Hybridization of Neural Networks and Genetic Algorithms

Ethan Tatlow
Advisor: Márton Kajó
Seminar Future Internet WS 2017/18
Chair of Network Architectures and Services
Departments of Informatics, Technical University of Munich
Email: tatlow@in.tum.de

ABSTRACT

Artificial neural networks are a versatile type of computing system that are capable of learning to compute functions through observation, as opposed to explicitly declaring how the function is meant to be calculated. When designing such artificial neural networks, a great deal of effort goes into choosing the parameters of such a network, with much trial and error often involved due to the ever increasing complexity of modern networks. This paper presents how genetic algorithms, a class of biologically inspired optimisation algorithms, can be used in combination with neural networks to help with such choices, as well as the effectiveness and the limitations of some of these combinations.

Keywords

Neural Networks, Genetic Algorithms, Hyperparameter Optimisation, Neuroevolution

1. INTRODUCTION

In the field of artificial intelligence, many concepts repeatedly go through popular and less popular periods. Neural networks are a prime example of this: based upon a reasonably simple basic concept that has in the meantime existed for over 70 years, they are once again receiving a heightened amount of attention from the current researching commmunity. This is, amongst other reasons, due to the success of so called deep neural networks, which are typically highly complex neural networks capable of solving equally complex problems [11]. Because of this complexity, ever more effort has to go into configuring these systems for them to perform well. This is a problem, as the exact effect of modifications to certain configuration parameters to the system's performance are often unknown, requiring much manual trial and error before any sort of success can be declared [11].

One way of solving this problem is to automate the choice of such parameters. Using genetic algorithms, a subclass of the optimisation algorithm class of evolutionary algorithms, is one method for which this can be done in a reasonably efficient manner, at least for some of the parameters [11, 8]. In the field of neuroevolution, networks are trained using genetic algorithms, as opposed to the more traditional method of gradient descent using backpropagation, while some approaches additionally modify the structure during the training process [13]. In the following, we shall introduce how some such applications of genetic algorithms to neural networks can be made, with the goal of giving the reader a generic idea of the usefulness of such methods and of their limitations.

2. NEURAL NETWORKS

As the name suggests, artificial neural networks (hereafter referred to simply as neural networks) are based upon a model of the large networks of neurons found in mammalian brains (an adult human brain, for example, has approximately 86 billion neurons [1]). The fundamental idea was first introduced in a paper by neurophysiologist Warren McCulloch and Mathematician Walter Pitts written in 1943 on how neurons might work, modeling a simple neural network with electrical circuits.

The main advantage of neural networks is that, much like a human brain, they are able to adapt and learn with given training data. Neural networks possess limitations, of course; although popular media often depicts them differently, neural networks are neither capable of thinking in the same way human brains can, nor are they some miracle of computing that is capable of solving all of the world's hardest computational problems [3]. However, their adaptive learning properties make them very useful for solving many types of problems for which an algorithm, or at least an efficient algorithm, doesn't exist or can't be found, but sufficiently large sets of data are present with which a network can be trained.

The author would like to note at this point that there is much to be found on the topic of neural networks, and that attempting to introduce all variations and their implementations in detail would go beyond the scope of this paper's topic. Consequently, this chapter shall attempt to give a basic introduction by focusing on the most commonly encountered variant, namely feedforward multilayer perceptrons.

2.1 Learning Process

One of the great advantages of neural networks is their ability to learn; in the field of machine learning, two different kinds of learning are generally distinguished: supervised and unsupervised learning [4].

Supervised learning is achieved with a set of training data consisting of pairs (x, d_x), with x being an input to a neural network, and d_x being the the desired output for the given input x. The actual output y of the network is then compared to the desired output d_x, and, based upon that, the network's parameters are adjusted.

doi: 10.2313/NET-2018-03-1_08

Unsupervised learning works differently: the training data lacks the desired results (also known as labels) d_x. Instead of learning to give a specific output for a given input, the network instead forms internal representations of the input data, encoding certain features of the input based upon the learning rule used by the network.

2.2 Application Example Digit Classification

In order to give an idea of what kind of problems are well suited to having neural networks applied to them, we shall consider the problem of digit classification: presented with a series of handwritten digits between 0 and 9, a human will usually have no problem recognizing them, more or less independent of the actual handwriting. When attempting to conceive an algorithm to perform this task, it doesn't take long to realise that it is hard to transform simple concepts that humans use such as "a 1 is more or less shaped like a line", "a 0 has an ellipse shape" or "a 9 looks like a circle with a squiggly bit at the bottom" into code. Assuming one has a 16 pixel by 16 pixel image, one would have to consider 256 pixels individually, their relationship to one another, allow for tolerances, and at the end still have a result that is vastly complicated to debug and may well only work properly for certain sets of handwriting that are considered in the algorithm's implication.

When presented with the same problem, a simple neural network requiring minimal implementation effort, on the other hand, can, given enough training data, recognize digits with an accuracy of around 95% [2], almost equally independent from handwriting as a human (assuming sufficient different sets of handwriting are present in the training data).

2.3 Artificial Neurons

Neural networks are composed of artificial neurons, with the number and their exact structure in a network varying depending on application and implementation. There are several different types of neurons that can be used, the simplest type being the perceptron. This artificial neuron was developed by Frank Rosenblatt in the 1950s and 1960s, and is based upon the artificial neuron model introduced by McCulloch and Pitts in 1943[4].

For the neurons we shall be considering, a neuron can be described as having a bias $b \in \mathbb{Z}$, $n \in \mathbb{N}$ binary inputs x_1, x_2, ..., x_n, each with a corresponding weight $w_i \in \mathbb{Z}$, and producing an output y. The output is determined through summation of the inputs and the bias, and passing this sum through an activation function ϕ. For perceptrons, the output y is thus determined in the following manner:

$$y = \begin{cases} 1, & \text{if } b + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

An interepretation of this is that the weights of the individual inputs indicate their relevance to this neuron's output, while the bias indicates the neuron's tendency to 'fire': given the same set of weights, a less negative bias will result in a higher likelihood to give an output of 1 than a more negative bias.

Although this neuron is simple and reasonably straightforward to understand, it has disadvantages making it unfavourable for usage in many applications [2], one of those being the lack of continuity in it's output. We shall therefore introduce a second type of neuron, the sigmoid neuron. This neuron gets it's name from the sigmoid function σ used as an activation function:

$$\sigma(z) = \frac{1}{1 + e^{-cz}} \tag{2}$$

with c being a fixed constant. For a sigmoid neuron, the output signal y is now computed in the following way:

$$y = \sigma(b + \sum_{i=1}^{n} w_i \times x_i) \tag{3}$$

While at a first glance this neuron doesn't seem to have too much in common with the perceptron, a closer look at the sigmoid function shows $\lim_{z \to -\infty} \sigma(z) = 0$ and $\lim_{z \to \infty} \sigma(z) = 1$, with the function having a distinctive s-like shape (hence the name). The effect of this is that instead of simply outputting either 0 or 1, a sigmoid neuron outputs values close to 0 for large negative values for z, and values close to 1 for large positive values for z; for such values for z, it mirrors the output of a perceptron, while still allowing for intermediate values. Put differently: while perceptrons are only capable of outputting 'yes' and 'no', sigmoids are capable of outputting different degrees of 'maybe', or instead of only differentiating between, say, a black and white pixel, differentiating between different degrees of grey. This may not always be ideal; in the previously used scenario, the waiter won't typically want to hear the phrase 'I think I might have the steak with a tendency of 0.7'. In such a case, however, it is reasonably simple to decide on an appropriate rule for interpreting the output along the lines of 'anything above 0.5 shall be interpreted as a 1'.

2.4 Network Topology

So far, we have only considered individual neurons, which on their own can model a simple decision making process. We are, however, dealing with entire networks of neurons, not just individual neurons; this means that the inputs of a neuron and, consequently, it's output, depend on the outputs of other neurons, which in turn also may depend on the outputs of still other neurons and so on. This allows for a higher complexity in decision making, giving the possibility to model more complex associations between the inputs and outputs of a network, with neurons further from the 'original' inputs being capable of making more complex and decisions through abstract internal representation of information [2].

The networks that we will consider will contain two restrictions:

- the neurons shall be arranged in layers, with each layer's outputs only being used as inputs for neurons in the next layer

doi: 10.2313/NET-2018-03-1_08

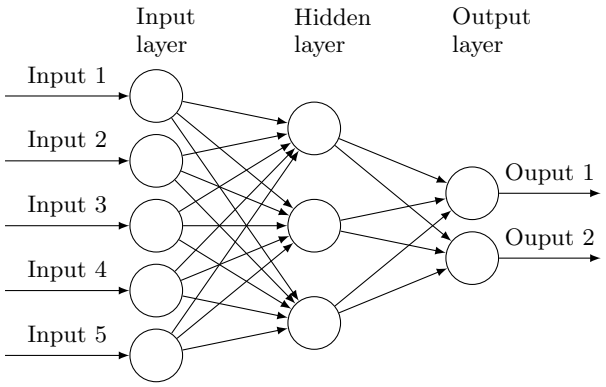

Figure 1: An example neural network architecture

- all neurons in a layer shall receive all of the prior layer's neurons' outputs as inputs

This means, among other things, that there are no connections between neurons within the same layer, and that there are no cycles in our network, making the network a so-called feedforward network; this means that information is only ever passed forward through the network [2]. An example network with these properties can be seen in figure 1.

In this depiction, a circle represents a neuron, while an arrow going from one neuron A to another neuron B is meant to indicate that the output from A is used as an input for neuron B; note that the neurons in the first layer, referred to usually as the input layer, only have one input each, and that these inputs are not outputs from other neurons. This is meant to illustrate that the input layer neurons don't actually do any 'decision making' themselves, but instead have their output values set to fixed values, as a way to encode the inputs to a network. Similarly, the outputs of the neurons in the last layer of neurons, referred to as the output layer, aren't used as inputs to other neurons as they themselves are already the output of the entire network. All layers between the input and output layer are referred to as hidden layers, indicating that when interacting with a neural network, only the input and output layers are 'visible'.

2.5 Gradient Descent

Although there are other methods of learning, in this paper we shall initially focus on one of the most popular and commonly used ones in neural networks that use supervised learning: gradient descent using the backpropagation algorithm.

Previously we have mentioned that the network's parameters are adjusted while learning, based on how much the actual output of the network differs from the expected output. But how exactly should the network's parameters be adjusted?

Before making any adjustments to the network, we need to know how wrong the current output actually is, meaning we need to measure the error in the output. The individual neuron's error e_k for the k-th neuron in the output layer can herefore be defined as $e_k = y_i - d_{x,i}$. Let the output layer consist of m neurons; then, as the networks error function E, we can use the mean squared error function:

$$E = \frac{1}{2}\sum_{i=1}^{m} e_i^2 \tag{4}$$

The goal of learning is now to minimize the average output of the error function E for all inputs; the approach used in gradient descent is to reach this goal by using the error function's gradient to modify the network's parameters, effectively 'descending' to a local minimum of the error function [2].

Before going into how the gradient is defined, we shall consider how we would like to make adjustments to the network in order to reduce the error output: it is fairly intuitive that making a small modification to a bias or weight in some neuron in the network should ideally lead to a small, predictable change in the output. Here, a disadvantage of using perceptrons in a multilayer network become evident: making a small adjustment to a weight or a bias in a neuron can potentially 'flip' this neuron's output, which can lead to a knock-on effect on subsequent layers that overall has a possibly large and unpredictable effect on the ouptut. Sigmoid neurons don't have this disadvantage: it can be shown [2] that when making a small change Δw_{ij} to the j-th weight of the i-th non-input neuron and Δb_i to the bias of the i-th non-input neuron in the network, the change ΔE to the error of the total network is well approximated by:

$$\Delta E \approx \sum_{i=1}^{n} \frac{\delta E}{\delta b_i}\Delta b + \sum_{j=1}^{m_i} \frac{\delta E}{\delta w_{ij}}\Delta w_{ij} \tag{5}$$

With m_i being the amount of weights for the i-th neuron and $\delta E/\delta w_i j$ and $\delta E/\delta b$ being partial derivatives of the output regarding the weight $w_i j$ and the bias b_i, respectively. Expressed in less mathematical terms, this means that the error will only change by a small amount depending upon how 'sensitive' the network's output is to change in those parameters.

The gradient of the error function is now defined as follows:

$$\nabla E = (\frac{\delta E}{\delta b_1}, ..., \frac{\delta E}{\delta b_n}, \frac{\delta E}{\delta w_{11}}, ..., \frac{\delta E}{\delta w_{nm_n}}) \tag{6}$$

If we now define the adjustments to network's parameters as a vector of all of the individual adjustements to the network's biases and weights $\Delta p = (\Delta b_1, ..., \Delta b_n, \Delta w_{11}, ..., \Delta w_{nm_n})^T$, then we can rewrite equation 5 as:

$$\Delta E \approx \nabla E \Delta p \tag{7}$$

As we want to choose a change to the network Δp in order

doi: 10.2313/NET-2018-03-1_08

to minimize the error, we choose Δp in the following way:

$$\Delta p = -\eta \nabla E^T \tag{8}$$

We make this choice because, inserted in equation 7, this results in:

$$\Delta E \approx \nabla E \times (-\eta \nabla E^T) = -\eta ||\nabla E||^2 \tag{9}$$

As $||\nabla E||^2 > 0$, this ensures that the error function will always decrease when applying the change Δp to the network as defined above, at least within the limits of the approximation used in equation 7.

There are now three approaches for making adjustments to the network based on this method, referred to as batch learning, on-line learning and mini-batch learning.

With batch learning, we calculate the gradient ∇E for each input-output pair in the training data set and, using this, calculate the change Δp that is to be made to the parameters in the network. Then, we compute the average over all of these adjustments and finally apply these changes.

With on-line learning, we calculate the gradient ∇E and the adjustment Δp for each input-output pair individually, apply the change, and calculate the next gradient and adjustment based upon the modified network. An advantage of this method in comparison to batch learning is that it isn't necessary to store all the Δps before calculating the average; unlike with batch learning, however, it isn't possible to process the training examples in parallel, as each input-output pair of the training data is processed by a modified network based upon prior modifications.

Mini-batch learning is the compromise of the aforementioned two methods: the training data is split into equal sized 'mini batches'; for each of these mini-batches, we apply the same method used for the batch learning method. As with on-line learning, the adjustments to the network are made directly after each mini batch, with subsequent mini batches being processed by the modified network.

2.6 The Backpropagation Algorithm

So far, it has not been mentioned how the gradient ∇E required to compute the changes made to the network in the gradient descent algorithm is calculated; calculating the gradient function analytically alone for several hundred parameters, which isn't that much when compared to the dimensions found in some neural networks, isn't an option. The backpropagation algorithm presents an efficient alternative, which is why it is used for calculating the gradient required for the gradient descent algorithm.

Before continuing with the algorithm itself, however, we shall briefly introduce a new notation to make it easier to refer to individual neurons and their parameters in the network in an unambiguous way: let w_{kj}^l be the weight of the k-th neuron in the l-th layer for the output of the j-th neuron in the $(l-1)$-th layer, z_k^l the sum of the weighted inputs and the bias for the k-th neuron in the l-th layer. Analogously, b_k^l and y_k^l are defined as the bias and the output of the k-th neuron in the l-th layer.

Additionaly, we now define δ_k^l as the error function's local gradient for the k-th neuron in the l-th row:

$$\delta_k^l = \frac{\delta E}{\delta z_k^l} \tag{10}$$

In other words, a neuron's local gradient δ_k^l tells us how a change in the sum z_k^l of the neuron's inputs will affect the total error. The gradient of the error function, it can be shown, can be calculated with the help of these local gradients [2]. Specifically:

$$\frac{\delta E}{\delta w_{ik}^l} = \sigma(z_k^{l-1})\delta_j^l \tag{11}$$

and

$$\frac{\delta E}{\delta b_j^l} = \delta_j^l \tag{12}$$

The algorithm now consists of two phases, or passes: the forward pass and the backward pass. In the forward pass, we simply pass an input through the network, storing the intermediary outputs y_j^l of all neurons in the process for later use. In the second phase, we calculate the error function's local gradients δ_k^L for all neurons in the output layer, and propagate the local gradients backward throughout the network, using the local gradient $delta_k^l$ to calculate (hence the algorithm's name).

For the output layer L, the local gradients are given as:

$$\delta_j^L = \frac{\delta E}{\delta y_j^L}\sigma'(z_k^L) \tag{13}$$

Note that given our cost function $E = \frac{1}{2}\sum(y_j^L - d_{x,j})^2$, the partial derivative $\frac{\delta E}{\delta y_j^L}$ equals $(y_j^L - d_{x,i})$. Additionally, the sigma function has the practical property that $\sigma'(z_k^L) = a \times \sigma(z_k^L) \times (1 - \sigma(z_k^L)) = a \times y_k^L \times (y_k^L - 1)$, which is also one of the main reasons for it often being used as an activation function in neural networks. This allows us to rephrase equation 13 as:

$$\delta_j^L = a \times (y_j^L - d_{x,i}) \times y_j^L \times (1 - y_j^L) \tag{14}$$

As all of these components are given, we evidently have no problem calculating the local gradients for the output layer. As described above, we are now able to calculate the local gradients in the previous layer using the local gradients of the output layer. This is the formula used herefore [2]:

doi: 10.2313/NET-2018-03-1_08

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{15}$$

Once again, thanks to the aformentioned quality of the sigmoid function, this can be simplified to only contain precalculated values:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} y_k^l \times (1 - y_k^l) \tag{16}$$

This process can now be repeated layer for layer until the local gradients have been calculated for all neurons in the network, allowing us to accurately calculate the gradient ∇E using equations 11 and 12.

3. GENETIC ALGORITHMS

Much as neural networks are based on a simplified model of how mammalian brains work, genetic algorithms are loosely based on concepts used in evolution. They are a type of optimisation algorithm, meaning that they attempt to find an optimal solution to a specific problem; this is done by modifying numerical parameters of individual solutions.

The name and the terminology used for this type of algorithm reflects it's fundamental idea's biological origin, with the main components of virtually every genetic algorithm being a way of encoding the candidate solutions as chromosomes, a fitness function for measuring the quality of such a chromosome, a selection operator and a crossover operator.

3.1 Basic Procedure

We start out with a randomized set of candidate solutions; each of these solutions is referred to as a chromosome, as has been mentioned, and the entire set is referred to as the population, with the initial population being referred to as the first generation.

In each generation, members of the population are chosen and combined in the attempt to 'breed' chromosomes with a higher fitness score; this process used to combine the chromosomes is referred to as crossover, with it's implementation varying depending on how the chromosome encodes the candidate solution and on the application of the algorithm. After a certain number of new chromosomes have been created in this way, a subset of them (usually those scoring the highest with the fitness function) replace an equal sized portion of the current population (often those scoring low with the fitness function).

This process is then repeated generation for generation, with the population staying a constant size, until either a certain number of generations have passed, or a chromosome scores higher than a predefined value for the fitness function; the algorithm then returns the fittest chromosome as the optimal solution [5].

3.2 Chromosome Encoding

A central question when applying a genetic algorithm to a problem is how to encode the candidate solution as a chromosome, as this has a direct impact on the crossover operation. Principally, a chromosome hasn't got any practical restrictions on how it can be encoded: one approach, for example, is to encode the candidate solution as an array of strings, while another might be to store it as a series of integer values or bits. As with all aspects of genetic algorithms, the exact implementation depends on the problem and the space of candidate solutions.

1	0	1	0	1	1	0	0	1	0

Figure 2: An example chromosome using binary encoding

A simple example would be encoding candidate solutions for the well known knapsack problem: one is presented with a selection of $n \in \mathbb{N}$ items, each with a weight w_i and a value v_i, and at one's disposal is a bag that can handle a certain maximum weight w_max. The problem now is to find a combination of items that maximize the value that one can carry in the bag.

In this case, a candidate solution could be encoded as a vector of bits c, with a 1 at the i-th position meaning that the i-th item should be included in the bag, and a 0 meaning it shouldn't. A chromosome encoding the candidate solution of including the 1st, 3rd, 5th, 6th and 9th item out of 10 items can be seen in figure 2.

An appropriate fitness function f could therefore be defined as:

$$f(c) = \begin{cases} v^T c & \text{if } w^T c \leq w_{max} \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

Note that we could remove the condition in the fitness function if we didn't allow chromosomes representing items with a total weight larger than w_{max} to exist in the first place; as this is just meant to be an example, however, this way of defining the fitness function is perfectly sufficient.

3.3 Crossover and Mutation

Crossover operations can be as varied as the different possibilities for encoding the chromosomes, and their exact workings will be closely linked to the chromosomes' form. To gain some insight into how such operators may work, we shall consider two reasonably simple methods: single-point and multi-point crossover.

For the single-point crossover operation, we select a single point in the chromosome's encoding, and generate the child by using the parameters of the first parent chromosome until that point, and using the parameters of the second parent from that point onward. To return to our previously used example of the binary encoding of the knapsack problem, this can be seen in figure 3.

The mullti-point crossover operation is defined similarly, but instead of using one point, we use several instead; once again

doi: 10.2313/NET-2018-03-1_08

Parent 1:	1	0	1	0	1	1	0	0	1	0
Parent 2:	1	1	0	0	1	0	0	1	1	1
⇓ Crossover ⇓										
Child 1::	1	0	1	0	1	0	0	1	1	1
Child 2:	1	1	0	0	1	1	0	0	1	0

Figure 3: An example of single-point crossover

Parent 1:	1	0	1	0	1	1	0	0	1	0
Parent 2:	1	1	0	0	1	0	0	1	1	1
⇓ Crossover ⇓										
Child 1:	1	0	0	0	1	1	0	1	1	0
Child 2:	1	1	1	0	1	0	0	0	1	1

Figure 4: An example of multi-point crossover

using the knapsack example, a multi-point crossover with 4 points is visualized in figure 4.

Usually, after the child chromosome has been generated, there is a chance of of the new chromosome 'mutating' new properties; this involves randomly changing one or more parameters of the chromosome, within the limits of the solution space. For a bit-sequence chromosome like the one in our example, this could mean randomly selecting one or more bits and flipping them. This element of randomness is meant to add 'genetic diversity' to the population so that the algorithm doesn't get stuck in non-optimal local maxima of the fitness function as easily [5].

3.4 Selection

In order to perform the crossover operation, we have to first select two parents; for this, we have to define a selection operator that selects two parents from our population. Ideally, we would like to 'breed' favourable qualities for our next generation of chromosomes, which are usually contained in the chromosomes with the best fitness scores. One way of achieving this is to define a selection operator that selects population members randomly, with the probability $P(c_i)$ of each chromosome c_i of a population of size n being picked being defined as:

$$P(c_i) = \frac{f(c_i)}{\sum_{j=1}^{n} f(c_j)} \tag{18}$$

Effectively, this means that chromosomes with higher fitness score will be preferred during selection, ensuring that favourable qualities are more likely to be contained in children created by the crossover operation.

Towards the end of the algorithm's run, however, the fitness values will only vary slightly, meaning that all members of a population have roughly equal chances of being selected; also, it has the disadvantage is that it only works for fitness functions for which higher outputs are defined as being better.

An alternative approach that avoids these restrictions may instead work by selecting chromosomes based upon the rank of their fitness score among the population, instead of based on their relative fitness score [6]. For a population of size n, the chromosome c_i with rank $i \in [1; n]$ regarding it's fitness score, the probability $P(c_i)$ could then be defined in the following way:

$$P(c_i) = \frac{2(n - i + 1)}{n(n + 1)} \tag{19}$$

4. EVOLVING NEURAL NETWORKS

Normally when designing a neural network, the implementer has to make a series of design choices: these could include, for example, the network's topology, the backpropagation algorithm's learning rate and the activation function used by the neurons in the network; these properties are sometimes referred to as a network's hyperparameters[2].

Although there are empirical results and heuristics that give some general rules of thumb that aid in those decisions, their exact effect on performance is generally not so well understood; this means that finding hyperparameters that work well usually involves some amount of trial and error. An alternative approach to manual selection is to use some form of optimisation algorithm to determine some of these parameters; thus, roughly since the 1980s, a good deal of effort has gone into combining genetic algorithms with neural networks for this purpose [6].

There are primarily three ways in which genetic algorithms have successfully been combined with neural networks: evolving the weights of a network with fixed hyperparameters, evolving a networks hyperparameters, and evolving a network's topology alongside it's weights [10]. In this paper, we shall consider the latter two.

4.1 Encoding Networks

Before getting to the actual ways in which we can use genetic algorithms to evolve neural networks, it is relevant to bring to the reader's attention some problems that arise when attempting to actually implement this.

As has been mentioned already, for genetic algorithms, a candidate solution's parameters have to be encoded in a chromsome. One problem one encounters when encoding a neural network is recurred to as the competing conventions problem: in essence, several networks that are functionally identical can be structurally different [7]. As a simple example, assume we naively encode the parameters $params_i$ of n of n individual neurons in a network sequentially:

$params_1$	$params_2$	...	$params_n$

We shall assume for this example that the parameters include the weights, the biases and the layer of each neuron, and that the neurons are sorted incrementally by their layer, meaning that the input layer neurons are encoded at the beginning and the output layer neurons are encoded at the end. If we consider a simple network with only one hidden layer with m neurons using this form of encoding, then there

doi: 10.2313/NET-2018-03-1_08

are $m!$ possible permutations for the hidden layer and, consequently, $m!$ different representations for functionally the same network.

This may not be seem to be an immediate problem, and there is in fact research that suggests that it may not have a significant impact on the genetic algorithm [6]. On the other hand, crossovers of functionally similar, but structurally different parent networks tend to result in impaired children, which needlessly cost the algorithm additional computation time [7].

Another complication occurs with larger networks: the solution space that the genetic algorithms have to search tend to simply have too many dimensions in which optimisation is possible [6]. When focusing merely on the hyperparameters of the network, however, it has been shown that even for more complex so-called deep neural networks, which use many hidden layers in their topology, it is possible to autonomously evolve state of the art hyperparameters using genetic algorithms[11].

4.2 Hyperparameter Optimisation

One way of using a genetic algorithm to select hyperparameters for a neural network lies in encoding merely a network's hyperparameters as a chromosome, with the networks' weights and biases not being encoded in the chromosomes; this is referred to as Baldwin learning [6]. The exact hyperparameters that are included may vary strongly depending on the implementation: using feedforward neural networks as described previously in this paper, this could only involve encoding the number of hidden layers used and the amount of neurons in each of these layers, with some fixed maximum amount of hidden layers [10], or additionally include the activation function, the learning rate η of the backpropagation algorithm and even the learning algorithm used [9].

As part of the evaluation by the fitness function, we first initialize a network with the encoded hyperparameters, with randomized starting weights and biases, and then train it. After finishing it's training, we test it's performance on a set of test data that wasn't used during training, using, for example, the mean square error of all outputs of the test data as the output of the fitness function.

Unfortunately, the process of training just a single generation, which can have tens or even hundreds of chromosomes, is very computationally expensive, as even training a single network with backpropagation can potentially take days. As a sufficiently well performing set of hyperparameters can theoretically take many generations to evolve; this means that even with a high degree of parallelization, e.g. training all chromosomes of a generation simultaneously, this process can take a large amount of time.

Interestingly, the choice of the amount of training that a network undergoes before evaluation has been shown to have a direct impact on the genetic algorithm's evolution rate [8], as well as how fast networks that are produced in this manner tend to learn [11].

Overall it can be summarized that through applying appropriate limitations to the search space of the genetic algorithms, such methods should be a very effective alternative to manually selecting the hyperparameters of a network, even if the computational cost may still often be significant.

4.3 Neuroevolution

Neuroevolution, as the name implies, is a description for methods in which evolutionary algorithms, of which genetic algorithms are a subtype, are used to train neural networks. When considering such methods, we differentiate between fixed topology systems, and so-called Topology and Weight Evolving Artificial Neural Networks, or TWEANNS [13]. Evolving the network's weights has the advantage that it isn't necessary to calculate the error function gradient to make adjustments (as opposed to gradient descent); this makes neuroevolution a good alternative to using gradient descent when the gradient is either very expensive, or not possible to calculate for a given problem [12]. On the other hand, as was already mentioned, the fact that all of the network's parameters are encoded in the genetic algorithm's chromosome means that training networks this way isn't really an option for larger networks [6].

When evolving the topology alongside the weights, an additional problem arises: when a chromosome evolves with an adjusted structure, this adjustment will often initially have a negative impact. As an example: by adding a neuron with randomly initiated weights at a critical point in the network, we could completely change the way the network functions, potentially 'ruining' a network that performed comparatively well beforehand. This will lead to a lower fitness score, which may well in turn lead to the new network structure disappearing from the population's 'gene pool', even if the new structure might perform far better than the old structure after correctly evolving the appropriate weights. This problem can usually be addressed by having networks only compete with similarly-structured networks, referred to as speciation [13]. This, on the other hand, means that a computationally affordable way of categorizing two networks as having a 'similar structure' in this context needs to be provided [7].

As an example of a TWEANN, we can consider the NeuroEvolution of Augmenting Topologies algorithm, referred to as NEAT [13]. The approach used attempts to keep the dimensionality of the algorithm's search space minimal by starting with networks of minimal structural complexity (i.e. only an input and output layer), and gradually incrementing the structure; mutations, it is worth noting, are only capable of adding structure, not removing it. Specifically, the first generation of chromosomes only consists of networks with an input and an output layer, and additional structure is evolved by adding neurons and connections between neurons through mutation; this technique is referred to as complexification [13]. When compared to methods that involve initial populations of random structure, complexification tends to result in neural networks with a comparatively minimal structure. This is favourable as a smaller structure means less computational effort to calculate the output for a given input, and, depending on the network encoding scheme, less storage space required for the network. Additionally, due to the structure being kept minimal, the dimensionality of the search space of the genetic algorithm is also relatively low.

doi: 10.2313/NET-2018-03-1_08

This benefits the performance of the algorithm [7]. Note that due to the way that structure is added, depending on the actual implementation, feedforward networks aren't the only type of network that can be evolved in this way as the network can also include loops.

In addition to complexification, NEAT features an encoding scheme that allows an efficient structural comparison between networks, enabling the protection of innovative structure through the aforementioned method of speciation. Chromosomes consist of node genes and connection genes, each of which is assigned a unique historical marker when evolved through mutation, with genes passed on to offspring through crossover using the same marker. Concerning the protection of topological innovations through speciation, NEAT uses a method referred to as explicit fitness sharing. By checking which genes overlap in two different chromosomes i and j, we can assign a distance $\delta(i,j)$ to these two networks, with a higher distance meaning a greater structural difference. After evaluating the fitness f_i of a chromosome i in a population of n chromosomes, it is assigned it's modified fitness f_i' in the following way:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{20}$$

With the share function sh being defined as a step function that outputs 0 if the structural distance between i and j is above some specified limit, and 1 otherwise. An output of 1 is thus to be interpreted as chromosome i and j belonging to the same species; if there are many chromosomes in a species, then they're modified fitness f' will be comparatively lower, while the reverse is true for species with few chromosomes. This means that new species that emerge, whether through mutation or crossover, will be assigned a higher fitness score if they structurally differ from other species sufficiently. Using this modified fitness function, each species is implicitly allocated a slot, or evolutionary niche, in the population based upon their fitness, with the weakest members of each species having high chances of being eliminated in every generation. [13].

Neuroevolutionary methods, as a whole, can be seen as a good alternative to gradient descent for many problems for which calculating the error function gradient isn't an option; also, regardless of whether the topology is fixed or not, due to the mutative qualities of genetic algorithms, it is less likely that the optimisation process will stall in a suboptimal local performance maximum [13]. Also, due to the nature of genetic algorithms, much of the computationally most expensive phase, namely that of the fitness evaluation, can be performed in parallel, enabling a great increase on performance [7]. TWEANNS carry with them the additional advantage that they relieve the programmer implementing the algorithm of having to make exact choices concerning the network's topology. The main disadvantage of using neuroevolution is the limit in the size of the networks: if the networks become too complex, then evolutionary algorithms will struggle to find fitting parameters due to the high dimensionality of the search space [6].

5. CONCLUSION

We have seen that neural networks and genetic algorithms are two concepts with biological backgrounds that synergize well for many problems: as long as the dimensionality of the search space isn't too big, genetic algorithms are capable of finding good parameters for many applications of neural networks, be it the hyperparameters of the network, the parameters of the individual neurons, or both. Due to the amount of attention that neural networks are receiving of late, and with the diversity of methods in neuroevolution and hyperparameter optimisation using genetic algorithms, future advances in this field don't seem unlikely, with new methods possibly surpassing some of the limitations that are present today.

6. REFERENCES

[1] F.A. Azevedo, L.R. Carvalho, L.T. Grinberg, et al.: *"Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain"*, The Journal of Comparative Neurology, 513 (5), pages 532-41, 2009

[2] Michael A. Nielsen: *Neural Networks and Deep Learning*, Determination Press, 2015

[3] Alan F. Murray: *Applications of Neural Networks*, pages 1-4, Springer, Boston, MA, USA, 1995

[4] Simon Haykins: *Neural Networks - A Comprehensive Foundation*, Pearson Education, Inc., 3rd edition, pages 65-69 and 157-172, 2009

[5] Jenna Carr: *An Introduction to Genetic Algorithms*, 2014

[6] Phillip Koehn: *"Combining Genetic Algorithms and Neural Networks: The Encoding Problem*, University of Tennesy, Knocksville, 1994

[7] William T. Kearney: *"Using Genetic Algorithms to Evolve Artificial Neural Networks"*, Colby College, 2016

[8] R. Keesing and D.G. Stork: *"Evolution and Learning in Neural Networks: the Number and Distribution of Learning Trials Affect the Rate of Evolution"*, Proceedings of the 6. Neural Information Processing Systems Conference, pages 804-810, 1993

[9] O.A. Abdalla, A.O. Elfaki, Y.M. AlMurtadha: *"Optimizing the Multilayer Feed-Forward Artificial Neural Networks Architecture and Training Parameters using Genetic Algorithm"*, International Journal of Computer Applications (0975-8887)2014

[10] Christopher M. Taylor: *"Selecting Neural Network Topologies: A Hybrid Approach Combining Genetic Algorithms and Neural Networks"*, Southwest Missouri State University, 1997

[11] Risto Miikkulainen et. al: *"Evolving Deep Neural Networks"*, Sentient Technologies, Inc. / The University of Texas at Austin, 2017

[12] A.J. Turner, J.F. Miller: *"The Importance of Topology Evolution in NeuroEvolution: A Case Study Using Cartesian Genetic Programming of Artificial Neural Networks"*, Research and Development in Intelligent Systems XXX, pages 213-226, Springer International Publishing Switzerland, 2013

[13] K.O. Stanley: *"Efficient Evolution of Neural Networks through Complexification"*, pages 7-41, The University of Texas at Austin, 2004