

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Genetic Evolution
of
Neural Networks

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Giacomo Rizzi

Sessione II
Anno Accademico 2017/2018

Contents

Introduction	1
1 Genetic Algorithms	3
1.1 The Canonical Algorithm	4
1.2 Terminology	5
1.3 Encoding	6
1.4 Selection	7
1.4.1 Fitness Proportionate Selection with “Roulette Wheel” and “Stochastic Universal” sampling	8
1.4.2 Sigma Scaling	8
1.4.3 Elitism	9
1.4.4 Rank Selection	9
1.4.5 Tournament Selection	10
1.5 Reproduction and Genetic Operators	10
1.5.1 Sexual and Asexual Reproduction	11
1.5.2 Mutation	11
1.5.3 Crossover	12
2 Evolving Neural Networks	13
2.1 Direct Encoding	14
2.1.1 Evolving a weight vector	15
2.1.2 Evolving neurons	16
2.1.3 Evolving Topologies	17
2.2 Indirect Encoding	22

2.2.1	Grammatical Encoding	22
2.2.2	Compositional Pattern-Producing Networks	23
2.3	Optimizing Backpropagation Problems	24
2.3.1	Evolving the topology of deep neural networks	26
3	Experimentation	27
3.1	Gene Pools	28
3.2	Neural Network Mechanics	29
3.2.1	The Neuron	29
3.2.2	The Synapse	29
3.2.3	The Signal	30
3.2.4	The Network	30
3.3	Network Generation	31
3.4	Genetic Operators	32
3.5	Implementation	33
3.6	Testing and results	34
3.6.1	Evaluation	35
3.6.2	Results	35
4	Conclusions	39
4.1	Future developments	39
	Bibliography	41

Introduction

In the modern days an huge effort is being put into the study of the field of artificial intelligence, with the final long term goal of creating a general intelligence, which is, the intelligence of a machine that could perform any intellectual task that a human being can. The most promising tool to achieve such results are artificial neural networks (*ANNs*), as they try to digitally represent a simplified version of the brain as observed in animals. ANNs do require some form of learning before actually being able to solve any problem, and thus a number of techniques has been gradually found to train them in the most efficient – as in time spent – and effective – as in quality of results – way possible.

Evolutionary computation is the attempt to simulate the natural process of evolution to generate and improve algorithms. The field is particularly interesting because it goes way beyond the domain of computer science, indeed it makes use of knowledge from a wide variety of fields such as genetics, biology, anatomy, zoology, botany and natural history. All of these different domains can be exploited to gain useful insights about how living beings reached the state in which they are today, and consequently how we can reproduce the factors that ultimately lead to proactive, intelligent, forms of life. The challenge is not easy tough.

As deep neural networks achieve unprecedented results, successfully training ANNs with millions of parameters, the evolutionary way of training the networks may seem a poor choice for such a task. Nonetheless the evolutionary algorithms, and in particular the genetic algorithms, are increasingly being coupled with backpropagation trained neural networks, as the two techniques excel on different but complementary tasks.

The aim of this elaborated is to illustrate the concepts at the core of genetic algorithms, explaining what are the essential steps that must be taken in order to apply them efficiently on generic optimization problems. Once the foundations on GAs have been laid down, the second chapter will elaborate on how they can be applied to solve the specific problem of evolving different parts of artificial neural networks, from the bare weights of fixed size NNs to the full architecture of unconstrained ones, finally introducing how GAs are being used to optimize the hyper-parameters and the architecture of backpropagation trained networks.

In the last chapter it is proposed a novel kind of dynamic neural network which is evolved by a simple yet powerful genetic algorithm, which makes use of techniques treated previously. The testing has been conducted on the task of classifying a common dataset, and while results do not yet achieve the state of the art performance, they open many doors for further research.

Chapter 1

Genetic Algorithms

Genetic algorithms (GAs) were first hypothesized by Alan Turing in the 50s in its publication “*The Imitation Game*” but it has been John Holland to set the first theoretical and practical foundations during the 1960s and the 1970s. In contrast with the evolution strategies of the time, Holland’s original goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be imported into computer systems.

Genetic algorithms are very flexible optimization techniques as they require minimal knowledge about the problem under investigation. They try to mimic the way nature evolves living creatures by generating a population of candidate solutions (i.e. *agents, individuals*) and applying natural selection on it to produce a new generation of novel candidate solutions. This strategy is iterated hundreds to thousands of times in order to produce a chain of generations containing increasingly fitter solutions.

The core principles of this approach follow those of biology and the evolving world, as discovered by Darwin and explained in *The Origin of Species*, and are three: heredity, variation and selection, all of which are necessary for an evolutionary process to happen and are thus common to all genetic algorithm implementations.

Heredity: there must be a way for which candidate solutions (the individuals) can replicate (through reproduction) and pass their own traits to the offspring, allowing thus

the inheritance of those qualities that lead the parent to be selected.

Variation: the solutions in the population must present some differences in their traits to allow the discrimination of the worse solutions and the reward of the better ones. Several strategies exist to maintain a population varied, the most commons being applying **mutations** on the traits of new agents, so that a child is never an exact copy of the parent, or generating an offspring from more than one selected solutions, mimicking the biological process of **crossover** which blends the *DNA* coming from father and mother during the formation of a new organism.

Selection: this is the mechanism by which better individuals are given a higher chance than their less fit counterparts to reproduce and generate offspring that will “live” in the following generation. Many different selection techniques have been proposed that try to find a balance between the exploration of new solutions and the exploitation of the known ones.

1.1 The Canonical Algorithm

While many variations of GAs exist, the most common procedure implies the creation of a population of a certain size that will be iteratively evolved thus producing completely new generations at each cycle. It follows a formalization of the procedure:

1. **create the first population**, either by generating agents having random traits, or by using pre-researched solutions;
2. **evaluate** the fitness of each individual in the population, which is represented as a numeric value to be maximized or minimized;
3. apply **selection** to produce a parent pool;
4. use the individuals in the parent pool to create new candidate solutions through **reproduction**;

5. **iterate** over 2 - 3 - 4 until the best solution found satisfies the requirements (e.g. effectively maximizes the evaluation function).

Using this scheme, the worst solutions would become rare and eventually disappear from the population because selection would not allow them to transmit their traits into the following generation; indeed, the better ones would replicate giving life to mutated copies of themselves, taking those free places left by unselected agents. These new copies may behave either better or worse than their parent(s), but again, the selection algorithm would not let the bad ones survive in future generations.

It is important to notice that the algorithm may never get to produce a satisfying solution, for example because it might get stuck in some local optima from which no offspring could escape. While diversity is the first insurance against this scenario, literature proposes several approaches to alleviate it further (e.g. dynamically modifying the evaluation function in a way that both preserves the objective and removes or alleviates such problem [1]).

1.2 Terminology

Before going deeper into the explanations, some definitions are given, as the terminology used for genetic algorithms often refers to domains foreign to computer science, such as biology and genomics.

Candidate solution: also called agent, individual, or phenotype: it is the set of parameters that compose a potential answer to an optimization problem. This is what we ultimately evaluate and assign a score to.

Fitness: the evaluated ability of an agent to adapt to a problem; it serves as a way to differentiate and rank them in order to apply selection. It is usually represented as a real number that must be either maximized or minimized.

Trait: is one part of a candidate solution, that directly influences its fitness. Using an analogy from biology, the long neck found in giraffes is a trait, because it directly influences a giraffe capability to adapt to its environment.

Gene: is the encoded form of one or more traits. The thing about genes is that they are atomically used in genetic operations, so, if one gene represents more than one trait, an operation applied to it will potentially affect all the traits it encodes. A gene does not directly affect the way an individual behaves, but does so only after it has been used to express its traits. Most of the times there is no actual distinction between genes and traits, for example when they represent numerical values.

DNA: also called *genome*, is the set of all the genes needed to create a functional agent, which we can refer to as the encoded form of an agent. The most common form of such is a vector containing all of the genes.

1.3 Encoding

When implementing a GA, one has to decide on a way to encode the parameters to represent a candidate solution. The encoded form must comply with the following constraints:

1. is manipulable by genetic operators, which are the means through which the DNA is changed over the generations;
2. can be translated back into a candidate solution.

The simplest method to build DNA is to translate each of the traits of an individual to just one apposite gene, thus producing a direct encoding in which a one-to-one relation between traits and genes is kept. This technique has the advantage of being straight forward to implement, especially when no particular translation is needed, but tends to become more and more complex to evolve as the size of the DNA increases, as it will

require more trials before a successful mutation is found.

Indirect encoding is one way to overcome such problem. This technique is different in that genes do not directly express traits, but represent a rule that is used to generate one or more of them - and they may even do so with the collaboration of other genes. The GA will thus evolve the set of rules, rather than the traits.

Selection and reproduction are the two fundamental parts in the implementation of a genetic algorithm because not only they define the method to generate the pool of possible parents and the way it should be used to create a new agent, but they also determine the agent's traits, and consequently how they must be encoded in the DNA.

A common problem is that reward and diversity do not couple well together: on one hand, the former will tend to forget all those individuals that perform poorly, which may lead to populations where the agents are all too similar to each other. This results in the exploration of a very small and dense fraction of the search space, which would be fine for a final tuning of the solutions, but would cripple the algorithm in case of a local optima, as no solution would be different enough to go escape the optima. On the other hand, diversity favors the proliferation of a variety of solutions, which are able to explore a less dense, but wider, search space, minimizing the chance of getting trapped in a local optimum. Maximizing variation would however slow down the whole algorithm, even crawling it to the point where it becomes a pure random search. Therefore, a balance between these two properties is required to get the best results from the algorithm.

1.4 Selection

Selection is the problem of picking the appropriate individuals in a population that will be given the chance to be the parents for the offspring of the next generation, and it also defines how many offspring each parent will create.

No rigorous guidelines exist for choosing the most appropriate technique; this is still an open question for GAs, as all of them are a compromise between several aspects.

1.4.1 Fitness Proportionate Selection with “Roulette Wheel” and “Stochastic Universal” sampling

These are the oldest and easiest approaches in which agents are given a chance at reproduction proportional to their fitness. In the *roulette wheel* (proposed by Holland) each individual is assigned a slice of a circular “roulette wheel”, the size of the slice being proportional to the individual’s fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel’s marker is selected to be in the pool of parents for the next generation. However, with the relatively small populations, the actual number of offspring allocated to an individual is often far from its expected value (an unlikely series of spins may always select the worst individual).

To overcome this problem, James Baker (1987) proposed a different sampling method - “stochastic universal sampling” (SUS) - to minimize this “spread” (the range of possible actual values, given an expected value). Rather than spin the roulette wheel N times to select N parents, SUS spins the wheel once but with N equally spaced pointers, which are used to select the N parents. The method has problems when all individuals in the population are very similar (the fitness variance is low): in such case there are no real fitness differences for selection to exploit, and evolution grinds to a near halt. This is known as “premature convergence”. Thus, the rate of evolution depends on the variance of fitness in the population.

1.4.2 Sigma Scaling

This approach tries to mitigate the problem raised by low variance populations by not using the direct fitness as expected value, but rather a function of the fitness, the population mean, and the population standard deviation. An example of sigma scaling would be:

$$ExpVal(i, t) = \begin{cases} 1 + \frac{f(i) - \bar{f}(t)}{2\sigma(t)} & \sigma(t) \neq 0 \\ 1 & \sigma(t) = 0 \end{cases}$$

where $ExpVal(i, t)$ is the expected value of individual i at time t , $f(i)$ is the fitness of i , $\bar{f}(t)$ is the mean fitness of the population at time t , and $\sigma(t)$ is the standard deviation of the population fitnesses at time t .

We could thus give an individual with fitness one standard deviation above the mean, 1.5 expected offspring. If $ExpVal(i, t)$ was less than 0, we could arbitrarily reset it to 0.1, so that individuals with very low fitness had some small chance of reproducing.

At the beginning of a run, when the standard deviation of fitnesses is typically high, the fitter individuals will not be many standard deviations above the mean, and so they will not be allocated the lion's share of offspring. Likewise, later in the run, when the population is typically more converged and the standard deviation is typically lower, the fitter individuals will stand out more, allowing evolution to continue.

1.4.3 Elitism

"Elitism", first introduced by Kenneth De Jong (1975), is an addition to many selection methods that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance ([2]).

1.4.4 Rank Selection

This is an alternative method in which the population gets sorted by fitness score, and an individual expected value is not determined according to its fitness, but rather by the position (the rank) in which it classifies. Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high. It also keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked i and $i + 1$ will

be the same whether their absolute fitness differences are high or low.

Rank selection has a possible disadvantage: slowing down selection pressure means that the GA will in some cases be slower in finding highly fit individuals. However, in many cases the increased preservation of diversity that results from ranking leads to more successful search than the quick convergence that can result from fitness-proportionate selection.

1.4.5 Tournament Selection

The fitness-proportionate methods described above require two passes through the population at each generation: one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and one pass to compute the expected value of each individual. Rank scaling requires sorting the entire population by rank, a potentially time-consuming procedure.

Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient as it doesn't require any sorting, and more amenable to parallel implementation. Two individuals are chosen at random from the population. A random number r is then chosen between 0 and 1. If $r < k$ (where k is a parameter, for example 0.75), the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected.

An example of this selection technique can be seen in [3].

1.5 Reproduction and Genetic Operators

Reproduction is the phase in which new individuals are actually created. This process depends very much on the encoding chosen to represent the DNA, so here I will proceed describing just some general encoding-agnostic techniques, but a more in depth argumentation will be found later on in the chapter describing the evolution of neural networks.

1.5.1 Sexual and Asexual Reproduction

Researchers have proposed two main methods for generating new agents which differ in the number of parents from which such agents will inherit their traits. The main distinction is between using only one parent, also known as asexual reproduction, and using two or more parents, in this case called sexual reproduction. This distinction is due to the fact that the two techniques require different genetic operations and strategies: both of them will in fact require applying genetic mutations to the DNA to maintain some variance between the agents in the population, but only the multi-parent reproduction will require crossover to take place, and other operators have been proposed for single-parent reproduction ([4]).

1.5.2 Mutation

The mutation operator is one that somehow modifies the DNA. It is usually implemented as a random operation applied to a trait. Suppose for example that we are trying to find a vector of rational numbers \vec{x} that maximizes $f(\vec{x})$. A trait would be a single element \vec{x}_i and a possible mutation operator could look like this:

$$m(\vec{x}_i) = \vec{x}_i + r([-1, +1])$$

where $r(i)$ is an arbitrary random number generator that yields a number in the interval i . Now, this raises a new question: how many elements in the vector, and which, should be mutated? Of course, there is no absolute answer. It is common to define a mutation rate hyper-parameter $R \in [0, 1]$ such that each \vec{x}_i has probability R of being mutated.

The hyper-parameter R is often not constant during the whole simulation, but initialized to a relatively high value and slowly decreased as the simulation advances. This method will increase variance in the initial generations, thus reducing the probability of premature convergence, but it will also allow a more fine grained search later on, after the agents have already found some local optima.

An attempt at improving the choice of this hyper-parameter is to assign each trait its own mutation rate \vec{r}_i and evolve this parameter as well, in the hope that a good mutation rate will yield a better evolution (either faster or less susceptible to the local optima problem).

1.5.3 Crossover

This operator is widely regarded as the most beneficial because, besides creating a good level of diversity, it also has the potential of mixing together the good traits from several individuals.

Single Point Crossover is one of the simplest implementations and tries to mimic the natural process of crossing over of genes between two chromosomes. It assumes two parents \vec{a} and \vec{b} . A random point j so that the children will inherit the first j traits from \vec{a} and the rest from \vec{b} :

$$\vec{c}_i = \begin{cases} \vec{a}_i & i \leq j \\ \vec{b}_i & i > j \end{cases}$$

Crossover has some shortcomings, though. For one thing, it may not produce all possible offsprings. For example, given the parents

$$\vec{a} = (1, 1, *, *, *, 1)$$

$$\vec{b} = (*, *, *, 1, *, *)$$

it could never produce an instance of

$$\vec{c} = (1, 1, *, 1, *, 1)$$

To solve this problem, many variations of this form exist, which also address the problem of using more than two parents, the simplest being selecting a random parent for each gene to be inherited.

Chapter 2

Evolving Neural Networks

Neural networks are the biologically motivated approaches to machine learning, inspired by ideas from neuroscience. Many attempts have been done towards evolving various aspects of neural networks, and the field has taken the name of Neuro-Evolution (NE).

The perceptron is the most fundamental unit of a neural network. It consists of a set of input connections, each with a corresponding weight represented as a floating point number. A perceptron output value is calculated as the activated sum of each weighted input: $f(\vec{x}, \vec{w}) = g(\sum \vec{x}_i \vec{w}_i)$ where g is the activation function. Once the output value is calculated, it can be used as an input value by another perceptron in the network.

A network is thus a graph tracing the input-output relations (synapses) between its nodes (the perceptrons, or neurons). In its simplest feed forward form (as seen in figure 2.1), a network is organized in layers, where each node in a layer uses all the nodes from the previous layer as its input. The first layer is called the input layer, while the last one is the output layer. The ones in between are called hidden layers.

Input layer nodes have no input connections, and their output value is manually set by external entities: such nodes behave as sensors, they just reflect the value of something else. Output layer nodes are instead used to get the final result from the network.

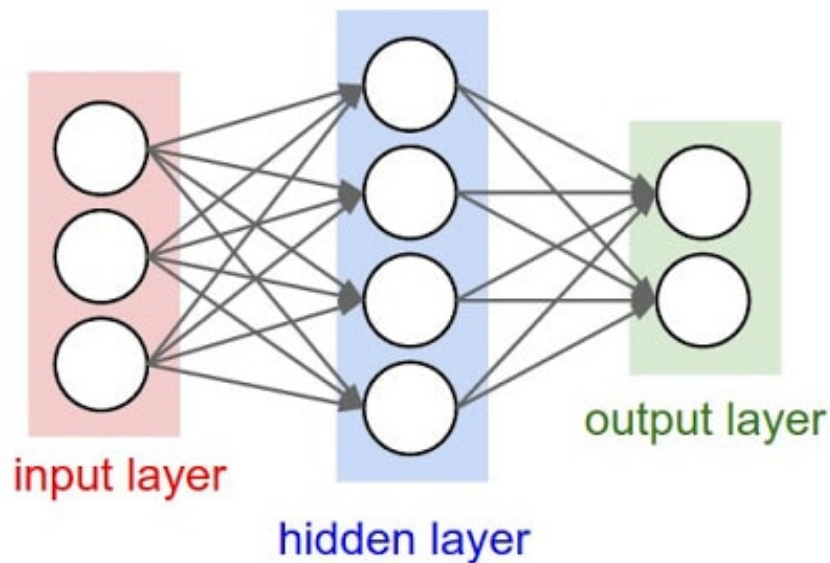


Figure 2.1: A feed forward neural network

Several techniques have been used to train neural networks, which is, finding appropriate weights and/or topologies, and depending on which technique is used, a different methodology for representing the network into the genome must be used.

2.1 Direct Encoding

As stated in the previous chapter, direct encoding employs a one-to-one mapping from genotype to phenotype, so one gene will represent one trait. An advantage of this representation is that it is easier to understand how the network is constructed from the genotype representation. Though, there are some negative effects as well. For example, when dealing with a bigger network, the genotype representation gets inherently larger, which implies a vastest search space for the GA to explore, thus augmenting the time the GA will have to work.

2.1.1 Evolving a weight vector

The first NE approaches used a fixed topology (number of layers and nodes per layer) which was chosen in advance by the experimenter. It could be, for example, a single hidden layer of neurons, with each hidden neuron connected to every network input and every network output. An encoding for such network can be a simple vector of floating point numbers each representing a weight in the network. The GA searches the space of connection weights of this fully connected topology by allowing best-performing networks to reproduce. The weight space is explored through the crossover and mutation of the weight vectors.

Using a fixed topology implies that some knowledge about the size of the final network is needed: when the chosen number of hidden nodes is underestimated, the optimal solution might not exist in the search space, thus requiring the GA to be restarted from scratch with a different topology. On the other hand, if the chosen number of nodes is too large, the search space becomes too big to find the optimal solution.

This technique dates back to 1990-1995 [5, 6] and was the first successful attempt at neuroevolution. Mutation was mostly done by letting every weight have a small chance of changing between generations. Recombination could be done by swapping parts of the vectors. This representation had a big chance of premature convergence and therefore other methods were developed, for example by Moriarty and Miikkulainen [7]; these methods will be discussed later.

One of the most notable successes using weight vector genotype has been that of Gomez et. al [8] in which they made use of a subpopulation for every individual weight in the network, and permuting these subpopulations to create and evaluate networks. In this way they ensured that the population stayed as diverse as possible. They called this technique CoSyNE.

In conclusion, weight vector representations are still applicable and have been gaining popularity in recent years. The fixed topology is still quite restricting, but recent

attempts with fixed topology have been performing better or equal to other more complex topology-evolving methods. This added to the fact that weight vectors are easily comprehensible makes them a viable class of NE algorithms.

2.1.2 Evolving neurons

In an attempt to improve diversity in populations of neural networks, Moriarty and Miikkulainen came up with an innovative idea in 1994 [7]: instead of using populations of networks, they used populations of neurons, each containing a vector of references to its inputs and their relative weight. They called this method *Symbiotic Adaptive Neuro Evolution* (SANE). The goal of this technique was to create specialized versions of individual nodes, each working to solve a task within the network. But a single neuron could not be evaluated on its own, so its score was set to be the average score of the networks in which it participated. NNs were built selecting a fixed amount of random nodes from the population, so that after several networks were created and evaluated, each individual neuron would have been scored at least a couple times. Ranking would have used the average score of the neuron, and the 25% of the population was kept for the reproduction phase, which included crossover and mutation. Crossover was performed by using one-point crossover within the real valued vector that composed each neuron, which creates two offspring for every two parents. In a second moment, mutation would change individual weights with a 1% chance.

In 1996 Gomez and Miikkulainen extended the system of SANE creating sub-populations and called the new technique *Enforced Sub-Populations* (ESP). In this new version, a node is allowed to recombine (through crossover) only with neurons belonging to the same sub-population, thus creating some sort of competition inside each group of nodes. Evolution in ESP proceeds as follows:

1. Initialization: the number of hidden units u and the number of neurons for each subpopulation n are specified and u subpopulations are initialized with n encoded

neurons, each containing two vectors of reals: one for the input weights and one for the output.

2. Evaluation: a certain number of networks (e.g. n times u) are created selecting one random node for each of the u groups. Each one is evaluated and its score gets added to the *cumulative fitness* of each of the neurons that participated into the network.
3. Recombination: the *average fitness* is calculated by dividing the cumulative fitness by the number of trials, and the individuals within subpopulations are ranked by average fitness. Each neuron in the top quarter is recombined with a higher-ranking neuron, creating offspring that replace the lowest-ranking half in the subpopulation.
4. The evaluation-recombination cycle is iterated until a network is found that performs sufficiently well.

Hidden nodes from different sub-populations could develop links to each other after a sufficient amount of generations. Another advantage of ESP in comparison to conventional neuroevolution is that in CNE networks with a single “bad” node could still have a high fitness, in ESP every type of node is evaluated by its own average performance, so this event is not likely to happen. In 2000, Kaikhah and Garlick [9], made another extension to this neuron representation technique, making it possible to build networks with a variable amount of hidden nodes.

These neuron representations were one of the first deviations from the conventional neuroevolution path, and are based on a quite unique representation that is worth mentioning. However, both implicit and other direct representations have been performing better over the last decade.

2.1.3 Evolving Topologies

One of the first attempts at evolving the structure of a network as well as its weights has been that of Stanley and Miikkulainen in 2002 which argued that the topology of

a network also affects its functionality. In their opinion, an algorithm that could find parameters such as the network size or the nodes' connections on its own would render it much more robust, also eliminating the need of trial and error heuristic to find those parameters.

Neuro Evolution through Augmenting Topologies

Abbreviated NEAT, the technique was published by Stanley and Miikkulainen in 2002. It is not the first topology evolving algorithm (Dasgupta and McGregor in 1992 already were dipping their toes into the argument, though their approach was far less elaborated [11]) but it has brought a big wave of innovation into the field of neuroevolution, as it achieved unprecedented efficiency and adaptability through its ability to generate complex topologies starting from minimal networks, thus keeping its structure as small as the requirements allowed [17, 18].

NEAT solved a problem that was common in previous topology evolving strategies, called *Competing Conventions*, which happens when different genotypes encode for the same phenotype. This phenomena could have several negative repercussions in the GA, because two fit individuals that accidentally had the same phenotype, but different genotype, will produce a less fit offspring when crossed-over. Two novel intuitions allowed NEAT to overcome such problem:

1. To battle the competing conventions problem, NEAT would store ancestral information of each individual.
2. NEAT uses speciation (subpopulations) and different fitness measures for newly formed topologies and fully weighted networks.

In NEAT, the initial population is set to have networks with no hidden nodes, just the input and output ones, and is not bound to be a feed-forward network, but has the capacity to evolve in any graph. The encoding used by the algorithm specifically developed to allow meaningful structural crossover, where each genome consists of a set of nodes and a set of connections (see figure 2.2). A gene for a node contains its ID, its type (either input, output or hidden) and an innovation number, while each connection

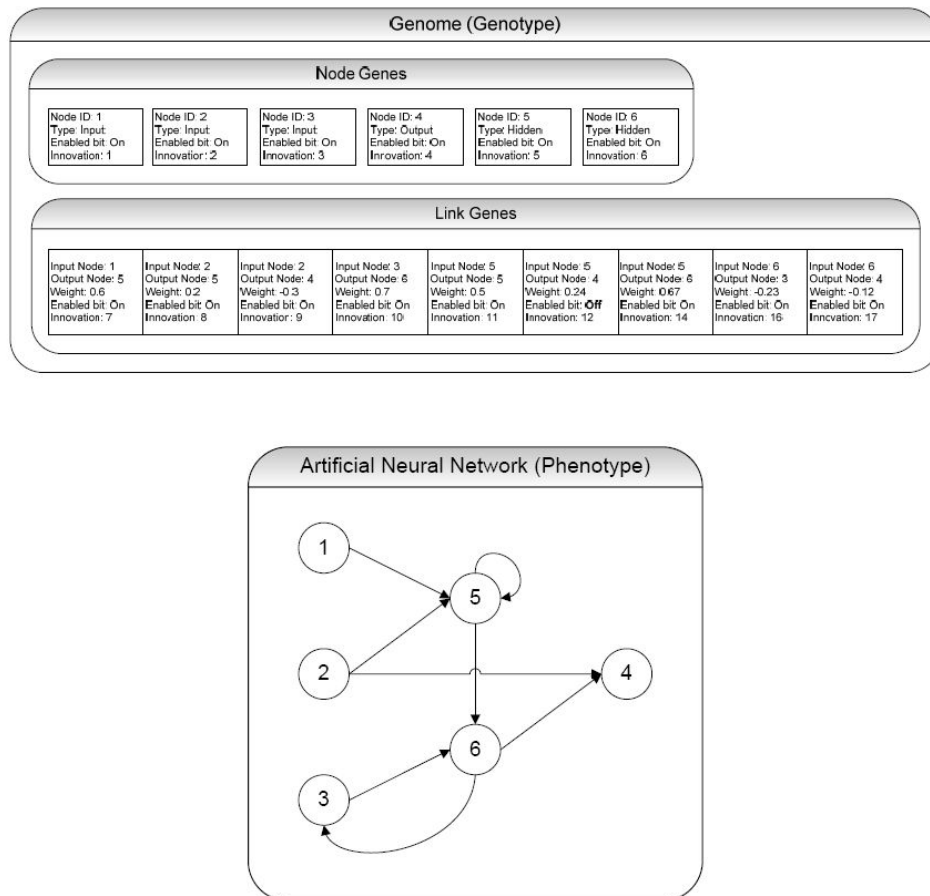


Figure 2.2: Genotype and Phenotype Example for the NEAT Algorithm. Above is an example of a genotype that represents the displayed phenotype. There are six nodes: three input, two hidden and one output. There are nine links, two of which are recurrent and one of which is disabled. The disabled gene (connecting nodes 5 and 4) is not displayed. [12]

specifies the input and output nodes, the connection weight, whether the connection is enabled, and an innovation number, which is used to identify corresponding genes.

When the GA mutates a link, it randomly chooses two nodes and inserts a new link gene with an initial weight of one. If a link already existed between the chosen nodes but was disabled, the GA re-enables it. Finally if there is no link between the chosen nodes and an equivalent link has already been created by another genome in this population this link is created with the same innovation number as the previously created link as it is not a newly emergent innovation.

A node mutation is similar to a link mutation but differs from it in that instead of choosing two nodes and inserting a link, the GA chooses and disables an existing link and inserts a node. The GA inserts this new node with a random activation value, as well as two link genes to connect the node to the now-disabled link's previous input and output nodes. The GA then transfers the weight from the disabled link gene to the new link gene, which is connected to the old output neuron. The weight of the link gene inserted between the new neuron and the old input node is set to one so as not to disturb any learning that has already occurred in this connection. Introducing a new node where a link once existed may fragment some evolved knowledge in the phenome. Copying the original link weight to one of the new node's links while setting the other connecting link weight to one minimizes the disturbance in learning.

Crossover is not always obvious to implement when optimizing dynamic genotypes, as it is the case with topology evolving neural networks, because the structures of different genotypes are not necessarily related. The innovation number (IID) cited above is the piece needed to solve this problem, serving as the link's historical marking, denoting hereditary information about the gene. Crossover happens between two parents, and the offspring will contain the union of genes from both parents, but if a node or link is present in both parents (has the same IID) it is taken only once.

The technique employed by NEAT allows the GA to build increasingly complex

genomes without imposing constraints on how parents are coupled together, but this level of complexity goes against the GA itself, as it implies a larger search space. Also, a genetic innovation may not directly influence positively the resulting network, but may require subsequent innovations to express its full potential. To prevent the deletion of such young genomic innovations, NEAT employs *speciation*.

In natural evolution entities that once shared a common genome sometimes diverge so much that they can no longer mate with one another. This divergence is known as speciation. In NEAT, as the genomes in a population grow complexity a new innovation in their topology may result in greater performance for the population's agents. NEAT uses speciation to protect such innovations. When an agent's structure diverges far enough from that of the other agents in the population NEAT identifies it and places it in its own species. Using innovation numbers NEAT can calculate the distance between two genomes. The distance is defined by the following function:

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

where E is the number of excesses (genes which are newer than any other gene in the other parent), D is the number of disjoints (the number of genes not common to both parents), \overline{W} is the average of weight differences of the two genomes; the arbitrary coefficients c_1 , c_2 , and c_3 are arbitrary and modify the weights of each of the variables and N is the number of genes in the larger of the two genomes.

In the initial generation only one species is present; the population is uniform. As complexity raises, distance between genomes increases until one will surpass an arbitrary threshold, at which point a new species is defined and it becomes the champion for the new species. As genomes distances from the champion of their species, they may also be moved to a different existing species, if their distance to a foreign champion becomes small enough.

This method not only creates an internal competition within each species, but makes sure that only similar genotypes get selected as parents in the crossover, thus making the recombination more incline at preserving those traits that have been previously learned,

and making sure they do not get deleted too early.

2.2 Indirect Encoding

Natural DNA can encode complexity at enormous scale, and this is due to the fact that it does not contain all the information about each trait that it represents, but rather describes it in a more abstract language made of definitions and rules. Researchers are attempting to achieve the same representational efficiency in computers by implementing developmental encodings, i.e. encodings that map the genotype to the phenotype through a process of growth from a small starting point to a mature form. A major challenge in this effort is to find the right level of abstraction of biological development to capture its essential properties without introducing unnecessary inefficiencies. Indeed, a good representation of the solution space can mean the difference between success and failure. In biology, the genes in DNA represent astronomically complex structures with trillions of interconnecting parts, such as the human brain [14, 15]; rather, somehow, less than 30 thousand genes encode the entire human body [16]. This observation has inspired an active field of research in artificial developmental encodings. Abstractions range from lowlevel cell chemistry simulations to high-level grammatical rewrite systems [19]. Because no abstraction so far has come close to discovering the level of complexity seen in nature, much interest remains in identifying the properties of abstractions that give rise to efficient encoding. Here one technique is presented as a reference.

2.2.1 Grammatical Encoding

The method of grammatical encoding can be illustrated by the work of Hiroaki Kitano ([29]) which points out that since direct-encoding methods explicitly represent each connection in the network, repeated or nested structures cannot be represented efficiently, even though these are common for some problems. The solution pursued by Kitano and others is to encode networks as grammars; the GA evolves the grammars, but the fitness is tested only after a “development” step in which a network develops from the grammar. That is, the “genotype” is a grammar, and the “phenotype” is a network derived from

that grammar.

Kitano applied this general idea to the development of neural networks using a type of grammar called a “graph-generation grammar” from which it was possible to derive an architecture (but not the weights).

The fitness of a grammar was calculated by constructing a network from the grammar, using back-propagation with a set of training inputs to train the resulting network to perform a simple task, and then, after training, measuring the sum of the squares of the errors made by the network on either the training set or a separate test set.

The GA used fitness-proportionate selection, multi-point crossover (crossover was performed at one or more points along the chromosome), and mutation. A mutation consisted of replacing one symbol in the chromosome with a randomly chosen symbol from the A-Z and a-p alphabets. Kitano used what he called “adaptive mutation”: the probability of mutation of an offspring depended on the Hamming distance (number of mismatches) between the two parents. High distance resulted in low mutation, and vice versa. In this way, the GA tended to respond to loss of diversity in the population by selectively raising the mutation rate.

Kitano’s idea of evolving grammars is interesting, and his informal arguments are plausible reasons to believe that the grammatical encoding method (or extensions of it) would work well on the kinds of problems on which complex neural networks could be needed. However, the particular experiments used to support the arguments were not convincing, since the problems may have been too simple.

2.2.2 Compositional Pattern-Producing Networks

A compositional pattern-producing network (CPPN for short, proposed by Kenneth O. Stanley [13] in 2007) is basically a way of compressing a pattern with regularities and symmetries into a relatively small set of genes. This idea makes sense because natural brains exhibit numerous regular patterns (i.e., repeated motifs) such as in the receptive fields in the visual cortex. CPPNs can encode similar kinds of connectivity patterns. The fundamental insight behind this encoding is that it is possible to directly describe the

structural relationships that result from a process of development without simulating the process itself. Instead, the description is encoded through a composition of functions, each of which is based on observed gradient patterns in natural embryos.

CPPNs are structurally similar to artificial neural networks used in most neuroevolution techniques, thus they can take full advantage of all the existing methods for neuroevolution. In particular, Stanley modified his previous works in NEAT to create CPPN-NEAT (later republished as HyperNEAT [20]), which evolves increasingly complex CPPNs. In this way, it is possible to evolve increasingly complex phenotype expression patterns, complete with symmetries and regularities that are elaborated and refined over generations.

The idea is actually quite simple, but very powerful at the same time. In CPPN an ANN is evolved with any traditional technique, but instead of using it to directly express the phenotype, it is used by a function that takes the parameters' spatial coordinates and produces a value for the parameters. This feature of having spatial coordinates for the input domain is almost never considered, but may often give vital information to the learning algorithm, for example when images are used as input. For those problems where the input does not have any spatial information - and for the hidden layers which miss this information as well - CPPN is able to generate and evolve such information.

2.3 Optimizing Backpropagation Problems

Backpropagation is the currently one of the preferred base algorithm for finding the correct weights in a neural network [21, 22, 23]. As architectures become larger, tuning the hyper-parameters [25] is getting more and more difficult and time consuming, especially for deep neural networks where many layers of neurons are added on top of each other making the evaluation of hyper-parameters computationally expensive and time consuming. Hyper-parameters are the variables which determines the network structure (e.g. number of hidden units) and the variables which determine how the network is

trained (e.g. learning rate). They are set before training (before optimizing the weights and bias) but some can be changed even as the training occurs to fine tune the learning process.

Some of the most common methods for finding appropriate hyper-parameters include manual search, grid search, random search [24], bayesian optimization and genetic algorithms.

Manual search is the most naive method for tuning hyper-parameters in which an expert uses his knowledge and experience to tune the parameters.

Grid Search or parameter sweep, is simply an exhaustive searching through a set of manually specified subset of the hyper-parameters of the learning algorithm. This search uses a brute-force like approach where all possible combinations of parameters are used to generate different training sessions and outputs the parameters that yielded the best results in the validation process.

Random Search works in the same fashion as the grid search, but instead of using an exhaustive search through all the subset of hyper-parameters, it only selects a few of them randomly. This method allows the experimenter to specify the distribution of hyper-parameters from which to sample, thus using prior knowledge to render the search more efficient.

Bayesian Optimization starting from a defined set of hyper-parameters, it continuously tests noised versions of the best performing values for each parameter, thus creating a balance between exploration (trial of values whose outcome is uncertain) and exploitation (using values close to the best performing ones) to produce results that are often better than the previous techniques described.

Evolutionary Optimization Evolutionary optimization is a methodology for the global optimization of noisy black-box functions. In hyper-parameter optimization, evolutionary optimization uses genetic algorithms to search the space of hyper-parameters.

As seen so far, a population of candidate solutions must be created a priori, and a fitness evaluation function that takes the hyper-parameters as input and assigns them a score is needed to start the evolutionary process.

2.3.1 Evolving the topology of deep neural networks

As seen for the grammar encoding used by Hiroaki Kitano in 1990, GAs have a long history of living in symbiosis with the backpropagation algorithm, leaving to the latest the responsibility of finding the correct weights for the synapses, and letting the GA train all the rest (e.g. the topology of the network). This coupling has demonstrated to be very reliable and reliable in surpassing conventional problems such as overfitting or getting stuck in local minima, but has the drawback of being embarrassingly time-consuming, as the fitness function will need to build, train and validate a whole neural network from scratch before being able to assign a score to the genome that produced it. With the increasing processing power in the latest decades, and the need for ways to automate the hyper-parameter discovery in the increasingly complex neural networks (as it is the case with deep neural networks), the argument has attracted new researchers and the field has become much more active.

One of the most relevant results has been achieved recently by Esteban Real et al. in 2017 [3] and proves that genetic algorithms can be able, given enough computational power, to evolve the structure of very complex deep neural networks, producing results comparable to those produced by the manual tuning experts. Specifically, the paper shows how the evolutionary process discovered models for solving the CIFAR-10 and CIFAR-100 datasets. To achieve such results, the researchers introduced novel genetic operators that could introduce, erase or change whole layers at a time, thus being able to cope with very large scale networks. While they experimented three kinds of crossover between individuals (the networks), Esteban et al. affirm that it didn't affect the performance of the process. The results have shown great accuracy, but at a very high computational cost. Nonetheless, this experiment is a demonstration of how effective the coupling between the two approaches can be.

Chapter 3

Experimentation

A program has been built to experiment with the different techniques, which is an attempt at evolving both the weights and the topology of a neural network.

The shape of the resulting network can be any directed graph with the only constraint that given two nodes n_i and n_j , there may exist at most one connection that starts from the first and gets to the second.

A notable thing is that, as in ESP, the entity being evolved is not the network itself, but rather its neurons taken independently, so there exist many populations, one for each neuron that has been discovered. The links between neurons are stored inside the neuron from which they originate, and thus are part of their genome.

The perceptron mechanic has been redesigned in an attempt to dynamically reduce the number of calculations required to produce a result: traditional neural networks must compute each neuron value before getting the output nodes' values, but this is a very unnatural thing. Suppose an ANN must be found that labels images in which there may be either a cat or a bird. In such case there may not always be the need to compute all of the neurons' output, instead it would be desirable that the network halted as soon as it detected a beak, a tail, or any other detail from which the answer could be derived with a sufficient degree of certainty.

The proposed perceptron implementation potentially allows, under the right circumstances, the formation of such behavior, but while it proved to yield reliable results, no effort has yet been spent into measuring and improving its efficiency.

3.1 Gene Pools

As Richard Dawkins described in the book *The Selfish Gene*, genes do not compete against each other, as crossover does not make any selection on them. Instead, the only fear of a gene is represented by its alleles. An allele is defined to be an alternative form of the same gene, occupying the same spot on the genotype. Alleles come into existence every time a mutation hits the gene. During crossover, the two parent genomes may present different alleles for the same gene, thus generating the rivalry. This is the philosophy followed during the creation of this algorithm: the genes are implemented as pools of alleles, where each pool stores all the discovered mutations for one neuron, keeping track of how beneficial each variant has been to the neural network making use of it.

```
1  GenePool {
2    genes: Map(String -> Vector(Allele))
3    hyper_params: {
4      mutation_rate: Float,
5      max_pool_size: Integer,
6      ...
7    }
8  }
9  Allele {
10   id: String, // generated randomly, see sect. Implementation
11   cumulative_score: Float
12   evaluation_count: Integer
13   node: Node
14 }
```

Fitness sharing

The evaluation technique used to assign each allele its score reflects that proposed by ESP, where the fitness score achieved by the network is shared across all of its nodes. This is done by looking up the corresponding allele in the gene pool for each node (or creating one if the node is new or is a mutation unknown to the gene pool), adding the score to the `cumulative_score` of the allele and incrementing the `evaluation_count`: these two values will be needed to compute the average fitness score in the selection algorithm.

3.2 Neural Network Mechanics

3.2.1 The Neuron

```
1  Node {
2    name: String
3    allele_id: Null or String
4    type: NodeType
5    value: Float
6    reset: Float
7    threshold: Float
8    links: Vector(Link)
9  }
```

Where the `name` is decided arbitrarily for input and output nodes, and is generated randomly for the hidden nodes. `allele_id` is needed to keep track of which allele generated the node, `type` is can be either *input*, *output* or *hidden*; `value` is only used to represent the neuron state in a given moment, `reset` is is the value assigned to `value` every time the node activates, `threshold` is the value it must reach in order to be activated, and a vector of outgoing links, and finally `links` is the set of outgoing synapses.

Input nodes are usually created with a `threshold` lower than the value they will represent, thus making sure they will always get activated when an external signal stimulates them.

3.2.2 The Synapse

```
1  Link {
2    weight: Float
```

```
3     delay: Integer // (min: 1)
4     output: String
5 }
```

Each link is composed of three values: a `weight` that can be both positive and negative, a `delay` which defines how many time steps are required for the signal to reach its destination - lower bounded to one, as signals are defined not to be instantaneous - and the latest parameter contains the name of the target neuron.

3.2.3 The Signal

```
1 Signal {
2     value: Float
3     time: Integer
4     target: String
5 }
```

`value` specifies the value to be added to the neuron named `target` at time `time`. There exist two ways in which signals get generated: the first happens during initialization, where signals are manually created by factors external to the network (the actual input of the network gets “transmitted” to it); the other way a signal can be created, is when a neuron gets activated: right after this happens, several signals are generated, one for each of the neuron outputs. In this case, the `value` of the signal is equal to the value of the source node at the time of activation multiplied by the link `weight`; `time` is set to the current network `time` plus the link `delay`.

3.2.4 The Network

```
1 Network {
2     time: Integer
3     nodes: Map(String -> Node)
4     signals: Map(Integer -> Vector(Signal))
5 }
```

Where `time` is the current network time, it starts from zero and gets incremented after each cycle, `nodes` is the set of nodes participating in the network mapped by their

name, and `signals` is the structure used to index all the signals by time of arrival (e.g. `signals[t]` will return all the signals reaching their target at time `t`).

A network is started by setting all nodes to their `reset` value, and then filling `signals[0]` with one signal for each input value, setting `target` to the corresponding input node and `time` to 0.

Once a network `net` is initialized, a `tick()` procedure is called in loop for an arbitrary number of times. Such procedure will do three things:

- for each signal `sig` in `net.signals[net.time]`, the output node is updated:
`net.nodes[sig.target].value += sig.value`
- each node in the network is checked for activation: if a node is found to be active (`node.value > node.threshold`) then new signals are generated for each of the node links and inserted into the `net.signals` map, indexed by time of arrival;
- the network time is incremented by one: `net.time += 1`

When the `tick()` loop ends, the output nodes' values can be used as the result. An example of the network usage follows:

```
1 fn process(net: Network, n: Integer, input: Map(String -> Float)) {
2   net.reset() // resets each node, net.time and net.signals
3   for (target, value) in input:
4     net.addSignal(value, time=0, target)
5   repeat n times:
6     net.tick()
7 }
```

3.3 Network Generation

SANE and ESP use the same schema, allocating one pool for each neuron, but a big change has been made in how the network gets built. SANE is only able to connect neurons to the input and output nodes, thus creating exactly one hidden layer; ESP extends

this allowing neurons to connect to themselves, creating recurrent links. Both algorithms will use a fixed amount of random neurons from all those available to generate a network.

To generate a network, a recursive procedure is called iteratively for each of the input and output nodes. Such procedure accepts a gene pool, a network `net` and a node name `name` as inputs, selects one node from the pool corresponding to the node name and inserts it into the network nodes. It then calls itself recursively for each of the node links:

```
1  fn add_node(gp: GenePool, net: Network, name: String) {
2      if name in net.nodes: return // do not add the same node twice
3      node = select_from_pool(gp.genes[name])
4      net.nodes[name] = node
5      for link in node.links:
6          add_node(gp, net, link.output)
7  }
8
9  fn create_net(gp: GenePool,
10             input: Vector(String),
11             output: Vector(String)) -> Network {
12      net = new Network()
13      for name in input: add_node(gp, net, name)
14      for name in output: add_node(gp, net, name)
15      return net
16  }
```

This is enough to generate any kind of directed graph neural network - traditional NNs as well - in a way where each node can be selected from a pool of possible alternatives, determining on its turn how other parts of the network are generated.

3.4 Genetic Operators

Genetic operators include mutation and crossover between alleles and resemble those introduced by NEAT. Mutation is implemented at the network level and can be one of the following:

- `mut_link_del` deletes a link from a random node.

- `mut_link_new` adds a link to a node, using a non-input node as the target of the connection.
- `mut_split` splits an existing link, inserting a new node between the source and the destination.
- `mut_reset` and `mut_threshold` alters the reset/threshold value of a random neuron by adding a random value in $[-1, 1]$
- `mut_weight` alters the weight of a randomly chosen link
- `mut_delay` changes the delay of a random link, adding an integral value in $[-4, 4]$

and each mutation has probability $mrate \in [0, 1]$ of happening. If the mutation happens, then it may happen again with the same probability. This is valid until the mutation does not happen:

```

1  fn mutate(net: Network, mrate: Float) {
2      while rand(0, 1) < mrate: net.mut_link_del()
3      while rand(0, 1) < mrate: net.mut_link_new()
4      ...
5  }
```

Crossover on the other hand is implemented at the gene pool level and may happen with probability $mrate_c \in [0, 1]$ every time there is the need to select one allele from a pool. Crossover mixes two alleles from the same pool by creating a new node of the same gene which inherits `reset` and `threshold` independently from either parent with equal chance. The `links` vector is instead the union of all the links present in the parents. If the resulting node has two links directed to the same target, one of them selected at random is deleted; this is repeated until no two links have the same target.

Every time a node is mutated or created, its `allele_id` is set to `None`. The `GenePool` will fill the field once the node is sent back to it with a fitness score.

3.5 Implementation

The algorithm has been implemented in the rust programming language to get maximum CPU performance, and can be run on multiple cores and multiple computers in

parallel: two dependent modes of execution thus exist: *master* and *worker*.

The role of the master is to coordinate the workers by hosting a `GenePool` and providing an interface - a RPC server - that generates new neural networks on demand and accepts evaluations (along with the networks that produce them) to be recorded in the pool.

A worker is thus programmed to require NNs from the server, evaluating them and sending them back to the server. To reduce load on the master, and to increase exploitation, the worker will not test only the received network, but instead it will run a local genetic algorithm (LGA), implemented with the same algorithm used for the master - the `GenePool` - to evolve a certain amount of networks starting from the received one; the best network found is finally sent back to the server with its score.

```
1  fn worker(master: Master) {  
2      while True:  
3          job = master.get_job()  
4          best = run_local_ga(job.network, job.params)  
5          master.send_evaluation(best.score, best.network)  
6  }
```

3.6 Testing and results

The proposed algorithm has been tested on the classification of the MNIST dataset, which contains 70.000 28x28 gray-scale images, each representing a decimal digit.

The dataset has been split into three parts:

- the train set (50.000 images): each LGA uses a very small mini-set of 10 to 100 images selected randomly from the train set;
- the validation set (10.000) is used to recalculate the score of best network found from the LGA before sending it to the master. To reduce the time needed for the evaluation, the set is split into slices and a network is only evaluated on one slice. The master dictates which slice must be used, and is able to change this parameter automatically when it detects overfitting on the current validation set slice;

- the test set (10.000 images) is only used offline (e.g. the results are never used by any GA) to effectively test a network performance.

3.6.1 Evaluation

The networks in this test are created with one input neuron per pixel (784 pixels per image) and ten output neurons - one per label. The final output is activated using the soft-max function, thus producing a vector of ten probabilities whose sum is one. The desired output vector for an input i is set to be:

$$\vec{o}_j = \begin{cases} 1 & \text{if } j \text{ is the correct label for the image } i \\ 0 & \text{otherwise} \end{cases}$$

A correct guess (a success) happens when the maximum value in the output vector is in position j , being j the correct label. The fitness score of a network on a certain dataset \vec{d} of N images is defined by:

$$f(\vec{d}) = 10\frac{s}{N} - \frac{\sigma^2}{N}$$

where s is the number of successes and $\frac{\sigma^2}{N}$ is the average variance between the desired output and the the actual output. The score can thus variate from -1 up to 10 in case of perfect accuracy.

3.6.2 Results

The algorithm needs around 10.000 reports from the workers to produce a network achieving 60% success rate, but the improvements start slowing down dramatically after the 75% accuracy is reached. The best networks produced by the algorithm usually reach a success rate slightly over 80%. The main cause for such high error rate is probably due to the limited size - between 100 and 1000 - of the validation set, which allows the network to memorize it without actually learning the general rule to distinguish the digits, even when changing the validation set periodically, as visible in figure 3.2.

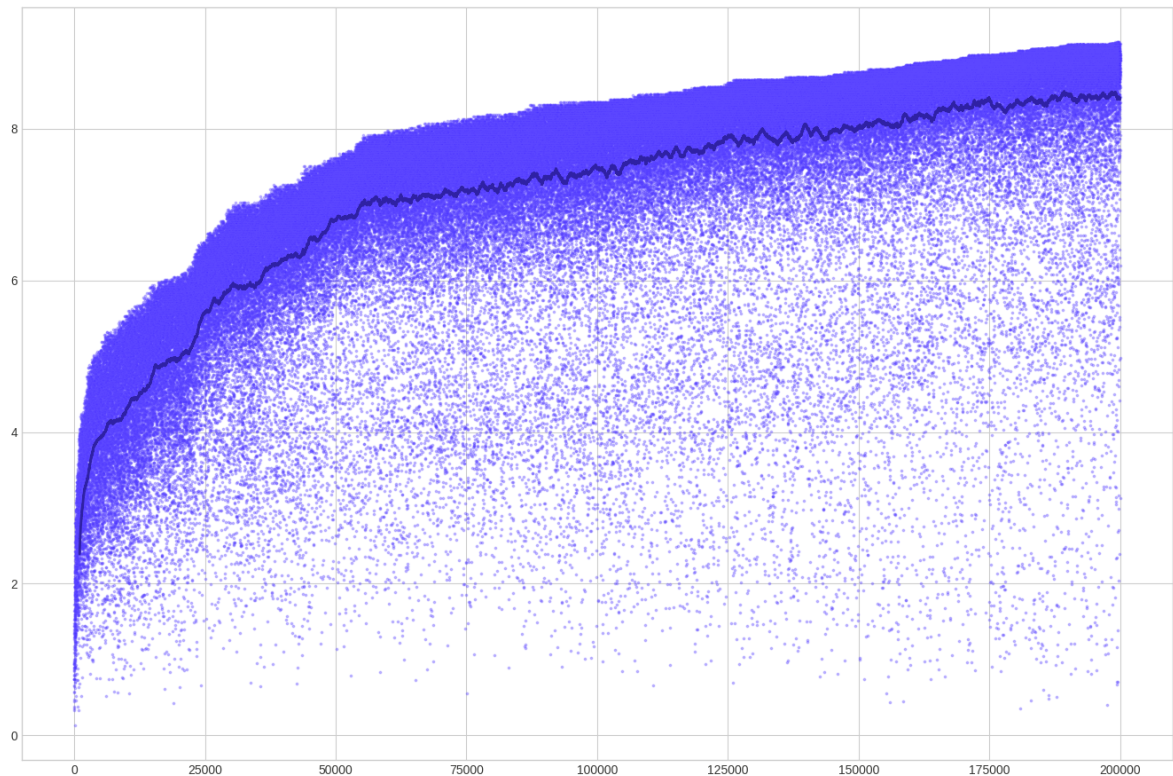


Figure 3.1: the chart displays how reports (dots) sent to the master increasingly get better over time. The x axis displays the report number and the y axis represents the fitness value; the darker line shows the moving average over 1000 reports. In this trial, the validation set slice was fixed, thus provoking overfitting. The best network scored 9.14 (over 91% accuracy on the validation set) but when evaluated on the test set it only yielded a 74.4% success rate.

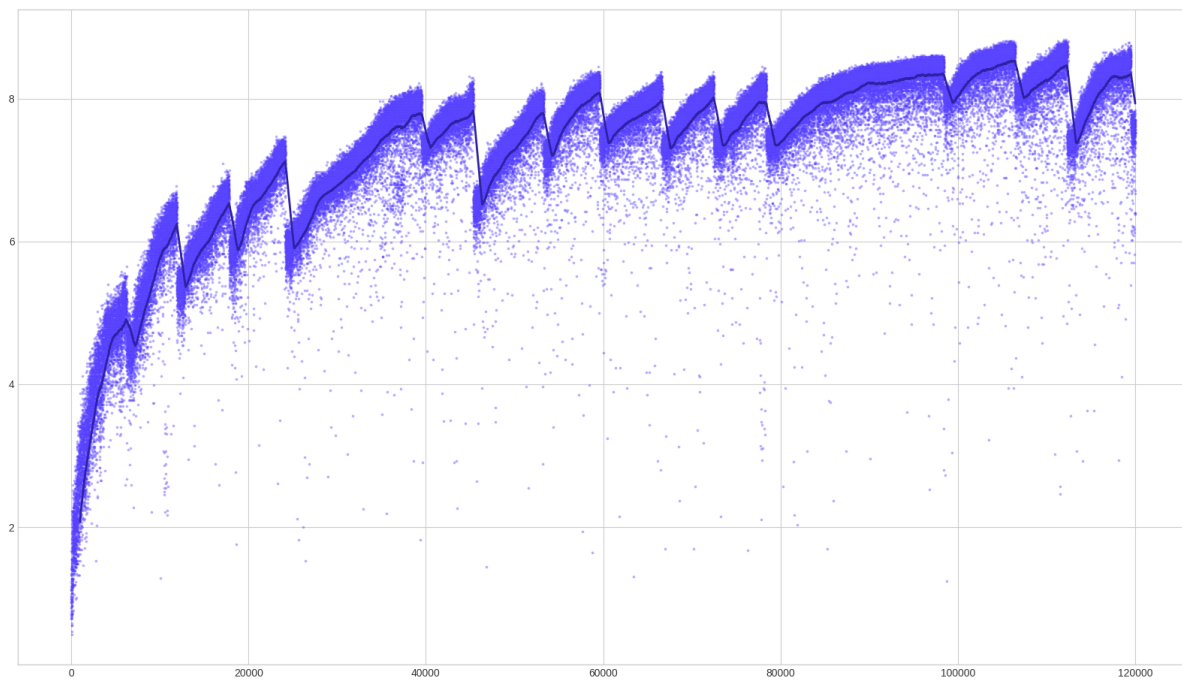


Figure 3.2: When overfitting is detected, the master chooses a new validation set and broadcasts it to the workers. The drops are in correspondence of such events and show how much overfitting is actually happening. The idea behind the validation test swaps is to constantly reward those neurons that are less susceptible to the problem, as they should be the ones to stand out during the drops, thus being more favorable to selection. The best network in this simulation scored 8.82 on the validation which yielded a 78.2% success rate on the test set

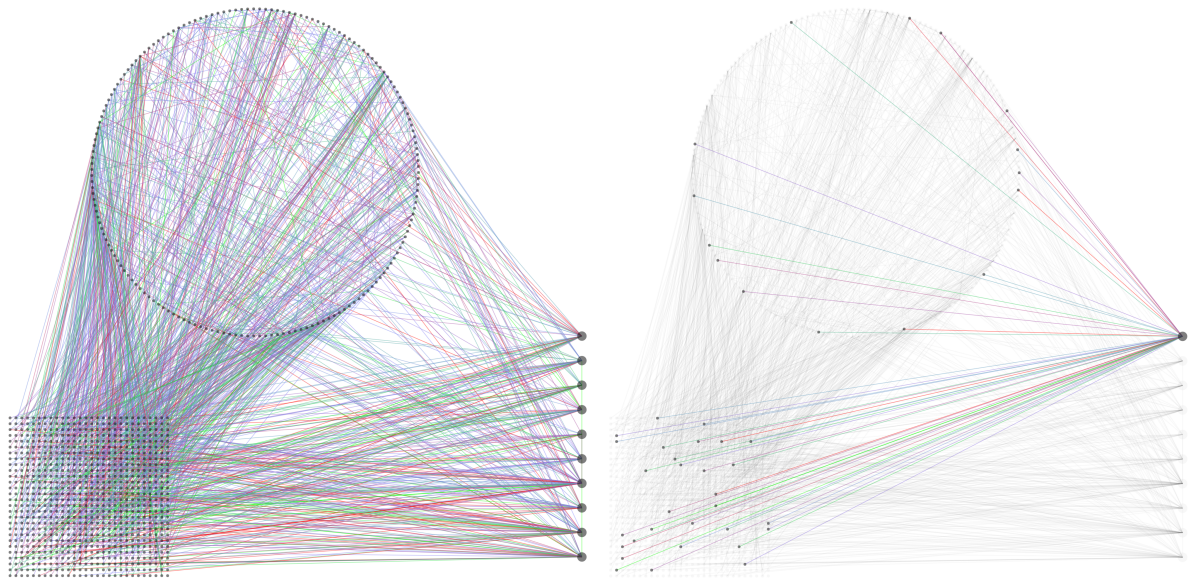


Figure 3.3: A visualization of a network achieving around 40% performance. The dark dots represent neurons, while the synapses are represented by the lines. Input nodes are disposed in a matrix in the lower left corner, the ten output nodes are on the right, and the hidden nodes are disposed in circle. The coloration represents the weight of the link: blue denotes weights close to zero, green is for positive values and red for the negative ones. On the right, the same network is shown emphasizing the direct connections of the label for the output 0.

Chapter 4

Conclusions

Genetic algorithms and their relations with neural networks have been clarified, and while the experimental results obtained classifying the MNIST dataset are not comparable to those of the literature, they provided a base on which to try novel approaches. This experiment also demonstrates how the artificial neural networks can be deeply redesigned, and while on one hand backpropagation and similar approaches wouldn't probably work on these new architectures, evolutionary algorithms offers a way to configure them with very low effort. It would be intriguing to build a program that generated implementations of ANNs with novel mechanics training them using genetic algorithms, in the search for the most effective and efficient solutions.

4.1 Future developments

As overfitting seemed to be the major obstacle in learning general tasks, the fitness evaluation function can probably be reviewed, for example it would be insightful to build the validation set on those images on which the network fail the most. Another interesting aspect would be to try the GA on a very different domain, such as building a controller for a virtual robot, as this is one domain on which genetic algorithms show promising results.

Also, the theoretical potential efficiency of the proposed neural network will be tested more scrupulously to understand if, once trained properly, it would be able to limit the amount of circuits - and thus the computational power - used on a per-input basis.

Bibliography

- [1] Konstantinos Parsopoulos, V. P. Plagianakos, George D. Magoulas, George D. Magoulas, Michael N. Vrahatis, Michael N. Vrahatis: "*Objective Function "Stretching" to Alleviate Convergence to Local Minima*" (2001)
- [2] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan: "*A fast and elitist multiobjective genetic algorithm: NSGA-II*" (2002, in IEEE Transactions on Evolutionary Computation)
- [3] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Sue-matsu, Jie Tan, Quoc Le, Alex Kurakin: "*Large-Scale Evolution of Image Classifiers*" (2017, arXiv:1703.01041)
- [4] Alireza F., Mohammad B. M., Taha M., Mohammad Reza S. M.: "*ARO: A new model-free optimization algorithm inspired from asexual reproduction*" (2010, <https://doi.org/10.1016/j.asoc.2010.05.011>)
- [5] Richard K. Belew, John Mcinerney, and Nicol N. Schraudolph: *Evolving networks: Using the genetic algorithm with connectionist learning* (1990)
- [6] Alexis Wieland: *Evolving neural network controllers for unstable systems* (1991, in Proceedings of the International Joint Conference on Neural Networks)
- [7] David E. Moriarty and Risto Miikkulainen: *Efficient reinforcement learning through symbiotic evolution* (1996, in Machine Learning, (AI94-224):11-32)

-
- [8] Faustino Gomez, Juergen Schmidhuber, and Risto Miikkulainen: *Efficient non-linear control through neuroevolution* (2006, In Proceedings of the European Conference on Machine Learning, pages 654-662, Berlin, 2006. Springer.)
- [9] Khosrow Kaikhah and Ryan Garlick: *Variable hidden layer sizing in elman recurrent neuro-evolution* (2000 in Applied Intelligence, 12:193-205)
- [10] Kenneth O. Stanley and Risto Miikkulainen: *Evolving neural networks through augmenting topologies* (2002 in Evolutionary Computation, 10(2m:99-127)
- [11] Dipankar Dasgupta and Douglas R. Mcgregor.: *Designing application-specific neural networks using the structured genetic algorithm* (1992, In Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks, pages 87-96. IEEE Computer Society Press)
- [12] AUTOMATONS ADRIFT <http://www.automatonsadrift.com/>
- [13] Kenneth O. Stanley: *Compositional Pattern Producing Networks: A Novel Abstraction of Development* (2007)
- [14] Kandel, E. R., Schwartz, J. H., and Jessell, T. M.: *Principles of Neural Science* (1991)
- [15] Zigmond, M. J., Bloom, F. E., Landis, S. C., Roberts, J. L., and Squire, L. R.: *Fundamental Neuroscience* (1999)
- [16] Deloukas, P., Schuler, G. D., Gyapay, G., Beasley, E. M., Soderlund, C., Rodriguez-Tome, P., Hui, L., Matise, T. C., McKusick, K. B., Beckmann, J. S., Bentolila, S., Bihoreau, M., Birren, B. B., Browne, J., Butler, A., Castle, A. B., Chiannilkulchai, N., Clee, C., Day, P. J., Dehejia, A., Dibling, T., Drouot, N., Duprat, S., Fizames, C., and Bentley, D. R.: *A physical map of 30,000 human genes* (1998 in Science)
- [17] Stanley, K. O., Bryant, B. D., and Miikkulainen: *Real-time neuroevolution in the NERO video game* (2005, in IEEE Transactions on Evolutionary Computation Special Issue on Evolutionary Computation and Games)

-
- [18] Stanley, K. O., and Miikkulainen, R.: *Competitive coevolution through evolutionary complexification* (2004, in *Journal of Artificial Intelligence Research*, 21:63-100)
- [19] Maryam Tayefeh Mahmoudi, Fattaneh Taghiyareh, Nafiseh Forouzideh, Caro Lucas: *Evolving artificial neural network structure using grammar encoding and colonial competitive algorithm* (2012)
- [20] Kenneth O. Stanley, D'Ambrosio, David B., Gauci, Jason: *A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks* (2009)
- [21] Jose Miguel Hernandez-Lobato, Ryan P. Adams: *Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks* (2015)
- [22] Danilo Jimenez Rezende, Shakir Mohamed, Daan Wierstra: *Stochastic Backpropagation and Approximate Inference in Deep Generative Models* (2014)
- [23] Jürgen Schmidhuber: *Deep learning in neural networks: An overview* (2015)
- [24] James Bergstra, Yoshua Bengio: *Random Search for Hyper-Parameter Optimization* (2012)
- [25] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, Balázs Kégl: *Algorithms for Hyper-Parameter Optimization* (2011 in "Advances in Neural Information Processing Systems 24")
- [26] Dougal Maclaurin, David Duvenaud, Ryan P. Adams: *Gradient-based Hyperparameter Optimization through Reversible Learning* (2015)
- [27] Jasper Snoek, Hugo Larochelle, Ryan P. Adams: *Practical Bayesian Optimization of Machine Learning Algorithms* (2012)
- [28] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, Ryan P. Adams: *Scalable Bayesian Optimization Using Deep Neural Networks*
- [29] Hiroaki Kitano: *Designing Neural Networks Using Genetic Algorithms with Graph Generation System* (1990)

- [30] Jason Gauci, Kenneth O. Stanley *Autonomous Evolution of Topographic Regularities in Artificial Neural Networks* (2010, in *Neural Computation* Volume 22 — Issue 7 — p.1860-1898)