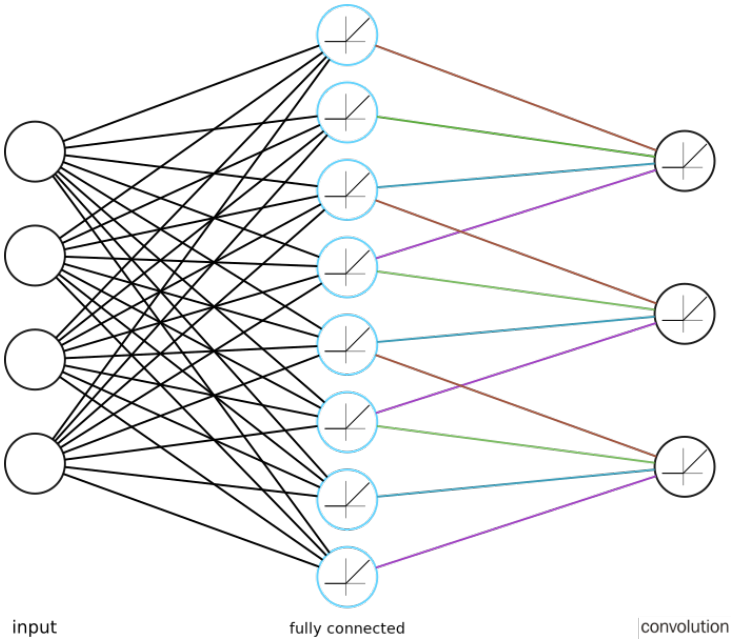

Figure 2.4: Layers of an artificial neural network: input, fully connected, and convolutional. Synapses which share weights in the convolutional layer are represented by the same color.

synapses. The same is true for the second, in green, the third, in blue, and the fourth, in purple. This layer therefore only has 4 synaptic weights, despite having 12 synapses.

Convolutional layers allow for processing on inputs which require shift invariance [LBV15]. For example, in image classification, a feature of an image necessary to classify it may not always be located in the same part of the image. By introducing shift invariance to neural networks, their performance in image classification greatly increased. In Figure 2.5, a deep convolutional neural network architecture known as VGG16 is shown [SZ15]. This architecture was designed for image classification and remains is now a popular choice for this task.

Other tasks use different network designs. Image segmentation tasks often use architectures like U-net [RFB15], which connects neurons in early layers to layers of the same size in a second portion of the network. Long short-term memory (LSTM) layers, which use recurrent connections to store information, are a popular choice for time-series data and natural language processing [HS97]. Designing a neural network is a difficult engineering problem and constitutes a large section of contemporary deep learning research.

Neuroevolution, the use of artificial evolution to determine the structure or weights of a neural network, has been used for decades to automatically create neural networks and will be described in section 2.5. Due to the difficulty of designing optimal deep ANN topologies, automatic methods of creating neural network architectures, neuroevolution and other search methods, are becoming more common [EMH18].

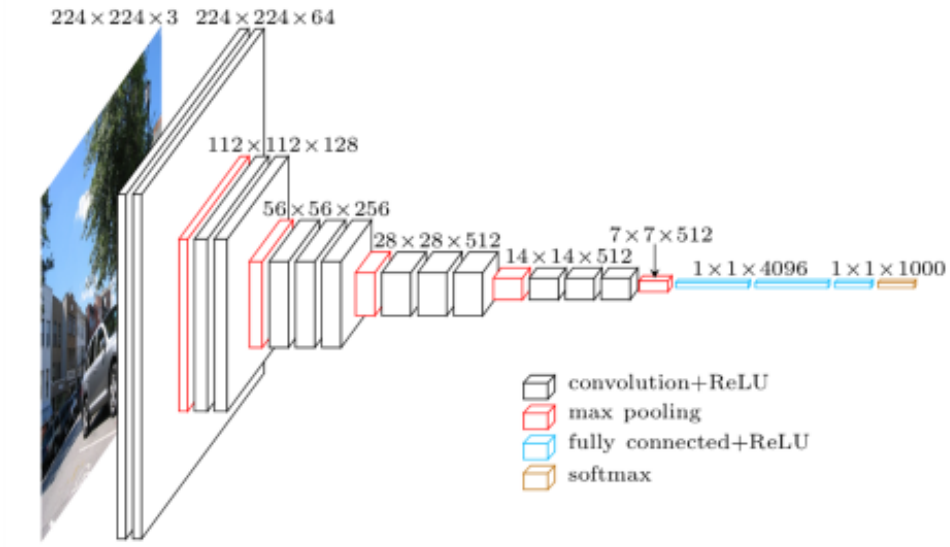

Figure 2.5: The VGG16 architecture [SZ15]

2.3 Learning in neural networks

In the brain, learning takes a variety of forms. Recent research has demonstrated that the creation of connections, and even new neurons, is an integral part of adult learning. Gray matter, which is mostly composed of neural cell bodies and glial cells, has been shown to increase during learning in the hippocampus, the center for storing navigation cues in the brain. London taxi drivers had an increase of hippocampal gray matter over years over learning the layout of the city, which then decreased after their employment [Mag+00]. Piano tuners also gradually increase their hippocampal gray matter, as well as auditory areas in their temporal and frontal lobes [Tek+12]. White matter, mostly composed of myelinated (insulated) axons, also increased, showing that new connections were formed during the learning of the complex soundscape of the piano.

However, the main focus of research in learning in artificial neurons is based on synaptic plasticity, the change in synaptic efficacy over time, which is also considered as a central component of learning in biology. This takes two forms in biological learning: long term potentiation (LTP) and long term depression (LTD). In LTP, the [CB06]

The NMDA receptor, or NMDAR, in neurons is a key component for synaptic plasticity [See+95]. This receptor, when active, allows positively charged ions to flow through the neuron membrane, increasing the membrane potential. NMDAR is activated by binding to glutamate, which is released from the presynaptic neuron on a spike. However, the effect of binding glutamate at the NMDA receptor depends on the potential of the postsynaptic neuron membrane where the NMDA receptor resides. The receptor will react weakly if binding glutamate when the membrane is at resting potential, but will

activate fully after depolarization events. The NMDA receptor therefore detects the coincidence of presynaptic inputs, represented by glutamate, and the postsynaptic response, depolarization of the cell membrane [SL15].

Coincidence detection is fundamental to Hebbian theory, an pattern for synaptic plasticity detailed in Hebb's foundational book in 1949 [Heb+49]. This theory, often cited as "cells that fire together wire together", describes the change in influence from one neuron to another based on their respective activity. Assuming, as is the case in artifical neural networks, that the relationship between two neurons i and j can be expressed as a single weight w_{ij}, Hebb's rule can be expressed as:

$$\Delta w_{ij} = \eta x_i x_j \tag{2.14}$$

where η is a learning rate, and x_i is the input of neuron i. This means that, when the synaptic inputs of both i and j are high, the weight between them will increase by a large amount. The coincidence of activity in i and j is detected by mechanisms like the NMDAR glutamate activation and leads to increased synaptic efficacy from i to j. This learning principled has been linked to memory formation [Tsi00].

2.3.1 Spike Timing Dependent Plasticity

Spike-timing-dependent plasticity (STDP) is a form of Hebbian synaptic plasticity observed in the brain [RBT00]. This mechanism uses the timing of spikes in a presynaptic, postsynaptic pair to determine the change in synaptic efficacy. When a presynaptic spike is closely followed by a postsynaptic spike, the synaptic efficacy increases, but when a postsynaptic spike precedes a presynaptic spike, the efficacy decreases [CD08a]. This is temporal coincidence detection, enabled by mechanisms such as the slow unbinding of glutamate in NMDAR.

The change in synaptic efficacy depends on the exact time difference between the presynaptic and postsynaptic spikes, as shown in Figure 2.6. Activity in biological neurons has confirmed the relationship between the timing difference and the synaptic change [BP01]. This has spawned a number of STDP learning rules for use in artificial spiking neural networks [SMA00], [BDK13].

Hebbian learning is a form of unsupervised learning alone. Synaptic plasticity is informed only by activity of the network, which can learn features about inputs but can not learn based on a reward or error signal. However, in the domain of unsupervised learning, STDP has shown impressive results. In [DC15], the MNIST handwritten digit set is used to demonstrate learning in a two-layer spiking neural network, which achieves accuracy competitive with deep learning methods using only STDP. [Khe+16] uses STDP for object recognition, also demonstrating competitive unsupervised learning for this visual task.

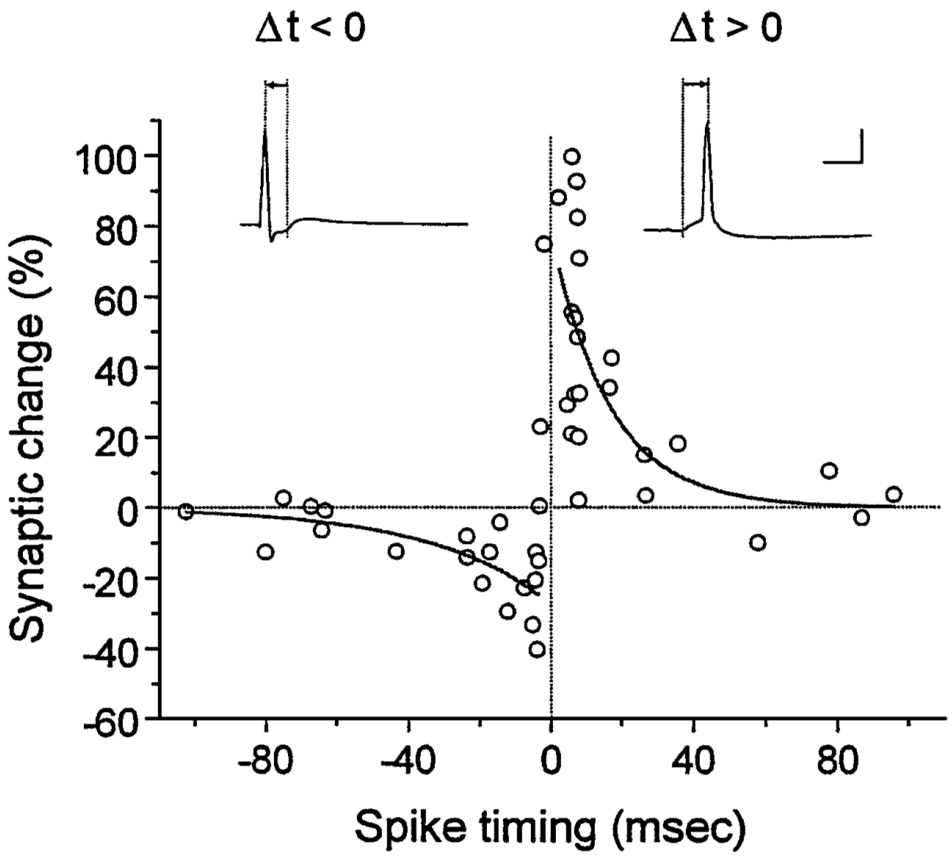

Figure 2.6: STDP observed in biological synapses, from [BP01]

Both of these works highlight the importance of local competition rules for STDP-based learning, which can be considered as part of the neural architecture design or learning algorithm design. In [DC15], inhibitory neurons prevent firing of more than one neuron at a time, and in [Khe+16], only the first neuron in a layer to fire will update its synaptic weights. Local competition among neurons is known in biology, albeit more complex.

Semi-supervised learning can be achieved in STDP by supplying a reward signal. In this form of learning, the exact correct response is not provided, but reward is provided, for instance, when a correct response is given. In the brain, the neuromodulator dopamine is responsible for this type of learning, as it increases synaptic plasticity based on unexpected reward. Neuromodulated STDP [FG16], also called Reward-modulated STDP (R-STDP), is a relatively new method for learning in spiking ANNs, but has shown promising results. In [Moz+18b], R-STDP is able to learn on three different visual categorization tasks to a high accuracy, outperforming standard STDP. Neuromodulation is more fully explored in chapter 6, and more examples of R-STDP are given in section 6.1.

2.3.2 Gradient Descent and Backpropagation

While reward-modulation can improve learning by providing a coarse reward signal, teaching the network whether or not a specific action was correct, many problems can provide more detailed feedback, specifically how each response is correct or incorrect. In supervised learning, neural network outputs are compared to expected output, and the difference forms an error signal which is given back to the network. The learning process is therefore considered as as an optimization problem of the neural network parameters θ, being synaptic weights and neuron biases, according to some loss function Q. In classification, for example, this loss function can be the mean squared error between a target classes, h_i, and the class given by the deep NN, $X(\theta, i)$:

$$Q_i(\theta) = (X(\theta, i) - h_i)^2 \tag{2.15}$$

$$Q(\theta) = \frac{1}{n} \sum_{i=1}^{n} Q_i(\theta) \tag{2.16}$$

The standard approach to optimizing the weights θ of the ANN is to use gradient descent over batches of the data. Classic stochastic gradient descent (SGD) uses a learning rate hyper-parameter, η to determine the speed at which weights change based on the loss function. This method can be improved with the addition of momentum [Nes83], which changes the weight update based on the previous weight update. An additional hyper-parameter, α, is then used to determine the impact of momentum on the final update:

$$\Delta\theta^{(t+1)} \leftarrow \alpha\Delta\theta^{(t)} - \eta\nabla Q_i(\theta^{(t)}) \tag{2.17}$$

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + \Delta\theta^{(t+1)} \tag{2.18}$$

The error signal is then passed back through the network, from the output layer through the intermediary layers to the input layer, adjusting synaptic weights throughout the network. This algorithm is called backpropagation and is the basis of synaptic plasticity in deep learning [LBH15]. An overview of backpropagation is presented in Figure 2.7.

In deep learning, there is a variety of gradient descent approaches to choose from. Adagrad [DHS11] implements an adaptive learning rate and is often used for sparse datasets. Adadelta [Zei12] and RMSprop [TH18] were both suggested to solve a problem of quickly diminishing learning rates in Adagrad and are now popular choices for timeseries tasks. Adam [KB14] is one of the most widely used optimizers for classification tasks. In section 6.2, this optimizer is presented in detail and improved upon using neuromodulation.

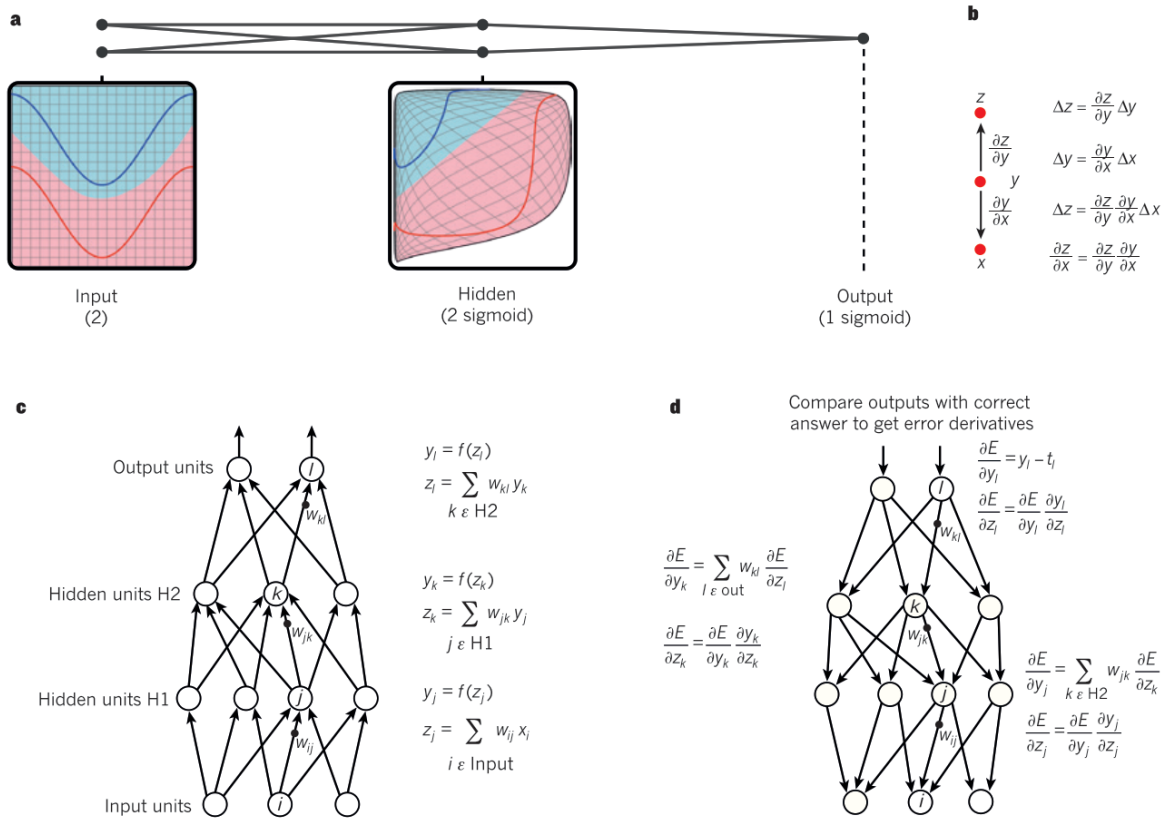

Figure 2.7: Backpropagation in multi-layer ANNs; from [LBH15]. a) A multi-layer neural network can transform different input signals, shown in red and blue, in order to linearly separate them; from http://colah.github.io. b) The chain rule demonstrating how changes of x on y and y on z are composed. c) The forward pass from the inputs to the outputs through the network. d) The backward pass of the error signal E through the network, using the chain rule to backpropagate the error signal throughout the network.

2.4 Evolutionary computation

The previous three sections have explained how neural networks act, connect, and learn, and it was shown that biological neural networks serve as inspiration for artificial neural networks throughout ANN design. Just as biological neural networks are an obvious source of inspiration for learning and cognition, biological evolution is a source of inspiration for optimization. There is possibly no other natural optimization process which produces the variety of complexity created by natural evolution.

Evolution has been used as an algorithm for optimization for decades [FOW66]. Throughout the field of evolutionary computation (EC), the basic algorithm is similar: genomes represent individuals, which compete in some way to increase their chances of passing their genetic material to the next generation. Genetic operators, i.e. mutation or crossover, randomly alter some of the genes to create new individuals. In this way,

an evolutionary algorithm searches for competitive individuals, based on the competition metric defined for the evolution. In general, the competition metric is an evaluation function, called the evolutionary fitness function or objective function, which interprets the genes of the individual and returns one or multiple fitness values, which are then used to rank the individuals during selection. This simple concept is the base of a multitude of algorithms, which are usually separated into three classes: evolutionary strategies, genetic algorithms, and genetic programming [BFM18], which we explore next.

2.4.1 Evolutionary strategies

Evolutionary strategies (ES) are characterized by their use of mutation-based search [BS02]. One of the simplest evolutionary strategies is the $(1 + 1)$ ES [DJW02]. In this algorithm, an individual, the parent, is randomly generated at initialization and evaluated according to an objective function. For each iteration, or generation, of the algorithm, a new individual, or offspring, is created by mutating this parent, randomly changing a portion of the parent's genes according to a mutation rate. The offspring is also evaluated according to the objective function and is compared to the parent. If the offspring individual has a superior fitness, it replaces the parent individual. Subsequent generations will use this offspring as the base for mutation and comparison, until a new offspring is found which is superior to it.

The name of the $(1 + 1)$ ES refers to the number of parents and number of offspring, respectively. Another popular algorithm is the $(1+\lambda)$ ES. This follows the same principles as the $(1 + 1)$ ES, but instead of creating 1 offspring via mutation at each generation, λ offspring are created. Multiple parents are sometimes used, denoted (μ, λ) ES. These ES are well studied, with theoretical understanding of their use on a variety of fitness functions [Bey94]. While the base algorithm is very simple, there is ongoing study of these ES, such as [DD18], which studies optimal mutation rates that change throughout evolution.

A very well-known ES is the covariance matrix adaptation ES (CMA-ES) [HO96]. In this (μ, λ) ES, the pairwise dependencies between the individual genomes are represented in a covariance matrix. Instead of individually mutating the individuals in the population, the covariance matrix is updated, using the distribution of the individuals to approximate a second order model of the objective function. CMA-ES has been demonstrated as a robust optimization method for a variety of real-valued problems [Han06].

2.4.2 Genetic Algorithms

Genetic algorithms (GA) provide a broad framework for a variety of evolutionary computation algorithms [Hol92a], [Hol92b]. A GA has, in most cases, three methods which

inform search: a selection method, a mutation method, and a crossover method. A population of individuals evolves according to these methods, with parents being selected at each generation via the selection method. These parents may create an offspring via crossover, combining their genetic material according to the crossover method. Individuals passing on to the next generation are mutated according to the mutation method. In some cases, a percentage of the population is passed directly to the next generation, which is called elitism.

A variety of methods exist for GAs. A common selection method is tournament selection, where a set of n individuals are selected from the population at random and then ranked according to their fitness. The best individual from the n is returned as a parent for the next generation. [GD91] presents a comparison of many other selection schemes.

Mutation operators are often similar to that of the $(1+1)$ ES, where a random subset of the parent's genome is replaced with new random values when copying to the child genome. However, other methods exist. In NEAT, an algorithm specialized for evolving networks, mutation involves the addition of network nodes and links [SM02].

The design of a beneficial crossover operator is a challenging part of using GAs. While genomes of binary or floating point values have been shown to benefit from crossover on benchmark problems, [OSH87], more complex individuals may struggle. Ideally, a crossover operator combines two or more parent genomes constructively to create a child offspring. However, when these genomes represent a network, or as we'll see in the next section, a program, the design of a crossover operator that constructively combines multiple genomes is non-trivial. Many genetic algorithms forego crossover, or use a small crossover probability, which determines the number of offspring per generation created via crossover.

Genetic algorithms have been well-studied in the context of multi-objective optimization (MOO). In this domain, the objective function returns not one but multiple values (otherwise considered as having not just one but multiple objective functions). The question of how to properly select between individuals then becomes non-trivial. An individual is said to have Pareto dominance over another individual if all of its fitness values are superior to the other, but how should parents be selected if not all of their fitness values are superior? NSGA-II [Deb+02] is a widely used algorithm for MOO which exploits the fact that the genetic algorithm can simultaneously represent multiple solutions in the population. Individuals are selected based on their distance along the Pareto front from other individuals, so at any one time a population has individuals which have been competitive at one or multiple objectives.

2.4.3 Genetic Programming

The final, and most recent, sub-field of evolutionary computation is genetic programming (GP), which focuses on the evolution of computational structures, or programs. This field was founded by Koza in 1994 [Koz94] with the evolution of LISP programs and now includes a variety of methods not specific to any computer language. LISP was chosen due to the representation of programs as functional trees, where every node in the tree represents a function and terminal nodes operands. This tree structure was simple to evaluate and evolve. [Ban+98].

Other representations are now popular. Grammatical evolution uses a grammar to convert the genome into a program [RCN98]. Cartesian Genetic Programming (CGP) interprets the genome as a graph of functional nodes [Mil11]. CGP is used in this work and is studied in detail in section 3.2. [Pol+97] presents another graph-based genetic programming method. PushGP [Spe02] is a stack-based genetic programming algorithm which has been used to study autoconstructive evolution, where individuals define their own evolution operators as programs [Har+12b].

Between these different representations, the underlying evolutionary algorithm can be an ES or a GA, but is usually specialized for the given program representation [Pol+08]. Crossover has been a specifically difficult operator to design for GP [PL98]. Tree-based GP has often used subtree-crossover, where parts of each parent tree are combined to create a child. However, this method creates increasingly large trees, a problem termed bloat. Standard GA crossover methods, such as single point crossover where the genes from one parent are taken up to a randomly chosen point, after which genes from the second parent are taken, are not guaranteed to constructively combine the two parents. [LS97] presents a study of different mutation and crossover operators for GP. In section 3.2, we design and study a variety of mutation and crossover operators for CGP.

2.5 Evolving artificial neural networks

Neuroevolution is the use of evolutionary computation for the design of neural structure, optimization of synaptic weights, or evolution of rules and principles which guide neural network design. These approaches can be divided into categories of direct encoding, where genes directly correspond to neural network components (e.g., synaptic weights), and indirect encodings, where genes determine a function or process which then informs neural network components.

EC has often been used to optimize the synaptic weights of neural networks with predefined topologies when gradient descent is not suitable. As EC does not require the definition of a differentiable fitness function, it can be used in many cases when gradient

descent cannot. Specifically, EC has proven to be a useful means of finding synaptic weights in control and reinforcement learning tasks. In [RS94], synaptic weights were evolved using a GA to control the landing of a toy lunar module. [Sal+17] uses an ES to evolve the weights of a deep neural network for playing Atari games. These represent direct encoding methods, as the genomes directly correspond to the synaptic weights.

SANE [MM97] is an interesting approach to neuroevolution where individuals in the population each represent a single neuron and must collaborate to create a neural network, by which their fitness is determined. This method was used to create neural networks that play Go [RMM98] and extended to study 2-d pole balancing in recurrent neural networks.

A well-known direct encoding method is NEAT [SM02], a GA specialized for neuroevolution. Individuals in NEAT begin as minimal neural networks, with neurons representing the inputs, outputs, and a single hidden neuron. Each generation, mutation may complexify the network by adding a new node or new connection. Crossover uses historical markers of mutation events to allow for crossover between parents to take place at similar network nodes. Finally, the GA selection operator is modified to favor innovation by reducing competition between individuals which are genetically different.

NEAT has spawned a variety of other neuroevolutionary methods. HyperNEAT [SDG09] uses the NEAT evolutionary method to instead evolve a program, which then determines the weights of a neural network. As the genome corresponds to the program and not the resultant neural network weights, HyperNEAT is an indirect encoding method. ES-HyperNEAT [RLS10] extends HyperNEAT by allowing the placement of neurons to be discovered during evolution.

CGP has been used for both direct encoding of neural network structure and weights in CGPANN [KKM10b] and in indirect encoding, using evolved programs to determine the structure and weights of an ANN [KMH11]. These methods are described in more detail in section 3.2.

Some indirect encoding methods take inspiration from the biological process of neural development to create the structure of ANNs during a live, developmental period. These models simulate the developmental process using a set of evolved rules, such as when a cell should divide, create a connection, destroy a connection, or commit apoptosis (cell death). Some of the earliest work in this field is [Kit90], where a graph generation system is used to develop a neural network, where the rules of the graph system are found using evolution. Another method from the same period is Cellular Encoding (CE) [GW93], which uses evolution to find a grammar that defines the development of an entire network, starting from a single cell.

These approaches use evolution to generate the rule set due to the complexity of the rules necessary for development, or to guide development towards a specific evolutionary goal. [Dow07] provides insight into the characterization of these developmental evolu-

tionary algorithms (DEAs) and proposes a DEA based on a popular neuroscience model, the Neuromeric Model. In this DEA, the neural network is created in three stages: first, a genome is translated into a set of neuron groups, then the size of these groups and their connectivity to other groups is modified iteratively, and finally neurons and their connections are generated based on the group sizes and connectivity. This method was used to create networks which controlled a starfish-like animat.

Developmental neural networks are an ongoing research topic, albeit less well-known than other modern neural network research. Adaptive spiking neural networks are created by development in [SBT09]. In [KMH11], programs are evolved to control different components of neural development. In [MWC18], two evolved programs control development: one controlling the somata, the cell body of the neurons, and the other controlling the dendrites, creating or removing connections between neurons. This model was able to develop different networks to solve multiple machine learning tasks using the same developmental programs. Further examples of developmental neural networks are given in chapter 5.

Neuroevolution is far from a new research topic; [Dow15] provides an excellent overview of the variety of neuroevolutionary methods, many of which are well-studied. A review of evolving plastic, or learning, neural networks is presented in [SSR18], which highlights the many cases of neuroevolution used in tandem with learning methods such as Hebbian learning. However, neuroevolution is currently experiencing a resurgence of interest, largely due to the success of deep learning. In [Mii+18], neuroevolution is applied to deep learning, automatically creating deep architectures. Due to the computational cost of evaluating many different architectures, these experiments are currently prohibitively expensive. [SZB18] presents a method for CGPANN which reduces the computational load of evolution by only evaluating certain individuals based on their genetic distance to others. Such methods can increase the viability of neuroevolution, allowing for new application of neural networks not constrained by stochastic gradient descent learning.

2.6 Objectives of the thesis

In this chapter, we presented an overview of the neuron, from the function of the individual neural cell, to the formation and design of neural connections, to methods of learning in biological and artificial neural networks. Evolutionary computation was described, and the domain of neuroevolution, the application of evolutionary computation to artificial neural networks, was highlighted.

The work in this thesis is intended to develop indirect neuroevolutionary methods. By evolving controllers for cell function, connectivity, and learning, we aim to use evolution to discover design principles important for artificial neural networks. This separates this

work from existing neuroevolutionary methods in certain significant ways.

First, we do not focus on the evolution of synaptic weights, either directly or indirectly. Evolution can be a very effective optimization method for finding synaptic weights, providing an advantage over gradient descent learning methods due to the flexibility of the objective function definition. Studies in neuroevolution which evolve synaptic weights are numerous, being used from as early as 1994 [RS94] to today [Sal+17]. Some of these methods use direct synaptic weight encoding, such as [Sal+17], and others are indirect. HyperNEAT evolves a program which determines the synaptic weight based on presynaptic and postsynaptic neuron information and has been demonstrably effective at a number of tasks [SDG09], [RLS10], [Hau+12].

We choose not to evolve synaptic weights for two reasons. The first is due to the dissonance with biology: synaptic weights are not encoded in the genome, but rather are the result of a complex process involving multiple agents, the rules for which are encoded in the genome. The second reason is that we hope to use evolution to gain insight into those rules, instead of using evolution to produce an ANN fit for a certain task. By requiring the learning of synaptic weights as we examine each component, we involve the learning process at each step of our study, allowing for a more complete understanding of it.

Secondly, we focus on emergent intelligence, evolving rules for local components and interactions instead of evolving global components or rules for the entire ANN. The field of neuroevolution has examples of both local and global approaches. The evolution of deep architectures in [Mii+18] is a global approach, where modules of layers of neurons are evolved using NEAT to create deep neural networks. CGPANN is also global, as the genome represents the entire neural architecture and weights [KKM10a]. HyperNEAT can be viewed as a local approach, as the evolved function for deciding synaptic weights uses only local information (the position, (x_i, y_i)) about the connected cells. However, the original motivation for this function was to decouple neuroevolution from the constraint of local rules [Sta07], the conception of the weight pattern as a 4 dimensional hypercube (x_i, y_i, x_j, y_j) is a decidedly global viewpoint, and the application of the function, from one neuron pair to the next, is a global computation. ES-HyperNEAT builds on this hypercube representation and decides the placement of neurons based on global information [RLS10].

Developmental neuroevolution approaches are often local. Cellular encoding [GW93] uses a grammar which is applied at each cell, developing a network using a series of local interactions. The soma-dendrite developmental model of [MWC18] is local; the soma program and dendrite program only receive information about connected components and are computed at each cell. The placement of computation is an important point of distinction: whether the cell computes its own function, or has its function computed in a global iteration. For simple functions, this may be little more than an implementation

difference, but as will be shown in chapter 3, controller functions can benefit from having state, remembering their previous inputs and outputs. When calling these functions inside the cell, the inputs and outputs remain local, but when calling them over a global iteration, as in HyperNEAT, global and local information are mixed.

In this thesis, we use local information, computed inside the relevant component. Our motivation for this is again to align with our goal of discovery; by evolving the rules which control local agents in dynamic ANNs and studying those rules, we aspire to gain insight into specific agent behavior and interactions. Specifically, we aim to understand how a series of local interactions in ANNs can lead to emergent intelligence.

In the next section, we expand on upon the study of evolutionary computation with new work in the study of two evolutionary methods, Artificial Gene Regulatory Networks and Cartesian Genetic Programming. Afterwards, we apply these methods to the neuron, following the same order as in this chapter. First, we examine cell function and evolve a spiking neural activation function using CGP. Then, we evolve AGRNs to control axon guidance, which creates the structure of a spiking NN. Finally, we study learning in two cases; in the first, we evolve the parameters of a novel reward-modulated STDP method using CMA-ES, and in the second, we evolve AGRNs to optimize the parameters of learning at each layer during the training of a deep neural network.

The background in this chapter have provided as insight and foundation for the upcoming chapters, all of which represent new work completed during the course of this thesis. This work can be understood as an advancement in neuroevolution, although each work focuses on specific components of the neural network, as opposed to evolving the structure or rules for entire networks. By using this compartmentalized approach, we hope to gain understanding of the important underlying principles for artificial neural design through evolution.

Chapter 3

Evolving controllers

To understand and model emergent intelligence, we focus on the behavior of neural agents. As described in Downing's framework for emergent AI, agents with a genotype and phenotype are exposed to an environment, in which their fitness is assessed. In all following chapters, the environments for assessing agents are artificial neural networks. Agents act as neurons, axon growth cones, or neuromodulators, influencing the behavior of an ANN as this ANN performs a task. The genes of this agent are then carried on to the next generation depending on the performance of the ANN, influenced by the agent's phenotypic behavior.

Agents make their decisions by taking in inputs from the environment, passing these inputs through an internal controller, and acting on the outputs of the controller. Controllers can range from simple rules to complex networks of millions of parameters. They can be designed by hand or automatically through a process like evolution. Throughout this work, we use two controller types, Artificial Gene Regulatory Networks (AGRN) and Cartesian Genetic Programming (CGP). These controllers each use a specialized evolutionary process, which we study in this chapter.

To ensure that the evolved controllers performed as well as possible, and to fully understand their usage in different agents and environments, we conducted parameter studies for the two controller representations used in this work. These studies are presented in this chapter. Both of these studies represent novel advances for the concerned controller and demonstrate the optimal usage for a variety of cases, which later informed their use in neural models. The experiments in both studies use different environments than neural components; we examine locomotion, obstacle avoidance, game playing and more. The motivation was to fully understand these controllers and demonstrate their capabilities, including but not limited to their use as neural component controllers.

The first controller presented is the Artificial Gene Regulatory Network. This algorithm is based on biological gene regulatory networks; the complex interaction of proteins

encoded by genes which enhance and inhibit other proteins in the network. AGRNs have a long history of use as controllers for a variety of tasks, from vehicle driving to cell behavior during artificial organism development.

The second controller is Cartesian Genetic Programming. CGP is a form of Genetic Programming (GP), where computational structures, like functions, circuits, abstract syntax trees, and programs, are evolved using an EA. CGP has been used many times in tandem with neural network models, acting as an means in neuroevolution in in CGPANN [Mah+13] and controlling synaptic development in [KMH11] and [MW17].

In the next sections, we present AGRNs, CGP, studies of their evolution, and results demonstrating their capabilities. Both studies are considered here in the context of evolving controllers, and the backgrounds of the two methods, especially concerning neural network development, are expanded upon. These studies inform the use of AGRNs and CGP in the works presented in the following chapters.

3.1 Artificial Gene Regulatory Networks

Biological genes interact in a complex network of protein regulation. Gene expression triggers the manufacture of certain proteins, which can enhance or inhibit the expression of other genes, boosting or limiting the transcription of other proteins. These interactions lead to various behaviors of the network as a whole; proteins may oscillate in a regular pattern under one condition or turn chaotic in another. External factors, the presence of proteins detected by receptors or derived from sources such as food or toxins, trigger a response from the network. The behavior of the expressed proteins, their response to external factors and to self-regulation, is encoded in the genes, in RNA and DNA.

The complex behavior of biological GRNs serves as great inspiration for an artificial agent, and the basis of GRNs in DNA and RNA allows for artificial evolutionary models to remain close to their biological counterparts. Early artificial GRN models used encoding schemes heavily inspired biology. In [Ban03a], binary genes encode an AGRN, with transcription sites and organization in codons. This AGRN model was shown in [Ban03b] to be capable of exhibiting many of the behaviors seen in biological GRNs.

Since then, AGRN models and the means of evolving AGRNs have undergone many changes. GRNEAT, a GA specifically designed for AGRN evolution, was presented in [CHP15] and showed improvement on a number of tasks. GRNEAT is designed around the AGRN model used in this study, which itself is based originally on the model presented in [Ban03a]. However, other models and implementations exist with a variety of differences. Parameters, encoding methods, and dynamics formulas all differ in the AGRN literature, and some AGRN implementations have a high degree of model complexity.

We present a comparative study of different implementations of the AGRN and intro-

duce new variants for comparison. We use GRNEAT to evaluate AGRN performance on a number of common benchmark tasks, with a focus on real-time control problems. We propose an encoding scheme and set of dynamics equations that simplifies implementation and evaluate the evolutionary fitness of this proposed method. Lastly, we use the comparative modifications study to demonstrate overall enhancements for AGRN models.

The complexity of implementation combined with the variety of AGRN models in the literature can be a deterrent from the use of AGRNs. This study evaluates whether or not the AGRN model can be simplified without losing performance quality. We use a set of signal processing and control tasks, both of which are common problem types for the AGRN. Through our comparison of multiple AGRN implementations, we propose individual improvements as well as a general best AGRN model.

First, we provide a brief overview of the history of application of AGRNs.

3.1.1 AGRN applications

The capability of AGRNs as controllers has been displayed in a number of different domains.

Signal processing has been a natural domain for AGRN application. The oscillatory behavior of protein concentrations facilitates the evolution of signal amplifiers, modifiers, and filters. Both [JW10a] and [CHP15] demonstrate that AGRNs can be evolved to produce and modify complex signals.

AGRNs have been used to drive virtual cars and robots. In [SC14], an AGRN controls a virtual cat in TORCS, a simulated car environment. Sensors from the car are given to the AGRN as inputs, and outputs control the wheel, accelerator, and brakes. This approach won the Simulated Car Racing Championship in 2015[1]. Joachimczak used AGRNs in a robotic foraging problem, where a two-wheeled robot collected randomly placed food particles and avoided poisonous ones ([JW10b]). An AGRN robot controller was also used in [Tre+10] to perform obstacle avoidance.

AGRNs have seen extensive use artificial embryogenesis contexts, similar to their original biological motivation. In most of these applications, cells are placed in an environment and are able to reproduce and differentiate. The cell actions are controlled by an AGRN; the same AGRN is placed inside each cell, but the inputs are specific per cell. Inputs can include sensing from neighboring cells, chemical signals (morphogens) from the environment or other AGRNs, and more.

The French flag problem, where the red, white, and blue pattern of the French flag must be reproduced, has been approached from a cell-based model in multiple works using AGRNs. In [CD08b], the goal was to explore the coevolution of shape and color control,

[1] http://cs.adelaide.edu.au/~optlog/SCR2015/index.html

both by the same AGRN. This problem was extended to the third dimension in [JW09], which also included a means of self-repair. When cells were killed during development, the organism was able to generate new cells to attain the desired shape.

Cussat-Blanc developed multi-cellular creatures from a single cell in [CLD08], where cells acted in an environment of substrates, modifying their environment to communicate with other cells. In [DS14], virtual creatures composed of many cells, termed animats or multicellular robots, develop into mature organisms optimized for walking or kicking a ball. Novelty, the selection of individuals during evolution based on their difference from the rest of the population as opposed to their fitness on a task, was demonstrated to aid the evolution of AGRNs for artificial embryogenesis in [DCD16]. In that work, virtual creatures were evolved in a divided environment with a nutrient-rich ground and a surface exposed to sunlight. The virtual creatures required both nutrients and sunlight to survive, and therefore developed like plants, with root structures and differentiation above ground.

The process of cellular development was extended to the application of wind farm layout optimization in [Wil+13]. Cells divided over a wind farm, with cellular division, the direction of division, and cellular apoptosis controlled by an AGRN. A turbine was placed in the center of each cell at the end of development, creating efficient layouts. This used a grid layout and later a spring-based cellular model for continuous space development ([Wil+14]).

Finally, AGRNs have a history of application to ANNs, notably in the Gene Regulatory evolving artificial Networks (GReaNs) platform ([WA12]). In this platform, spiking neural networks are extracted from the structure provided by the AGRN by translating the protein excitation and inhibition signatures into synaptic weights and treating the protein concentrations as spiking neural networks. The effectiveness of this platform on signal processing, control, and development was shown in [WJ14].

Neuromodulation has been demonstrated using SARSA in [CH15], where AGRNs controlled the learning of an agent in many common reinforcement learning benchmarks, including mountain car, maze navigation, and acrobat. Nicolau originally showed the capabilities of AGRNs on a single pole problem in [NSB10].

There has also been considerable study into the behavior and improvement of AGRNs. AGRN topology is examined in [DBL06] to determine if evolved networks were scale-free and small-world, which are considered hallmarks of natural evolution. The timing dynamics of AGRNs were studied in [Kna+06]. The benefits of variable-length AGRNs are demonstrated in [Tre+10].

3.1.2 AGRN overview

The AGRN described in this section was designed in [Ban03a], with modifications made in [CD08b]. The components under review in this study are further explored in subsection 3.1.3.

A AGRN is composed of multiple artificial proteins, which interact via evolved properties. These properties, called tags, are

- The protein *identifier*, encoded as an integer between 0 and u_{size}. u_{size} can be changed in order to control the precision of the AGRN.
- The *enhancer identifier*, encoded as an integer between 0 and u_{size}. The enhancer identifier is used to calculate the enhancing matching factor between two proteins.
- The *inhibitor identifier*, encoded as an integer between 0 and u_{size}. The inhibitor identifier is used to calculate the inhibiting matching factor between two proteins.
- The *type*, either *input*, *output*, or *regulator*. The type is a constant set by the user and is not evolved.

Each protein has a concentration, representing the use of this protein and proving state to the network similar to neurotransmitter concentrations in spiking neural networks. For *input* proteins, the concentration is given by the environment and is unaffected by other proteins. *output* protein concentrations are used to determine actions in the environment; these proteins do not affect others in the network. The bulk of the computation is performed by *regulatory* proteins, an internal protein whose concentration is influenced by other *input* and *regulatory* proteins.

The dynamics of the AGRN are calculated as follows. First, the absolute affinity of a protein a with another protein b is given by the enhancing factor u^+_{ab} and the inhibiting u^-_{ab}:

$$u^+_{ij} = u_{size} - |enh_j - id_i| \quad ; \quad u^-_{ij} = u_{size} - |inh_j - id_i| \tag{3.1}$$

where id_x is the identifier, enh_x is the enhancer identifier and inh_x is the inhibitor identifier of protein x. The maximum enhancing and inhibiting affinities between all protein pairs are determined and are used to calculate the relative affinity, which is here simply called the affinity:

$$A^+_{ij} = \beta(u^+_{ij} - u^+_{max}) \quad ; \quad A^-_{ij} = \beta(u^-_{ij} - u^-_{max}) \tag{3.2}$$

β is one of two control parameters used in a AGRN, both of which are described below. Variants of this equation used in this study are detailed in subsection 3.1.3.

These affinities are used to then calculate the enhancing and inhibiting influence of each protein, following

$$g_i = \frac{1}{N}\sum_{j}^{N} c_j e^{A_{ij}^{+}} \quad ; \quad h_i = \frac{1}{N}\sum_{j}^{N} c_j e^{A_{ij}^{-}} \tag{3.3}$$

where g_i (resp. h_i) is the enhancing (resp. inhibiting) value for a protein i, N is the number of proteins in the network, c_j is the concentration of protein j.

The final modification of protein i concentration is given by the following differential equation:

$$\frac{dc_i}{dt} = \frac{\delta(g_i - h_i)}{\Phi} \tag{3.4}$$

where Φ is a function that normalizes the output and regulatory protein concentrations to sum to 1.

β and δ are two constants that determine the speed of reaction of the regulatory network. The higher these values, the more sudden the transitions in the AGRN. The lower they are, the smoother the transitions. For this study, they are evolved as part of the AGRN chromosome and are both kept within the range $[0.5, 2.0]$.

In this study, AGRNs are evolved using Gene Regulatory Network Evolution Through Augmenting Topologies (GRNEAT), a specialized Genetic Algorithm for AGRN evolution proposed in [CHP15] and based on Stanley's NeuroEvolution of Augmenting Topologies (NEAT) algorithm ([SM02]). This algorithm has been shown to improve evolution performance on neural networks, and complex pattern producing networks, a type of evolved program ([SDG09]). A major contribution is the design of a crossover method for network controllers, in which structure has a significant influence on overall network behavior. GRNEAT uses a similar crossover, and imports three key elements from NEAT:

- the initialization of the algorithm - small networks are generated that resemble a select subpopulation termed initial species leaders
- speciation, which limits competition and crossover to similar individuals. This both protects some new mutants from immediately competing with champion individuals and protects novel solutions by allowing them to optimize their structures before competing with the whole population
- an alignment crossover that compares individual genes before selection for a new individual

The distance metric in this study for speciation and alignment crossover was

$$D_{prot}(i,j) = \frac{a|id_i - id_j| + b|enh_i - enh_j| + c|inh_i - inh_j|}{u_{size}} \tag{3.5}$$

where $a = 0.75$, $b = 0.125$, and $c = 0.125$. Proteins were aligned in each AGRN during comparison first based on *type* and secondly based on minimum $D_{prot}(A, B)$. The distance between AGRNs was then calculated as

$$D(G_1, G_2) = \frac{D_{in} + D_{out} + D_{reg} + D_\beta + D_\delta}{max(N_1, N_2) + 2}$$
$$D_\beta = \frac{\beta_1 - \beta_2}{\beta_{max} - \beta_{min}}$$
$$D_\delta = \frac{\delta_1 - \delta_2}{\delta_{max} - \delta_{min}}$$

where N_i is the number of proteins in AGRN G_i, and D_{type} is the sum of the difference of all aligned proteins of that *type*. For regulatory proteins, where N can differ between two AGRNs, the distance for all non-aligned proteins was taken between the protein parameters and u_{size}. The use of alignment without replacement for the distance metric is novel in this work and differentiates it from [CHP15]. This was found to improve results in preliminary trials but is not presented as a part of this study.

Apart from the differences listed above, GRNEAT functions as a standard GA. The mutation operations available during this work were

- Modify (probability $p_{modify} = 0.25$)
- Add a new regulatory protein with random parameters, $p_{add} = 0.5$
- Delete $p_{delete} = 0.25$

When mutating a genome, a random mutation operation was selected with probability p_{select}. The delete mutation operation was not allowed when selected for input and output genes.

During the crossover mutation, aligned proteins are randomly selected from either parent with probability $p_{cross} = 0.5$. If the parent genomes are of different lengths, the unaligned regulatory proteins from the longer parent genome are appended to the child genome with probability $p_{append} = 0.5$.

3.1.3 AGRN dynamics

In this study, we focused on specific improvements to the AGRN that either vary in the literature or could be used to simplify AGRN implementation. These modifications impact the AGRN encoding e, affinity metric a, the influence function f, the and the normalization step n. For each of these modifications, we evaluate the potential fitness contribution in subsection 3.1.5.

Equation 3.4 can be generalized with f, a, and n as follows:

$$\frac{dc_i}{dt} = n(\frac{\delta}{N}\sum_{j}^{N} c_j(f(a^+(i,j)) - f(a^-(i,j))) \tag{3.6}$$

with the encoding e changing the evolution dynamics, the parameter and the possible range of u_{ij}. e, a, f, and n are all described below.

In early versions of AGRNs, and in some modern implementations, proteins were encoded in binary format. The affinity between two proteins was defined by the number of bits in common, and mutation and crossover operations happened at a binary level ([Ban03a]). Here we strive to simplify this model by proposing real values between $[0, 1]$ for the protein tags. The mutation operation is altered as a result, as the distance change from a mutation can be smaller than an integer step size. The aligned crossover operation is also affected, as the distance between proteins operates on real values. Lastly, u_{size} is set to 1.0 in this encoding, which affects the affinity metric.

In [Ban03a] and in many works since, the affinity metric uses the maximum affinity metric, u_{max}, as a scaling factor. This is the maximum of the relative affinity metric, $u_{size} - |enh_j - id_i|$, (resp inh) across all (i, j). However, for a reasonably large network, in which two protein tags will become arbitrarily close, u_{max} will approach u_{size}. For this reason, we propose the following novel affinity metric, which reduces complexity by removing the maximization factor:

$$A_{ij}^+ = -\frac{\beta|enh_j - id_i|}{u_{size}} \quad ; \quad A_{ij}^- = -\frac{\beta|inh_j - id_i|}{u_{size}} \tag{3.7}$$

In comparison with the original equation, it was noted that on some implementations, only u_{ij} was multiplied by β. This results in the following equation:

$$A_{ij}^+ = \beta(u_{size} - |enh_j - id_i|) - u_{max}^+ \tag{3.8}$$
$$A_{ij}^- = \beta(u_{size} - |inh_j - id_i|) - u_{max}^- \tag{3.9}$$

Lastly, the original affinity metric is evaluated:

$$A_{ij}^+ = \beta(u_{size} - |enh_j - id_i| - u_{max}^+) \tag{3.10}$$
$$A_{ij}^- = \beta(u_{size} - |inh_j - id_i| - u_{max}^-) \tag{3.11}$$

In [Ban03a], [DBL06], and many others, an exponential function of the affinity is used to determine the influence of one protein onto an other. This is the first influence function we evaluate:

$$f(A_{ij}) = e^{A_{ij}} \tag{3.12}$$

$s = 0$	integer encoding scheme
$s = 1$	real encoding scheme
$a = 0$	Equation 3.7, simply using u_{ij}
$a = 1$	Equation 3.8, with u_{max} outside β's influence
$a = 2$	Equation 3.10, the original equation
$f = 0$	Equation 3.12, e
$f = 1$	Equation 3.13, $tanh$
$f = 2$	Equation 3.14, inverse e
$n = 0$	normalization of concentrations by their sum
$n = 1$	constraining all concentrations to $[0.0, 1.0]$

Table 3.1: GRN modifications evaluated and their corresponding labels

In [Kna+06], a hyperbolic tangent function is used. The constants of this implementation were modified to provide the same results as Equation 3.12 at $A_{ij} = 0$:

$$f(A_{ij}) = tanh(A_{ij}) + 1 \tag{3.13}$$

[JW10b] uses an inverse exponential, and also decreases the protein concentration by this influence instead of increasing. As such, the inverse exponential in this study is modified to match the same relationship as the other metrics, and the constants are again modified to provide the same results as Equation 3.12 at $A_{ij} = 0$:

$$f(A_{ij}) = \frac{2}{1 + e^{-A_{ij}}} \tag{3.14}$$

An important component of AGRN dynamics is the normalization of protein concentrations at each step, such that the output and regulatory protein concentrations sum to 1. This makes the output layer of the AGRN function similarly to the softmax layer of modern ANNs, and is often useful in problem implementation. However, it can also be a difficult concept to grasp when understanding AGRNs, increases implementation and computation complexity, and requires a knowledge of common AGRN inputs and outputs for good problem design. For example, in the experiments in subsection 3.1.4, some of the control problem outputs are designed with normalization in mind by forcing an action only when one output concentration exceeds another, favoring the periodic dynamics resultant from this normalization step.

In this study, we propose the simple use of boundaries $[0, 1]$ for protein concentrations as an alternative to normalization. We refer to this method as capping.

In review, the modifications proposed, and the variables used to denote them, are as listed in Table 3.1. We propose the model corresponding to $s = 1$, $a = 0$, $f = 0$, $n = 1$

as a simplified AGRN. The dynamics of this AGRN implementation are, according to its modifications:

$$\frac{dc_i}{dt} = \frac{\delta}{N} \sum_{j}^{N} c_j (e^{-\beta|enh_j - id_i|} - e^{-\beta|inh_j - id_i|}) \tag{3.15}$$

$$c_{i,t+1} = (c_{i,t} + \frac{dc_i}{dt})\Big|_0^1 \tag{3.16}$$

This AGRN formula reduces complexity by using the simplest affinity metric, $a = 0$, which does not include determining u_{max}. It is a real encoding, which removes the determination of the u_{size} parameter. Finally, it uses a min-max step, $n = 1$, instead of normalization of protein concentrations, which is computationally complex and another implementation step. We evaluate the performance of this model to determine if it is equally capable, as well as evaluating all modifications independently.

3.1.4 AGRN experiments

To evaluate the impact of each modification mentioned above, we have used standard problems from a broad spread of the literature. More specifically, AGRNs are often used as real-time controllers, as they are in the following signal processing, robot control, and game problems.

For each problem, all 36 AGRN modification combinations (2 encoding schemes, 3 affinity equations, 3 influence functions and 2 normalization methods) were evaluated over 40 runs. The parameters of the evolutions can be found in Table 3.2.

The code used for all following experiments, including the problems and the details of their parameters, the AGRN and GRNEAT, are available in C++ on GitHub[2]. Video of the best performing AGRN on the Ship Escape problem is also available.

In this first problem, from [JW10a], a sinusoidal signal of frequency F_i is fed to the AGRN by varying the concentration of its only input protein. The goal of the AGRN is to make the concentration of its output protein to vary at twice the input frequency, i.e $F_o = 2 * F_i$, with F_o being the variation frequency of the output protein. We used the same fitness function as in subsection 3.1.4, being the sum of the absolute distances between the desired and obtained signal at each time step, divided by the absolute distance between F_o and F_i. The input signal is divided into three sequences of equal length (1000 time steps) and at frequencies equals to 125Hz, 500Hz and finally 0Hz (flat signal). The proteins concentrations are reset between each sequence.

[2]https://github.com/jdisset/grnbenchmarks/

initial population	500
generations	300
tournament size	3
minimum species size	15
number of elites per species	1
speciation threshold	0.3
maximum speciation threshold	0.8
minimum speciation threshold	0.01
mutation rate	0.75
crossover rate	0.25

Table 3.2: GRNEAT parameters used in all experiments

In another classic signal treatment problem from [JW10a], the AGRN must act as a low pass filter, meaning it must strip the input signal of any frequency greater than the cut-off frequency $F_c = 50$Hz. Here, the fitness is the average squared distance between the output signal (scaled by a constant factor $C = 5$ in order to not penalize the normalized concentrations implementations).

The input signal is divided into three sequences of equal length (1000 time steps):

- A signal composed of 3 combined subsignals: one at 7Hz with an amplitude of 0.7, another at 250Hz with an amplitude of 0.2 and the last one at 1250Hz and an amplitude of 0.1. The desired output signal should have a frequency of 7Hz with an amplitude of 0.7

- A signal composed of 4 combined subsignals: one at 17Hz with an amplitude of 0.4, another at 350Hz with an amplitude of 0.2n, a third one with a frequency 1100Hz and an amplitude of 0.2 and the last one at 2000Hz and an amplitude of 0.1. The desired output signal should have a frequency of 17Hz with an amplitude of 0.4

- A flat "zero" signal, which should be exactly reproduced at the output.

The AGRN proteins concentrations aren't reset between each sequences.

The next problem is a classic coverage problem where the AGRN controls a robot in a 2D grid. It has 8 inputs: the number of obstacles on the next 3 grid cells in each four directions (north, south, east, west), one protein per direction, and the number of unexplored cells in each directions. It has 4 output proteins (one for each direction), the protein with the highest concentrations deciding the direction in which the robot will move at the next time step. Each AGRN runs for 200 steps on 3 different 10 by 10 maps with 20 obstacles and the fitness is the average of the discovered portions of the maps.

In order to challenge the capabilities of a AGRN as a game AI controller, we implemented a version of the famous small game Flappy Bird. In this game, a small bird progresses through an horizontal world, bounded by a ceiling and a floor. It must pass through gates whose positions and aperture height are randomly generated. The only control the player has over the bird is the timing of its wings' flaps, which provide upward thrust. The horizontal speed increases over time. Here, we defined three input proteins for our AGRN, whose concentrations respectively corresponds to:

- the bird's height, normalized by the height of the world
- the next gate's position, normalized by the length of the screen
- the next gate's aperture height, normalized by the height of the world

The flap's timings are controlled by concentrations c_{o1} and c_{o2} of its two output proteins. The bird flaps its wings each time $c_{o1} > c_{o2}$. The fitness is equal to the average horizontal distance at which the bird first hit a gate, the floor, or the ceiling.

The last problem requires the most inputs and outputs. Here, the AGRN must learn to drive a ship in a vertical world bounded by walls and filled with randomly placed obstacles. The goal is to drive the ship as far as possible without hitting anything, with the added dificulty of gates slowly closing ahead of the ship. The distance between two gates increases after each passed one, and the next gate starts to close as soon as the previous one has been passed through. This puts pressure on the ship to go accelerate while still avoiding the obstacles. For this problem, we used 13 inputs: 11 of them represented laser beams casted by the ship in 11 evenly distributed directions, from $\frac{\pi}{2}$ to $-\frac{\pi}{2}$ relatively to the ship direction, each beam directly setting the concentration of an input protein C_i as equal to $\frac{D_i}{H}$, i.e the distance betweed the ship and the nearest obstacle in the direction of the beam, normalized by a maximum distance H. The remaining two inputs are indications of the ship's orientation, being set respectively as $sin(\theta)$ and $cos(\theta)$, with θ the current oriented angle of the ship.

The ship also needs to have output that allow it to control its direction and its propulsion. To do so, we add 3 pairs of output proteins: one pair that will allow the ship to turn left when the concentration of the first protein of the pair goes above the concentration of the other, one pair to turn right using the same principle, and one pair to turn the engine on using again the same principle.

3.1.5 AGRN results

To independently compare the impact of each modification, we first compare pairs of implementations with only one modification, such as $e = 0$, $a = 0$, $f = 0$, $n = 1$ to $e = 0$,

$a = 0$, $f = 2$, $n = 1$. For each pair, the 40 runs were used to fit two normal distributions using maximum likelihood estimation, $N(\mu_1, \sigma_1)$ and $N(\mu_2, \sigma_2)$. The probability that values from one distribution are greater than the values from the second distribution is used to compute a Competitive Probability Score (CPS). The CPS of a modification is the summed probability difference for each modification pair. To determine this, first the competitive probabilities for implementations with all but one modification in common are summed, here shown for f:

$$P_f[i,j] = \frac{1}{\eta} \sum_i \sum_j 1 - \Phi(\frac{\mu_2 - \mu_1}{\sqrt{\sigma_1 + \sigma_2}}) \tag{3.17}$$

η is the number of implementations in the sum, and Φ is the cumulative distribution function of the normal distribution $N(0,1)$, making $P_f[i,j]$ the average probability that modification $f = i$ is greater than $f = j$. The CPS is then simply the sum:

$$CPS(f = i) = \sum_j P_f[i,j] \tag{3.18}$$

The global best AGRN implementation was determined by the same process as the CPS, only over entire implementations. Each of the 36 implementations were compared and a normal distribution was fit to the results from their last generation. The probability of each one exceeding the other was summed and the implementation with the highest average probability over all other implementations was chosen. The top 5 implementations using this metric are shown in Table 3.3.

	s	a	f	n
0.738265	1	1	0	0
0.726697	1	0	1	0
0.710317	1	0	0	0
0.703542	1	1	2	0
0.697183	1	2	1	0

Table 3.3: The top five implementations based on CPS

The clear trends from the results are the advantage of the real encoding and the necessity of the normalization step. The best implementation, $s = 1$, $a = 1$, $f = 0$, $n = 0$ is surprising to us given the use of $a = 1$, which uses a non-scaled u_{max}. We believe this result to be sensitive to the β range, and that a clear affinity metric is not determined by the results. Similarly, it seems that $f = 0$ is the best influence function, but this is not as conclusive as the advantages of $s = 1$ and $n = 0$.

Analysis of variance (ANOVA) tests were also conducted for each problem. Groups were constructed for each implementation within a method, such as a group corresponding

	s	a	f	n
doubling	9.176e-38	1.069e-08	1.215e-03	1.597e-07
lowpass	3.044e-16	0.012	0.725	0.544
coverage	2.572e-18	0.046	0.080	3.645e-17
flappy	8.808e-20	0.089	0.233	0.062
ship	1.196e-16	0.175	0.715	0.150

Table 3.4: One-way ANOVA p values between the different implementations for each problem

to $s = 0$ and another to $s = 1$, and the variance of fit distributions to the groups were evaluated. The p values from this analysis are presented in Table 3.4. The findings of this analysis show a clear difference across problems in s and low p values for most problems on a and n. These results demonstrate that the implementations can be significantly different depending on the problem.

While the simplified AGRN proposed, $s = 1$, $a = 0$, $f = 0$, $n = 1$, was overall the seventh best implementation, the second best and third best implementations use $a = 0$, the simplest affinity metric, and all of the top implementations use $s = 1$, which simplifies encoding and reduces parameters.

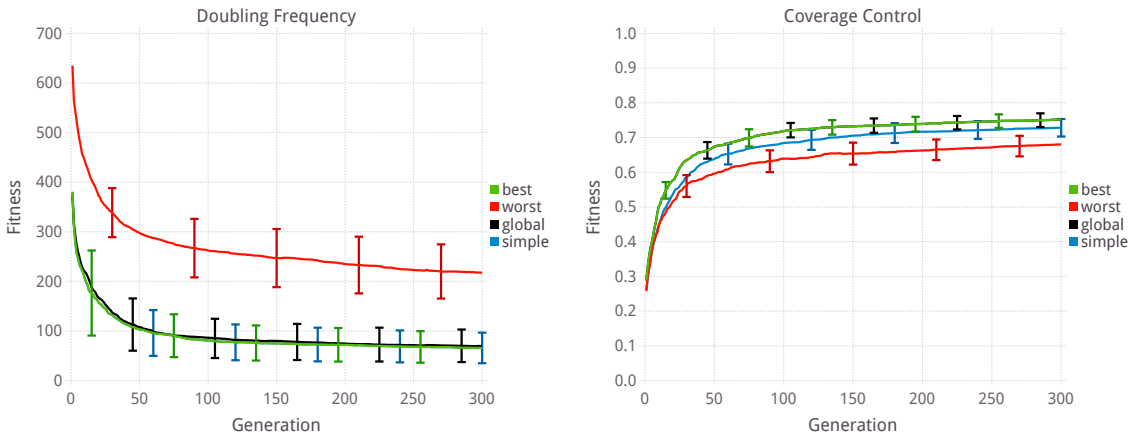

Figure 3.1: The best implementation $s = 1, a = 0, f = 0, n = 1$ and the worst implementation $s = 0, a = 0, f = 2, n = 1$ of the doubling frequency problem compared to the global best and the proposed simplified AGRN (left), and the best implementation $s = 1, a = 1, f = 0, n = 1$ and the worst implementation $s = 0, a = 0, f = 0, n = 1$ of the coverage control problem (right)

In the signal processing benchmark problems, the top implementations were not significantly different and all performed similarly to results found in [CHP15]. While some implementations fared very poorly, we found these results not independently conclusive for determining implementation fitness. While the low pass experiment has the only best implementation using $s = 0$, it was not a significant advantage over the other implemen-

tations on this rather simple problem. The lowpass results are omitted due to space; all methods performed similarly on this task, quickly solving it.

The implementations used here show improvement on the coverage control problem over [CHP15], but performance didn't vary significantly over different implementations. On this problem, the best implementation is also the global best.

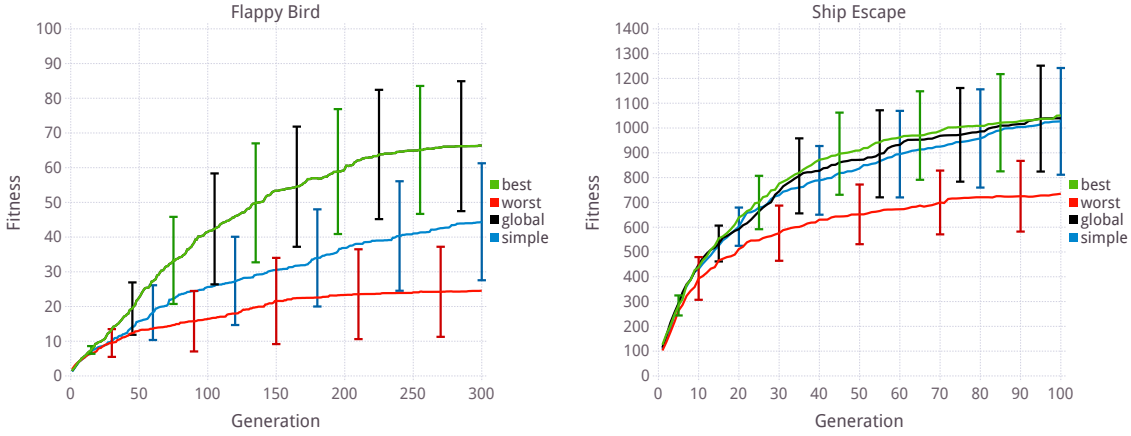

Figure 3.2: Comparison with the best implementation $s = 1, a = 0, f = 1, n = 0$ and the worst implementation $s = 0, a = 2, f = 0, n = 1$ of the Flappy Bird problem (left), and the best implementation $s = 1, a = 2, f = 1, n = 0$ and the worst implementation $s = 0, a = 2, f = 0, n = 0$ of the ship escape problem (right)

The results on the Flappy Bird problem are impressive for their variety and for the best performance. A human user scored an average of 20.349 over 20 trials, and we find it difficult to believe that a human player could achieve the same scores as the best AGRN. As in the Flappy Bird problem, the AGRN performance on the Ship Escape problem rivals or outperforms human capability.

Overall, this study demonstrated the capabilities of AGRNs on a variety of problems, including control tasks. AGRNs can achieve human levels of performance in responsive situations, like Flappy Bird and ship escape, and reach these levels with a relatively inexpensive evolution. The insights gained from this studies, the use of real-valued proteins and the need for normalization, informed further work throughout this manuscript. Other implementations of AGRNs used in this work use $s = 1, f = 0, n = 0$, meaning a real-valued AGRN with normalization and exponential protein interactions, and either $a = 1$ or $a = 0$, which had less influence on the final outcome. Specifically, the need for the normalization step informed the design of the integration of AGRNs into deep neural networks, which will be presented later in this chapter.

We now present a similar study in Cartesian Genetic Programming. We detail an overview of CGP and present a floating-point encoding of it, which is then used to explore possible genetic operators and parameters for CGP. As in this study on AGRNs, CGP

is evaluated on a variety of benchmark tasks, i.e. classification, regression, and robotic control.

3.2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a form of Genetic Programming (GP) where program components are represented as functional nodes in on two dimensional grid ([Mil11]). Connections between these nodes are made based on their Cartesian coordinates and create a final computational structure. The node coordinates, the function of each node, and occasionally function parameters or node weights are encoded in a genome which is evolved using a $1 + \lambda$ EA.

Originally created for circuit design in [MT00], CGP has since been shown to have impressive results in many domains. Many modifications and novel versions of CGP have been proposed, leading to a large body of work inside the field of Genetic Programming dedicated to branches of the original CGP algorithm. In this study, we present a brief overview of these applications and CGP-based methods, and then conduct experiments to gauge the utility of certain modifications.

In [Mil04], CGP was demonstrated on the French flag problem. This used a cellular approach to the problem, similar previously mentioned AGRN methods, and included self-repair. Another standard problem, the "lawnmower" or coverage problem, was used in [WM06]. In this problem, a robot must cover the entire area, navigating through it as efficiently as possible by not spending time re-covering certain areas.

CGP has produced impressive results in image processing. In [MW03], object detection on simple objects was performed using raw pixel data, with the goal of detecting the centroid of the presented objects. Image filters for denoising and image reconstruction were done in [Har08], which used a GPU implementation of CGP. This work was expanded in [HLS13], which used the OpenCV image library as the function set for CGP. In other words, CGP constructed programs by composing OpenCV functions together, creating novel and optimized image filters. In [PPN15], images with multiple geometric targets are filtered by CGP to isolate specific shapes from the original image, i.e. simple rectangles and circles, and also musical notes from sheet music in a second experiment.

The programs created by CGP can often be very simple and legible. While the genome represents many possible programs, the final program used by output nodes may be very small, even directly connecting to inputs. In [Wil+18b], Atari video games are played with CGP. The simple behavior of some programs, discernible through the actor's play style and through reading the underlying program generated by CGP, demonstrate novel solutions to problems considered complex benchmarks. These results are presented in more detail in subsection 3.2.2.

CGP also has a long history of use in creating artificial neural networks. Much of the work concerning CGP and ANNs has been done by Gul Muhammad Khan in his work with the creator of CGP, Julian F. Miller. In [KMH07], a developmental model of neural computation is proposed using CGP. This model is further studied in [KMH08], where it is used in two agents to play checkers. CGPANN, a method for evolving artificial neural networks using CGP, was proposed in [KKM10b]. In this method, neural network weights, topology, and neural functions are encoded as in the CGP chromosome, thus functioning very similarly to CGP when using weights. However, the functions using in CGPANN are the common activation functions for neural networks, being hyperbolic tangents and sigmoid functions.

CGPANN was first demonstrated on the double pole balancing problem, where two poles hinged on a moving cart must be balanced to stay within a specific angle range. Recurrent connections in CGPANN were studied in [KKM10a], also on pole balancing. These experiments are further discussed in [Mah+13], where breast cancer detection is also approached as a problem. CGPANN is still considered a leading method for neuroevolution, the direct evolution of neural networks; other examples include NEAT, HyperNEAT, and those covered in Chapter 2.

Indirect encodings of neural networks have also been explored using CGP. Specifically, CGP was been used for to explore developmental models where artificial neurons grow dendrites and form synapses, which can be destroyed during dendritic pruning. This leads to architectural learning, where the learning process of the ANN involves structural changes, controlled by a CGP. This idea was introduced in [KMH11], where a controller consisting of seven CGP chromosomes constructs an ANN capable of learning, itself controlling an agent in the Wumpus World problem. In this classic problem, an agent must avoid enemies (Wumpus) or eliminate them in order to traverse a space, find gold, and return back to its starting position.

A simpler developmental neural network method was explored in [MW17]. In this work, two CGP chromosomes control somata and dendrites, respectively, as dendrites grow or are pruned, attaching to somata to create synapses. These programs are run during a developmental stage, at the end of which a static ANN is extracted. Multiple problem types are presented to the developmental program, and a single ANN is extracted for each problem.

CGP has also been the subject of a number of studies concerning its dynamics, evolution, and properties when compared to other methods in Genetic Programming. Programmatic redundancy, the re-use of certain functional nodes, and the associated efficiency of resultant computational structures is examined in [MS06]. In [Mil01], the problem of bloat is examined in the CGP context. In other forms of GP, specifically tree-based GP, the combination of multiple individuals, i.e. genetic crossover, and the complexification

caused by evolution can lead to excessively large programs, a process known as "bloating". Bloat is not an issue in CGP, however, due to its fixed-length genome; the number of nodes in the genome, therefore the maximum number of possible nodes in the final program, is constant. Finally biases and limitations of search in CGP were presented in [GP13a], which inspired improvements detailed further in this study.

One such improvement is the reduction of wasted evaluations, proposed in [GP13b]. In this modification, only the parts of the CGP genome which are active in the final output graph are subject to genetic mutation. This was shown to reduce increase the efficiency of evolution by eliminating the evaluation of individuals that were identical to their parents in phenotype.

Other improvements to classic CGP include Mixed Type CGP (MT-CGP), which allows for the evolution of programs which take in mutliple types, namely numeric and array types ([Har+12a]). To achieve this, node functions change (overload) depending on the input type they are given. In [CWM07], a crossover method is proposed for CGP. This also facilitates the usage of CGP with a Genetic Algorithm for the first time, which uses a much larger population size that can be evaluated in parallel. Finally, [TM14] proposes recurrent cartesian genetic programming (RCGP), which allows evolution to create recurrent connections in CGP. Nodes in the recurrent programs are run in genetic order to avoid infinite loops.

Self-modifying Cartesian Genetic Programming (SMCGP) is an extension of CGP allowing live program modification. In this algorithm, functions are included in the function set which allow for the addition, removal, and modification of nodes in the program. These functions are run when activated during the evaluation of the program, meaning the program changes during its use. A survey of SMCGP is presented in [HBM10], and a later extension to two dimensions is given in [HMB11]. By having a program which changes during evaluation, SMCGP is able to solve problems which a static program would not be able to solve. SMCGP provided inspiration for some of the operators and representations in this survey.

CGP is an instance of graph-based GP, an attractive representation for computational structures given that they can reuse subgraph components and are used in many areas of computer science and engineering. In this work, we use ideas from other forms of graph-based GP to design new mutation and crossover methods for CGP. We also examine improvements to CGP that have been proposed, evaluating them as hyper-parameters and using a parameter search to determine when they are effective. These genetic operators and CGP enhancements are all evaluated as hyper-parameters on nine different benchmark problems, with three problems from each of the domains of classification, regression, and reinforcement learning.

Some of the new genetic operators are made possible by using a floating point rep-

resentation of the CGP genome, as is done in [CWM07], with the addition of 'snapping' connections which form connections to their nearest target node. Beyond enabling certain genetic operators, this representation allows for the evolution of the node positions itself, adding a new dimension to the CGP evolution. We term this representation Positional Cartesian Genetic Programming (PCGP) and evaluate it using the same hyper-parameter search used to evaluate CGP.

3.2.1 CGP representation

In its original formulation, CGP nodes are arranged in a rectangular grid of R rows and C columns. Nodes are allowed to connect to any node from previous columns based on a connectivity parameter L which sets the number of columns back a node can connect to; for example, if $L = 1$, nodes could connect to the previous column only. In this paper, as in others, $R = 1$, meaning that all nodes are arranged in a single row.

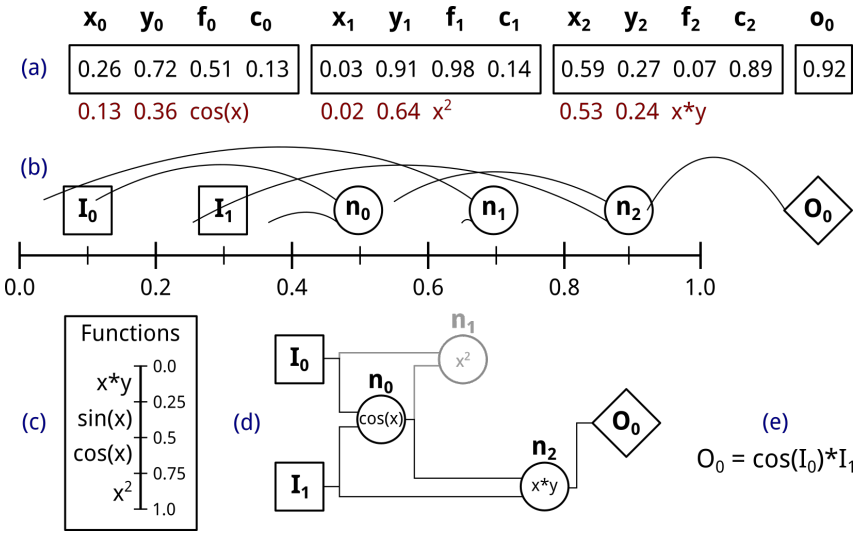

Figure 3.3: Translation of a floating point CGP genome into a program. The genome (a) is converted into positions and functions. The positions are informed by multiplying the connection genes, x_n and y_n, by the position of the nodes, shown in (b). The functions are found by indexing the function gene f_n in the Functions table (c). The resultant graph (d) is formed by "snapping" each connection to the closest node. As no output path uses n_1, this node is inactive or "junk". Finally, the graph can be read as a simple program (e).

In this work, a floating point representation of CGP is used. A similar representation was previously used in [CWM07], but involved translation from the traditional integer CGP representation to floating point. Here, floats are used throughout. All genes are floating point numbers in $[0.0, 1.0]$, which correspond to the connections of each node n, x_n and y_n, the node function f_n, and a parameter gene c_n which can be used for node weights or as a part of the function. Nodes are evenly spaced in one dimension between

0 and 1, with equal space around each node. Connections are formed by converting the connection genes x_n and y_n to coordinates by multiplying the genes by the node position, and then "snapping" these branches to the nearest node. An example of this process is shown in Figure 3.3.

Each program output has a corresponding gene which connects to a node in the graph. The output gene o_n specifies a connection which then "snaps" to the nearest node. By following connections back from the program outputs, an output program graph can be constructed. In practice, only a small portion of the nodes described by a CGP chromosome will be connected to its output program graph. These nodes which are used are called "active" nodes here, whereas nodes that are not connected to the output program graph are referred to as "inactive" or "junk" nodes. While these nodes do not actively contribute to the program's output, they have been shown to aid evolutionary search ([MS06]).

Two established CGP modifications are explored in this work: recurrent CGP and node weights. In RCGP, a recurrency parameter was introduced to express the likelihood of creating a recurrent connection; when $r = 0$, standard CGP connections were maintained, but r could be increased by the user to create recurrent programs. This work uses a slight modification of the meaning of r, but the idea remains the same. Here, the final connection position is modified by r:

$$p_{x_n} = x_n(r(1.0 - p_n) + p_n) \tag{3.19}$$

where p_n is the position of node n, x_n is its connection gene, and p_{x_n} is the position of the final connection. When $r = 0.0$, this is as in standard floating point CGP, as presented in Figure 3.3, where $p_{x_n} = x_n p_n$. when $r = 1.0$, the connection positions are simply the gene values, $p_{x_n} = x_n$. An example of this is shown in Figure 3.4. r in this work therefore indicates the end of the possible range of connections for a node, from that node's position p_n to the end of the positional space, 1.0.

In Figure 3.4, node weights are also used. In this scheme, the output of each node is multiplied by its parameter gene c_n. This CGP modification has allowed for differentiable CGP [IBM17] and is referred to in this work by the binary hyper-parameter w, which is true ($w = 1$) when weights are used.

3.2.2 Playing games with CGP

In order to demonstrate the benefits of CGP alone, before presenting the study of various CGP improvements, we present a brief overview of the results from [Wil+18b], where CGP was used to evolve game-playing agents. The games were emulated in the the Arcade Learning Environment (ALE, [Bel+13]), a popular benchmark set which has recently been

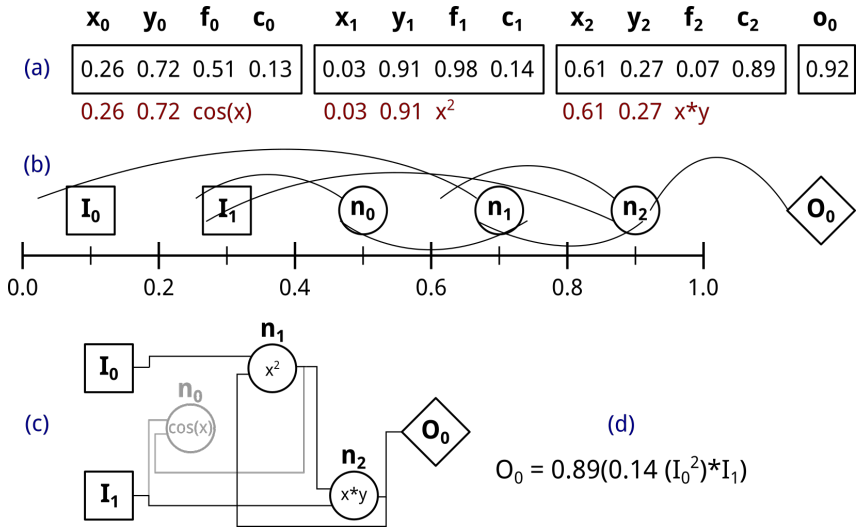

Figure 3.4: The same genome (a) as in Figure 3.3, but using a recurrency of $r = 1.0$ and node weights $w = 1$. The recurrency parameter changes the connection position calculation and allows nodes to connect to downstream nodes on their right (b). The output program graph (c) can then have recursive connections. Here, node weights are also used, modifying the final program (d).

used to compare many controller algorithms, from deep Q learning to neuroevolution. This environment of Atari games offers a number of different tasks with a common interface, understandable reward metrics, and an exciting domain for study, while using relatively limited computation resources. The ALE offers a quantitative comparison between CGP and other methods. Atari game scores are directly compared to published results of multiple different methods, providing a perspective on CGP's capability in comparison to other methods in this domain.

One of the difficulties across the Atari domain is using pure pixel input. While the screen resolution is modest compared to modern game platforms, processing this visual information is a challenging task for artificial agents. Object representations and pixel reduction schemes have been used to condense this information into a more palatable form for evolutionary controllers. Deep neural network controllers have excelled here, benefiting from convolutional layers and a history of application in computer vision.

CGP's history of use in image processing tasks makes it a suitable candidate for use on pure-pixel implementations. In [Wil+18b], a number of image processing functions were used in the CGP function set, as well as array manipulation functions. MT-CGP was used, as the controller input is an array, the pixel input, but the output is a number of scalar values corresponding to the possible actions.

CGP has unique advantages that make its application to the ALE interesting. By using a fixed-length genome, small programs can be evolved and later read for understanding. While the inner workings of a deep actor or evolved neural network might be hard to

discern, the programs CGP evolves can give insight into strategies for playing the Atari games. Finally, by using a diverse function set intended for matrix operations, CGP is able to perform comparably to humans on a number of games using pixel input with no prior game knowledge. Certain demonstrative results are presented here, but the full results can be found in ([Wil+18b]).

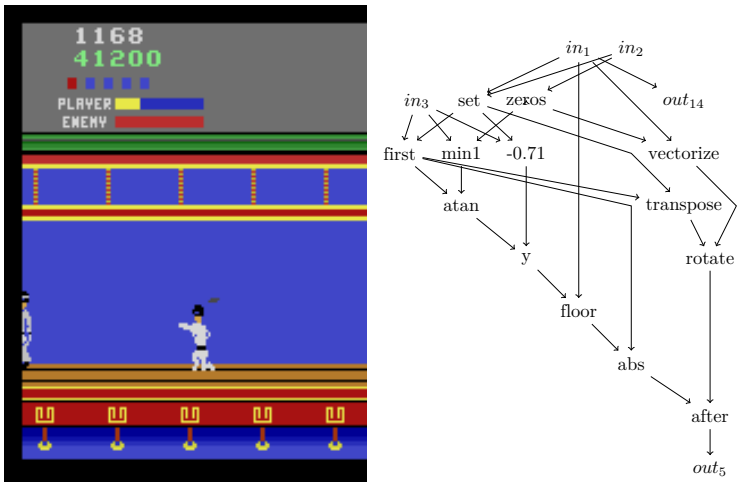

Figure 3.5: The Kung-Fu Master crouching approach and the functional graph of the player. Outputs which are never activated, and the computational graph leading to them, are omitted for clarity.

By inspecting the resultant functional graphs of an evolved CGP player and observing the node output values during its use, the strategy encoded by the program can be understood. For some of the best performing games for CGP, these strategies can remain incredibly simple. One example is Kung-Fu Master, shown in Figure 3.5. The strategy, which can receive a score of 57800, is to alternate between the crouching punch action (output 14), and a lateral movement (output 5). The input conditions leading to these actions can be determined through a study of the output program, but output 14 is selected in most cases based simply on the average pixel value of input 1.

While this strategy is difficult to replicate by hand, due to the use of lateral movement, interested readers are encouraged to try simply repeating the crouching punch action on the Stella Atari emulator. The lateral movement allows the Kung-Fu Master to sometimes dodge melee attacks, but the crouching punch is sufficient to wipe out the enemies and dodge half of the bullets. In fact, in comparison to the other attack options (low kick and high kick) it appears optimal due to the reduced exposure from crouching.

Other games follow a similar theme. Just as crouching is the safest position in Kung-Fu Master, the bottom left corner is safe from most enemies in Centipede. The graph of an individual from early in evolution, shown in Figure 3.6, demonstrates this. While this strategy alone receives a high score, it does not use any pixel input. Instead, output 17 is the only active output, and is therefore repeated continuously. This action, down-left-

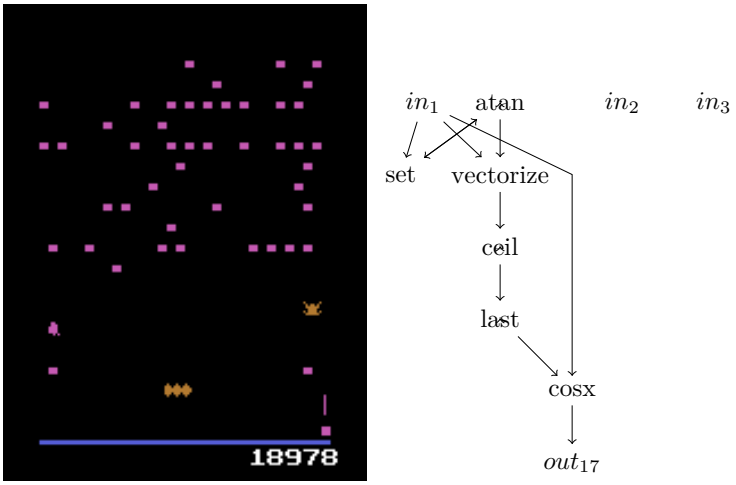

Figure 3.6: The Centipede player, which only activates output 17, down-left-and-fire. All other outputs are linked to null or constant zero inputs and are not shown.

and-fire, navigates the player to the bottom left corner and repeatedly fires on enemies. Further evolved individuals do use input to dodge incoming enemies, but most revert to this basic strategy once the enemy is avoided.

The common link between these simple strategies is that they are, on average, effective. Evolution rewards agents by selecting them based on their overall performance in the game, not based on any individual action. The policy which the agent represents will therefore tend towards actions which, on average, give very good rewards. As can be seen in the case of the Kung-Fu Master, which has different attack types, the best of these is chosen. Crouching punch will minimize damage to the player, maximizing the game's score and therefore the evolutionary fitness. The policy encoded by the program doesn't incorporate other actions because the average reward return for these actions is lower. The safe locations found in these games can also be seen as an average maximum over the entire game space; the players don't move into different positions because those positions represent a higher average risk and therefore a worse evolutionary fitness.

Not all CGP agents follow this pattern, however. A counter example is boxing, which pits the agent against an Atari AI in a boxing match. The CGP agent is successful at trapping the Atari player against the ropes, leading to a quick victory, as shown in Figure 3.7. Doing this requires a responsive program that reacts to the Atari AI sprite, moving and placing punches correctly to back it into a corner. While the corresponding program can be read as a CGP program, it is more complex and performs more input manipulation than the previous examples. Videos of these strategies are available online[3].

Finally, in results presented in [Wil+18b], CGP is compared to other state of the art methods. CGP performs better than all other compared artificial agents on 8 games,

[3]https://vimeo.com/d9w/

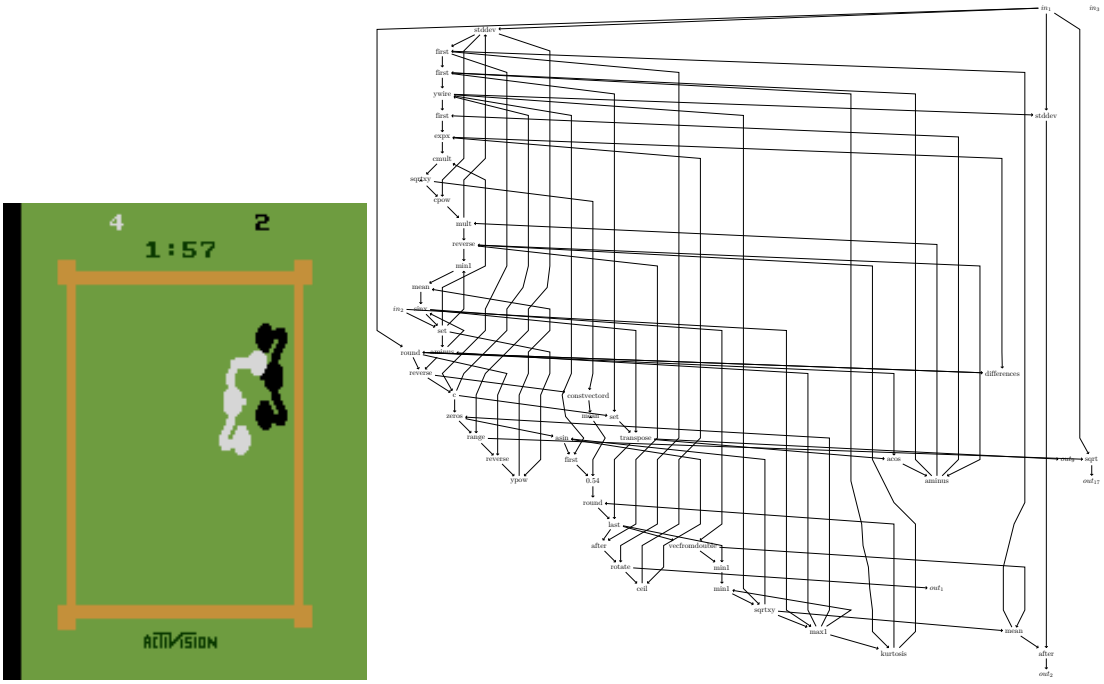

Figure 3.7: Boxing, a game that uses pixel input to continuously move and take different actions. Here, the CGP player has pinned the Atari player against the ropes by slowly advancing on it with a series of jabs (right). The boxing program (left), which is more complex as it uses image processing to determine the location of the enemy sprite an move to it. While this program is complex, it can still be parsed as a function graph.

and is tied for best with HyperNEAT for one game. On a number of games where CGP does not perform the best, it still achieves competitive scores to other methods. However, there are certain games where CGP does not perform well. There appears to be a degree of similarity between evolved agents in which games they perform well one. There is also a degree of similarity between the deep learning agents. We attribute this similarity to the creation of a policy model for deep learning agents, which is trained over a number of frames, as opposed to a player which is evaluated over an entire episode, as is the case for the evolutionary methods. This difference is discussed further in the next section.

Taking all of the scores achieved by CGP into account, the capability of CGP to evolve competitive Atari agents is clear. Not only is CGP able to find competitive strategies for many of the games, including games like Boxing where pixel input is necessary, the programs which inform those strategies are legible as graphs. This allows for understanding not only the programs, but also the fitness environments used in evolution. While the ALE is a widely used benchmark, many of the Atari games commonly tested have very simple, even constant, solutions. This was suspected in [Hau+14], which used HyperNEAT and CMA-ES on different input representations, including a noise input which did not contain the game information, revealing that the pixels are not necessary for playing

some games. However, with CGP we can clearly see from the evolved programs whether or not they rely on pixel values or use constant strategies.

3.2.3 Positional Cartesian Genetic Programming

Returning to the study of CGP, we use floating point CGP as a base to present Positional CGP (PCGP), which introduces a small modification which allows for many possibilities. Each node also has a position gene, p_n, which determines the position of the node, instead of spacing each node equally between 0.0 and 1.0. In CGP, a connection has equal probability of connecting to each node previous to its parent. In PCGP, this probability is evolved based on the positions of each node.

Evolving the node positions complicates the role of the input nodes, however. In SM-CGP, where it also isn't certain the graph will include input nodes, program input is a function which nodes can choose [HBM10]. In this work, we chose to place input nodes in an evolved space to the left of the node space, ensuring that nodes form connections to inputs while allowing the inputs to also form their own connection distributions through evolution. The input nodes each have a positional gene, i_n, which is multiplied by a hyper-parameter which determines the start of the input space, I_{start}. Node position calculation is then modified to contain the entire space, including the input space:

$$p_{x_n} = x_n((r(1.0 - p_n) + p_n) - I_{start}) + I_{start} \tag{3.20}$$

When $I_{start} = -1.0$, as in Figure 3.8, the input space is large and nodes have a high probability of connecting directly to input nodes. As this is not desirable for complex programs, the I_{start} parameter was tested in the following experiments.

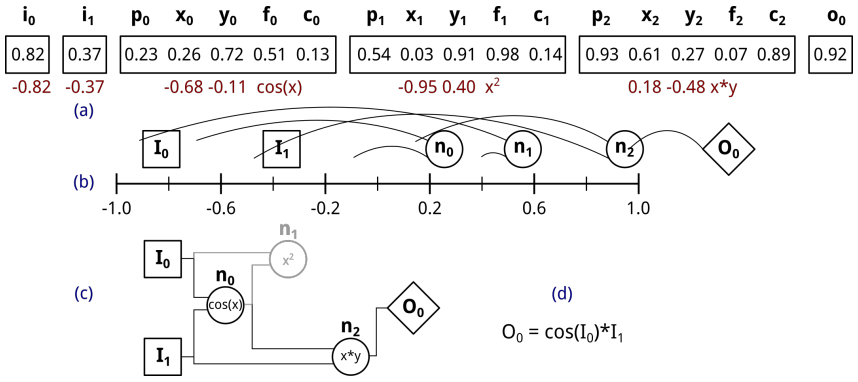

Figure 3.8: A PCGP genome (a), including input i_n and positional p_n genes. These are translated to input and node positions (b) and connection positions "snap" to the nearest node, as in floating point CGP. As in CGP, a resultant graph (c) and output program (d) are then extracted.

Due to the evolution of the positions, it is highly likely that no two nodes occupy the same position, even between different genomes. Furthermore, over evolution, nodes which

are connected can have positional genes and connection genes which are highly related. Finally, a node's connection positions depend only on that node's position, which is in its genes, and not on the node's placement in the genome or other nodes in the network. This allows node genes to be exportable; the same genes in a different individual will form connections in the same place. If multiple genes are exported together, entire sections of the graph can be migrated between individuals. In PCGP, nodes can be added or removed from a genome without disturbing the existing connection scheme, unlike in CGP, where a node addition and deletion causes a shift in all downstream node positions. This is the inspiration for the following study, where graph based operators from other forms of GP are used in PCGP.

3.2.4 Genetic operators for CGP

GP has numerous genetic operators defined across its many implementations. Genetic mutation and single point crossover have been used extensively, but tree-based GP also mutates and crosses specific parts of a genome. Autoconstructive evolution introduced many operators as part of a program modification instruction set ([Spe02]). Evolution of ANNs ([SM02]) and AGRNs ([CHP15]) provide examples of genetic operators especially suited for genomes that encode graphs, which is relevant to both CGP and PCGP. Parallel distributed GP (PDGP) was inspired by ANNs and included a subgraph addition mutation and a subgraph crossover method called subgraph active-active node (SAAN, [Pol+97]). A comparison of different crossover operators and ideal parameters for crossover are presented in [HK18]. Here we define multiple genetic operators for both CGP and PCGP drawn from other GP methods, ANN evolution, and GRN evolution. Operators are defined in **bold** and common methods used by multiple operators are defined in *italics*.

First, we define common methods used by multiple operators:

Node addition: $m_\delta size_{min}$ hidden nodes with random genes are added to the end of the genome. In PCGP, these are then sorted into the genome based on position.

Node deletion: $m_\delta size_{min}$ hidden nodes are randomly selected from the genome and removed. In the event that there are fewer than $m_\delta size_{min}$ hidden nodes, all hidden nodes are removed.

Subgraph addition: $m_\delta size_{min}$ hidden nodes are added to the genome. The position, function, and parameter genes are randomly chosen, but connection genes are selected randomly out from a pool. For each new node i, this pool composed of all other new nodes with position $p < p_i$. An equal number of randomly selected hidden and input nodes with $p < p_i$ are also added to the pool from the parent chromosome. By fixing the connection genes, the new genetic material is guaranteed to either contain new subgraphs or to create a subgraph with existing nodes. Due to the requirement of knowing exact

node positions for connection, this operation is available only in PCGP.

Subgraph deletion: The parent genome is evaluated into functional trees, both those which result in a final output (active) and those which don't (inactive or junk). A tree with more than 1 hidden node is chosen randomly and up to $m_{delta}sizemin$ hidden nodes are removed from it.

These methods are used in the following three mutation operators:

(1) Gene mutation: In CGP and PCGP, each gene of the hidden nodes has a m_{node} chance of being replace by a new random value in $[0.0, 1.0]$. Outputs are similarly mutated with a m_{output} chance. In PCGP only, the positions of the input nodes are mutated with a m_{input} chance. If m_{active} is true, this mutation is repeated until a gene in an active node is mutated ([GP13b]).

(2) Mixed node mutate: A random method between **gene mutation**, *node addition*, and *node deletion* is chosen according to m_{modify} and the size of the parent genome. A random number r is chosen in $[0.0, 1.0]$. If r is less than m_{modify}, **gene mutation** is selected. If r is less than $m_{modify} + m_{add}$, *node addition* is selected. Otherwise, *node deletion* is selected. m_{add} is calculated based on the number of nodes in the parent genome, n:

$$m_{add} = \frac{(n - size_{min})(1 - m_{modify})}{size_{max} - size_{min}}$$

(3) Mixed subgraph mutate: Following the same logic as **mixed node mutate**, a method between **gene mutation**, *subgraph addition*, and *subgraph deletion* is chosen according to m_{modify} and the size of the parent genome.

For CGP, three crossover methods have been defined: **single point**, **random node**, and **proportional**. As PCGP allows for program structure to be preserved during genetic transfer and contains additional genetic material in the form of positions, further crossover methods can be defined: **aligned node**, **output graph**, and **subgraph**.

(1) Single point crossover: In this classic crossover operator, a single point in the two parent genomes is selected randomly. The genetic material before this point is taken randomly from one parent and the genetic material after this points is taken from the other parent. In CGP and PCGP, the point is constrained to the beginning of a node's genetic material.

(2) Random node crossover: Nodes are randomly selected equally from both parents. A child is constructed using randomly selected input and output genes from both parents, the selected hidden node genes from the first parent, then finally the selected hidden node genes from the second parent. The ordering of the genetic material is important for CGP, but in PCGP the nodes and their corresponding genes are ordered by their position.

(3) Aligned node crossover: This operator is only applicable for PCGP. Nodes are first paired from each parent based on position proximity, This operator then follows the same method as **random node crossover**, however nodes are randomly chosen from their position aligned pairs.

(4) Proportional crossover: This operator was previously explored in [CWM07]. The child's genetic material C, up to the minimum size of both parents (A and B), is combined using a vector of randomly chosen weights, w:

$$C_i = (1 - w_i)A_i + w_i B_i \quad \forall i \tag{3.21}$$

If one parent genome was longer than the other, the remaining genetic material is appended to the end of the child genome.

(5) Output graph crossover: Outputs from each parent are randomly selected for the child genome. For each selected output, the full functional graph resulting in this output is computed, and the set of all hidden nodes in the selected output graphs for each parent is used to construct the child genome. Functional arity is ignored in this output trace, meaning that inactive genetic material from 1-arity functions will be passed on to the child genome. Otherwise, this operator only takes active nodes from each parent. If an input node is used in only one parent's selected output graph, it is passed on to the child directly. Otherwise, each input node is randomly selected from both parents. As this operator assumes that the functional graph directly corresponds to the transferred genetic material, it is only available in PCGP.

(6) Subgraph crossover: Similarly to **output graph crossover**, the functional graphs from the parent genomes are computed. However, in this operator, active and inactive subgraphs from both parents are randomly selected equally. Input and output genes are selected randomly from both parents. As with **output graph crossover**, this operator is only applicable to PCGP.

3.2.5 CGP experiments

To explore the utility of these different genetic operators in CGP and PCGP, a parameter study is done using irace [Lóp+16]. irace is an automatic algorithm configuration package which selects from ranges of parameters and explores the parameter space efficiently by focusing on high performing parameter sets in a method known as racing.

The different genetic operators are parameterized and included with all CGP and PCGP parameters for irace optimization. A $1 + \lambda$ EA and a GA are used, and the necessary parameters for the two are also optimized. The GA includes the parameters $GA_{elitism}$, determining the number of top individuals retained each generation, $GA_{crossover}$, the percentage of new individuals produced by crossover, and $GA_{mutation}$, the percentage

Parameter	type	range
mutation	c	genetic, mixed node, mixed subgraph
crossover	c	single point, proportional, random node, aligned node, output graph, subgraph
λ (population)	i	[1, 10]
GA population	c	20, 40, 60, 80, 100, 120, 140, 160, 200
I_{start}	r	[-1.0, -0.1]
r	r	[0.0, 1.0]
w	c	true, false
m_{active}	c	true, false
m_{input}	r	[0.0, 1.0]
m_{output}	r	[0.1, 1.0]
m_{node}	r	[0.1, 1.0]
m_{δ}	r	[0.1, 0.5]
m_{modify}	r	[0.1, 0.9]
$GA_{elitism}$	r	[0.0, 0.8]
$GA_{crossover}$	r	[0.1, 1.0]
$GA_{mutation}$	r	[0.1, 1.0]

Table 3.5: Ranges used in irace. The different range types are choice (c), integer (i), and real-valued (r). The precision for real-valued parameters was 0.1.

of individuals produced by mutation. If these three sum to less than 1.0, random tournament winners are added to the population unmodified. CGP and PCGP are evaluated separately, and each is evaluated on an EA and GA separately, creating four different parameter optimization cases.

These four cases are evaluated using nine benchmarks: three classification problems, three regression problems, and three reinforcement learning or control problems. In this chapter, we focus primarily on the reinforcement learning controller problems; full results from the other experiments can be seen in [Wil+18c]. The reinforcement learning tasks are three locomotion tasks from the PyBullet library [CB16]. In these tasks, a robotic ant, cheetah, and humanoid must be controlled to walk as far as possible from the starting point.

3.2.6 CGP method comparison

First, we compare the four different methods, $1 + \lambda$ EA and a GA, using both CGP and PCGP, using the best parameters found by irace for the set of three problems of each type. The optimized methods, e_0 the CGP $1 + \lambda$ EA, e_1 the PCPG $1 + \lambda$ EA, e_2 the CGP

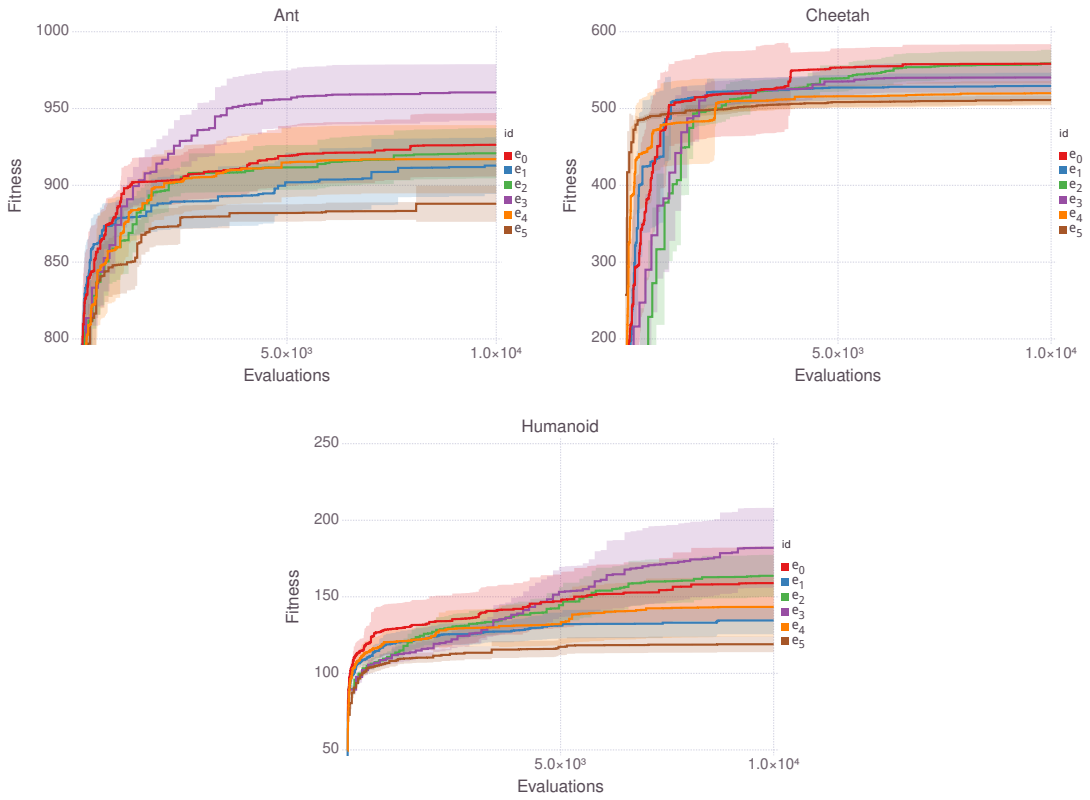

Figure 3.9: Comparison of the different methods on the reinforcement learning controller problems. Lines show the average of the best individuals and ribbons show one standard deviation from 20 trials.

GA, and e_3 the PCGP GA, are compared to the default parameters of CGP, e_4, and the parameters reported in [CWM07], e_5, as it is a similar work which uses crossover and a floating point representation. The crossover rate was chosen based on the results from [CWM07], although that work includes an interesting study of a variable crossover rate, which was not implemented for these experiments.

These parameters were used in 20 evolutions on each of the nine problems. To compare the different population sizes and methods, the results are compared based on the number of fitness evaluations, not by generation. For the classification and regression problems, each evolution was run for 20000 evaluations and 10000 for the RL problems, as these problems were far more computationally expensive.

Overall, the results, displayed in Figure 3.9, show that the optimized parameter sets perform better than the e_5 default GA parameter set. However, the standard $1 + \lambda$ CGP EA, e_4, does well on these problems and outperforms the $1 + \lambda$ PCGP EA expert, e_2. e_3, the PCGP GA, performs the best on the controller problems. However, the full results from [Wil+18c] demonstrate that there is no generally best method. Specifically, e_3 does not perform well on classification and regression tasks.

The main conclusion that can be drawn from this comparison is that the choice of

method, and the parameters of the chosen method, can greatly improve CGP performance. To better understand appropriate parameters, therefore, we next analyze the full results from irace, beyond the best individual used in these comparisons. As these results vary between classification, regression, and reinforcement learning, we include all parameter results from [Wil+18c], even those that do not concern controller evolution explicitly, to demonstrate the variety of CGP parameter choices.

3.2.7 CGP parameter study

Finally, we explore the parameter choices produced by irace. The top 20 parameters, called expert parameter sets, for each method are displayed for the different class types in Figure 3.11 and Figure 3.12, and the correlation between each parameter and evolutionary fitness is shown in Figure 3.10. In Figure 3.11 and Figure 3.12, all parameters are represented as values between 0.0 and 1.0. To achieve this, the mutation operators were ordered as [genetic, mixed node, mixed subtree], being [0.0, 0.5, 1.0], and the crossover operators were ordered [single point, proportional, random node, aligned node, output graph, subgraph], being [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]. The population parameter represents $\frac{\lambda}{10}$ for the $1+\lambda$ EA and $\frac{population}{200}$ for the GA. Finally, I_{start} is represented as $1+I_{start}$.

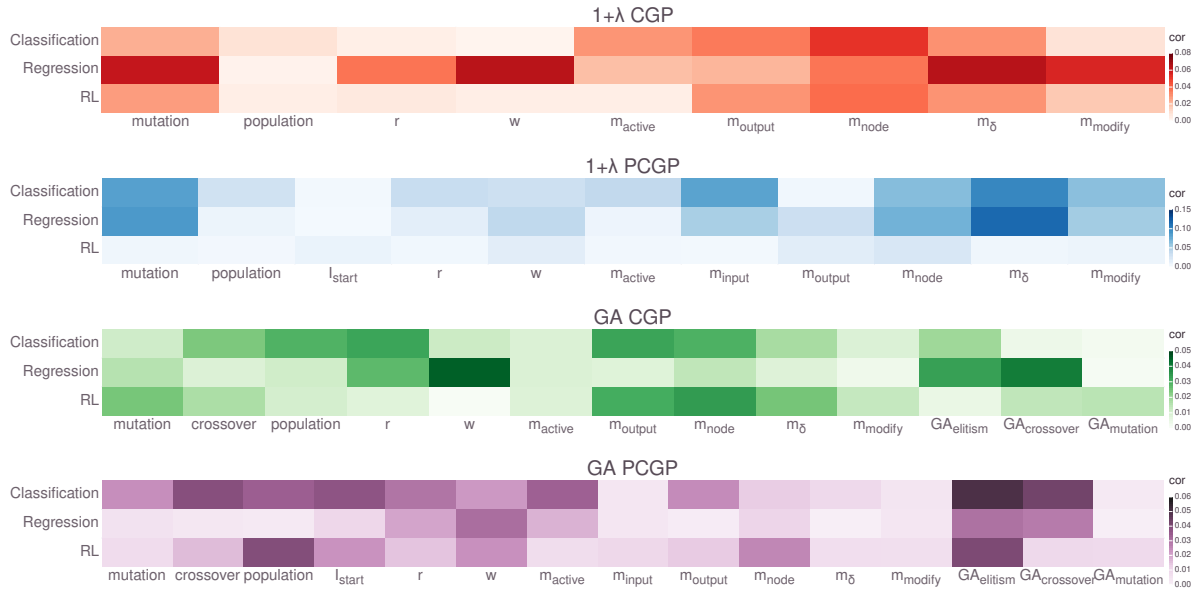

Figure 3.10: Correlation of all parameters with evolutionary fitness, using all parameter sets explored by irace.

For CGP with the $1+\lambda$ EA, there is a clear preference for gene mutation over mixed node mutation across all problems. Mutation rates are important, and most expert parameters have a slightly higher output node mutation rate than node mutation rate, with m_{output} reaching as high as 0.5 in many experts. Output node weights, w, especially have

an impact for regression, where they should not be used. Active mutation, m_{active}, appears useful mostly for classification and regression and isn't correlated with fitness for RL tasks. Finally, λ appears to have little effect on the outcome, and r appears to impact the result only in the case of regression tasks, when it should be low.

Figure 3.11: Parameters for the top 20 $1 + \lambda$ EA configurations, for CGP and PCGP

The expert parameter results for PCGP with the $1 + \lambda$ EA are similar to that with CGP. Gene mutation is a clear winner, using a higher mutation rate for output nodes than input and hidden nodes. λ, I_{start}, and r aren't strongly correlated with fitness. Node weights appear useful in RL problems but are detrimental in classification and regression problems, and active mutation appears generally useful. An interesting difference in $1+\lambda$ EA PCGP is the prevalence of mixed node mutation in expert parameter sets for RL problems. However, due to a high m_{modify} of these parameter sets, the main functional mutation in these expert sets remained a gene modification mutation, with rare node

addition and deletion events.

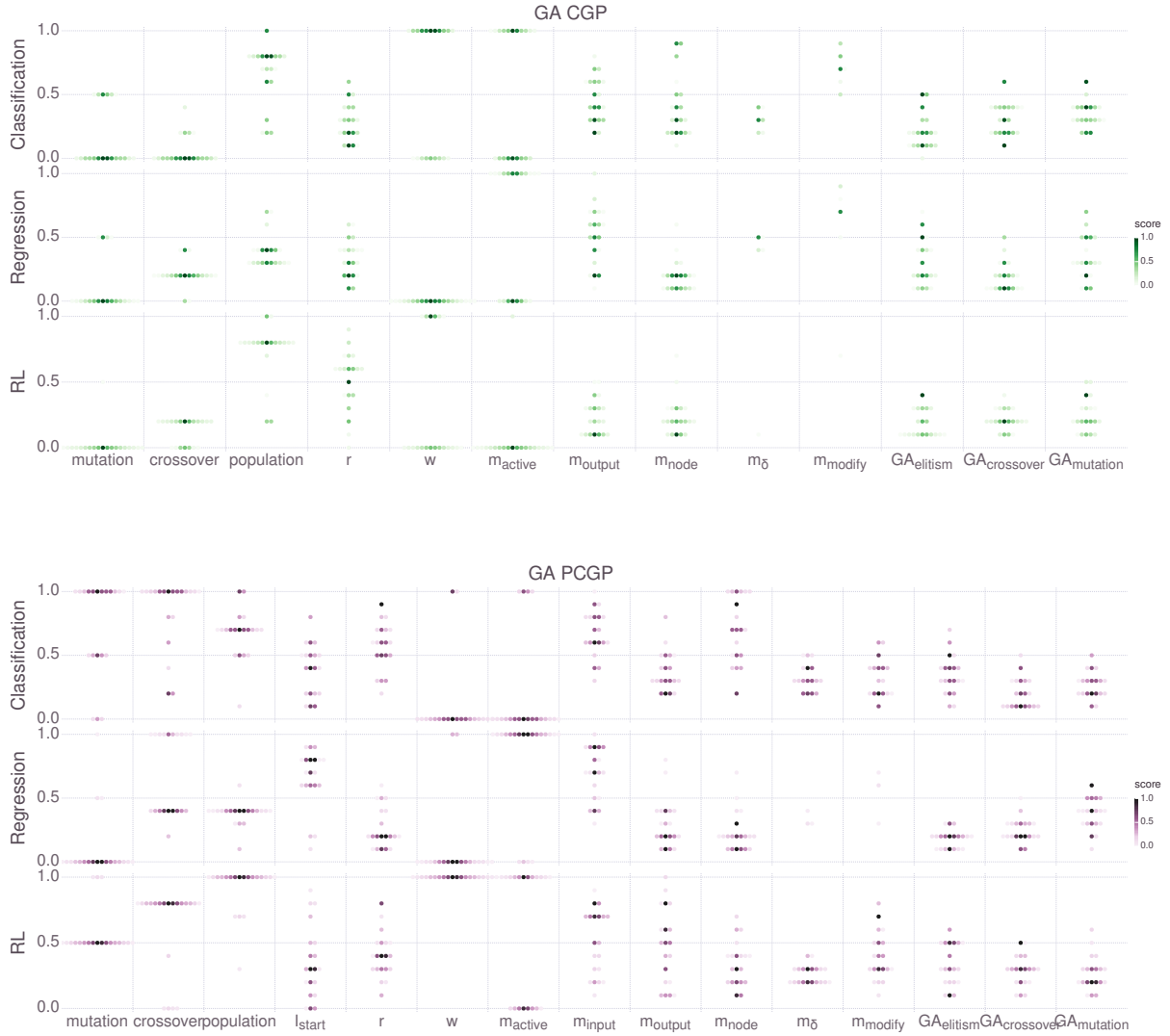

Figure 3.12: Parameters for the top 20 GA configurations, for CGP and PCGP.

For CGP, the expert parameters for a GA are very similar to the $1+\lambda$ CGP EA expert parameters. Population is much more important than in the $1+\lambda$ EA, with medium to large populations (100 to 200 individuals) showing an advantage in classification and regression problems. Genetic mutation is the clear choice for a mutation operator, while crossover is split between single point for classification and proportional for regression and RL. Elitism is rather high, reaching 50% in some expert sets. Crossover rates are low, except in the case of classification, where it has little bearing on the final outcome.

The parameter results for PCGP when using a GA are very interesting and different from all other sets. Here, we see usage of the other mutation and crossover operators; classification prefers mixed subtree mutation and subgraph crossover, regression uses gene mutation and random node crossover, and RL uses mixed node mutation with output

graph crossover. The population is somewhat problem dependent, but is especially important in RL, where large populations are favorable. Node weights are highly preferred in RL, but not in regression or classification. Elitism has a large impact on the final fitness, although the values for RL and classification are spread almost evenly between 0.1 and 0.5.

Considering the success of e_3 on the RL problems, the PCGP GA parameters show that output graph crossover and node-based mutation can be viable strategies for PCGP evolution. It's notable also that subgraph crossover was used in expert regression sets and favored in classification, showing that graph-based operations can be useful generally. The RL problems have outputs corresponding to the control of different limbs, which may offer more modularity than the different classes of classification.

Positional CGP opens the possibility of doing graph operations during CGP evolution. The experiments in this work demonstrate that there is the potential for improvement of CGP's evolution, even if no single method proposed is universally dominant. The possibilities in improving CGP evolution are expanded by PCGP, and more work is needed to explore these potential improvements.

Some of the parameters explored in this work are at the level of evolution and require global coordination. Others, such as r, I_{start}, w, etc, could be included at the level of the genome. Even the choice of CGP or PCGP could be a binary parameter within the genome, deciding if the positional genes are used or not. This would allow an individual optimization of the hyper-parameters and reduce the burden of parameter choice.

The global parameters could also benefit from dynamic change over evolution. In [CWM07], a variable crossover rate which begins high and reduces to 0 as the population converges is used. Adaptive mutation rates have also been proven to increase search for the $1 + (\lambda, \lambda)$ EA ([DD18]) and could benefit CGP.

3.3 Conclusion

As demonstrated in the AGRN and CGP studies, both methods are capable of evolving effective controllers for a variety of problems. The results from these studies demonstrating optimal parameter and implementation choices influenced all of the following work concerning using AGRNs and CGP as controllers of neural components. The two methods have different benefits and the choice of method was largely based on the experiment at hand. AGRNs offer a biological analogy which is lacking in CGP, where biological GRN interactions are well studied and can be used in the design of AGRN controllers. On the other hand, CGP offers interpretable results; an AGRN's dynamics can be observed, but they can't be read with the same clarity as a program graph generated by CGP.

In the next chapter, CGP is used to evolve the activation function of spiking neural

networks. We use an existing activation function, the leaky integrate and fire model, as a starting point for evolution, which is only possible with CGP. In chapter 5, axon branching is explored using AGRNs in an experiment largely based on biology, where gene knockout was a critical component of the biological experiments, making AGRNs the preferred controller. AGRNs are also used in chapter 6 as neuromodulatory agents controlling learning, although, in this case, CGP could have also been used. Finally, in chapter 7, CGP is used in order to evolve functions which are understandable inside a model with multiple interacting evolved functions. The two methods, AGRNs and CGP, are both effective means of evolving controllers, but each has its specific benefits.

Chapter 4

Evolving neural cell function

The first aspect of ANNs we investigate is the cell function. In neural networks, each neuron is assigned a function which describes its activity, called a transfer function or activation function. Across the different levels of abstraction in ANNs, from modeling actual neurons in computational neuroscience to applied deep learning tasks, very different activation functions are used. In perceptron-based neural networks, the activation function describes the average activity of the cell. Study into the function to use for these networks has often been based on statistical information about activity rates of the cell, looking at spiking events over time, or on improving the efficiency of the network in artificial tasks. On the other end of abstraction, in biological simulations of neurons, the activation function simulates physical components of the cell: the membrane potential, sodium channel activation, and more. For the models closest to biology, their design and parameters are based directly on cell physiology from the brain.

At the most abstract level, the activation function uses an average synaptic activation input, summed over all input synpases, to determine the average activation of the cell. This is the case for most ANNs, including those in deep learning. The logistic function has historically been a popular choice for this function, as the derivative of this function is easy to calculate, simplifying gradient descent calculations. Neurons that use a logistic function have been called "sigmoid neurons" and are still commonly used. Recently, deep learning has shown a preference for the rectified linear function, defined as $y = max(0, x)$. This function has been shown to be more effective for learning tasks than the sigmoid function, while still having a simple derivative [Hah+00].

Spiking neural models use more complicated activation functions, as the function is responsible for not only integrating synaptic inputs, but also determining the spike event timing. The simplest of these models is the integrate and fire (IF) model, where synaptic inputs are added to the membrane potential, V, until V surpasses a fixed threshold, at which point the neuron spikes and the potential is reset. More complicated models include

phasic activity, bursting, adaptive membrane thresholds, and more. These models will be covered in the next section.

In general, SNN activation functions have been designed to approximate biology. As SNNs are used in artificial tasks, however, modifications are made to the base models to improve the performance of the network. This can be a difficult engineering process, as neural activation depends on the learning algorithm and neural architecture, and the way in which to tune neural activation towards a certain goal can be obfuscated by these other components.

In this chapter, we use evolution to find an neural activation function suited for an artificial learning task. CGP is used to evolve a function which alters the state of each neuron and decides when the neuron fires. This evolved activation function is used as the network performs clustering on standard datasets, with STDP tuning the synaptic weights. Using a leaky integrate and fire (LIF) function as a base and standard of comparison, we show that CGP can improve on existing neural activation functions or generate new ones which are competitive. In the next section, we present an overview of existing spiking activation functions.

4.1 Spiking neural activation functions

In the brain, neurons exhibit a variety of different spiking behaviors, an overview of which is given in Figure 4.1. Some neurons have individual spikes with a constant phase, while others spike in bursts of multiple consecutive spikes. These differences arise from neural placement in the brain, for example retinal neurons have a different activity pattern than those of the thalamus, or from different species.

One of the goals of the design of neural activation functions is the ability to simulate these different types of behaviors with a change of parameters. The Izhikevich model [Izh03] is one such model, where four model parameters can change the activity type, allowing one model to exhibit all of the known behaviors of biological neurons. The [RH89] model also has a similar flexibility, as this model separates the activation function into multiple independent functions which can be chosen to match a specific neuron type.

Another goal of spiking neural function design has been the approximation of a specific biological cell type. The Morris-Lecar model of the giant muscle fiber of barnacles has been used for neural simulations, with biophysically meaningful parameters based on barnacle experiments [ML81]. The Hindmarsh-Rose model is an abstraction of thalamic neurons, where the definition of different neural functions can be chosen to approximate different neural behavior; in the presentation of the model, these functions were defined for thalamic neurons [RH89]. The Hodgkin-Huxley model is one of the most well-known biological models of neurons, as it simulates not only the membrane potential and spiking

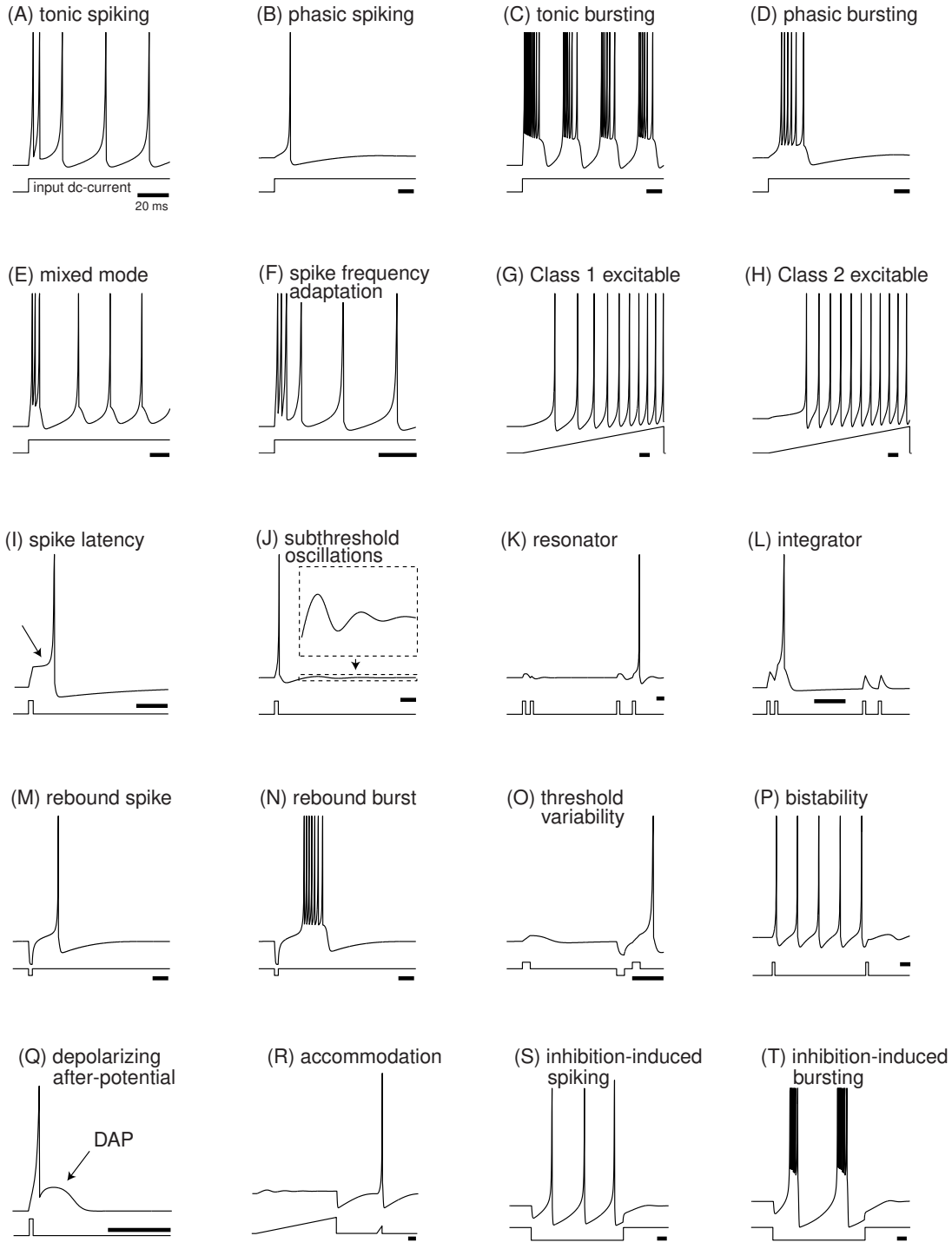

Figure 4.1: Different spiking behavior exhibited by biological neurons, simulated using the Izhikevich model [Izh03]. Each figure displays the neuron behavior on top, the input current on bottom, and a black horizontal bar to denote a 20-ms time interval. Electronic version of the figure and reproduction permissions are freely available at www.izhikevich.org

events, but also the sodium and potassium currents and the inactivation of the sodium current. [HH52]. The parameters of this model also have biophysical significance, which more abstract models lack.

For the use of SNNs in artificial tasks, the choice of model is usually based on the computational cost of the activation function. This function is calculated in each neuron, which can number in the hundreds even for small tasks, at each timestep in simulations which can last thousands of timesteps; it is important that the function is efficient. The integrate and fire model is a popular choice for that reason, with the leaky IF introducing enough complexity for STDP and requiring only 5 floating point operations (FLOPs). The quadratic IF model (also called the theta neuron or Ermentrout-Kopell model) brings the integrate and fire calculation closer to biological reality while only increasing computation to 7 FLOPs. The IF-or-Burst model, designed to model thalamo-cortical neurons, enables bursting activity but requires 13 FLOPs [Smi+00]. Biologically accurate models can be very expensive, with the Hodgkin-Huxley model requiring 1200 FLOPs. A comparison of different methods based on their computational cost versus biological reality is presented in Figure 4.2.

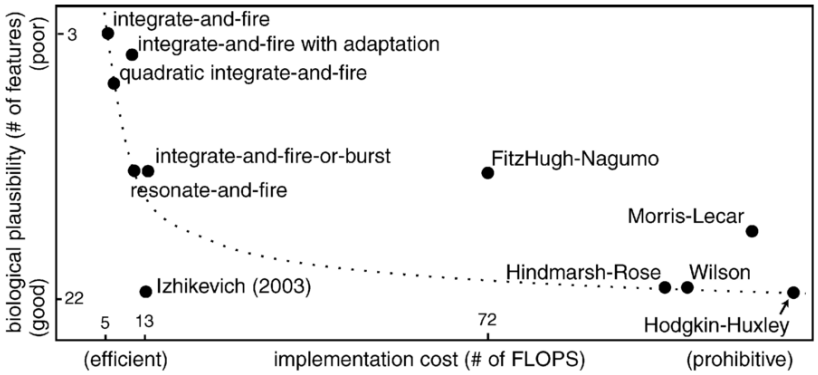

Figure 4.2: Biological abstraction of different neural models versus their computational cost [Izh04]

We've shown three main goals for SNN activation function design: flexibility to different behavior types, proximity to biological neurons, and computational cost. In this work, we focus on a new goal: improved learning on artificial tasks. So far, there have been no SNN models designed explicitly for this purpose. Applied SNNs often use modified versions of existing models which have been tweaked to increase fitness. In [Moz+18a], an IF model is used, but neurons are prohibited from spiking more than once per input image during an image classification task. In [BDK13], the Izhikevich model is used with different neuron types (different model parameters) for the excitatory neurons and inhibitory neurons.

Many of the choices in this chapter were based on [DC15], such as the network architecture and learning rule. In that work, an two-layer architecture was used to train a

spiking neural network on the MNIST dataset; this is shown in Figure 4.3. The excitatory weights were trained using STDP and the different classification labels were assigned to the excitatory neurons based on their firing pattern during training. In this work, they also used an LIF model modified to improve accuracy. Specifically, the LIF model is modified by the addition of an adaptive membrane threshold, which increases the spiking threshold after each spike, and a refactory period, where new spikes are not allowed for a certain time after each spike. These design choices constitute a difficult engineering process which could be done automatically, i.e. through evolution.

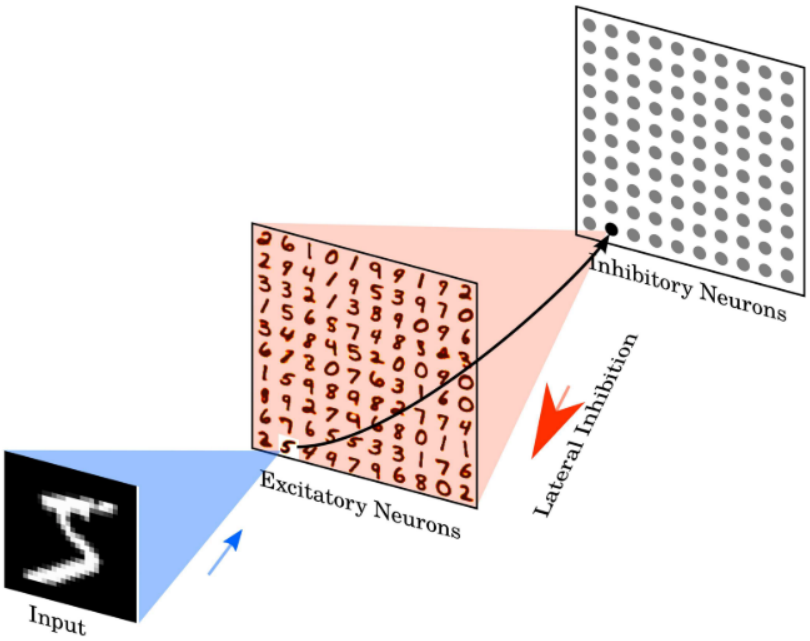

Figure 4.3: The two-layer architecture used in [DC15] for training on the MNIST dataset. The architecture used in this chapter is based on this work, as are the base neural activation function and learning method.

In the next section, we present our neuron model, which uses CGP to create the activation function. The goal of the model is purely the performance on artificial tasks; it is not based on any biological neuron, nor is it required to be computationally efficient. Indeed, some CGP individuals create activation functions much more costly than the popular LIF or quadratic IF models. We therefore evaluate this model on small tasks due to the high computational cost.

4.2 Neural network model

The IF model simulates only the membrane potential, increasing this state of the neuron until the threshold and resetting it. The Izhikevich model has a second state variable,

u, which modifies the membrane potential update and is itself reset after a spike. The adaptive membrane threshold used in [DC15] is introduced by a state variable, θ, which increases the threshold after every spike and decreases otherwise. All of these models share the principle of enlarging the state space of the neuron from just the membrane potential to multiple different state variables.

In this model, we choose a state size of 4 variables. These state variables, $s_{n,i}$, have no semantic meaning, however the first two are initialized in order to be compatible with the [DC15] adaptive LIF model. The first variable is set to the initial membrane potential -65mV, represented as -0.65, and the second to 1.0, which is used for the adaptive membrane threshold. However, evolution can use these state variables in a different way than the initial meaning.

The evolved neural functions have five inputs and five outputs. The first four inputs are the neural state variables, and the fifth input is the synaptic input to the neuron. The first four outputs represent the change to each neural state value at each update: $out_i = \Delta s_{n,i}$. The fifth output indicates spiking activity; if the fifth output is positive, the neuron is considered to have spiked, regardless of the neural state values. In this way, the function is entirely responsible for the spiking decision, as this decision is complex and not simply based on passing a membrane potential threshold in existing models.

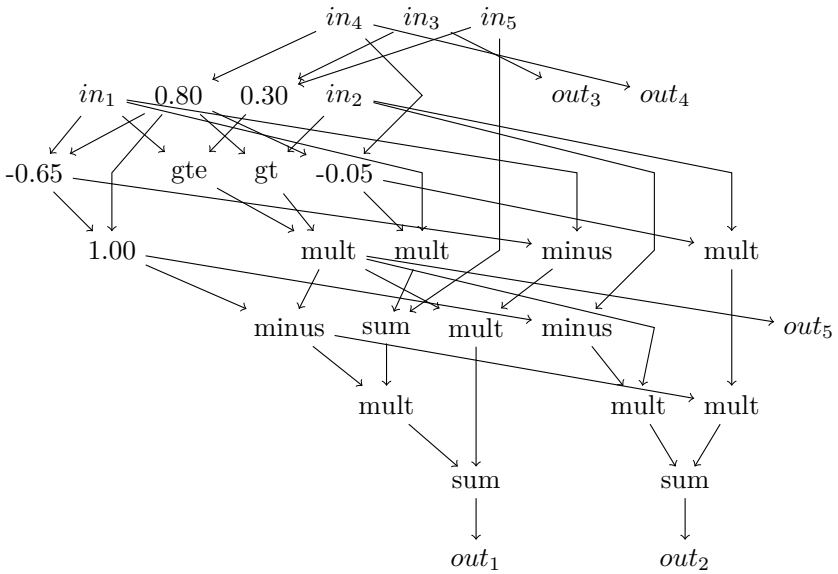

Figure 4.4: The adaptive threshold LIF model, represented as a graph. Random CGP chromosomes are generated for initializing evolution with this graph as the active nodes. In this graph, each node has a function, represented by its label, and two inputs, represented as incoming arrows. The 5 inputs and outputs are represented as in_i and out_i, respectively.

As CGP individuals represent functions, we can incorporate existing functions into the evolutionary search. For this, we use the adaptive membrane threshold LIF model

from [DC15]. This also provides a baseline and example of a working function. The LIF model graph is presented in Figure 4.4. The third and fourth neural state values are present in the graph, but they do not factor into the computation of out_1, out_2, or out_5, which represent the membrane potential, homeostasis variable, and spiking activity in the LIF model. New CGP chromosomes can be created from this expert "seed", as the graph only displays the active part of the chromosome. Inactive genetic material can be added randomly, resulting in different genetic offspring. The use of this seed in evolution is detailed in the next section.

4.3 Experiment

During evolution, each CGP individual is evaluated as a neural activation function. Neurons are configured in a network all with the same initial state, and are trained using STDP on a data clustering task. We then investigate the quality of the evolved models on different random initial weights and on different problems.

4.3.1 Clustering tasks

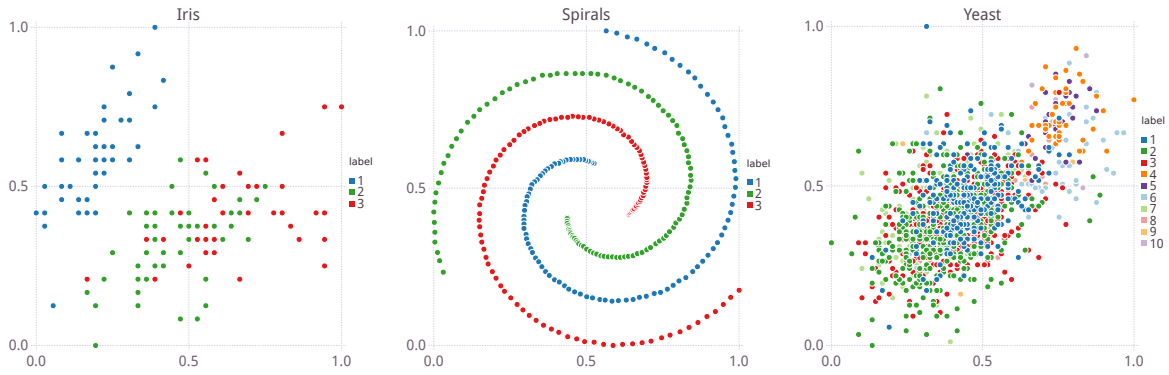

Figure 4.5: The data sets used for clustering in this work. All data were normalized in $[0.0, 1.0]$ to determine an input Poisson spike train. Axes shown were chosen for display purposes only.

To assess the learning ability of networks with different neural functions based on Hebbian learning, clustering was used. Clustering is a classic task in unsupervised learning; there are many existing methods against which to compare and standard benchmark data. In this work, we use the three data sets presented in Figure 4.5. These problems were chosen based on the different challenges they present; the spirals problem is difficult for spatial methods, and the yeast data is more numerous and higher dimensional. The iris problem is a standard benchmark and was used during evolution due to its small size

and dimensionality, while still providing a challenging clustering task. State of the art clustering for these three problems is presented in Table 4.1.

Method	Iris ARI	Spirals ARI	Yeast ARI
Kmeans	0.7163	-0.0054	0.1608
Kmediods	0.7445	-0.0043	0.1024
DBSCAN	0.6171	1.0	0.0917

Table 4.1: ARI from state of the art clustering methods. For DBSCAN, parameters were sampled and selected to have the best accuracy over all three problems.

Labels from the clustering network are compared to the original labels using the adjusted Rand index (ARI). This measure indicates the similarity between the two clusterings and is referred to here as the accuracy. The order of the output neurons does not affect the ARI, only the clustering of data. This accuracy metric is used to determine the fitness of the clustering network and its spiking activity neural function.

All data are normalized over each feature to $[0.0, 1.0]$ and input into the clustering network as Poisson spike trains with firing rates between from 0 to 55 Hz, proportional to the normalized data. Each input is presented for 350ms, and then a rest period of 100ms of no input follows each data point.

4.3.2 Network

For each problem, a SNN with a single hidden layer was used for clustering. The number of inputs is the number of features, n_{in}, and the size of the hidden layer is set to twice this amount, $2n_{in}$. The output layer size, n_{out}, is the expected number of clusters. Popular clustering methods, such as kmeans, also require providing the expected number of clusters.

As in [DC15], lateral inhibition is used to create competition between neurons of the same layer. Each neuron has an inhibitory connection to all other neurons in the same layer. The weight of this connection is -0.1 and is not learned. Therefore, when a neuron fires, it sends a constant inhibitory signal to all other neurons, suppressing their firing activity. The layout of the network is presented in Figure 4.6.

4.3.3 Training

STDP is used to train the weights of the excitatory connections at each layer. The excitatory connections of the network are initialized with random weights following a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with $\mu = 0.2, \sigma = 0.3$. Excitatory weights are constrained

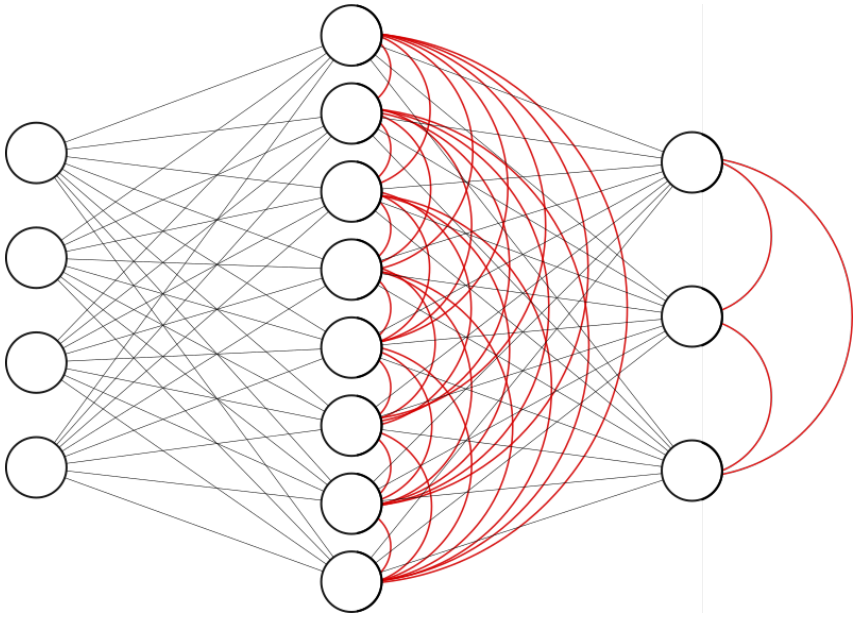

Figure 4.6: Network architecture. Red lines indicate inhibitory links, which are not learned.

to $[0.0, 1.0]$, and inhibitory weights are fixed to -0.1. The STDP rule from [DC15] is used:

$$\delta w = \eta(x_{pre} - x_{tar})(w_{max} - w)^{\mu} \tag{4.1}$$

where x_{pre} is a presynaptic trace which increases at every presynaptic spike and otherwise decreases exponentially and x_{tar} is the target value for the presynaptic trace at the moment of a postsynaptic spike. η is the learning rate and μ is a parameter that determines the dependence of the weight update on the previous weight. The update is applied on postsynaptic spikes, making this rule rather inexpensive.

The weights are trained during two passes through the data, called epochs, after which they are fixed. A final pass through the data is then performed to compute the expected label. The label is determined by counting the number of spikes in each neuron in the output layer during the 350ms window during which the corresponding input is presented; the output neuron which spiked the most is used as the label.

4.3.4 Evolution

Two different methods are used to initialize the evolutionary search for a spiking neural function:

- **evo0**: λ random CGP programs are generated and the best among them is selected as the first expert. This is the normal initialization for a $1 + \lambda$ EA.
- **evo1**: A graph matching the LIF model is used to generate the first expert. λ individuals are then generated by mutating this individual, and then evolution proceeds

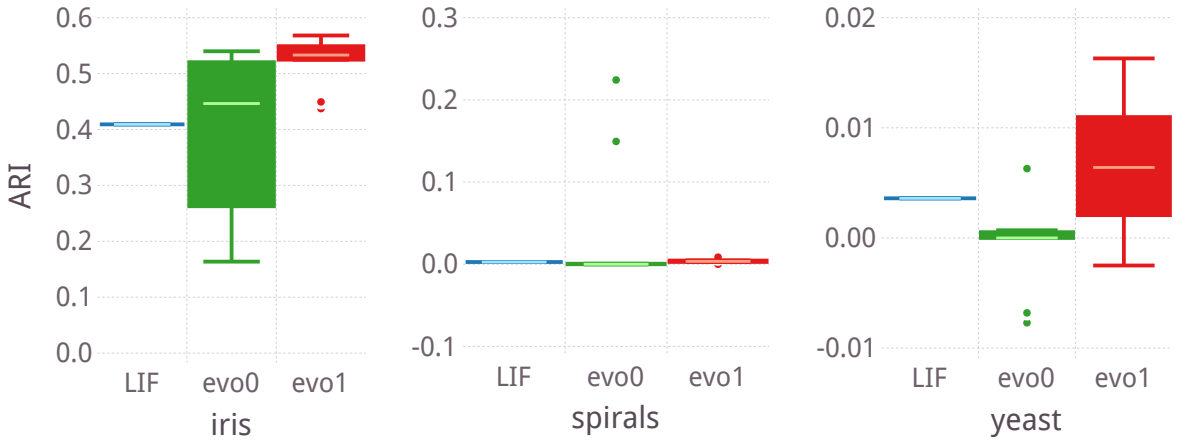

Figure 4.7: Generalization over different problems. The distributions for evolved models represent the best individuals from each evolution, using the same initial weights as in evolution.

as normal.

The evolutionary fitness for both methods is the ARI on the iris data set. The other data sets are used to determine generalization ability of the evolved models. 10 instances of both evolution types were run for 1000 generations.

After evolution, we test the evolved models for two different types of generalization. First, we examine generalization over different problems, testing the individuals evolved on the iris dataset using the other two datasets. Then we test for generalization to initial weights on the three datasets, as STDP can be very sensitive to initial weights.

4.4 Results

First we compare the evolved models using the same initial weights as in evolution. In Figure 4.7, we can see that the distribution of the best individuals is different between the two evolution types on the iris problem. As **evo1** was initialized with the LIF function, it is guaranteed to do at least as well as LIF on this problem with the same seed. However, despite starting from random programs, **evo0** is able to find models that compete with the LIF model.

Figure 4.7 also displays the generalization ability of the models on different problems. While some individual models from **evo0** generalized well on the spirals problem, most models did not. **evo1** models were mostly able to generalize to the yeast problem, performing better than LIF.

Next, we compare generalization over different initial conditions. While the evolved functions were selected only on one random seed, some models display greater general-

ization than the LIF model. The best individual from each evolution was tested on 20 different random seeds; these results are presented in Figure 4.8. For the iris problem, many of the best models have a higher mean accuracy than the LIF model, and they are able to show competitive generalization to the other problems, similar to the LIF model.

In Table 4.2, the best results from the two evolutionary methods and LIF over all random seeds are shown ("max"), as well as the best average accuracy ("mean") from the different evolutionary methods. While both **evo0** and **evo1** were able to improve upon LIF's best accuracy, **evo0** was able to generalize better to the spirals problem and to different random seeds. So, while initializing the evolutionary search with the LIF model guarantees an increased accuracy when using the same random seed as evolution, a random initialization may lead to models which generalize better.

Model	Iris ARI	Spirals ARI	Yeast ARI
LIF max	0.454	0.0196	0.0377
evo0 max	0.5438	0.2242	0.0468
evo1 max	0.5681	0.0173	0.0621
LIF mean	0.175 (± 0.196)	0.004 (± 0.006)	0.006 (± 0.013)
evo0 mean	0.267 (± 0.224)	0.012 (± 0.05)	0.007 (± 0.01)
evo1 mean	0.224 (± 0.205)	0.005 (± 0.006)	0.012 (± 0.02)

Table 4.2: ARI from the best model for each problem. "max" indicates the maximum accuracy obtained by a model on a single random initialization, and "mean" indicates the mean accuracy obtained by a model over 10 random initializations. Standard deviations are shown in parentheses.

Finally, the individual graphs representing the models can be examined for use as neural functions. In Figure 4.9, two evolved models are presented. The model evolved from LIF bears a close resemblance to it, using many of the same functions and constants as the LIF graph. Interestingly, both models make use of the other state values, $s_{n,3}$ and $s_{n,4}$, in the determination of firing activity.

4.5 Conclusion

In this chapter, we've shown that evolution can automatically find the activation function of spiking neurons. This process optimizes the functions towards a specific goal, which was clustering accuracy after a learning period in this work. However, other goals could be used, such as the prevalent activation function design goal of minimizing the number floating point operations used. Given that existing functions can be provided as seeds, as the adaptive LIF function was used in this work, costly functions like Hodgkin-Huxley model could be automatically simplified or approximated using CGP. This work was

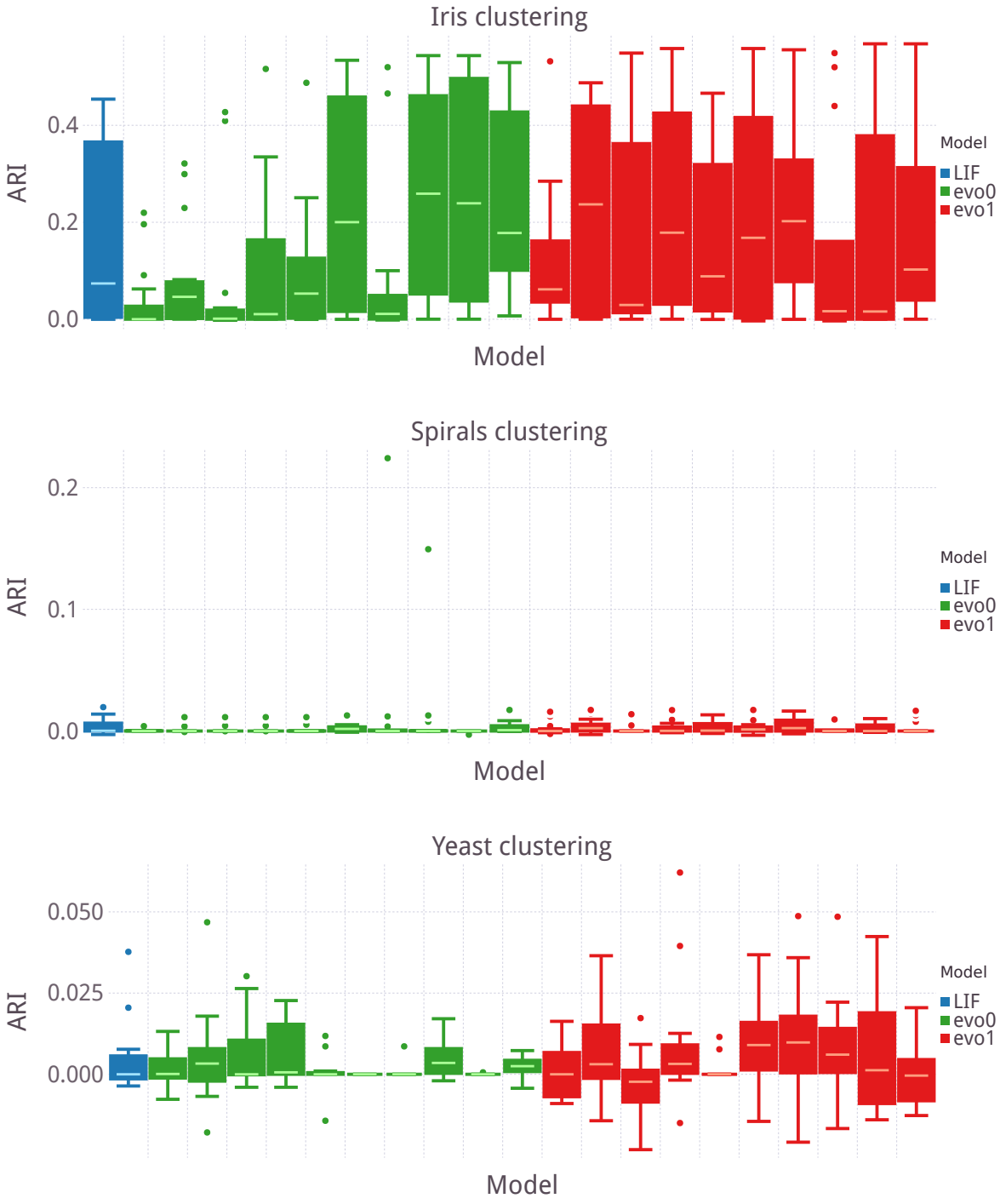

Figure 4.8: Generalization over different initial conditions. Each best individual from the 10 different evolutions of each type, and the LIF model, are tested on 20 random initial conditions.

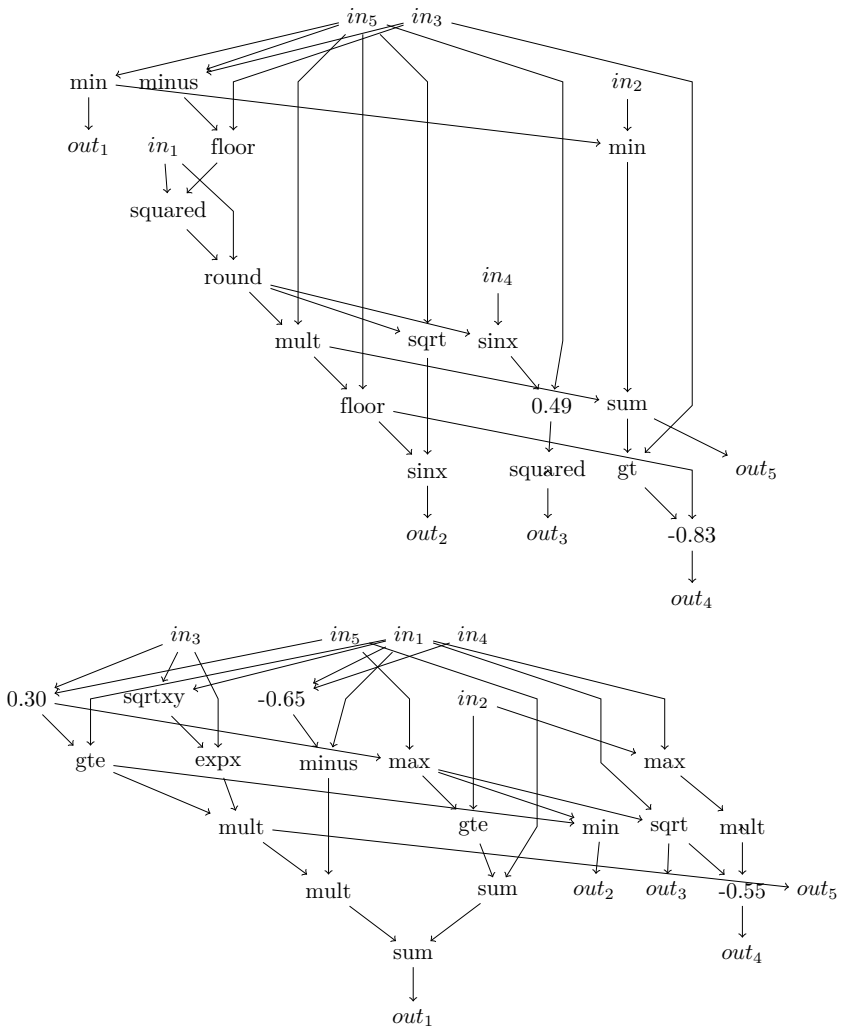

Figure 4.9: Best individuals from the two evolution types. On the top is a model from the **evo0** evolution and on the bottom is a model from **evo1**. The individual from **evo1** has similarities to the LIF model in Figure 4.4.

intended to demonstrate only that such an optimization is possible.

Unfortunately, the clustering accuracy achieved by all models, including the adaptive LIF model, were not competitive with state of the art models. Much of this work was based on [DC15] perhaps without sufficient adaptation for the different problems used here. In that work, MNIST was used, which has many more examples for each class, a larger input space, and geometric significance to the features, all of which were taken into account in the original work. For example, the inner layer of the network used in that work has 400 neurons, while MNIST has 10 classes, meaning each class is represented by roughly 40 neurons. In this work, for the iris dataset which has 4 features and 3 classes, we used an inner layer with 8 neurons, meaning each class was represented by only 2.7 neurons, on average. A different architecture could result in better base accuracy for the

adaptive LIF model and lead to a better evolution.

The difficulty of optimizing the neuron function with a fixed architecture, decided upon before the optimization, is one seen throughout this thesis. The components of a neural network are not independent, and changes to one part may demand changes to another. In this chapter, we isolate the neuron function and optimize it without changing the architecture nor the learning rule. In the next chapter, we will look at optimizing the developmental rules for an architecture without changing the neural activation function. The optimization of different parts at the same time constitutes a much more difficult problem, but, as this chapter shows, may be necessary to improve ANN capacity.

Chapter 5

Evolving developmental neural connectivity

Once each neuron is assigned a function, they must be organized in a way that best utilizes that function. This aspect of an ANN, its architecture, is one of the most important aspects for performance on artificial tasks. In deep learning, static topologies are nearly universal. These different topologies are crafted by engineers and researchers, and familiarity with a large selection of different architectures is considered a necessary part of working with ANNs. Different layer types, such as convolutional, pooling, or fully connected, form the connections between neurons, and these layers are combined by experts into architectures, which are then trained, tested, and modified in an expensive engineering process. Different tasks use different topologies crafted in this way; image classification uses architectures like VGG16 ([SZ15]), where information is condensed at each layer, whereas image reconstruction prefers something like U-net [RFB15], which also carries information from early to later layers of the same size. Model zoos, such as [JS15], host a menagerie of different static architectures trained on various tasks, the topology of each having been designed by hand.

Neuroevolution, the process of evolving neural networks, can alleviate some of this engineering. While some neuroevolution methods, such as HyperNEAT ([SDG09]), only optimize the weights of a network, requiring a fixed architecture to be designed, other neuroevolutionary methods also evolve the neural architecture. ES-HyperNEAT ([RLS10]) and CGPANN ([KMH11]) both evolve neural architecture in addition to synaptic weights. Developmental approaches such as Cellular Encoding ([GW93]) and L-system based neural networks ([HLP03]) evolve rules which then determine neural architecture. In this chapter, we follow a similar principle, evolving AGRNs which act in an environment to determine neural architecture.

One aspect of biological neural networks not well represented in ANNs, whether using

fixed topologies or evolving them, is the role of active topological changes during learning. Examples of this in neuroscience, however, are well-known. In [Val71], mice which had been raised in darkness were able to recover functionality in the visual cortex once placed in the light, developing new dendrites within days. The hippocampi of London taxi drivers was famously shown in [Mag+00] to change during the time of their employment, growing new dendrites to retain the complex map of the city and pruning these connections afterwards. Even neurogenesis, the development of new neurons, is now understood to occur in the adult brain, long having been considered a process unique to the developmental phase [SSG02]. Active dendritic development has been recently explored in [MWC18], where two CGP-based models control somata and dendrites, respectively, and grow and modify connections over learning. While dendritic development is a necessary component to understand in biological neural models and ANNs, in this chapter, we will focus on axons and their role in developing neural topology.

In the biological brain, axon guidance is at the base of neural topology. The structure of the brain, within the central nervous system and connections made to sensory organs and other parts of the body, is the result of axons being guided by a variety of cues during development. The local processeses of axon guidance and branching lead to the large architectures responsible for organized structures like the columns found in the visual cortex. This process is centered at the axon growth cone, which uses local chemical signals and neural activity to determine its movement ([KS08]). These mechanisms are beginning to be understood in biology, with examples such as the visual system providing insight to this complex process ([EH07]). However, while the axon guidance process is integral to development and cognition, it is not yet fully understood and there are few in-silica models of this process.

Both genetic factors and morphogenetic cues, such as netrins, are important in axon guidance. However, despite the seemingly concrete nature of neural topology, (eyes must connect to the visual cortex, which must then connect to other specific sections of the brain, for example), neural activity has also been shown to play an important role in axon guidance ([Pfe+07]). Axon guidance can therefore serve as a model for not only for static architectures but also for active ANN development, where neural activity modifies the architecture during use. As identified in Chapter 2, this essential improvement for ANNs may mitigate current issues in deep learning, i.e. catastrophic forgetting, and lead to increased ANN performance.

We therefore propose an axon guidance model for two reasons: first, to inspire developmental ANN models with the biological process responsible for much of the brain's architecture, and to move towards an in-silica model of axon guidance. Due to the importance of gene regulation in axon guidance, shown by the numerous gene knockout experiments in this domain, we use an AGRN as the controller of both morphogenetic

cues and axon decisions. An AGRN is optimized through artificial evolution to control glial cells, which secrete morphogens in a 3D space, and axon growth cones, which follow these morphogenetic cues to eventually connect with other neurons.

The model presented in this work therefore relies in the three factors identified in the biological literature: gene expression, morphogenetic signals, and neural activity. The model is an abstraction, with a parameterized number of morphogens which don't represent any specific axon guidance protein and an evolved artificial gene regulatory network (AGRN). This abstraction allows for flexibility of applying the model to different experimental configurations, where the number of axon guidance cues is known, and to allow artificial evolution to determine the relationship between an artificial morphogen and its biologic counterpart, if any. In this work, AGRNs are evolved in an experiment based on [Pfe+07], where neural activity disrupts the differentiation of axonal projections from the eyes into the visual cortex. Before presenting the model, we first present an overview of biological axon development and the state of the study and modeling of this process.

5.1 Biological axon development

Early in their life, neurons project multiple spines outward from their main body, the soma. All but one of these projections will become dendrites, forming the reception, or input, mechanism of the neuron. Neural output is directed along the other spine, the axon. This spine grows wider and longer than any of the others, and a specialized growth cone forms on its end. This growth cone extends thin, finger-like filopodia and flat, veil-like lamellipodia to sense and follow chemical pathways in the brain ([EH07]). These pathways often take the axon far from the soma, extending the neuron over a long and complex path. At certain points, the axon will form a branch, complete with a new growth cone. These branches may be pruned, retracting back to the axon base, or may specialize to form connections with dendrites of other neurons. A diagram of these different processes is shown in Figure 5.1. In general, one can observe three phases of axon development: 1) specification, the formation of the axon amongst the neural projections, 2) guidance, the movement of the growth cone, and 3) branching ([LCP13]). In this work, we focus on axon guidance and branching.

[Chi06] and [EH07] offer comprehensive reviews on the subject of axon guidance. Early experiments on frog neurons in lymph clots demonstrated that axons had a surprising motility, but it was decades before the mechanisms underlying their movement was better understood ([Chi06]). Four major different types molecular signal have been identified: ephrins, semaphorins, netrin, and slit. In [EH07], a detailed list of the different molecules involved in retinal gangion cell (RGC) guidance is given, explaining the different role each

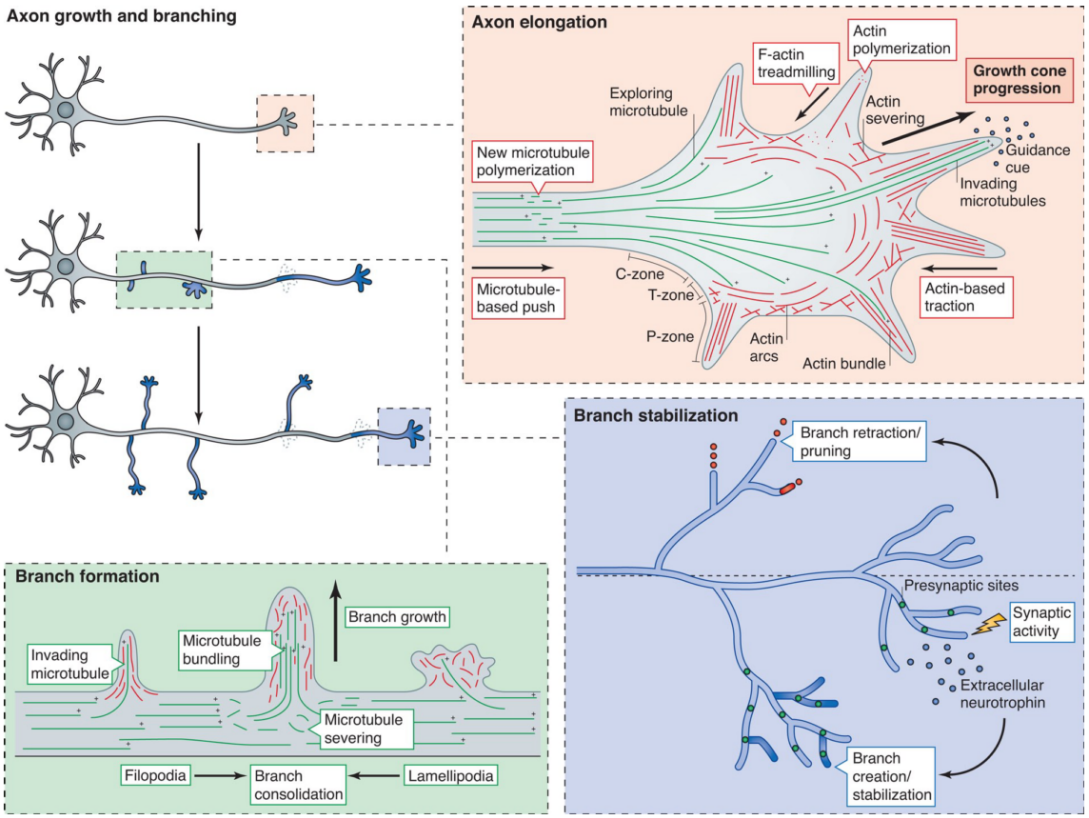

Figure 5.1: A diagram of different stages of axon development from [LCP13]. Axon guidance is presented in the red box, which details the axon growth cone, showing the actin-based filopodia and lamellipodia and the guidance cues they follow. In the green box, newly formed branches are highlighted, showing how the microtubules forming the axon cytoskeleton change during branching. Finally, the blue box shows the differentiation and pruning steps of axon development, where synaptogenesis with other neurons leads to activity in the branches.

molecule is known to play. The way in which the axon growth cone changes to follow or be repelled by certain molecular signals is shown in Figure 5.2.

An example of the flexibility of these molecular cues is found in the ephrin family. Ephrin-A5, one of the subgroup of ephrins called ephrin-As, is responsible for guiding RGC axons to the superior colliculus, a part of the midbrain, where they form connections in vast axonal arbors. In [Hub+09], ephrin-As were demonstrated to be necessary for patterning eye-specific projections to the lateral geniculate nucleus (LGN), a part of the thalamus. In the LGN, these axons form connections in discrete layers. The same morphogenetic cues, ephrins, are responsible for these two very different morphologies, a topographic map in the superior colliculus and discrete layers in the LGN ([Chi06]).

In [KS08], the way in which Netrin and Slit influence the axon's direction and axis of migration was explored in a review of experiments on *Drosophila*, fruit flies, and *C. elegans*, roundworms. While much of the work on axon guidance reviewed in [Chi06] and [EH07],

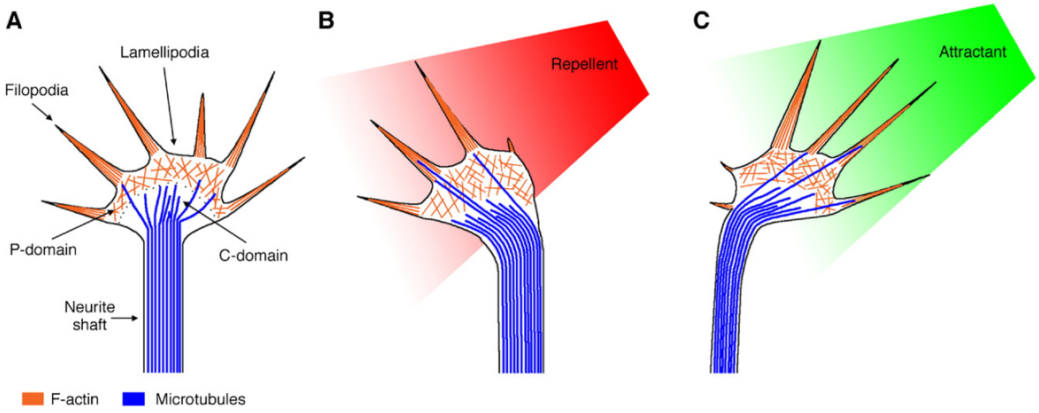

Figure 5.2: Diagram from [EH07] of the axon growth cone. In (A), the structure of the growth cone is shown, with fully extended filopodia and lamellipodia, constructs based on F-actin. These extensions retract or extend in the presence of a repellent (B) or attractant (C), respectively.

has been based on experiments in mammalian visual cortices, this review demonstrates how these molecular signals impact axon guidance over the entire organism. It also shows a stunning similarity in these organizational mechanisms across evolution; for example, Slit and Netrin are expressed at the midline in mammals, zebrafish, *Drosophila*, and *C. elegans*. These same cues guide axons across the midline in all of these organisms, with a balance of Netrin attraction and Slit repulsion delicately controlling the migration down and crossing of the midline, whether that be the complex spinal column of mammals or the simple axis of *C. elegans*.

These chemical cues are not the only factor in axon guidance, however. The process of eye-specific patterning detailed in [Hub+09], where ephrin-As guide axons to develop into distinct layers in the LGN, was also the subject of experiments in [Pfe+07]. In [Pfe+07], neural activity was also demonstrated to act with ephrin-As to control this patterning. Normally, activity in RGCs leading to the LGN comes in waves; bursts of action potentials are followed by long periods of silence, leading to correlated activity. These retinal waves, as well as ephrin-As concentration, were disrupted in mice populations. Three populations of mice were analyzed: (A) mutated mice which were deficient in ephrin-As, (B) normal (wild-type) mice without correlated activity, and (C) ephrin-A deficient mutants without correlated activity. In (A), eye-specific inputs segregated as normal, but the location and shape of the layers was greatly changed. In (B), axons from the different eyes did not segregate into separate regions. In (C), the retinal projections, meaning axons from the two eyes, overlapped and were in the wrong region of the LGN. (B) shows that activity plays an important role in axon guidance, as the eye-specific inputs did not segregate in wild-type mice, but the difference between (A) and (C) shows that activity also cooperates with chemical signals to properly guide axons.

Some of the same mechanisms involved in axon guidance have been shown to also play a role in axon branching ([GS00]). [GM11] presents a review of the different axon branching processes in vertebrate nervous systems. Like axon guidance, these processes rely on extracellular cues and neural activity. Wnt, a protein family shown in [KS08] to influence axon guidance, is amongst the extracellular cues which determine branching. [BB11] gives a detailed explanation of the role of each molecular cue involved in axon branching, and can be compared to [EH07] to see the similarity of the mechanisms underlying these processes. Neural activity also plays a part axon branching, influencing the rates of branch addition and retraction and the competition between neighboring branch arbors ([GM11]).

While these studies represent enormous progress in the understanding of axon development, there is still much to be done. Specifically, many of the experiments informing these works were conducted *in-vivo* in the developmental phases of mammals, and a large portion of them focus on the visual cortex. In these *in-vivo* experiments, it can be difficult to understand all factors present in the environment, and harder still to control them. [KLH11] presents a review of experiments done in cortical slices, which suggest that the same mechanisms which regulate axon growth and guidance *in-vivo* are at work dissociated neural cultures. These *in-vitro* experiments can offer a new way in which to study axon development, allowing greater control over and observation of all of the possible factors at work in the creation of neural architectures in general.

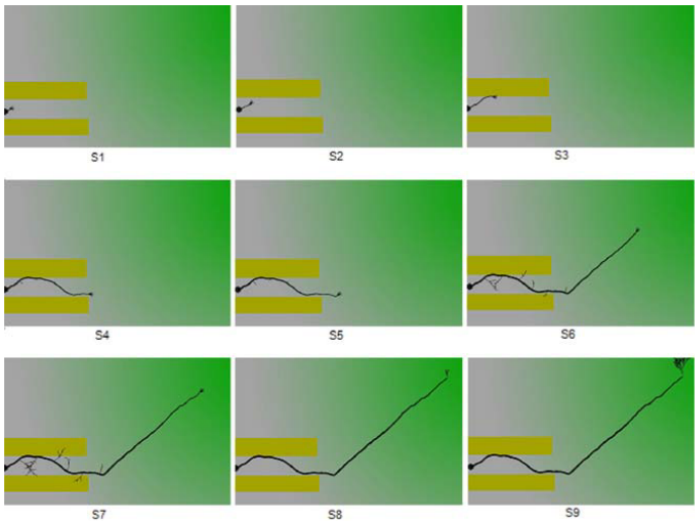

Figure 5.3: Results from [FNZ05], where an L-system axon growth cone follows a set of designed rules to avoid the two yellow rectangles and develop in the direction of the attractant, shown in green.

A further aid to this domain would be precise *in-silica* or computational models of axon development. Such models could reduce the number of experiments necessary to test certain hypothesis, could investigate situations difficult to produce in experiments, and

could offer richer insights into the entire developmental process. Despite these benefits, very few computational models of axon development exist. An early model of chemical cues and their effects on axon guidance is given in [HV99], where morphogens are modeled using differential equations and guide the movement of a bundle of neural projections, which form the axon. Different hand-written models of morphogen diffusion equations and axon guidance rules are compared. In [FNZ05], an L-system controls the guidance, growth, and branching of an axon inside an environment with extracellular morphogens. The article presents the framework, open to any L-system rules, and demonstrates an example using hand-written L-system rules to grow a single axon in the direction of a chemoattractant. The axon growth from this work is shown in Figure 5.3.

Computational models of axon guidance can not only inform biological experiments, but can also improve ANN models. Morphogenesis was used in early works in developmental ANNs, notably in [FB92], where chemical signals controlled the movement of neural projections, eventually forming the connections of an ANN. In [Bal03], continuous time recurrent neural networks (CTRNNs) were designed via a morphogenetic process, where the rules of morphogen diffusion were controlled by evolution.

In the next section, we present a model of axon guidance and branching which uses evolved rules for both the axon decisions and the control of the morphogens. This model aims to be between these two domains: close enough to the biological reality to be used to model it, but simple enough to be integrated into ANN models used for computation. The use of evolution allows for this wide scope; by changing the evolutionary fitness, the goal of the model can be precisely defined.

5.2 Axon guidance model

The model simulates cells and morphogens in a 3D space, a unity cube. Two cell types are simulated: glial cells and neurons. Glial cells regulate $N_{morphogen}$ different morphogen concentrations in the environment and neurons project axons, which navigate within the environment by following or moving against morphogen gradients. The actions of both cells are controlled by an evolved AGRN, which can be evolved for any fitness. In this work, we explore evolutionary fitnesses based on final neural topology, on activity-based growth, and on performance in a simulated robotic environment.

5.2.1 Cellular models

Neurons and glial cells are both modeled and have different capacities based on cell type. In Figure 5.4, the two cell types are shown. The two cell types are controlled by a single AGRN, a distinct copy of which is placed in each cell. By acting as the controller for these

two cell types, the AGRN is responsible for morphogen diffusion in the environment, axon guidance, and axon branching.

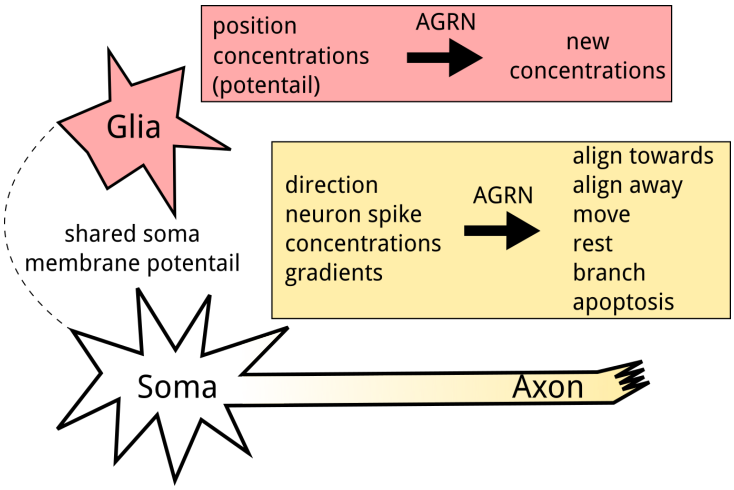

Figure 5.4: The two cellular models. Glial cells use an AGRN, here pictured in pink, to process information about their location and nearby morphogen concentrations to diffuse new morphogens. Some glial cells are attached to a soma and also sense its membrane potential. Somas have no AGRN, but do have a membrane potential, location, and an axon. The axon can have multiple growth cones, each of which is controlled by an AGRN, here pictured in yellow, which is used to process morphogen gradients and neural information to choose between six possible actions.

The neuron is modeled as two separate parts, the soma and the axon. The soma has a position, a membrane potential, a type, and also keeps track of the synapses formed with it. The neuron type is based on the environment; it indicates problem-level specifics such as an input neuron, an eye neuron, or an output neuron. The soma is not controlled by the AGRN; it is only a vehicle for spiking activity. Rather, the AGRN controls the axon growth cone, which has a position and a direction. The AGRN decides between four to six possible actions for the axon at each update step: 1) align the growth cone direction with a morphogen gradient, 2) align opposite a morphogen gradient, 3) move in growth cone's direction, 4) quiescence (rest), and, depending on model parameters, 5) branch (duplicate), and 6) apoptosis (pruning or cell death). The other outputs of the AGRN specific to the axon growth cone decide which morphogen gradient to align to or away from, and specify the length of the movement taken.

The glial cell model by comparison is very simple. In general, glial cells only have a position. If a glial cell is placed at the same location as a soma, it also uses the neuron type and membrane potential as AGRN inputs. The other AGRN inputs for the glial cells are the glial cell's position and the morphogen concentrations at that position. The outputs of the AGRN decide the concentration of each morphogen emitted by the glial cell. The inputs and outputs of the AGRN are not shared; inputs which are not used by a cell type are set to 0 in the other cell type, and outputs which are not used are ignored.

The neurons and glial cells are the only biological components simulated in this model, and both are fixed in space; they do not move. In biological neural development, neurons and glia move, and other cells inform the distribution of morphogenetic guidance cues, as well as impact axon guidance through contact. For simplicity, and to focus this model on the study of axon guidance, only the axon growth cone is able to move. This choice makes the placement of neurons and glia an important configuration option in the design of the environment. Future work with studying neurogenesis could start with a single cell and allow for cell duplication and movement.

5.2.2 Environment initialization

Each morphogen is modeled by a concentration c_i on a discrete grid in the unity cube (each dimension in $[0.0, 1.0]$). The precision of the grid is given by the parameter G, which indicates the number of points in each physical dimension. This morphogen grid is then used to construct a continuous 3D morphogen space using linear interpolation. In this work, $G = 20$ was used throughout, meaning points in the grid were spaced evenly at a distance of 0.05. The morphogen grid and interpolation method is used to allow precise morphogen concentration calculation while not depending on the number of elements in the model. At initialization, the grid is filled with a concentration of c_{start} for all morphogens.

Cells are placed throughout the space according to the problem configuration. For the first work presented here, the eye projection differentiation experiment, neurons were placed in two clusters, representing eyes, and throughout a section representing the brain, and glial cells were placed throughout. In the robot experiment, neurons and glial cells were placed in distinct 2D layers evenly spaced in the third dimension. A glial cell is placed at the same location as each soma and is considered linked to it for AGRN input.

Neurons are initialized with membrane potential of V_{start} and a single axon growth cone, which is placed at the same position as the neuron and has a random direction. All AGNs are initialized with the same concentration for each protein.

5.2.3 Environment update

At each timestep, external inputs are fed into the input neurons in the environment by directly increasing their membrane potential. Neurons, axons, and glial cells are then updated according to their respective update step count, U_{neuron}, U_{axon}, and U_{glia}. In these experiments, $U_{neuron} = 1$ and U_{axon} and U_{glia} are both 5, meaning neurons are updated every timestep, while axons and glia are updated every 5.

Morphogens are updated during glial cell update, as glial cells are the only controller which can change morphogen concentrations. Morphogen concentrations, c, are bound

between $[0, 1]$ and decay exponentially to model natural absorption according to an absorption rate parameter $\tau_{absorption}$:

$$\tau_{absorption}\frac{dc}{dt} = -c \tag{5.1}$$

Glial cells diffuse morphogens based on output proteins of their AGRN, d_i, where the AGRN has a dedicated output for each such morphogen. The morphogen grid is updated based on the euclidean distance, D of each point in the grid, $[x, y, z]$ to the glial cell, $[x_g, y_g, z_g]$. Two model parameters, $\tau_{diffusion}$ and $\beta_{diffusion}$, control the speed and distance propagation respectively of diffusion:

$$D = \sqrt{(x - x_g)^2 + (y - y_g)^2 + (z - z_g)^2} \tag{5.2}$$

$$\tau_{diffusion}\frac{dc_i[x, y, z]}{dt} = e^{-\beta_{diffusion}Dd_i} \tag{5.3}$$

Updates for somata modify their membrane potential V, integrating synaptic inputs if there are any. Conductance based synapses in a leaky integrate and fire (LIF) are used. Only excitatory synapses are considered in this model, where each excitatory synapse has a conductance g_e which decays exponentially and is increased by presynaptic firing:

$$\tau_m\frac{dV}{dt} = V_{start} - V + g_e(E_e - V) + I_{external} \tag{5.4}$$

$$\tau_e\frac{dg_e}{dt} = -g_e \tag{5.5}$$

V_{start} determines the baseline membrane potential which the neuron returns to in the absence of activation from synapses, g_e, or from external input, $I_{external}$. Only input neurons receive external stimulation. E_e determines the level of synaptic activation, and is set to 0 in these experiments, making the synapses highly efficient. Finally, τ_m and τ_e determine the update speed of the neurons and synapses, respectively. A neuron is considered to spike when its membrane potential exceeds V_{thresh}, at which point the spike propagates down the axon. All synaptic weights in this model are fixed to 1.0. The spike is used by the axon as an AGRN input, and if the axon has connected with other neurons, it excites their synapses.

Axon growth cones follow morphogen gradients to move in the space. Based on their AGRN outputs, at each axon update step, the axons can rotate towards or away from a morphogen gradient, move in their current direction, rest, divide, or be pruned. Rotation immediately sets the axon growth cone on or against the direction of the gradient, but movement speed is determined by the parameter $\tau_{movement}$. Two AGRN outputs, o_{m0} and

o_{m1}, determine the magnitude of the growth cone's displacement, r, which is then used with the axon's direction $\vec{d_{a_i}}$ to update the growth cone's position, $\vec{p_{a_i}}$:

$$r = \frac{o_{m0}}{o_{m0} + o_{m1}} \quad ; \quad \tau_{movement} \frac{d\vec{p_{a_i}}}{dt} = r\vec{d_{a_i}} \tag{5.6}$$

When an axon is within $D_{synapse}$ of a neuron, the axon and neuron can form a synapse. Post-synaptic neurons have a maximum number of possible axons they accept, $N_{synapse}$. If a neuron has this number of input synapses, it is no longer considered as a valid target during synaptogenesis. Neuron type can also inform synaptogenesis as required by the problem setup; in the eye-specific patterning experiment, synapses are limited to eye neurons connection to brain neurons, and in the robot navigation experiment, they are limited from one layer to downstream layers.

Axon branching and pruning are both controlled by model parameters. Somas keep track of all axon growth cones resulting from axon branching and have a minimum of 1 and a maximum of N_{branch} branches. When the number of branches is less than N_{branch}, axon growth cones are able to activate a duplicate action to create a new branch. This creates a new axon growth cone at the same position as the current axon growth cone, with a new copy of the AGRN, which remains in place for $\tau_{duplication}$ timesteps. After this time, the new branch functions like any other axon growth cone. Similarly, when the number of branches for a neuron is above the minimum of 1, each axon can activate an apoptosis action, which prunes the axon growth cone. In biology, pruning involves the retraction of an axon growth cone along the axon, tracing back to its original point of branching. In this model, however, axons do not have a physical representation, but are considered as points at each growth cone. Pruning therefore removes one of the growth cones but does not alter other components of the axon.

5.2.4 Model configuration and evolution

The model has a number of parameters adaptable for a variety of use cases. The initial cell placement and model parameters can be used to replicate experimental data with different neural cell types. Artificial problems can be examined using more conventional artificial architectures, such as layers.

However, the core of the flexibility of the model is from the evolution of the AGRN, as the AGRN controller drives the decisions of all cell actions. In the next section, we will investigate different evolutionary fitness metrics. The generality of an evolutionary goal is an important part of this model, as the AGRN can be optimized to match biological axon behavior or to perform in an artificial task easily, only by designing an evaluation function of the model that returns a fitness value. Many learning tasks for artificial neural networks

require the learning goal to be differentiable, but by using evolution, this requirement is not needed.

5.3 Eye-specific patterning

In this experiment, we simulate the eye-specific patterning found in biology. Neurons from two eye clusters project axons into a brain section, and the evolutionary fitness is based on the resultant neural topology, rewarding synaptogenesis and differentiation from the two eyes into different sections of the brain. We then disrupt the neural input to the eyes to investigate the impact of activity on topology, as in [Pfe+07].

5.3.1 Visual system environment

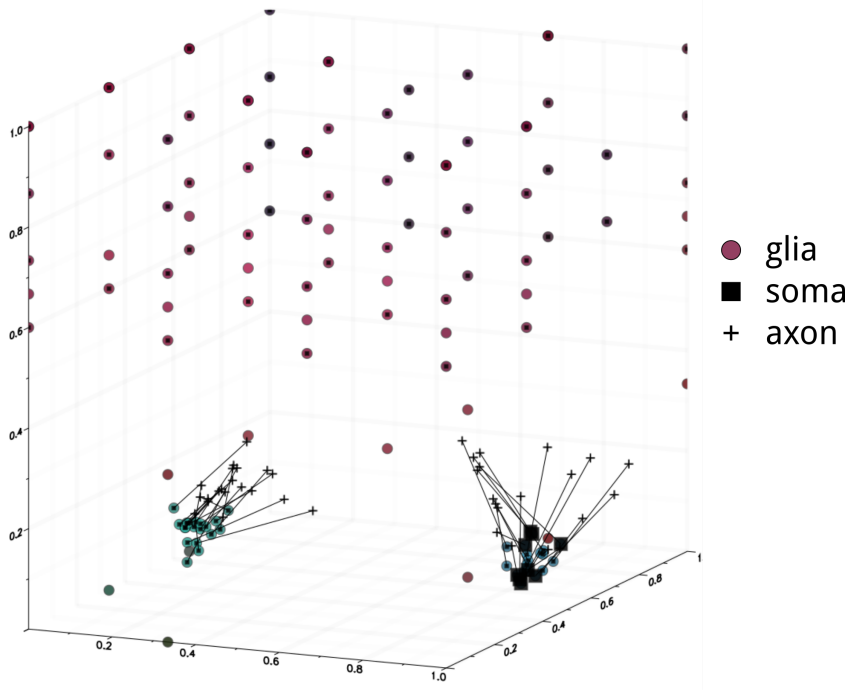

Figure 5.5: The environment, rendered in 3D at t=220 ms. Glial cells are shown as circles with coloration using the morphogen concentrations as RGBA values. Somata are represented by squares, with the size of the square corresponding to spiking activity. At the timestep shown, some neurons in the right eye are firing. Axon growth cones are represented by crosses, and a line connects each growth cone to its respective neuron.

The initial cell placements are based on the eye biology, but are highly simplified. In biology, RGCs travel through sections of the brain to reach the LGN, where they develop into distinct layers. Here, we observe connections made directly in front of the eyes, with

no physical structure like the skull influencing axon guidance. A 3D rendering of the environment is shown in Figure 5.5.

Eyes are modeled as two distinct clusters. N_{eye} eye neuron positions are placed around $x = 0.5 \pm 0.5 * D_{eye}, y = 0.5, z = z_{eye}$ following a normal distribution with a variance of σ_{eye}, where the left eye is centered around $x = 0.5 - 0.5 * D_{eye}$ and the right eye around $x = 0.5 + 0.5 * D_{eye}$. Each eye neuron has a corresponding axon growth cone and glial cell.

N_{brain} brain neurons are spaced evenly between $z = z_{brain}$ and $z = 1.0$. This three dimensional volume is split into equal sections and a neuron is placed at the vertices of each section. If N_{brain} is less than the number of vertices, neurons are placed randomly at vertices until N_{brain} is reached. Each neuron in the brain has a corresponding glial cell but does not have an axon growth cone. While this is not biologically plausible, axon projections from the brain were not considered part of this study and were therefore left out of the model.

N_{glia} extra glial cells are placed throughout the environment. These glial cells follow the same evenly spaced placement method as brain neurons, but they are placed throughout the entire environment from $z = 0$. The total number of glial cells in the environment is therefore $2N_{eye} + N_{brain} + N_{glia}$, and the total number of neurons is $2N_{eye} + N_{brain}$. A full list of the parameters used in this experiment is given in Table 5.1.

$N_{morphogen}$	4	c_{start}	0.1
N_{brain}	60	V_{start}	-70 mV
N_{eye}	20	V_{thresh}	-55 mV
N_{glia}	30	τ_m	20 ms
$N_{synapse}$	1	τ_e	20 ms
N_{branch}	1	$\tau_{absorption}$	10 ms
$D_{synapse}$	0.08	$\tau_{movement}$	200 ms
D_{eye}	0.8	$\tau_{duplication}$	100 ms
σ_{eye}	0.03	$\tau_{diffusion}$	40 ms
z_{eye}	0.1	$\beta_{diffusion}$	6.0
z_{brain}	0.6		

Table 5.1: All model parameters for the eye-specific patterning experiment

As in biology, the expected outcome is that the axonal projections differentiate in the visual cortex during periodic activity but fail to differentiate under non-periodic activity. Eye neurons are provided with periodic activity during evolution, and then we examine the performance of the evolved AGRN when eye input is random. The fitness during evolution measures the differentiation and requires a specific topology, i.e. axons are

required to cross to the opposite side on the x axis. The fitness is increased by 1 for each right eye neuron that connects to the left brain section, and vice versa. Incorrect connections, such as from the right eye to the right brain section, are still rewarded in order to encourage forming connections during evolution, but much less: each connection is worth 0.1.

5.3.2 Evolutionary results

The best AGRN from a single evolution was used to evaluate the simulation. While the results from different evolutionary runs will vary greatly, the strategies seen from a sample of 10 different evolution runs displayed similar traits. In the topology created by this evolved individual, seen in Figure 5.6, the eyes were able to separate into different sections of the brain. Some eye neurons did not form connections to the brain, as it seems that the evolutionary penalty of creating an incorrect connection discouraged this behavior. There are few incorrect connections made, with a small number of neurons from the right eye extending into the right section of the brain.

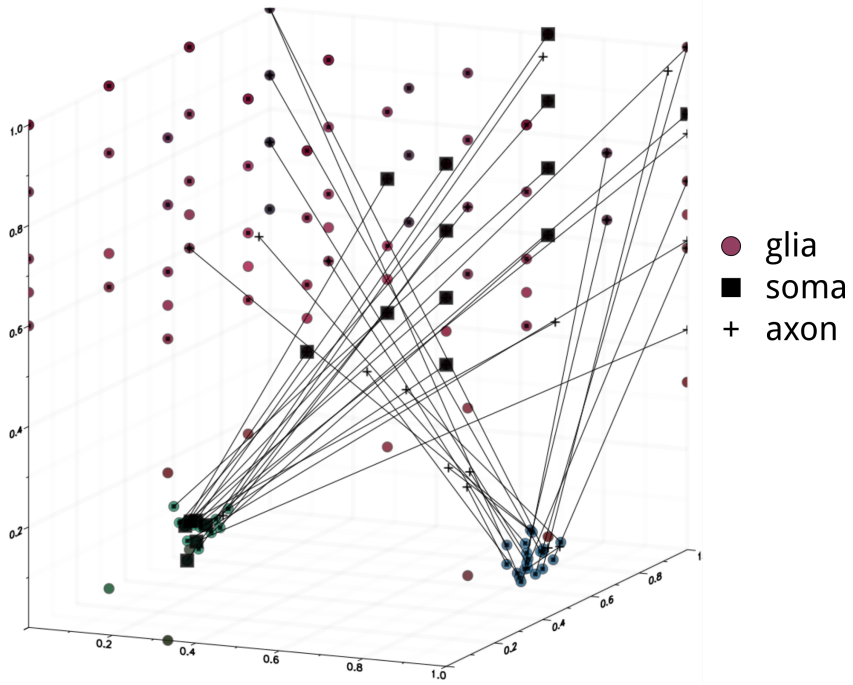

Figure 5.6: The final topology created by the AGRN. Most axons reach into the brain section and cross to the other side, although some axon growth cones from the right eye extend to the right section of the brain, leading to fewer connections in the left side of the brain. The left eye is almost entirely connected to the right side of the brain.

To understand the behavior of the AGRN, we observe the morphogen concentrations emitted by the glial cells. From the first timestep, drastic changes in morphogen con-

centration are observed, with different morphogens appearing to serve different roles. In the best individual examined, morphogens 2 and 4 hardly differentiated between the left and right brain sections, making them difficult guidance cues for axon patterning. Morphogens 1 and 3 displayed higher concentration differentiation in the brain, although it was still minimal. All morphogens exhibit differences between the two eyes, allowing for clear differentiation from the axons and driving the eye-specific patterning.

The concentration distributions were mostly unchanging over the duration of the simulation, with little difference between the morphogen concentrations at $t = 1000$ and $t = 3000$. As there was no reliance on simulation time in the model, this is not entirely surprising, although it appears that the activation of the brain neurons did not play a role in morphogen distribution. As connections formed over the course of the simulation, activity in the brain neurons increased, and while the majority of glial cells in the brain sensed this activity in the membrane potentials of their nearby neurons, it did not influence their morphogenetic output. The morphogen concentration distributions are shown in Figure 5.7.

To understand the importance of neural activity to the axon development, we replace the periodic neural activity with random activity. The resultant activity is shown in Figure 5.8 and the formation of connections is shown in Figure 5.9. Activity is disrupted in each eye alone and then in both eyes. The main impact of this change is that axons did not form connections with as many neurons in the brain section. Each eye disruption drastically reduced the number of connections made, but the connections that remained were often the same connections made during normal eye activity.

The reduced connectivity of the axons during disrupted neural activity and the relatively low connectivity throughout evolution suggest that the model does not favor synaptogenesis as strongly as biological axon guidance. In [Pfe+07], RGCs without correlated visual activity still formed synapses in the LGN; only the segregation of these synapses into different sections was disrupted. Many possible reasons exist for this dissimilarity between the model and biology, but one large factor could be contact guidance for the axon growth cones. In the biological brain, axons make contact other neurons and glial cells, and the wide dendritic trees of neurons in the brain may guide the axons towards synaptogenesis. However, in the model, dendrites are not present and neurons may not exhibit a particularly strong signal to indicate their presence to nearby growth cones. This could be a future enhancement to the model, although the simulation of the physical characteristics of the cells to model contact guidance would be a computationally expensive addition. Other methods could include a proximity signal of cells with a much smaller radius of diffusion than the other morphogenetic signals, which all use the same $\beta_{diffusion}$ tuned to allow communication across the entire 3D space.

This experiment was inspired by the experiment conducted in [Pfe+07], and a major

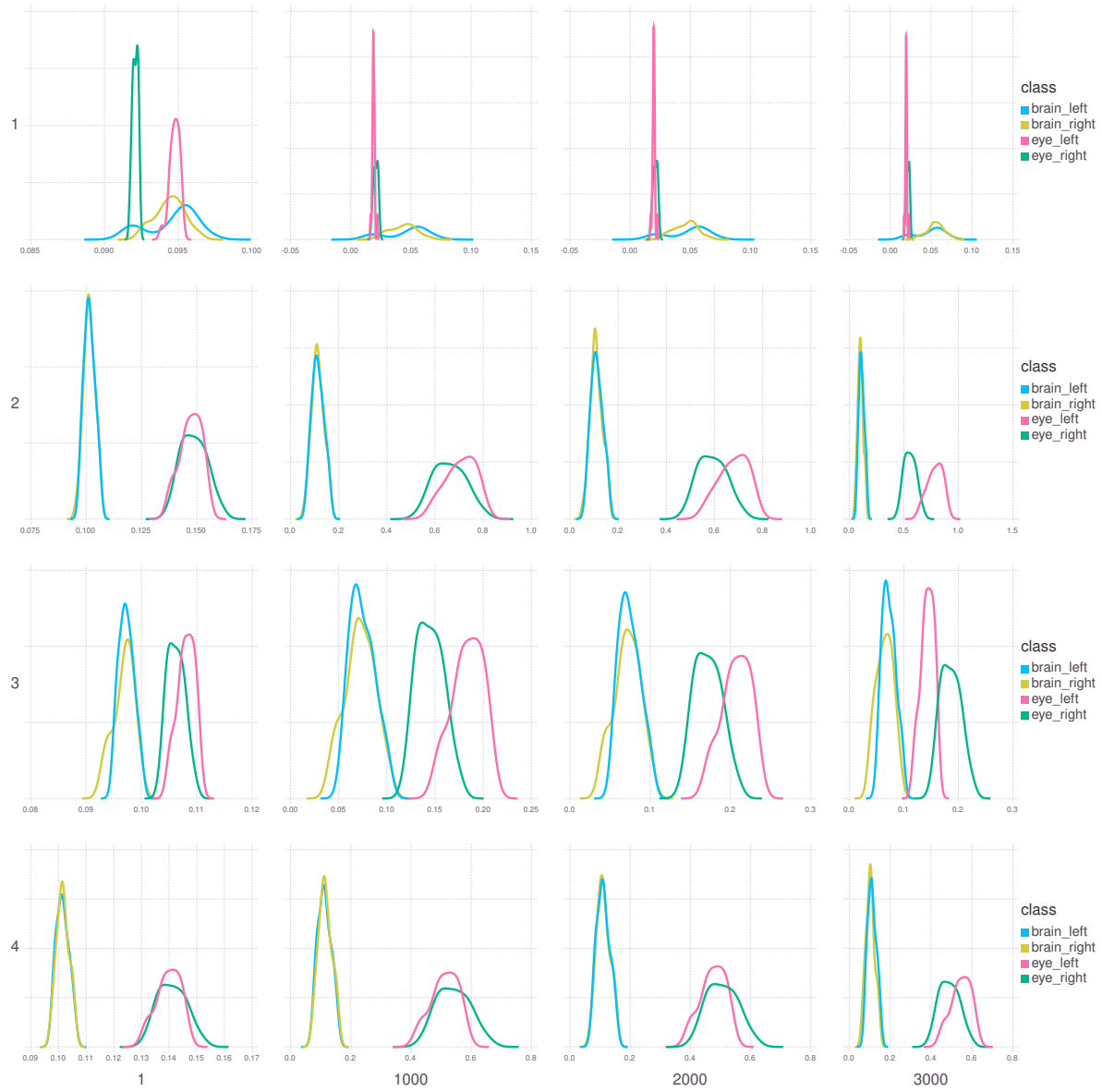

Figure 5.7: Morphogen concentration distributions during the simulation. The four different morphogens are shown along the y axis, and four timesteps, from the first timestep to the last, are shown along the x axis. All four morphogens exhibit different concentration distributions between the two eyes, although the difference is slight in some, i.e. morphogens 1 and 4. Morphogens 1 and 3 differentiate between the left and right sections of the brain, while morphogens 2 and 4 have nearly identical concentration distributions in these sections.

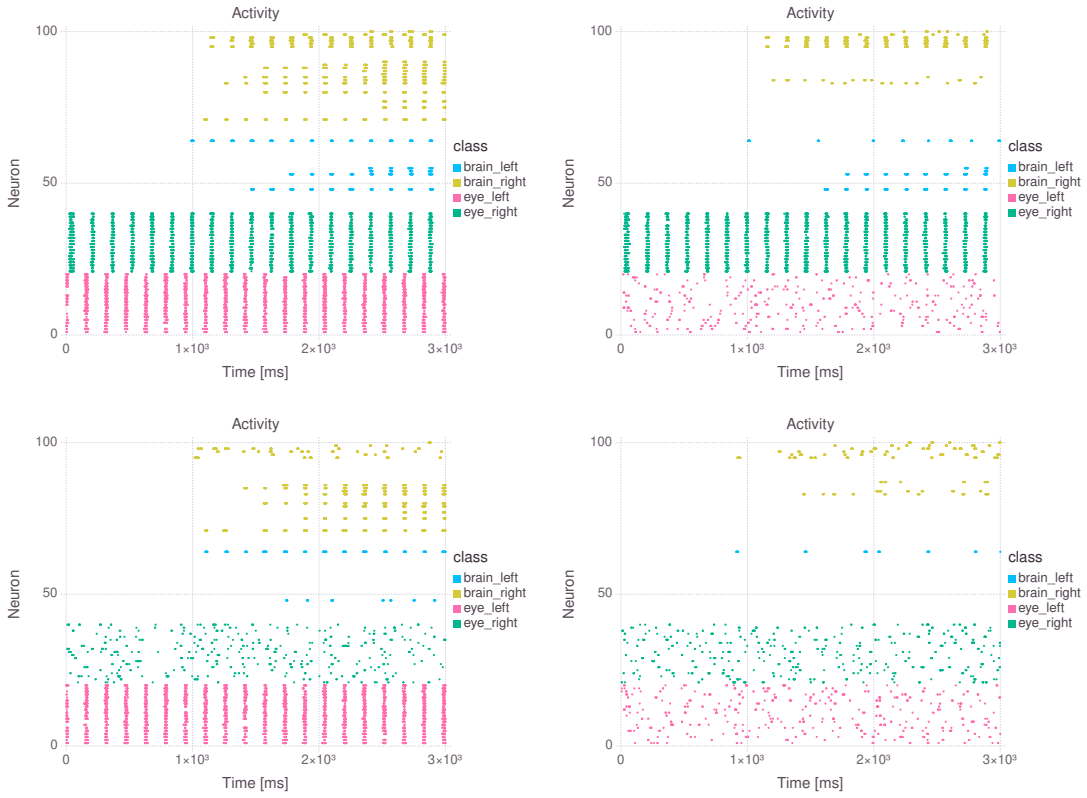

Figure 5.8: Neural activity in the four cases examined. In the top left, both eyes are provided with periodic spiking activity. This is the case used during evolution, and the most brain activity is visible in this case. In the top right, the left eye is disrupted, which leads to a large decrease in the activity in the right brain section. In the bottom left, the right eye is disrupted, leading to some disruption in the left brain; the left brain had lower connectivity in the normal case, however. Finally, in the bottom right, both eyes are disrupted, leading to the least activity in the brain.

design decision in this simulation was the placement of the neurons. The visual cortex, while often the subject of study in axon guidance, is much larger than in this simulation, and the position of all of the different neurons and glial cells is impossible to know precisely. However, other models of neural development have used simpler organisms with well-known and simpler morphologies. In [IB18], a model of locomotion in *C. elegans* is shown, modeling detailed neuromechanical processes. This type of experiment could be a further study using the axon development model presented here.

This work demonstrates activity-dependent development is possible for artificial neural networks, i.e. that different neural topologies arise from different input activity patterns. This reliance on activity arose naturally during evolution without any explicit requirement for activity-dependence in the evolutionary fitness. Evolution was only exposed to the single case of periodic activity in both eyes, and the other activity cases were only used during testing of the evolved individual. In the next experiment, we will investigate the use

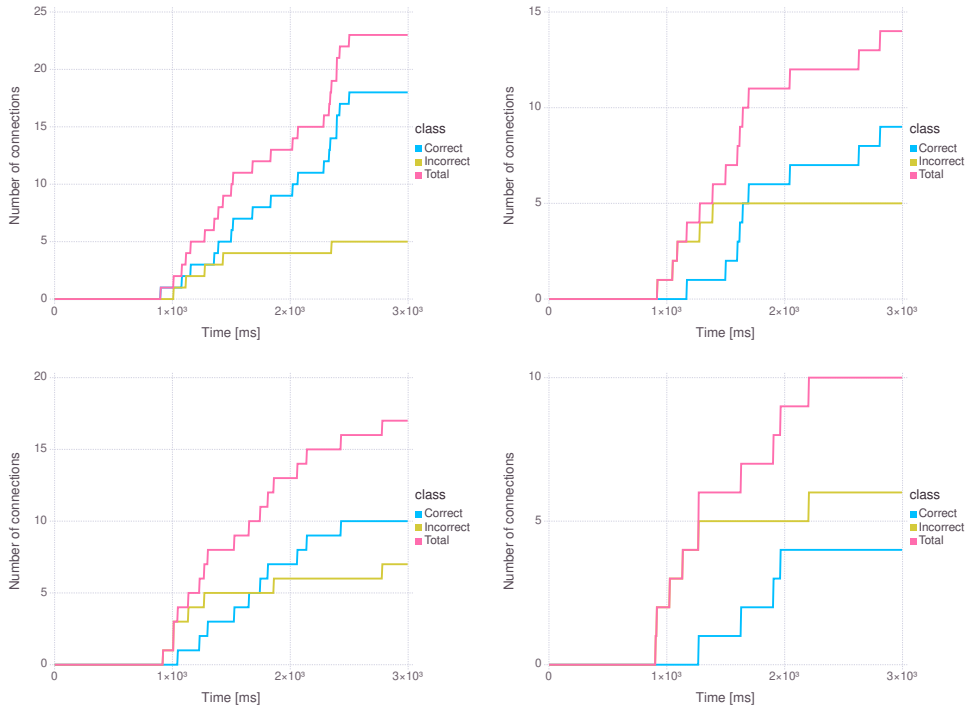

Figure 5.9: Synaptogenesis in the four different neural activity cases examined. In the top left, both eyes are given normal input and lead to a majority of correct connections, where the axon crosses the x axis midline. In both the top right and bottom left, where the left and right eyes are disrupted, respectively, the number of correct synapses formed decreases. Finally, when activity is disrupted in both eyes, shown in the bottom left, the number of correct synapses drops below the number of incorrect synapses.

of this model with an evolutionary fitness designed to rely on activity-based development. Specifically, the task of robot navigation is used with the goal that robot inputs will influence the development of a network able to control the robot in a foraging task.

5.4 Robot coverage

To understand the utility of this model in artificial tasks, a robotic foraging environment was used. In this environment, a two-wheeled robot navigates through a 2D space to consume food placed randomly throughout the environment. The evolutionary fitness is the amount of food the robot is able to consume in a constant amount of time. The robot receives input informing it about nearby food and activates output neurons to move. The 3D shape of neural environment is used in both input and output encoding. This environment is displayed in Figure 5.10.

For the inputs, 8 different sensors were represented, which project in rays from the front of the robot. If food is within one of these rays, a corresponding column of neurons is activated. The number of neurons, from the bottom to the top, which is activated

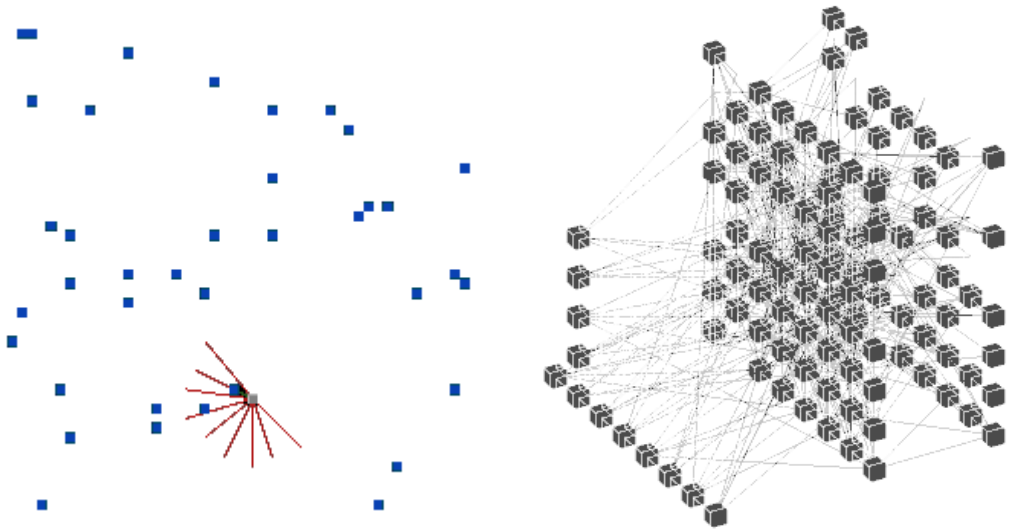

Figure 5.10: The robot foraging environment and the neural network controller of the robot at the same timestep. The robot senses food in the second of its 8 radar areas, activating most of the corresponding neurons of this input column to indicate the close proximity of the food. Squares indicate active neurons at this timestep; inactive neurons are not shown.

corresponds to the distance of the food; if the food is very close, all neurons in the column are activated, but if the food is far away, few neurons will spike. A baseline of 1 neuron per column receives constant excitation. This activation comes in the form of a strong $I_{external}$ which forces the neuron to spike. The input neurons are all located at $z = 0$.

Output neurons were positioned at the other end of the 3D space, at $z = 1$. These are split at $x = 0.5$ into a left side and right side. Neural spiking on the left output side turns the left wheel, and spiking on the right side turns the right wheel. The more neurons which spike in a given side, the faster the wheel turns, meaning the neural network can control robot speed and direction using the same output layer. The inputs and outputs therefore require spatial translation; food proximity activates more neurons in the y direction, but output activity must be split along the x axis to properly control the robot.

Intermediate neurons were placed evenly throughout the environment, at each vertex between the input and output layers. The 3D environment is split into 8 sections in each dimension; there are 64 input neurons, 64 output neurons, and 384 intermediate or "hidden" neurons. When observed in the z dimension, this forms 2D layers of neurons, with one input layer, 6 intermediate layers, and one output layer. Each neuron i could connect only to another neuron j if $z_j > z_i$, meaning that neurons connected to other neurons in the same 2D layer as them, and in downstream layers. This created topologies similar to contemporary ANNs, with the addition of connections within a layer and connections that skip layers. Synaptic weights were also constant in this model, as in the previous experiment.

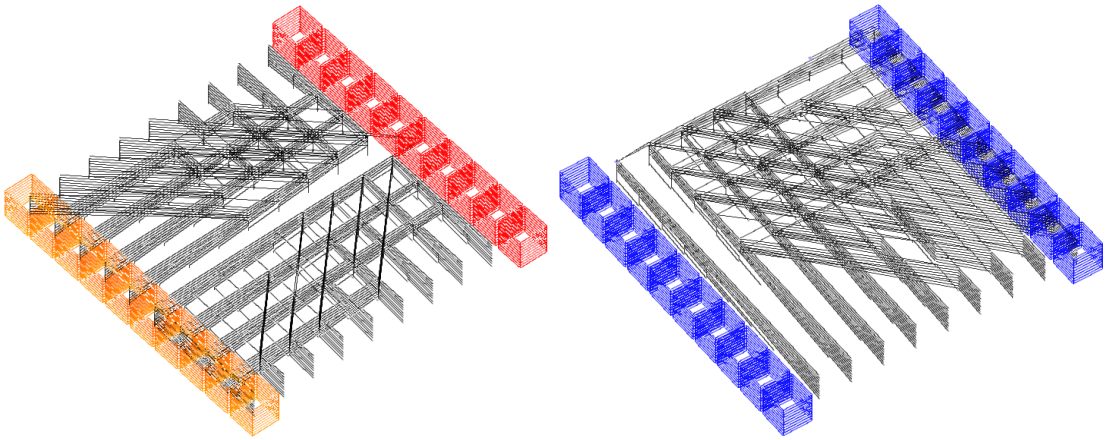

Figure 5.11: Two different topologies developed by axon connections in this model. A topology-based evolutionary fitness rewards symmetry in the left individual and a high clustering coefficient in the right. Inputs and outputs are represented by colored cubes; intermediate neurons are not represented, but are placed evenly throughout the space.

This is a very dense placement compared to the eye patterning experiment, and many more connections formed. Branching was also prevalent, with $N_{branch} = 10$ and $N_{synapse} = 5$. To validate the ability of this model and specific neural placement to create different network topologies, evolutionary runs were completed with topology-based fitnesses. The resultant network was evaluated for symmetry in one evolution and clustering coefficient in another. Representative topologies from these experiment are shown in Figure 5.11.

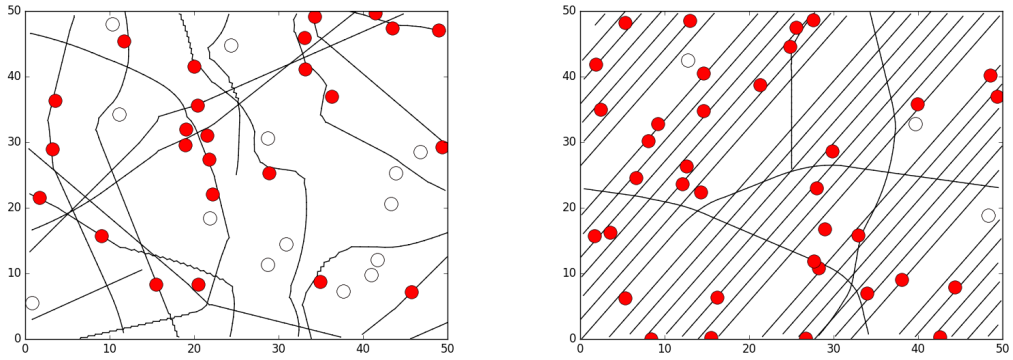

Figure 5.12: The path and food consumption of the robot. Consumed food is filled red; ignored food is empty. On the left, the robot displays a use of its sensors, turning towards food when it is within its radar radius. On the right, the robot uses an evolved strategy which ignores the sensors, tracing a straight path through the toroid. This static strategy is also effective for foraging, collecting more food than the responsive strategy.

The topologies created by the evolved AGRN proved flexible to different evolutionary fitnesses, but neural activation was not considered during these topology-based evolution fitness metrics. The robot environment, on the other hand, was designed with neural

activation in mind. The spatial representation of the input and output information was intended to involve the neural activity of the input layer in the axon growth cone decisions. While this was the case for some involved individuals, many of the solutions preferred by evolution ignored the input neural activity. Such results are shown in Figure 5.12, where a static foraging strategy is used to collect food by moving in a straight line.

Due to the efficacy of a static foraging strategy, evolution was not required to find complex AGRNs capable of forming responsive architectures. This highlights one of the difficulties of using evolutionary algorithms; simple but effective strategies may take precedence over more complex strategies, as individuals that employ these simple strategies can dominate selection from early generations. There are means of addressing this problem in evolutionary algorithms, such as novelty, speciation, or open-ended search [LS08]. GRNEAT, used in this work to evolve the AGRN, already includes automatic speciation, but further could be done to ensure the survival of complex but underdeveloped strategies in the evolutionary population.

5.5 Conclusion

In this chapter, we focused on the formation of neural architecture. AGRNs controlled axons to develop in a 3D space, following morphogen gradients also controlled by AGRNs. Glial cells were shown to play an important role in the diffusion of morphogens, guiding the axon growth cones. We showed that this can allow for activity-dependent development, a variety of neural topologies, and can be used for artificial tasks.

The networks in this chapter were limited in their capacity to learn, however, by using constant synaptic weights throughout. In the brain, learning requires both architectural changes and synaptic tuning. The use of synaptic formation and pruning alone for learning is an expensive and imprecise method; synapses must have adjustable weights to attain the precision required in most tasks.

In the next chapter, we present two experiments which approach weight-based learning from different points, looking specifically at how learning signals can be modulated. These experiments, like many others in ANN research, use static topologies. In this chapter, we've demonstrated that architectural designs can be made automatically and can rely on neural activity to shape the neural network over its lifetime. In the next, we show that weight-based learning is effective for a variety of tasks, but it is important to remember the architectural optimization done in these works. For the most part, it is an offline process based on evaluating multiple static architectures not mentioned in the presentation of the ANN's performance. While developmental neural networks present many difficulties shown in this chapter, the model presented here could replace this expensive and difficult architecture engineering process.

Chapter 6

Evolving learning methods

In the brain, learning can happen in a variety of ways. New neurons can be generated or old ones die, dendritic paths can extend or be pruned, and synapses can be created, be destroyed, or change in form and efficacy. In this chapter, we focus on this last process, the change in synaptic efficacy between one neuron and another. Specifically, we look at how the process of changing synaptic efficacy is guided towards a specific goal. In the brain, this happens through a process of neuromodulation, where specific chemical signals reinforce or discourage recent neural activity.

In *C. elegans*, a nematode worm, the neuromodulator octopamine is released when food reserves fall; this chemical binds to specific target neurons, modifies their excitability, and changes their synapses. The behavior of the worm is affected by this change, as turning is inhibited and the forward motion pattern excited. This neuromodulatory response therefore alters the worm's neural circuitry to exhibit a "roam" behavior.

When food is found, a second neuromodulator, dopamine, is released. This checmical signal binds to target neurons, turning off the octopamine receptors and restoring the previous state of the neural circuit. Turning is no longer inhibited and the forward motion pattern is less excited, allowing the worm to stay at the food source and "graze" [SL15]. This example demonstrates how two chemical signals, octopamine and dopamine, provide the motivator and reward for brains as simple as that of the worm.

In mammals, the role of dopamine is more complex than in the worm. Instead of being released when food is found, dopamine is released when the reward received exceeds the expectation of reward, among other instances. Learning is therefore the result of a prediction made in the brain; neural activity of prediction in one part of the brain therefore influences the reward given to and learning performed in other parts of the brain. Dopaminergic neurons are capable of encoding three functional states: "as expected", "better than expected", or "worse than expected", and are capable of processing both positive and negative learning strategies [PB15].

Teaching signals in the brain use a mixture of chemical signaling, called volume transmission (VT), and synaptic, or wired, transmission (WT) [Agn+10]. VT, such as the release of dopamine, sends a neurotrasmitter throughout an area, contacting many neurons and synapses at once. WT uses axons and dendrites to send signals from one neuron to another. Using pure chemical signaling over the entire brain would be too slow and critical timing information would be lost. However, using wired connections for all teaching signals would require an enormous increase in connections, as each synapse would need an additional teaching connection [SL15].

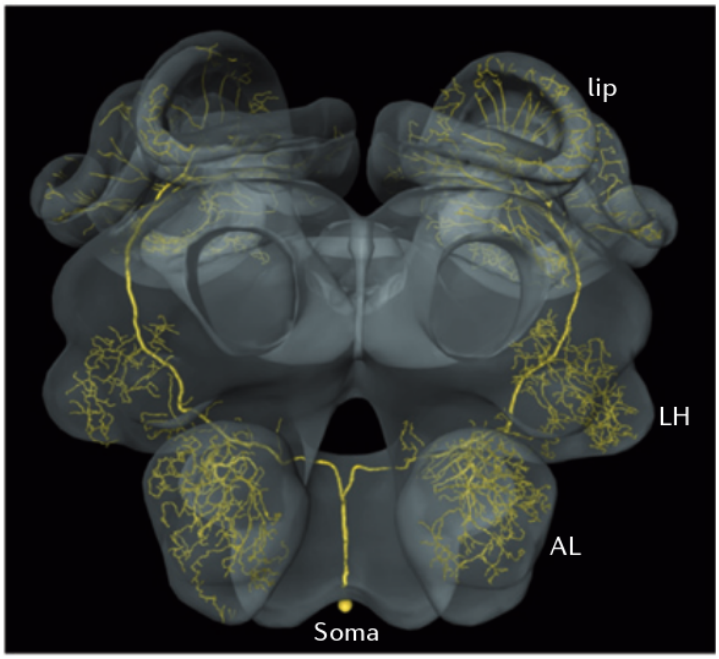

Figure 6.1: A teaching neuron in the honeybee brain, from [Men12]. This single neuron reaches many sections of the brain and encodes reward in olfactory learning. Similar to dopamine neurons in mammals, this neuron operators on prediction: it decreases its response to an expected reward but increases its response to an unexpected reward.

Instead, certain teaching neurons have axons which form massive terminal arbors. An example, from the honeybee, is shown in Figure 6.1. In mammals, these are dopamine neurons; the axon branches release dopamine when they fire. This allows for a simultaneous teaching signal to reach many synapses through the brain without requiring explicit connection to each one. The signal then reinforces recent activity in these varied sections, which leads to learning [SL15].

Volume transmission is therefore an important aspect of learning [Agn+06], where nearby neurons receive the same chemical signal, but physically distant neurons receive different signals. In ANN learning, this concept is mostly unexplored. Synapses are trained individually based on error functions specific to their existing weights, and not

based on their location in an ANN.

The importance of neuromodulation in learning is well-known. A review of neuromolulatory factors, their impact on learning, and their relationship to reinforcement learning theory is given in [Doy08]. However, there has been little exploration of the application of neuromodulation to supervised learning in artificial neural networks, and to our knowledge, no existing application to deep neural networks. Much of the existing work using ANNs has focused on reward-based modulation for unsupervised Hebbian learning to allow for semi-supervised learning. In the case of Spike-timing Dependent Plasticity, reward-modulated learning has been used in a variety of tasks [FG16]. In [FF07], a spiking neural network is trained using neuromodulated STDP to elicit specific spike train. Similarly, in [Izh07], a population response of neurons firing in a specific group is demonstrated, using a model of neuromodulation based on a chemical dopamine signal. A non-spiking example of neuromodulated Hebbian learning is in [VC17], where diffusion-based neuromodulation is used to eliminate catastrophic forgetting in ANNs. The ANNs used in this work are shallow networks trained using Hebbian learning to perform multiple tasks.

Artificial neuromodulation has been studied in contexts other than ANNs. Dopaminergic neurons were the inspiration for a study of neuromodulation in reinforcement learning, which improved Q learning on a Markov decision problem [SD03]. In [Har+13], a robot agent learns to cover an area in a reinforcement learning scheme. The agent is trained by SARSA, which is itself controlled by an evolved neuromodulator. This work was extended in [CH15], where evolved neuromodulation agents facilitate multi-task learning. This work also uses SARSA as the agent. [SSR18] provides a review of evolved neuromodulation in a variety of contexts.

The task of artificial neuromodulation can be seen as part of the "learning to learn" or "meta-learning" problem of optimizing the learning process. In [Doy02], neuromodulation is studied in the context of the computational theory of metalearning. Active learning optimization of this nature has been the topic of much study, such as in [And+16], where a secondary ANN is introduced to optimize the learning of the primary network. MAML is a recent meta-learning algorithm which increases adaptation of deep neural networks to new problems [FAL17].

The field of artificial neuromodulation is in general very new, especially when applied to ANNs. In [Har+13], a robot agent learns to cover an area in a reinforcement learning scheme. The agent is trained by SARSA, which is itself controlled by an evolved neuromodulator. This work was extended in [CH15], where evolved neuromodulation agents facilitate multi-task learning. This work also uses SARSA as the agent.

In [VC17], diffusion-based neuromodulation is used to eliminate catastrophic forgetting in ANNs. The ANNs used in this work are small agents trained using Hebbian

learning to perform multiple tasks. The use of neuromodulation in Hebbian learning, specifically in STDP, is the most advanced topic in artificial neuromodulation and is covered in more detail in section 6.1.

Our focus in this chapter is to evolve the rules of neuromodulation, using two existing learning methods, STDP and SGD, as a base and improving upon these methods with evolution. In both of these works, the physical aspects of the neural network are central to the design of the neuromodulation controller. SGD is traditionally a global process, with a static learning rate used for all neurons. While methods like [KB14] offer dynamic learning rates, these rates are applied throughout the network. In the brain, the location of a neuron highly impacts its learning. Neurons closer to dopaminergic sites will receive a signal before others, and clusters of neurons physically concentrated together share chemical signals, even if they have disparate connections. Some neuromodulatory signals are wide-ranged, effecting entire sections of neurons together, while others are highly localized, modulating learning in only a few neurons.

The neuromodulation controllers were also designed to be responsive, changing over time and in response to neural activity. In the brain, this is an important aspect of learning, where neuromodulation is itself controlled by activity in the brain. In [PB15], the role of synaptic plasticity of dopaminergic neurons is examined, showing that external pressures such as stress can change the behavior of these neurons. The response of a synapse to dopamine changes also over time and depends on the state of the neuron.

The goal of the evolved controllers is therefore to use information about the neurons, i.e. their placement, activity, and learning history, to modify learning as it happens. As evolution requires a means of comparison between individuals, the quality of learning must be reduced to a fitness value for selection. In these works, we have focused on the efficiency of learning by allowing all individuals to learn for the same amount of time and using their state at the end of this learning period as the evolutionary fitness. Many other metrics of learning exist, such as the ability to generalize to similar cases based on learning a specific case. However, both of these works are novel forays into the artificial neuromodulation, itself a nascent topic, and learning efficiency is the primary objective of much of the research concerning artificial learning.

In the following sections, we present these two works on neuromodulation. In the first section, based on [Wil+17], the parameters of a dopamine-based STDP method are evolved. These parameters control the propagation of the dopaminergic signal and its use in STDP, modifying learning in an artificial creature which is learning to swim in an aquatic environment. In the second section, AGRNs are evolved to modify the learning at each layer of a deep neural network, changing the parameters of SGD during learning. This is shown to improve upon standard SGD with optimal parameters and matches the Adam optimizer on CIFAR10, an image recognition dataset. While these sections use different

learning algorithms and evolutionary methods, common principles of neuromodulation are explored in both.

6.1 Reward-Modulated Spike-Timing Dependent Plasticity

In this work, we model a physical dopamine signal as it travels through a 3D space. This dopamine signal modulates learning by directly affecting Spike-Timing Dependent Plasticity (STDP). Reward from the environment triggers a release of dopamine, allowing for reinforcement learning using STDP, which is an unsupervised learning method. We demonstrate the dopamine model's capabilities on an existing demonstration problem, instrumental coding, and on a novel animat locomotion problem designed for RL with spiking ANNs. For the animat locomotion problem, we use evolution to determine the parameters of the dopamine model, allowing evolution to determine how the reward signal influences learning.

Reward-modulated STDP (R-STDP) is a relatively recent approach for learning in ANNs, with much of the background work on this topic focused on biological modeling. [LPM08], for example, provides a theoretical framework for R-STDP and demonstrates this framework on a previous biological experiment. Many such models are discussed in a large review on neuromodulation in STDP [FG16], which offers a framework for "three-factor" learning rules. The three factors are presynaptic activity, postsynaptic activity, and neuromodulation. This extends the classic Hebbian learning perspective of STDP, which only focuses on the first two factors.

An early use of R-STDP for ANNs is found in [FF07]. In this work, a SNN was trained to elicit specific spike trains and population responses using a reward signal. Similarly, in [Izh07], a population response of neurons firing in a specific group was demonstrated. This work influences the one presented here, as the reward signal is modeled as a chemical dopamine signal with degrades over time. This is shown to solve the distal reward problem, where reward is only given a certain amount of time after the rewarded action.

More recently, [Moz+18b] demonstrated the capability of R-STDP on object recognition in images. This work used a convolutional SNN and rewarded neuron firing events in the last layer when they correctly matched an object category, and also used anti-STDP when the fired neurons were incorrect. R-STDP was also combined with standard STDP in [Moz+18a], where early layers of a deep convolutionary SNN were trained using standard STDP and later layers with R-STDP. This achieved similar results on the MNIST digit recognition benchmark as [Khe+16], which is a fully unsupervised approach using STDP. Both of these results are competitive with state of the art deep learning on the

MNIST benchmark, reaching 97.2% and 98.4% accuracy, respectively.

R-STDP has mostly been investigated using data-based tasks, i.e. signal recreation and classification. One of the novel contributions of this work is the learning task used. We present a reinforcement learning problem of animat locomotion which is specialized for SNNs. Animat locomotion tasks consist of agents learning a strategy to propel a virtual creature or robot forward, or along a specific direction. This problem type has been used to study many artificial agents, such as AGRNs in [Joa+16].

The animat locomotion problem in this work was designed specifically for RL and SNNs. It was also designed to encourage the study of embodied cognition, the theory that the body plays an integral role in cognition [Sha10] , as the placement of neurons inside the animat is an important component of its learning strategy. In this problem, the reward is given at a specific central location, requiring the learning strategy to distribute the reward to the rest of the network.

In the next section, we present the STDP model and the basis for its design. This model considers other possible STDP models, such as R-STDP, and uses parameters to differentiate between these models. We then test the new model on a simple problem, the instrumental conditioning problem from [Izh07]. Finally, we present the animat locomotion problem and the results of the evolution of the STDP method parameters on this problem.

6.1.1 Neuron and learning models

In this work, the Izhikevich SNN model is used [Izh03], as it can exhibit a variety of natural behaviors. In this model, each neuron n has a membrane potential v_n and a membrane recovery variable u_n. v_n is increased by input I_n either from external sources or from other neurons:

$$v_i(t=0) = v_R$$
$$v_i(t+1) = 0.04v_i(t)^2 + 5v_i(t) + 140 - u_i(t) + I_i(t)$$
$$u_i(t+1) = a(bv_i(t) - u_i(t))$$

The membrane potential is increased from a resting potential v_R until reaching a threshold v_T, at which point the neuron spikes, resetting v to a membrane potential c and updating u. A signal from the spiking neuron then propagates to postsynaptic neurons, increasing their synaptic input I_j by the weight s from the spiking neuron n_i to the postsynaptic neuron n_j:

$$v_i(t+1) = c \quad ; \quad u_i(t+1) = u_i(t) + d$$
$$I_j(t+1) = I_j(t+1) + s_{i,j}$$

v_T	30.0	v_R	-65.0
a	(0.02, 0.1)	b	0.2
c	-65.0	d	(8.0, 2.0)
s_e	1.0	s_i	-1.0
A_+	1.0	A_-	1.5

Table 6.1: Neural parameters from [Izh07]. The two values of a and d correspond to excitatory and inhibitory neurons, respectively.

Synapses are modeled as a matrix of real valued weights; excitatory synapses are initialized to s_e and bound between $[0.0, 4.0]$ during STDP training , and inhibitory synapses are held constant at s_i.

STDP modifies the synaptic weights of an SNN based on the fire timing of the synapses's respective neural endpoints. If a presynaptic neuron n_i fires and then a post-synaptic neuron n_j fires shortly after, the synaptic weight $s_{i,j}$ is increased. If the firing order is reversed, $s_{i,j}$ is decreased. Using this Hebbian learning scheme, hidden neurons are tuned to features in the input layer, as captured visually in [DC15]. Many STDP schemes use a neural competition rule, such as in [Khe+16], where the first neuron in any layer to fire is the only one trained for a given input sequence. In this work, no fixed competition rule is used; instead, it is through the distribution of reward that STDP applies variably to competing neurons. The STDP update rule from [Izh07] is used:

$$\Delta s_{i,j} = A_+ e^{-(t_j - t_i)} \delta(t - t_i), \quad if \quad t_j - t_i > 0 \tag{6.1}$$

$$\Delta s_{i,j} = -A_- e^{-(t_j - t_i)} \delta(t - t_i), \quad if \quad t_j - t_i < 0 \tag{6.2}$$

where t_i indicates the most recent spike time of neuron n_i, A_+ and A_- are STDP learning parameters, and $\delta(t)$ is the Dirac delta function. Euler integration with a 1ms time step is used for computation.

6.1.2 Neuromodulation reward model

To explore neuromodulation using STDP, we expand upon and parameterize two existing semi-supervised learning methods. Both methods build on STDP using an artifical dopamine concentration, which is a function of a global reward signal, rw, provided to the controller. The dopamine concentration is calculated for each neuron, dependent on its position. Neurons in this work are positioned in a 3D space, with topology determined per task. A dopamine signal starts at the center of mass of the network and propagates at a speed based on a delay parameter, p_{dd}. The concentration of this signal also attenuates

as it travels based on the parameter p_{dat}. Lastly, a fraction of dopamine concentration is absorbed each timestep based on p_{dab}:

$$dist = \frac{\sqrt{\sum_{d=0}^{2}(n_i[d] - com[d])^2}}{dist_{max}}$$
$$dx = p_{dd} size(h) dist$$
$$r_i = rw[\lfloor dx \rfloor] + (dx - \lfloor dx \rfloor)(rw[\lfloor dx \rfloor + 1] - rw[\lfloor dx \rfloor])$$
$$D_i(t+1) = (1.0 - p_{dab})D_i(t) + e^{-p_{dat} * dist} r_i$$

where com is the network's center of mass, rw is a fixed-size array of the most recent reward values calculated at a fixed interval, d indicates the positional dimension, and $dist_{max}$ is the maximum radius of the network. The dopamine concentration at each neuron is therefore a scaled version of the linear interpolation of the delayed reward based on the propagation delay, p_{dd}, with accumulation over time based on p_{da}.

The first reward method proposed is the direct input of reward as an activation mechanism, termed Induced Firing STDP (IF-STDP). This method is based on dopaminergic activation in biologic brains [PB15] and has similarities to the teaching neurons of semi-supervised methods such as ReSuMe [Pon05]. IF-STDP functions by directly activating neurons based on the extracellular dopamine at their position:

$$I_j = I_j + p_{rs} D_j$$

where p_{rs} is a reward signal coefficient parameter. This is intended to induce firing based on a reward signal, which will then further strengthen the synapse between activated neurons through basic STDP.

The second reward method is the modulation of STDP using the dopamine concentration, Dopamine Modulated STDP (DM-STDP). The synaptic weight change of STDP is modified by the dopamine concentration of the involved neurons n_i and n_j, based on p_{df}, a dopamine factor parameter:

$$D_{i,j} = \frac{D_i + D_j}{2.0}$$
$$\Delta s_{i,j} = (0.01(1 - p_{df}) + p_{pd} D_{i,j}) \Delta s_{i,j}$$

By using different values of the parameters p_{rs}, p_{df}, p_{dd}, p_{dat}, and p_{dab}, different STDP modulation methods can be recreated. Classic STDP is achieved when no reward induced firing or modulation take place, hence p_{rs} and p_{df} must both be 0. R-STDP, defined in [FG16] as "gated Hebbian learning", modulates STDP based on instantaneous reward only, therefore the absorption rate parameter p_{dab} is 1.0; all dopamine is absorbed at each

method	p_{rs}	p_{df}	p_{dd}	p_{dat}	p_{dab}
STDP	0.0	0.0	0.0	0.0	1.0
R-STDP	0.0	**1.0**	0.0	0.0	1.0
DA-STDP	0.0	**1.0**	0.0	0.0	**0.001**
IF-STDP	**1.0**	0.0	**1.0**	**0.1**	**0.001**
DM-STDP	0.0	**1.0**	**1.0**	**0.1**	**0.001**
IFDM-STDP	**1.0**	**1.0**	**1.0**	**0.1**	**0.001**

Table 6.2: Parameterization of different reward modulation methods. Bold values are tunable within the method; the values for these parameters were chosen for the instrumental conditioning experiment defined in subsection 6.1.3

timestep. The signal does not travel over distance or attenuate over time, so both p_{dd} and p_{dat} are 0.0. This is the same as DA-STDP [Izh07], which added the novel concept of dopamine absorption over time, here reflected in p_{dab}. These parameters are given fully in Table 6.2.

The new methods proposed in this work, IF-STDP, DM-STDP, and their combination IFDM-STDP, use a chemical dopamine signal that propagates through the physical space of the network, attenuating as it travels and being absorbed over time. While the underlying effects of both methods are not novel, their use of such a dopamine signal is new. We therefore present both a comparison of these new methods with previous ones and an exploration of the parameter space that defines the dopamine signal and its use.

6.1.3 Instrumental conditioning

First, to display simply the functionality of each model, the instrumental conditioning experiment from [Izh07] is reproduced. This type of experiment is common in SNN literature, as it focuses on eliciting a specific spiking response from the network and not on any application of the output.

For this experiment, a network with N_{in} input, N_h neurons, and N_{out} output neurons was used. This input, hidden, output designation does not indicate topology as is common in other ANN literature, but rather the use of the neuron. Input neurons receive a stimulus current of ϕ_s mV every ϕ_i ms.

The topology of the network is random: each neuron has a ρ_c chance of connecting with another neuron and a ρ_{in} chance of being an inhibitory neuron. Therefore, in this experiment, the network was composed of 1000 total neurons, 800 being excitatory and 200 being inhibitory, with 100 synaptic connections each. The neurons were placed randomly in a 3D space following a uniform distribution over $[-1.0, 1.0]$ in each dimension. Parameters for this experiment can be found in Table 6.3.

N_{in}	50	N_h	850	N_{out}	100
ρ_c	0.1	ρ_{in}	0.2	rw	0.1
ϕ_g	1	ϕ_i	1000	ϕ_s	15

Table 6.3: Parameters used in the instrumental conditioning experiment. ϕ_g indicates the number of distinct input groups, which in this experiment was 1, meaning the inputs were not subdivided.

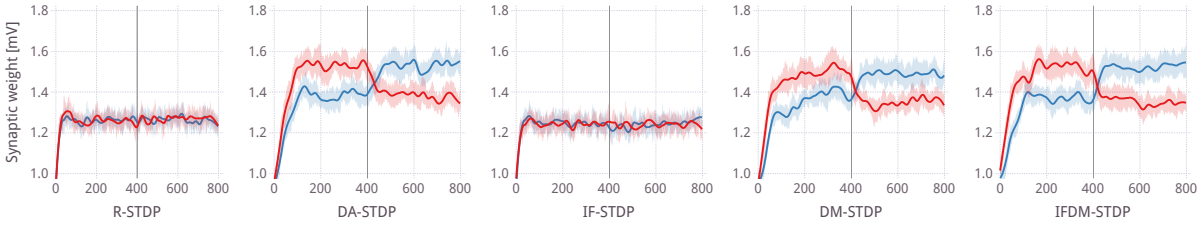

Figure 6.2: The synaptic weights between the input and the A (red) and B, (blue) groups in the instrumental conditioning experiment. By rewarding firing from group A during the first 400 episodes, the weights between the input and A should increase, as with group B for the second 400 episodes. Ribbons represent the standard deviation over 20 trials.

The output neurons were split into two exclusive groups of 50 neurons each, A and B. For a short period, 20ms, after each input stimulus ϕ, the number of spikes in each group was recorded as $|A|$ and $|B|$. For the first 400 stimulus intervals, a constant reward rw was provided following the stimulus if $|A| > |B|$, and when $|B| > |A|$ for the second 400 stimulus intervals. The reward was delayed by a maximum of 1s and the stimulus intervals were 10s apart.

This task is difficult because the reward is delayed and is therefore challenging to correlate with the firing events that caused it. Furthermore, the goal changes after 400 intervals, requiring the weights between the input and A to decay during this second interval.

In the instrumental conditioning experiment (Figure 6.2), DA-STDP displayed its capabilities as in [Izh07]. While the delayed reward in this problem is difficult to properly assign, by introducing an absorption rate p_{dab}, the dopamine concentration is able to last until further firing episodes between the inputs and the output group occurred, triggering STDP. Over many cycles of stimulation and reward, the events become correlated and the synaptic weights between the inputs and the rewarded group increased.

R-STDP is not able to solve this problem due to the instantaneous gating of STDP it performs. The random delay does not allow it to correlate the reward with the proper firing events, so the weights from the inputs to both groups are increased. This is also seen in IF-STDP, where the induced firing alone is not enough to influence the weights. However, when combined with DM-STDP, some improvement is seen in IFDM-STDP; the gap between the weights widens and is reached faster. DM-STDP is a reduction in total

weight change from DA-STDP, due to the dopamine attenuation p_{dat}, and the induced firing from IF-STDP appears to match DM-STDP with DA-STDP for total weight change.

While this task is challenging, the different STDP methods either fail or succeed at the task, and it is difficult to discern their quality. Furthermore, the application is only abstract; the different output groups can be considered different motor responses, but it is not clear what the output firing corresponds to or what the delayed response should represent. The random assignment of neural positions also reflects the abstract nature of this experiment; to fully explore the impact of physical parameters in an embodied network, specific topologies must be considered. To address these issues, we propose the following benchmark problem, animat locomotion, and present an exploration of the parameterized methods using evolutionary search.

6.1.4 Aquatic Locomotion Problem

For the animate locomotion problem, we create virtual creatures, animats, composed to linked cells which propelled themselves in an aquatic environment by contracting its cells in coordinated motion. This allows for the simplistic output of synaptic firing, a binary event, to be used for control in a complex environment. Not only does each animat have to learn to coordinate firing events to create large-scale body movement, it has to do so in an advantageous way based on the fluid dynamics present and its morphology.

An SNN with specified STDP parameters is placed inside an animat with all input and hidden cells located at the animat's center of mass. Output neurons are placed evenly throughout the morphology and control the contraction in clusters of cells. The cells are connected to an output neuron based on proximity; each cell is connected to its nearest output neuron. Upon firing, an output neuron causes its connected cluster to contract, leading to deformation of the animat, allowing for locomotion. Input neurons are separated into ϕ_g groups and given a stimulus signal ϕ_s every ϕ_i ms, with the chosen input group rotating each stimulus.

Two static morphologies were used in this experiment to diversify the neural topologies and movement strategies. These morphologies are a four-legged octopus (quadropus) and a stingray, shown in Figure 6.3. These morphologies were designed by hand.

Reward was initially provided to the animat based on the movement of its center of mass *com*. While this constant reward signal is desirable in many reinforcement learning settings, we found that discrete reward events were more suitable in this problem. The reward was therefore the percentage increase of animat velocity whenever the velocity eclipsed its previous maximum, v_{max}. To continue to reward velocity increases over the life of the animat, v_{max} decayed exponentially.

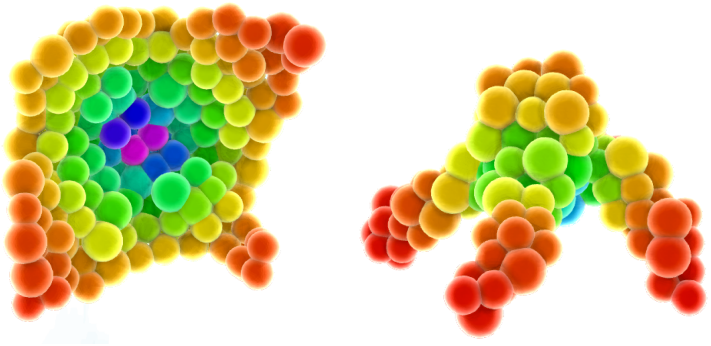

Figure 6.3: Top: The stingray morphology and bottom: the quadropus morphology. Coloring is based on dopamine distribution. The quadropus has a fourth arm which is obscured in this image

$$dist(t) = \sqrt{\sum_{d=0}^{2}(com(t)[d] - com(t=0)[d])^2}$$
$$v(t) = dist(t) - dist(t-1)$$
$$v_{max}(t) = 0.99 v_{max}(t-1)$$
$$rew(t) = rw * max(0.0, (v(t) - v_{max}(t))/v_{max}(t))$$

The goal for STDP was to therefore correlate input stimulus firing with output behavior that increased velocity, similar to the instrumental conditioning experiment. Unlike that experiment, however, the mapping between output firing and reward was highly complex, as the animat had to continuously find new output firing patterns that increased its velocity.

N_{in}	440	N_h	570	N_{out}	(87, 74)
ρ_c	0.13	ρ_{in}	0.20	rw	10.0
ϕ_g	6	ϕ_i	90	ϕ_s	45
T_{cont}	20	c_{cont}	0.9	F_{fluid}	0.0005

Table 6.4: Parameters used in the aquatic locomotion experiment, where the two values for N_{out} correspond to the quadropus and stingray morphologies, respectively

We based our experiments on the Artificial Life platform MecaCell in which we created an aquatic environment [DCD15]. The organism was composed of several tightly packed cells linked with elastic bonds, using a mass-spring-damper system for modeling both the adhesions and the collisions. The bonds were created between neighbouring cells at the beginning of the simulation and were then set to be unbreakable. In order to obtain the creatures shapes, we used 3D meshes which we filled with cells.

Each cell contracted by changing its desired radius to c_{cont} times its original radius, which amounts to shortening the rest length of the collision springs and thus pulling on connected bonds. After a set duration T_{cont}, the cell reset to its original spring length. If an output neuron fired for a previously contracted cell, the cell remained contracted for another T_{cont} ms.

The problem is therefore suitable for reinforcement learning with SNNs. Binary spiking events cause contractions in continuous time, removing the need to design an output encoding scheme. Two reward goals have been defined, center of mass movement and increased velocity, which could be selected based on experiment. Neuron location is important to the problem; input and hidden neurons are located around the center of reward signal distribution, whereas output neuron location determines the cell clusters and movement ability of the animat. To encourage the use of this problem, we have made the source code available[1].

6.1.5 Evolution of neuromodulation method

We now present an exploration of learning methods using this benchmark. By evolving the proposed STDP parameters, p_{rs}, p_{df}, p_{dd}, p_{dat}, and p_{dab}, different methods of learning are used by the animats. We use CMA-ES to evolve these parameters, within the ranges given in Table 6.5. The evolutionary fitness was the cumulative sum of the distance traveled away from the center of mass at each timestep.

$$dist(t) = \sqrt{\sum_{d=0}^{2}(com(t)[d] - com(t=0)[d])^2}$$

$$fitness = \sum_{t} dist(t)$$

R-STDP, DA-STDP, and IFDM-STDP were optimized independently by fixing the non-tunable parameters of these methods and optimizing the others (see Table 6.2 for the tunable parameters of each method). The population size λ for CMA-ES was chosen as a function of the parameter space size P: $\lambda = 4 + \lfloor 3log(P) \rfloor$.

CMA-ES was run for 50 generations and 20 independent trials were conducted for statistical testing. All parameters were optimized within $[0.0, 1.0]$ and then scaled to their respective ranges for fitness evaluation. Uniform random values were used to initialize CMA-ES and the step size for all parameters was 0.5.

[1] https://github.com/d9w/lala

	p_{rs}	p_{df}	p_{dd}	p_{dat}	p_{dab}
min	0.0	0.0	0.0	0.0	0.0
max	2.0	1.0	1.0	10.0	1.0

Table 6.5: Reward parameters ranges for CMA-ES

6.1.6 Evolution results

By evolving the parameters using CMA-ES, significant improvement in the total distance traveled was achieved, especially for the quadropus morphology, as seen in Figure 6.4. Neither R-STDP nor DA-STDP reached the distances that IFDM-STDP was able to, indicating the importance of a physically situated dopamine concentration for this problem. As the network topology is directly representative of the animat morphology, with output neurons positioned throughout the animat, having a dopamine signal with delayed propagation and attenuation appears to have been very important.

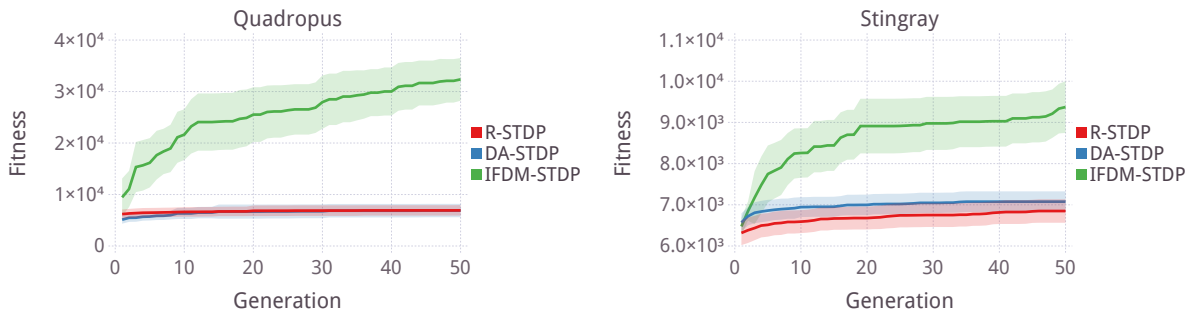

Figure 6.4: CMA-ES optimization of the tuneable parameters of the different STDP strategies for both morphologies. Ribbons indicate standard deviation over 20 trials.

To understand which parameters are responsible for the success of IFDM-STDP, the parameters of the best individuals are shown in Figure 6.5 as the normalized parameter values, before they are set to the parameter ranges in Table 6.5. The values of the parameters of single best individual are also show in Table 6.6. Also shown in this table are the best individuals from the evolution of R-STDP and DA-STDP.

Some parameters confer a consistent benefit. The dopamine factor p_{df} is high for all top IFDM-STDP individuals, as is the dopamine decay parameter p_{dd}. First, this that STDP utilized the dopamine concentration to modify weights. That alone is not sufficient, though, as demonstrated by R-STDP's performance. The usage of the p_{dd} parameter means that delaying the reward signal to the distal parts of the animat morphology was beneficial. As contraction events near the center of the animat often caused movement in the distal parts of the morphology, but not vice versa, this delay can be seen as a way

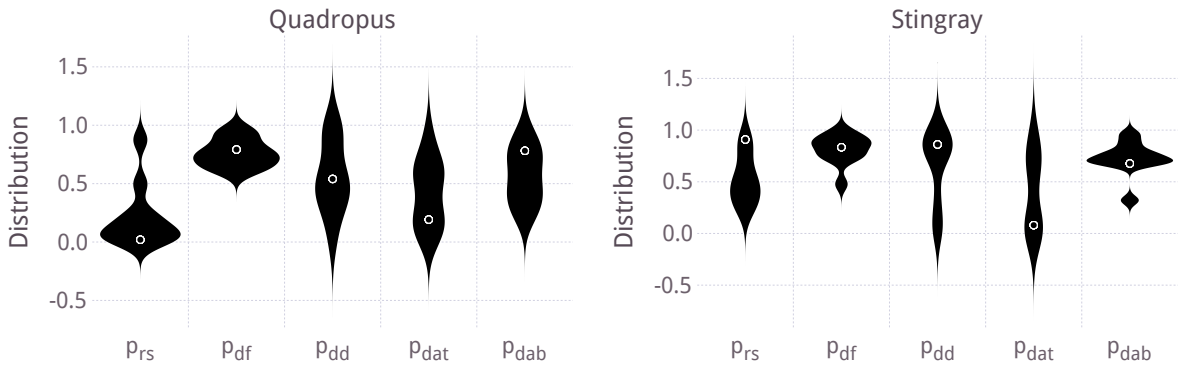

Figure 6.5: Distribution of the reward parameters of the 10 best individuals for both morphologies. Circles show the best single individual. p_{rs} is scaled by 0.5 and p_{dat} by 0.1.

to properly correlate reward with firing events in the distal regions and not with motion caused by central contractions.

Quadropus

method	p_{rs}	p_{df}	p_{dd}	p_{dat}	p_{dab}
R-STDP	0.0	**0.80**	0.0	0.0	1.0
DA-STDP	0.0	**0.68**	0.0	0.0	**0.95**
IFDM-STDP	**0.04**	**0.79**	**0.54**	**1.92**	**0.78**

Stingray

method	p_{rs}	p_{df}	p_{dd}	p_{dat}	p_{dab}
R-STDP	0.0	**0.69**	0.0	0.0	1.0
DA-STDP	0.0	**0.08**	0.0	0.0	**0.91**
IFDM-STDP	**1.82**	**0.83**	**0.86**	**0.80**	**0.67**

Table 6.6: Evolved parameters for each method on both morphologies. Bold values indicate the best evolved value, while non-bold values were held constant.

Other parameters are not consistent between the morphologies. The reward signal factor p_{rs} was not used by most top quadropus individuals, but was by top stingray individuals. One possible explanation for this is that the quadropus is more rigid than the stingray, and excess firing can more easily have a negative effect on the movement pattern of the quadropus than the stingray. Neither morphology had a consistent strategy concerning p_{dat} either; while both best individuals had relatively low attenuation parameters, the distribution over the top individuals is wide.

6.1.7 Summary of Reward-Modulated STDP

Given the increase in evolutionary fitness by modifying the method parameters, it is clear that some of the proposed reward mechanisms provide benefits in this problem. These benefits have been explored in the context of this work, but future work is necessary to continue to assess their impact in different settings. Specifically, these methods should be assessed in other problems in which the neural network is situated within the controlled object, giving each neuron a position in space.

Throughout this work, it was necessary to decide and assume certain factors. The dopamine signal for both experiments originated at the network's center of mass, but biologic dopamine signals have multiple origins and do not diffuse equally throughout the brain. Whether or not this is the product of biologic design or a feature of learning can be explored.

The learning feature of delayed reward information, here found in both p_{dab} and p_{dd}, is one that is being explored in artificial learning. The abstraction of dopamine delay can be taken from this model and used even in networks that don't have neural positioning, as long as some delay coordinate, such as layer depth, is provided. This can serve many training methods on problems with temporal reward, especially in the presence of a delay between the action and the reward.

6.2 Neuromodulation of learning parameters in deep neural networks

As described in section 2.3, a common learning method used for ANNs is stochastic gradient descent. In this method, the synaptic weights and neuron biases, θ, are optimized according to a loss function, Q. In classic SGD, a hyperparameter η is used to determine the speed with which weights change based on the loss function. This method can be improved with the addition of momentum [Nes83], which changes the weight update based on the previous weight update. An additional hyper-parameter, α, is then used to determine the impact of momentum on the final update:

$$\Delta\theta^{(t+1)} \leftarrow \alpha\Delta\theta^{(t)} - \eta\nabla Q_i(\theta^{(t)}) \tag{6.3}$$

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + \Delta\theta^{(t+1)} \tag{6.4}$$

In contemporary deep learning, there is a variety of gradient descent approaches to choose from. Adagrad [DHS11] implements an adaptive learning rate and is often used for sparse datasets. Adadelta [Zei12] and RMSprop [TH18] were both suggested to solve a

problem of quickly diminishing learning rates in Adagrad and are now popular choices for timeseries tasks. Adam [KB14] is one of the most widely used optimizers for classification tasks, where past gradients are stored in a variable m, and past squared gradients are stored in a variable v. Two hyper-parameters, β_1 and β_2, control the update rate of m and v, respectively. m and v are then used to update the weights, instead of using the gradient directly. This update has a learning rate hyper-parameter, η, as well as a "fuzzing factor" hyper-parameter ϵ which controls the ratio between m and v in the final update:

$$m_\theta^{(t+1)} \leftarrow \beta_1 m_\theta^{(t)} + (1-\beta_1)\nabla Q_i(\theta^{(t)}) \tag{6.5}$$

$$v_\theta^{(t+1)} \leftarrow \beta_2 v_\theta^{(t)} + (1-\beta_2)(\nabla Q_i(\theta^{(t)}))^2 \tag{6.6}$$

$$\hat{m}_\theta = \frac{m_\theta^{(t+1)}}{1-\beta_1^t} \tag{6.7}$$

$$\hat{v}_\theta = \frac{v_\theta^{(t+1)}}{1-\beta_2^t} \tag{6.8}$$

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta} + \epsilon} \tag{6.9}$$

These methods, as well as others, are all the result of empirical study on specific problems. An overview of their different benefits and weaknesses is presented in [Rud16], and the choice of optimizer represents an important but difficult decision on the part of the human expert, followed by the equally difficult choice of hyper-parameters for the chosen method. These choices depend on domain, on the data available, on the architecture of the network, and on the training resources available. Furthermore, the choice is restrained to these existing methods, or to the rigorous development of a new optimization method.

In this work, we propose a method to automatically develop an optimizer. Using evolution, a neuromodulatory agent is generated for a training task. This AGRN agent uses an existing optimizer as a base and modifies the parameters of learning at each layer and at each update. Two different optimizers were tested: SGD and Adam. We therefore denote the neuromodulatory versions of these optimizers Nm-SGD and Nm-Adam. These optimization bases were chosen based on their popularity for the chosen task, image classification. We use classification on the CIFAR benchmark to demonstrate this method, but it can be applied to any domain, as evolution can create an optimizer specialized for the domain of interest. We show that the evolved agent can generalize during evolution to different deep ANN architectures and after evolution to a longer training time and to new problems. Furthermore, by analysing the behavior of the evolved AGRN during training, we demonstrate that the location-specific and time-dependent qualities of neuromodulation are important for deep learning training, as they are in the biological brain. This represents a novel foray into location-specific learning for deep

neural networks.

6.2.1 AGRN neuromodulation model

The neuromodulation architecture consists of AGRNs placed between all layers of a deep neural network where weights and gradients are defined (pooling layers, for example, do not have a corresponding AGRN). The parameters of learning in the first of the two layers surrounding each AGRN are decided by the AGRN. The AGRN receives information about the two layers surrounding it, and about the weights and gradients in both layers. The neuromodulation computation happens in three steps: 1) collecting the appropriate inputs, 2) processing these through the AGRN, and 3) using the outputs as learning parameters. This computation takes place at each update step, i.e. at the end of each batch, which we refer to as one iteration. In the neuromodulation architecture, each AGRN has the same genetic code, which is found by evolution; different behavior from the different AGRN copies arises due to the different inputs given at each layer. A separate AGRN is used for the synaptic weights and neuron biases of each layer, so there are two AGRN copies for each layer. All layers except pooling layers use biases. A diagram of the neuromodulation architecture is given in Figure 6.6.

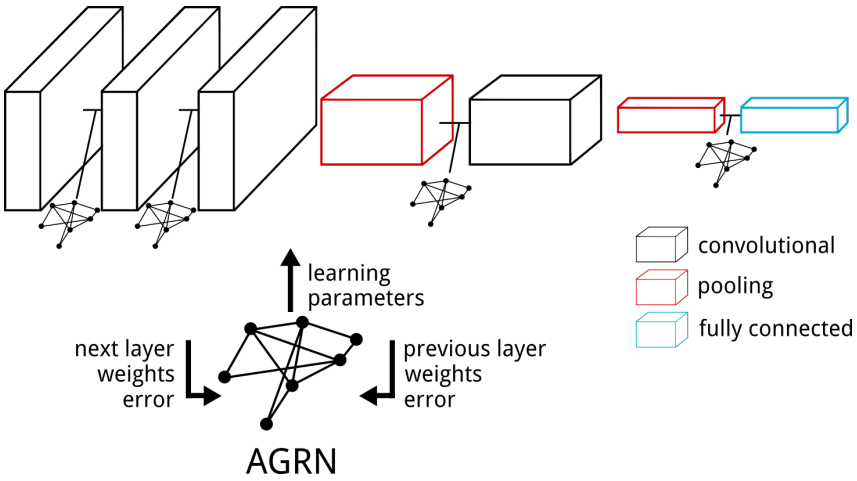

Figure 6.6: The neuromodulation architecture. A copy of the evolved AGRN is placed between all weighted layers of the network. Inputs are given to the AGRN at each batch update with information about the two layers between which the AGRN is placed. The AGRN then outputs the learning parameters to be used in that batch update for the first of the two layers. In this figure, convolutional layers are shown in black, fully connected layers in blue, and pooling layers in red.

The inputs of each AGRN consist of static information about the layer, i.e. its location and size, and statistical information about the weights and gradients, i.e. the mean and standard deviation of both. To be processed by the AGRN, each input must be between 0.0 and 1.0, so different normalization methods or constraints are used. For the layer

location input, each deep ANN layer is given a location index l from 0, the first layer, to L, the last layer, and the input is input is l/L. The layer size input is similarly normalized over the entire ANN; the size of each layer, being the number of parameters in the layer, is divided by the number of parameters of the largest layer in the network. The statistical information is not normalized but instead constrained in $[0.0, 1.0]$. The absolute value of the weights are used for the mean and standard deviation inputs, μ_θ and σ_θ, based on the observation that the magnitude of the weights rarely exceed 1.0. Similarly, the absolute value of the gradients was used to calculate $\mu_{\nabla Q}$ and $\sigma_{\nabla Q}$.

These six inputs, layer location, μ_θ, σ_θ, $\mu_{\nabla Q}$, $\sigma_{\nabla Q}$, and layer size, are found for each layer. The AGRN receives the inputs of the layer before and after it, making 12 inputs. An additional 13th input is also included; this input provides constant activation of 1.0. This was included to mitigate the possible case that, for certain layers, none of the inputs would have a high enough magnitude to provide sufficient activation of the regulatory proteins of the AGRN.

The outputs of the AGRN are the hyper-parameters of the relevant optimizer. For Nm-SGD, the outputs are the learning rate, η, and the momentum parameter α. For Nm-Adam, the outputs are the two β parameters, β_1 and β_2, ϵ, and the learning rate η. The full list of inputs and outputs are given in Table 6.7.

Inputs	SGD output	Adam output
layer location	η	η
μ_θ	α	β_1
σ_θ		β_2
$\mu_{\nabla Q}$		ϵ
$\sigma_{\nabla Q}$		
layer size		

Table 6.7: The layer inputs and hyper-parameter outputs of the AGRN. All inputs are specific to a layer, and the AGRN receives two copies of these inputs, one for each layer surrounding the AGRN. With the constant activation input, there are 13 total input proteins. For each output parameter, two AGRN output proteins are used. There are therefore 4 output proteins for SGD and 8 output proteins for Adam.

In the standard AGRN update step, protein concentrations are normalized to sum to 1. This is a part of AGRN computation that has been shown to be necessary [Dis+17], but it can have the undesirable consequence of restraining the protein concentration levels. In order to allow the AGRNs to control the magnitude of its outputs, irrespective of normalization, two output proteins are assigned to each learning hyper-parameter. The normalized difference between the two output protein concentrations, o, is then used to

compute the hyper-parameter output, O:

$$O_i = \frac{|o_{2i} - o_{2i+1}|}{o_{2i} + o_{2i+1}} \tag{6.10}$$

The AGRN used in Nm-SGD therefore has 13 input proteins and 4 output proteins, and for Nm-Adam it has 13 input proteins and 8 output proteins. These two different neuromodulation schemes present two different optimization problems; to find an AGRN, with the respective number of inputs and outputs, capable of improving overall learning of a deep ANN by making local changes to the learning parameters at each layer. In the next section, we describe the use of artificial evolution to find the neuromodulatory AGRNs.

6.2.2 Evolution of the neuromodulatory agent

The evolutionary method used in this work is GRNEAT, a genetic algorithm specialized in AGRN evolution which is covered in full detail in Chapter 3. A key component of using GRNEAT, or any genetic algorithm, is the design of the evolutionary fitness function used for selection. In this work, we aim to find an AGRN which improves learning. Specifically, we use each AGRN individual during training for the same number of epochs, $E = 20$, on the same deep ANN model with the same random initialization. We then compare these individuals based on their accuracy on the trained task. In order to ensure generalization, we modify the deep ANN model and random initialization seed at each generation.

The task used for training during evolution is CIFAR-10 [KH09]. This is a standard image classification task where 60000 32x32 color images are presented from 10 classes, with 6000 images per class. We chose this benchmark due to its prevalence in the literature and for the ease of testing a more difficult problem, CIFAR-100, without needing to change deep ANN models or the data infrastructure. CIFAR-100 is the same size as CIFAR-10, but has 100 classes containing 600 images each. Results on the CIFAR-100 benchmark are presented in subsection 6.2.4.

At each generation during evolution, a deep ANN model is chosen. This model is used to evaluate all individuals in the generation, providing a standard platform for comparison. We use three different models, $m0$, $m1$, and $m2$, which are presented in Table 6.8. These models were based on popular image classification architectures (LeNet and VGG16) and were chosen to evaluate neuromodulation on a variety of model sizes, from the small $m0$, which is unable to solve CIFAR-10, to the complex $m2$. The choice of model during evolution is random; one of the three models is chosen per generation according to a uniform distribution.

Each AGRN individual is therefore used to train a deep ANN, of one of the three architectures, on the CIFAR-10 dataset. To evaluate this individual, a fitness metric must

$m0$	$m1$	$m2$
conv 32	conv 64	conv 64
conv 32	maxpool	maxpool
maxpool	conv 128	conv 128
conv 64	maxpool	maxpool
conv 64	conv 256	conv 256
maxpool	maxpool	maxpool
fc 512	conv 512	conv 512
fc n_{out}	maxpool	maxpool
	fc 4096	conv 512
	fc 4096	maxpool
	fc n_{out}	fc 4096
		fc 4096
		fc n_{out}

Table 6.8: The three models used during evolution, $m0$, $m1$, and $m2$. Layer types are convolutional (conv), maxpool, and fully connected (fc), with the size of the layer indicated. n_{out} is the number of outputs and depends on problem; for CIFAR-10, used during evolution, $n_{out} = 10$.

be calculated. We compare three different fitness metrics for learning: "test", "avg", and "train". The CIFAR datasets are split into training and testing sets of 50000 and 10000 images, respectively. All individuals are trained only using the training set. The "train" fitness metric is simply the accuracy at the end of this training. The "test" fitness metric is the accuracy of the trained model on the test dataset. The "avg" fitness metric is the sum of the "train" and "test" metrics, divided by 2.

For all metrics, the population is able to converge quickly, as seen in Figure 6.7. Large oscillations are visible in the fitness, especially for the training accuracy fitness metric. These are caused by changes in the neural network model; as $m0$ is especially small, generations using this model suffered in fitness. However, it is evident that the 50 generations used here are not all necessary. The evolution of the AGRN can be a costly process, where each individual evaluation consists of training a deep ANN, so it is useful that as few as 10 generations may suffice.

The size of the evolved AGRNs is also included in Figure 6.7. The sizes of the best individuals are rather surprising as they remain small; SGD has a minimum of 17 proteins (13 inputs, 4 outputs), where Adam has a minimum of 21 proteins (13 inputs, 8 outputs), and many of the expert individuals have only a few additional regulatory proteins. GRNEAT will tend to increase the number of regulatory proteins over time, which can be seen here, but it is also evident that complex AGRNs are not required. For SGD on training fitness especially, few regulatory proteins were necessary for the best individual,

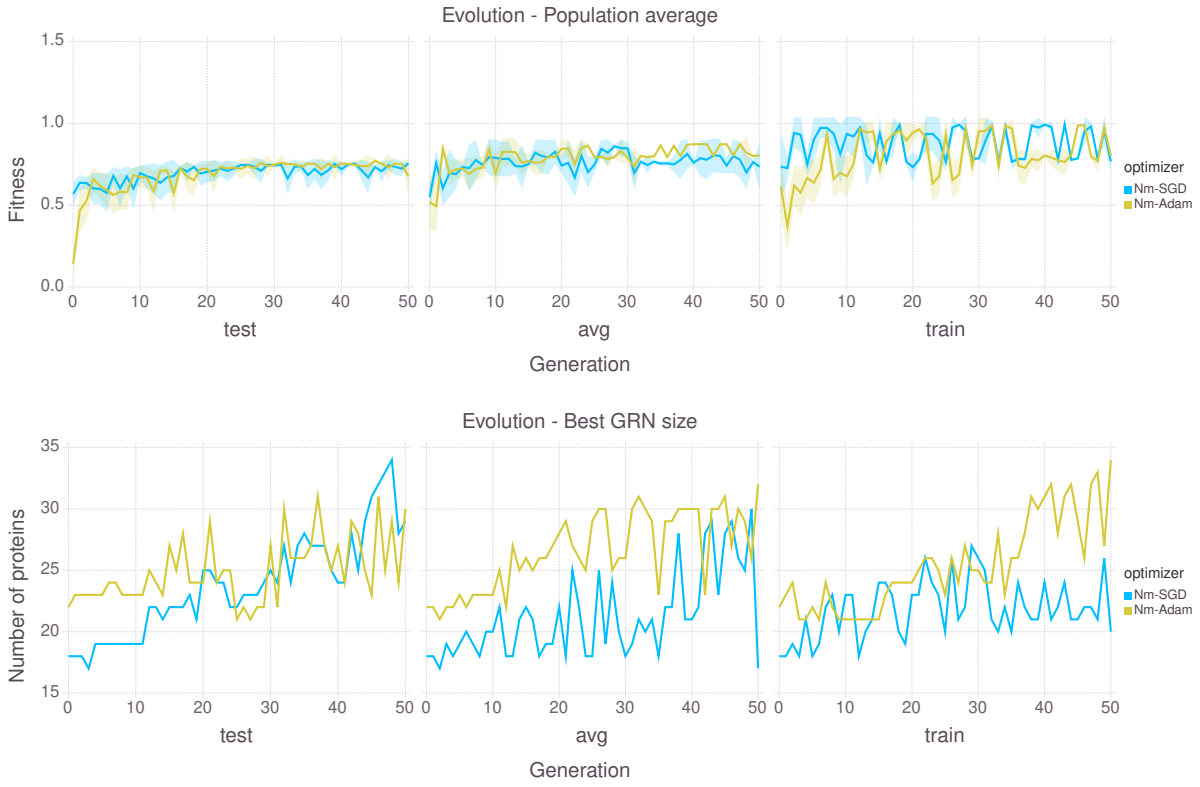

Figure 6.7: Top: The average fitness of the population during evolution, for the three different evolutionary fitness metrics evaluated. Bottom: The size of the best individual from each generation, in the number of proteins.

suggesting that even simple neuromodulatory functions can be effective.

The performance of the best evolved individuals from the three different fitness metrics are shown in Figure 6.8. As can be seen, the three individuals have very similar performance, despite having been evolved for different goals. This may be due to the choice of benchmark: both the train and test accuracy achieved by neuromodulatory learning are near the maximum reported values for CIFAR-10. Given that the performance was the same for all three individuals, we chose to proceed with the individual evolved only on the training data. In this way, the AGRN has no advantage from exposure to test data and can be compared to hyper-parameter choices made based on training performance alone. For the next sections, we observe the behavior of the two best individuals (one for Nm-SGD and one for Nm-Adam) from the 50th generation of the "train" evolution and compare them to their base optimization methods.

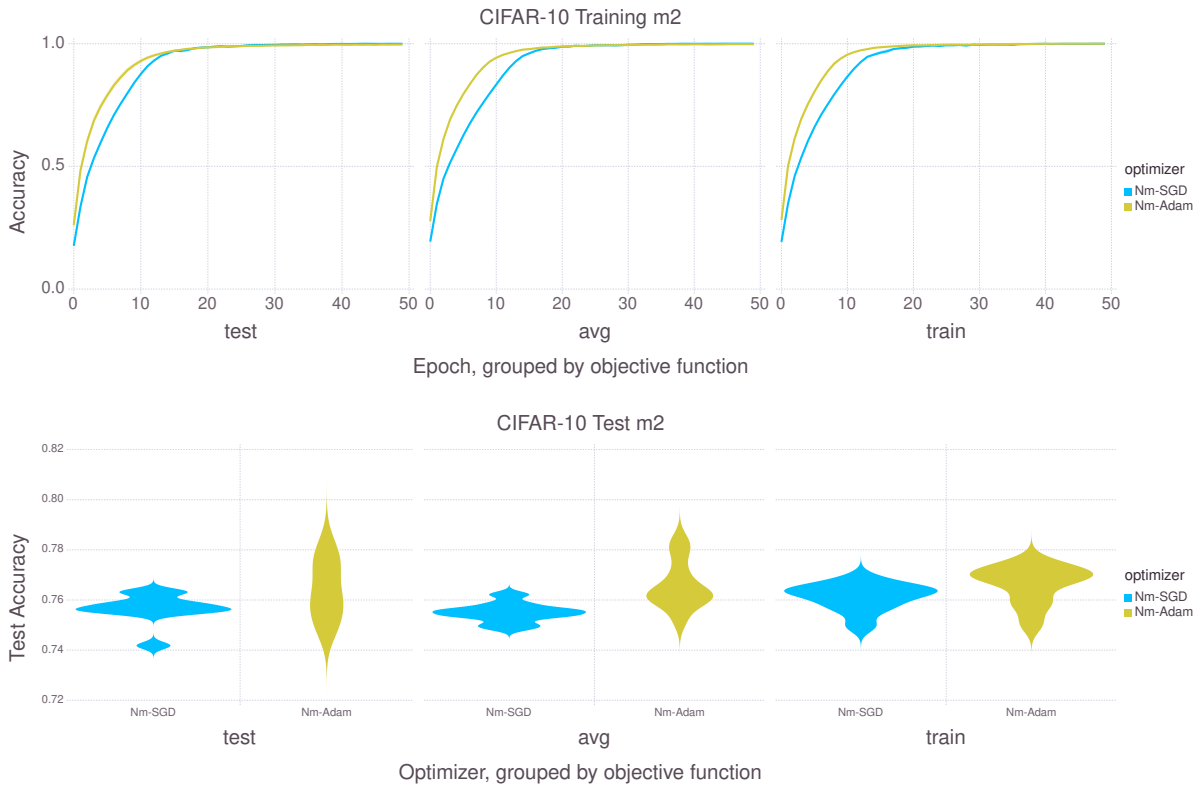

Figure 6.8: Training and test accuracy for the three different evolutionary fitness metrics. Despite representing different learning challenges, the evolved AGRNs had similar performances. Only training on $m2$ is shown to conserve space; $m0$ and $m1$ showed similar results.

6.2.3 Comparison of neuromodulation to standard optimization

Using the best individual from the last generation of the "train" fitness metric evolution for Nm-SGD and Nm-Adam, we compare neuromodulation with standard methods. We first compare them on the task used during evolution, CIFAR-10. For SGD and Adam, we use the default hyper-parameters of Keras. For SGD, this is $\eta = 0.01$ and $\alpha = 0.0$. For Adam, this is $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e-08$. These choices are standard for both of these methods, and also standard for image classification tasks. We chose these hyper-parameters due to their prevalence and in order to have a single parameter set across models. Normally, hyper-parameters are optimized for a specific architecture. As we use the same AGRN individual for the three different architectures, we wanted a single hyper-parameter set per optimizer for comparison.

We trained the three different models using the compared optimizers 10 times each. This was done to ensure fair comparison with different random initial weights. The results of the comparison on training and test accuracy are presented in Figure 6.9.

First, we observe that the neuromodulation methods, Nm-SGD and Nm-Adam, are able to generalize to longer training times. These methods were evolved for 20 epochs,

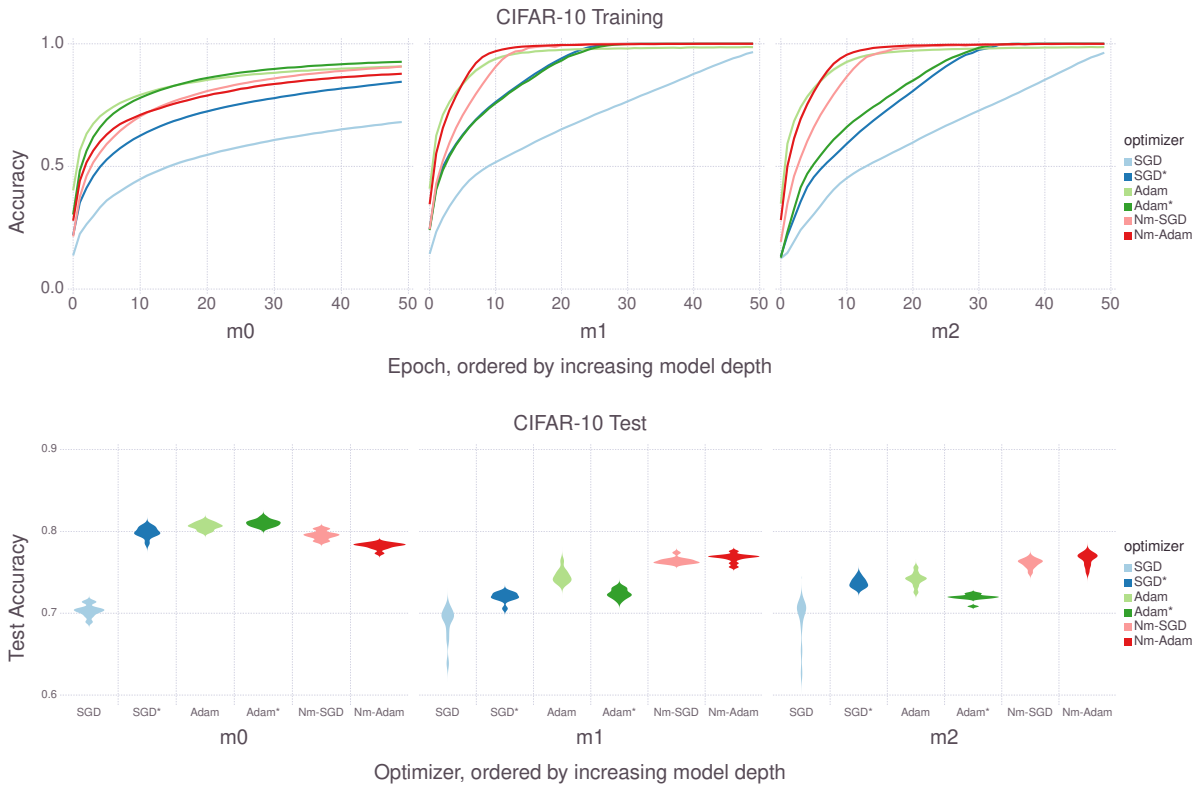

Figure 6.9: Training and test accuracy on the CIFAR-10 benchmark of the neuromodulatory methods, Nm-SGD and Nm-Adam, compared to their base methods, SGD and Adam, using the default parameters of Keras. Training was performed 10 times on each model and method. In the top training plot, average accuracy is shown as a line and one standard deviation is shown as a ribbon.

but are here trained for 50 epochs. Especially when training on the model $m0$, we can see that Nm-SGD and Nm-Adam both continue to aid training after 20 epochs.

We can also see that neuromodulation can limit overfitting. While the training accuracy achieved by Nm-SGD and Nm-Adam are both very near 1.0 by the end of 50 epochs on $m1$ and $m2$, the test accuracy for both methods on these models is superior to that of either standard SGD or Adam.

Finally, we see that the neuromodulatory training can converge faster than standard training, seen especially for models $m1$ and $m2$. This may be a result of the use of 20 epochs in the evolutionary fitness, as convergence speed wasn't explicitly selected for. Instead, by evaluating AGRNs after 20 epochs, those that converged early may have presented an evolutionary benefit.

6.2.4 Generalization of the neuromodulatory agent

Next, we compare these four optimization methods on the CIFAR-100 benchmark. The experiment is the same, i.e. the three different models are trained in 10 separate trials, but

the dataset is different and 100 epochs are used to account for the more difficult dataset. CIFAR-100 is a challenging change from CIFAR-10. The number of classes increases from 10 to 100, and the number of examples from each class decreases from 6000 to 600 (5000 to 500 in the training set). The network must therefore be trained on sparser data for more classes. This new task would often present the need to find new hyper-parameters, requiring an expensive tuning search. Here, we use the AGRNs evolved on CIFAR-10 to test their generalization to CIFAR-100 without change. The results from this experiment are presented in Figure 6.10.

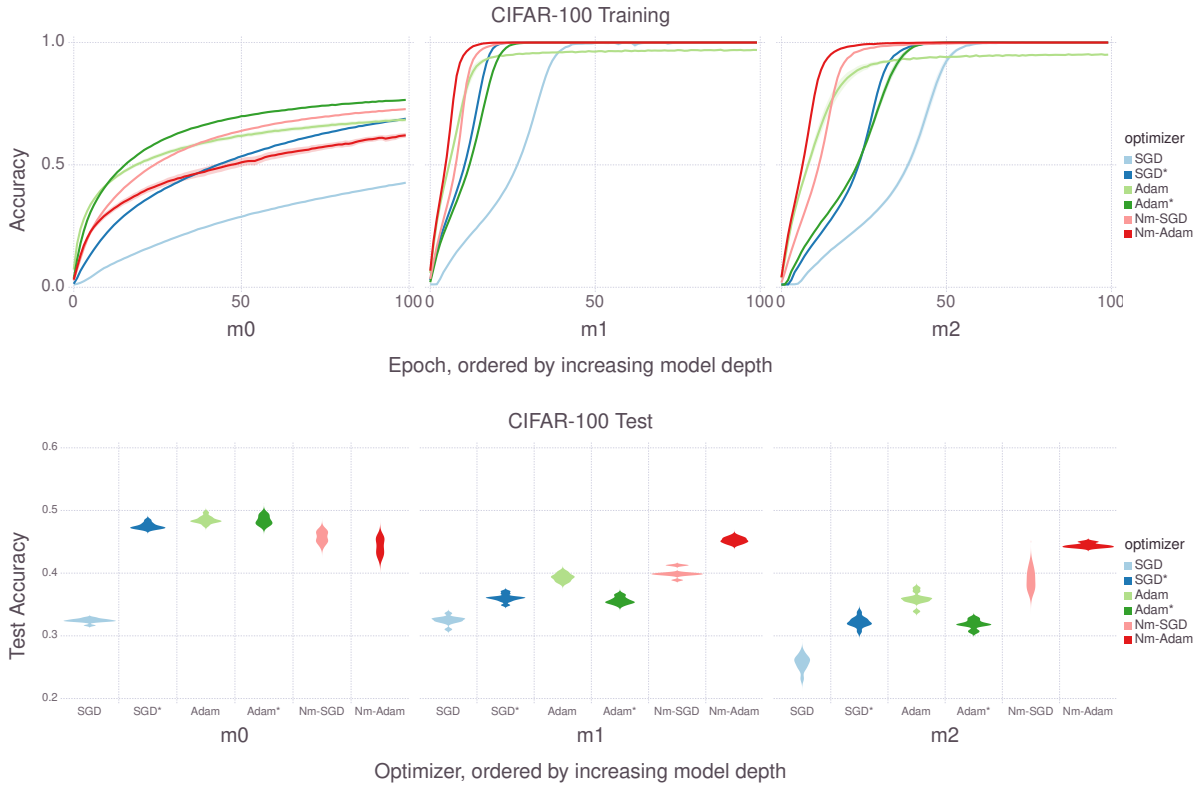

Figure 6.10: Training and test accuracy on the CIFAR-10 benchmark of the neuromodulatory methods, Nm-SGD and Nm-Adam, compared to their base methods, SGD and Adam, using the default parameters of Keras. Training was performed 10 times on each model and method. In the top training plot, average accuracy is shown as a line and one standard deviation is shown as a ribbon.

Nm-SGD and Nm-Adam show clear capabilities to adapt to this new task, outperforming the base methods on $m1$ and $m2$, while producing early equivalent test results on $m0$. The training accuracy on $m0$ of Nm-SGD is the highest of the methods tried, but Nm-Adam is lower than standard Adam. However, it appears that the Nm-Adam optimization did not converge in the 100 epochs provided, while standard Adam appears closer to convergence.

The conclusions drawn from the CIFAR-10 experiments hold for CIFAR-100. Both

neuromodulation methods exhibit an ability to generalize to longer training, here extending from 20 epochs during evolution to 100 epochs here. The test accuracy, especially of Nm-Adam, remains high across different models and is better than that of SGD and Adam for $m1$ and $m2$.

These experiments demonstrate the viability of this method. Evolved AGRNs can make effective hyper-parameter choices at each layer, leading to optimized learning. As shown in subsection 6.2.2, these AGRNs may be small, suggesting that the function of processing the inputs to generate hyper-parameters may not be incredibly complex. We now examine the behavior of the AGRNs to understand this function and the hyper-parameters chosen.

6.2.5 Neuromodulation behavior

To understand the behavior of the evolved AGRN, we observe the inputs and outputs of each AGRN copy during training. Specifically, we present the Nm-Adam CIFAR-10 training of $m1$. This training is shorter than CIFAR-100 and involves fewer AGRN copies than $m2$. We evaluate a single training, not the training over 10 different initial weight conditions, as presented in the previous sections. These choices were made to allow the results to be better understood, as the inputs and outputs of each AGRN over training represents a large amount of data, while providing an interesting use-case, the Nm-Adam training on $m1$. The training is presented in iterations, which are the update steps at each batch. The batch size used in all experiments was 128, meaning there were 391 iterations per epoch and a total of 19550 iterations over 50 epochs.

We first present the inputs given to each AGRN in Figure 6.11. We show only the first six inputs, as this contains all of the relevant information. The next six inputs of each AGRN are the same values for the next layer, already represented in the shown inputs. The final input, the constant activation input, is always 1.0 for all layers.

The surprising aspect of these inputs is that the biases of the final layer do exceed 1.0 after a small amount of training. This restricts the information the AGRN is able to receive about these weights, as μ_θ is constrained to 1.0. We also see that the gradient mean, $\mu\nabla Q$ is generally very small and could potentially be scaled when provided as an input for easier use by the AGRN. Finally, while the gradient provides noisy oscillations, most inputs are static throughout the training, especially after the halfway point of 1e4 iterations.

The hyper-parameters decided by the AGRN are presented in Figure 6.12. For this evolved AGRN individual, it is clear that there is a nearly direct relationship between μ_θ and η for the biases in the last layer, as the learning rate mirrors the bias mean almost exactly. There are notable differences in the early layers, however, with the learning rate

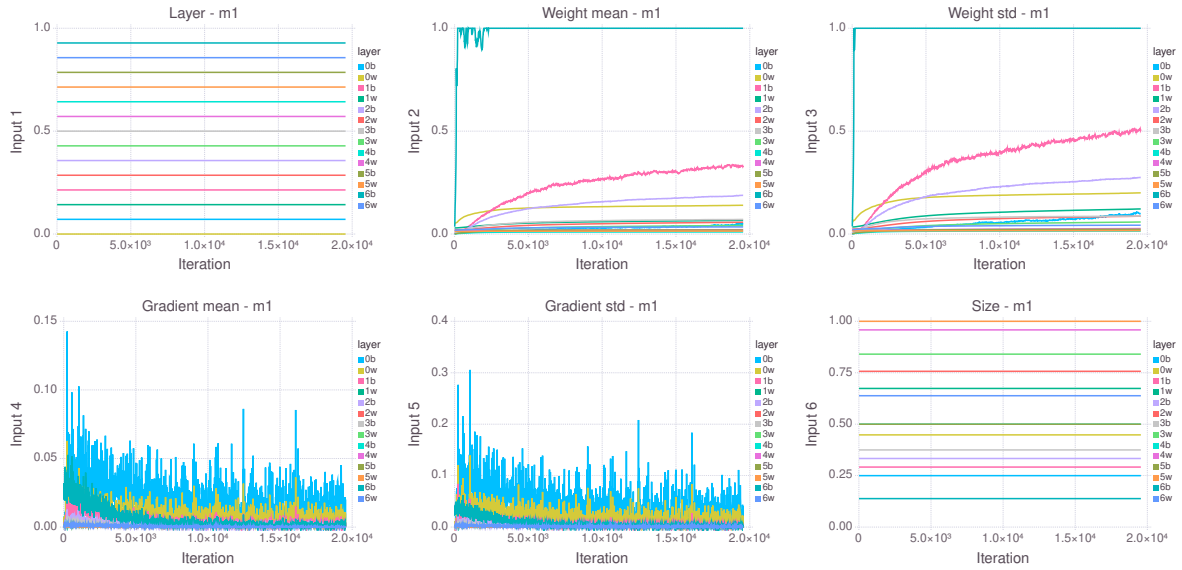

Figure 6.11: Input protein concentrations during Nm-Adam training on $m1$, with each AGRN copy represented by a different color. Layer names indicate either neuron bias, "b", or synaptic weight, "w".

of the weights of layer 0 displaying an inverse relationship with μ_θ. This is an example of a location-specific neuromodulation strategy, a behavior which is not possible with standard optimization methods.

Most hyper-parameters are static over time, or like the training, stabilize after 1e4 iterations. The β_2 parameter of the biases of layer 1 is an exception to this, decreasing during the second half of training. This means that, near the middle of the training, the update of v_θ depends almost entirely on its previous state and not on the squared gradient. It only changes based on the squared gradient at the beginning and ends of training. While this behavior was not common for the hyper-parameter choices, this sort of variability over time is a known aspect of neuromodulation and is also not possible using standard optimization methods.

6.2.6 Summary of neuromodulation of learning parameters in deep ANNs

We have shown in this work that neuromodulation can improve deep ANN learning, using principles based on biological neuromodulation. Local signals which change over time decide the rate of learning and the importance of other factors, such as momentum, in weight change. We have shown that a neuromodulatory agent can be found automatically through evolution, and then applied to further use cases after evolution. The agent analyzed in this work displayed novel behavior during optimization, changing learning strategy based on location in the ANN and training time. We believe these characteristics

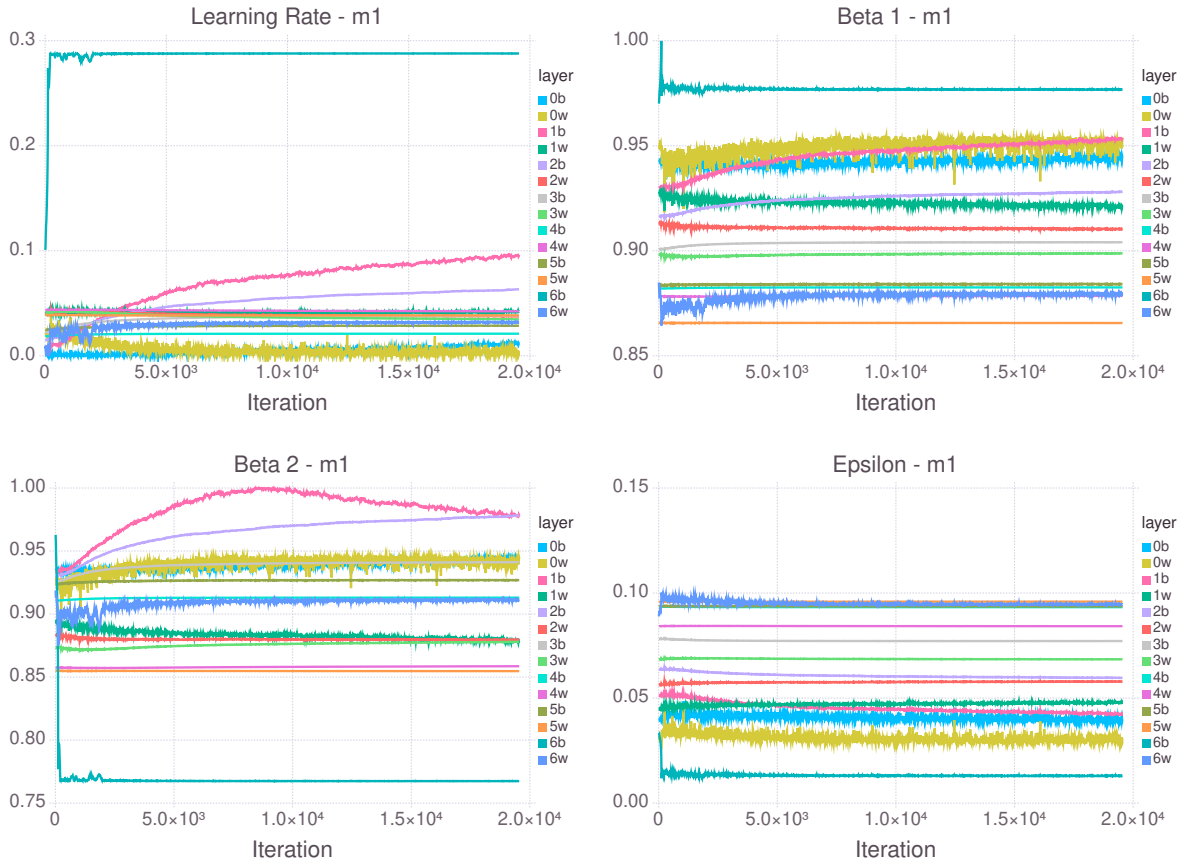

Figure 6.12: The hyper-parameters chosen during Nm-Adam training on $m1$, with each AGRN copy represented by a different color. Layer names indicate either neuron bias, "b", or synaptic weight, "w".

of learning could lead to the design of new optimization methods.

We also believe that this method could lead to shared improvements for deep learning. At the end of evolution, a single AGRN individual is selected from the entire population, across all generations. In this work, this was the best individual from the final population. However, a single evolution creates a wealth of different individuals, which can be selected based on other characteristics, like their size. We have shown that a single individual can generalize to longer training and new problems without needing more training. We can therefore imagine that neuromodulatory agents could be shared, much as trained neural networks are shared, with agents evolved for certain task types and architectures, i.e. image classification. As in the case of "model zoos", a repository of neuromodulatory agents could be available for download, eliminating the need to choose hyper-parameters and improving optimization.

In this work, we have shown a preliminary experiment in the nascent field of artificial neuromodulation. We focused on improving learning and used standard classification benchmarks to evaluate the neuromodulatory agent. However, neuromodulation could

assist learning in other ways. For example, catastrophic forgetting is a large problem in deep learning, where ANNs trained on one task completely forget their knowledge when trained on a second task. A neuromodulatory agent could be evolved to mitigate forgetting. In the next section, we focus on reinforcement learning, where neuromodulation acts on a reward signal instead of error gradients. The same could be applied to deep reinforcement learning, where the definition of a gradient is a difficult task.

6.3 Conclusion

In this chapter, we've presented two separate cases of using evolution to create methods of learning. Both of these works use existing learning methods as a base, but offer improvements which perform learning in novel ways. Specifically, the importance of neural placement is taken into consideration for learning, and the learning methods are dynamic, changing over time and in response to neural activity. Both of these present new directions in the domain of artificial neuromodulation. In the first work, reward-modulated STDP is considered in a physical space, with a dopamine signal that acts as a physical chemical in extra-cellular space. In the second work, neuromodulation is used for the first time on deep neural networks, showing clear benefits of layer-specific training.

This chapter focused on weight-based learning alone, as this is the most common form of learning in ANNs. In biology, learning is not only synaptic but also structural. New information can require new dendrites and synapses, which is not reflected in ANN models. The same principles of neuromodulation used here could be extended to developmental neural models, encouraging synaptogenesis during positive reward signals and pruning synapses during negative ones.

In the works presented here, evolution is shown as a capable means of automatically discovering neuromodulation agents and methods. However, novel learning methods could be entirely created evolution. Both the SGD equation (Equation 6.4) and the STDP equation (Equation 6.2) are feasible targets for genetic programming; they don't require a large variety of functional components, and have rather simple topologies when represented as abstract syntax trees. One could imagine GP finding these methods or improvements to them automatically, without working with an existing learning base.

In the next chapter, we present a complete ANN model designed for GP. All three components covered in the previous chapters, neural cell function, neural connections, and learning methods, are considered as programs which can be optimized using GP. This framework is designed to create novel learning methods, not building on top of existing learning methods as done in this chapter, but generating new learning programs automatically.

Chapter 7

Discussion and conclusion

In chapter 2, we presented the objective of this thesis: to discover, through evolution, the principles of neural design, inspired by biology, which can improve artificial neural networks. We have approached this goal by separating ANNs into cellular function, connectivity, and learning, and evolving rules for each of these components. We have demonstrated that these components can be evolved to construct or improve ANNs for data classification, robot foraging, animat locomotion, and computer vision. In this chapter, we discuss the principles discovered by evolution during these experiments. We then apply these principles in the formation of a novel framework for functional developmental neuroevolution, presented in section 7.1.

The methods used for evolution were presented in chapter 3, being the Artificial Gene Regulatory Network with GRNEAT evolution and Cartesian Genetic Programming. We demonstrated that both methods can be used to evolve controllers capable of performing a variety of tasks. AGRNs were used for different signal processing tasks and controlling agents in two games: Flappy Bird and Ship Escape. This study demonstrated important components of AGRN evolution, such as the necessity of the protein normalization step in AGRN dynamics. This also demonstrated that a floating point genome representation can benefit AGRN evolution, which creates new possibilities for AGRN evolution. For example, CMA-ES, which optimizes over continuous space, could be used for fixed-length AGRN evolution. These findings were used in chapter 5 and chapter 6 to inform AGRN model used and its evolution.

We also presented a number of possible genetic operators for CGP, a new gene representation using floating point numbers, and a new CGP method called Positional CGP, where the position of nodes is evolved. We demonstrated that CGP is a very capable method for evolving controllers, achieving results competitive with deep learning on the Atari benchmark set. We then studied optimal representation and evolution parameters on multiple robot locomotion tasks. These methods were used in chapter 4, and will again

be used in the next section.

In chapter 4, we evolve the function of a spiking neuron for use in data classification. This represents a new approach in activation function design, which is the automatic optimization of the function for improved performance. Many activation functions exist, most having been according to specific biological data or to reduce the complexity of other biological models. We use the Leaky Integrate and Fire function as a base for CGP and show that CGP is able to improve the function for data classification.

CGP is also used to evolve a completely new activation function, which is able to compete with the LIF model and generalized better than the LIF model to new datasets. However, by inspecting the evolved function, we observe that the function does not resemble the simulation of membrane potential, as all activation functions do. It is periodic, similar to LIF function, using the *sin* function, and takes advantage of the neural state for decision, as does the LIF function, but it has no reliance on a semantic understanding of the membrane potential.

This informs the first principle found by evolution: ANNs do not need to rely on the simulation of neural activation alone for computation. Excitability is certainly an important part of neural functionality, if not the most important, and has proved an important base for ANNs. However, neural behavior is more complex than just spiking. In biological neural networks, many neurons have been observed which never or rarely spike, but which have been shown to be crucial for learning and cognition [SOS06]. Furthermore, the brain is composed of a variety of cells beyond the neuron, which should be represented by other functions. As found in [Por+11], other cell types can improve learning. In ANN design, cell functionality should be expanded beyond neural activation, not only to draw ANNs closer to biological reality, but also to improve their training and use.

We next evaluated the way in which connections form in the brain, focusing on the process of axon guidance in chapter 5. We evolved the controllers for glial cells, which emitted morphogens in a 3D space, and axon growth cones, which followed these morphogens to connect to somata in a different section of the space. We used this model to study the dependence of activity in neural development, showing that evolution automatically used activity in these controllers. This model was then used to develop network topologies for robotic foraging.

In this model, glial cells could only communicate with growth cones through chemical diffusion; there was no synaptic wiring from the brain section to the eyes in the eye-specific patterning experiment. Using four different chemical channels, evolution discovered morphogen diffusion rules which allowed for meaningful information passage to guide the growth cones. Some of these rules communicated spatial information, differentiating between left and right in the eyes or brain section. All signals changed over time, as seen in the changing concentration distributions during simulation.

In this experiment, evolution demonstrated the principle that volume transmission, communication using chemical as opposed to synaptic medium, can be an effective form of information transfer in ANNs. Volume transmission can be considered as a broadcasting communication, where a single cell sends the same signal to many cells, a one-to-many scheme. Modern ANNs use synaptic communication between neurons, which is one-to-one, and pooling layers, which are many-to-one. Volume transmission could also be considered as a case of one-to-one communication, as each cell receives a different version of the signal based on factors of signal attenuation and delay, if these are simulated. However, that also has potential benefits for ANNs. In wire transmission, each synapse has a weight property which must be optimized, but in volume transmission, the signal strength between two cells depends on the properties of the cells, such as their position. This can drastically reduce the search space [Wil+18a].

Finally, in chapter 6, we presented two new methods of artificial neuromodulation. The first is based on spiking-timing-dependent plasticity and improves semi-supervised learning by introducing a dopamine signal which travels through the network, becoming weaker as it travels and being absorbed over time. We demonstrated that this model, using fixed parameters for the dopamine signal, is capable of learning based on delayed reward in an instrumental conditioning experiment. By optimizing the parameters of the dopamine signal and reward-based activation using CMA-ES on two animat locomotion problems, we were also able to demonstrate the important properties of reward-modulated. Evolution demonstrated that a transmission delay in the dopamine signal and absorption over time are important factors for learning.

The second experiment in chapter 6 used stochastic gradient descent as a base for neuromodulated learning. Using standard image classification benchmarks, we demonstrated that the parameters of gradient descent can be tuned at each layer during learning to increase learning. AGRNs evolved to increase classification accuracy at the end of a fixed training period on a specific problem were able to generalize to learning on longer training periods and to a more difficult problem. The behavior of the AGRNs demonstrated learning strategies which adapted over the course of learning and varied based on location in the neural network.

In both experiments, evolution favored learned strategies which rely on spatial and temporal information, and in the second experiment, recent synaptic change was also shown to influence learning. This coincides with biological principles of learning; in the brain, the learning process changes based on neural location, age, and past learning [Hos+11], [SG10]. ANNs have already benefited from learning which uses state to modify learning based on recent synaptic change; the Adam optimizer updates weights using a moving average of the gradient [KB14]. As show in subsection 6.2.5, this can be further improved by changing the reliance on past gradients over time for specific neurons.

The principle supported by these experiments is that ANN learning should be dynamic, change over time and location, and use feedback about previous learning.

These principles together motivate the design of ANNs that use a variety of cell behavior. These cells should communicate not only through synapses, but also using chemical signals. The learning in these ANNs should use local strategies that change over time and have self-feedback. In the next section, we present a framework designed to address these principles and to facilitate and encourage study in the evolution of controllers for ANN design.

7.1 A framework for developmental neuroevolution

A number of excellent frameworks exist for the simulation of neural networks and for training ANNs [Bre+07]. The Brian spiking neural network simulator [Sti+14] is well-known and used in a number of works which informed this thesis [DC15], [WA12], [WAJ14]. Tensorflow [Aba+16] and Keras [Cho+15] are popular deep learning frameworks and were used for the hyper-parameter experiment in section 6.2. However, there are no frameworks which are well-suited for neuroevolution. In the existing frameworks, replacing a component of the neural network with an evolved component is non-trivial and often requires a unique interface to the evolved controller. Extracting information from these frameworks to inform evolution can be difficult, especially when creating behavioral metrics necessary for novelty search methods. These frameworks are also not suited for neural development; there is, to our knowledge, no neural network framework which natively allows for changes in neural architecture during learning, for example.

We address this need by proposing a framework, informed by the principles discussed in the last section and created to encourage similar experiments. In this section, we present the Neurodevo framework, a framework for neural development and evolution, and preliminary results using this framework for data classification. The framework is designed for customization, with a separation of functions which allows for experiments that use fixed neural architectures with specific learning rules as easily as it allows for experiments which evolve developmental rules and learning strategies. An open-source implementation of this framework in Julia is available online [1].

Variable representations used in the Neurodevo framework can be found in Table 7.1. The Neurodevo framework uses two object types: Cells and Connections (C and X, respectively), both of which have parameters and state. Parameters can represent information like cell type, connection location, and information which changes infrequently about the object, such as average recent activity. Parameters are updated every T_{learn}

[1] https://github.com/d9w/Neurodevo.jl

C	Cells	I_C	Input from source Cell
X	Connections	I_X	Input from Connections
P_C	Cell parameters	I_C	Cellular output
P_X	Connection parameters	O_X	Connection output
S_C	Cell state		
S_X	Connection state		

Table 7.1: Variable representations used in the Neurodevo framework.

timesteps, which encourages cells which change during learning. Object state updates more frequently, at each timestep, and can represent the immediate action of cells or connections, such as membrane potential or firing rates.

However, the semantic meaning of each parameter or state variable is not fixed in the Neurodevo framework; it is configurable, either by the user or by evolution. Parameters, P, and state, S, are represented as arrays of floating-point numbers with fixed lengths: N_{P_C}, N_{S_C} for cell parameters and state, and N_{P_X}, N_{S_X}. These length hyper-parameters can be configured by the user or included as part of evolution.

The flexibility of the Neurodevo framework is given by the definition of 10 functions which determine how neurons function, form connections, change over time, and how the connections change over time. Using object parameters and states as inputs, these functions decide the behavior of the ANN. The 10 functions and their inputs and outputs are given in Table 7.2. These functions can be supplied by the user or evolved. All functions have a fixed number of input and outputs, and the range of all inputs is guaranteed by the framework to be in $[-1, 1]$, which makes them suitable for many evolutionary controllers without further modification.

The `cell_state_update` function can serve as an example of flexibility. This function is applied for each cell at each timestep and determines the new state for the cell, S'_C, as well as its output, O_C. The inputs to the function are the cell parameters, P_C, the current cell state, S_C, and input from any connections which connect to the cell, I_X, which is reduced to a single value using a sum. One function definition could be a ReLU: $O_C = max(0, I_x)$. As a ReLU does not use state, S'_C could be set to S_C or other values. An LIF neuron could instead be defined by storing the membrane potential as part of S_C, i.e. s, an element of S_C:

$$V = \eta s_i + I_x \tag{7.1}$$

$$s'_i, O_C = \begin{cases} V, 0 & \text{if } V < V_{thresh} \\ V_{reset}, 1 & \text{if } V \geq V_{thresh} \end{cases} \tag{7.2}$$

Function	Inputs	Outputs
`cell_division`	P_C	boolean
`new_cell_parameters`	$P_{C_{parent}}$	$P_{C_{child}}$
`cell_death`	P_C	boolean
`cell_state_update`	P_C, S_C, I_X	S'_C, O_C
`cell_parameter_update`	P_C, S_C	P'_C
`connect`	P_{C_i}, P_{CX_j}	boolean
`new_connection_parameters`	P_{C_i}, P_{CX_j}	P_X
`disconnect`	P_{C_i}, P_{CX_j}	boolean
`connection_state_update`	P_X, S_X, I_X, I_C	S'_X, O_X
`connection_parameter_update`	P_{C_i}, P_{CX_j}, P_X, S_X	P'_X

Table 7.2: The ten functions used in the Neurodevo framework and their inputs and outputs. P and S refer to to parameters and state, respectively, C and X to Cells and Connections, and CX to inputs which can be either from a Cell or a Connection. The inputs and outputs which pass between objects are given as I and O, with I_C and I_X corresponding to input from Cells and Connections, respectively.

where V_{thresh} and V_{reset} are set by the user or evolved, but must be in $[-1, 1]$ to conform to the framework requirements. As demonstrated by this case, different function definitions can lead to vastly different ANNs. Example function sets are provided with the Neurodevo framework which implement a feed-forward ReLU network with Hebbian learning and a static architecture, and a spiking neural network which includes STDP, neurogenesis, cell apoptosis, synaptogenesis, and connection pruning.

The Neurodevo framework provides some overhead functionality in calling the provided functions, but was designed to allocate as much control as possible to the function set. In general, the overhead of the framework exists solely to ensure that the functions operate correctly; i.e. that connections made to an object which is removed are also then removed, and that the number of cells and connections stays under configured global limits. However, some assumptions were made in the framework design. Cells have a single output O_C, which is immediately transferred to all Connections leading from this Cell as their cellular input, I_C. Connections are directional, and the output of all Connections, O_X leading to a cell are summed to inform its input; i.e. $I_{C_i} = \sum O_{X_{i,j}}$. Connections can also be formed from a Cell to a Connection, not only to other Cells. This was done to allow for cells which communicate learning signals to be able to directly influence synapses, which has a basis in biology [PNA09] and is similar to the neuromodulation methods in chapter 6.

The principles defined in the last section are foundational for the Neurodevo framework. Cell function is defined as a state update function which can be designed or evolved to change based on the cell parameters, allowing for cells with specific behavior beyond

excitation and for heterogenous networks. Connections are similarly flexible; in one of the provided function sets, volume transmission is used to distributed reward information from a single cell to all connections in the network, which is then used during their learning stage. Structural and synaptic learning are possible in the Neurodevo framework, and both are based on functions which receive local and temporal inputs, and which have state which can be used for self-feedback. The evolution of functions which adhere to these principles is a challenging task, and in the next section we present a preliminary demonstration of the Neurodevo framework on a data classification task.

7.2 Evolving to learn for data classification

To demonstrate the Neurodevo framework, we use a well-known benchmark in machine learning: classification accuracy on the iris dataset [Fis36]. We provide an initialization function to the framework which places input and output neurons in the environment, protecting these neurons from cell death. During training, I_X of the input neurons is set to the corresponding iris data, which is normalized to $[0, 1]$. After a number of timesteps, determined by evolution, the O_C of the output neurons is used to determine a class. We similarly provide a reward neuron at initialization, which receives a positive input upon each positive classification. This is a semi-supervised approach to classification; the full error is not reported for each item, but only whether or not classification was accurate.

Individuals are composed of 11 chromosomes, the first of which specifies various parameters for use in the Neurodevo framework, such as the frequency of learning updates in the network T_{learn}. The other 10 chromosomes are interpreted as PCGP programs and create the Neurodevo function set. For the four boolean functions, the output of the PCGP program, which is in $[-1, 1]$, is converted to a boolean based on its sign. We chose to evolve all 10 functions, which is a challenging task for evolution, in order to study evolution in the Neurodevo framework. We use a GA with 100 individuals, genetic mutation within each chromosome, n-point chromosome crossover, and elitism. Individuals are evaluated based on their classification accuracy at the end of training.

In Figure 7.1, we can see that evolution is able to find a solution which properly classifies the iris set. The order which the data is presented is randomly shuffled every 10 generations, and evolution was able to choose a random parameter initialization for the input, output, and reward neurons, which can explain the fluctuations in the best reported fitness.

However, on inspection of the individuals and their learning method, we can see in Figure 7.2 that the individuals do not learn but instead begin training with a high classification accuracy, which improves across generations. We observe that evolution has not discovered how to learn to classify data, but has instead internalized the data in the

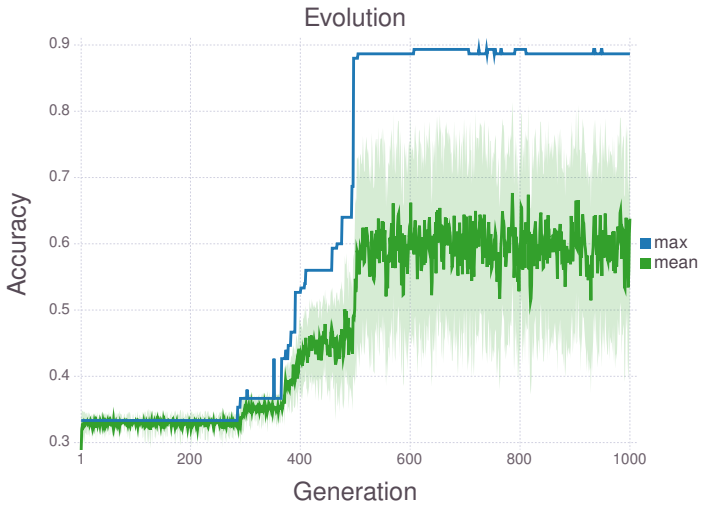

Figure 7.1: The best individual and average population accuracy during evolution on the iris classification problem.

genome, creating individuals which are capable of classification without learning. This is an understood problem in neuroevolution, and specifically in the automatic discovery of learning rules, which we discuss next.

7.3 The evolution of learning

It is interesting to understand why evolution did not discover learning rules, but instead internalized the classification task into the genome. The objective function in this experiment was identical to that in section 6.2: training accuracy after a fixed period of learning. In that experiment, however, evolution found strategies to improve learning and was able to generalize to new data.

Another example is found in [Wil+18a]. In this work, we evolve differentiable AGRNs which can continue to change their protein signatures; in other words, these AGRNs can learn through gradient descent. We perform three separate evolutions with different periods of learning: no learning (0 epochs), 1 epoch, and 10 epochs. The results from these evolutions on the Boston housing dataset are shown in Figure 7.3. Here, the 10 epoch evolution favors individuals who are able to learn well, even at the cost of initial performance on the data regression task. The evolutionary fitness used was again the training accuracy at the end of learning, and yet evolution was able to prefer individuals fit for learning as opposed to learning the data task.

However, the task in this chapter is different from these two cases: in the first example, neuromodulation, an existing learning function was improved upon by evolution. In

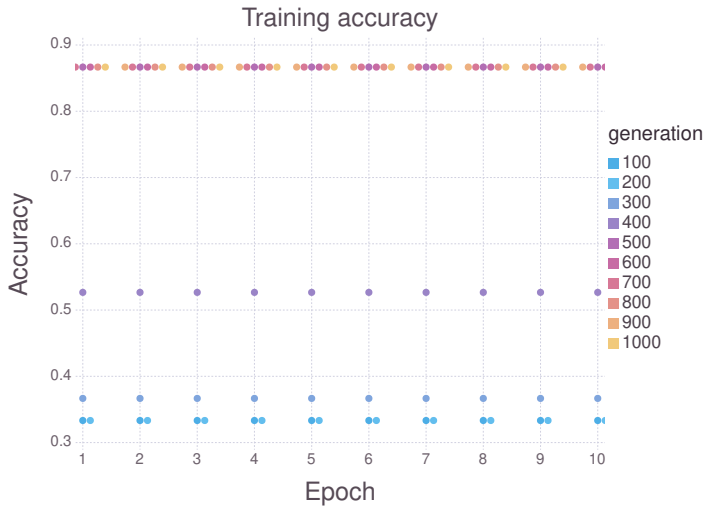

Figure 7.2: Training accuracy over 10 epochs at each 100 generations. Evolution finds strategies able to properly classify the data without requiring learning.

the second, learnability was selected for by evolution, but learning was performed by an existing method, gradient descent. Instead, in section 7.2, evolution is expected to discover how to learn. This task is fundamentally different from the other objective functions; the evolution of learning is a special case of evolution which is not yet fully understood [SSR18].

Learning has been shown to influence evolution in multiple studies [FM91], [NF99], [Bul01]. The most well-known effect occurs when learning is used to acquire constant information about the environment [SSR18]. In this case, learning can accelerate the transfer of information from the environment into the genotype, which is known as the Baldwin effect [Bal96]. When information is constant across generations, individuals which learn this information faster or more completely will have an evolutionary benefit. Individuals which are born knowing even part of the information will therefore require less learning and be selected for against others which have to learn from nothing. Over generations, the static information is integrated into the genotype in this way, creating competitive individuals which don't need to learn the information but instead are born knowing it.

The Baldwin effect applies not only to learning, but to phenotypic plasticity in general, and has been observed in biology in house finches [Bad09] and lizards [Cor+18]. Multiple studies have observed this effect for artificial evolution [HN87], [GW93], [BBS95], [Bul01]. However, it should be noted that this effect is actively debated in the study of both biological and artificial evolution, with the impact of the effect and the means by which to study and measure it being challenged [Anc00], [SSF15].

In this chapter, evolution was able to learn the information due to the static nature

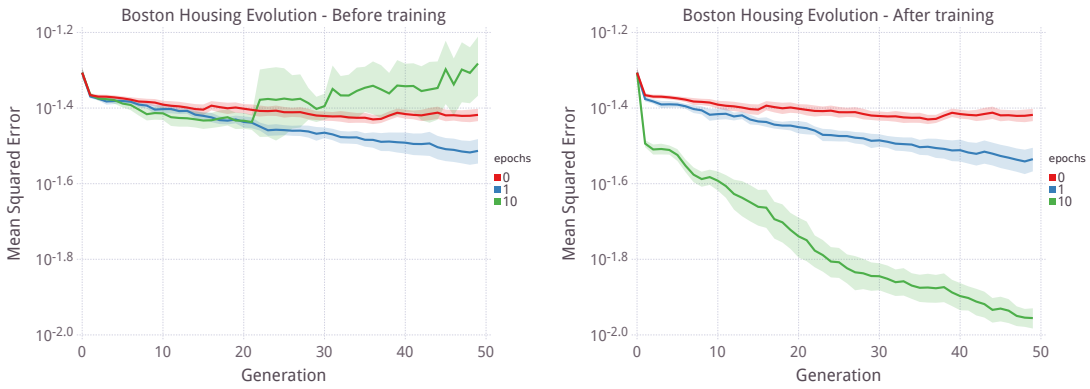

Figure 7.3: Evolution of differentiable GRNs which were trained for 0, 1, or 10 epochs. On the left is the regression error of the evolved agents before training, and on the right is the error after training. The 10 epoch evolution demonstrates the evolution of learnability.

of the problem; at each generation, individuals which knew more of the iris classification problem at birth had a high chance of creating offspring. However, this is not the only case in this work of evolution preferring a static strategy when faced with discovering learning. In section 5.4, evolution was tasked with the discovery of structural learning rules in a robotic foraging environment. In that experiment, the placement of food changed at each generation, creating dynamic information which can not be as easily integrated into the genome. For the continuation of neuroevolution, it is crucial to understand how to encourage the automatic discovery of learning rules.

As demonstrated in this work, static information can be learned by evolution more easily than discovering learning strategies. Dynamic environments eliminate this possibility, as the changing information cannot be encoded in the genome. The amount by which to vary information and the frequency of the change must then be understood. The first matter, the variance of information, has been the object of little study [SSR18]. The frequency of environment changes has been shown to impact the evolution of learning [Ell14], and changes at every generation has been shown to lead to phenotypic plasticity [OL16].

Another means of encouraging the discovery of learning rules is to encourage exploration. Curiosity has been used to improve reinforcement learning, with a recurrent neural network learning the existing behavior patterns to encourage new ones [Sch91]. Intrinsic motivation has also been studied in the context of reinforcement learning and combined with evolutionary search to understand the relationship between intrinsic motivation such as curiosity and extrinsic motivation which leads to higher evolutionary success, showing that this relationship is continuous [Sin+10]. Evolution can also reward individuals who differ from other individuals in a population or throughout a lineage, which has been demonstrated to improve learnability in neural networks [RHS10].

A relatively new idea in the domain of learning discovery is that learning requires the evolution of long periods of parental protection, and that maturity and subsequent evolutionary fitness evaluation should be delayed in evolved individuals [Bul09]. Evolved neural networks have been demonstrated to have sensitive periods of learning [Ell13], and delaying fitness evaluation could protect these individuals from being selected out of the population before sufficiently learning. This idea is supported by ecology, where nurturing has been shown to promote the evolution of learning in uncertain environments [EH12].

In this thesis, we have focused on the evolution of ANN design, including specific forays into the topic of evolving to learn. However, this issue must be understood for neuroevolution to proceed, and we believe that the principles and framework provided in this chapter can serve as guides and tools for studying the complex issue of automatic learning discovery.

7.4 Conclusion

The evolution of learning is one exciting future direction for neuroevolution, and for artificial intelligence in general. Biological evolution has provided a ample inspiration of evolved learning systems, from organisms like ants which rely heavily on social intelligence, to the human brain, to other biological systems which exhibit memory and learning, for example the immune system [FPP86]. In this thesis, we have studied neuromodulation methods which automatically improve existing learning methods, and methods which modify and generate components for learning systems in the creation of new cell functions. Guiding evolutionary computation towards learning methods, following the recommendations of the previous section, could generate a wealth of learning systems adapted for computational tasks or specific hardware.

An equally promising direction is further integration and study of ideas from neuroscience. In this work, we studied activation functions, axon guidance, and neuromodulation, finding that these biological components can serve as meaningful platforms to study and improve ANNs. However, numerous other ideas and structures from the biological brain remain insufficiently studied in ANNs. Dendrites were not included in models in this thesis, as is common in ANNs, but specific dendrite structure and synapse location do impact learning [LKS06]. Astrocytes play a large role in learning in the biological brain and could be used in heterogeneous networks of neurons and glia [PNA09]. Neurogenesis, the development of new neurons, could lead to networks which are able to learn new information more quickly and retain previously learned knowledge [Dow07], [Rus+16].

Finally, as ANNs become more complex and capable, their study can provide insights for the study of the biological brain. We believe that the automatic creation of new,understandable activation functions, as presented in this work, could be used in neu-

roscience to model neurons of specific organisms or under certain chemical conditions where current models break down. We also plan to develop the axon guidance model for use in biological experimentation in order to predict the path an axon will take. Research in deep learning has also started to narrow the distance between biological learning and deep learning, allowing for an understanding of the brain from the perspective of machine learning [Lil+16], [GLR17].

We began this work with the intent to use evolution to discover principles important for artificial neural design. Through the study of neural cell function, connectivity, and learning, we have discovered new principles and demonstrated the importance of these principles on a variety of tasks. In this chapter, we have used these principles in the design of a novel framework for artificial neuroevolution. We believe that these methods and the principles they conferred will be beneficial for the future of artificial neuroevolution and neural networks in general as these domains continue to elucidate and improve learning.

Bibliography

[Aba+16] Martín Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[Agn+06] Luigi Francesco Agnati et al. "Volume transmission and wiring transmission from cellular to molecular networks: history and perspectives." In: *Acta Physiologica* 187.1-2 (2006), pp. 329–344.

[Agn+10] Luigi F Agnati et al. "Understanding wiring and volume transmission." In: *Brain research reviews* 64.1 (2010), pp. 137–159.

[AMJ06] Harvard L Armus, Amber R Montgomery, and Jenny L Jellison. "Discrimination learning in paramecia (P. caudatum)." In: *The Psychological Record* 56.4 (2006), pp. 489–498.

[Anc00] Lauren W Ancel. "Undermining the Baldwin expediting effect: does phenotypic plasticity accelerate evolution?" In: *Theoretical population biology* 58.4 (2000), pp. 307–319.

[And+16] Marcin Andrychowicz et al. "Learning to learn by gradient descent by gradient descent." In: *Advances in Neural Information Processing Systems*. 2016, pp. 3981–3989.

[Ari+15] Luis Arias-Darraz et al. "A Transient Receptor Potential Ion Channel in Chlamydomonas Shares Key Features with Sensory Transduction-Associated TRP Channels in Mammals." In: *The Plant Cell* 27.1 (2015), pp. 177–188. ISSN: 1040-4651. DOI: 10.1105/tpc.114.131862. URL: http://www.plantcell.org/content/27/1/177.

[Bad+97] Roland Baddeley et al. "Responses of neurons in primary and inferior temporal visual cortices to natural scenes." In: *Proceedings of the Royal Society of London B: Biological Sciences* 264.1389 (1997), pp. 1775–1783.

[Bad09] Alexander V Badyaev. "Evolutionary significance of phenotypic accommodation in novel environments: an empirical test of the Baldwin effect." In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 364.1520 (2009), pp. 1125–1141.

[Bal03] Andrew Balaam. "Developmental Neural Networks for Agents." In: *Advances in Artificial Life, Proceedings of the 7th European Conference on Artificial Life (ECAL 2003)*. Springer, 2003, pp. 154–163.

[Bal96] J Mark Baldwin. "A new factor in evolution." In: *The american naturalist* 30.354 (1896), pp. 441–451.

[Ban+98] Wolfgang Banzhaf et al. *Genetic programming: an introduction*. Vol. 1. Morgan Kaufmann San Francisco, 1998.

[Ban03a] Wolfgang Banzhaf. "Artificial regulatory networks and genetic programming." In: *Genetic programming theory and practice*. Springer, 2003, pp. 43–61.

[Ban03b] Wolfgang Banzhaf. "On the dynamics of an artificial regulatory network." In: *European Conference on Artificial Life*. Ed. by W. Banzhaf et al. New York City, NY, USA: Springer, 2003, pp. 217–227.

[BB11] Parizad M Bilimoria and Azad. Bonni. "Molecular Control of Axon Branching." In: *The Neuroscientist* 19.1 (2011), pp. 16–24. ISSN: 1073-8584.

[BBS95] Egbert JW Boers, Marko V Borst, and Ida G Sprinkhuizen-Kuyper. "Evolving neural networks using the Baldwin effect." In: *Artificial Neural Nets and Genetic Algorithms*. Springer, 1995, pp. 333–336.

[BDK13] Michael Beyeler, Nikil D Dutt, and Jeffrey L Krichmar. "Categorization and decision-making in a neurobiologically plausible spiking network using a STDP-like learning rule." In: *Neural Networks* 48 (2013), pp. 109–124.

[Bel+13] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents." In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[Bey94] Hans-Georg Beyer. "Towards a theory of evolution strategies: Results for (1,+)-strategies on (nearly) arbitrary fitness functions." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 57–67.

[BFM18] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. *Evolutionary computation 1: Basic algorithms and operators*. CRC press, 2018.

[Bon+99] Eric Bonabeau et al. *Swarm intelligence: from natural to artificial systems*. 1. Oxford university press, 1999.

[BP01] Guo-qiang Bi and Mu-ming Poo. "Synaptic modification by correlated activity: Hebb's postulate revisited." In: *Annual review of neuroscience* 24.1 (2001), pp. 139–166.

[Bre+07] Romain Brette et al. "Simulation of networks of spiking neurons: a review of tools and strategies." In: *Journal of computational neuroscience* 23.3 (2007), pp. 349–398.

[BS02] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategiesA comprehensive introduction." In: *Natural computing* 1.1 (2002), pp. 3–52.

[Bul01] John A Bullinaria. "Exploring the Baldwin effect in evolving adaptable control systems." In: *Connectionist models of learning, development and evolution*. Springer, 2001, pp. 231–242.

[Bul09] John A Bullinaria. "Lifetime learning as a factor in life history evolution." In: *Artificial Life* 15.4 (2009), pp. 389–409.

[CB06] SF Cooke and TVP Bliss. "Plasticity in the human central nervous system." In: *Brain* 129.7 (2006), pp. 1659–1673.

[CB16] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. 2016. URL: http://pybullet.org.

[CD08a] Natalia Caporale and Yang Dan. "Spike timingdependent plasticity: a Hebbian learning rule." In: *Annu. Rev. Neurosci.* 31 (2008), pp. 25–46.

[CD08b] Arturo Chavoya and Yves Duthen. "A cell pattern generation model based on an extended artificial regulatory network." In: *BioSystems* 94.1-2 (2008), pp. 95–101. ISSN: 03032647. DOI: 10.1016/j.biosystems.2008.05.015.

[CH15] Sylvain Cussat-Blanc and Kyle Harrington. "Genetically-regulated Neuromodulation Facilitates Multi-Task Reinforcement Learning." In: *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*. New York, New York, USA: ACM Press, 2015, pp. 551–558. ISBN: 978-1-4503-3472-3. DOI: 10.1145/2739480.2754730. URL: http://dl.acm.org/citation.cfm?doid=2739480.2754730.

[Chi06] John K. Chilton. "Molecular mechanisms of axon guidance." In: *Developmental Biology* 292.1 (2006), pp. 13–24. ISSN: 00121606.

[Cho+15] François Chollet et al. *Keras*. 2015. URL: https://keras.io/.

[CHP15] Sylvain Cussat-Blanc, Kyle Harrington, and Jordan Pollack. "Gene regulatory network evolution through augmenting topologies." In: *IEEE Transactions on Evolutionary Computation* 19.6 (2015), pp. 823–837.

[CLD08] S Cussat-Blanc, H Luga, and Yves Duthen. "From single cell to simple creature morphology and metabolism." In: *Artificial Life XI* (2008), pp. 134–141. URL: http://www.cs.bham.ac.uk/%20wbl/biblio/cache/bin/cache.php?alifexi_cussatblanc_134,http___www.alifexi.org_papers_ALIFExi_pp134-141.pdf,http://www.alifexi.org/papers/ALIFExi_pp134-141.pdf.

[Cor+18] Ammon Corl et al. "The genetic basis of adaptation following plastic changes in coloration in a novel environment." In: *Current Biology* 28.18 (2018), pp. 2970–2977.

[CWM07] Janet Clegg, James Alfred Walker, and Julian Frances Miller. "A new crossover technique for Cartesian genetic programming." In: ACM Press, 2007, p. 1580.

[DA01] Peter Dayan and Laurence F Abbott. *Theoretical neuroscience, vol. 806*. Cambridge, MA: MIT Press, 2001.

[Dah+12] George E Dahl et al. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition." In: *IEEE Transactions on audio, speech, and language processing* 20.1 (2012), pp. 30–42.

[DBL06] P. Dwight Kuo, Wolfgang Banzhaf, and André Leier. "Network topology and the evolution of dynamics in an artificial genetic regulatory network model created by whole genome duplication and divergence." In: *BioSystems* 85.3 (2006), pp. 177–200. ISSN: 03032647. DOI: 10.1016/j.biosystems.2006.01.004.

[DC15] Peter U Diehl and Matthew Cook. "Unsupervised learning of digit recognition using spike-timing-dependent plasticity." In: *Frontiers in computational neuroscience* 9 (2015), p. 99.

[DCD15] Jean Disset, Sylvain Cussat-Blanc, and Yves Duthen. "MecaCell: an Open-source Efficient Cellular Physics Engine." In: *13th European Conference on Artificial Life (ECAL 2015)*. MIT Press, 2015, pp–67.

[DCD16] Jean Disset, Sylvain Cussat-Blanc, and Yves Duthen. "Evolved Developmental Strategies of Artificial Multicellular Organisms." In: *Artificial Life XV: Proceedings of Fifteenth International Symposium on the Synthesis and Simulation of Living Systems*. Ed. by C. Gershenson, T. Froese, et al. Cambridge, MA, USA: MIT Press, 2016, pp–1.

[DD18] Benjamin Doerr and Carola Doerr. "Optimal Static and Self-Adjusting Parameter Choices for the $1+(\lambda, \lambda)$ Genetic Algorithm." In: *Algorithmica* 80.5 (May 2018), pp. 1658–1709.

[Deb+02] Kalyanmoy Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[Dis+17] Jean Disset et al. "A comparison of genetic regulatory network dynamics and encoding." In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 91–98.

[DJW02] Stefan Droste, Thomas Jansen, and Ingo Wegener. "On the analysis of the (1+ 1) evolutionary algorithm." In: *Theoretical Computer Science* 276.1-2 (2002), pp. 51–81.

[Dow07] Keith L. Downing. "Supplementing evolutionary developmental systems with abstract models of neurogenesis." In: *Proc. Conf. on Genetic and evolutionary Comp.* 2007, pp. 990–996.

[Dow15] Keith L Downing. *Intelligence emerging: adaptivity and search in evolving neural systems*. MIT Press, 2015.

[Doy02] Kenji Doya. "Metalearning and neuromodulation." In: *Neural Networks* 15.4-6 (2002), pp. 495–506.

[Doy08] Kenji Doya. "Modulators of decision making." In: *Nature neuroscience* 11.4 (2008), p. 410.

[DS14] René Doursat and Carlos Sánchez. "Growing fine-grained multicellular robots." In: *Soft Robotics* 1.2 (2014), pp. 110–121.

[DSN17] Sanjoy Dasgupta, Charles F Stevens, and Saket Navlakha. "A neural algorithm for a fundamental computing problem." In: *Science* 358.6364 (2017), pp. 793–796.

[DY+14] Li Deng, Dong Yu, et al. "Deep learning: methods and applications." In: *Foundations and Trendső in Signal Processing* 7.34 (2014), pp. 197–387.

[ED18] Eric Edsinger and Gül Dölen. "A conserved role for serotonergic neurotransmission in mediating social behavior in octopus." In: *Current Biology* 28.19 (2018), pp. 3136–3142.

[EH07] Lynda Erskine and Eloisa Herrera. "The retinal ganglion cell axon's journey: insights into molecular mechanisms of axon guidance." In: *Developmental biology* 308.1 (2007), pp. 1–14.

[EH12] Brent E Eskridge and Dean F Hougen. "Nurturing promotes the evolution of learning in uncertain environments." In: *Development and Learning and Epigenetic Robotics (ICDL), 2012 IEEE International Conference on*. IEEE, 2012, pp. 1–6.

[Ell13] Kai Olav Ellefsen. "Evolved Sensitive Periods in Learning." In: *ECAL*. 2013, pp. 409–416.

[Ell14] Kai Olav Ellefsen. "The Evolution of Learning Under Environmental Variability." In: *The Evolution of Learning: Balancing Adaptivity and Stability in Artificial Agents* (2014), p. 101.

[EMH18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey." In: *arXiv preprint arXiv:1808.05377* (2018).

[FAL17] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks." In: *International Conference on Machine Learning*. 2017, pp. 1126–1135.

[FB92] K. Fleischer and A.H. Barr. "A Simulation Testbed for the Study of Multicellular Development: The Multiple Mechanisms of Morphogenesis." In: *Artificial Life III: Proceedings of the Workshop on Artificial Life*. Ed. by C. G. Langton. Addison-Wesley Longman, 1992, p. 389.

[FC54] BWAC Farley and W Clark. "Simulation of self-organizing systems by digital computer." In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (1954), pp. 76–84.

[FF07] Michael A. Farries and Adrienne L. Fairhall. "Reinforcement Learning With Modulated Spike Timing-Dependent Synaptic Plasticity." In: *Journal of neurophysiology* 98.6 (2007), pp. 3648–3665.

[FG16] Nicolas Frémaux and Wulfram Gerstner. "Neuromodulated Spike-Timing-Dependent Plasticity, and Theory of Three-Factor Learning Rules." In: *Frontiers in Neural Circuits* 9.January (2016). ISSN: 1662-5110. DOI: 10.3389/fncir.2015.00085. URL: http://journal.frontiersin.org/Article/10.3389/fncir.2015.00085/abstract.

[Fir+17] Orhan Firat et al. "Multi-way, multilingual neural machine translation." In: *Computer Speech & Language* 45 (2017), pp. 236–252.

[Fis36] Ronald A Fisher. "The use of multiple measurements in taxonomic problems." In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[FM91] JF Fontanari and R Meir. "Evolving a learning algorithm for the binary perceptron." In: *Network: Computation in Neural Systems* 2.4 (1991), pp. 353–359.

[FNZ05] Ning Feng, Gangmin Ning, and Xiaoxiang Zheng. "A framework for simulating axon guidance." In: *Neurocomputing* 68.1-4 (2005), pp. 70–84. ISSN: 09252312.

[FOW66] Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. "Artificial intelligence through simulated evolution." In: (1966).

[FPP86] J Doyne Farmer, Norman H Packard, and Alan S Perelson. "The immune system, adaptation, and machine learning." In: *Physica D: Nonlinear Phenomena* 22.1-3 (1986), pp. 187–204.

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[GD91] David E Goldberg and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algorithms." In: *Foundations of genetic algorithms*. Vol. 1. Elsevier, 1991, pp. 69–93.

[GJ09] Simona Ginsburg and Eva Jablonka. "Epigenetic learning in non-neural organisms." In: *Journal of biosciences* 34.4 (2009), p. 633.

[GLR17] Jordan Guerguiev, Timothy P Lillicrap, and Blake A Richards. "Towards deep learning with segregated dendrites." In: *ELife* 6 (2017), e22901.

[GM11] Daniel A Gibson and Le Ma. "Developmental regulation of axon branching in the vertebrate nervous system." In: *Development (Cambridge, England)* 138.2 (2011), pp. 183–195. ISSN: 0950-1991.

[GP13a] B W Goldman and W F Punch. "Length Bias and Search Limitations in Cartesian Genetic Programming." In: *Gecco'13: Proceedings of the 2013 Genetic and Evolutionary Computation Conference* (2013), pp. 932–940.

[GP13b] Brian W Goldman and William F Punch. "Reducing wasted evaluations in cartesian genetic programming." In: *European Conference on Genetic Programming*. Springer, 2013, pp. 61–72.

[GS00] Timothy M. Gomez and Nicholas C. Spitzer. "Common mechanisms underlying growth cone guidance and axon branching." In: *Journal of Neurobiology* 44.2 (2000), pp. 145–158. ISSN: 00223034.

[GW93] Frederic Gruau and Darrell Whitley. "Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect." In: *Evolutionary computation* 1.3 (1993), pp. 213–233.

[Hah+00] Richard HR Hahnloser et al. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit." In: *Nature* 405.6789 (2000), p. 947.

[Han06] Nikolaus Hansen. "The CMA evolution strategy: a comparing review." In: *Towards a new evolutionary computation*. Springer, 2006, pp. 75–102.

[Har+12a] Simon Harding et al. "MT-CGP: Mixed Type Cartesian Genetic Programming." In: *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12* (2012), p. 751.

[Har+12b] Kyle I Harrington et al. "Autoconstructive Evolution for Structural Problems." In: (2012).

[Har+13] Kyle I. Harrington et al. "Robot Coverage Control by Evolved Neuromodulation." In: *The 2013 International Joint Conference on Neural Networks*. Ed. by Plamen Angelov, Daniel Levine, and Peter Erdi. Piscataway, NJ, USA: IEEE, 2013, pp. 1–8. (Visited on 11/29/2016).

[Har08] Simon Harding. "Evolution of image filters on graphics processor units using cartesian genetic programming." In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 2008, pp. 1921–1928.

[Hau+12] Matthew Hausknecht et al. "HyperNEAT-GGP: A HyperNEAT-based Atari general game player." In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 2012, pp. 217–224.

[Hau+14] Matthew Hausknecht et al. "A neuroevolution approach to general atari game playing." In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.4 (2014), pp. 355–366.

[HBM10] Simon Harding, Wolfgang Banzhaf, and Julian F Miller. "A Survey of Self Modifying Cartesian Genetic Programming." In: *Genetic Programming Theory and Practice VIII* 8 (2010), pp. 91–107.

[Heb+49] Donald O Hebb et al. *The organization of behavior*. New York: Wiley, 1949.

[HH52] Alan L Hodgkin and Andrew F Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve." In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[HK18] Jakub Husa and Roman Kalkreuth. "A Comparative Study on Crossover in Cartesian Genetic Programming." In: *European Conference on Genetic Programming*. Springer, 2018, pp. 203–219.

[HLP03] G.S. Hornby, H. Lipson, and Jordan B. Pollack. "Generative Representations for the Automated Design of Modular Physical Robots." In: *IEEE Trans. on Robotics and Automation* 19 (2003), pp. 703–719.

[HLS13] Simon Harding, Jürgen Leitner, and Juergen Schmidhuber. "Cartesian genetic programming for image processing." In: *Genetic programming theory and practice X*. Springer, 2013, pp. 31–44.

[HMB11] Simon Harding, Julian F Miller, and Wolfgang Banzhaf. "SMCGP2: Self Modifying Cartesian Genetic Programming in Two Dimensions." In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (2011), pp. 1491–1498.

[HN87] Geoffrey E Hinton and Steven J Nowlan. "How learning can guide evolution." In: *Complex systems* 1.3 (1987), pp. 495–502.

[HO96] Nikolaus Hansen and Andreas Ostermeier. "Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation." In: *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*. IEEE, 1996, pp. 312–317.

[Hol92a] John H Holland. "Genetic algorithms." In: *Scientific american* 267.1 (1992), pp. 66–73.

[Hol92b] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[Hos+11] Jonas A Hosp et al. "Dopaminergic projections from midbrain to primary motor cortex mediate motor skill learning." In: *Journal of Neuroscience* 31.7 (2011), pp. 2481–2487.

[HR84] James L Hindmarsh and RM Rose. "A model of neuronal bursting using three coupled first order differential equations." In: *Proc. R. Soc. Lond. B* 221.1222 (1984), pp. 87–102.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Hub+09] Andrew D Huberman et al. "Ephrin-As mediate targeting of eye-specific projections to the lateral geniculate nucleus." In: 8.8 (2009), pp. 1013–1021.

[HV99] HGE Hentschel and A Van Ooyen. "Models of axon guidance and bundling during development." In: *Proceedings of the Royal Society of London B: Biological Sciences* 266.1434 (1999), pp. 2231–2238.

[IB18] EJ Izquierdo and RD Beer. "From head to tail: a neuromechanical model of forward locomotion in Caenorhabditis elegans." In: *Philosophical transactions of the Royal Society of London. Series B, Biological sciences* 373.1758 (2018).

[IBM17] Dario Izzo, Francesco Biscani, and Alessio Mereta. "Differentiable Genetic Programming." In: *European Conference on Genetic Programming*. Springer, 2017, pp. 35–51.

[Izh03] Eugene M Izhikevich. "Simple model of spiking neurons." In: *IEEE Transactions on neural networks* 14.6 (2003), pp. 1569–1572.

[Izh04] Eugene M. Izhikevich. "Which model to use for cortical spiking neurons?" In: *IEEE Transactions on Neural Networks* 15.5 (2004), pp. 1063–1070.

[Izh07] Eugene M. Izhikevich. "Solving the distal reward problem through linkage of STDP and dopamine signaling." In: *Cerebral Cortex* 17.10 (2007), pp. 2443–2452.

[JM80] Kristjan R Jessen and Rhona Mirsky. "Glial cells in the enteric nervous system contain glial fibrillary acidic protein." In: *Nature* 286.5774 (1980), p. 736.

[Joa+16] Michal Joachimczak et al. "Spiral autowaves as minimal, distributed gait controllers for soft-bodied animats." In: *Proceedings of the Artificial Life Conference 2016*. MIT Press, 2016, pp. 140–141.

[JS15] Yangqing Jia and E Shelhamer. *Caffe model zoo*. 2015. URL: http://caffe.berkeleyvision.org/model_zoo.html.

[JW09] Micha Joachimczak and Borys Wróbel. "Evolution of the morphology and patterning of artificial embryos: scaling the tricolour problem to the third dimension." In: *Proceedings of the European Conference on Artificial Life*. Ed. by György Kampis, István Karsai, and Eörs Szathmáry. Cambridge, MA, USA: MIT Press, 2009, pp. 35–43.

[JW10a] M.a Joachimczak and B.a b Wróbel. "Processing signals with evolving artificial gene regulatory networks." In: *Artificial Life XII: Proceedings of the 12th International Conference on the Synthesis and Simulation of Living Systems, ALIFE 2010* (2010), pp. 203–210. URL: http://www.scopus.com/

inward/record.url?eid=2-s2.0-84874698466&partnerID=40&md5=b3e8e6139d13bf54e4c80cb7b697d604.

[JW10b] Michal Joachimczak and Borys Wrobel. "Evolving Gene Regulatory Networks for Real Time Control of Foraging Behaviours." In: *Artificial Life XII. Proceedings of the 12th International Conference on the Synthesis and Simulation of Living Systems* (2010), pp. 348–355.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[KH09] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[Khe+16] Saeed Reza Kheradpisheh et al. "STDP-based spiking deep neural networks for object recognition." In: *arXiv preprint arXiv:1611.01421* (2016).

[Kit90] H. Kitano. "Designing neural networks using genetic algorithms with graph generation system." In: *Complex Systems* 4 (1990), pp. 461–476.

[KKM10a] M.M. Khan, G.M. Khan, and J.F. Miller. "Efficient representation of recurrent neural networks for markovian/non-markovian non-linear control problems." In: *Intelligent Systems Design and Applications (ISDA), 2010 10th International Conference on*. IEEE, 2010, pp. 615–620.

[KKM10b] M.M. Khan, G.M. Khan, and J.F. Miller. "Evolution of optimal anns for non-linear control problems using cartesian genetic programming." In: *Proceedings of International Conference on Artificial Intelligence (ICAI 2010)*. 2010.

[KLH11] Katherine Kalil, Li Li, and B. Ian Hutchins. "Signaling Mechanisms in Cortical Axon Growth, Guidance, and Branching." In: *Frontiers in Neuroanatomy* 5.September (2011), pp. 1–15. ISSN: 1662-5129.

[KMH07] G.M. Khan, J.F. Miller, and D.M. Halliday. "A developmental model of neural computation using cartesian genetic programming." In: *Proc. Conf. on Genetic And Evolutionary Computation*. Vol. 7. 2007, pp. 2535–2542.

[KMH08] G.M. Khan, J.F. Miller, and D.M. Halliday. "Developing neural structure of two agents that play checkers using Cartesian Genetic Programming." In: *Proc. Conf. on Genetic and evolutionary computation*. ACM, 2008, pp. 2169–2174.

[KMH11] Gul Muhammad Khan, Julian F Miller, and David M Halliday. "Evolution of cartesian genetic programs for development of learning neural architecture." In: *Evolutionary computation* 19.3 (2011), pp. 469–523.

[Kna+06] Johannes Knabe et al. "Evolving Biological Clocks using Genetic Regulatory Networks." In: *Artificial Life X : Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems* Alife 10 (2006), pp. 15–21. URL: c:/Daniel/Work/Library/workLibrary.Data/PDF/2088839662/Knabe2006.pdf%5Cnhttp://panmental.de/GRNclocks/.

[Koz94] John R Koza. "Genetic programming as a means for programming computers by natural selection." In: *Statistics and computing* 4.2 (1994), pp. 87–112.

[Kri+12] Alex Krizhevsky et al. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information and Processing Systems (NIPS)* (2012), pp. 1–9. ISSN: 10495258.

[KS08] Marie T. Killeen and Stephanie S. Sybingco. "Netrin, Slit and Wnt receptors allow axons to choose the axis of migration." In: *Developmental Biology* 323.2 (2008), pp. 143–151. ISSN: 00121606.

[Lap07] Louis Lapicque. "Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization." In: *Journal de Physiologie et de Pathologie Generalej* 9 (1907), pp. 620–635.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), p. 436.

[LBV15] Quoc V Le, Google Brain, and Mountain View. "A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks." In: (2015), pp. 1–20.

[LCP13] Tommy L Lewis, Julien Courchet, and Franck Polleux. "Cell biology in neuroscience: Cellular and molecular mechanisms underlying axon formation, growth, and branching." In: *The Journal of cell biology* 202.6 (2013), pp. 837–48. ISSN: 1540-8140.

[LDP16] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. "Training Deep Spiking Neural Networks Using Backpropagation." In: *Frontiers in Neuroscience* 10 (2016), p. 508. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00508. URL: https://www.frontiersin.org/article/10.3389/fnins.2016.00508.

[LeC+98] Yann LeCun et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Lil+16] Timothy P Lillicrap et al. "Random synaptic feedback weights support error backpropagation for deep learning." In: *Nature communications* 7 (2016), p. 13276.

[LKS06] J. J. Letzkus, B. M. Kampa, and G. J. Stuart. "Learning Rules for Spike Timing-Dependent Plasticity Depend on Dendritic Synapse Location." In: *Journal of Neuroscience* 26.41 (2006), pp. 10420–10429.

[Lóp+16] Manuel López-ibáñez et al. "The irace package : Iterated racing for automatic algorithm configuration." In: 3 (2016), pp. 43–58.

[LPM08] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. "A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback." In: *PLoS Computational Biology* 4.10 (2008).

[LS08] Joel Lehman and Kenneth O Stanley. "Exploiting open-endedness to solve problems through the search for novelty." In: *ALIFE*. 2008, pp. 329–336.

[LS97] Sean Luke and Lee Spector. "A Revised Comparison of Crossover and Mutation in Genetic Programming." In: (1997).

[Mag+00] E. A. Maguire et al. "Navigation-related structural change in the hippocampi of taxi drivers." In: *PNAS* 97 (2000), pp. 4398–4403.

[Mah+13] Maryam Mahsal Khan et al. "Fast learning neural networks using Cartesian genetic programming." In: *Neurocomputing* 121 (2013), pp. 274–289.

[Men12] Randolf Menzel. "The honeybee as a model for understanding the basis of cognition." In: *Nature Reviews Neuroscience* 13.11 (2012), p. 758.

[Mii+18] Risto Miikkulainen et al. "Evolving deep neural networks." In: *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 2018, pp. 293–312.

[Mil01] J. F. Miller. "What bloat? Cartesian Genetic Programming on Boolean problems." In: *Proc. Conf. Genetic and Evolutionary Computation, Late breaking papers*. 2001, pp. 295–302.

[Mil04] J.F. Miller. "Evolving a Self-Repairing, Self-Regulating, French Flag Organism." In: *Genetic and Evolutionary Computation Conference (GECCO'04)*. New York City, NY, USA: Springer, 2004, pp. 129–139.

[Mil11] Julian F Miller. *Cartesian Genetic Programming*. Springer Berlin Heidelberg, 2011.

[ML81] Catherine Morris and Harold Lecar. "Voltage oscillations in the barnacle giant muscle fiber." In: *Biophysical journal* 35.1 (1981), pp. 193–213.

[MM97] David E Moriarty and Risto Miikkulainen. "Forming neural networks through efficient and adaptive coevolution." In: *Evolutionary Computation* 5.4 (1997), pp. 373–399.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836.

[Moz+18a] Milad Mozafari et al. "Combining STDP and Reward-Modulated STDP in Deep Convolutional Spiking Neural Networks for Digit Recognition." In: *arXiv preprint arXiv:1804.00227* (2018).

[Moz+18b] Milad Mozafari et al. "First-Spike-Based Visual Categorization Using Reward-Modulated STDP." In: *IEEE Transactions on Neural Networks and Learning Systems* (2018).

[MP69] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 1969.

[MS06] J. F. Miller and S. L. Smith. "Redundancy and computational efficiency in Cartesian Genetic Programming." In: *IEEE Trans. on Evolutionary Computation* 10.2 (2006), pp. 167–174.

[MT00] J. F. Miller and P. Thomson. "Cartesian Genetic Programming." In: *Proc. European Conf. on Genetic Programming*. Vol. 10802. LNCS. 2000, pp. 121–132.

[MW03] Héctor A Montes and Jeremy L Wyatt. "Cartesian Genetic Programming for Image Processing Tasks." In: *Neural Networks and Computational Intelligence*. Citeseer, 2003, pp. 185–190.

[MW17] Julian F Miller and Dennis G Wilson. "A developmental artificial neural network model for solving multiple problems." In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 2017, pp. 69–70.

[MWC18] J. F. Miller, D. G. Wilson, and S. Cussat-Blanc. "Evolving developmental programs that build neural networks for solving multiple problems." In: *Genetic Programming Theory and Practice XV*. Ed. by W. Banzhaf, W. Tozier, and W. Worzel. Springer, 2018, TBC.

[Nes83] Yurii E Nesterov. "A method for solving the convex programming problem with convergence rate O($1/k^2$)." In: *Dokl. Akad. Nauk SSSR*. Vol. 269. 1983, pp. 543–547.

[NF99] Stefano Nolfi and Dario Floreano. "Learning and evolution." In: *Autonomous robots* 7.1 (1999), pp. 89–113.

[NSB10] Miguel Nicolau, Marc Schoenauer, and Wolfgang Banzhaf. "Evolving genes to balance a pole." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6021 LNCS (2010), pp. 196–207. ISSN: 03029743. DOI: 10.1007/978-3-642-12148-7-17.

[OL16] Charles Ofria and Alexander Lalejini. "The evolutionary origins of phenotypic plasticity." In: *Proceedings of the Artificial Life Conference 2016 13*. MIT Press, 2016, pp. 372–379.

[OSH87] IM Oliver, DJd Smith, and John RC Holland. "Study of permutation crossover operators on the traveling salesman problem." In: *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987., 1987.

[PB15] Marco Pignatelli and Antonello Bonci. "Role of Dopamine Neurons in Reward and Aversion: A Synaptic Plasticity Perspective." In: *Neuron* 86.5 (2015), pp. 1145–1157.

[Pfe+07] Cory Pfeiffenberger et al. "Ephrin-As and neural activity are required for eye-specific patterning during retinogeniculate mapping." In: 8.8 (2007), pp. 1022–1027.

[PL98] Riccardo Poli and William B Langdon. "On the search properties of different crossover operators in genetic programming." In: *Genetic Programming* (1998), pp. 293–301.

[PNA09] Gertrudis Perea, Marta Navarrete, and Alfonso Araque. "Tripartite synapses: astrocytes process and control synaptic information." In: *Trends in Neurosciences* 32.8 (2009), pp. 421–431.

[Pol+08] Riccardo Poli et al. *A field guide to genetic programming*. Lulu. com, 2008.

[Pol+97] Riccardo Poli et al. "Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming." In: *ICGA*. Citeseer, 1997, pp. 346–353.

[Pon05] Filip Ponulak. "ReSuMe-new supervised learning method for Spiking Neural Networks." In: *Inst. Control Information Engineering, Poznan Univ.* 22.2 (2005), pp. 467–510.

[Por+11] Ana B. Porto-Pazos et al. "Artificial astrocytes improve neural network performance." In: *PLoS ONE* 6.4 (2011), pp. 1–8.

[PPN15] Paulo Cesar Donizeti Paris, Emerson Carlos Pedrino, and MC Nicoletti. "Automatic learning of image filters using Cartesian genetic programming." In: *Integrated Computer-Aided Engineering* 22.2 (2015), pp. 135–151.

[RBT00] M C van Rossum, G Q Bi, and G G Turrigiano. "Stable Hebbian learning from spike timing-dependent plasticity." In: *The Journal of neuroscience : the official journal of the Society for Neuroscience* 20.23 (2000), pp. 8812–8821.

[RCN98] C. Ryan, J. J. Collins, and M. O. Neill. "Grammatical evolution: Evolving programs for an arbitrary language." In: *Genetic Programming*. Ed. by Wolfgang Banzhaf et al. Springer Berlin Heidelberg, 1998, pp. 83–96.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." In: *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[RH89] RM Rose and JL Hindmarsh. "The assembly of ionic currents in a thalamic neuron I. The three-dimensional model." In: *Proc. R. Soc. Lond. B* 237.1288 (1989), pp. 267–288.

[RHS10] Sebastian Risi, Charles E Hughes, and Kenneth O Stanley. "Evolving plastic neural networks with novelty search." In: *Adaptive Behavior* 18.6 (2010), pp. 470–491.

[RLS10] Sebastian Risi, Joel Lehman, and Kenneth O. Stanley. "Evolving the Placement and Density of Neurons in the HyperNEAT Substrate." In: *Proc. Conf. on Genetic and Evolutionary Computation*. 2010, pp. 563–570.

[RMM98] Norman Richards, David E Moriarty, and Risto Miikkulainen. "Evolving neural networks to play Go." In: *Applied Intelligence* 8.1 (1998), pp. 85–96.

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[RS94] Edmund Ronald and Marc Schoenauer. "Genetic Lander: An experiment in accurate neuro-genetic control." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 452–461.

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization algorithms." In: *CoRR* abs/1609.04747 (2016). URL: http://arxiv.org/abs/1609.04747.

[Rus+16] Andrei A Rusu et al. "Progressive neural networks." In: *arXiv preprint arXiv:1606.04671* (2016).

[Sal+17] T. Salismans et al. "Evolution Strategies as a scalable Alternative to Reinforcement Learning." In: *ArXiv* 170303864v2 (2017), pp. 1–13.

[SBT09] Hooman Shayani, Peter J. Bentley, and Andrew M. Tyrrell. "A multi-cellular developmental representation for evolution of adaptive spiking neural microcircuits in an FPGA." In: *Proceedings - 2009 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009* January 2016 (2009), pp. 3–10.

[SC14] Stéphane Sanchez and Sylvain Cussat-Blanc. "Gene regulated car driving: using a gene regulatory network to drive a virtual car." In: *Genetic Programming and Evolvable Machines* 15.4 (2014), pp. 477–511.

[Sch91] Jürgen Schmidhuber. "Curious model-building control systems." In: *Neural Networks, 1991. 1991 IEEE International Joint Conference on*. IEEE, 1991, pp. 1458–1463.

[Sch92] Jürgen Schmidhuber. "Learning complex, extended sequences using the principle of history compression." In: *Neural Computation* 4.2 (1992), pp. 234–242.

[SD03] Nicolas Schweighofer and Kenji Doya. "Meta-learning in reinforcement learning." In: *Neural Networks* 16.1 (2003), pp. 5–9.

[SDG09] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. "A hypercube-based encoding for evolving large-scale neural networks." In: *Artificial life* 15.2 (2009), pp. 185–212.

[See+95] Peter H Seeburg et al. "The NMDA receptor channel: molecular design of a coincidence detector." In: *Proceedings of the 1993 Laurentian Hormone Conference*. Elsevier, 1995, pp. 19–34.

[Ser+07] Thomas Serre et al. "Robust object recognition with cortex-like mechanisms." In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 3 (2007), pp. 411–426.

[SG10] Susan R Sesack and Anthony A Grace. "Cortico-basal ganglia reward network: microcircuitry." In: *Neuropsychopharmacology* 35.1 (2010), p. 27.

[Sha10] Lawrence Shapiro. *Embodied cognition*. Routledge, 2010.

[Sin+10] Satinder Singh et al. "Intrinsically motivated reinforcement learning: An evolutionary perspective." In: *IEEE Transactions on Autonomous Mental Development* 2.2 (2010), pp. 70–82.

[Ski06] F. K. Skinner. "Conductance-based models." In: *Scholarpedia* 1.11 (2006), p. 1408. DOI: 10.4249/scholarpedia.1408.

[SL15] Peter Sterling and Simon Laughlin. *Principles of neural design*. MIT Press, 2015.

[SM02] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies." In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[SMA00] S Song, K D Miller, and L F Abbott. "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity." In: *Nature neuroscience* 3.9 (2000), pp. 919–926.

[SMB10] Dominik Scherer, Andreas Müller, and Sven Behnke. "Evaluation of pooling operations in convolutional architectures for object recognition." In: *Artificial Neural NetworksICANN 2010*. Springer, 2010, pp. 92–101.

[Smi+00] Gregory D Smith et al. "Fourier analysis of sinusoidally driven thalamocortical relay neurons and a minimal integrate-and-fire-or-burst model." In: *Journal of Neurophysiology* 83.1 (2000), pp. 588–610.

[SOS06] Shy Shoham, Daniel H OConnor, and Ronen Segev. "How silent is the brain: is there a dark matter problem in neuroscience?" In: *Journal of Comparative Physiology A* 192.8 (2006), pp. 777–784.

[Spe02] L E E Spector. "Genetic Programming and Autoconstructive Evolution with the Push Programming Language." In: (2002), pp. 7–40.

[Spr08] Nelson Spruston. "Pyramidal neurons: dendritic structure and synaptic integration." In: *Nature Reviews Neuroscience* 9.3 (2008), p. 206.

[SSF15] Mauro Santos, Eörs Szathmáry, and José F Fontanari. "Phenotypic plasticity, the baldwin effect, and the speeding up of evolution: The computational roots of an illusion." In: *Journal of theoretical biology* 371 (2015), pp. 127–136.

[SSG02] Hongjun Song, Charles F Stevens, and Fred H Gage. "Astroglia induce neurogenesis from adult neural stem cells." In: *Nature* 417.6884 (2002), p. 39.

[SSR18] Andrea Soltoggio, Kenneth O Stanley, and Sebastian Risi. "Born to learn: The inspiration, progress, and future of evolved plastic artificial neural networks." In: *Neural Networks* (2018).

[Sta07] K. O. Stanley. "Compositional pattern producing networks: A novel abstraction of development." In: *Genetic Programming and Evolvable Machines* 8 (2007), pp. 131–162.

[Sti+14] Marcel Stimberg et al. "Equation-oriented specification of neural models for simulations." In: *Frontiers in neuroinformatics* 8 (2014), p. 6.

[SW15] C. Stangor and J. Walinga. *Introduction to Psychology*. [BC open textbook collection]. Flat World Knowledge, L.L.C., 2015. ISBN: 978-1-936126-49-1. URL: https://books.google.fr/books?id=uKeQSQAACAAJ.

[SZ15] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." In: *ICLR* (2015).

[SZB18] Jörg Stork, Martin Zaefferer, and Thomas Bartz-Beielstein. "Distance-based Kernels for Surrogate Model-based Neuroevolution." In: *arXiv preprint arXiv:1807.07839* (2018).

[Tek+12] Sundeep Teki et al. "Navigating the auditory scene: an expert role for the hippocampus." In: *Journal of Neuroscience* 32.35 (2012), pp. 12251–12257.

[TH18] T Tieleman and G Hinton. *Divide the gradient by a running average of its recent magnitude*. 2018. URL: https://zh.%20coursera.org/learn/neuralnetworks/lecture/YQHki/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude (visited on 10/26/2018).

[TM14] Andrew James Turner and Julian Francis Miller. "Recurrent cartesian genetic programming." In: *International Conference on Parallel Problem Solving from Nature*. Springer, 2014, pp. 476–486.

[Tre+10] Martin A Trefzer et al. "Evolution and analysis of a robot controller based on a gene regulatory network." In: *International Conference on Evolvable Systems*. Ed. by Gianluca Tempesti, Andy M. Tyrrell, and Julian F. Miller. New York City, NY, USA: Springer, 2010, pp. 61–72.

[Tsi00] Joe Z Tsien. "Linking Hebbs coincidence-detection to memory formation." In: *Current opinion in neurobiology* 10.2 (2000), pp. 266–273.

[Tur09] Alan M Turing. "Computing machinery and intelligence." In: *Parsing the Turing Test*. Springer, 2009, pp. 23–65.

[Val71] F. Valverde. "Rate and extent of recovery from dark rearing in the visual cortex of the mouse." In: *Brain Res.* 33 (1971), pp. 1–11.

[VC17] Roby Velez and Jeff Clune. "Diffusion-based neuromodulation can eliminate catastrophic forgetting in simple neural networks." In: *PloS one* 12.11 (2017), e0187736.

[WA12] Borys Wróbel and Ahmed Abdelmotaleb. "Evolving Spiking Neural Networks in the GReaNs (Gene Regulatory evolving artificial Networks) Plaftorm." In: *EvoNet2012: Evolving Networks, from Systems/Synthetic Biology to Computational Neuroscience Workshop at Artificial Life XIII* (2012), pp. 19–22.

[WAJ14] Borys Wrobel, Ahmed Abdelmotaleb, and Michal Joachimczak. "Evolving networks processing signals with a mixed paradigm, inspired by gene regulatory networks and spiking neurons." In: *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST* 134 (2014), pp. 135–149. ISSN: 18678211. DOI: 10.1007/978-3-319-06944-9_10.

[Wer74] Paul Werbos. "Beyond Regression:" New Tools for Prediction and Analysis in the Behavioral Sciences." In: *Ph. D. dissertation, Harvard University* (1974).

[Wil+13] Dennis Wilson et al. "On learning to generate wind farm layouts." In: *Fifteenth annual conference on Genetic and evolutionary computation conference* (2013), pp. 767–774. DOI: 10.1145/2463372.2463462. URL: http://dl.acm.org/citation.cfm?id=2463372.2463462.

[Wil+14] Dennis Wilson et al. "A continuous developmental model for wind farm layout optimization." In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 745–752.

[Wil+17] Dennis G Wilson et al. "Learning aquatic locomotion with animats." In: *Artificial Life Conference Proceedings 14*. MIT Press, 2017, pp. 585–592.

[Wil+18a] Dennis G Wilson et al. "Evolving Differentiable Gene Regulatory Networks." In: *arXiv preprint arXiv:1807.05948* (2018).

[Wil+18b] Dennis G Wilson et al. "Evolving simple programs for playing Atari games." In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2018.

[Wil+18c] DG Wilson et al. "Positional Cartesian Genetic Programming." In: *arXiv preprint arXiv:1810.04119* (2018).

[WJ14] Borys Wróbel and Michal Joachimczak. "Using the Genetic Regulatory evolving Artificial Networks (GReaNs) platform for signal processing, animat control, and artificial multicellular development." In: *Growing Adaptive Machines*. New York City, NY, USA: Springer, 2014, pp. 187–200.

[WM06] James Alfred Walker and Julian Francis Miller. "Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems." In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, pp. 911–918.

[Woo88] David C Wood. "Habituation in Stentor: a response-dependent process." In: *Journal of Neuroscience* 8.7 (1988), pp. 2248–2253.

[Zei12] Matthew D Zeiler. "ADADELTA: an adaptive learning rate method." In: *arXiv preprint arXiv:1212.5701* (2012).