UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EVOLVING SPIKING NEURAL NETWORKS WITH NEAT

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

BY

MARY G HIRSCH
Norman, Oklahoma
2020

EVOLVING SPIKING NEURAL NETWORKS WITH NEAT

A MASTER'S THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Dean Hougen

Dr. Dimitrios Diochnos

Dr. Andrew Fagg

Dr. Christan Grant

Abstract

Spiking neural networks (SNNs) attempt to computationally model biological neurons. While similar to artificial neural networks (ANNs), SNNs preserve the temporal and binary aspects of neurons. Computational evolution is also a biologically inspired computing method, and it has been used to evolve neural networks. NeuroEvolution of Augmenting Topologies (NEAT) is a method to simultaneously evolve the structure and weights of a ANNs. In this work, I apply the NEAT algorithm to SNNs. I compare the performance of ANNs evolved with NEAT and SNNs evolved with NEAT on XOR, a cosine function, and the single pole balancing problem. Multiple values are used for the compatibility threshold (3 options), compatibility weight coefficient (2 options), compatibility disjoint coefficient (2 options), and spiking threshold (2 options). On the XOR problem, 15 SNNs with different parameter combinations found solutions on all five test repetitions while only two ANN parameter combinations did. On the cosine problem, only one SNN parameter combination found a solution on every repetition, but all ANNs did. However, the successful SNNs appeared to capture more of the nonlinearity of the cosine curve than the ANNs. On the single pole balancing problem, no SNNs found any solution while many ANNs were able to find solutions on multiple repetitions. The results indicate that SNNs evolved with NEAT can solve and perform comparably to ANNs evolved with NEAT on

some problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Evolutionary computing and artificial neural networks are both computational ways to represent and use biological ideas. Evolutionary computing uses the ideas of natural evolution, survival of the fittest, reproduction, and mutation, to find solutions to problems. Artificial neural networks (ANNs) and spiking neural networks (SNNs) are inspired by the organization and mechanisms of the brain. ANNs create networks of artificial neurons and synapses using real-valued weights to represent the strength of the synapse and mathematical functions to determine the output of a neuron. Spiking neural networks also use mathematical functions to determine neuron output, but rather than real-valued weights they use binary spikes over a time interval.

Neural networks have a number of factors that determine their success. Two of these are their weights and their architecture. Both of these aspects must be well-tuned in order for networks to have low errors on problems. However, both of these can be difficult, if not impossible, to tune by hand. Hebb [15] presented a method for finding weight values of spiking neural networks without manual calculation. The method, known as Hebbian Learning, laid the ground-

work for algorithms to calculate weight values for both ANNs and SNNs. These algorithms are broadly called *learning algorithms*. Networks are *trained* by repeatedly slightly altering the weight values in ways that slowly decrease the error of the network.

One of the most popular learning algorithms for ANNs today is called *backpropagation*. Introduced in Rumelhart et al. [33], backpropagation changes each weight by using partial derivatives to calculate their effect on the output value. However, because backpropagation relies on derivatives, the calculations the network does must be differentiable, which is not the case in SNNs. There have been many learning methods created specifically for SNNs because traditional backpropagation does not work [1, 31, 11, 8, 27].

Network performance is also highly reliant on the network structure (its *architecture* or *topology*). Like weights, well-performing architectures can be difficult to design by hand. Unlike weights, however, there does not exist a prevailing method to learn them. While methods exist [5, 6, 19], they have not been as widely adopted as backpropagation.

Evolution has also been used to find both network topologies and weights. Because evolution does not rely on derivatives, it can be used on both differentiable and non-differentiable functions. This allows for it to be applied to learn weights for SNNs as well as ANNs. When evolution is used to evolve weights (or other factors) or a network without changing the network architecture, the architecture is considered *static*. A popular method for evolving both the architecture and weights of a network together is NeuroEvolution of Augmenting Topologies (NEAT) [37]. NEAT and other methods for evolving ANNs and SNNs will be discussed in Chapters 4 and 5.

This work looks specifically at evolving SNNs with NEAT. The exploration

and use of this combination has not been extensive, but has been promising. NEAT with SNNs has been compared to static-architecture SNNs on classification problems with the evolved networks having a lower error by about 10% [30]. NEAT with SNNs has also been applied to control problems, and when compared to NEAT with ANNs, found solutions more often and in fewer generations [32]. These works will be discussed further in Chapter 5. NEAT with SNNs has been demonstrated to be successful on both classification and control problems on neuromorphic hardware, hardware specifically designed to run spiking neural networks [42]. (While Vandesompele et al. [42] provides further demonstration of the success of NEAT and SNNs, it provides little discussion of the algorithm, focusing instead on the hardware, which is out of scope of this work.)

This work explores the use of the NEAT algorithm in evolving spiking neural networks. I apply this method to solve the XOR problem, a cosine function, and the single pole balancing problem. These problems are used to show the algorithm's performance on supervised learning problems (binary classification and multi-class classification) and reinforcement learning problems. (I consider cosine as a multi-class classification problem because the experiments use a discretized version as discussed in Section 6.2.2). I intend to provide a proof of concept of evolving SNNs with NEAT through direct comparisons with ANNs with NEAT on these simple problems. I use multiple types of neurons and a range of parameter combinations to allow for comparison and discussion. Such a parameter evaluation is not performed in previous works. This thesis lays groundwork for future work with SNNs and NEAT, providing results for simple problems across a range of parameters.

I find that SNNs evolved with NEAT are able to solve and perform comparably to evolved ANNs on XOR and the cosine function. However, SNNs were unable

to find solutions to the single pole balancing problem while ANNs were. The compatibility threshold and genomic distance parameters are demonstrated to direct effect the number of population extinctions. The spiking threshold for the SNNs appeared to have little effect.

The contributions of this work are: 1. provide further research showing the evolution of SNNs with NEAT to be viable for binary classification tasks; 2. provide evidence that NEAT and SNNs are capable of solving multi-class classification problems; 3. add to research regarding SNNs and NEAT performing on reinforcement learning problems, despite negative results; and 4. provide an initial study of the effects of the compatibility threshold, genomic distance parameters, and the spiking threshold, on NEAT with SNNs.

This work is organized as follows. The next chapter will give a brief introduction to neural networks. Chapter 3 will look at the basics of artificial evolution, going over its main components. Chapters 4 and 5 will discuss previous works regarding the evolution of ANNs and SNNs, respectively. Chapter 6 will cover the methodology of this work, followed by the results in Chapter 7, discussion in Chapter 8, and conclusions in Chapter 9.

Chapter 2

Neural Networks

2.1 Biological Basis for Neural Networks

Animal brains are complex organs capable of complex intense loads of computation. Among other things, human brains are made of an estimated 100 billion of neurons. *Neurons* are nerve cells that communicate and convey information to other neurons. Neurons are made of *dendrites*, a *soma*, an *axon*, and *terminal branches* [22]. The terminal branches connect to other neuron's dendrites. Neurons communicate by sending neurotransmitters to each other through their axons to connected neuron's dendrites. Because of ion imbalances between the inside and outside of the neuron, neurons naturally have a small charge, or *resting potential*. Further potential is generated by interactions ions flowing into and out of the cell membrane [24]. These interactions can cause an increase in *membrane potential*, the electric charge of the cell. As the membrane potential increases, it causes more ionic movement. A high charge can affect the potential of nearby points in the membrane, which can then trigger ion flow at those neighboring locations, and the cycle propagates down the synapse [17]. This is called

a *spike*. Eventually, the membrane reaches its maximum charge, the ions flow back outside the membrane and the charge returns to its resting potential [24]. When returning to the resting potential, the neuron goes through a refractory period. During a *refractory period*, the neuron cannot emit another spike. A series of spikes from one neuron over a time period is called a *spike train*. When discussing a spike going from neuron to neuron, the neuron that emits the spike is called the *presynaptic* neuron, and the neuron that receives the spike is called the *postsynaptic* neuron.

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) consist of neurons (or *nodes*) connected together. Neurons take input values, apply a computation, and output a value. The outputs of some neurons then become the inputs to others. *Input neurons* take user input and values are passed through the network until they reach *output neurons* that give a final output. This section will review some basic information about ANNs.

2.2.1 ANN's Relationship to Biology

Artificial neural networks are inspired by the neurons and connections in the brain. However, because of the complexity of the biological networks, ANNs have abstracted many of the details and are more computationally efficient. They remove the temporal aspect of spiking and change the binary input signals to real numbers. A neuron simultaneously receives a real number value from all the neurons that feed to it. These values are summed and sent through an activation function to give the output of the neuron.

2.2.2 ANN Methods

Artificial neural networks are comprised of weights and activation functions. *Weights* are values assigned to the connections between neurons. The weight between neuron i and neuron j is represented as w_{ij}. The output value of neuron i is multiplied by w_{ij} before being passed to neuron j. *Activation functions* are the functions that neurons use to transform input values to output values. Some popular functions include sigmoid, hyperbolic tangent (tanh), and rectified linear units (ReLu). The output y of a neuron j with a single input x from neuron i using activation function f can then be represented as

$$y = f(w_{i,j}x)\,. \tag{2.1}$$

The output of a node with multiple inputs x_j can be calculated by summing over the inputs. The output of a neuron with J inputs can be calculated with

$$y = \sum_{j=0}^{j=J} f(w_{i,j}x_j)\,.$$

Traditional networks are usually organized into layers. In this case, the layer of neurons that receive input is called the *input layer*, the layer that produces the final output is called the *output layer*, and any layers in between are called *hidden layers*. For example, in a 3-layer network, the neurons in layer 1 receive the initial input and only output to the neurons in layer 2. Layer 2 neurons only output to layer 3 neurons, which then gives the final output. There are no cross connections between neurons in the same layer and no looped connections from a later layer to a previous layer. However, these are not strict rules. Many of successful networks to date have connections that skip over layers [2, 41]. There

are also networks that have connections that loop backwards, called Recurrent Neural Networks (RNNs) [7].

Equation 2.1 can be expanded to the computation of an entire layer using matrix multiplication. Vectors are used to represent the inputs and outputs, and a matrix is used to represent the weights. Each row in the weight matrix contains the weight values from a single input neuron to all output neurons. The output vector y_l of layer l with inputs equal to the output vector y_k from the previous layer k and weight matrix $W_{l,k}$ can be computed with

$$y_l = f(W_{l,k} \times y_k)\,. \tag{2.2}$$

Using Equation 2.2, the output of previously discussed three-layer network can be calculated. Given an input vector x to layer 1, a hidden layer 2, and output layer 3, the final output, y_3 can be calculated with

$$y_3 = f_3\left(W_{2,3} \times \left(f_2\left(W_{1,2} \times f_1\left(W_{in,1} \times x\right)\right)\right)\right)\,. \tag{2.3}$$

Any function can be used as an activation function. However, using a linear function will across multiple layers is computationally equivalent to a single linear function, which eliminates the functionality of multiple layers and unnecessarily increases computation, so it is important to choose a non-linear activation function.

These weights can be hand-designed; however, more often they are determined through a learning process. A *learning* or *training* process is a process of computationally determining the weight values for a network that will optimize performance on a problem.

In order to perform training, the network needs a function E for calculated the

error of the network. In the case of supervised learning, networks are generally trained with a dataset mapping inputs to desired outputs. The error can then be calculated by comparing the network output with the desired output. There are many different methods for calculating error. Mean-squared error, which is average squared difference between the desired label and output label, is a common error function. In reinforcement learning, the error function is often a function of the environment. For example, in the single pole balancing problem, the error is defined as the difference between the maximum possible time to balance the pole and the actual time the network balanced the pole. The training process then changes the weights to minimize this error.

Backpropagation is the predominate method to train the weights of ANNs. Weight updates in backpropagation can be divided into two main steps: the forward pass and the backwards pass. In the *forward pass*, the network output and error are computed with the current weights, like Equation 2.3, and the error is computed. In the *backwards pass*, the partial derivative of the error with respect to each weight, $\frac{\partial E}{\partial w_{i,j}}$, is calculated using the chain rule. Weights are then updated proportional to their effect on the error:

$$\Delta w_{i,j} = -\eta \frac{\partial E}{\partial w_{i,j}} .$$

η is called the *learning rate*, and is a hyperparameter that controls the magnitude of the changes. The change here is negative in order to minimize the error function.

Evolution can also be used to train ANNs. This is explored in Chapter 4.

2.3 Spiking Neural Networks

Like ANNs, spiking neural networks are modeled after brain circuits. Spiking neural networks differ from ANNs because they retain the temporal aspect of the biological circuits. Rather than receiving all inputs from previous neurons at once, neurons receive inputs over a period of time. Neurons then have a function to determine whether to output a value. The following sections will introduce prominent neuron methods and ways to convert real valued input into spikes.

2.3.1 Types of Spiking Neurons

There are many types of spiking neurons that compute the method of determining an output spike in different ways. Here I will discuss four common types of neurons: Hodgkin-Huxley neurons, Leaky Integrate-and-Fire (LIF) neurons, the Spike Response Model, and Izhikevich neurons.

Hodgkin-Huxley

In 1952, Hodgkin and Huxley [16] created a set of equations based on an electrical circuit representation to model the current in the membrane of a giant squid axon. They present a series of differentiable equations modelling the ion channels of sodium (Na) and potassium (K) as well as a leakage (L) channel. Each channel has a conductance g, maximum conductance $\overline{g}$, a resistance R, and voltage, or reversal potential, V. The conductance of the sodium and potassium channels are dependent on gating variables m, n, and h. The gating variables are used to model whether the ion channel is open or closed. Together m and h control the sodium channel and n controls the potassium channel. If the sodium or potassium channels are open, they reach their maximum conductances of $\overline{g}_{Na}$

and $\overline{g}_K$, respectively.

The local conductance g of each channel is the inverse of its resistance R. This gives $g_{Na} = \frac{1}{R_{Na}}$, $g_K = \frac{1}{R_K}$, and $g_L = \frac{1}{R_L}$. Using Ohm's Law, the local current I can be written as $I_L = g_L(v - V_L)$, $I_{Na} = g_{Na}(v - V_{Na})$, and $I_K = g_K(v - V_K)$. Summing these provides the total current into the neuron:

$$I = g_{Na}(v - V_{Na}) + g_K(v - V_K) + g_L(v - V_L)\,. \tag{2.4}$$

The resistance of the leaky channel is constant, so the conductance g_L is always equal to its maximum conductance $\overline{g}_L$. The sodium and potassium resistances, however, depend on the gating constants. Hodgkin and Huxley [16] defines the current conductance g in terms of gating values as $g_{Na} = \overline{g}_{Na}m^3h$ and $g_K = \overline{g}_K n^4$. This gives the sodium and potassium conductances as $I_{Na} = \overline{g}_{Na}m^3h$ and $I_K = \overline{g}_K n^4$ respectively. Substituting these values into Equation 2.4 yields

$$I = \overline{g}_{Na}m^3h(v - V_{Na}) + \overline{g}_k n^4(v - V_K) + \overline{g}_l(v - V_l)\,. \tag{2.5}$$

The gating values are changed through differential equations dependent on two parameters α and β with separate values for each ion channel. The initial values of the gating values, as well as values for the α and β values can be chosen to allow the model to mimic different types of human neurons.

The refractory period of a neuron is not separately addressed in Hodgkin-Huxley neurons. Rather, it is modelled using the gating variables.

This model is biologically plausible because of its basis in the specific ion channels. However, it requires a high amount of computational time because of the complexity and number of differential equations.

Leaky Integrate-and-Fire

The Leaky Integrate-and-Fire (LIF) model also has its basis in electric circuits. Its inspiration comes from a simple circuit with a capacitor C and resistor R in parallel, where the driving current I represents the input current to the neuron. The conductance into neuron i at time t is represented as:

$$I(t) = \sum_j w_{ij} \sum_f \alpha(t - t_j^{(f)}) \,, \tag{2.6}$$

where $\sum_j$ sums over all the input neurons, and $\sum_f$ sums over all of the spikes from neuron j. w_{ij} is the weight value from neuron j to neuron i. $t - t_j^{(f)}$ becomes 0 if there is a spike fired at time t from neuron j. When considering input spikes to be of an infinitesimally short length, the α function can be represented abstractly by the Dirac-δ function, where $\delta(x) = 0$ if $x \neq 0$, and $\int_{-\infty}^{\infty} \delta(x)dx = 1$ [10]. It is often concretely calculated as $\exp\left(\frac{t}{\tau}\right)$[10]. Time-based simulations also often use a step function across a time-step.

In LIF neurons, the change in voltage v at time t is represented by

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{v_0 - v(t) + RI(t)}{\tau} \,, \tag{2.7}$$

where v_0 is the resting potential of the neuron — the voltage value of the neuron when there is no input — and $\tau = RC$ is a constant called the *membrane time constant*.

Spikes output from neuron i are represented by the time of their firing $t_i^{(f)}$, where f is the index of the spike. A spike occurs when the neuron voltage exceeds a threshold value called the spiking threshold. After a neuron emits a spike, the voltage is reset to a static value v_{reset}. This is different from the Hodgkin-Huxley

model, where variables within the system cause the voltage reset.

Spike-Response Model

The Spike-Response Model (SRM) is a generalization of the LIF model. The main difference between LIF and SRM is the direct inclusion of the refractory period. The refractory period is represented with a kernel function η. While η can be arbitrary, it usually decreases the membrane's voltage below a threshold, during which time the neuron is not able to spike no matter what input it receives. The response of the neuron to an input spike is represented by a kernel κ. The voltage update equation is generally presented as:

$$\frac{v(t)}{\mathrm{d}s} = \eta\left(t - \hat{t}\right) + \int_{-\infty}^{+\infty} \kappa(t - \hat{t}, s) I(t - s)\,, \tag{2.8}$$

where t is the current time, $\hat{t}$ is the time the neuron last output a spike, and I is the incoming current. The kernels η and κ can be chosen to model different types of neurons. For example, Kistler et al. [25] calculated kernels based on the Hodgkin-Huxley model, and Jolivet et al. [21] calculated kernels based on cortical interneurons.

Izhikevich Model

In 2003, Izhikevich presented a model for neuronal current as a system of ordinary differentiable equations as a compromise between the biological plausibility of the Hodgkin-Huxley model and the efficiency of the integrate and fire model [20]. Izhikevich calculated the current of a neuron by using a separate variable u that controls the refractory period:

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{2.9}$$

$$u' = a(bv - u) \tag{2.10}$$

$$\text{if } v \geq 30\text{mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{2.11}$$

The parameters a, b, c, and d are dimensionless and control different aspects of the model. The parameter a models the time scale of u — how slowly or quickly the neuron returns to resting potential after a spike. The parameter b "describes the sensitivity of the recovery variable to the subthreshold fluctuation of the membrane potential" [20]. Parameter c is the neuron reset voltage, and parameter d is the rest value of u.

Izhikevich explains the parameters for the parametric part of the model as being chosen to give time and voltage in ms and mV, respectively, and to keep the resting potential between -70 and -60 mV, which is in the range for the resting potential of biological neurons.

Using different values for a, b, c, and d, Izhikevich demonstrates that these equations can model many types of neurons observed in biology. The simplicity of the model makes it computationally inexpensive, but it still retains biological plausibility.

2.3.2 Real Value to Spike Translation

Most problems contain inputs that are real values, not spikes. Because of this, a translation method has to be used. There are two common types of translation methods: rate-based and temporal-based.

Rate-Based Encoding

Rate-based encodings convert real number values into spikes by producing a number of spikes that is proportional to the value when averaged over a period of time [24]. The spikes can be evenly or randomly distributed over the time period. For example, given a time window of 100 milliseconds, encoding the number 3 could produce 3 spikes at random times, spikes at 33, 66, and 99 milliseconds, or a number of spikes proportional to 3 (given that all input values have the same multiplier), like 30. Rate-based encoding can also be used to distribute spikes across trials, and/or multiple neurons [24].

Temporal-Based Encoding

Temporal-based encodings convert real number values into spikes by producing spike trains with spikes at specific points during the time window. There are many different types of temporally-based encoding methods [24]. One popular method is reverse correlation encoding. In this method, the magnitude of the input is inversely represented by the distance to the beginning of the spike train so that larger values will appear sooner during the time period [24]. For example, given a time window of 100 milliseconds, a value of 3 might appear only at the 97th millisecond, while a value of 82 might appear at the 18th millisecond.

Chapter 3

Evolution

Evolutionary computing and *evolutionary algorithms* mimic biological computation as a mechanism to solve specific problems. In nature, individuals survive based on their ability to navigate the environment. If they survive long enough to reproduce, their genes are passed on in the population. Darwin [3] described this mechanism as "natural selection." Genes that code for successful traits for survival are more likely to be passed on and spread throughout the population. Scientists have adapted this to computational tasks.

Evolutionary algorithms can find solutions to problems that would not have been thought of by man but are extremely successful. For example, Hornby et al. [18] at NASA used an evolutionary algorithm to design a successful flight antenna. Not only did these antennas outperform hand-designed ones, but they could be created in four weeks as opposed to three months.

Evolutionary algorithms evolve a population of *genomes*, or *individuals*, that represent solutions to a problem. Genomes are made of *genes*, values that correspond to a specific part of the solution.

There are many components of evolutionary algorithms. These algorithms

generally involve some or all of these components: representation of an individual, the evaluation method, a parent selection mechanism, variation operators, and a replacement mechanism. The following sections include a brief review of each.

3.1 Overview

The general outline of an evolutionary algorithm is as follows:

1. Initialize a population of solutions
2. Evaluate solutions on the problem
3. Select parents for variation
4. Perform:
    - Recombination
    - Mutation
    - Other variational operators
    - A combination of the above
5. Select survivors
6. Repeat steps 2 through 5 until a terminating condition

Steps 2–5 represent a generation. A *generation* in computational evolution is representative of a generation in biological evolution: individuals in a population reproduce to create new individuals with slightly different genes. However, in computational evolution, what happens during a generation is less constrained. For example, recombination, the analog of reproduction, can take many different forms, and is not required.

3.2 Representation

The way solutions are encoded into genomes is called *representation*. Some of the basic representation methods are bit string encoding, integer string encoding, and tree encoding, although there are many other methods, and often the representation is chosen based on the specific problem [7]. Representation directly influences variation operators, which will be discussed later.

For example, artificial neural networks can be represented for evolution in many different ways, and this is one of the challenges of evolving network topologies as will be discussed in Section 4.1. However, when evolving a static structure, one can use a vector where each value represents a specific weight in the network.

3.3 Evaluation

Evolution also requires a method to evaluate an individual. An *evaluation function* is used to assign "fitness" to each of the individuals. *Fitness* measures each individual's performance on the problem. Fitness values are then used to compare individuals for parent and survivor selection. Fitness is often used as a termination condition for the algorithm as well — when an individual's fitness reaches a certain value, the problem is considered solved and the algorithm is stopped.

For example, if an ANN is being evolved to solve a classification task, the fitness could be assigned to be the number of examples correctly classified. The algorithm could be terminated when the network correctly classifies a certain percentage of the examples.

3.4 Variation Operators

Variation operators modify individuals. Without modification, the population would never change. The initial population would stay stagnant and persist through the generations. This would keep populations from ever evolving, finding, and converging to a solution. There are two main methods of variation — recombination and mutation. Using either or both can allow the populations to change and evolve over the generations.

3.4.1 Parent Selection

Before performing variation, the individuals to undergo these variations must be selected. Parents can be selected at random. While this method may work well in some situations, it is possible that well-performing individuals are not chosen to undergo variation and their genes are removed from the population. To try to prevent this, proportionality can be introduced to selection, giving solutions with a higher fitness a larger likelihood of being selected [4]. Parent selection and survivor selection can use many of the same algorithms like tournament selection and roulette wheel selection. These methods will be discussed in Section 3.5.

3.4.2 Recombination

Recombination takes two or more individuals — *parents* — from the population and combines them to make new individuals — *children* (or *offspring*) — with parts of both. There are many different methods of combining solutions. The method used is dependent on the representation of the individuals. As such, methods may be individualized to the specific problem. I will describe a few of the common methods here.

Single-Point Crossover

In single-point crossover, two parents create two offspring. A point is chosen in the length of the bit string. One child gets the first section of the genome from the first parent and the second section from the second parent, while the other gets the first section from the second parent and the second section from the first parent [4]. Where to divide the bit string can be deterministic or random. This method can be expanded to multi-point crossover, where the genomes are divided into more than two sections and the children get multiple smaller sections from each parent [4]. Single point and multi-point crossover are mainly used with bit strings and real valued vectors.

Consider evolving a static neural network with 4 layers, represented by a vector of weights sorted by their layer in the network. Using single point crossover with Parent A and Parent B could yield Child C with layer 1 and 2 weights from Parent A and layer 3 and 4 weights from Parent B, and Child D with layer 1 and 2 weights from Parent B and layer 3 and 4 weights from Parent A.

Uniform Crossover

In uniform crossover, each value has the possibility of coming from each parent [4]. This prevents the need for a hyperparameter for cut points as in single-point and multi-point crossover.

Using a four layer neural network, uniform crossover could yield Child A with layers 1 and 4 from Parent A and layers 2 and 3 from Parent B, and Child B with layers 1 and 4 from Parent B and layers 2 and 3 from Parent A.

Blend Crossover

Another simple crossover method is blending. In blend crossover, the children have the average value between each parent for each gene [7].

With a static architecture network, blend crossover yields two networks with all weights being the average of their parents.

Tree Crossovers

Methods to perform successful crossover on tree and graph structures have been extensively studied. A lot of this difficulty is also based in how the graph is represented. Depending on the representation, crossover methods described above work, although they may not be intuitive. A common method is subtree-crossover, where a random node in each tree is chosen and the subtrees of those nodes are swapped [7].

3.4.3 Mutation

Various methods of mutation exist, and, like recombination, they depend on the chosen representation. For bit string representations, it is common to flip the bit value [4]. For real valued vectors there are a few more options. One option is to completely replace the previous value with a new random value; another is to add a value drawn from a distribution [35]. Using a uniform distribution here would allow for a random value within a certain range.

Mutation is generally considered as a per-gene operator: each gene is considered individually and is mutated with a probability. This is in contrast with selection and replacement, which consider the genome as a whole. However, two probabilities can be used to separately determine if the genome will undergo

mutation and then if a gene will undergo mutation.

3.4.4 Inversion

Inversion is a variational method that is used with vector representations. Inversion is the process of reversing the order of part of or all of the vector string [4]. This method keeps the same genes but changes their order.

3.5 Replacement

After recombination and mutation, members of the next population must be selected. This is called *replacement*. One approach is to replace members randomly. While this may work, it risks replacing well-performing individuals. Generally, it is better to choose a method of replacement that takes fitness into account.

Two popular selection methods are (μ+λ) and (μ, λ). In these methods, μ refers to the parent population and λ refers to the child population. These methods consider individuals' ranks in the population. Ranks can be assigned linearly or through other functions to apply different selective pressures. In (μ+λ) methods, the next population is selected based on rank from the merged parent and child populations [7]. In (μ, λ) methods, the population is selected based on rank from only the child populations [7]. Rank-based selection methods like these avoid some of the problems of fitness-proportional selection that occur from varying fitness distributions [7].

Another option is *roulette wheel selection*. In roulette wheel selection, a 'roulette wheel' is created by having a list of individuals with the number of each individual proportional to its normalized fitness. Then until the population is replaced, individuals at random indices are copied to the next generation [35].

Yet another method is *tournament selection*. Tournament selection is a method to select the well-performing individuals with replacement. Until the new population is full, n individuals are selected at random from the merged parent and child populations. The individual among those n with the highest fitness value is copied into the new population [35]. The number of individuals selected for each tournament, n, is called the selection pressure [35]. The higher the n, the higher the selection pressure. Low selection pressure increases the chance that poor performing individuals will be copied into the new population. A higher selection pressure generally decreases diversity and leads to faster convergence. A lower selection pressure may keep the population from reaching convergence. As such, n must be chosen thoughtfully and with consideration to the problem.

Elitism is a method that can be added to any other selection method. *Elitism* is directly transferring the top n performing individuals to the next generation without performing recombination [35]. Other variation operators may or may not be performed with elitism.

Chapter 4

Evolution of Artificial Neural Networks

Artificial neural networks can be evolved in a number of ways. These methods can be broadly categorized as using static architectures or evolving architectures. In both of these methods, there are a variety of aspects of the network that can be evolved. Some examples include network weights, activation functions, learning rules, and hyperparameters like learning rate and momentum. Evolving the weights of the network is most common.

In this chapter, I will review some literature on the evolution of both static topologies and evolving topologies. First, I will start with a common problem that occurs when evolving neural networks: choosing an encoding method. Section 4.2 will review literature on the evolution of networks with static architectures. Section 4.3 will discuss literature on evolving the topologies of networks.

4.1 Problems with Encodings

One challenge when it comes to evolving neural networks is finding an appropriate encoding. When evolving network weights with a fixed structure, encodings are straight-forward. For example, a vector of real numbers can be used, where each value represents a specific weight in the network. However, this is not possible when the network structure is changing. The encoding now has to account for the structure of the network as well as the weights of the changing connections. One of the main challenges trying to find an encoding for evolving topologies is developing a method for appropriate crossover. One approach is to swap random subtrees, similarly as in genetic programming. However, because of the nature of neural networks, two differently structured networks may solve the problem, and a random swap may have a negative effect. Many different methods have been developed to address this problem and some will be described in later sections.

4.2 Evolution of Static Architectures

Consider evolving the weights of an artificial neural network with a predefined, static architecture. In this problem, the weights can be represented as a vector, with each index of the vector corresponding to a particular weight in the architecture. Each vector is evaluated for fitness by instantiating a network with the encoded weights, training it for a predefined amount of time, and using the end error as the fitness. Within this basic framework, there is a wide variety of implementation options.

4.2.1 Fogel et al. 1990: Evolving Neural Networks

Fogel et al. [9] demonstrates the evolution of neural networks with static architectures on XOR and a gasoline blending problem. An evolutionary programming approach is used to evolve the networks. Children are created by mutating each value in the parent vector by adding a number drawn from a normal distribution with a mean of zero. The variance of the mutation distribution decreases over time. The next population is computed through a variation of $(\mu + \lambda)$ selection. This method is applied to XOR using a network with a single hidden layer with two nodes and to a gasoline blending problem, using a network with two hidden layers with two nodes in the first and five in the second. In both cases, the evolutionary algorithm was able to solve the problem in fewer generations on average than backpropagation. Fogel et al. [9] claims that one of the strengths of evolutionary programming over backpropagation is its generalization: at the time, backpropagation required networks to be in layers, but evolutionary programming could be applied to networks with skip connections or feedback loops. Although backpropagation has since been applied to networks with skip connections and feedback loops [14, 29], the generalizability of evolutionary algorithms influenced the use of them in this work.

4.2.2 Weiland 1990: Evolving Neural Network Controllers for Unstable Systems

Wieland [44] uses genetic algorithms to evolve recurrent neural networks on variations of the pole balancing problem. The pole balancing problem consists of applying forces to a moving cart to balance a pole upright on top of it. Five variations of the problem are addressed: a single pole with the location and the

angle of the pole as inputs to the network, a single pole with only the angle as input to the network, a single pole with no network inputs, multiple poles, and jointed poles. The networks are fully recurrent: each node has an input from every other node as well as itself and knows the full state of the network. The number of neurons in the networks varied depending on the problem. The networks are represented as vectors. The genetic algorithm uses mutation, crossover, and inversion. The algorithm was able to balance a single, unjointed pole for 10^6 timesteps in only 6 generations. The two-pole problem with poles of lengths 1.0 and 0.1 meters required 150 generations to reach 10^6 timesteps, and the jointed pole problem with the bottom pole with a length of 1 meter and the top pole a length of 0.1 meters required 30 generations. Wieland [44] shows that fully connected recurrent networks with a set number of neurons can be evolved with a genetic algorithm to control unstable systems. The use of evolution on recurrent networks reinforces Fogel et al. [9]'s claim regarding evolutionary algorithms' generalizability. Wieland [44] also shows that ANNs can be evolved to solve reinforcement learning problems and not just supervised learning problems.

4.3 Evolution of Weights and Topologies

It is possible to evolve the architecture of the network alongside the weights or other objectives. There are a wide variety of methods to address the encoding and crossover problems discussed in Section 4.1. The following sections review three methods that address these problems to simultaneously evolve weights and topologies of networks.

4.3.1 Evolving Connectionist Systems

Evolving connectionist systems (ECOS) were introduced by Kasabov [23]. These systems learn both the weights and the topology of an artificial neural network. Kasabov [23] gives seven main principles of the systems: 1) fast learning from a large amount of data, 2) adaptation in real-time, 3) input variables, output variables, connections, and neurons are changed, 4) data learning and knowledge representation are comprehensive and flexible, 5) interaction with the environment, 6) representation of space and time in their respective scales, and 7) self-evaluation [23, 43, 34]. ECOS learning methods can be applied to any type of artificial neural network. The initial application by Kasabov was fuzzy neural networks [23]. Other methods implementing ECOS are evolving self organizing maps and evolving clustering methods [24].

4.3.2 SANE

In 1997, Moriarty and Miikkulainen [28] developed the SANE algorithm to evolve both the architecture and weights of neural networks. Genetic algorithms are used to coevolve populations of neurons and architectures. To evaluate fitness, multiple combinations of neurons and architectures are run on the problem. Architectures' fitnesses are assigned as the average of all of these runs. Neurons' fitnesses are assigned as the average of the top five runs that they participated in. Taking only the top five helps make sure that good neurons are not discarded because of bad architectures. Moriarty and Miikkulainen [28] shows that the method allows for the specialization of neurons using lesion studies on the evolved networks.

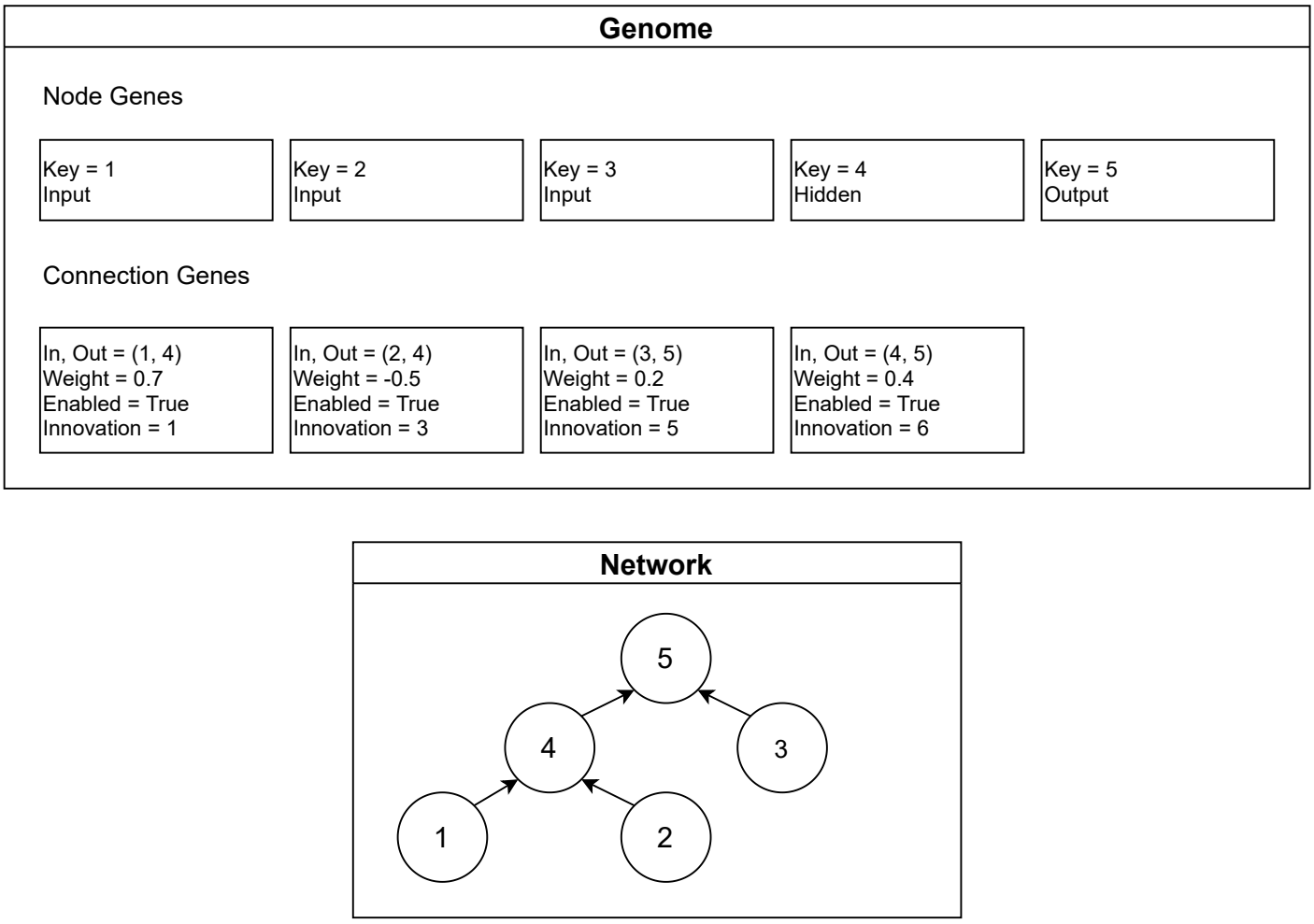


Figure 4.1: An example genome and corresponding network used in NEAT. Replicated with modification from [37].

4.3.3 NEAT

NeuroEvolution of Augmenting Technologies (NEAT) is an algorithm developed by Stanley and Miikkulainen [37] from the University of Texas at Austin in 2001. The authors sought to overcome some of the challenges of evolving network topologies discussed above. The main contributions are an encoding method allowing for crossover, the use of historical markers, speciation to protect innovation, and starting all networks from a minimal size.

In order to encode the networks, there are two types of genes included in each genome: node genes and connection genes. *Node genes* include the node's key and its role in the network (input, output, or hidden). *Connection genes* contain the connection's inputs and outputs, its weight, whether it's enabled or disabled, and its historical marker. An example genome and its respective network is shown in Figure 4.1.

Mutations control the changing structure of the networks. Nodes can be added by splitting a connection between two nodes and inserting the new node at the split. The old connection is still included in the gene but is marked as disabled and does not effect the output of the network. The connection leading into the new node is given a weight of one while the connection coming from the new node uses the weight of the old connection. To add a connection, two unconnected nodes are connected.

Historical markings are used to track the origin of each gene. The algorithm defines a *global innovation number* that tracks the latest assigned innovation number. New nodes or connections can only be created through mutation; when this happens, the gene is assigned the current global innovation number, and the global innovation number is incremented. When a mutation is inherited, it main-

tains the same innovation number. In this way, the markers represent the role of the node in the network. When genomes crossover, the genes with the same innovation number are paired, matching up the different topological parts. During crossover, the genes that are in both parents with the same innovation numbers are inherited randomly from either parent. Genes with innovation numbers that are only present in one of the parents are classified as disjoint or excess. *Disjoint* genes are genes that have innovation numbers that are within the range of the other parent's innovation numbers; *excess* genes are genes that have innovation numbers that are greater than all the other parent's innovation numbers. Disjoint and excess genes are inherited from the more fit parent.

Stanley and Miikkulainen [37] uses speciation to help protect network innovation. A *species* is a group of networks within the population that share certain qualities. Early structural innovation can be punished by the fitness function because of a bad weight value even if the new structure is a helpful change. To mitigate this, a speciation method is introduced. New innovations are evaluated within the context of the species of similar networks, rather than the entire population. Species are determined by the *genomic distance* δ between genomes, measured by the number of excess and disjoint genes and the difference in matching genes' weights. The genomic distance is defined as

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}\,, \tag{4.1}$$

where E is the number of excess genes, D is the number of disjoint genes, N is the number of genes in the larger genome, and $\overline{W}$ is the average weight differences of matching genes, and c_1, c_2, and c_3 are parameters that control the influence of the excess genes, disjoint genes, and weight differences, respectively. A variable

δ_t is defined to be the maximum distance genomes can have and still exist in the same species.

Explicit fitness sharing is used to keep species small and decrease the chance that a single species might dominate a population. This helps promote diversity within the population. An individual's fitness with fitness sharing, f'_i, is defined as:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}\,, \tag{4.2}$$

where f_i is the individual's raw fitness. The function $sh(\delta(i,j))$ is equal to 0 if $\delta(i,j) > \delta_t$, otherwise it is equal to 1. $\sum_{j=1}^{n} sh(\delta(i,j))$, then, is the number of individuals in the same species.

In the NEAT algorithm, each genome is started with a minimal structure — using only input and output nodes, and no hidden nodes. Nodes are then only added through mutation. This promotes the algorithm to find the smallest network(s) that can solve the problem and prevents users from initializing networks larger than needed.

The authors used their algorithm to solve XOR and the double pole balancing problem, with and without velocity information. NEAT successfully solved XOR in an average of 32 generations without failing any trial. Evolved networks tended to be small, with only one or two hidden nodes. NEAT took an average of 24 generations to solve the double pole balancing problem with velocity information. While one compared method that evolved static networks with ten hidden nodes took a similar number of generations, NEAT evolved networks with zero to four hidden nodes, demonstrating the algorithms ability to find small working structures. NEAT solves the double pole balancing problem without velocity in

fewer generations than the compared methods.

NEAT has been applied to several different problems, including competitive coevolution problems [38] and Pacman [45]. NEAT has also been extended in various ways to evolve compositional pattern producing networks [39], to generate video game content [13], and to control online, decentralized multi-robot systems [36]. The success and influence of NEAT in a broad array of domains influenced this work's exploration of its utility when combined with SNNs.

Chapter 5

Evolution of SNNs

Like ANNs, SNNs can be evolved in order to solve problems. Evolving SNNs removes the problem of determining a learning rule that will work well with the temporal and binary aspects of spikes. The following sections review literature that uses evolution for spiking neural networks.

5.1 Hagras et al. 2004: Evolving Spiking Neural Network Controllers for Autonomous Robots

Hagras et al. [12] uses the evolution of spiking neural networks to train a robot to follow a wall at a specified distance. The robot uses 16 photoreceptors to receive black and white input on a wall. The image is convolved with a Laplace filter and scaled between zero and one. The scaled values are encoded into spikes using latency encoding: the strength of the input is the inverse delay of the spike. That is, if the robot received a strong input, the neuron would spike earlier than

if the robot received a low input. Network outputs are used to control the robot's movements. The firing rate of the two output neurons, measured over 20 milliseconds, is transformed to appropriate units, scaled, and controls the speed of the wheels. The spike response neuron model is used and networks did not include any hidden neurons. Evolution is performed using an adaptive online genetic algorithm. The genomes consist of 20 bits: 5 bits per weight, representing the values 0 to 31. Crossover probabilities are adaptively changed to speed up the algorithm. Changes are controlled by comparing the best individual's fitness to the population's average fitness to determine the status of the convergence of the algorithm. The fitness of individuals is measured by the average standard deviation of the robot from the wall. The method is compared to a handcrafted SNN, SNNs evolved with a standard genetic algorithm, and a Fuzzy Logic Controller (FLC). The standard genetic algorithm improved more quickly at first, but then was unable to get out of local minima, while the adaptive algorithm started slower but was able to exit local minima. The adaptive algorithm also outperformed the other methods in terms of standard deviation fitness. The success of the evolved networks over the handcrafted networks of the same structure demonstrates the capability of evolution to find solutions that may not be obvious to humans.

5.2 O'Halloran et al. 2011: Evolving Spiking Neural Network Topologies for Breast Cancer Classification in a Dielectrically Heterogeneous Breast

O'Halloran et al. [30] uses the NEAT algorithm with SNNs, similar to this thesis, to classify whether a breast cancer tumor is malignant. The generated dataset consisted of three different tumor models with two classes: malignant or non-malignant. The data is preprocessed and converted to spike trains using rate-based encoding. Networks use LIF neurons. The networks are initialized with thirty input neurons, two output neurons, and no hidden neurons. The output neuron that fires more spikes is considered the classification. Fitness of each individual is determined by the count of correct classifications. Networks with evolved topologies showed classification increase from a fixed-topology network of about 10%. O'Halloran et al. [30] demonstrates the success of SNNs evolved with NEAT on binary classification problems. The use of LIF neurons and rate-based encodings provides evidence that these two selections could be successful on other problems as well, and this work follows uses the same methods.

5.3 Qiu et al. 2019: Evolving Spiking Neural Networks for Nonlinear Control Problems

Qiu et al. [32] also uses a combination of SNNs and NEAT. Using the Izhikevich neuron model and rate-based encoding, networks are evolved to solve the single

pole balancing problem to test the viability of applications to nonlinear control problems. A background current is injected into the neurons to promote firing, ensuring that the neurons would fire at a certain rate even with no input. This is similar to biological synaptic noise. The input data from the pole balancing environment is normalized in the range [0, 1] and linearly converted into a spiking rate with rate-based encoding. The population consists of 150 individuals. Species-based elitism is used: if a species contains more than 5 individuals, the best individual is copied to the next generation. The best 20% of individuals in each species is allowed to reproduce, and the offspring form the next generation. These and other parameters are the same or similar to those used in Stanley and Miikkulainen [37]. Fitness is defined as the number of timesteps the network keeps the pole balanced within set criteria. For pole balancing problem without velocity information, the authors slightly modify the NEAT genomic distance function to add a positive decay variable from the force function. On both problem variations (with and without velocity information), SNNs evolved with NEAT outperformed ANNs evolved with NEAT in terms of the number of generations to find a solution and the number of runs that did not find solutions within the generation limit. Qiu et al. [32] demonstrates the ability of NEAT and SNNs to find solutions to reinforcement learning problems. This work provides further exploration into this and follows many of the same evolutionary methods used in Qiu et al. [32] that are not present in Stanley and Miikkulainen [37], like species elitism. The use of rate-based encoding in Qiu et al. [32] work supports evidence from Hagras et al. [12] that this encoding method can be used to successfully convert real values into spikes for the use of SNNs evolved with NEAT and thus is chosen for this work.

Chapter 6

Evolving Spiking Neural Networks with NEAT

The following chapters will discuss the methods, experiments, and results of this work.

6.1 Goals

The goal of this work are to compare ANNs and SNNs evolved with NEAT with various parameter combinations, and examine the effects of the parameter changes on each type of network.

While both this work and the work of O'Halloran et al. [30] and Qiu et al. [32] discussed above both use NEAT to evolve SNNs, this work differs from theirs in a three of ways:

1. Use and comparison of neuron variations
2. Comparison of various NEAT and SNN parameter values

3. Direct comparison to NEAT with ANNs (different than O'Halloran et al. [30])

6.2 Problems

This work studies three problems: XOR, x to y mapping of a cosine function, and the single pole balancing problem. I chose these three problems because they represent different domains: XOR is a traditional binary classification problem, cosine is a regression problem, and single pole balancing is a reinforcement learning problem. They also all limit the number of inputs and outputs and thus the initial size of the networks. Limiting the initial network size decreased the run time of the simulations.

6.2.1 XOR

XOR is a traditional proof-of-concept problem used to show that a method can learn nonlinearities. It is a supervised learning problem where the algorithm has to learn the binary operator XOR: (0, 0) = 0, (1, 0) = 1, (0, 1) = 1, (1, 1) = 0. In this work, the fitness of an individual is zero minus the average mean squared error over the examples; all fitnesses are negative or zero with the maximum possible fitness equal to zero. The fitness required for the problem to be considered "solved" is -0.05. Because SNNs only output integers, there are only five possibilities for their fitness: -1, -0.75, -0.5, -0.25, and 0. Because of this, SNNs had to have a fitness of 0 to solve the problem. However, ANNs with real valued outputs could have fitnesses between the 4 binary options. This is why I allow for the problem to be solved at -0.05 rather than 0. I do not round the ANN outputs before calculating their fitness because during test runs, they

were never able to find a solution when rounded.

6.2.2 Cosine

The cosine problem used here involves mapping x coordinates to the y coordinates of a cosine wave. I chose to examine the networks on the cosine function for two reasons: I wanted to look at network performance on a regression problem (however, because of discretization it becomes a multi-class supervised learning problem), and because it shows that the networks can learn to output positive values with little to zero input.

I limit the domain of the function to [0, π), inclusive of 0, exclusive of π. The cosine function has a range of (-1, 1) of real numbers. However, spiking neural networks are limited to outputting zero and positive integer values. I had the option of scaling the function to be positive and large enough that integer roundings would be decent approximates, or scaling the outputs of the networks to be map between (-1, 1) (for example, make 0 spikes map to -1, 5 spikes map to -1/2, 10 spikes map to 0, etc.). Either way the SNN output would suffer from rounding problems. This demonstrates one of the limitations of SNNs. I chose the former option of scaling the function. Rather than use $y = \cos(x)$, I train the networks on $y = \text{round}(10 \times (\cos(x) + 1))$. This gives an output range [0, 20], inclusive. In order to better compare the ANNs and SNNs, the ANN outputs are also rounded to integers before calculating fitness.

The networks are given an explicit input bias unit. Networks have two inputs, the x value of the function, the constant 1, and have one output: the y value of the function.

The error on a single example is the mean squared error between the predicted

value and the actual value. The fitness of an individual is zero minus the average mean squared error over all of the examples; all fitnesses are negative or zero, with zero being the best possible fitness. For the problem to be considered "solved," a network has to achieve a fitness greater than -1.

6.2.3 Single Pole Balancing

The single pole balancing (SPB) problem is a reinforcement learning problem where the algorithm has to balance a pole on a moveable cart. The pole is attached to the cart at one end and stands straight up. In this work, the network is given the full state of the environment: the location of the cart, the velocity of the cart, the angle of the pole, and the rotational velocity of the pole. The network gives a single output. If the output is greater than 0.5, a force of +10 Newtons is applied to the cart. Otherwise, a force of -10 Newtons is applied. There is no stochasticity in the simulation. The simulation is run for 15 computational seconds with a timestep of .05 seconds, totalling to 750 timesteps. Each SNN is simulated for 100 milliseconds at each of these timesteps to produce an output. Qiu et al. [32] uses a timestep of .02 and ran for 2,000 seconds, totalling 100,000 timesteps. I chose to change these parameters for the sake of experiment run time. The fitness score of the network is the time before the pole's angle exceeds $\pm$ 45 degrees, with the maximum time being 15 seconds. The threshold to be considered as "solved" is the entire 15 seconds. The SNN's output is interpreted as a binary: if the network emits any spikes, the output is 1, otherwise it is 0. The ANN output is not rounded before it is input to the simulation because the simulation rounds it to determine the force to apply.

6.3 Set-up

I used the NEAT-Python [26] implementation of the NEAT algorithm, paired with the Brian2 [40] SNN implementation. Both of these simulations are freely available online. The following sections details the specifics of my simulations.

6.3.1 NEAT

NEAT Simulation

For these experiments, I made use of the NEAT-Python [26] implementation of NEAT. This section outlines some of the key implementation details of this library.

Node genes allow for the inclusion of more information than explicitly included within the NEAT algorithm. By default, node genes contain a key, a bias value, an aggregation function, and an activation function. Keys are assigned to nodes as innovation numbers. When a new node is created through mutation, it is assigned the next integer value compared to the present largest key. During crossover, if a node is passed to the child, the key does not change. The bias value is a real value that is added to the node output. The aggregation function is the method used to combine network inputs. This work uses addition as the aggregation function. The activation function specifies an activation function for an ANN. This work uses sigmoid and elu activation functions. Further attributes can be added, but this work uses only these four. The the bias, aggregation function, and activation function values are only used by the ANNs and are discarded by the SNNs. The original NEAT implementation stores the role of the node (input, output, or hidden) within the node gene. In this library, this information is not stored in the node genes, rather a list of input and output nodes is stored accessed

through a configuration object.

Connection genes include the inputs and outputs of the connection, the weight of the connection, and whether or not that connection is enabled. In contrast to the original NEAT algorithm, rather than using explicit innovation numbers in the connection genes, the library replaces this by comparing input and output nodes. If two connections have the same input and output nodes, they are considered to match. Because of the lack of innovation numbers, there is no difference between disjoint and excess genes. Variables c_1 and c_2 in Equation 4.1 collapse into a single variable as shown in Equation 6.1.

A genome consists of a unique key, a list of node genes, a list of connection genes, and the fitness of the genome. An example genome and its respective network can be seen in Figure 6.1.

The genomic distance calculation is modified to include the differences in the node and connection genes. The genomic distance is the sum of the distances between the nodes and the distances between the connections. Each distance is calculated the same as Equation 4.1, except with the excess and disjoint values merged. The modified equation becomes:

$$\delta = \frac{\text{CDC} \times D}{N} + \text{CWC} \times \overline{W}\,, \tag{6.1}$$

where CDC, the compatibility disjoint coefficient, is analogous to c_1 and c_2 in Equation 4.1 and CWC is equivalent to c_3 in Equation 4.1.

The number of disjoint node genes is calculated as the number of node keys that are contained in one genome but not the other. The number disjoint connection genes is calculated by the number of connections between input/output pairs that are in one genome but not the other. The weight difference, $\overline{W}$, is

calculated differently for node and connection genes. For node genes, it is the difference between the bias values, plus 1.0 if the aggregation functions are different, and plus 1.0 if the activation functions are different. Because in this work, the aggregation and activation functions are the same for all nodes, the weight difference is simply the difference between bias values. The weight difference for connection genes is the difference between weights plus one if one is enabled and the other is not. The total genomic distance is the sum of these two values.

Static Parameters

Each problem uses a population size of 100 individuals, initialized with no hidden nodes and completely connected. Because one of the principles of NEAT is to find minimal structures [37], networks are generally started without hidden nodes. There is a 50% chance that a connection will be added or deleted and a 20% chance that a node will be added or deleted. While both Stanley and Miikkulainen [37] and Qiu et al. [32] used much lower mutation rates, early tests found that the higher values worked well. Species were removed after 10 generations if they did not show improvements. This was chosen as less than Stanley and Miikkulainen [37] to decrease run time. Each species used elitism with the two best individuals being copied to the next generation. 20% of the species was allowed to reproduce each generation, similarly to Qiu et al. [32]. Species were defined to have at least 2 individuals. Connections are not allowed to change by changing between enabled and disabled. During testing, I found allowing weight disabling lead to more neurons being disconnected but still evaluated, which increased run time. Weights are initialized from a Gaussian distribution with a mean of zero and a standard deviation of one. Weights are limited to a maximum of 30 and a minimum of -30. Weights are constrained to prevent any one from overpowering

the network with an exceedingly large magnitude. Weights values have an 80% chance of mutating and a 20% chance of being replaced with a new value drawn from the same initial distribution. Stanley and Miikkulainen [37]'s mutation method was slightly different, having an 80% chance of a genome being mutated and then a 90% chance of being perturbed and a 10% chance of being replaced. The values I chose attempted to emulate that within the NEAT-Python library's [26] constraints.

Varied Parameters

The *compatibility threshold*, the maximum genomic distance for two individuals to be considered part of the same species, varies between 2.0, 2.5, and 3.0. This value corresponds to δ_t in Equation 4.1. Stanley and Miikkulainen [37] used a δ_t of 3.0. In test runs, I found that values higher than 3.0 put too much pressure on the population to have a lower number of species, causing it to go extinct quickly. Because of this, I decided to evaluate the differences between a range of lower compatibility thresholds, while not deviating too far from Stanley and Miikkulainen [37].

The *compatibility disjoint coefficient*, the coefficient for the disjoint and excess gene counts' contribution to the genomic distance, varies between 0.5 and 1.0. This value corresponds to c_1 and c_2 in Equation 4.1, which were both assigned to 1.0 in Stanley and Miikkulainen [37]. The NEAT-Python simulation uses the same value for both variables [26]. Similarly to the compatibility threshold, this parameter influences the number of species, and higher values tended to push the population towards extinction. I chose to only use one smaller option to allow for a larger difference between the values while still being between zero and one.

The *compatibility weight coefficient*, the coefficient for the weight multiplier

difference's contribution to the genomic distance, varied between 0.5 and 1.0. This value corresponds to c_3 in Equation 4.1. Stanley and Miikkulainen [37] used a compatibility weight coefficient of 0.4. However I found in test simulations that values lower than 0.5 allowed the population to break into a large number of very small species that never found solutions, so I decided to use larger values. I chose 0.5 and 1.0 so that these values would match the compatibility disjoint coefficient.

6.3.2 SNNs

SNN Simulation

In this work, I use the Brian2 library to run the spiking neural network simulations [40]. Brian2 takes equations from the user and calculates the variable values at each timestep. The experiments in this work use the default timestep of 0.05 milliseconds. The α function used to calculate the current into a postsynaptic neuron at the time of a presynaptic spike (Equation 2.6) is a step function with a width of one time-step; there is no synaptic effect on the incoming current. If there is a presynaptic spike, the voltage of the postsynaptic neuron is increased directly by the weight value on the next time-step.

Neurons

I use two neurons modelled from LIF. Both use a voltage reset value of 0 and a 5 millisecond refractory period. The first neuron variation, "N0," uses a constant input current of 1 millivolts. The second neuron, "N1," does not use an input current. N1 also *clamps* the voltage during the refractory period: the voltage is not updated and remains at 0. This is different from N0, where, although the

neuron is kept from spiking, the voltage is still updated by its related equation.

N0:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{1 - v}{\tau} \tag{6.2}$$

N1:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \begin{cases} 0 & \text{if refractory} \\ \frac{-v}{\tau} & \text{else} \end{cases} \tag{6.3}$$

Static Parameters

In order to calculate an output from the SNN, the network must be simulated for a specific amount of time for each input. In this work, simulations were run for 100 milliseconds. In test simulations, I found that this time period was long enough for the neurons to spike multiple times, but short enough to limit the run time. Both neurons have a refractory period of 5 milliseconds, which reset the voltage to 0. τ is fixed at 10 milliseconds. These values were used in many of the Brian2 examples and I found they worked well.

Varied Parameters

The *spiking threshold*, the value the voltage a neuron had to reach before outputting a spike, varies between 0.75, 1.0, and 1.25. In test experiments, I found that 1.0 worked well, and that differences more that 0.25 appeared to have a large impact. For this reason, I chose a step of 0.25 above and below 1.0.

These parameters and those described in Sections 6.3.1 were chosen to be constant for simplicity and consistency of simulations. These values could have been included in the genome and evolved alongside the weights and architectures. In future work, evolving these values should be explored.

Real Value to Spike Translation

The inputs for the XOR, cosine, and single pole balancing problems are real values. I chose to use rate-based encoding to turn these values into spikes. I chose rate-based because of its simplicity and easy integration with the Brian2 simulation. Input values are multiplied by 50 and evenly spaced across the 100 milliseconds of simulation time. During test experiments, I found that this value worked well to give enough input for the neurons to spike. Using a high multiplier, I address the problem of neurons needing several input spikes to ever emit an output spike. I did this as an alternative to injecting a constant background current in the N1 neurons as was done in [32].

6.4 Experiments

SNN trials take a combination of neuron type (2 possibilities), compatibility threshold (3 possibilities), compatibility disjoint coefficient (2 possibilities), compatibility weight coefficient (2 possibilities), and spiking threshold (3 possibilities) from the options listed above. This led to a total of 72 SNN trials, evenly split between N0 combinations and N1 combinations. Each trial was repeated five times. ANN trials take a combination of compatibility threshold (3 possibilities), compatibility disjoint coefficient (2 possibilities), and compatibility weight coefficient (2 possibilities). This led to a total of 12 trials, again each run five times. XOR and cosine trials were run for a maximum of 500 generations while SPB trials were run for a maximum of 400 generations for the sake of time.

Trials were performed at OU Supercomputing Center for Education & Research (OSCER) at the University of Oklahoma (OU). ANN trials took less than 15 minutes on all problems. SNN trials took about 30 minutes on the XOR

problem, and up to 48 hours on the cosine and SPB problems. Spiking neural networks take more time than artificial neural networks because simulating them is computationally expensive: each network is run for a specified period of computational time. In these trials, simulations were run for 100 milliseconds at 0.05 millisecond timesteps, resulting in 2,000 timesteps. In contrast, ANNs are limited only by number of computations and not timesteps, so they can often be evaluated faster.

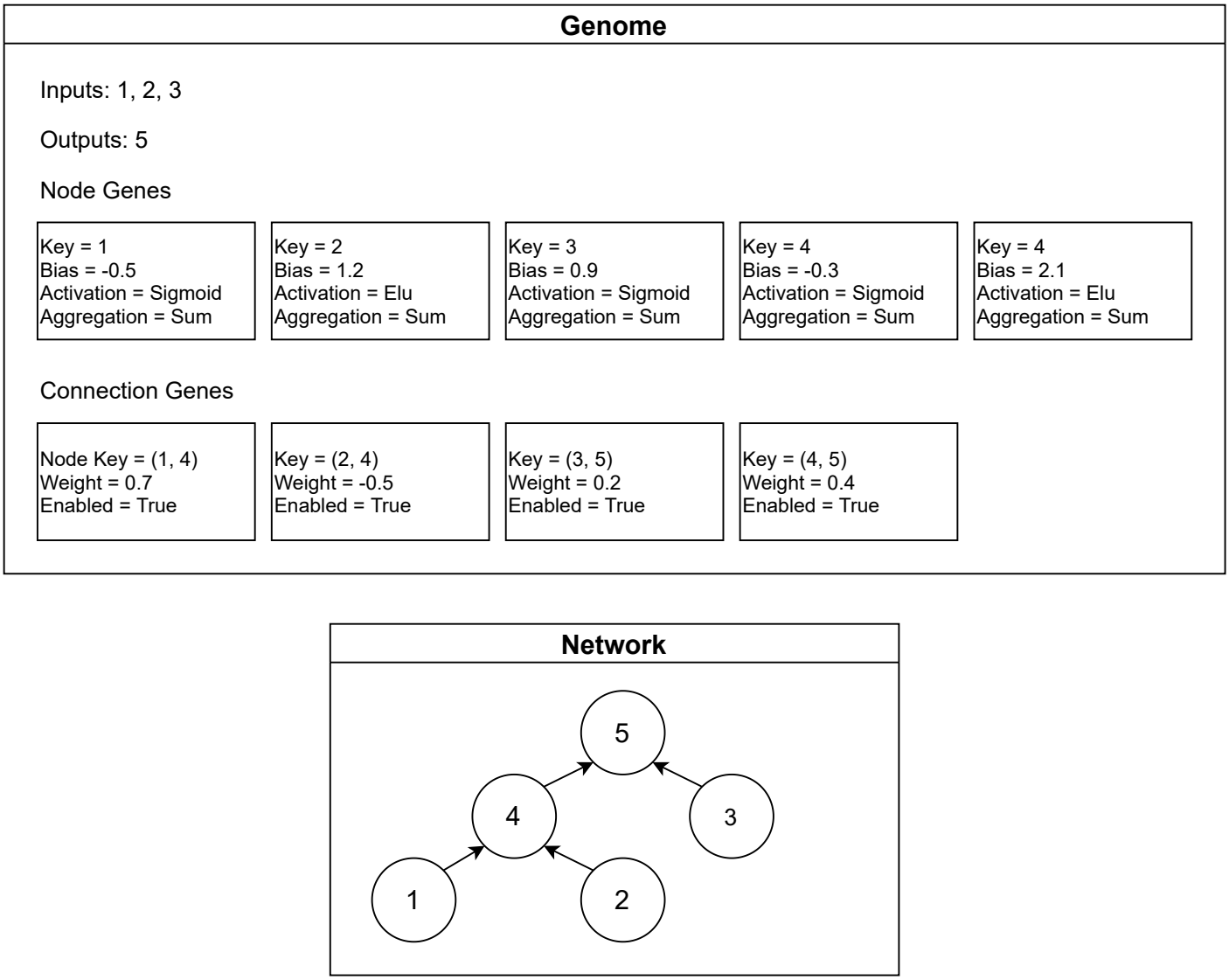


Figure 6.1: An example genome and its corresponding network as represented in the NEAT-Python simulation [26].

Chapter 7

Results and Discussion

Abbreviations:

CT - compatibility threshold

CWC - compatibility weight coefficient

CDC - compatibility disjoint coefficient

ST - spiking threshold

The plots provided are of singular examples that were specifically taken because they were found to be good representations of the whole of the results. Data provided in tables are averaged over all repetitions of parameter combinations except for the parameter values specified.

7.1 XOR

For the XOR problem, nine of the N0 combinations, six of the N1 combinations, and two of the ANN combinations found solutions during all five repetitions (four other ANN combinations found solutions during four of five repetitions). Table 7.1 shows the average number of generations, max fitness, number of solutions

Neuron	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	167.79 ± 177.42	-0.153 ± 0.127	2.25 ± 2.10	1.86 ± 2.08	1.03 ± 1.96
N1	110.87 ± 88.40	-0.108 ± 0.081	2.83 ± 1.62	1.92 ± 1.89	0.39 ± 0.76
ANN	192.71 ± 108.93	-0.060 ± 0.033	3.33 ± 1.25	0.50 ± 0.76	1.00 ± 1.29

Table 7.1: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for XOR.

found, number of extinctions and number of timeouts for each of the different neuron types.

Despite more SNNs finding solutions every time, Many ANNs found solutions two, three, or four times, whereas SNNs tended to find solutions either rarely or always. Because of this, ANNs had a higher average number of solutions found per combination, with 3.33 ± 1.25 out of 5 times ($66.7 \pm 25.0\%$) compared to 2.25 ± 2.10 out of 5 times ($45.0 \pm 42.0\%$) for N0 combinations and 2.83 ± 1.62 out of 5 times ($56.6 \pm 32.4\%$) for N1 combinations. The standard deviations here illustrate the larger breadth of the average number of solutions found by the SNN combinations.

None of the N0 networks that used a spiking threshold of 0.75 ever found a solution. A lower spiking threshold allows for more spikes, potentially decreasing the chance that a network would never spike and give an output of 0. N0 networks with spiking thresholds of either 1 or 1.25 were able to find solutions, with networks with a threshold of 1.25 only slightly less successful. This trend was not observed in N1 networks. Because of the clamping on the N1 networks after a spike, they spike less than N0 networks, which could account of their being less affected by lower spiking thresholds.

Table 7.2 shows the average number of generations, max fitness, number of solutions found, number of extinctions and number of timeouts. In N1 networks,

Neuron	CT	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	2.0	218.42 ± 178.74	-0.138 ± 0.132	2.58 ± 2.18	1.08 ± 1.89	1.33 ± 2.13
N0	2.5	174.58 ± 190.94	-0.158 ± 0.126	2.08 ± 2.22	1.67 ± 2.09	1.33 ± 2.13
N0	3.0	110.37 ± 141.85	-0.163 ± 0.123	2.08 ± 1.85	2.83 ± 1.86	0.42 ± 1.38
N1	2.0	181.63 ± 99.38	-0.083 ± 0.066	3.33 ± 1.31	0.75 ± 1.36	1.00 ± 0.91
N1	2.5	96.88 ± 59.24	-0.050 ± 0.073	3.17 ± 1.46	1.92 ± 1.75	0.17 ± 0.55
N1	3.0	54.10 ± 40.36	-0.150 ± 0.087	2.00 ± 1.73	3.08 ± 1.75	0.00 ± 0.00
ANN	2.0	282.65 ± 100.63	-0.071 ± 0.039	3.00 ± 1.22	0.50 ± 0.87	1.75 ± 1.48
ANN	2.5	186.90 ± 88.84	-0.057 ± 0.034	3.25 ± 1.48	0.75 ± 0.83	1.00 ± 1.22
ANN	3.0	108.58 ± 48.76	-0.052 ± 0.021	3.75 ± 0.83	0.25 ± 0.43	0.25 ± 0.43

Table 7.2: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for XOR, separated by varying compatibility thresholds.

Neuron	CDC	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	0.5	105.82 ± 149.84	-0.192 ± 0.120	1.61 ± 1.80	3.00 ± 1.94	0.61 ± 1.57
N0	1.0	229.76 ± 181.17	-0.114 ± 0.122	2.89 ± 2.18	0.72 ± 1.52	1.44 ± 2.22
N1	0.5	81.42 ± 54.12	-0.129 ± 0.072	2.42 ± 1.44	2.33 ± 1.80	0.33 ± 0.75
N1	1.0	197.10 ± 85.78	-0.046 ± 0.032	4.08 ± 0.64	0.33 ± 0.62	0.83 ± 0.90
ANN	0.5	147.87 ± 81.50	-0.054 ± 0.016	3.67 ± 0.75	0.83 ± 0.90	0.33 ± 0.47
ANN	1.0	237.55 ± 114.31	-0.066 ± 0.043	3.00 ± 1.53	0.17 ± 0.37	1.67 ± 1.49

Table 7.3: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for XOR, separated by varying compatibility disjoint coefficients.

Neuron	CWC	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	0.5	118.07 ± 156.52	-0.194 ± 0.129	1.67 ± 1.89	2.94 ± 2.07	0.56 ± 1.57
N0	1.0	217.51 ± 183.06	-0.111 ± 0.111	2.83 ± 2.14	0.78 ± 1.44	1.50 ± 2.19
N1	0.5	108.98 ± 90.68	-0.117 ± 0.083	2.67 ± 1.67	2.08 ± 1.88	0.33 ± 0.78
N1	1.0	169.53 ± 87.22	-0.058 ± 0.040	3.83 ± 0.80	0.58 ± 1.11	0.83 ± 0.90
ANN	0.5	130.83 ± 73.65	-0.052 ± 0.020	3.67 ± 0.75	0.83 ± 0.90	0.33 ± 0.75
ANN	1.0	254.58 ± 103.19	-0.068 ± 0.041	3.00 ± 1.53	0.17 ± 0.37	1.67 ± 1.37

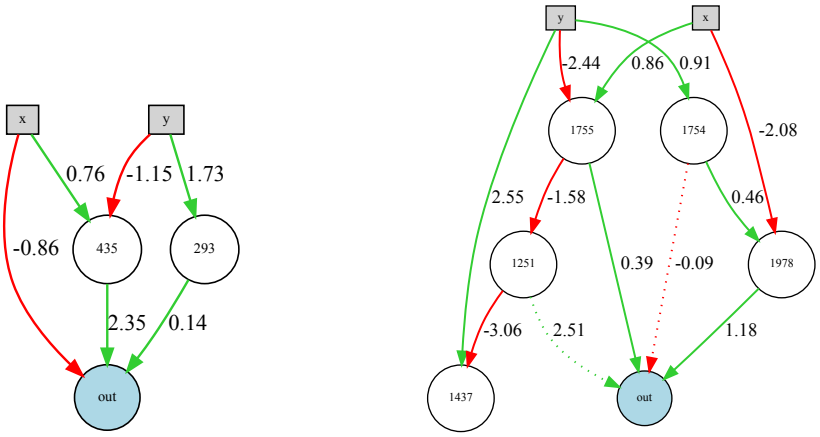
Table 7.4: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for XOR, separated by varying compatibility weight coefficients.

with each increase in the connection threshold, the average number of solutions decreased (from 3.33 to 2.83 out of five) and the average number of extinctions increased (from .75 to 3.08 out of five). This trend is seen to a lesser extent in the N0 networks, where the average number of extinctions did increase (from 1.08 to 2.83), but the average number of solutions found held relatively steady; the increase of extinctions came more directly from the decrease in timeouts. This pattern did not appear to happen in the ANNs.

Tables 7.3 and 7.4 show that, on average for all network types, increasing either the CWC or CDC decreases the number of extinctions. In N0 and N1 combination averages, there is also an increase in the number of solutions found.

The N0 network trials with both CWC and CDC = 0.5 went extinct an average of 4.44 times out of five. N1 exhibited similar behavior, but to a lesser extent, going extinct an average of 3.78 times out of five. Only four ANN neurons ever went extinct, and three of the four of them were the trials with CWC = CDC = 0.5. This behavior occurred with every compatibility threshold.

The ANNs that found solutions either four or five times all either had CWC = 0.5 and CDC = 1 or CWC = 1 and CDC = 0.5. As mentioned above, the networks that had both CWC = CDC = 0.5 tended to go extinct more often — at a rate of 1.67 out of five times compared to 0.33 out of five times when CWC = CDC = 1 and 0 out of five times when they had the paired values of 1 or 0.5. The networks with both CWC = CDC = 1 tended to time out more often, with 2.67 out of five times, compared to 0 out of five times for when CWC = CDC = 0.5 and 0.67 out of five times when they had the paired values of 1 and 0.5. Proportionately, the groups that had the paired values of 1 and 0.5 found solutions at a higher rate: 4.33 out of five times compared to three out of five times when CWC = CDC = .05 and 1.67 out of five times when CWC =



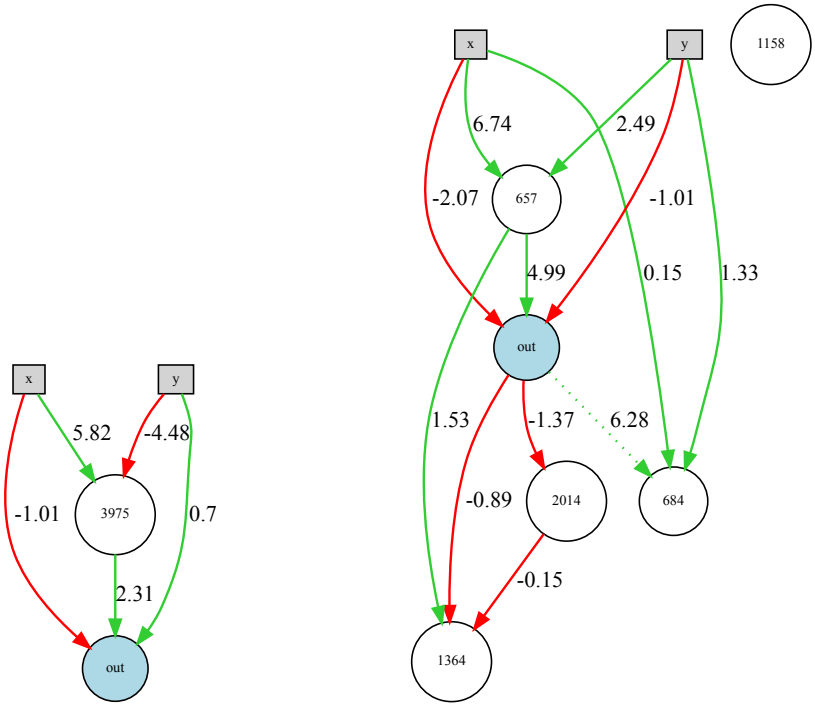
(a) An N0 network with CT = 2.0, CWC = 1.0, CDC = 0.5, ST = 1.25 that solved XOR in the 27th generation.

(b) An N1 network with CT = 2.0, CWC = 1.0, CDC = 0.5, ST = 0.75 that solved XOR in the 93rd generation.

Figure 7.1: Example SNNs that found solutions to XOR.

CDC = 1. Looking at both N0 and N1 networks, this pattern does not appear to hold. Although when CWC = CDC = 0.5, the SNNs found fewer solutions and went extinct at higher rates, as mentioned above, when CWC = CDC = 1, the networks performed about the same as when one was 0.5 and the other was 1.

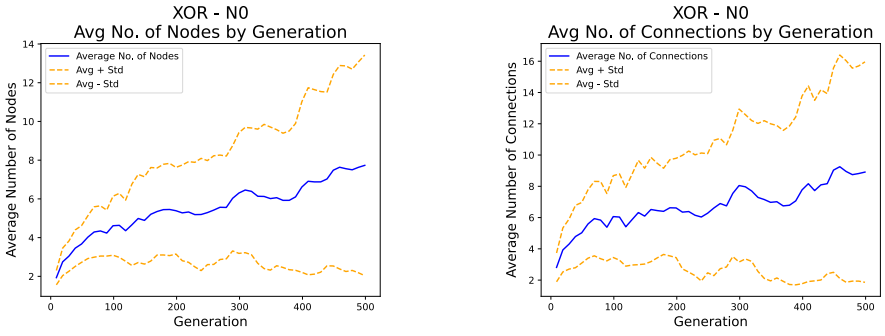
Figure 7.1 shows SNNs and Figure 7.2 shows ANNs that found solutions to the XOR problems. All types of networks were able to evolve small structures, with one to two hidden nodes as well as larger, more complicated networks with several hidden nodes. Figures 7.3, 7.4, and 7.5 show the averages and standard deviations of the number of nodes and connections for every 10th generation. The graphs average across all parameter combinations and repetitions. As the number of generations increases, the number of networks included in these averages decreases because of trials solving the problem and going extinct. N0 and N1 combinations tended to have the larger networks than the ANNs. N0 combinations reach a maximum average of 7.7 nodes at generation 499 and 9.3



(a) An ANN with CT = 3.0, CWC = 0.5, and CDC = 1.0 that found a solution to XOR in the 306th generation.

(b) An ANN with CT = 2.5, CWC = 0.5, and CDC = 1.0 that found a solution to XOR in the 146th generation.

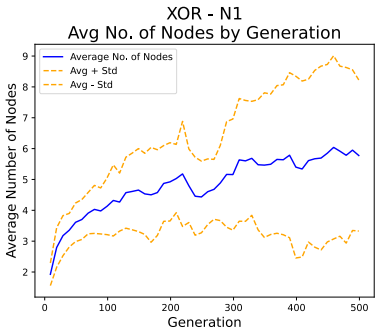
Figure 7.2: Example ANNs that found solutions to XOR



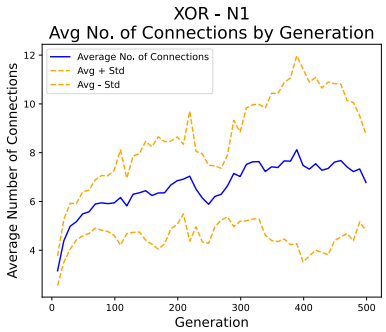
(a) Average number of nodes in each N0 network.

(b) Average number of connections in each N0 network.

Figure 7.3: Average N0 network sizes on XOR by generation, averaged over all N0 parameter combinations.

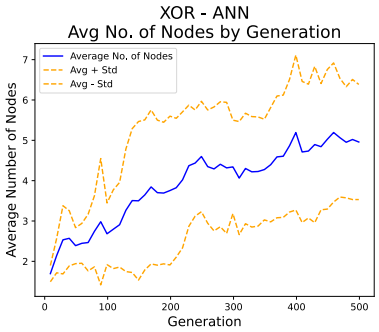


(a) Average number of nodes in each N1 network.

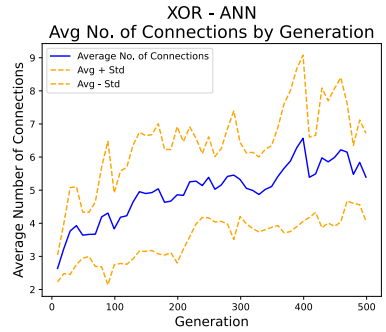


(b) Average number of connections in each N1 network.

Figure 7.4: Average N1 network sizes on XOR generation, averaged over all N1 parameter combinations.



(a) Average number of nodes in each ANN.



(b) Average number of connections in each ANN.

Figure 7.5: Average ANN sizes on XOR by generation, averaged over all ANN parameter combinations.

Neuron	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	213.56 ± 146.15	-1.353 ± 0.570	2.03 ± 1.34	1.64 ± 1.80	1.33 ± 1.55
N1	336.83 ± 184.14	-4.947 ± 0.306	0.00 ± 0.00	2.08 ± 2.28	2.92 ± 2.28
ANN	49.52 ± 69.64	-0.818 ± 0.114	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

Table 7.5: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for cosine.

connections at generation 459. N1 combination sizes reach a maximum average of 6.0 nodes at generation 459 and 8.1 connections at generation 389. ANNs average sizes reached 5.2 average nodes at generation 459 and 6.6 average connections at generation 399. All network combinations show a quick increase in size in the first 50 generations, after which the growth rate slows. N0 neurons show both the maximum average number of nodes and connections; they also show the largest average standard deviation across the generations. By the last generation, N0 combinations had a total of 40 networks left to average, compared to N1 combinations with 10 and ANN combinations with 11. The larger number of networks left in N0 combinations would account to some extent the larger standard deviation.

7.2 Cosine

On the cosine problem, every ANN combination found a solution every repetition. One N0 combination found a solution every repetition, while three others found a solution four out of five repetitions. N0 trials had an average max fitness of -1.35 ± 0.57. No N1 neuron combination ever found a solution, with an average maximum fitness of -4.95 ± 0.31 and an overall maximum fitness of -3.95 when CT = 3, CWC = 0.5, CDC = 1, and ST = 0.75.

Table 7.6 shows the effect of the compatibility threshold on the different types

Neuron	CT	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	2.0	317.15 ± 128.58	-1.324 ± 0.602	1.50 ± 0.96	1.00 ± 1.73	2.50 ± 1.71
N0	2.5	225.17 ± 136.12	-1.238 ± 0.503	2.42 ± 1.32	1.50 ± 1.61	1.08 ± 1.32
N0	3.0	212.38 ± 146.15	-1.254 ± 0.570	1.98 ± 1.34	1.62 ± 1.80	1.37 ± 1.55
N1	2.0	421.32 ± 138.43	-5.002 ± 0.502	0.00 ± 0.00	0.83 ± 1.52	4.17 ± 1.52
N1	2.5	313.95 ± 177.99	-4.892 ± 0.109	0.00 ± 0.00	2.58 ± 2.43	2.42 ± 2.43
N1	3.0	329.44 ± 198.55	-4.684 ± 0.102	0.00 ± 0.00	2.06 ± 2.23	2.89 ± 2.23
ANN	2.0	103.70 ± 99.24	-0.744 ± 0.125	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
ANN	2.5	27.15 ± 15.37	-0.826 ± 0.095	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
ANN	3.0	17.70 ± 4.10	-0.884 ± 0.068	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

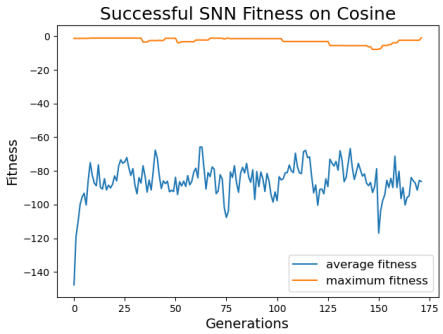
Table 7.6: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for cosine, separated by varying compatibility thresholds.

Neuron	CDC	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	0.5	176.51 ± 125.02	-1.341 ± 0.507	1.83 ± 1.38	2.39 ± 1.89	0.78 ± 0.97
N0	1.0	250.61 ± 156.03	-1.364 ± 0.626	2.22 ± 1.27	0.89 ± 1.33	1.89 ± 1.79
N1	0.5	296.96 ± 187.20	-4.993 ± 0.411	0.00 ± 0.00	2.44 ± 2.17	2.56 ± 2.17
N1	1.0	376.70 ± 172.03	-4.901 ± 0.116	0.00 ± 0.00	1.72 ± 2.33	3.28 ± 2.33
ANN	0.5	19.97 ± 6.12	-0.840 ± 0.090	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
ANN	1.0	79.07 ± 88.97	-0.796 ± 0.130	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

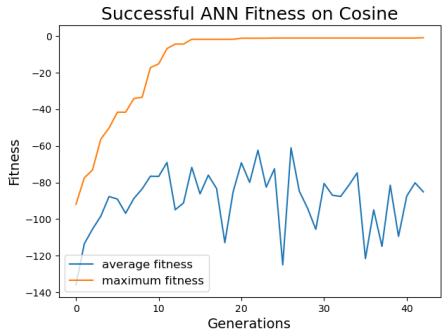
Table 7.7: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for cosine, separated by varying compatibility disjoint coefficient.

Neuron	CWC	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	0.5	155.44 ± 150.93	-1.506 ± 0.614	1.28 ± 1.04	2.89 ± 1.56	0.83 ± 1.74
N0	1.0	271.68 ± 114.82	-1.199 ± 0.474	2.78 ± 1.18	0.39 ± 0.95	1.83 ± 1.12
N1	0.5	188.89 ± 152.52	-5.056 ± 0.386	0.00 ± 0.00	3.89 ± 1.88	1.11 ± 1.88
N1	1.0	484.77 ± 27.96	-4.838 ± 0.120	0.00 ± 0.00	0.28 ± 0.56	4.72 ± 0.56
ANN	0.5	29.67 ± 32.64	-0.785 ± 0.092	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
ANN	1.0	69.37 ± 88.58	-0.850 ± 0.124	5.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00

Table 7.8: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for cosine, separated by varying compatibility weight coefficients.



(a) Fitness plot of an N0 network that solved cosine.



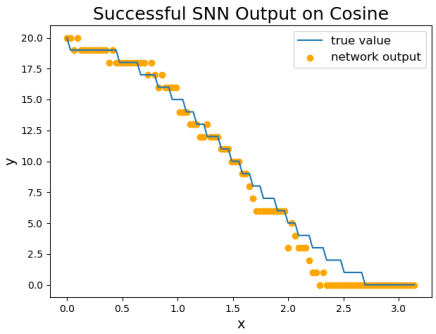
(b) Fitness plot of an ANN that solved cosine.

Figure 7.6: Example fitness plots of (a) an SNN and (b) an ANN that found solutions to cosine. The SNN used N0 neurons and an ST = 1.25. Both networks used a CT = 2.5, CWC = 1.0, and CDC = 0.5.

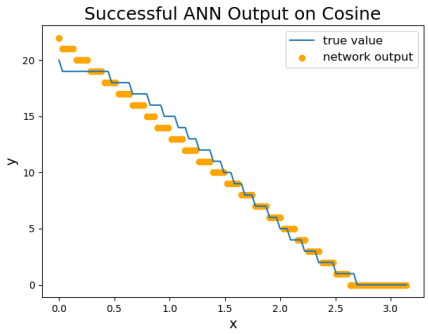
of networks. In the ANNs, increasing the CT decreased the number of generations it took for the network to find a solution (from 85.68 with a CT = 2.0 to 17.7 with a CT = 3.0). In SNN trials, increasing the CT increased the average number of extinctions.

CDC and CWC, averages shown in Table 7.7 and Table 7.8 respectively, also influenced the number of extinctions in SNNs and number of generations in ANNs. Similarly to in the XOR problem, increasing either the CDC or the CWC lead to a decrease in extinctions for both N0 and N1 combinations. N0 combinations also saw an increase in number of solutions found with either variable's increase. Both SNNs and ANNs saw a decrease in average number of generations with an increase of the CDC or the CWC. For the SNNs, this can be attributed to the increase in extinctions. For the ANNs, combinations with higher CDC or CWC values found solutions faster than their counterparts with lower variable values.

Figure 7.6 shows fitness plots of a successful N0 network and a successful ANN. While both networks' average fitness displays standard average learning curve, increasing quickly in the first generations before levelling off, only the

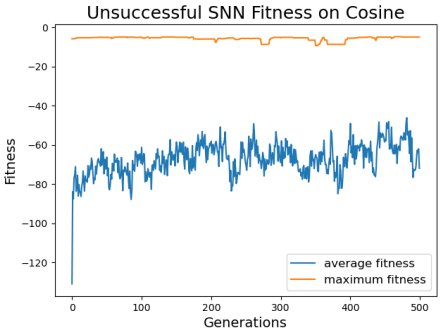


(a) Fitness plot of an SNN that solved cosine.

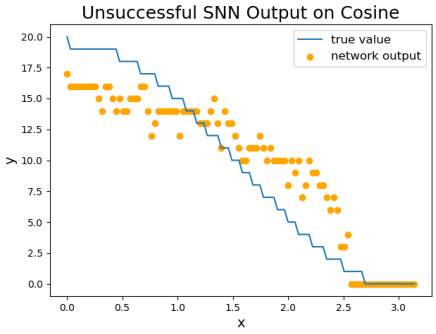


(b) Output figure of an ANN that solved cosine.

Figure 7.7: Example outputs of (a) an SNN and (b) an ANN that found a solutions to cosine. Values in nodes are the node The SNN used N0 neurons and an ST = 1.25. Both networks used a CT = 2.5, CWC = 1.0, and CDC = 0.5.



(a) Output figure of an SNN that failed to solve cosine.



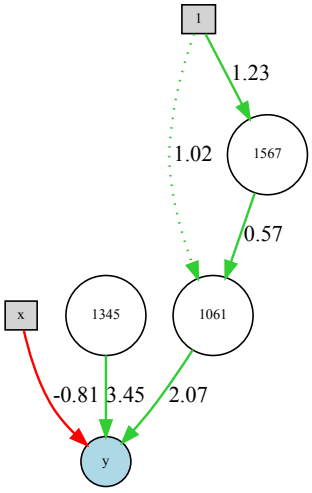
(b) Output figure of an SNN that failed to solve cosine.

Figure 7.8: Example of (a) a fitness plot and (b) the output of an SNN that failed to find a solution to cosine. This network had N1 neurons, CT = 2.5, CWC = 1.0, CDC = 0.5, and ST = 1.25. This network's end fitness was -4.9.

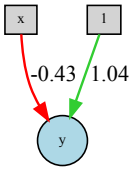
ANN displays the same type of learning with its maximum fitness. The SNN, in comparison, found a very successful network in the first trial, and its maximum fitness varied little across the generations. This behavior is consistent across networks and trials.

The outputs of the same SNN and ANN are shown in Figure 7.7. These graphs are the outputs of the networks where they reached the solution threshold. All values are rounded to the nearest integer, as they were during the simulations. The output of the N0 network follows the cosine curve closely from $x = 0$ to 1.5, fairly well capturing the nonlinearity of the beginning of the curve. Through the range of 1.5 and 2.25, however, rather than decreasing the slope as the curve does, the network's output is somewhat more scattered, following the same slope as the middle of the curve and reaching zero at 2.25 rather than 2.75. The ANN, between 0 and 2.5, follows a very linear path, disregarding the changes in the curve's slope. The network's output at zero was 22, rather than 20 and between 2.5 and π the network outputs constant 0. Rather than learning the curve itself, the ANNs tended to find a simpler solution, a linear mapping, that still reached the maximum fitness threshold.

Figure 7.8 shows fitness and output graphs of an unsuccessful N1 network. This network used the same parameters as the previous networks discussed. This network timed out without finding a solution. The fitness curve in 7.8a is very similar to the successful SNN fitness curve in 7.6b, except with lower maximum fitness values. The output figure shown in 7.8b is from generation 469, where network's overall maximum fitness of -4.78 occurred. The network is beginning to learn a downward curve, but the shape is not right. Near zero the output is too low, and while it does decrease, it keeps a small slope until about 2.5, where it drops quickly to zero.

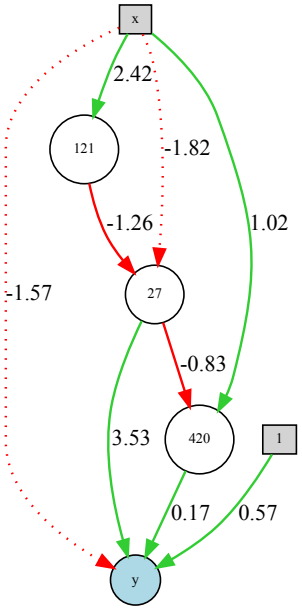


(a) An N0 network with CT = 2.0, CWC = 1.0, CDC = 0.5, and ST = 0.75 that found a solutions to cosine in the 137th generation.

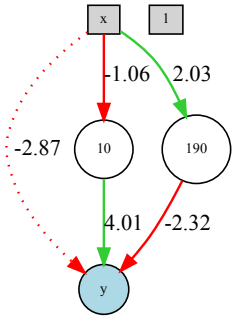


(b) An N1 network with CT = 2.5, CWC = 0.5, CDC = 1.0, and ST = 1.0 with a fitness of -4.95 from the 279th generation.

Figure 7.9: Example SNNs networks evolved on the cosine problem.

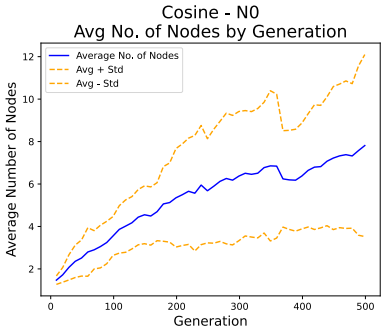


(a) An ANN with CT = 2.5, CWC = 1.0, and CDC = 1.0 that solved cosine in the 64th generation.

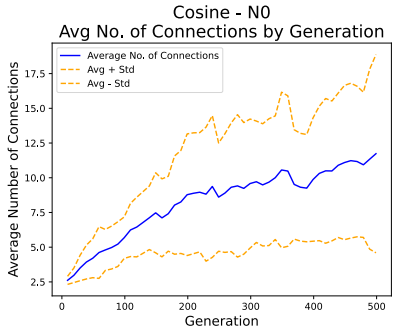


(b) An ANN with CT = 2.5, CWC = 1.0, and CDC = 0.5 that solved cosine in the 25th generation.

Figure 7.10: Example ANNs that found a solutions to cosine.

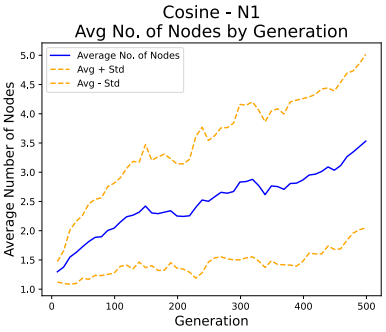


(a) Average number of nodes in each N0 network.

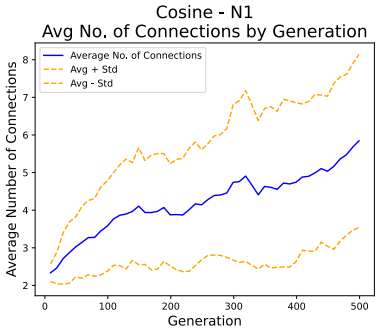


(b) Average number of connections in each N0 network.

Figure 7.11: Average N0 network sizes on the cosine problem by generation, averaged over all N0 parameter combinations.

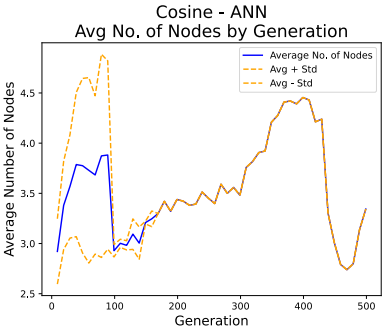


(a) Average number of nodes in each N1 network.

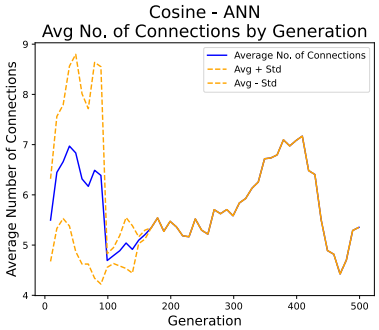


(b) Average number of connections in each N1 network.

Figure 7.12: Average N1 network sizes on the cosine problem by generation, averaged over all N1 parameter combinations.



(a) Average number of nodes in each ANN.



(b) Average number of connections in each ANN.

Figure 7.13: Average ANN sizes on the cosine problem by generation, averaged over all ANN parameter combinations.

Figure 7.9 show SNNs and Figure 7.10 show ANNs that solved cosine. All types of networks were able to evolve networks of various sizes and complexities. Not all evolved networks used the bias unit, for example the ANN in Figure 7.10b. Figures 7.11, 7.12, and 7.13 show the average sizes of N0, N1, and ANN combinations, respectively. N0 combinations held a fairly steady increase in both number of nodes and connections across all generations. By generation 499, N0 combinations had 52 networks left to include in the averages. N0 combinations' maximum average number of nodes and connections both occurred at the last generation, with 11.7 average nodes and 10.6 average connections. 101 total N1 networks timed out and were able to be included in the averages by generation 499. Like the N0 combinations, the N1 combinations maintain a fairly steady increase in size until the last generation, with maximum nodes and connections occurring at that point. N1 combinations had a maximum average node count of 3.5 and a maximum average connection count of 5.8. These networks are considerably smaller than the N0 combinations. This could potentially account, to some extent, the N1 combinations' worse performance. ANNs' longest running trial completed at generation 424 and was the only trial remaining since generation 349. There were only 16 trials that exceeded generation 49, and by 169 there were already only four trials left. Because of this, the standard deviation drops to near zero by generation 169. ANNs tended to solve cosine in very few generations, which accounts for this dramatic decrease. ANN combinations had a maximum average number of nodes of 4.5 at generation 399 and a maximum average number of connections of 7.2 at generation 309. The ANN networks tended to have sizes larger than N1 combinations but smaller than N0 combinations. It is possible that the larger complexity compared to N1 combinations is one of the reasons ANNs were able to solve the problem. The smaller complexity

Neuron	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	286.40 ± 123.26	1.709 ± 0.172	0.00 ± 0.00	1.39 ± 1.96	3.61 ± 1.96
N1	271.28 ± 130.93	2.020 ± 0.403	0.00 ± 0.00	2.00 ± 2.25	3.00 ± 2.25
ANN	117.37 ± 92.39	11.582 ± 3.233	3.17 ± 1.52	0.92 ± 1.55	1.08 ± 1.50

Table 7.9: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for SPB.

compared to N0 combinations could reasonably be accounted for by the lower average number of generations the ANNs took.

7.3 Single Pole Balancing

On the single pole balancing problem, neither N0 nor N1 trials were able to reach the solution threshold of 15 seconds. The maximum fitness of any N0 network run was 2.7 seconds, with N1 networks having an average maximum fitness of 1.70 ± 0.17. The maximum fitness of any N1 network run was 8.6 seconds with N1 networks having an average maximum fitness of 2.03 ± 0.40. Comparatively, two ANN networks were able to reach 15 seconds for all five repetitions, and five were able to reach 15 seconds for four of the five repetitions. Together, all ANN networks had an average maximum fitness of 11.58 ± 3.23 and found solutions an average of 63.4% of the time with a standard deviation of 30.4%.

Increasing the compatibility threshold increased the average number of extinctions for both SNNs and ANNs, as seen in Table 7.10. Across the ANN repetitions, increasing the CT also decreased the number of generations before finding a solution. The change in the CT did not have an average trend on the ANNs' max fitness, average solutions found, or average number of timeouts.

Table 7.11 and Table 7.12 show that increasing either the CWC or the CDC

Neuron	CT	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	2.0	371.72 ± 40.46	1.83 ± 0.109	0.00 ± 0.00	0.42 ± 0.76	4.58 ± 0.76
N0	2.5	281.08 ± 119.38	1.704 ± 0.162	0.00 ± 0.00	1.50 ± 2.10	3.50 ± 2.10
N0	3.0	206.40 ± 126.43	1.591 ± 0.147	0.00 ± 0.00	2.25 ± 2.20	2.75 ± 2.20
N1	2.0	310.40 ± 102.35	2.133 ± 0.402	0.00 ± 0.00	1.17 ± 2.03	3.83 ± 2.03
N1	2.5	288.63 ± 120.46	2.145 ± 0.410	0.00 ± 0.00	1.58 ± 2.06	3.42 ± 2.06
N1	3.0	214.80 ± 146.37	1.782 ± 0.270	0.00 ± 0.00	3.25 ± 2.09	1.75 ± 2.09
ANN	2.0	189.10 ± 102.71	10.675 ± 2.542	2.75 ± 1.30	0.25 ± 0.43	2.25 ± 1.92
ANN	2.5	88.00 ± 51.52	12.908 ± 2.042	3.75 ± 1.09	0.75 ± 1.30	0.75 ± 0.83
ANN	3.0	75.00 ± 67.84	11.163 ± 4.239	3.00 ± 1.87	1.75 ± 2.05	0.25 ± 0.43

Table 7.10: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for SPB, separated by varying compatibility thresholds.

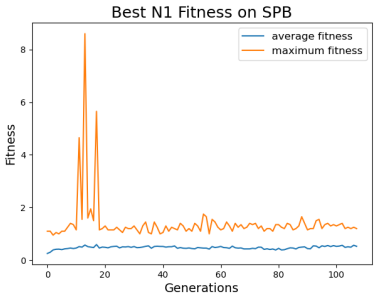
Neuron	CDC	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	0.5	250.69 ± 250.69	1.689 ± 1.689	0.00 ± 0.00	2.06 ± 2.06	2.94 ± 2.94
N0	1.0	322.11 ± 322.11	1.728 ± 1.728	0.00 ± 0.00	0.72 ± 0.72	4.28 ± 4.28
N1	0.5	212.67 ± 129.95	1.931 ± 0.366	0.00 ± 0.00	3.06 ± 2.12	1.94 ± 2.12
N1	1.0	329.89 ± 102.61	2.109 ± 0.418	0.00 ± 0.00	0.94 ± 1.84	4.06 ± 1.84
ANN	0.5	52.90 ± 27.36	13.012 ± 1.843	3.83 ± 1.07	1.00 ± 1.15	0.17 ± 0.37
ANN	1.0	181.83 ± 89.51	10.152 ± 3.663	2.50 ± 1.61	0.83 ± 1.86	2.00 ± 1.63

Table 7.11: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for SPB, separated by varying compatibility disjoint coefficient.

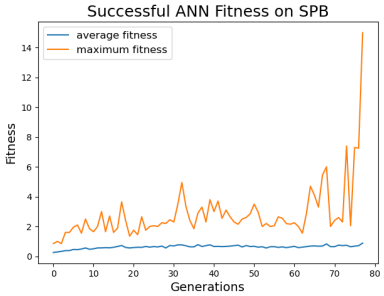
Neuron	CWC	Avg No. of Generations	Avg Max Fitness	Avg No. of Solutions	Avg No. of Extinctions	Avg No. of Timeouts
N0	0.5	219.41 ± 138.21	1.676 ± 0.179	0.00 ± 0.00	2.67 ± 2.08	2.33 ± 2.08
N0	1.0	353.39 ± 48.07	1.741 ± 0.158	0.00 ± 0.00	0.11 ± 0.31	4.89 ± 0.31
N1	0.5	191.41 ± 138.88	1.912 ± 0.423	0.00 ± 0.00	3.39 ± 2.19	1.61 ± 2.19
N1	1.0	351.14 ± 47.34	2.128 ± 0.350	0.00 ± 0.00	0.61 ± 1.21	4.39 ± 1.21
ANN	0.5	101.10 ± 81.89	10.298 ± 3.507	2.50 ± 1.38	1.83 ± 1.77	0.83 ± 1.21
ANN	1.0	133.63 ± 99.19	12.865 ± 2.305	3.83 ± 1.34	0.00 ± 0.00	1.33 ± 1.70

Table 7.12: Average number of generations, max fitness, number of solutions, and number of timeouts for the three types of networks for SPB, separated by varying compatibility weight coefficient.

(a) The fitness plot for the N0 network that had the highest maximum fitness. The network had CT = 2.5, CWC = 1.0, CDC = 1.0, and ST = 1.25.



(b) The fitness plot for the N1 network that had the highest maximum fitness. The network had CT = 1.5, CWC = 0.5, CDC = 0.5, and ST = 1.25.



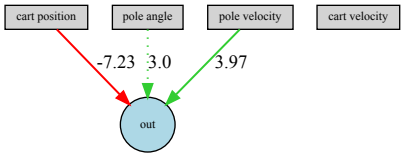
(c) A fitness plot for one of the ANN networks that found a solution during all batches. The network had CT = 2.5, CWC = 1.0, and CDC = 0.5.

Figure 7.14: Example fitness plots for the (a) N0 and (b) N1 networks that got the highest maximum fitness and (c) one of the ANN networks that successfully found a solution during all five repetitions.

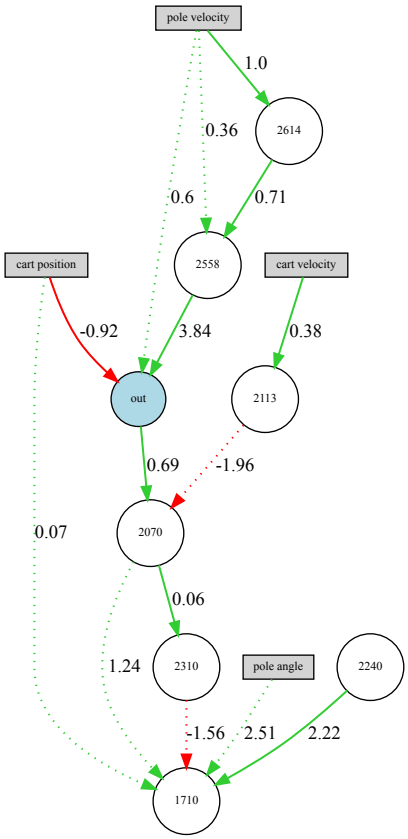
decreased the average number of extinctions across all combinations. Decreasing either the CWC or CDC more than halved the average number of extinctions for the SNN combinations. The greatest effect can be seen when increasing the CWC with N0 combinations, dropping the average number of extinctions from 2.67 ± 2.08 to 0.11 ± 0.31 ($53.4 \pm 41.6\%$ to $2.2 \pm 6.2\%$). Increasing the CWC on ANN combinations dropped the average number of extinctions from 1.83 ± 1.77 ($36.6 \pm 35.4\%$) to 0. Although there was not as sharp of an effect on the ANNs with the CDC increase, the average number of extinctions did decrease slightly, from 1.00 ± 1.15 to 0.83 ± 1.86 out of five ($20.0 \pm 23.0\%$ to $16.6 \pm 37.2\%$.

Figure 7.14 shows fitness graphs for the best performing N0 and N1 combination trials and an ANN that found a solution for the single balancing problem. In all graphs, the average fitness demonstrates a slight increase in the first few generations, increasing from nearly zero to around 0.5, where it remains. The N0 trial shows the same trend for the maximum fitness: a slight increase in the first few generations before leveling off. This population had a network that reached a sudden, high fitness, but that was not maintained. The N1 trial had a few spikes of fitness within the first 20 generations, but again that performance was not maintained and the maximum fitness dropped. The ANN shows a slight increase in maximum fitness overall before spiking a few times and reaching 15 seconds. Other ANN trials showed similar trends: a few medium spikes before a sudden large increase.

Figure 7.15 shows SNNs and Figure 7.16 shows ANNs that were evolved for the pole balancing problem. Both network types were able to evolve solutions that did not use all of the inputs. Both networks also evolved more complex solutions. The average size of evolved networks increased with further generations. This is illustrated in Figures 7.17, 7.18, and 7.19 for N0 networks, N1 networks,



(a) An N0 network with CT = 2.5, CWC = 0.5, and CDC = 0.5, and ST = 0.75 that has a fitness of 1.1 from the 49th generation.



(b) An N1 network with CT = 2.0, CWC = 0.5, and CDC = 0.5, and ST = 1.0 that has a fitness of 1.3 from the 139th generation.

Figure 7.15: SNN networks evolved on the single pole balancing problem.

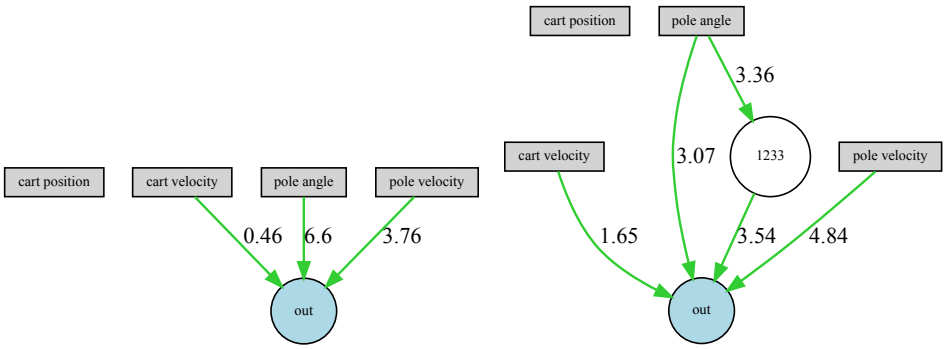
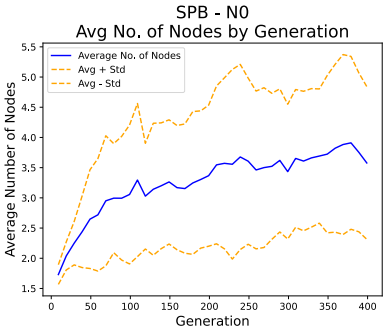
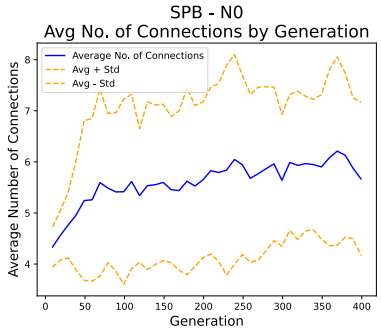


Figure 7.16: ANNs that solved SPB. Both networks used CT = 2.5, CWC = 1.0, CDC = 0.5.

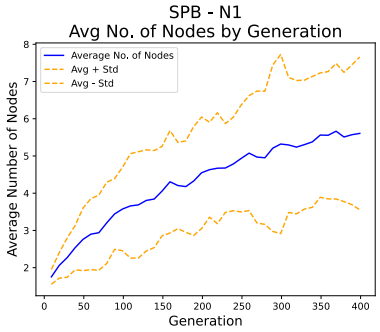


(a) Average number of nodes in each N0 network.

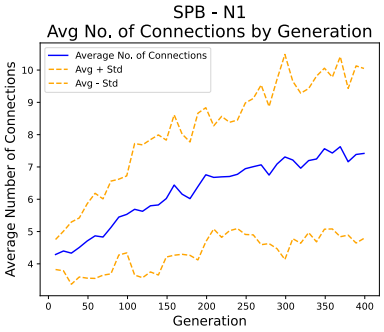


(b) Average number of connections in each N0 network.

Figure 7.17: Average N0 network sizes on SPB by generation, averaged over all N0 parameter combinations.

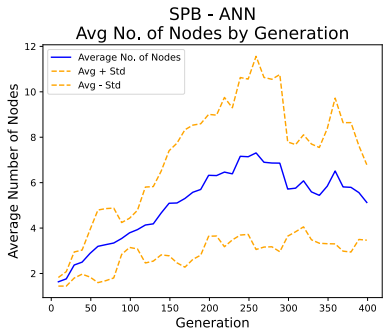


(a) Average number of nodes in each N1 network.

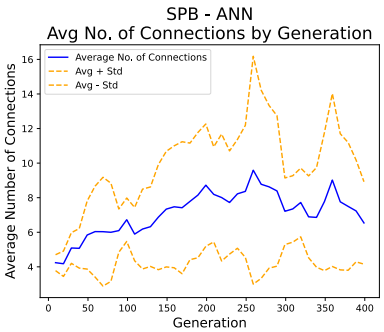


(b) Average number of connections in each N1 network.

Figure 7.18: Average N1 network sizes on SPB by generation, averaged over all N1 parameter combinations.



(a) Average number of nodes in each ANN.



(b) Average number of connections in each ANN.

Figure 7.19: Average ANN sizes on SPB by generation, averaged over all ANN parameter combinations.

and ANNs respectively. All network types show an increase in both node and connection size with an increase in number of generations. The increase in sizes tends to be more dramatic in the beginning generations before the growth slowing. This is most easily seen in the ANN trials in Figure 7.19, where the number of nodes and connections increases steadily until generation 250, where it levels off and even begins to decrease. At generation 259 the average number of nodes peaks at 7.3 and the average number of connections peaks at 9.6. As the networks grow, the standard deviation increases, and when the networks begin to shrink, the standard deviation similarly shrinks. Because networks begin with a minimal structure, it is always possible that small networks will still exist in the population at an one point; this could account for the increase in standard deviation with the increase in average size. In N0 repetitions, shown in Figure 7.17, the networks appear to grow quickly during the first 50 generations, and continue growing less quickly during the remainder of the generations. There is a small downturn before the last generation, however it can't be determined if this is the beginning of a trend or a small dip. The average number of nodes and connections on N0 trials peaked at 3.9 at generation 369 and 6.2 at generation 379, respectively. The standard deviation of the sizes increases quickly with the initial growth before remaining relatively constant after generation 100. N1 trials, shown in Figure 7.18, have the most steady growth, continuing to generally increase over all generations and at a faster rate than N0 trials. The maximum average number of nodes reached 5.7 and the maximum average number of connections reached 7.6, both in generation 369. The standard deviation increases with the increase in network size, more similarly to the ANN trials than the N0 trials.

The ANN trials that reached at least generation 50 could be expected to

have more connections than their SNN counterparts, and those that reached at least generation 200 could be expected, on average, to have both more nodes and connections than their SNN counterparts. It is possible that the ANN's success could be partially attributed to these larger sizes. However, the ANNs that found solutions ended on average by generation 50 with only 2 networks found solutions after generation 200. This makes it less likely that more nodes of the ANNs compared to the SNNs was a major contributing factor to their success. However, it is possible that the increased number of connections was beneficial.

Chapter 8

Discussion

The SNNs were able to find solutions to the XOR problem, with 15 of the network combinations finding solutions during all five repetitions. Although only two ANN parameter sets found solutions every repetition, across all of the parameter sets they had a higher average number of solutions found than the SNNs, with fewer networks never finding solutions.

On cosine, only one of the SNNs was able to find solutions every repetition, with three more finding solutions four out of five times. Every ANN found a solution every repetition.

On the single pole balancing problem, no SNN run was able to reach the time threshold of 15 seconds, whereas two ANN networks reach 15 seconds for all five repetitions.

Across all three problems, increasing the compatibility threshold increased the number of extinctions in trials. This is unsurprising. A higher CT allows for more diverse individuals to be in the same species, which allows for the convergence of a population into a single species. If this species then becomes similar enough and stagnates, it will be removed from the population and the population goes

extinct. The increase in extinctions, however, did not necessarily mean a decrease in fitness or the number of solutions found. In some instances, the average number of solutions found did decrease with an increase in the CT value, like N1 networks on the XOR problem. However, in other instances, like N0 networks on cosine, the number of solutions found actually increased with the increase of CT from 2.0 to 2.5. And in several other cases, either an increase or decrease in fitness value was minimal. The compatibility threshold appears to mainly effect the ratio of extinction to timeouts and not the fitness of the networks.

Across problems, the compatibility weight and disjoint coefficients appeared to have strong influence on the networks. Increasing either the CWC or CDC almost always had the effect of decreasing the number of extinctions. The CWC and the CDC are used in the calculation of the genomic distance. The CWC determines the influence the difference in genome weights have on the distance, and the CDC determines the influence the number of disjoint and excess neurons have on the distance. A combination of two low values for these variables would mean all neurons would be considered closer together, while two higher values would mean the neurons would be considered further apart. The trend of increasing these values leading to fewer extinctions across all compatibility thresholds implies that a larger genomic distance is favorable. With a larger distance, networks would be categorized into a larger number of species. This directly affects extinction probability: the more species, the less likely that all species will go extinct at the same time, leading to population extinction.

Using ANNs on cosine, we find that the networks perform best with either a low CWC and a high CDC or a high CWC and a low CDC. In this case, it appears that the value of the number of disjoint and excess genes versus the average weight differences is less important than not having a particularly large

or small genomic distance. Had either the weight or gene differences been more important to solving the problem, we would not expect to see similar performance between both combinations.

The spiking threshold did not appear to affect the networks on the cosine or pole balancing problems. However, it did appear to have influence on XOR. For the XOR problem, N0 combinations with the lowest ST option never found solutions, while their counterparts with higher STs did. It is possible that the lower spiking threshold made it too easy for the networks to emit a spike that they struggled to give a zero output. Contrastingly, the lower spiking threshold did not affect the N1 combinations the same way. N1 combinations with an ST of 0.75 found as many or more solutions than their counterparts.

On the cosine problem, the N1 combinations struggled to reach the peak of the cosine curve at zero across trials. This is partially due to researcher error. The cosine curve at $x = 0$ has a value of 20. The simulations are run for 100 milliseconds and the neurons have a refractory period of 5 milliseconds, so the most they could ever spike is twenty times. While the N0 neurons were able to reach 19, they were not clamped during their refractory period, so they could have a higher voltage when the refractory period ended, potentially already higher than the spiking threshold and able to spike immediately afterwards. However, the N1 neurons' voltage was clamped during the 5 millisecond refractory period, so when the refractory period ended, the voltage was still 0. At the very least, the network would have to wait until the next input spike to be able produce an output spike, which could be a few milliseconds. This clamping is one reason why the N1 neurons did not perform as well as the N0 neurons. Had I decreased the refractory period or increased the simulation time, the N1 neurons may have performed better. This should be explored in future work.

The results on the single pole balancing problem in this work do not mirror the results found in [32]. There are two main possible reasons for this. First, there is the difference in the simulators: the NEAT-Python [26] library changed a few implementation details of the NEAT algorithm. Qiu et al. [32] used the original NEAT library, which would not have these alterations. Secondly, there were various parameter differences. I used higher probabilities to add or remove nodes and connections, a larger timestep value, and elitism was implemented for all species rather than with a size threshold.

When ANNs were run on the cosine problem, every combination found a solution every repetition. However, the solutions found tended to be linear rather than following the cosine curve. I think that there are two main factors that could be effecting the networks' performance here. First, these networks used the elu activation function. I chose this function so that the network would be able to learn values in the range (0, 20) without scaling. However, when the input to the function is greater than zero, the activation is linear. It is possible that the linearity in the activation function influenced the linearity in the output. The second possible factor is that the fitness function, average mean squared error, only accounted for the distance from the curve and not the shape. The linear function that the networks found was a simpler solution that still fit this criteria.

Across all problems, the network sizes increase with more generations. This is consistent with [37]. Network sizes grow quickly in the first generations and less quickly in later generations. This could indicate that adding to the complexity of the networks stops having as large of an influence over time and weights become more important. The comparative network sizes do not remain consistent across problems and there is not enough evidence to draw conclusions about the influence of the type of neuron on network size.

Chapter 9

Conclusions and Future Work

In this work, I compared ANNs and SNNs evolved with NEAT with different parameter combinations. While the results are far from conclusive, they are promising that SNNs can rival ANNs in performance.

For the XOR problem, the SNNs were able to find solutions, and in some cases consistently. Comparing SNNs to ANNs, while the SNNs had a lower average number of solutions found across all parameter sets, they had more networks that found solutions consistently on every repetition.

For the cosine problem, while SNNs with multiple parameter sets were able to find solutions, only one was able to find a solution across all five repetitions. Comparing SNNs with ANNs, although by fitness criteria alone the ANN parameter combinations outperform the SNN combinations, examining the outputs of the networks reveals that the ANNs may not be learning the full nonlinearity of the curve as well as the SNNs.

For the single pole balancing problem, the SNNs were not able to find solutions while the ANNs were. This is inconsistent with the results from Qiu et al. [32], however, this is possibly due to differences in simulations and parameters.

The results included in this work contribute a foundation for studies on the combination of SNNs and NEAT. While it is known from other studies that the combination can solve problems [32, 30], the presented parameter and comparison studies on these simple problems give insight into configurations that can be used for more complicated studies. This work shows the effect of the compatibility threshold, compatibility disjoint coefficient, compatibility weight coefficient, and spiking threshold. I demonstrate the effect these have on the evolution of the networks, including extinction rate, fitness, and number of generations before solutions are found.

Further exploration of the combination of SNNs and NEAT is necessary before drawing definitive conclusions. As discussed, the networks are sensitive to changes in parameters. Further combinations of the parameters in this study should be tried, and other parameters, like stagnation rate, elitism, and the percentage of the population allowed to reproduce, should be included. These parameters could also be included in the genome and evolved with the networks to further optimize performance. Other neuron types, like Izhikevich neurons, should also be explored. This work reviewed only a few simple problems, and the algorithm should be applied to more complex problems. The method should be tested for generality by using a larger dataset that can be split into training, validation, and testing datasets. Methods should be explored to decrease run time, as time is one of its most detrimental features.

Bibliography

[1] S M Bohte, H La Poutré, and J N Kok. Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons. *Neurocomputing*, 48:17–37, 2000.

[2] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1251–1258, 2017.

[3] Charles Darwin. *On the Origin of Species*. PF Collier & Son New York, 1909.

[4] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Springer, 2003.

[5] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks, 2017.

[6] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution, 2018.

[7] Andries P Engelbrecht. *Computational Intelligence: An Introduction*. John Wiley & Sons, 2007.

[8] Dario Floreano and Claudio Mattiussi. Evolution of spiking neural controllers for autonomous vision-based robots. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2217:38–61, 2001. ISSN 16113349.

[9] David B Fogel, Lawrence J Fogel, and VW Porto. Evolving neural networks. *Biological Cybernetics*, 63(6):487–493, 1990.

[10] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014.

[11] Robert Gütig and Haim Sompolinsky. The tempotron: A neuron that learns spike timing-based decisions. *Nature Neuroscience*, 9(3):420–428, 2006. ISSN 10976256. doi: 10.1038/nn1643.

[12] Hani Hagras, Anthony Pounds-Cornish, Martin Colley, Victor Callaghan, and Graham Clarke. Evolving spiking neural network controllers for autonomous robots. *Proceedings - IEEE International Conference on Robotics and Automation*, 2004(5):4620–4626, 2004. ISSN 10504729. doi: 10.1109/robot.2004.1302446.

[13] E. J. Hastings, R. K. Guha, and K. O. Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[15] Donald Olding Hebb. *The Organization of Behavior: a Neuropsychological Theory*. J. Wiley; Chapman & Hall, 1949.

[16] A Hodgkin and A Huxley. A quantitative description of membrane current and its application to conductance and excitation. *Journal of Physiology*, 117:500–44, 1952.

[17] AL Hodgkin. Evidence for electrical transmission in nerve: Part i. *The Journal of Physiology*, 90(2):183, 1937.

[18] Gregory Hornby, Al Globus, Derek Linden, and Jason Lohn. Automated antenna design with evolutionary algorithms. In *Space 2006*, page 7242. 2006.

[19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

[20] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.

[21] Renaud Jolivet, J Timothy, and Wulfram Gerstner. The spike response model: a framework to predict neuronal spike trains. In *Artificial neural networks and neural information processing–ICANN/ICONIP 2003*, pages 846–853. Springer, 2003.

[22] Janusz Kacprzyk and Witold Pedrycz. *Springer Handbook of Computational Intelligence*. Springer, 2015.

[23] Nikola Kasabov. Evolving fuzzy neural networks — algorithms, applications and biological motivation. *Methodologies for the Conception, Design and Application of Soft Computing*, pages 271–274, 1998.

[24] Nikola K Kasabov. *Time-space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*. Springer, 2019.

[25] Werner Kistler, Wulfram Gerstner, and Leo van Hemmen. Reduction of the hodgkin-huxley equations to a single-variable threshold model. *Neural Computation*, 9, 09 2000. doi: 10.1162/neco.1997.9.5.1015.

[26] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. Neat-python. `https://github.com/CodeReclaimers/neat-python`.

[27] Ammar Mohemmed, Stefan Schliebs, Satoshi Matsuda, and Nikola Kasabov. Span: Spike pattern association neuron for learning spatio-temporal spike patterns. *International Journal of Neural Systems*, 22(04):1250012, 2012.

[28] David E Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, 1997.

[29] Michael Mozer. A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, 3, 01 1995.

[30] Martin O'Halloran, Brian McGinley, Raquel Cruz Conceicao, Fearghal Morgan, Edward Jones, and Martin Glavin. Spiking neural networks for breast cancer classification in a dielectrically heterogeneous breast. *Progress In Electromagnetics Research*, 113:413–428, 2011.

[31] Filip Ponulak. ReSuMe-new supervised learning method for Spiking Neural Networks. *Inst. Control Information Engineering, Poznan Univ.*, 22(2):467–510, 2005. ISSN 1530-888X. doi: 10.1.1.60.6325.

[32] Huanneng Qiu, Matthew Garratt, David Howard, and Sreenatha Anavatti. Evolving spiking neural networks for nonlinear control problems. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1367–1373. IEEE, 2018.

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[34] Stefan Schliebs and Nikola Kasabov. Evolving spiking neural network-a survey. *Evolving Systems*, 4(2):87–98, 2013. ISSN 18686478. doi: 10.1007/s12530-013-9074-9.

[35] Nazmul Siddique and Hojjat Adeli. *Computational Intelligence: Synergies of Fuzzy Logic, Neural Networks and Evolutionary Computing*. John Wiley & Sons, 2013.

[36] Fernando Silva, Paulo Urbano, Luís Correia, and Anders Lyhne Christensen. odneat: An algorithm for decentralised online evolution of robotic controllers. *Evolutionary Computation*, 23(3):421–449, 2015.

[37] Kenneth O Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[38] Kenneth O Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

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

[40] Marcel Stimberg, Romain Brette, and Dan FM Goodman. Brian 2, an intuitive and efficient neural simulator. *eLife*, 8:e47314, aug 2019. ISSN 2050-084X. doi: 10.7554/eLife.47314. URL `https://doi.org/10.7554/eLife.47314`.

[41] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[42] Alexander Vandesompele, Florian Walter, and Florian Röhrbein. Neuroevolution of spiking neural networks on spinnaker neuromorphic hardware. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6, 2016.

[43] Michael J Watts. A decade of kasabov's evolving connectionist systems: a review. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 39(3):253–269, 2009.

[44] Alexis P Wieland. Evolving neural network controllers for unstable systems. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 667–673. IEEE, 1991.

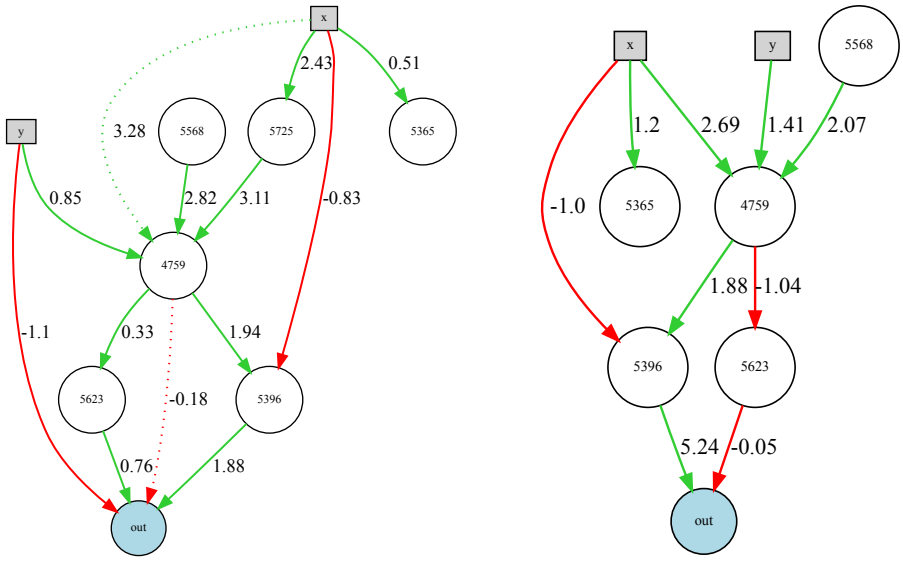
[45] M. Wittkamp, L. Barone, and P. Hingston. Using neat for continuous adaptation and teamwork formation in pacman. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 234–242, 2008.

Chapter 10

Appendix

10.1 Gallery of Evolved Networks

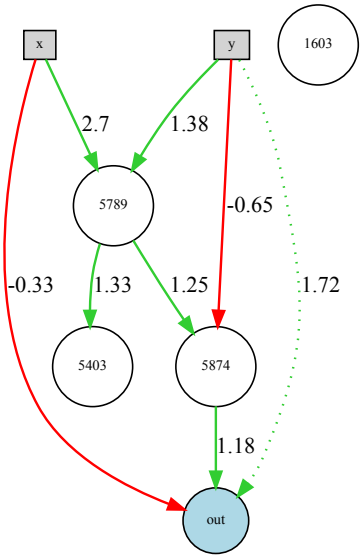
10.1.1 XOR



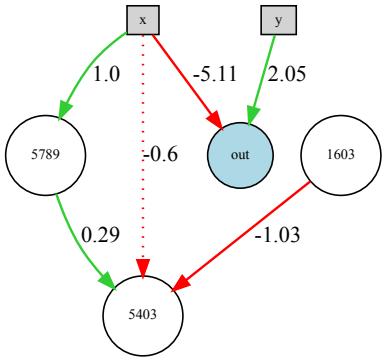
(a) End fitness = 0.

(b) End fitness = -0.25.

Figure 10.1: N0 networks evolved on XOR with CT = 2.5, CWC = 0.5, CDC = 1.0, and ST = 1.25 at generation 211.

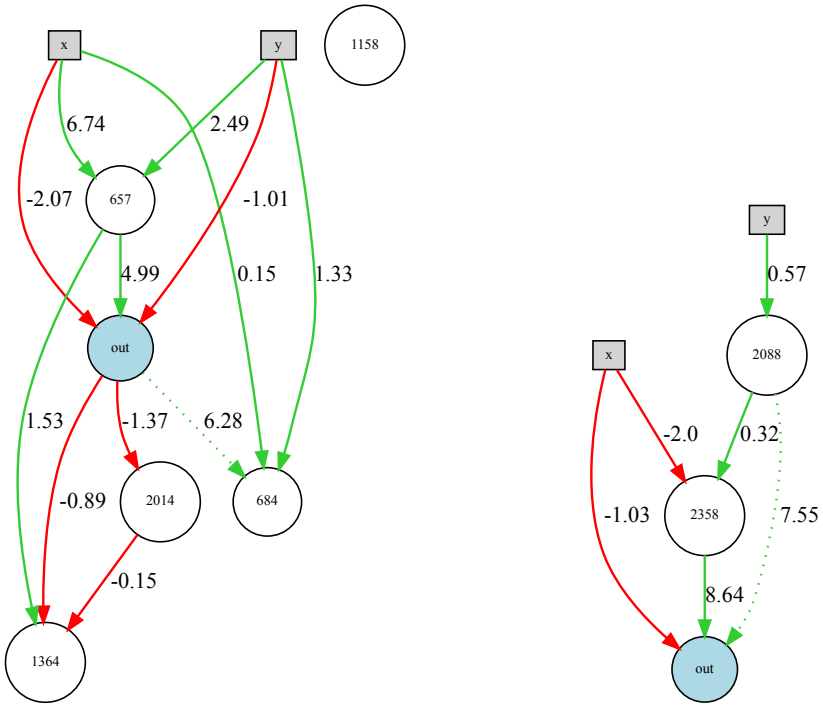


(a) End fitness = 0.



(b) End fitness = -0.25.

Figure 10.2: N1 networks evolved on XOR with CT = 3.0, CWC = 1.0, CDC = 1.0, and ST = 1.0 at generation 221.

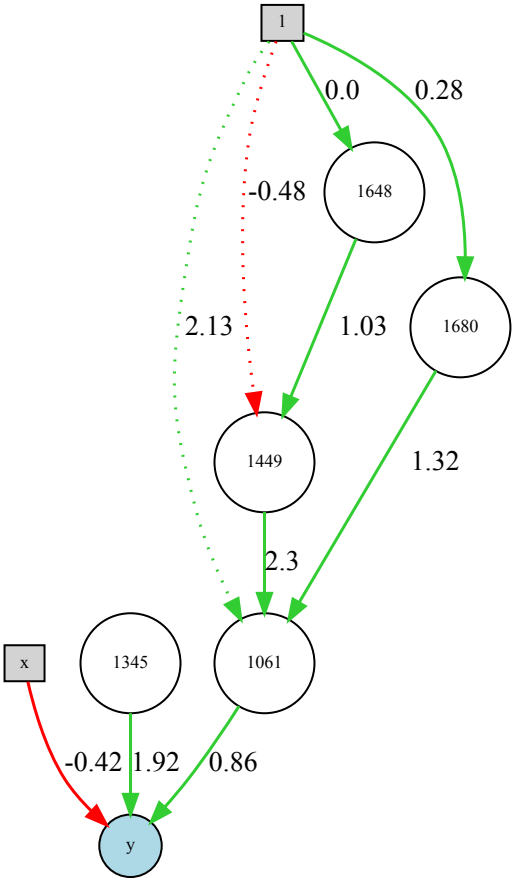


(a) End fitness = -0.01.

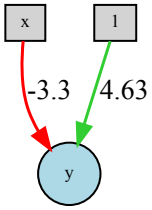
(b) End fitness = -0.43.

Figure 10.3: ANNs evolved on XOR with CT = 2.5, CWC = 0.5, and CDC = 1.0 at generation 146.

10.2 Cosine

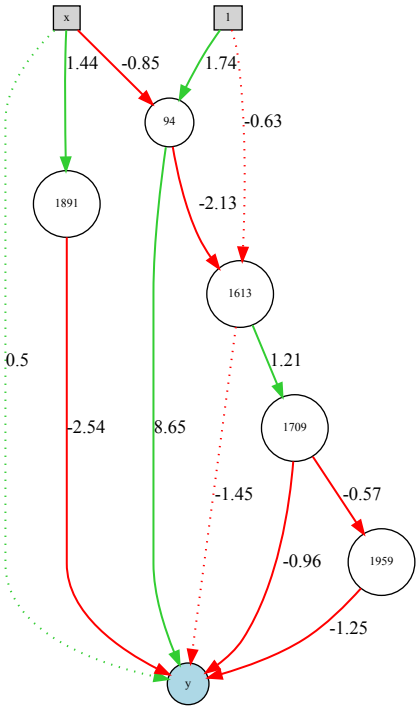


(a) End fitness = -4.46.

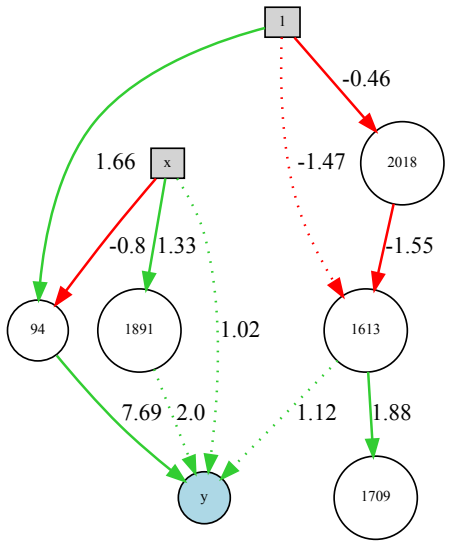


(b) End fitness = -10.19.

Figure 10.4: N0 networks evolved on cosine with CT = 2.0, CWC = 1.0, CDC = 0.5, and ST = 0.75 at generation 137.

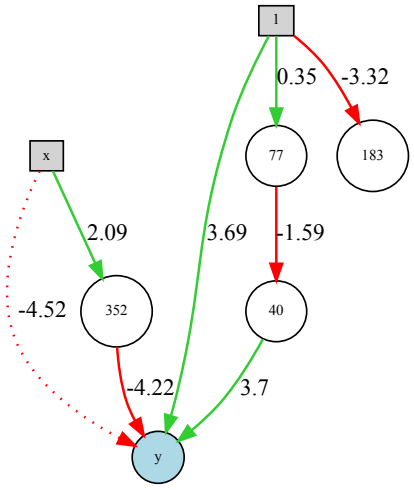


(a) End fitness = -5.59.

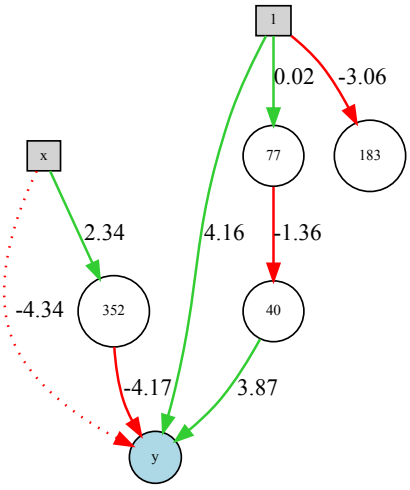


(b) End fitness = -5.80.

Figure 10.5: N1 networks evolved on cosine with CT = 2.0, CWC = 0.5, CDC = 0.5, and ST = 1.0 at generation 119.



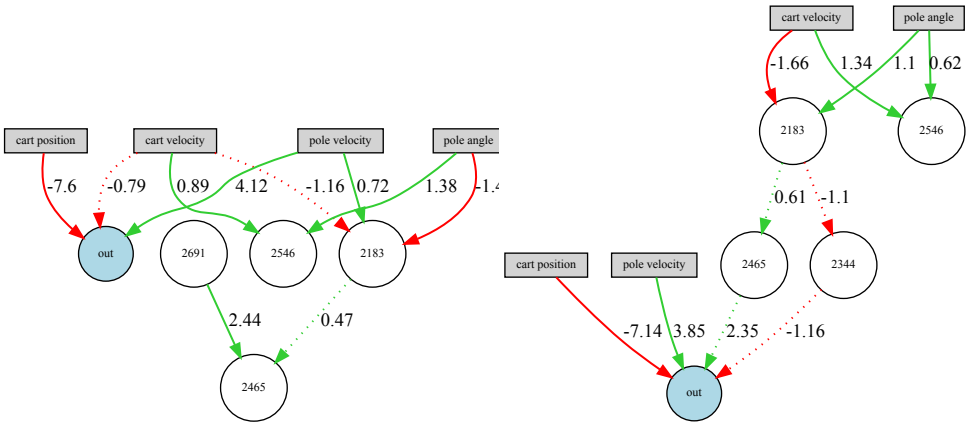
(a) End fitness = -0.43.



(b) End fitness = -1.9.

Figure 10.6: ANNs evolved on cosine with CT = 2.0, CWC = 0.5, and CDC = 1.0 at generation 41.

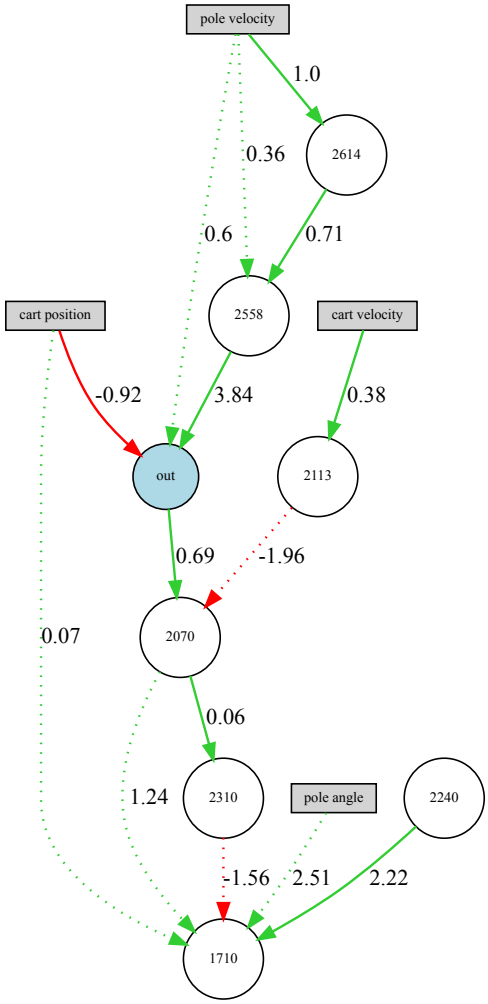
10.3 Single Pole Balancing



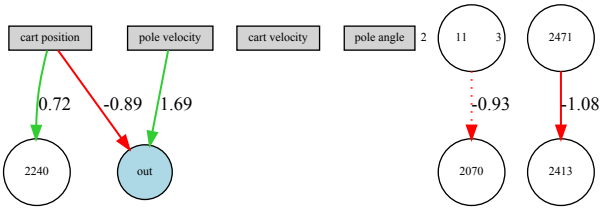
(a) End fitness = 1.25.

(b) End fitness = 0.80.

Figure 10.7: N0 networks evolved on SPB with CT = 2.5, CWC = 0.5, CDC = 0.5, and ST = 0.75 at generation 149.

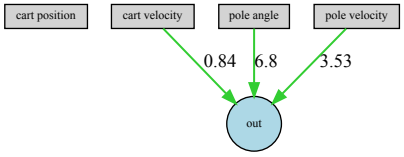


(a) End fitness = 1.30.

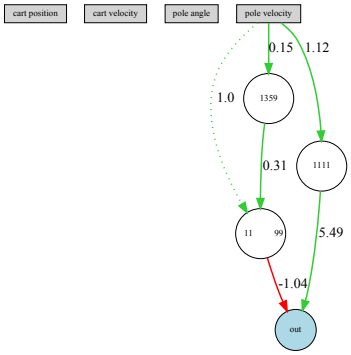


(b) End fitness = 0.75.

Figure 10.8: N1 networks evolved on SPB with CT = 2.0, CWC = 0.5, CDC = 0.5, and ST = 1.0 at generation 128.



(a) End fitness = 15.00.



(b) End fitness = 1.15.

Figure 10.9: ANNs evolved on SPB with CT = 2.5, CWC = 0.5, and CDC = 1.0 at generation 93.