Chapter 1

Lucas Dreyer

1 Introduction

Artificial Neural Networks (ANNs) are used in machine learning because they can be trained to recognize handwriting or speech, control vehicles, or make predictions based on data. Developing techniques to construct and train ANNs quickly and effectively is part of ongoing research [27, pp. 1]. There are two main approaches in the training of ANNs: supervised learning with the use of labelled training data and using evolutionary algorithms to evolve the ANN, like neuroevolution [10]. Neuroevolution traditionally works with fixed topology networks, but one branch of neuroevolution allows adaption of the network. One of the core ideas behind this type of neuroevolution is that the initial network is minimal, and then additional structures will grow as necessary. This contrasts with nature, where there is no requirement that complexity should increase. Adaption by any means is the main driver, leading to a higher chance of reproduction. Indeed, there is evidence that biological systems can benefit from reducing complexity, a process called simplification . There are many examples of simplification of organisms due to natural selection. Simpler organisms can reproduce faster and therefore adapt faster, because they require less resources to grow and reproduce. Most animals that adapt to caves lose eyesight or pigmentation. A good example is the Mexican tetra or blind cave fish that lost its eyes and pigmentation during adaption [13, pp. 1]. The normal form still exists on the surface. Parasites also tend to lose complexity; they can often lose functions that they can get from the host. Indeed mites and their hosts have parallel evolution, but in opposite directions [9, pp. 1273].

Analogous to biological systems, simpler neural networks have several benefits: reduced computing time needed to evaluate and search for solutions; solutions are less prone to overfitting; and solutions with redundant sections are much harder to understand and reason about. [There are two main approaches in the training of ANNs: supervised learning and reinforcement learning. Supervised learning consists of finding a relationship between[37]]

1.1 Motivation

NeuroEvolution of Augmenting Topologies (NEAT) [32, pp 3] is an effective method of doing neuroevolution. One of the features of this method is complexification: starting with a minimal structure the ANN grows in complexity over time, similar to what happens to evolving organisms in nature.

1.2 Old Motivation

NeuroEvolution of Augmenting Topologies (NEAT) [32, pp 3] is an effective method of doing neuroevolution. One of the features of this method is complexification: starting with a minimal structure the ANN grows in complexity over time, similar to what happens to evolving organisms in nature. At every new generation, some percentage of the netwoks will increase in complexity by either linking two unlinked vertices or by splitting an edge in two, adding a new vertex in the middle. Therefore, the population's average fitness will increase at every generation, even if the average fitness remains the same. If the problem domain is very complex it may take many generations before NEAT finds a new maximum fitness. The network with the highest fitness, the champion, will maintain its complexity during this process. In order to preserve the record fitness, one identical copy of the the champion's genome is carried over to the next generation. The champion might even become the least complex network over time, because its genome won't change as all others steadily increase in complexity. When a new champion is found, its complexity will probably be close to the average of the population complexity[1]. Another effect that increases complexity is a feature that prevents stagnation: if a species does not improve its fitness for a set number of generations, it gets a fitness penalty, which eventually leads to it being eliminated. This effect applies to the champion, so in problems where progress is slow the champion might be eliminated and average complexity will again increase.

The combination of these effects cause the genome to inevitably increase in complexity, even while maximum fitness stagnates or even decreases. Unnecessary complexity greatly increases the search space and slows the computing time required to activate the network, especially in the case of recurrent connections. Clearly there is a need to curb the proliferation of this unnecessary complexity; this can be done by pruning the neural network.

[1] Not always true: if the mutation rates are too high, it's possible that the new champion is always a direct descendant of the previous champion. Therefore, assume that the evolutionary search is working properly.

1.3 Research Goals

The goal is to contrast complexification and simplification methods in ANNs across a range of tasks of increasing complexity.

- Which pruning mechanism is preferred.

We investigate these questions in the context of NEAT, and look at a general way to improve NEAT's performance by pruning.

1.4 Old research Goals

In the context of neuroevolutionary pruning there are two important questions:

- Duration: when should pruning start, and how long should it continue? Should pruning and complexification happen simultaneously?
- Mechanism: how should we determine the edge to be pruned? Should it be random or is there a better way?

I investigate these questions in the context of NEAT, and look at a general way to improve NEAT's performance by pruning.

1.5 Approach

NEAT uses complexification as part of its search. At every new generation, some percentage of the netwoks will increase in complexity by either linking two unlinked vertices or by splitting an edge in two, adding a new vertex in the middle. Therefore, the population's average fitness will increase at every generation, even if the average fitness remains the same. If the problem domain is very complex it may take many generations before NEAT finds a new maximum fitness. The network with the highest fitness, the champion, will maintain its complexity during this process. In order to preserve the record fitness, one identical copy of the the champion's genome is carried over to the next generation. The champion might even become the least complex network over time, because its genome won't change as all others steadily increase in complexity. When a new champion is found, its complexity will probably be close to the average of the population complexity[2]. Another effect that increases complexity is a feature that prevents stagnation: if a specie does not improve its fitness for a set number of generations, it gets a fitness penalty, which eventually leads to it being eliminated. This effect applies to

[2] Not always true: if the mutation rates are too high, it's possible that the new champion is always a direct descendant of the previous champion. Therefore, assume that the evolutionary search is working properly.

the champion, so in problems where progress is slow the champion might be eliminated and average complexity will again increase.

NEAT is applied to increasingly complex problems in order to compare the pruning methods in a comprehensive way: XOR, dual pole balancing, robot duel, tic-tac-toe, and Mario. For each domain we compare several pruning methods to the original complexification paradigm.

The thesis consists of 10 chapters, grouped into four parts: Introduction and Background (Chapters 1 and 2), Pruning Methods (Chapter 3), Applications (Chapters 4, 5, 6, 7, 8), and Discussion and Conclusion (Chapters 9 and 10).

2 Background

This chapter provides the background to the core components of the thesis: Artificial Neural Networks, NeuroEvolution of Augmenting Topologies and pruning methods. The first section describes the foundations of machine learning and Artificial Neural Networks. The next sections describes genetic algorithms and the combination of the two as Neuroevolution. The final two sections expands on NeuroEvolution of Augmenting Topologies and pruning of neural networks.

2.1 Machine learning paradigms

[TODO: format definition of ANN, explain Supervised/Unsupervised/Reinforcement learning.]

2.2 Artificial Neural Networks

An ANN is a network of processing elements called nodes or neurons, originally inspired by biological nervous systems [19, pp. 2-7]. The nodes are typically structured as layers of nodes, designated as the input, hidden, and output layers. Each connection between two nodes has a direction and weight associated with it, representing the hierarchy and influence. We categorize ANNs into feedforward or recurrent networks. In feedforward networks, signals start at the input and move strictly closer to the output at every step. Recurrent networks contain connections that form cycles by allowing a signal to flow in the opposite way: from closer to the output to further from the output. A feedforward ANN is therefore a directed *acyclic* graph, while a recurrent ANN is a directed *cyclic* graph.

Every node, except those in the input layer, produces an output that is some function of their input signals. This function is usually a linear combination of the inputs and their associated connection weights, fed into an activation function. We can express the *kth* node as:

$$y_k = \varphi\left(\sum_{j=1}^{m} w_{kj}x_j + b_k\right) \tag{2.1}$$

where x_1, x_2, ..., x_m are the input signals; w_{k1}, w_{k2}, ..., w_{km} are the connection weights, b_k is the bias; φ is the activation function; and y_k is the output signal [16, p. 33]. The activation function is usually a non-linear function, like the sigmoid function $\frac{1}{1+e^{-x}}$ or the hyperbolic tangent function $\frac{2}{1+e^{-2x}} - 1$.

ANNs have several useful properties:

- Computational power: A large enough recurrent ANN can simulate a universal Turning machine [29, p. 1].
- Universal function approximator: A feedforward ANN with a single hidden layer can approximate any continuous function of n real variables.[5, p. 1]

Training: Backpropagation places several requirements on the ANN. The activation function must be differentiable.

2.3 Genetic Algorithms

A genetic algorithm (GA) is a search procedure inspired by the biological process of natural selection [17]. GAs maintain a collection of candidate solutions called a *population*, which evolves over time. Each solution is represented by a set of *chromosomes*, or a *genotype*. The expression of the genotype as a solution is called a *phenotype*. The population evolves by going through an iterative process where each iteration is called a *generation*. During each generation every phenotype is evaluated as a solution, and assigned a *fitness* as a measure of how good the solution is. The next generation's population is selected with more copies of the fittest genotypes. This process will usually continue until it achieves either the target fitness or reaches a certain number of generations.

The genotype is represented by a set of strings and can be changed by genetic operators like *mutation*, *crossover* and *selection*. Selection refers to the preference given to fitter individuals when composing the next generation. This ensures that better solutions are promoted, while poorer solutions are eventually discarded. Crossover refers to the mixing of two or more parent's genotypes to form a child, analogous to biological sexual reproduction. A common method of doing crossover is single-point crossover: we choose a random crossover point in the genome, and the resulting genotype inherits from the first parent up to the crossover point and the rest from the second parent. Mutation is a way to introduce more diversity by making random modifications to the genotype, for example by flipping a bit in the genotype's representation. Not all new genotypes have to be a result of crossover and mutation, some may be exact clones of and existing genotype. The solution with the highest fitness, called the *champion*, is often cloned to ensure that progress isn't lost by mutation.

2.4 Neuroevolution

Neuroevolution (NE) is the use of evolutionary algorithms to evolve artificial neural networks, including genetic algorithms, evolutionary strategies and evolutionary programming [37]. Other optimization techniques like particle swarm optimization have also been used [7]. Typically the

ANN represents the phenotype, and a definition of the network is the genotype. There is a lot of variation in the amount of information stored in the genome, depending on the encoding scheme used. The first NE methods [35] used a user-defined topology knows as direct encoding. Direct encoding, also known as *Conventional Neuroevolution* (CNE), means there is a direct mapping between the network structure and the genome: every neuron and connection is associated with a specific value in the genome. Algorithms that use direct encoding are divided between two main groups. Fixed-Topology systems work with a predefined structure and usually evolve only the weights; some examples of this method is EPNet [36]. Because the whole network is encoded as a list or vector of real numbers, any optimization algorithm can be used, for example very efficient algorithms like Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [14].

Despite the success and simplicity of CNE, the main drawback is that the network size and layout, or topology, has to be chosen beforehand. If the network layout is larger than required, the time to search for the solution will take much longer than necessary. It may also adapt to the noise in the test input and fail to generalize (also known as overfitting, this is explained below). If the network layout is smaller than required, the search will never arrive at a solution. Thus the creation of *Topology and Weight Evolving Neural Networks* (TWEANNs) which evolve both. Some examples of this method is GNARL [2], *NeuroEvolution of Augmenting Topologies* (NEAT) [31], and EANT [28].

Indirect encoding utilizes more compact ways to represent the ANN: Every aspect of the ANN can be defined as part of the genome and therefore subject to evolution: the connection weights, the structure or topology of the network, the choice and behaviour of the activation function, or even experiment hyperparameters.

Neuroevolution has several advantages over algorithms like backpropagation (BP) that rely on gradient descent:

- BP require that the activation function be differentiable.
- Gradient descent methods are prone to getting stuck in local minima. If the error function is nondifferentiable, it may never find the global minimum.
- Training recurrent ANNs with BP is slower than feedforward ANNs.
- BP has difficulty with reinforcement learning domains, because there is no output target to compare with the model's output.

2.5 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) is a system for constructing TWEANNs that presents solutions for several common problems with TWEANNs [31].

2.5.1 Genetic Tracking

One of the problems with NE is *Competing Conventions*, or also known as the *Permutations Problem* [24]. Competing conventions refers to the fact that there are many way to express a neural network that satisfies a particular solution.

For example, take a fully connected feedforward neural network with three layers: three inputs (A, B, C), three hidden nodes (D, E, F) and one output (G). A simple encoding scheme might express this network as $[ABC, DEF, G]$. Notice that the three hidden nodes can be reordered as (F, E, D), while keeping their connections intact, without affecting the mathematical behaviour of the neural network. But now the structural definition has changed to $[ABC, FED, G]$! A crossover operation might produce a network like $[ABC, DED, G]$, losing half of the connection weights and probably most of the parent's fitness. As if this wasn't bad enough, the hidden nodes can take on any of the six ($3! = 6$) permutations possible by reordering that layer without changing network functionality. Stanley [32, p. 19]calculates that in this simple network the probability of offspring with severe genetic damage is 66.6%. The problem dramatically worsens in larger networks as the number of permutations increase in a combinatorial explosion.

Radcliffe [24] and Thierens [34] presented solutions to this problem, however they required some assumptions about the topology that do not hold for TWEANNs. Thus, the part of this problem specific to TWEANNs is called the *Variable Length Genome Problem*. Initial solutions to this problem revolved around a trade-off between genome compatibility and representational ability. One style focused on minimizing the damage induced during crossover by doing complex graph analysis or imposing structural restrictions [6]; [21]. The problem with this approach is that just because parent's subgraphs are topographically similar, it does not mean are functionally equivalent or even similar. If topologies diverge due to the competing conventions problem, a functional combination might not exist at all. Additionally, the topographical comparison required is computationally expensive. Another approach to the variable length genome problem is to completely disregard crossover and do only mutation [1].

NEAT addresses this problem without a trade-off between compatibility and representational ability. NEAT maintains a *global innovation number* that is incremented after being assigned to every new gene that appears. The innovation number therefore serves as an indication of the chronology of appearance of all genes. If the same mutation happens to more

than once individual during a generation, they receive the same innovation number. During crossover, NEAT knows which genes match and can then line up the genomes. Genes that appear in both parents are called *matching* genes. Those that don't match but are within the parent's range of innovation numbers are are called *disjoint*, and those outside the range are called *excess*. During crossover the matching genes are randomly chosen from either parent, while the disjoint and excess genes are included if they belong to the fitter parent [31, p. 10]. This approach effectively solves the competing convention problem with very little overhead.

2.5.2 Speciation

New mutations take some time to start adding to the fitness of the individual; it's highly unlikely that a new connection with its randomly initialized weight happens to immediately function. It might even slightly reduce the individuals' fitness, which in turn will strongly discourage the mutation required to form new functional structures. As a result networks need some time to optimize their new and old structure after a mutation.

GNARL [1] handled this problem by adding a disconnected node as a frame to support the formation of future connections. Over time this runs the risk of polluting the genome with useless bloat.

NEAT addresses this problem by creating niches or *species* of topographically similar networks, using the genetic tracking described above. It defines a measure of compatibility distance based on the number of disjoint and excess genes between two genomes. A high number of disjoint genes imply a big difference in evolutionary history and therefore probably functionality. The compatibility distance δ is defined as follows:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 D \tag{2.2}$$

E is the number of excess genes, D is the number of disjoint genes, W is the average weight difference between the matching genes. The coefficients $c1$, $c2$, and $c3$, allows control of the relative weight of these three factors. These coefficients are also know as the excess coefficient ($c1$), the disjoint coefficient ($c2$), and the disjoint coefficient ($c3$). The factor N is the number of genes in the larger genome and normalizes the first and second terms.

In order to separate genomes into species the first step is to define a compatibility threshold δ_t. Next a genome from each species in the previous generation is randomly chosen as a representative r of that species. Each new genome g is compared to each r, and g joins the first species for which the compatibility distance between g and$r$ is less than δ_t. If g is incompatible with all species then it forms a new one.

NEAT utilizes *explicit fitness sharing* to ensure species stability and diversity. Fitness sharing

forces related individuals to share their fitness payoff. Fitness sharing was originally done implicitly by grouping individuals by fitness [17]. Explicit fitness sharing works within species, so all members of a species share their fitness payoff [11]. This prevents one species from taking over, because the penalty of the species size offsets what would have been a big increase in offspring due to the high fitness. Explicit fitness sharing is implemented by adjusting the fitness f_i' for each organism i according to its compatibility distance δ from every other other organism j in the population:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{2.3}$$

The sharing function sh is set to 0 when $\delta(i,j) > \delta_t$, else it is set to 1 [26, p. 1]. The denominator is therefore just the number of organisms in the same species. This means that at every new generation each species is assigned a number of organisms in proportion to the adjusted fitness f_i'.

2.5.3 Initializing Populations

Many TWEANNs initialize a random set of starting networks as a way to create enough diversity so that solutions don't focus on one point in the search space. This can lead to networks with no path from the inputs to the output. More importantly, it dramatically enlarges the search space by increasing the number of parameters the search algorithm must consider. There is no fitness cost to increasing the size of a network, so many TWEANNs will simply keep growing in size.

Some TWEANNs include the size of the network in the fitness function to try an alleviate this problem [38]. This will surely encourage smaller networks, but this again places a subtle restriction on the topology. Some problems that call for a large network will likely never be found by this algorithm. Another approach is to add pruning to the reduce the complexity *after* creating the unnecessary nodes and connections [23].

NEAT ensures a minimal structure by starting out with only a fully connected input and output layer; no hidden nodes. The important difference is that not only will the final network be minimal, but all intermediate representations will also be minimal, effectively reducing the dimensionality of the problem. NEAT does not need the diversity boost of random starting structures, since the speciation provides it.

2.5.4 Internal structure

The genome consists of a set of connection genes, each representing a directed connection between two node genes. Each connection gene has the following properties: its in-node, its out-node, whether the connection is enabled, and its innovation number.

2.6 Pruning

2.6.1 Overfitting

In any machine learning task it is important to consider the performance of the learned model on data outside the training set. Here an increase in performance refers to a smaller error in the case of supervised learning, and a greater fitness in GAs. As long as the model has sufficient complexity (meaning the degrees of freedom, weights, or other parameters that the optimization process can tune) the performance on the training set will continuously improve during training. A model of sufficient complexity can eventually perform with 100% accuracy on a training set. However at some point the performance in the test set, also known as the holdout set, starts to increase while the performance in the training set still improves. This is the point where overfitting starts to happen. Overfitting means the model has started inferring rules about the training data or environment that does not hold true in the general case [25].

A simple way to avoid overfitting is to divide the training data into training and validation sets. The validation set is then used as a proxy for the test set, and the training process can halt when the performance on the validation set stops increasing or starts decreasing. The downside to this approach is it reduces the amount of data available for training; this many not be possible if the amount of data is small. *k-fold cross-validation* extends the idea of splitting the training set. Also called rotation estimation, the dataset D is split into k mutually exclusive subsets, or *folds*, of approximately equal size. The model is then trained and tested on each of the k folds. The cross-validation estimate of accuracy is then the average accuracy over all k runs. Commonly used values for k include 10 and 20. [20].

Another approach is to limit the model complexity or degrees of freedom. Overfitting happens because the model is underconstrained by the training data; generally speaking there are more model parameters than training samples. Smaller models are also quicker to train and may be easier to analyze and understand. This supports NEAT's principle to initialize minimal networks. This lead to the question: what is the smallest possible network that can adequately fit the data?

Formal learning theory has been utilized to place bounds on the model complexity [8][4]. Unfortunately these bounds do not apply to networks with multiple continuous outputs and

they say nothing about the architecture required.

Therefore, a possible approach to overfitting and unnecessary connections in TWEANNs is to prune the network structure when they arise. A number of pruning methods are discussed in the next section.

2.6.2 Bloat

Bloat is a significant problem in Genetic Programming [22], but it also applies to methods that use variable-length representations like NE. Bloat is the growth of the genome's size without a corresponding improvement in fitness [30]. Bloat can cause problems like: (1) it significantly increases the computing time required to evaluate solutions due to the cures of dimensionality [3]; (2) solutions take up more computer memory; (3) the effectiveness of the genetic operators like crossover and mutation decrease because they lead to no correspond change in fitness; (4) solutions with redundant sections are much harder to understand; (5) redundant structures exacerbate the NE problem of competing conventions because it effectively grows the genotype without measurable prototypical change.

This relates to the reasoning behind NEAT's speciation: it's important to provide evolutionary time for new structures to be calibrated. If the genome is large and the search space is complex, it might take a long time for mutations to randomly calibrate the new set of genes. But what if that new structure never results in any fitness benefits? If the new set of genomes has been around for several generations, it's likely that it's spread outside the specie, and will likely live on in the population as a whole even if that species dies off. This becomes a bigger problem in domains where the starting genome is larger and the search runs for a long time.

2.6.3 Pruning Methods

There are a large amount of techniques for pruning ANNs. Sensitivity-based methods compare the error with and without each node and remove nodes if that difference drops below a certain threshold. Penalty-term methods rely on modifying the error function such that some node's weights decrease to eventually reach zero [25]. However most rely on an error function and BP, and is therefore not usable in Neuroevolution. A summary of pruning techniques that are applicable in Neuroevolution:

Random node removal is used by both GNARL and EPNet [2, 36]. The number of nodes to delete per generation is small, between 1 and 3. Nodes are selected randomly, although the input and output nodes are sometimes excluded because it might break the network. A variation of this method is to consider the age of the node such that new nodes are less likely to be removed.

Random link removal is also used by GNARL and EPNet [2, 36]. The number of links is usually significantly larger, often bound by a parameter. Connections are selected randomly.

Absolute parameter values: Connections with a small absolute weight value are more likely to be removed. The idea is that if a connection weight is very small it can be approximated by zero, which is functionally the same as pruning the link. However in some cases some small parameters can still have a large influence on the behaviour of the network [27].

Constant output: Nodes that have constant output during training can be replaced by or incorporated into existing bias nodes.

Correlated nodes: If two or more nodes have highly correlated positive or negative responses over all inputs, then they can be combined into a single unit. All their output weights should be added together so the combined unit has the same effect on following units [25].

2.6.4 Pruning Policies

Blended search [18] is perhaps the simplest pruning policy. Pruning is always active alongside the normal complexification methods, pruning just happens at a lower rate in order to allow the network to grow.

Phased search [12] switches between distinct complexification and pruning stages. Phased search switches between the two phases based on a threshold defined as the starting mean population complexity (MPC, that is the average sum of number of nodes and connections) plus an experiment dependent threshold offset τ that ranges between 30 and 100. The search starts in a complexifying phase like classic NEAT, until the current MPC breaches the threshold the first time. However if the population fitness is still increasing, the pruning phase is delayed until the fitness stops increasing for a given number of generations. The pruning phase is similar to the complexification phase, only instead of adding nodes and links they are are removed. The only exception is that crossover is disabled because it can cause a genome to increase in complexity. The pruning phase is active until the MPC stops decreasing for a given number of generations, typically between 10 and 50. The threshold is then reset to the current MPC plus the threshold offset τ. The benefits of phase pruning are: (1) the complexification phase of the search is identical to NEAT, so it's easier to understand and reason about it's effectiveness since NEAT is a proven method; (2) the pruning completely removes all non-functional genes from the genome, effectively resetting the search at a new baseline.

Phased Searching with NEAT in a Time-Scaled Framework [33] **[TODO: explain]**

Power-Law Ranking Probability based Phased Searching (PLPS) [15] builds on phased search by biasing the random gene (node and connection) removal. Each gene's removal probability is based on a ranking that depends on the gene type. A connection gene's pruning is similar to

the absolute parameter values method described above: links with a smaller absolute value are more likely to be removed. Node genes are ranked based on the number of incoming connections: nodes with more connections are considered more important and are less likely to be removed. The probability of a node removal is $rank_n^{-1}$, where$rank_n$ refers to the rank of that node in an ascending list of all nodes' number of connections. The probability of a link removal is $rank_l^{-1}$, where $rank_l$ refers to the rank of that link an ascending list of all links' absolute weight values. The benefit of PLPS is due to it's targeted selection it can remove redundant genes much faster.

2.7 Conclusion

[TODO]

3 Methods

In this chapter we explain and compare the different pruning methods.

3.1 Overview

All of the pruning methods consist of switching between two stages: *complexification* and *simplification*. The *complexification* stage means everything works like classic NEAT: specifically as part of every generation, each genome can undergo one or more of reproduction or mutation. In classic NEAT, there are three types of mutations: adding a new node by splitting a link in half, adding a new link connecting two nodes, or changing the weight of a link. Each of these have some probability of happening as defined by the experiment parameters.

The *simplification* or pruning phase is the same as the *complexification* phase, except structural mutations that increase complexity are not allowed. This refers specifically to the first two mutations mentioned above: adding a node or adding a link. Instead, during simplification we only allow one mutation: removing a link. The probability of this mutation is usually double the link add mutation probability.

There are many different ways to decide when and how we do simplification. We discuss these strategies in section 3.3,

3.2 Changes relative to original NEAT

We also made some improvements to the basic NEAT algorithm. Improving the basic algorithm was not the goal of this project, but it was necessary due to specific problems encountered.

3.2.1 Unstable speciation

One problem was unstable speciation. NEAT relies on speciation to protect innovation, so the number of species should be at least two, but the number of organisms in a species should also be at least two. Ideally there should be a balance between the two, for example having the number of species close to the square root of the population size. This will ensure the average number of organisms per species is also the square root of the population size.

Unfortunately, in some experiments the number of species either exploded to equal the population size, or collapsed to one. In NEAT the number of species indirectly depends on the *compatibility distance* metric, which is in turn based on four coefficients: the excess coefficient ($c1$), the disjoint coefficient ($c2$), the disjoint coefficient ($c3$), and the compatibility threshold (δ_t). The calculation is shown in equation 2.2. is This means we could not explicitly set a fixed number of species, because the number of species is the result of the speciation process.

To remedy this we pick the desired number of species and do a binary search for the new compatibility threshold ($\hat{\delta}_t$) that results in that number of species. The initial bounds on $\hat{\delta}_t$ are $[0, 100]$ and, the search terminates once the bounds are less than 0.01. This means we evaluate what the number of species are going to be given at most 14 times ($\log_2(100/0.01) = 13.23$). Each evaluation takes time proportional to the population size. Therefore this search is fast enough to take an insignificant amount of time in relation to the rest of the NEAT algorithm: For example the spawning of new organisms takes time proportional to the population size multiplied by the average size of the genome.

3.2.2 Uncontrolled population growth

NEAT assigns an expected number of offspring to each organism, based on its fitness divided by the average fitness. This number will usually have a fractional component. When deciding the size of each species, this fractional residue is added up, but in the end the number of organisms and species must be an integer, so a naive solution might be to round off the fractional part. But then it's possible that the number of organisms doesn't match the population size at the end of the epoch. For example: if the population size is 10, and there are three species, each with a total expected offspring of 3.333. The expected offspring would be rounded down in each case and the population size would incorrectly drop to 9.

The solution is to accumulate the fractional part if the number is rounded off. However due to inherent inaccuracies in floating point arithmetic, the original implementation of NEAT had a bug where the number of organisms would slowly increase above the population size. The rate was about one organism every four generations. This was especially noticeable in experiments with a large number of generations, which would end up with two times the number of organisms as the population size. While there are some benefits to increasing the population size over generations, it significantly slows down the run time and it was clearly a bug. We fixed it by adding an error margin of 10^{-9}, commonly called epsilon or eps., to the comparison.

3.2.3 Plausibility

[Talk about the pruning plausibility experiment, where we took an average complexity champion from each experiment and run simplification on it to find a smaller solution.]

3.2.4 Software

[talk about the foundation of NEAT]. We based our code on Kenneth Stanley's original NEAT

C++ implementation. We added several features, like a GUI for the robot duel experiment, multithreading, statistical summaries, and a thread-safe pseudorandom number generator. The multithreading simply runs multiple experiments in parallel, so we didn't have to modify existing experiments too much to get the benefit of multithreading.

3.3 Pruning strategies

3.3.1 No Pruning

The first method we include in our comparison is just classic NEAT. Specifically as part of every generation, each genome can undergo one or more of reproduction or mutation. In classic NEAT, there are three types of mutations: adding a new node by splitting a link in half, adding a new link connecting two nodes, or changing the weight of a link. Each of these have some probability of happening as defined by the experiment parameters.

3.3.2 Only Pruning

[Discuss the investigation into it, even though we ended up not doing it]

We considered a strategy of doing only pruning: the network starts with a maximal configuration, and gradually prunes away all unneeded connections. There are a number of problems with this approach:

- How do you decide on the size of the maximal network? Depending on the problem, we can use the Vapnik–Chervonenkis (VC) dimension [need ref] measure to match the upper bound of the network with the function we are trying to replicate. This gives rise to its own problems: What if we don't know understand the target function well enough to set the required VC dimension? This is likely the case with most reinforcement learning problems. Another problem is the VC dimension is a theoretical upper bound, it does not mean the resulting network size will be practical.
- The curse of dimensionality [need ref] means the large starting network will take exponentially longer to train, requiring the use of use of specialized techniques (like residual nets), which are out of scope of this project.

Even so, we tried a purely pruning solution, but it was to slow that we could not hope to run it enough to do useful comparisons with it. For comparison: with 100 hidden nodes, one generation of the XOR experiment took 100 seconds. Usually this takes 6 ms, so it's 16667 times slower.

3.3.3 Random Pruning

Random pruning starts off with the minimal network, and then switches between complexification and simplification with a default of 50% chance of picking either method. Another variation of this method simultaneously allows *complexification* and *simplification* mutations simultaneously. This is a little more realistic because it sounds more biologically plausible. However we did some testing of the simultaneous paradigm and found it performed significantly worse than the discrete system.

One possible explanation is the synchronized changes between methods helps the complexity move in a specific direction, while in the simultaneous approach the *complexification* and *simplification* cancel each other out to a larger extent. This can happen during crossover, when parents' genomes are mixed. If one parent has just lost a gene, and the other parent has just gained one, the change in the number of genes cancel each other out in the offspring. Regardless of the reason, the next method increases the structuring of the evolution policies.

3.3.4 Static Phased Pruning

Taking random pruning as a baseline, the next method is static phased pruning. The main problem with random pruning is there are no hard limits on the number of pruning generations, or the duration of a pruning cycle. Over a large number of runs the number pruning generations will average out, but a specific run can be very unbalanced one way or the the other, especially if the number of generations are small. To remedy this we set a fixed schedule to the length and proportion of complexification and simplification phases.

As mentioned in the previous section, we suspect that quickly alternating the phases will lead to worse performance. Another reason why this might happen is that the network might need some time to "recover" after pruning. It's reasonable that some link weights will have to be recalibrated to compensate for the removed link.

As such we can draw a direct comparison to 50% random pruning by doing one generation of complexification for each generation of simplification. We will refer to this arrangement as a 1:1 static cycle, with a cycle length of 2. Keeping with the same ratio, we can lengthen the cycle to 2:2 or 3:3 and compare it to random pruning.

Stemming from the need for recovery after pruning, it might make more sense to balance the ratio in favour of complexification. We consider a 7:3 cycle to be a good balance between complexification and simplification.

3.3.5 Dynamic Phased Pruning

A main problem with random and static phased pruning is that they might not prune the right amount, or at the right time. It's reasonable that most searches would need more pruning at a later stage, but maybe not as they're approaching the target fitness. Likewise, a longer test might need a longer recovery time between pruning phases. Dynamic pruning aims to remedy this problem by switching between complexification and simplification as needed.

At the start of every generation, dynamic pruning relies on the dot product of a provided weight vector and the experiment parameters in Table 1 to determine if it should trigger a pruning phase. Specifically, if the following condition is satisfied:

$$\sum_{0}^{n} w_i p_i > 0 \tag{3.1}$$

with w as the weight vector, and $p$as the experiment parameters. If the pruning is triggered, it remains active for 5 generations. For those 5 generations, it cannot be triggered again. This ensures that factors that take several generations to change can contribute to the weight vector. If equation 1 is still satisfied 5 generations after the pruning started, it will start a new pruning cycle.

We calibrated the weight vector (Table 2) using an CMA-ES optimiser, with a number of generations used by a subset of experiments as the objective function. We discarded the seeds used by the pseudorandom number generator to avoid bias and overfitting. This is an important step often ignored.

3.4 Testing Methodology

[how do we go about testing the research questions?]

Question: what is the benefit of sticking complexification and simplification phases together? Is the notion of recovery useful to pruning methods?

Answer: we can measure this by comparing random pruning and phased pruning with the same ratio of complexification and simplification generations.

Question: how much pruning is needed:

Answer: We can determine this by changing the ratio of ratio of complexification and simplification generations in random pruning and phased pruning.

Question: which parameters of dynamic pruning lead to improved search performance by reducing the number of generations taken to reach a solution

Answer: We can determine this by testing each parameter individually.

Variable Name	Description
poc_num_species	Number of species
poc_bias	Generic bias, set to 1.0
poc_champ_gene_complexity_velocity	Rate of change of the champion's gene complex
poc_mean_gene_complexity_velocity	Rate of change of the average gene complexit
poc_mean_fitness_velocity	Rate of change of the average organism's fitne
pocn_champ_gene_complexity	Normalized champion gene complexity
pocn_mean_gene_complexity	Normalized average gene complexity
pocd_fitness_stagnation	How many generations has max fitness been the s
pocd_fitness_nonincreasing	How many generations has max fitness not increa
pocd_max_narrowing_fitness	How many generations has the difference between max and min fi
pocd_max_widening_fitness	How many generations has the difference between max and min f
pocd_stdev_narrowing_fitness	How many generations has the standard deviation of the fitnes
pocd_stdev_widening_fitness	How many generations has the standard deviation of the fitne

Tab. 1: Dynamic pruning factors. Rate of change is measured relative to only the previous generation

Variable Name	Weight
poc_num_species	0.5924609909592676
poc_bias	-1.163680590152447
poc_champ_gene_complexity_velocity	-0.7473259597432785
poc_mean_gene_complexity_velocity	0.8820273301727122
poc_mean_fitness_velocity	-0.35785250985239037
pocn_champ_gene_complexity	-0.03209651311100094
pocn_mean_gene_complexity	0.4846405427925076
pocd_fitness_stagnation	-0.107203217309849
pocd_fitness_nonincreasing	-1.1901862115885395
pocd_max_narrowing_fitness	-0.6158866080584453
pocd_max_widening_fitness	-0.07137114371905738
pocd_stdev_narrowing_fitness	0.24138105198502355
pocd_stdev_widening_fitness	0.4704918571768239

Tab. 2: Dynamic pruning weights

3.5 Summary

In this section we discussed the four pruning strategies that we will focus on. We will investigate them in the following chapters.

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Tab. 3: The XOR truth table

4 XOR Comparisons

4.1 The XOR Problem

The XOR problem tasks an agent to predict the outcome of the **exclusive or** logical operation, also know as XOR. It's a very simple simple operation, sometimes used as a test of an experimental setup because it's very easy to mentally inspect the state. Table 1 shows the XOR truth table.

XOR is useful as a first test for algorithms that evolve network topology and weights. The fitness calculation takes only a few CPU cycles, and a network requires at least one extra hidden node to solve the function. If an algorithm struggles to solve the XOR problem, it's unlikely to make significant progress in the other, more complex problems.

[TODO: citations] The XOR operation was first described by Boole in The Mathematical Analysis of Logic (1847), as part of the description of Boolean algebra. Zhang and Muhlenbein (1993) were one of the first to evolve a network topology capable of solving XOR. Pujol and Poli (1998) used evolutionary methods to find the topology and calibrate the weights.

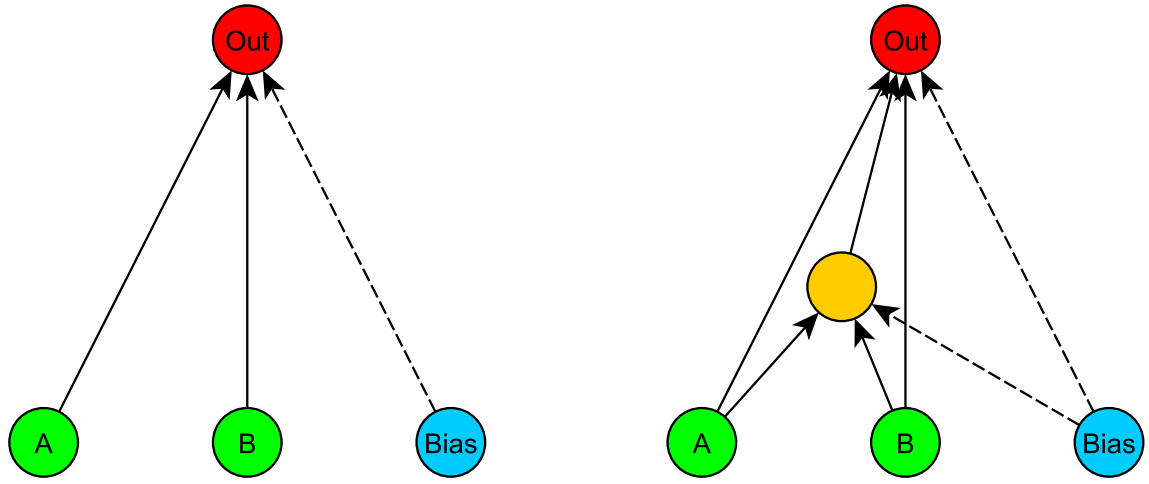
4.2 Task Setup

Due to it's simplicity, the task evaluation is easy to do in a single line of code. A single test involves applying one of the four possible inputs in the truth table to network's input. The signal propagates through the network and we measure the output in response.

We calculated the fitness by squaring the sum of one minus the error for all four input combinations:

$$f = \left(\sum_{i=1}^{4} 1 - e_i \right)^2 \tag{4.1}$$

The error is the difference between the correct output and the network's output. The resulting fitness is on the interval $[0\,..\,16]$.



(a) The initial network

(b) Network of the smallest solution

Fig. 4.1: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows the smallest possible solution.

4.3 XOR Experiments

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.1.

The experiment ran 1000 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 0.24 seconds, and the total running time was 16 minutes.

4.3.1 Other Methods

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning.

method	mean	min	max	std. dev.
Complexification	40.13	6.00	205.00	24.58
Random	83.33	9.00	845.00	84.79
Static Phased Pruning	52.36	6.00	573.00	40.39
Dynamic Phased Pruning	40.13	6.00	205.00	24.58

Tab. 4: Number of generations required to solve the problem

method	mean	min	max	std. dev.
Complexification	10.47	3.00	58.00	4.92
Random	9.73	3.00	25.00	3.20
Static Phased Pruning	9.91	3.00	41.00	3.92
Dynamic Phased Pruning	10.47	3.00	58.00	4.92

Tab. 5: Number of links in the network that solved the problem

4.3.2 Results

[TODO: add graphs] On average Complexification and Dynamic Phased Pruning took significantly fewer generations to solve the task (Table 2). However, Random and Static Phased Pruning ended with slightly smaller networks (Table 3). Random Pruning took the longest to finish, but it did have the smallest networks on average. This suggests that with a less aggressive and more structured pruning method we could improve the average number of generations required. Static Phased Pruning attempts to do this, and seems to successfully trade off some time for increased average complexity.

4.3.3 Summary

The XOR task is very simple, but despite that it has given us some insight into the different pruning methods.

method	mean	min	max	std. dev.
Complexification	6.53	4.00	21.00	1.72
Random	6.57	4.00	18.00	1.65
Static Phased Pruning	6.50	4.00	24.00	1.77
Dynamic Phased Pruning	6.53	4.00	21.00	1.72

Tab. 6: Number of nodes in the network that solved the problem

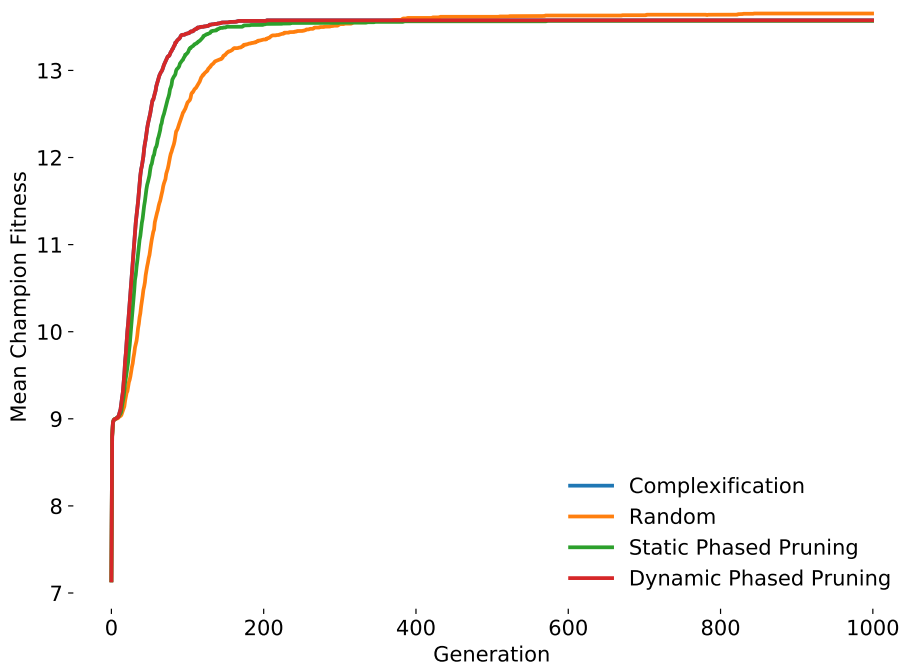


Fig. 4.2: Mean champion fitness over time for each pruning strategy

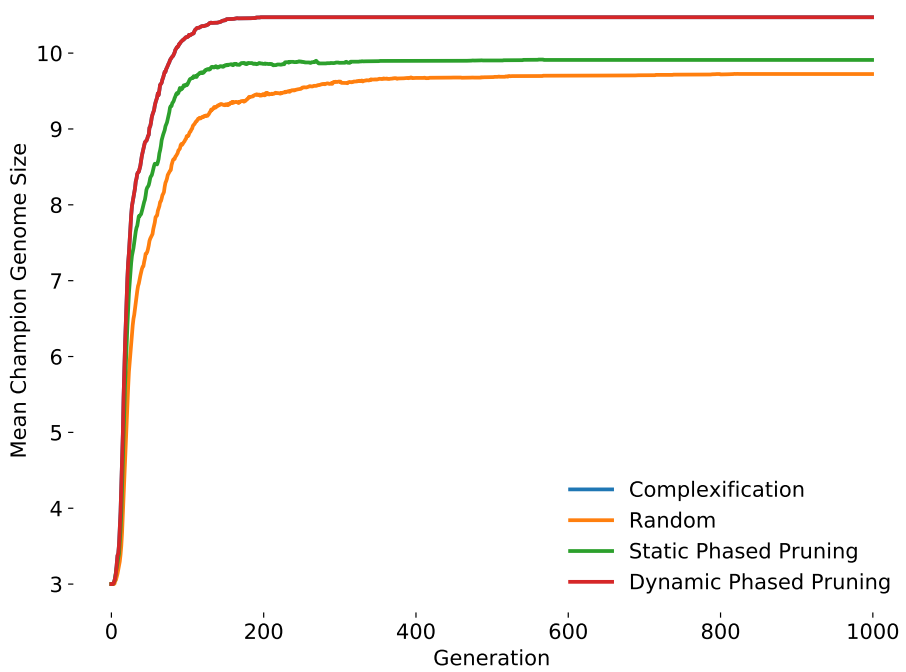


Fig. 4.3: Mean champion genome size over time for each pruning strategy

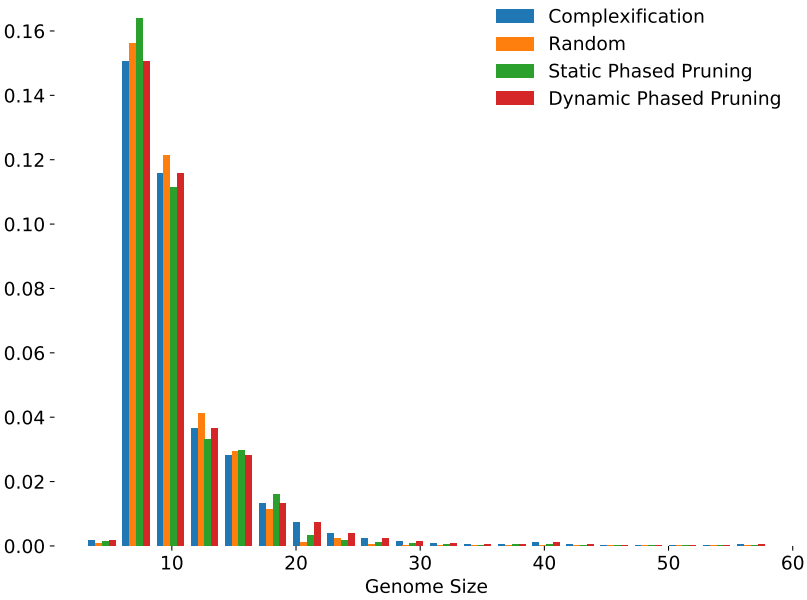


Fig. 4.4: Champion complexity in terms of genome size for each pruning strategy

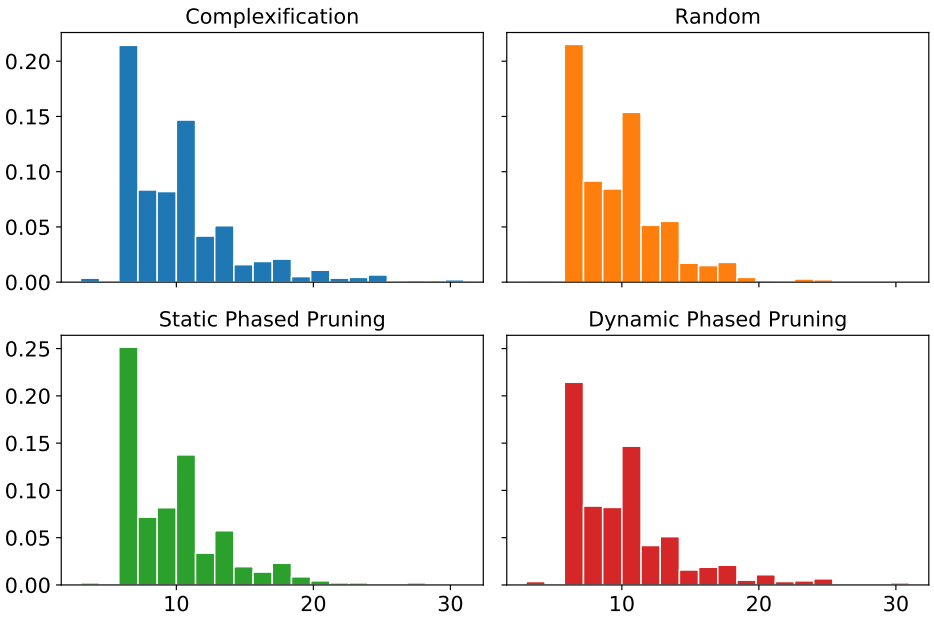


Fig. 4.5: Champion complexity in terms of genome size for each pruning strategy

5 Pole Balancing Comparisons

5.1 The Pole Balancing Problem

[basic description, why is it useful]

The pole balancing problem, or inverted pendulum problem involves balancing a pole on a cart that is mounted on a rail that restricts its movement to one dimension, typically left and right. By moving from side to side, the cart swings the pendulum up into an inverted position, and then balances it by compensating for any movement of the pendulum. This is an interesting problem because it deals with physics and "real time" control. Humans can perform the task which gives us an intuitive sense of what the solution would look like. The task can also be performed by a physical robot and controlled by a classical control system.

The pole balancing task is a good example of a basic reinforcement learning problem. The controlling network received input for each time step of the simulated system, but fitness is only assigned at the end of the task. This can be as many as 100 thousand steps.

The original version of the experiment was with a single pole. Due to advances in machine learning methods and computer hardware, it's no longer considered an appropriate problem. The problem is exacerbated by machine learning methods that often find solutions in the initial population. (Efficient reinforcement learning through symbiotic evolution. Machine Learning, 22:11–32). Wieland (Evolving neural network controllers for unstable systems. In (IJCNN 1991), 667–673) proposed several extensions to the problem:

- Decreasing the amount of state information that the controller receives as input, e.g. the velocity information.
- Adding additional poles, either to the cart, or to the pole in order to create a segmented pendulum.

The first variation makes the problem non-Markovian, meaning the solution requires maintaining a state of some sort. In this case, the position of the cart in the previous time step must be retained to calculate the speed. These variations help to make the problem more challenging to adapt to better learning methods.

5.2 Task Setup

The duel pole balancing task environment was simulated with a realistic physical model. The model uses fourth order Runge-Kutta integration to derive the state of the system; the time step size is 0.01 s. The state variables used are the following:

- x: the cart's position
- $\dot{x}$: the cart's velocity
- θ_i: the i'th pole's angle
- $\dot{\theta}_i$: the i'th pole's angular velocity

See figure x for a diagram of the initial network controlling the cart. Both of the poles are directly attached to the cart, as opposed to a segmented pendulum configuration. The length of the two poles are 0.1 m and 1.0 m, and start in close to an upright position. The track is 4.8 m long. The different pole lengths mean that it's possible to achieve control, because the poles react differently to the applied force on the cart.

The network receives the state variables at each time step, all of which are normalised to $[-1.0, 1.0]$. The network output is scaled to a force of $[-10, 10]$ N, which is then applied to the cart. Next the system transitions to it's next state, which then produces the new set of input state variables.

5.3 Pole Balancing Experiments

[an image of the starting network, inputs and outputs]

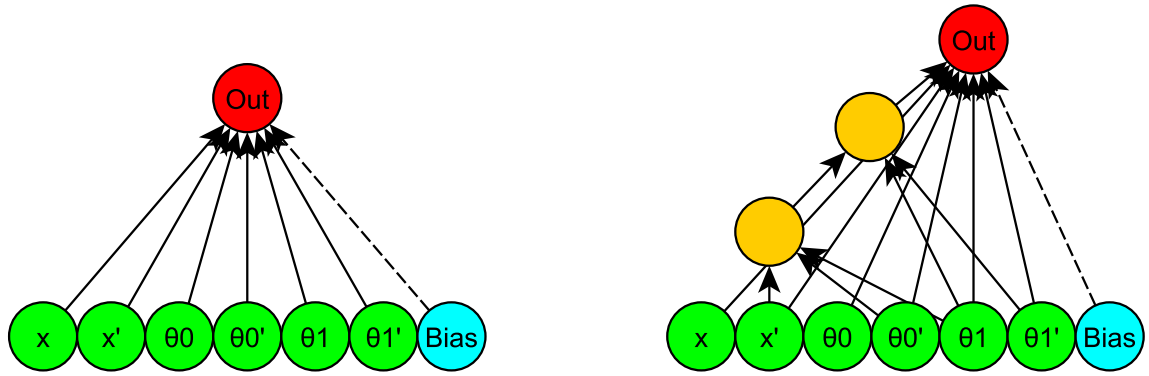
The pole balancing task is split up into two sections: with and without velocity information. This means in the second case, the agent does not receive the cart's velocity or the angular velocity of the poles ($\dot{x}$,$\dot{\theta}_i$), therefore the system is now a non-markov process, and it has to use a form of memory to calculate the cart or pole velocity.

We calculate fitness as the number of time steps the network could keep both poles balanced: they must remain in the range $[-36°, 36°]$ from the vertical position. Additionally, the cart may not move off the track. A controller completes the task if it can stay within the described boundaries for 100 000 time steps, equivalent to 33.3 minutes of simulated time.

5.3.1 Dual pole balancing with velocity info

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.1.

The experiment ran 1000 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 1.08 seconds, and the total running time was 72 minutes.



(a) The initial network

(b) Network of a typical solution

Fig. 5.1: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows a typical solution.

method	mean	min	max	std. dev.
Complexification	25.03	2.00	86.00	14.46
Random	21.11	2.00	183.00	15.77
Static Phased Pruning	24.76	2.00	149.00	16.89
Dynamic Phased Pruning	26.32	2.00	483.00	23.91

Tab. 7: Number of generations required to solve the problem

The Random strategy did surprisingly well in this experiment. It's likely due to some runs reaching a solutions in the second generation, because the problem is fairly simple. This prompted us to repeat the experiment for different probabilities of random pruning: see Table 4. Interestingly, the number of generations slowly drops to from 26.7 to 20.8. The reason is that because the starting network has all the links it needs, as pruning becomes more aggressive, the lower the average complexity and the optimally small solution is more likely to show up, which then means a smaller number of weights to tune in order to reach the solution.

In some aspects this is a case of the simplification-only method discussed in Chapter 3.

5.3.2 Dual pole balancing without velocity info

Solving the problem without velocity information requires the network to retain some state using a recurrent connection. This takes some time to develop, and training methods tend to find

method	mean	min	max	std. dev.
Complexification	10.81	6.00	30.00	4.27
Random	8.15	6.00	20.00	1.95
Static Phased Pruning	9.47	6.00	26.00	3.13
Dynamic Phased Pruning	10.93	6.00	150.00	6.79

Tab. 8: Number of links in the network that solved the problem

method	mean	min	max	std. dev.
Complexification	9.48	7.00	17.00	1.66
Random	8.51	7.00	13.00	0.89
Static Phased Pruning	9.07	7.00	16.00	1.37
Dynamic Phased Pruning	9.53	7.00	55.00	2.42

Tab. 9: Number of nodes in the network that solved the problem

Probability	mean generations
0	26.731
10	24.101
20	24.185
30	22.936
40	21.891
50	22.351
60	21.714
70	20.995
80	21.487
90	20.885

Tab. 10: Number of generations required to solve the problem for Random pruning with different probabilities to prune

the simplest solution that solves the problem. Specifically, this refers to a solution where the network quickly jerks the cart back and forth, causing the poles to rapidly wiggle and not fall over. This solution exploits the finite resolution of the physics engine and solves the problem without calculating the velocity information, thereby defeating the point of the experiment. Gruau et al. (1996) solved this problem by creating a modified fitness function that penalises this behaviour:

$$F = 0.1f_1 + 0.9f_2 \tag{5.1}$$

$$f_1 = t/1000 \tag{5.2}$$

$$f_2 = \begin{cases} 0 & \text{if t} < 100 \\ \frac{0.75}{\sum_{i=t-100}^{t}(|x^i|+|\dot{x}^i|+\left|\theta_1^i\right|+\left|\dot{\theta}_1^i\right|)} & \text{otherwise} \end{cases} \tag{5.3}$$

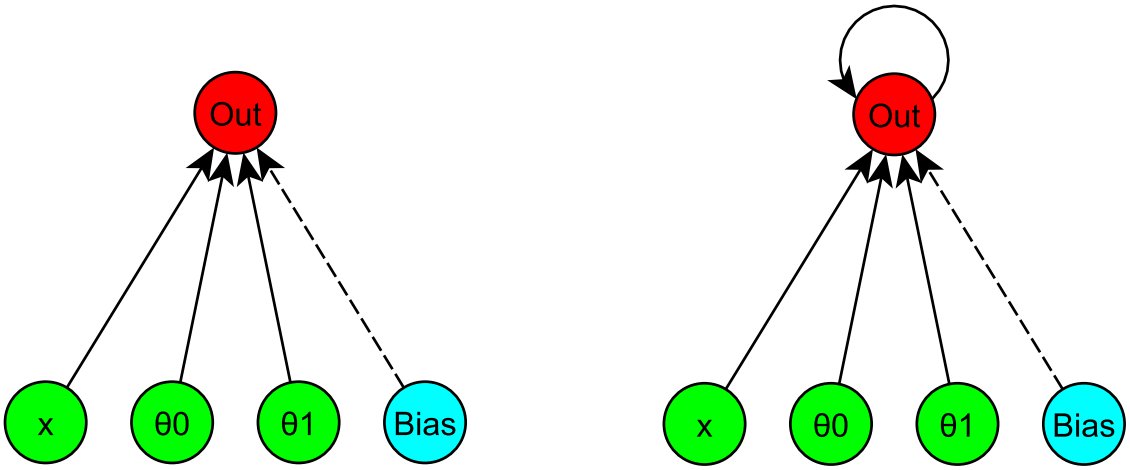
with F being the new fitness. The functions are defined over an interval of 1000 steps. The denominator in f_2 computes the magnitude of the offsets between the cart and the long pole, and their respective resting positions. Added to that is the magnitude of the cart and long pole's speed. This means fitness is maximised by minimising movement, not just how long poles remain balanced.

As with the version of this test with velocity information, the controller completes the task if it can stay within the described boundaries for 100 000 time steps. But because this versions is significantly more complex, an overfitted solution become much more likely. To combat this we add a generalisation test: to be completed by the most fit organism, provided that it's passed the first test.

The generalisation test involves balancing both poles for only 1000 time steps, but from 625 different starting positions. The organism must pass 200 of the 625 to pass the generalisation test. These starting positions are generated by scaling each state variable $(x, \dot{x}, \theta, \dot{\theta})$ to each of the following percentages of the variable's range: 5%, 25%, 50%, 75%, 95%.

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.1.

The experiment ran 1000 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 2.08 seconds, and the total running time was 135 minutes.



(a) The initial network

(b) Network of a typical solution

Fig. 5.2: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows a typical solution.

method	mean	min	max	std. dev.
Complexification	68.34	6.00	437.00	52.06
Random	80.83	11.00	583.00	65.32
Static Phased Pruning	77.57	9.00	949.00	64.95
Dynamic Phased Pruning	68.34	6.00	437.00	52.06

Tab. 11: Number of generations required to solve the problem

The Random and Static Phased strategies did a bit worse than the other two, they did however have simpler solutions on average: the Random strategy did however have simpler solutions: 0.55 fewer links than Complexification case on average.

5.4 Results

[graph of results:
average max fitness for each
average max normalised (by complexity) fitness for each
best network for each]

method	mean	min	max	std. dev.
Complexification	6.07	5.00	20.00	2.47
Random	5.52	5.00	15.00	1.44
Static Phased Pruning	5.71	5.00	18.00	1.81
Dynamic Phased Pruning	6.07	5.00	20.00	2.47

Tab. 12: Number of links in the network that solved the problem

method	mean	min	max	std. dev.
Complexification	5.20	5.00	8.00	0.46
Random	5.15	5.00	7.00	0.39
Static Phased Pruning	5.16	5.00	7.00	0.39
Dynamic Phased Pruning	5.20	5.00	8.00	0.46

Tab. 13: Number of nodes in the network that solved the problem

5.4.1 Dual pole balancing with velocity info

5.4.2 Dual pole balancing without velocity info

5.5 Summary

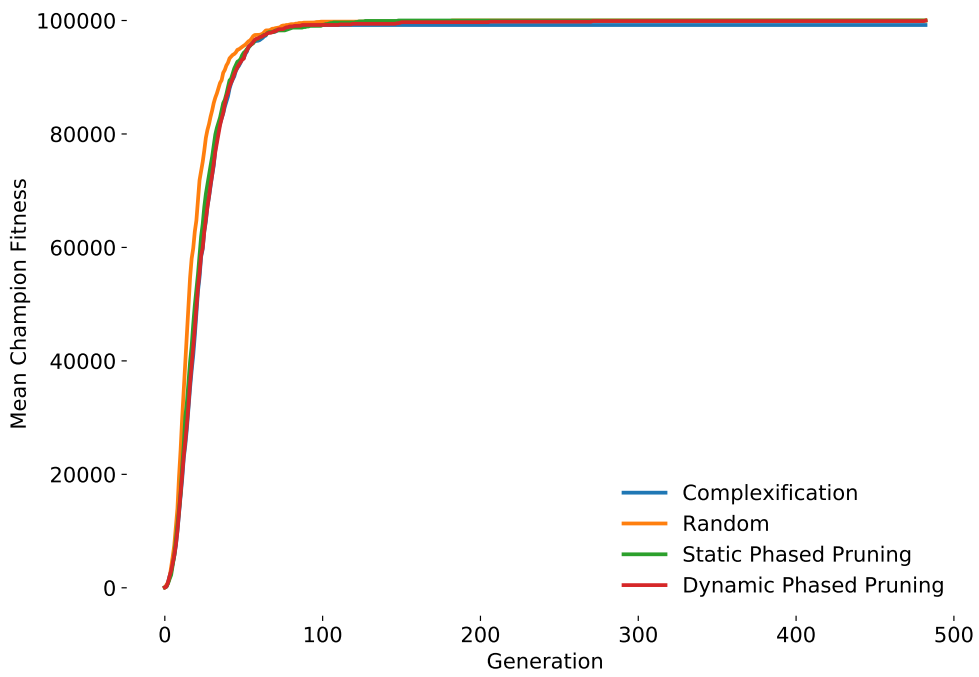


Fig. 5.3: Mean champion fitness over time for each pruning strategy

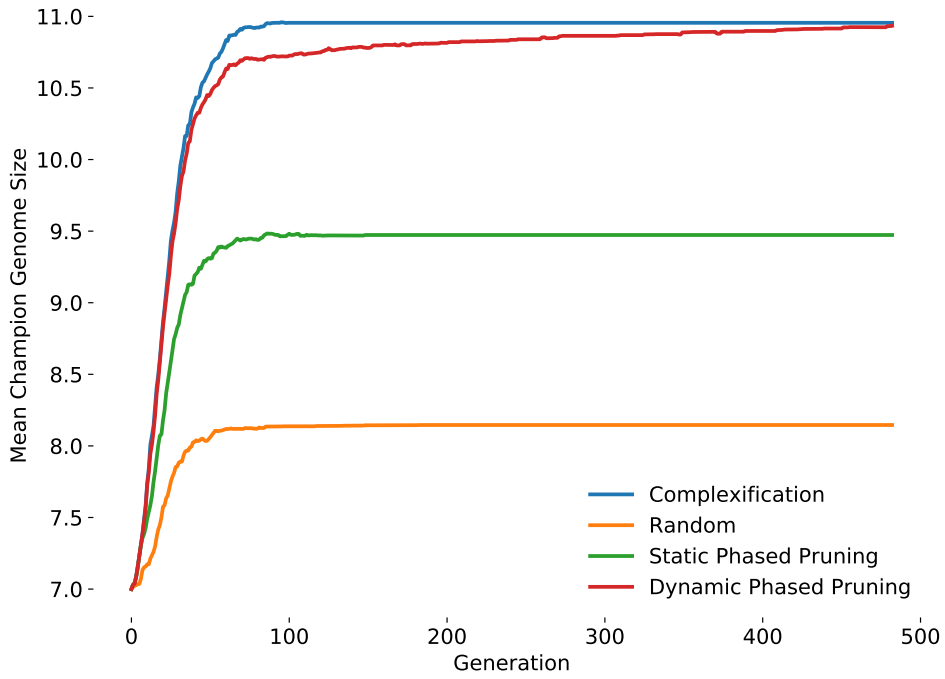


Fig. 5.4: Mean champion genome size over time for each pruning strategy

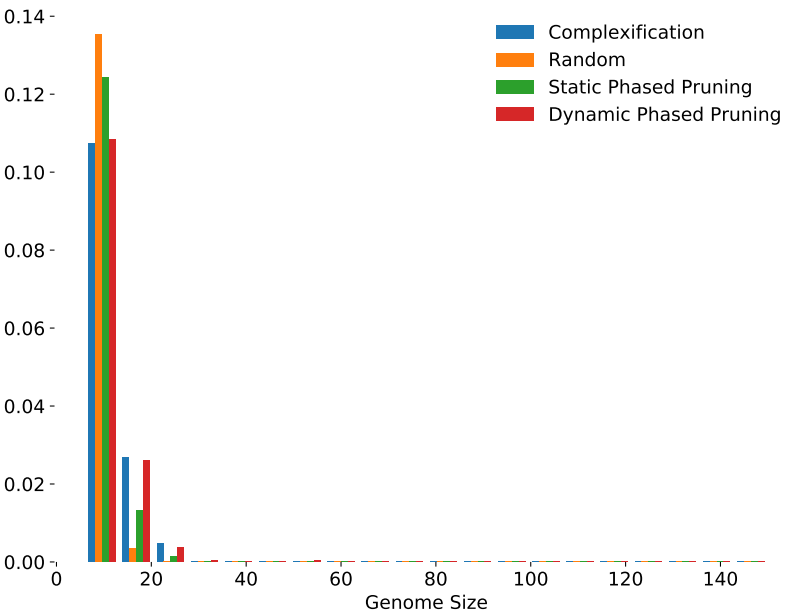


Fig. 5.5: Champion complexity in terms of genome size for each pruning strategy

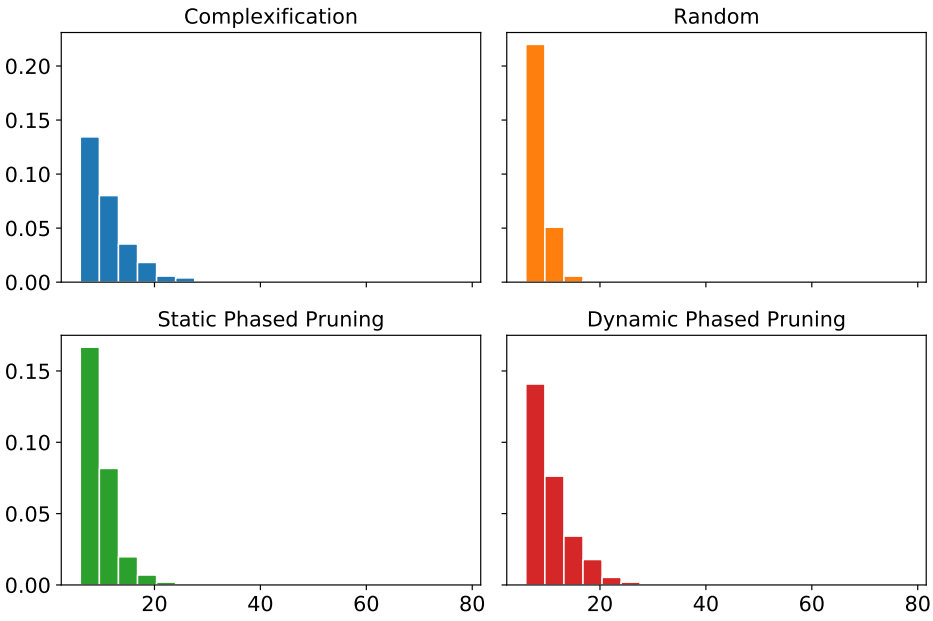


Fig. 5.6: Champion complexity in terms of genome size for each pruning strategy

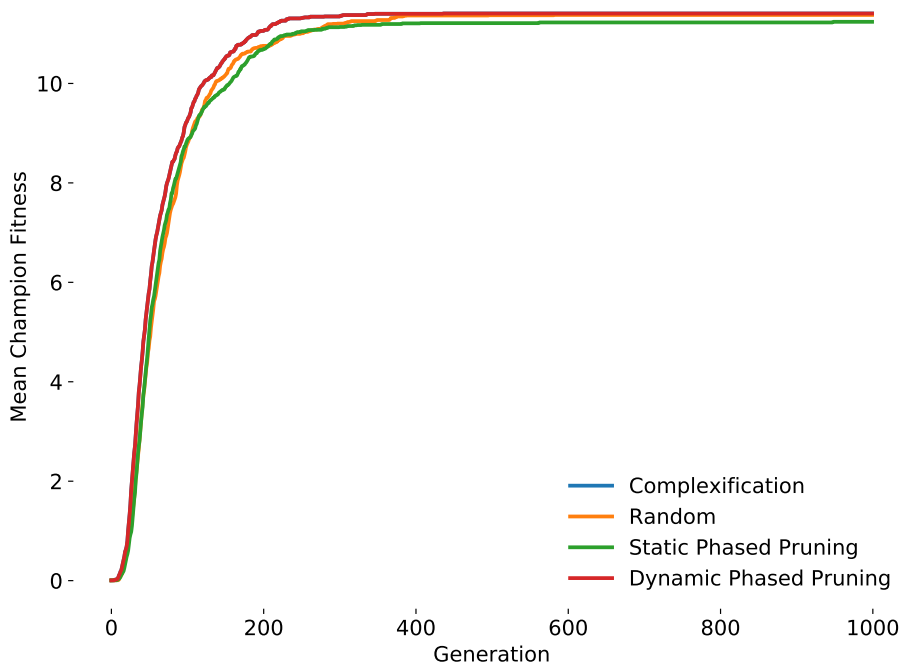


Fig. 5.7: Mean champion fitness over time for each pruning strategy

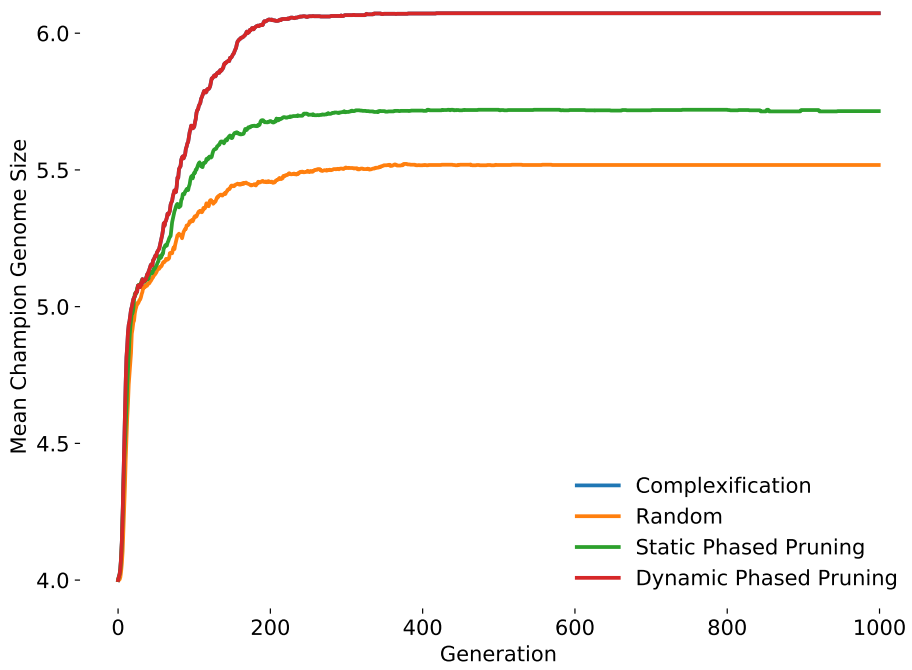


Fig. 5.8: Mean champion genome size over time for each pruning strategy

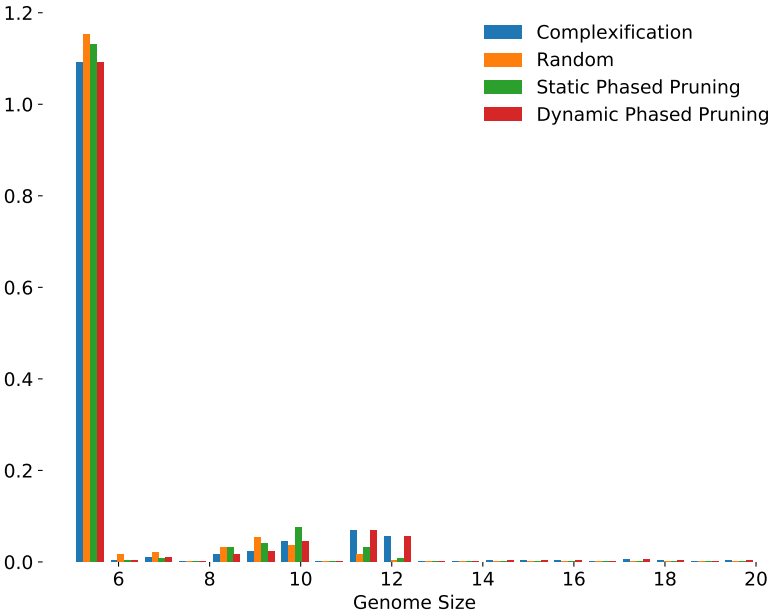


Fig. 5.9: Champion complexity in terms of genome size for each pruning strategy

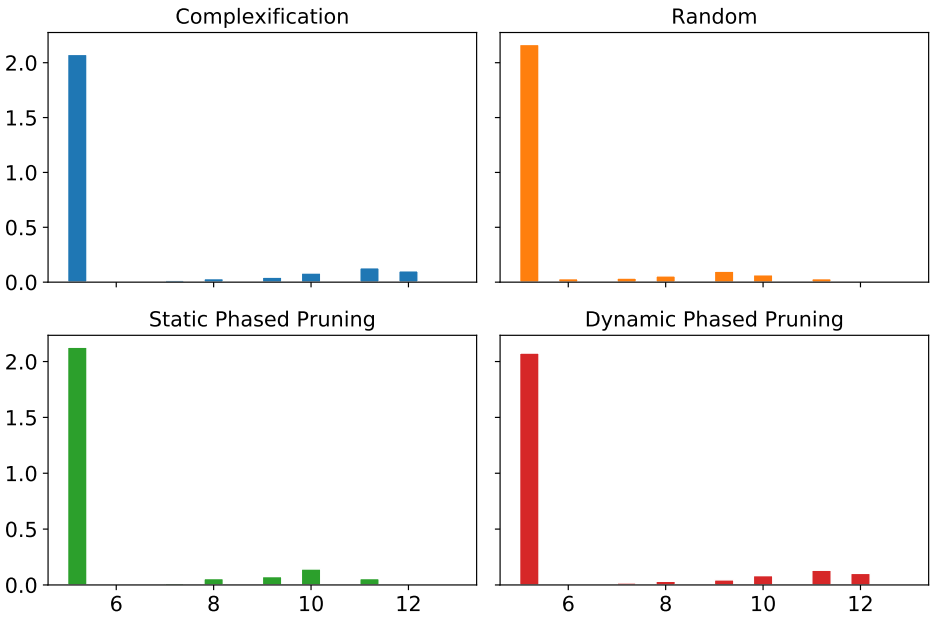


Fig. 5.10: Champion complexity in terms of genome size for each pruning strategy

Game state	Value
Marked by the current player	1
Marked by the opponent	-1
Empty	0

Tab. 14: Mapping of board state to network input

6 Tic-tac-toe

6.1 The Tic-tac-toe Problem

Tic-tac-toe, or noughts and crosses, is a simple pen and paper game where two players (denoted by X and O) take turns to put their mark on a three by three grid. The _cross_ or X player plays the first move. The first player to get three in a row (horizontal, vertical, or diagonal), wins the game. The oldest three-in-a-row game originates from Egypt, dating from 1300 BCE. The first printed reference to modern tic-tac-toe was in a British quarterly scholarly journal, Notes and Queries, (Notes and Queries, 2nd Series Volume VI 152, Nov. 27 1858.) in 1858.

Tic-tac-toe is a fairly simple game, compared to other deterministic zero-sum games, like chess and go. A casual player soon discovers that the best strategy leads to a draw. This has been formally proved because the game has been solved. With 9! (362880) total game states, and 5478 (Symbolic exploration in two-player games: Preliminary results) of them reachable with legal play, a modern computer can explore the entire game tree in less than a millisecond.

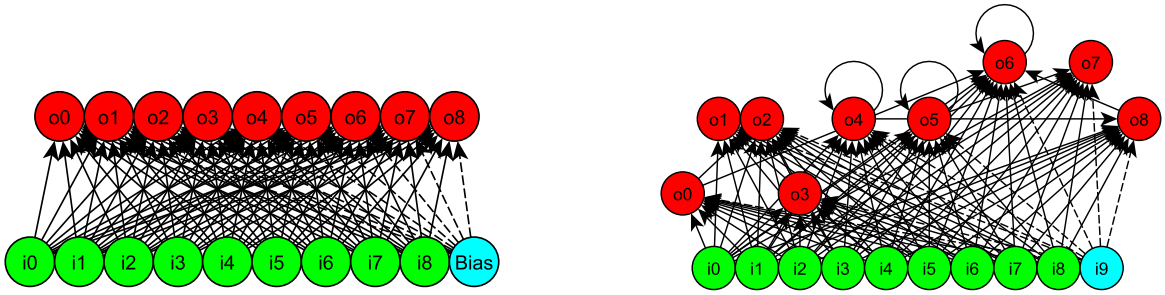
This means we can write a computer controlled opponent that can perform optimal play, and just as importantly we can change it's difficulty. This makes it easier to train a NEAT based agent, because we can accurately evaluate a network's fitness whether it's weak or strong.

[history of the game in RL]

6.2 Task Setup

The network has ten inputs, one for each board position plus a bias node, and then nine outputs for the nine board positions. There are three board states: having a cross, having a zero, and empty, so these board states' mapping to scalars is part of the network input. This mapping is shown in Table 1. We determine the next move by the highest valued output that corresponds to an empty cell.

We created five opponent players: LoserPlayer, RandomPlayerWeak, RandomPlayerStrong, OptimalPlayerWeak, and OptimalPlayerStrong. LoserPlayer tries to lose on purpose by trying to not complete three in a row of its own mark. The RandomPlayerWeak will play random moves, while RandomPlayerStrong will try to play in the centre, otherwise random. Optim-



(a) The initial network

(b) Network of a typical solution

Fig. 6.1: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows a typical solution.

Strategy	Number of games			Average points per game
	200	20000	2000000	
OptimalPlayerStrong (blockFork = true)	733	72895	7288096	3.64
OptimalPlayerWeak (blockFork = false)	727	72790	7277934	3.64
RandomPlayerStrong (playCentre = true)	353	34814	3485707	1.74
RandomPlayerWeak (playCentre = false)	311	30869	3078568	1.54
LoserPlayer	376	36051	3605400	1.80

Tab. 15: Fitness of each opponent player when evaluated as if it were a NEAT agent

alPlayerWeak and OptimalPlayerStrong are both optimal players, except that OptimalPlayerWeak does not block forks by its opponent. Forking is when you create an opportunity where you have two ways to win, meaning two non-blocked lines with two marks. Therefore OptimalPlayerWeak will not attempt to restrict this behaviour.

In order to evaluate the NEAT agent's fitness, we play 20 games per side against each of the five opponents for a total of 200 games. The score per game is 5 for a win, and 2 points for a draw. The fitness is the sum of the scores for all 200 games. All opponents will eventually encounter a situation where they have more than one possible move valued as equally good. In that case they use a pseudorandom number based on how many games have passed in the current fitness evaluation. This increases the diversity of opponent moves, while keeping the fitness evaluation deterministic.

In order to determine appropriate fitness thresholds to determine the target fitness levels, we measure the fitness of each opponent as described above. These results are shown in Table

2. As such we chose 650 as the required fitness to pass the first test threshold. If any organism in a generation passes the first threshold, we run the generalisation test against the one with the highest fitness. The generalisation test is the same as the first test, except it plays 100 times more games and the fitness threshold is 68000. In terms of average points per game these scores are 3.25 and 3.4, respectively. The generalised score is quite close to the highest possible score. Even though this is a simple game, guaranteed perfect play is fairly difficult to beat and complex to express programatically. The C++ code to get the next move in OptimalPlayerStrong is 230 lines long.

6.3 Tic-tac-toe Experiments

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.06.

The experiment ran 1000 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 9.45 seconds, and the total running time was 630 minutes.

6.3.1 Results

[graph of results:
average max fitness for each
average max normalised (by complexity) fitness for each
best network for each]

Complexification, Random and Static Phased Pruning required a similar amount of generations to solve the problem on average. In terms of complexity, the Random strategy has the lowest, and Complexification the highest, but the difference is very small. It might be that the number of generations is too low to draw sufficient benefit from pruning. The number of links in the Random strategy's case was 3.7% less than in the Complexification case. The Dynamic Phased Pruning strategy did significantly worse than the other strategies.

6.3.2 Summary

The tic tac toe problem is a nice example of neuroevolution training to play a simple adversarial game. The complexity is significantly higher than the pole balancing problem.

method	mean	min	max	std. dev.
Complexification	75.95	16.00	464.00	69.10
Random	77.51	4.00	431.00	91.55
Static Phased Pruning	76.88	5.00	488.00	76.76
Dynamic Phased Pruning	84.15	11.00	496.00	86.74

Tab. 16: Number of generations required to solve the problem

method	mean	min	max	std. dev.
Complexification	98.21	89.00	159.00	10.15
Random	94.56	89.00	116.00	5.75
Static Phased Pruning	96.41	88.00	142.00	8.32
Dynamic Phased Pruning	97.01	89.00	157.00	9.69

Tab. 17: Number of links in the network that solved the problem

method	mean	min	max	std. dev.
Complexification	19.60	19.00	25.00	1.03
Random	19.67	19.00	24.00	1.03
Static Phased Pruning	19.59	19.00	25.00	1.02
Dynamic Phased Pruning	19.57	19.00	25.00	0.94

Tab. 18: Number of nodes in the network that solved the problem

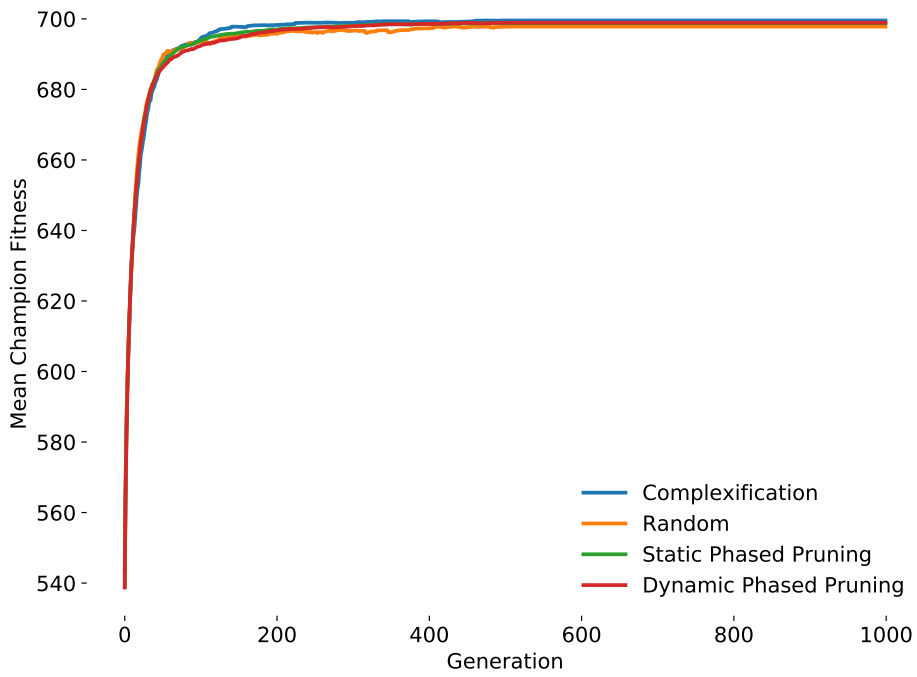


Fig. 6.2: Mean champion fitness over time for each pruning strategy

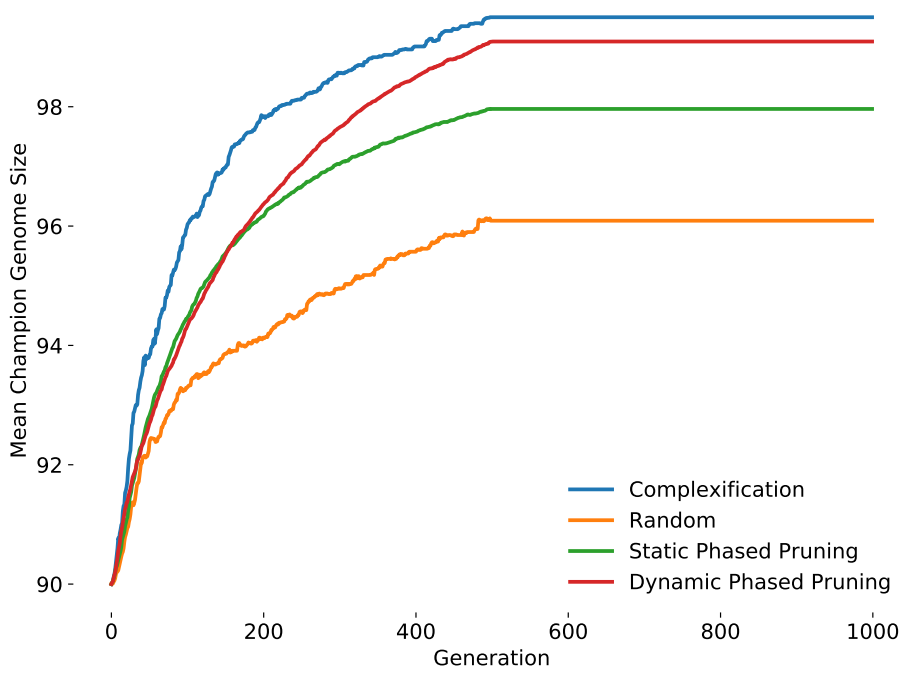


Fig. 6.3: Mean champion genome size over time for each pruning strategy

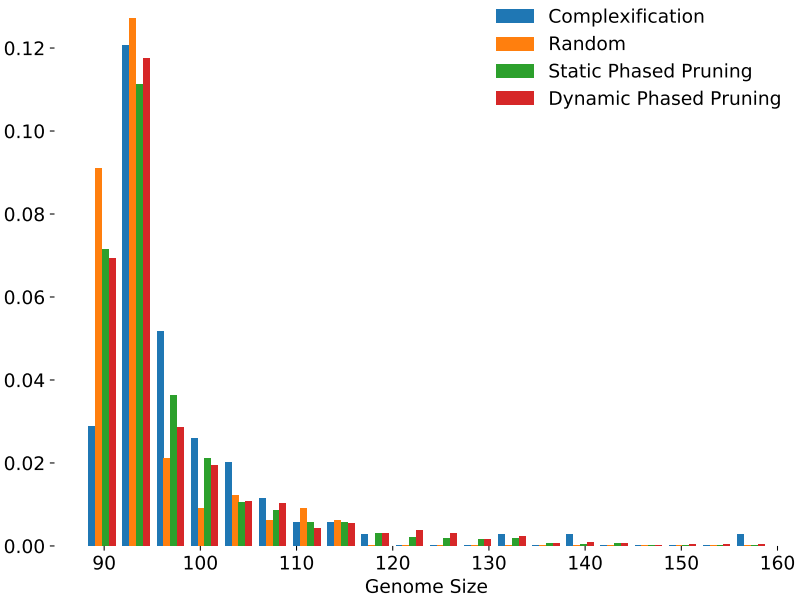


Fig. 6.4: Champion complexity in terms of genome size for each pruning strategy

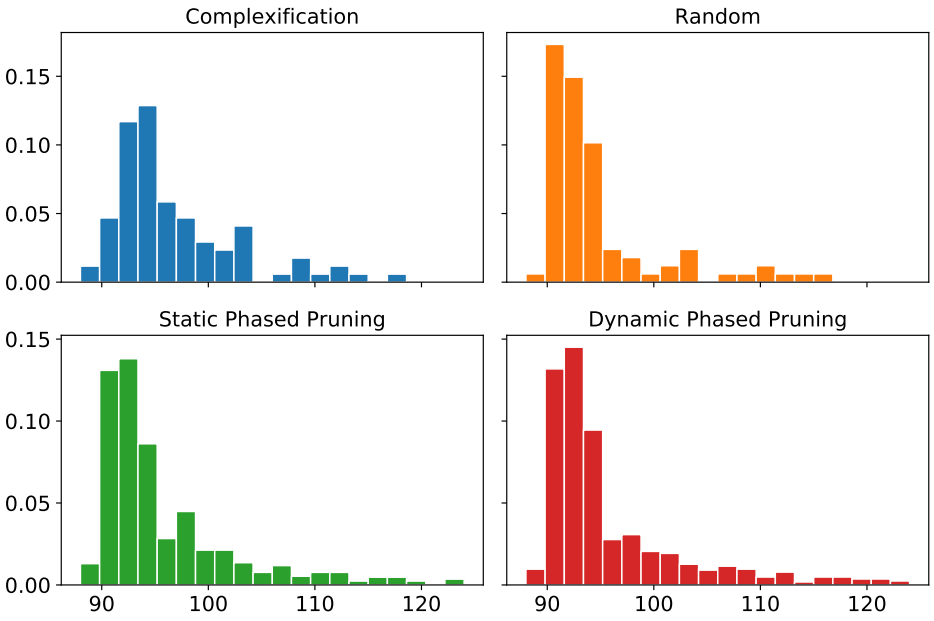


Fig. 6.5: Champion complexity in terms of genome size for each pruning strategy

7 Time Series Prediction

7.1 The Time Series Prediction Problem

[basic description, why is it useful, history]

The time series prediction means taking some measurement over time and predicting the future value. Examples include predicting the price of a company's share price, or an irregular heartbeat. If we have some understanding of the laws that govern the system, prediction becomes very easy, like predicting the motion of a thrown projectile in a vacuum. But if we lack that understanding, prediction is much harder. We need a generalised model with memory capability. Recurrent neural networks fit this requirement very well, and the neuroevolution method of building networks is a suitable approach. (Time series prediction: forecasting the future and understanding the past, AS Weigend - 2018)

As an experiment it's a interesting way to look at how easily and efficiently our methods develop the required recurrent connections.

7.2 Task Setup

[inputs, output, state] The static problem state is fairly simple: the series, x, is a single array of values, so at any given point in time t we are only concerned with two state variables:

- x_t: the value of the series at the current time
- x_{t+1}: the value of the series at $t + 1$

As such, the starting network is fairly simple, with only one state input node, a bias node, and the output node.

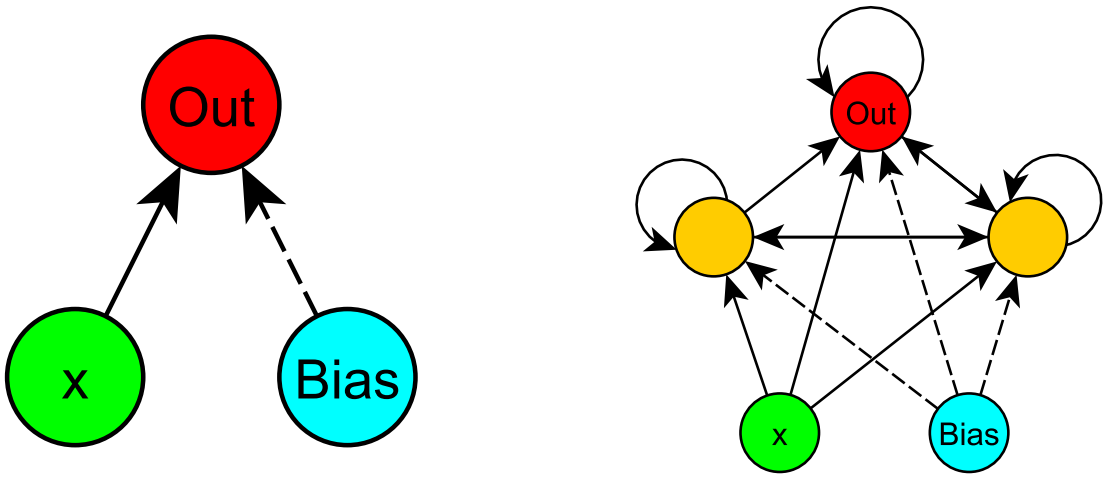
7.3 Time Series Prediction Experiments

The task is to accurately predict the value of the series (s) at the next timestep. The series' length is 100, and it consists of a repeated cycle of 8 values: 0, 0, 0, 0, 1, 1, 1, 1. See Figure 1 for a graph of the series.

We pass the series into the network, one value at a time, and measure the network's output (y) at each step. To determine the fitness we first get the mean absolute error:

$$\bar{e} = \frac{\sum_{i=L_c}^{L_s-1} |s_{i+1} - y_i|}{L_s - L_c - 1} \tag{7.1}$$

with the series length (L_s) equal to 100, the cycle length (L_c) equal to 8, s_{i+1} the future value of the series at i, and y the i-'th output. The sum starts at L_c because the output during the first



(a) The initial network

(b) Network of a typical solution

Fig. 7.1: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows a typical solution.

cycle is not included; this is consider a warm-up period. It ends at $L_s - 1$ because otherwise the future series value would move out of bounds. Then we use it to calculate the fitness:

$$f = \frac{1}{\bar{e} + 10^{-9}} \tag{7.2}$$

This fitness measurement is designed to minimise the total error. Note that the network's output is not an integer, but any real number in the range $[0, 1]$.

We consider the problem solved with any fitness greater than 1000. This corresponds to a mean absolute error of 0.001.

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.1.

The experiment ran 1000 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 1.56 seconds, and the total running time was 104 minutes.

method	mean	min	max	std. dev.
Complexification	282.18	68.00	1293.00	143.03
Random	526.99	55.00	1925.00	328.51
Static Phased Pruning	516.67	59.00	1784.00	275.87
Dynamic Phased Pruning	284.39	68.00	1293.00	147.65

Tab. 19: Number of generations required to solve the problem

method	mean	min	max	std. dev.
Complexification	12.54	7.00	33.00	3.82
Random	9.88	7.00	17.00	1.86
Static Phased Pruning	11.04	7.00	21.00	2.41
Dynamic Phased Pruning	12.60	7.00	33.00	3.86

Tab. 20: Number of links in the network that solved the problem

7.3.1 Results

[graph of results:
average max fitness for each
average max normalised (by complexity) fitness for each
best network for each]

The Random strategy did quite poorly in this case. Due to the strict fitness measurement, a lot of generations were spent on weight tuning, and pruning didn't offer any benefit in terms of how long the experiment took. The Random strategy did however have simpler solutions: 2,66 fewer links than Complexification case on average (21.2% reduction).

7.3.2 Summary

The time series test significantly increased the difficulty as shown by the increase in the number of generations needed to solve the problem, compared to previous problems.

There is a lot of room for expansion on this problem. We used a very simple series, and

method	mean	min	max	std. dev.
Complexification	5.44	5.00	9.00	0.64
Random	5.45	5.00	8.00	0.62
Static Phased Pruning	5.45	5.00	10.00	0.63
Dynamic Phased Pruning	5.44	5.00	9.00	0.64

Tab. 21: Number of nodes in the network that solved the problem

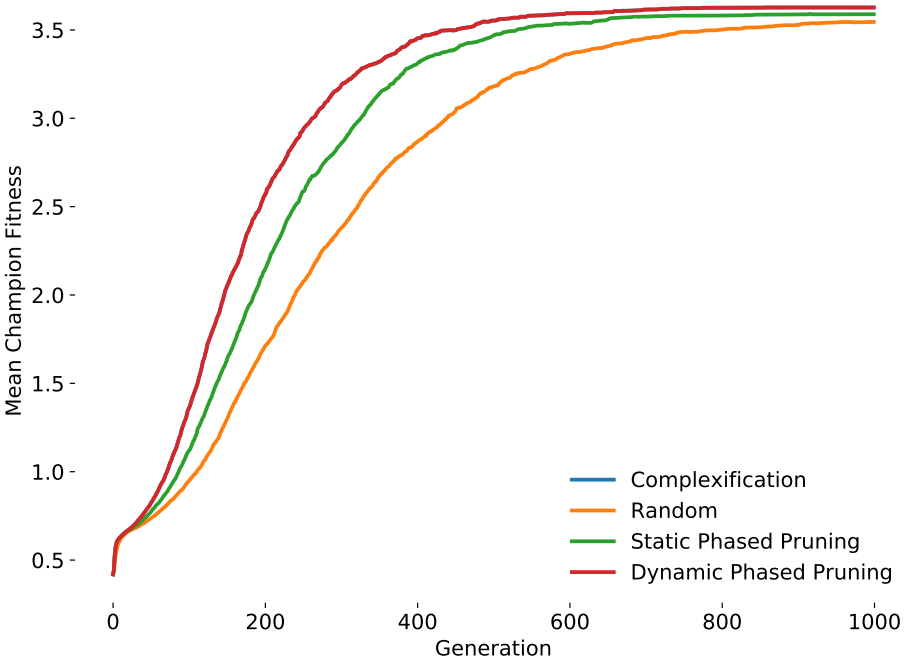


Fig. 7.2: Mean champion fitness over time for each pruning strategy

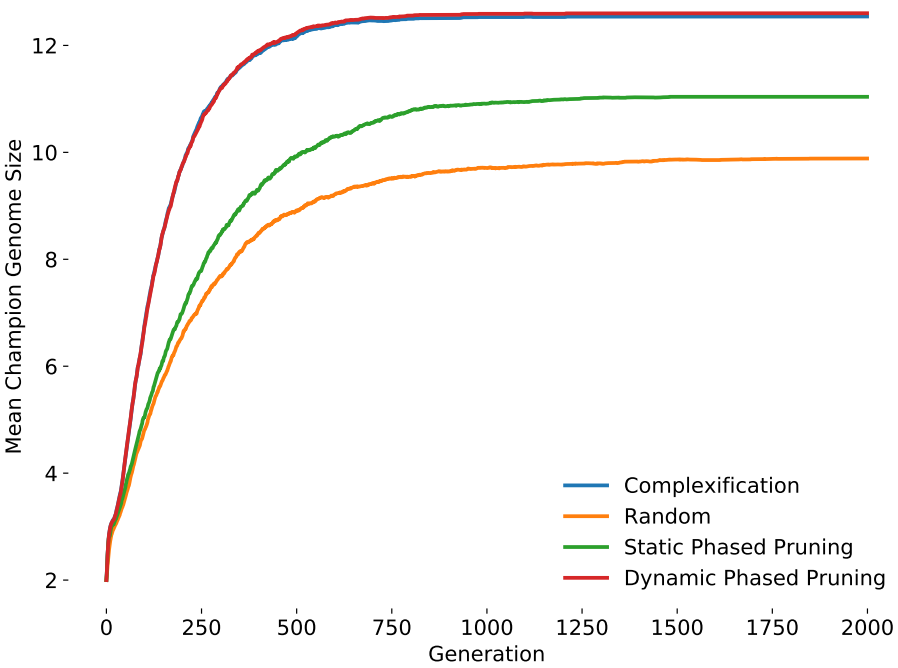


Fig. 7.3: Mean champion genome size over time for each pruning strategy

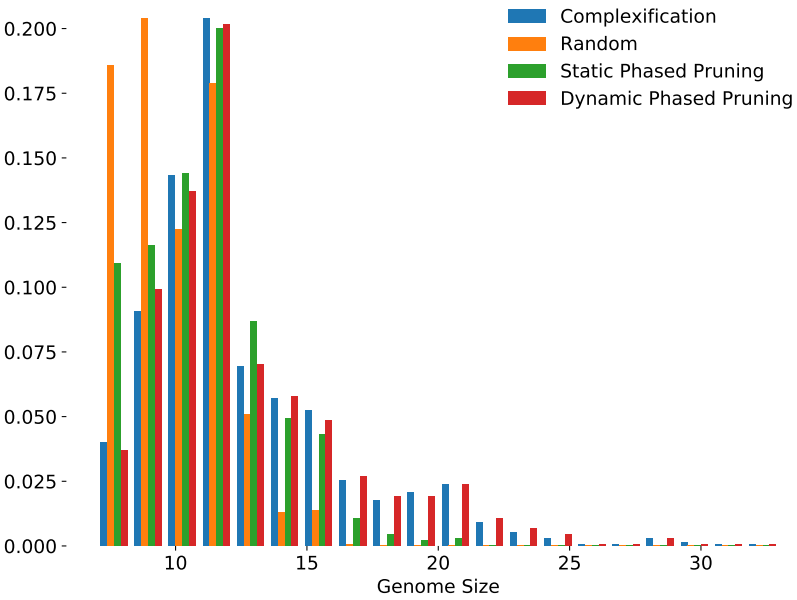


Fig. 7.4: Champion complexity in terms of genome size for each pruning strategy

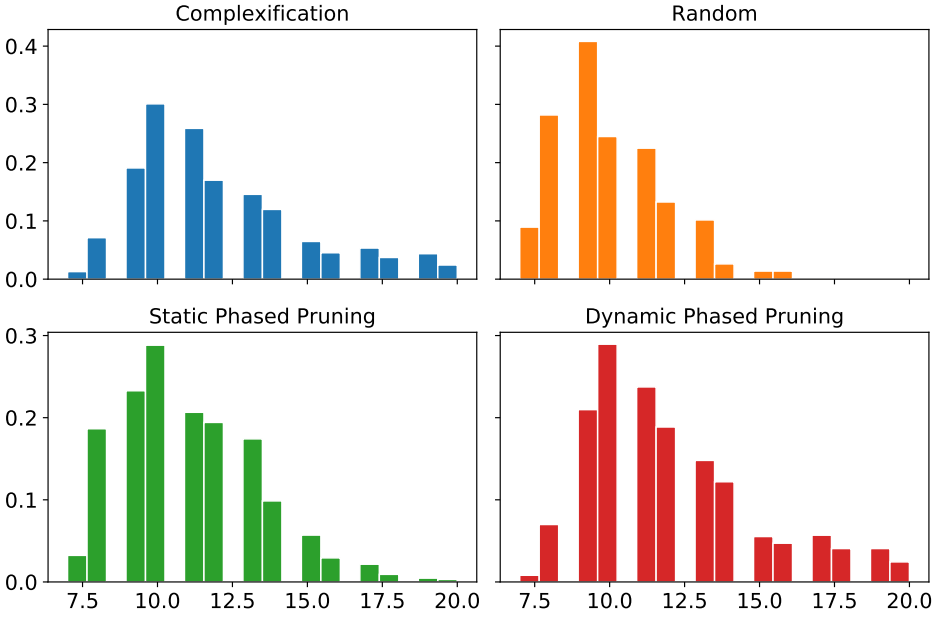
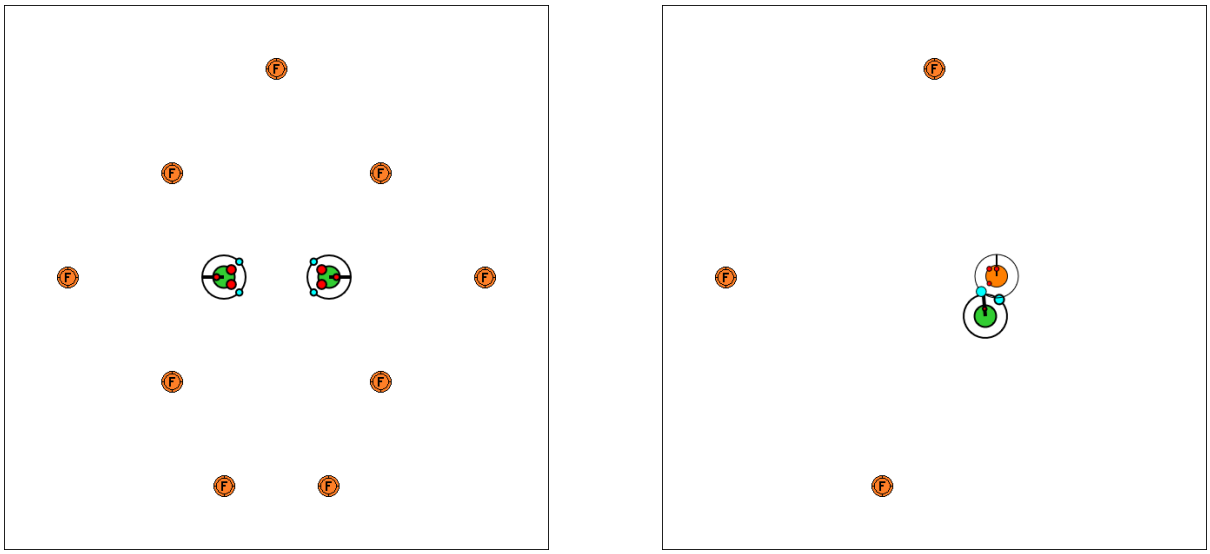


Fig. 7.5: Champion complexity in terms of genome size for each pruning strategy

although neuroevolution is not the best way to evolve recurrent networks, we can expand the model to include other data streams as inputs, or more complex patterns. The challenge is around designing a fitness that rewards accuracy but not to the point of making the task take too long to finish.



(a) The starting position (b) The moment before the end of a round

Fig. 8.1: Figure (a) shows the starting positions. Figure (b) shows the moment before the end of a round.

8 Robot Duel

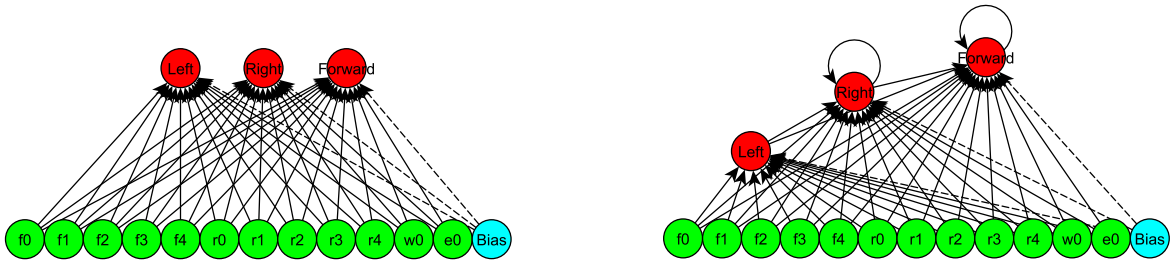
8.1 The Robot Duel Problem

[basic description, why is it useful, history] The robot duel environment is significantly more complex than the tasks in the previous chapters. The neural network must learn to control a robot in an environment where it competes against another robot for resources in the form of food. An interesting feature of this environment is that there is no know optimal strategy, and fitness is can only be measured with respect to opponents.

Previous neuroevolution research has used similar environments, pursuit and evasion tasks are one example [ref Gomez, F., and Miikkulainen, R. (1997). Incremental evolution of complex general behaviour; Miller, G., and Cliff, D. (1994). Co-evolution of pursuit and evasion i: Biological and game- theoretic foundations.], usually evolving separate controllers for the predator and prey roles. Stanley incorporated the two roles into one, to elicit more complex strategies.

8.2 Task Setup

The robot duel environment places two opposing robots in a room, facing away from each other. The robots have two wheels which they use to move, and an array of sensors. The sensors detect food, their opponent's position, and the distance to the closest wall. Each robot



(a) The initial network

(b) Network of a typical solution

Fig. 8.2: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows a typical solution.

starts off with a certain amount of energy, which is used to move around. The higher the input to the virtual wheel motors are, the faster the energy is drained. The robot's energy level will last until the end of the experiment, but they should prioritise gathering food as efficiently as possible. Because the ultimate goal of the robot duel game is to collide with the opposing robot while your energy level is higher than the opponent's energy level. Right from the outset, the controller must deal with competing sub-goals: it should try to gather food, but that costs energy. If it has accumulated more energy that the opponent, it faces a similar dilemma: should it chase the opponent or try to hide? It can even pretend to hide as a way to trap the opponent.

The structure of the robots are based on Kheperas (Mondada, F., Franzi, E., and Ienne, P. (1993). Mobile robot miniaturisation: A tool for investigation in control algorithms). The robot has 12 sensors: five food sensors and five robot sensors are arranged around it in the same pattern: one facing directly ahead at 0°, two at 35° and -35°, and two more at 165° and -165°. Then there is a wall sensor that activates with an intensity proportional to the distance to the closest wall. The last sensor is an energy difference sensor, equal to the robot's energy minus the opponent's energy. The food and robot sensor's widths are 40°, meaning the robots have 90° wide blinds spot on either side. The sensor's output strength is proportional to the square root of the distance to the food or robot.

Each robot has two wheels which control speed and direction. There are three inputs to the system: left, right, and forward. The turning speed is proportional to the difference between the left and right inputs, scaled from 0 to 0.24 rads/turn (13.75°/turn). After rotating, the robot moves proportional to the forward input, scaled from 0 to 13.3 units/turn.

Each robot starts with 100 energy, and consuming food grants 500 energy. The food pellet is

Fig. 8.3: The robot duel starting position: The two robots start facing away from each other. The black bar shows which direction they're facing. The size of the red and cyan circles represent the amplitude of directional sensors. The red circles represent the enemy sensors and the cyan circles represent food sensors. The highlighted outer black circle indicates which robot has higher energy.

Strategy	Fitness	Average points per game
Smart	44722	4.47
Basic	25900	2.59
Dead	16358	1.64
Random	11660	1.17
LazyKill	19380	1.94

Tab. 22: Fitness of each opponent player when evaluated as if it were a NEAT agent

then removed from the game. The room size is 600 by 600 units, and each round lasts up to 500 turns. The food pellets and the robot's radii are both 12.5. Contact with food or with another robot is defined as being apart 20 units or less (their centres will be 45 units apart). Energy loss due to movement is equal to the distance moved plus the angle rotated in radians.

The initial network as 13 inputs (12 sensors and one bias) and is fully connected to the output layer of three nodes (left, right, and forward). Note that the sensors do not provide complete information about the environment: the sensors are limited in range and angular width.

8.3 Robot Duel Experiments

In order to measure the fitness we created five opponent players: SmartBot, BasicBot, DeadBot, RandomBot, and LazyKillBot. DeadBot doesn't move at all, which is a very simple strategy, but not terrible because it conserves all its energy. RandomBot moves feed random input to the robot controls, and performs the worst as a result. BasicBot targets food until there is none left, then targets the enemy. LazyKillBot targets the enemy as soon as its energy is 50 more than theirs, otherwise does nothing.

SmartBot is a bit more sophisticated. If at any point it has 50 more energy than the opponent and the opponent is visible, engage it. Else it checks to see if there is anything visible on the sensors (food or the opponent); if there is nothing it turns until something is visible. If it finds some food it will move to the closest food at maximum speed. Otherwise, the food has run out so it tries to engage the opponent anyway.

In order to evaluate the organism's fitness, we play two game per side against each of the five opponents for a total of 20 games. The points per game is 5 for a win, and 2 points for a draw. The fitness is the sum of the scores for all 20 games. In order to increase the behavioural diversity of the bot players, they use a pseudorandom number based on how many games have passed in the current fitness evaluation to make small changes to their behaviour, like small changes to their speed. This ensures that the fitness evaluation remains deterministic.

In order to determine appropriate fitness thresholds to determine the target fitness levels, we measure the fitness of each opponent as described above. These results are shown in Table 1. As such we chose 90 as the required fitness to pass the first test threshold. If any organism in a generation passes the first threshold, we run the generalisation test against the one with the highest fitness. The generalisation test is the same as the first test, except it plays 100 times more games and the fitness threshold is 450. In terms of average points per game these scores are 3.25 and 3.4, respectively. [TODO: Fix fitness scores]

8.3.1 Other Methods

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.08. We set the link removal probability close to the link add probability (0.1) in this case due to the very large number of starting links.

The experiment ran 1000 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 3 minutes, and the total running time was 8.3 days.

8.3.2 Results

[graph of results:
average max fitness for each
average max normalised (by complexity) fitness for each
best network for each]

8.3.3 Summary

The robot duel problem increased the difficulty significantly and encouraged interesting and adaptive solutions.

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 23: Number of generations required to solve the problem

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 24: Number of links in the network that solved the problem

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 25: Number of nodes in the network that solved the problem

9 Mario Agent

9.1 The Mario Agent Problem

Super Mario Bros. is a platform video game released in 1985 for the Famicom and Nintendo Entertainment System (NES) consoles. Players control Mario, whom they have to guide through a series of side-scrolling stages. Players have to avoid obstacles like enemies and pits in the process. The game is considered one of the greatest video games of all time, and is still being played. For example, at the time of writing there were two active live streams of it on Twitch, a popular live streaming video platform.

Humans can quickly start playing and enjoying the game because it's very easy at the start. Concepts like running, jumping, and avoiding enemies are obvious and natural to a human player. The game steadily increases its difficulty, which makes it fun to play while remaining challenging. This is a very useful feature for training artificial intelligence agents, because sharp increases in difficulty can cause training to slow down or stop.

The Mario problem involves training a neural network to control Mario. It's an interesting problem due to the big input space: we feed the the game's screen output buffer into the neural network, and the output is fed to the game controller. The neural network has to learn how to avoid enemies and obstacles, while not slowing down too much because the stages have time limits. The gradual increase in difficulty is also appealing. There are many different types of obstacles: there are fourteen different enemies, each with different behaviours and weaknesses.

The Super Mario Bros game has some prior use in neuroevolution research. It has been used to test an indirect encoding scheme called HyperGP (ref: Super Mario evolution, 2009). They used a slightly different implementation called Infinite Mario Bros, which has some features that make it useful for research.

9.2 Task Setup

We used an open-source C++ NES emulator to create the game state (github.com/neveraway/nes-emulator). The main reason we chose this emulator is our project is also written in C++. This allowed us to modify the emulator to run as a headless library (DLL) that outputs the new game state at each frame. At the start of each game, it loads the Mario ROM and a saved game state that has already started a new game. Emulators use optimisations like frame skipping to increase performance, but we could not use this feature because we had to ensure deterministic fitness evaluations.

There are several ways to feed the game state into the neural network:

- Provide the screen buffer as input

- Provide the entire game state (a dump of the RAM) as input

The screen size is 256 by 224, or 57344 pixels. Prior research have used the screen buffer as input, but that requires a convolutional layers, which is out of scope. An alternative is using the game RAM as input. The NES as 2KB of onboard RAM, and Mario uses a lot of it. But we only need a small subset of the game state

In order to extract information from the emulated RAM, we used the game's RAM memory map (ref: http://datacrystal.romhacking.net/wiki/Super_Mario_Bros.:RAM_map) to extract the needed state information. We extract the following information:

- Positions of all enemies on the screen, limited to five at a time
- The type of tiles on the screen (wall, floor, sky, etc.)
- Whether Mario is alive
- Mario's position

Although the screen size is 256 by 224 pixels, most of the static game elements are composed of 16 by 16 pixel tiles aligned to a 16 by 14 grid. For example in Figure 1, the floor blocks and the question brick blocks are tiles on this grid. Notice that the floor is composed of 16 floor blocks.

Using the memory map, we construct a simplified 13 by 13 display, a rectangle centred on Mario with a border of six tiles in each direction. Each of the 169 cells is set to -1 if the tile contains something dangerous (an enemy or a projectile thrown by an enemy), 1 if the tile is a solid object (wall, floor, pipe, etc.), and otherwise 0 indicates an open space. Note that an open space doesn't mean it's safe because it could be a open pit.

The NES controller has eight buttons: up, down, left, right, A, B, start, select. Since the start and select buttons are not used during gameplay, we ignore them.

As such the starting network has 169 normal inputs for each cell in the display, and one bias node, totalling 170. The output layer has six nodes, one for each of the active buttons. The input layer is fully connected to the output layer, resulting in 1020 links in the starting network.

At each new frame, we update the network with the current state as produced by the display. The network processes the input and sets the output nodes. For each output that's greater than 0.5, we set the corresponding button as pressed (key down event). If an output drops below 0.5 in the next frame, we release the key (key up event), and if it's still below 0.5 in the frame after that one we set the key as unpressed. After we've set the input keys to the correct states we generate the next frame. Next we check if the game is still active, and if so, continue.

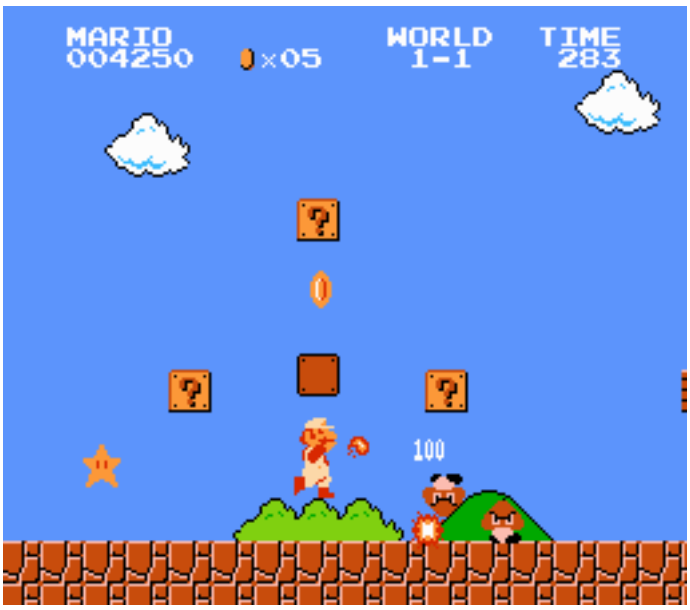
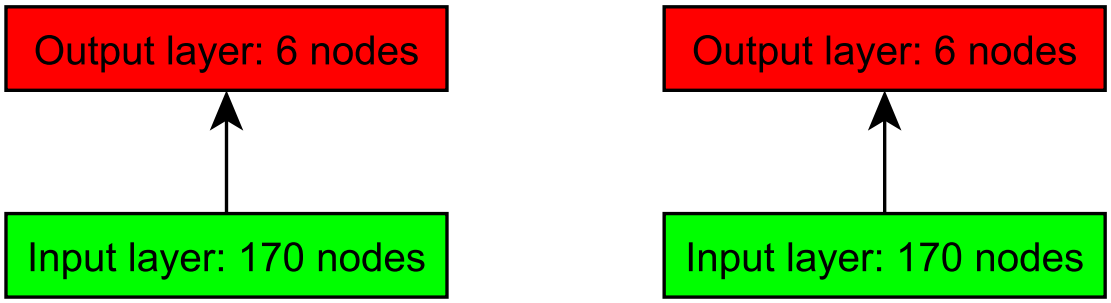


Fig. 9.1: Gameplay example: here Mario is hitting a special brick from below to collect a coin, while throwing fireballs at two enemies called Goombas. The left Goomba has just been hit by a fireball. (ref: https://en.wikipedia.org/wiki/File:NES_Super_Mario_Bros.png)



(a) The initial network

(b) Network of a typical solution

Fig. 9.2: Figure (a) shows the initial network given to the population. It follows the NEAT convention of a fully connected single layer network: the smallest possible fully connected network consisting of all the model inputs, the model outputs, and a bias input. Figure (b) shows a typical solution.

9.3 Mario Agent Experiments

There are several conditions that will cause the stage to end:

- contact with an enemy
- falling down a pit
- running out of time
- completing the stage by touching the flag pole at the end

Mario has no health bar, so the first contact with an enemy will kill him. If he consumes a Super Mushroom, then he will grow to twice his starting size, and enemy contacts shrink him back to the original size. He will always die when falling down a pit, regardless of his state. The absolute time limit per stage is 400 in-game seconds (as per the timer at the top left of the screen), equal to 160 seconds in real time at normal game speed.

Instead of the standard three lives, this game ends as soon as Mario loses a life. This makes the fitness calculation easier.

The game scrolls to the right, mean that after Mario spawns on a new level, he has to move right until he reaches the end of the stage. Therefore we define fitness as the distance, in pixels, that Mario is from the starting point when the round ends. Although jumping over obstacles is very important, it's a tactical manoeuvre, so there is no reward for the vertical position at the end of the round. The fitness per stage is calculated as follows:

$$f = \frac{x_{mario}}{x_{max} + 0.001} \tag{9.1}$$

with x_{mario} the absolute distance from the left-most point on the stage to Mario's position when the stage ends for whatever reason, and x_{max} the maximum distance on the stage. Due to the large time cost of emulating the game, we played only one stage. The maximum distance on stage 1, level 1, is 3266. Therefore, the total fitness calculation simplifies to:

$$f = \frac{x_{mario}}{3266.001} \tag{9.2}$$

In addition, we impose a movement requirement: if Mario's furthest recorded position hasn't changed in 600 ticks, or 30 in-game seconds, we end the run on the assumption that the agent is stuck. Sometimes the organism hasn't quite mastered jumping over large obstacles, and will just keep jumping a low wall. This ensures that we don't waste time waiting in such cases.

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 26: Number of generations required to solve the problem

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 27: Number of links in the network that solved the problem

9.3.1 Other Methods

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.1. We set the link removal probability to the same value as the link add probability in this case due to the very large number of starting links.

The experiment ran 100 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 48 hours, and the total running time was X hours.

9.3.2 Results

[graph of results:
average max fitness for each
average max normalised (by complexity) fitness for each
best network for each]

9.3.3 Summary

The Mario problem is the final application. The massive network size pushes the bounds of what is possible with traditional neuroevolution.

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 28: Number of nodes in the network that solved the problem

10 OCR

10.1 Optical Character Recognition Problem

Optical Character Recognition (OCR) is the task of translating text characters in an image to a machine-encoded charaters. For example, extracting the text from a printed document that has been scanned and stored as a bitmap. Neural networks perform very well on this task (ref).

OCR is not typically framed as a reinforcement learning problem, because we know what the output corresponding to each input should be. There is no cycle of interactions between the agent and the environment. We chose this problem because our focus is to compare the relative performance of pruning algorithms, not specifically how well-suited NEAT is to evolving a neural network capable of OCR. One advantage of an OCR problem is it's computationally very cheap to test. Another is that the input and output pairs are easy to represent visually.

previous use?

10.2 Task Setup

The task has been simplified in order to shift the focus to the behaviour of the pruning methods. The input to the neural network is a single letter, encoded by a 7x7 bitmap. Therefore we have 50 inputs, considering the one bias node. Our input dataset is the 26 uppercase Latin characters. The output is 26 nodes, each corresponding to one of the letters. We choose the output with the highest value to be the network's answer.

10.3 OCR Experiments

We calculate fitness as the number of characters that are recognized correctly, given the 26 input characters. As a result the possible fitness ranges between 0 and 26.

(add formula)

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 29: Number of generations required to solve the problem

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 30: Number of links in the network that solved the problem

10.3.1 Other Methods

We compared four evolutionary pruning strategies: classic NEAT (complexification), random pruning, static phased pruning, and dynamic phased pruning. All experiments used the same NEAT parameters, see Appendix A for the values used. The parameters were the same as in the original NEAT study [ref], and the link removal probability was 0.1. We set the link removal probability to the same value as the link add probability in this case due to the very large number of starting links.

The experiment ran 100 times for each pruning method. The experiment was run on a 3.40GHz CPU with four cores (Intel Core i5-3570K). Each experiment took 48 hours, and the total running time was X hours.

10.3.2 Results

[graph of results:
average max fitness for each
average max normalised (by complexity) fitness for each
best network for each]

10.3.3 Summary

method	mean	min	max	std. dev.
Complexification	0	0	0	0
Random	0	0	0	0
Static Phased Pruning	0	0	0	0
Dynamic Phased Pruning	0	0	0	0

Tab. 31: Number of nodes in the network that solved the problem

11 Discussion

11.0.1 Revisiting research questions

Question: what is the benefit of sticking complexification and simplification phases together? Is the notion of recovery useful to pruning methods?

Answer:

Question: how much pruning is needed:

Answer:

Question: which parameters of dynamic pruning lead to improved search performance by reducing the number of generations taken to reach a solution

Answer:

11.0.2 Overall Performance of each pruning strategy

11.0.3 Limitations

11.0.4 Summary

12 Conclusion

12.1 Contributions

Pruning methods: No Pruning, Only Pruning, Random Pruning, Static Phased Pruning, Dynamic Phased Pruning, Testing Methodology

References

[1] P. J. Angeline and J. B. Pollack. Competitive Environments Evolve Better Solutions for Complex Tasks. In *ICGA*, 1993.

[2] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, January 1994. ISSN 1045-9227. doi: 10.1109/72.265960.

[3] Richard Bellman. *Dynamic Programming*. Dover Publications, Mineola, N.Y, reprint edition edition, March 2003. ISBN 978-0-486-42809-3.

[4] Anselm Blumer, A. Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the Vapnik-Chervonenkis Dimension. *J. ACM*, 36(4):929–965, October 1989. ISSN 0004-5411. doi: 10.1145/76359.76371. URL http://doi.acm.org/10.1145/76359.76371.

[5] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, December 1989. ISSN 0932-4194, 1435-568X. doi: 10.1007/BF02551274. URL https://link.springer.com/article/10.1007/BF02551274.

[6] D. Dasgupta and D. R. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96, June 1992. doi: 10.1109/COGANN.1992.273946.

[7] Eberhart and Yuhui Shi. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 1, pages 81–86 vol. 1, 2001. doi: 10.1109/CEC.2001.934374.

[8] Andrzej Ehrenfeucht, David Haussler, Michael Kearns, and Leslie Valiant. A general lower bound on the number of examples needed for learning. *Information and Computation*, 82 (3):247–261, September 1989. ISSN 0890-5401. doi: 10.1016/0890-5401(89)90002-3. URL http://www.sciencedirect.com/science/article/pii/0890540189900023.

[9] Alex Fain. Adaptation, specificity and host-parasite coevolution in mites (ACARI). *International Journal for Parasitology*, 24(8):1273–1283, December 1994. ISSN 0020-7519. doi: 10.1016/0020-7519(94)90194-5. URL http://www.sciencedirect.com/science/article/pii/0020751994901945.

[10] Dario Floreano, Peter DÃŒrr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, March

2008. ISSN 1864-5909, 1864-5917. doi: 10.1007/s12065-007-0002-4. URL https://link.springer.com/article/10.1007/s12065-007-0002-4.

[11] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 978-0-201-15767-3.

[12] Colin Green. Phased Searching with NEAT, August 2004. URL http://sharpneat.sourceforge.net/phasedsearch.html.

[13] Joshua B. Gross, Richard Borowsky, and Clifford J. Tabin. A Novel Role for Mc1r in the Parallel Evolution of Depigmentation in Independent Populations of the Cavefish Astyanax mexicanus. *PLOS Genetics*, 5(1):e1000326, January 2009. ISSN 1553-7404. doi: 10.1371/journal.pgen.1000326. URL http://journals.plos.org/plosgenetics/article?id=10.1371/journal.pgen.1000326.

[14] Nikolaus Hansen and Andreas Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*, 9(2):159–195, June 2001. ISSN 1063-6560. doi: 10.1162/106365601750190398. URL https://doi.org/10.1162/106365601750190398.

[15] Will Hardwick-Smith, Yiming Peng, Gang Chen, Yi Mei, and Mengjie Zhang. Evolving Transferable Artificial Neural Networks for Gameplay Tasks via NEAT with Phased Searching. In *AI 2017: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 39–51. Springer, Cham, August 2017. ISBN 978-3-319-63003-8 978-3-319-63004-5. URL http://homepages.ecs.vuw.ac.nz/ yimei/papers/AI2017-Yiming.pdf.

[16] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1994. ISBN 978-0-02-352761-6.

[17] John Henry Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992. ISBN 978-0-262-58111-0. Google-Books-ID: 5EgGaBkwvWcC.

[18] Derek James and Philip Tucker. A Comparative Analysis of Simplification and Complexification in the Evolution of Neural Network Topologies. Seattle, Washington, USA, June 2004. URL http://www.cs.bham.ac.uk/ wbl/biblio/gecco2004/LBP019.pdf.

[19] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. Technical Report RAND-RM-704, RAND PROJECT AIR FORCE SANTA MONICA

CA, RAND PROJECT AIR FORCE SANTA MONICA CA, December 1951. URL http://www.dtic.mil/docs/citations/ADA596138.

[20] Ron Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. 1995.

[21] Rajendra Krishnan and Victor B. Ciesielski. Delta-Gann: A New Approach To Training Neural Networks Using Genetic Algorithms. In *University of Queensland*, pages 194–197, 1994.

[22] Riccardo Poli, William B Langdon, and Nicholas Freitag McPhee. A Field Guide to Genetic Programming, 2008. URL http://cswww.essex.ac.uk/staff/poli/gp-field-guide/113Bloat.html.

[23] Figueira Pujol and Riccardo Poli. Evolution of the Topology and the Weights of Neural Networks using Genetic Programming with a Dual Representation. March 1997.

[24] Nicholas J. Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90, March 1993. ISSN 0941-0643, 1433-3058. doi: 10.1007/BF01411376. URL https://link.springer.com/article/10.1007/BF01411376.

[25] R. Reed. Pruning algorithms-a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747, September 1993. ISSN 1045-9227. doi: 10.1109/72.248452.

[26] B. Sareni and L. Krahenbuhl. Fitness sharing and niching methods revisited. *IEEE Transactions on Evolutionary Computation*, 2(3):97–106, September 1998. ISSN 1089-778X. doi: 10.1109/4235.735432.

[27] N. T. Siebel, J. Botel, and G. Sommer. Efficient neural network pruning during neuro-evolution. In *2009 International Joint Conference on Neural Networks*, pages 2920–2927, June 2009. doi: 10.1109/IJCNN.2009.5179035.

[28] Nils T. Siebel and Gerald Sommer. Evolutionary reinforcement learning of artificial neural networks. *International Journal of Hybrid Intelligent Systems*, 4(3): 171–183, January 2007. ISSN 1448-5869. doi: 10.3233/HIS-2007-4304. URL http://content.iospress.com/articles/international-journal-of-hybrid-intelligent-systems/

[29] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, January

1991. ISSN 0893-9659. doi: 10.1016/0893-9659(91)90080-F. URL http://www.sciencedirect.com/science/article/pii/089396599190080F.

[30] LÃ©o FranÃ§oso dal Piccol Sotto and VinÃcius Veloso de Melo. Studying bloat control and maintenance of effective code in linear genetic programming for symbolic regression. *Neurocomputing*, 180(Supplement C):79–93, March 2016. ISSN 0925-2312. doi: 10.1016/j.neucom.2015.10.109. URL http://www.sciencedirect.com/science/article/pii/S0925231215015866.

[31] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, June 2002. ISSN 1063-6560. doi: 10.1162/106365602320169811. URL https://doi.org/10.1162/106365602320169811.

[32] Kenneth Owen Stanley. *Efficient evolution of neural networks through complexification*. PhD thesis, The University of Texas at Austin, August 2004. URL https://repositories.lib.utexas.edu/bitstream/handle/2152/1266/stanleyk74304.pdf.

[33] Maxine Tan, Rudi Deklerck, Jan Cornelis, and Bart Jansen. Phased searching with NEAT in a Time-Scaled Framework: Experiments on a computer-aided detection system for lung nodules. *Artificial Intelligence in Medicine*, 59(3):157–167, November 2013. ISSN 0933-3657. doi: 10.1016/j.artmed.2013.07.002. URL http://www.sciencedirect.com/science/article/pii/S0933365713000985.

[34] D. Thierens. Non-redundant genetic coding of neural networks. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 571–575, May 1996. doi: 10.1109/ICEC.1996.542662.

[35] A. P. Wieland. Evolving neural network controllers for unstable systems. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume ii, pages 667–673 vol.2, July 1991. doi: 10.1109/IJCNN.1991.155416.

[36] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8(3):694–713, May 1997. ISSN 1045-9227. doi: 10.1109/72.572107.

[37] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, September 1999. ISSN 0018-9219. doi: 10.1109/5.784219.

[38] Byoung-Tak Zhang and Miihlenbeinl Miihlenbeinl. Evolving Optimal Neural Networks U sing Genetic Algorithms with Occam's Razor*. *Complex Systems*, 7(3):199–220, 1993.

13 Appendix

13.1 Appendix A

NEAT parameters for all experiments.

13.2 Appendix B: Pole Balancing

The equations of motion for N unjointed poles balanced on a single cart are:

$$\ddot{x} = \frac{F - \mu_c sgn(\dot{x}) + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i}$$

13.3 Appendix C: Statistical significance of results

13.3.1 XOR

13.3.2 Duel pole balancing with velocity info

13.3.3 Duel pole balancing without velocity info

13.3.4 Tic-tac-toe

13.3.5 Time series

13.3.6 Robot duel

13.3.7 Mario

Variable name	XOR	DPM	DPNM	Tic-tac-toe	Time series	Robot duel	Mario
trait_param_mut_prob	0.5	0.5	0.5	0.5	0.5	0.5	0.5
trait_mutation_power	1	1	1	1	1	1	1
linktrait_mut_sig	1	1	1	1	1	1	1
nodetrait_mut_sig	0.5	0.5	0.5	0.5	0.5	0.5	0.5
weight_mut_power	2.5	2.5	1.8	2.5	1.8	2.5	1.5
recur_prob	0	0	0	0	0	0	0
disjoint_coeff	1	1	1	1	1	1	1
excess_coeff	1	1	1	1	1	1	1
mutdiff_coeff	0.4	0.4	3	0.4	3	2	2
compat_thresh	3	3	4	3	4	3	10
age_significance	1	1	1	1	1	1	1
survival_thresh	0.2	0.2	0.4	0.2	0.4	0.2	0.2
mutate_only_prob	0.25	0.25	0.25	0.25	0.25	0.25	0.25
mutate_random_trait_prob	0.1	0.1	0.1	0.1	0.1	0.1	0.1
mutate_link_trait_prob	0.1	0.1	0.1	0.1	0.1	0.1	0.1
mutate_node_trait_prob	0.1	0.1	0.1	0.1	0.1	0.1	0.1
mutate_link_weights_prob	0.9	0.9	0.9	0.9	0.9	0.9	0.9
mutate_toggle_enable_prob	0	0	0	0	0	0	0
mutate_gene_reenable_prob	0	0	0	0	0	0	0
mutate_add_node_prob	0.02	0.03	0.001	0.001	0.001	0.025	0.005
mutate_add_link_prob	0.05	0.05	0.03	0.03	0.03	0.1	0.1
mutate_remove_link_prob	0.1	0.1	0.1	0.06	0.1	0.08	0.1
interspecies_mate_rate	0.001	0.001	0.001	0.001	0.001	0.05	0.05
mate_multipoint_prob	0.6	0.6	0.6	0.6	0.6	0.6	0.6
mate_multipoint_avg_prob	0.4	0.4	0.4	0.4	0.4	0.4	0.4
mate_singlepoint_prob	0	0	0	0	0	0	0
mate_only_prob	0.2	0.2	0.2	0.2	0.2	0.2	0.2
recur_only_prob	0	0.2	0.2	0.2	0.2	0.2	0.2
pop_size	150	150	150	150	100	100	100
dropoff_age	15	15	15	15	50	20	200
newlink_tries	20	20	20	20	20	20	20
babies_stolen	0	0	0	500	0	0	0
num_runs	1000	1000	10	0	30	1	2
num_generations	1000	1000	1000	99	2000	1000	700
Target number of species	N/A	N/A	N/A	500	N/A	10	10

Tab. 32: NEAT parameters for all experiments

Variable name	Description
trait_param_mut_prob	Probability of mutation on a single trait parameter
trait_mutation_power	Power of mutation on a single trait parameter (amplification factor).
linktrait_mut_sig	Amount that mutation_num changes for a trait change inside a link
nodetrait_mut_sig	Amount a mutation_num changes on a link connecting a node that changed its trait
weight_mut_power	The power of a linkweight mutation
recur_prob	Prob. that a link mutation which doesn't have to be recurrent will be made recurrent
disjoint_coeff	Genome compatibility: Weight of % disjoint genes. Also known as C2
excess_coeff	Genome compatibility: Weight of % excess genes. Also known as C1
mutdiff_coeff	Genome compatibility: Average absolute difference of matching links' mutation values. Also known
compat_thresh	Genome compatibility: Threshold at which genomes join the same species
age_significance	Young species (i.e. age <= 10) has their fitness multiplied with this number
survival_thresh	The percentage of the population that will survive each generation, ranked by fitness
mutate_only_prob	Probability of asexual reproduction
mutate_random_trait_prob	Trait mutation
mutate_link_trait_prob	Trait mutation
mutate_node_trait_prob	Trait mutation
mutate_link_weights_prob	Mutation: Probability to mutate (randomly perturb) all links in the genome
mutate_toggle_enable_prob	Mutation: Probability to enable or disable (depending on its current status) a random link
mutate_gene_reenable_prob	Mutation: Probability to enable a random disabled link
mutate_add_node_prob	Mutation: Probability to add a new node
mutate_add_link_prob	Mutation: Probability to add a new link
mutate_remove_link_prob	Mutation: Probability to remove a link
interspecies_mate_rate	Mating: Probability to mate outside the species
mate_multipoint_prob	Mating: Probability to mate by multipoint: the child inherits the union of the parents' genes, shared genes are s
mate_multipoint_avg_prob	Mating: Probability to mate by multipoint_avg: the child inherits the union of the parents' genes, shared gene
mate_singlepoint_prob	Mating: Probability to mate by normal crossover: the child inherits the first x% of one parent, the rest fro
mate_only_prob	Probability of mating without mutation
recur_only_prob	Probability of a new link being recurrent*
pop_size	The population size
dropoff_age	Number of generations of stagnation after which species are penalized
newlink_tries	Number of tries that the system will try to add a new link
babies_stolen	Steal babies from the weakest species in addition to normal fitness based discrimination
num_runs	Number of times to repeat the experiment
num_generations	Number of generations until the experiment stops

Tab. 33: NEAT parameter definitions

Symbol	Description	Value
x	Position of cart on track	$[-2.4, 2.4]$m
ϑ	Angle of pole from vertical	$[-36, 36]$deg.
F	Force applied to cart	[-10,10] N
li	Half length of ith pole	l1 = 0.5m l2 = 0.05m
M	Mass of cart	1.0 kg
mi	Mass of ith pole	m1 =0.1kg&m2 =0.01kg
μc	Coefficient of friction of cart on track	0.0005
μp	Coefficient of friction if ith pole's hinge	0.000002

Tab. 34: Parameters used in the duel pole balancing problem

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.0000	0.0000	—
Dynamic Phased Pruning	1.0000	0.0000	0.0000

Tab. 35: xor: p-values that the differences between the mean number of generations of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0001	—	—
Static Phased Pruning	0.0047	0.2636	—
Dynamic Phased Pruning	1.0000	0.0001	0.0047

Tab. 36: xor: p-values that the differences between the mean champion genome size of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.7101	0.0000	—
Dynamic Phased Pruning	0.1454	0.0000	0.0936

Tab. 37: dual_pole_markov: p-values that the differences between the mean number of generations of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.0000	0.0000	—
Dynamic Phased Pruning	0.6190	0.0000	0.0000

Tab. 38: dual_pole_markov: p-values that the differences between the mean champion genome size of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.0005	0.2641	—
Dynamic Phased Pruning	1.0000	0.0000	0.0005

Tab. 39: dual_pole_non_markov: p-values that the differences between the mean number of generations of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.0002	0.0072	—
Dynamic Phased Pruning	1.0000	0.0000	0.0002

Tab. 40: dual_pole_non_markov: p-values that the differences between the mean champion genome size of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.8949	—	—
Static Phased Pruning	0.9083	0.9413	—
Dynamic Phased Pruning	0.3649	0.4832	0.0521

Tab. 41: tic_tac_toe_bot: p-values that the differences between the mean number of generations of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0028	—	—
Static Phased Pruning	0.0461	0.0362	—
Dynamic Phased Pruning	0.2430	0.0168	0.1490

Tab. 42: tic_tac_toe_bot: p-values that the differences between the mean champion genome size of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.0000	0.4471	—
Dynamic Phased Pruning	0.7340	0.0000	0.0000

Tab. 43: time_series: p-values that the differences between the mean number of generations of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random	0.0000	—	—
Static Phased Pruning	0.0000	0.0000	—
Dynamic Phased Pruning	0.7227	0.0000	0.0000

Tab. 44: time_series: p-values that the differences between the mean champion genome size of each of the given methods are statistically significant.

method	Complexification	Random	Static Phased Pruning
Random			
Static Phased Pruning			
Dynamic Phased Pruning			

Tab. 45:

method	Complexification	Random	Static Phased Pruning
Random			
Static Phased Pruning			
Dynamic Phased Pruning			

Tab. 46:

method	Complexification	Random	Static Phased Pruning
Random			
Static Phased Pruning			
Dynamic Phased Pruning			

Tab. 47:

method	Complexification	Random	Static Phased Pruning
Random			
Static Phased Pruning			
Dynamic Phased Pruning			

Tab. 48: