**Autor:** Aiordăchioaei Marius | **Assistant:** Suciu Mihai

# Table of contents

1. Intro
   1. Abstract
   2. Motivation
2. Core concepts
   1. Genetic Programming
   2. Artificial Neural Networks
   3. Integration
   4. Classical Problems
3. Aplicability
4. Feasability and Impact Analysis
   1. Metrics and testing
   2. Observations
5. References

# Intro

## Abstract

With the Artificial Neural Networks becoming more and more popular and used across all domains, a common problem with designing an eficient one is the tuning of the so-called "Hyperparameters".

An empirical aproach that addresses this problem is through manual trial and error. This aproach is (most of the time) inefficient and over the hand in almost all the possible ways. Another aproach is doing R&D (Research and Development) on other scientifical publications and finding problems simmilar to ours' that have been solved.

The aim of this paper is to demonstrate the feasibility of using a third aproach, automating the process of tweaking the parameters using methods such as Evolutionary Algorithms to find the right Neural Network, fit for a specific domain problem.

The process of finding the right Evolutionary Algorithm was all trial and error, with each step documented. Based on some ad-hoc metrics, the algorithm could be brought to a state in which is better for the specific task that are performing.

Keywords:

Evolutionary algorithm, genetic algorithm, genetic programming, artificial neural network, hyperparameter, brute-force, fitness, natural selection

## Motivation

This concept is nothing new. It was researched before and proved to be efficient, and the idea of automating a trial-and-error job can cross anyone's mind at some point.

The work will focus more on the Evolutionary part rather than on the Neural Network part. It will also contain a detailed feasibility study of the found solution and testing it with some classical Machine Learning problems

# Core concepts

In the following lines we are going to explain the two concepts/techniques used in our case study, as well as the integration between the two

## Genetic programming

Genetic programming and Evolutionary Algorithms refer to a technique of solving problems which solutions are in a wide space of possible solutions. Imagine a solution space which grows exponentially-proportional with the size of the input data.

[4] Classical deterministic algorithms are a possible solution to this. Any deterministic algorithm is presumed to do the transition between the current state and the following one in a predictable (or deterministic) manner. Deterministic algorithms will solve this kinds of problems, but the issue is that their time complexity is unreasonably high, or even exponentially. So for large input sizes, these sort of algorithms will not scale (in terms of time or space efficiency) on conventional semiconductor machines.

For solving our problem, we will focus on intelligent algorithms.

Genetic programming is a method of finding solution inspired from nature, more exactly from the real evolution. [3] The basic idea of having individuals in population that can mix their genes and get evaluated based on certain characteristics, seems like a good solution for our problem. The brain itself was created in the same manner.

The very general picture of the concept is expressed in the following pseudo-code:

Begin  
 Initialize population with random individuals  
 Evaluate each individual from the initial population  
 While ( Certain stop condition )  
 Select and crossover parents;  
 Mutate the resulting sons;  
 Evaluate the new individuals;  
 Select survivors  
 End\_While  
End

A very important characteristic to some of the above-mentioned operations (natural selection, mutation) is that they are stochastic which means that in their way of determining the result, they make choices based on randomness or chance. So we know for sure that EAs don't fall under the deterministic algorithm category.

In our case study, the individuals are considered to be the Neural Networks that we are trying to evolve.

[3] Components of Evolutionary Algorithms

In the following line we are going to describe the basic components which make up an EA.

* Representation (how the individuals are encoded in our algorithm)
* Fitness measurement method
* Population
* Parent selection strategy
* Crossover
* Mutation
* Replacement mechanism

#### Representation

Usually the first step in implementing our EA is choosing a representation. We need to map or translate the objects in the real world space to our defined solution space. The terminology for the objects in the real world space is **phenotypes** and the one for the objects in our solution space (EA space) is **genotypes**.

As an example, consider the following problem: Given an undirected graph, find the path with the most nodes. This problem is part of the NP-Complete problem class (which we are not going to prove now). The individual in our EA is the path (a sequence of linked nodes). The phenotype is the path itself. One genotype can be a binary number which has 1 on pozition j for j node is in sequence, 0 otherwise (even though this genotype doesn't consider the order of the nodes, but this might not be a concern in our algorithm). Another genotype can be a simple list of ordered nodes.

#### Fitness

The role of the fitness function is to objectively measure how close is one individual to the desired solution. Based on this metric we can decide if one individual is fit enough to continue in our population or is not fit enough and needs to be replaced by better individuals (natural selection)

#### Population

The population is the set of individuals at some point (any point) in our EA. It can be seen as a set of genotypes of many generations. The genotypes themselves is not evolving in any way, but the population is, through the creation of the new and more fit genotypes.

An important metric here is the diversity of the population which measures how many different solution we have. To maintain diversity we need to give weaker genotypes a chance to reproduce, because they might have valuable genetic material which combined with the right counterparty, might give us a fit individual. A population which is not diverse, tends to converge with the best fitness and might never give us a fit enough individual. Elitism in a EA (often choosing the best) can damage the diversity. This is the equivalent of a greedy algorithm which at each given step, it chooses only the local optimum.

#### Parent selection

The parent selection mechanism is the rule by which we allow individuals to bring offsprings into the population by combining their genes (becoming parents). In this part we get to choose parents, mostly based on their quality (fitness) but also based on some stochastic elements.

The reason that we are introducing stochastic elements here, is that we want to avoid a homogenous population by giving the unfit individuals the chance to become parents so we can improve the diversity discussed above.

#### Crossover

The crossover operation has two genotypes as input and one or two offspring(s) as an output. By merging the genes of the input individuals, we are able to create the offspring which inherits traits from the parents and probabilistically has a higher fitness.

In the designing of our EA, it is crucial to choose a good crossover mechanism.

#### Mutation

Mutation naturally happens in the real world. But in EAs are a method of keeping the population diverse. When is born through crossover, the genotype has a pre-defined chance to become mutated. It means that the features inherited from the parents are slightly changed.

#### Replacement

Replacement distinguishes individual based on their fitness and age (with stochastic influences). It is used to eliminate the individuals that haven't proved themselves to have any value to the evolution process, or just the genotypes that are unfortunately chosen by the stochastic elements (which translates into environmental hazard). This is mostly one in the same step as the Parent selection part, when the individuals that are not selected for reproduction, are disposed from the population.

## Artificial Neural Networks

Artificial Neural Networks (ANN) is an approach to the Supervised Learning problems that are more and more used in multiple industries. The method inspires heavily from how our brain works, even though it is an overly-simplified process.

### Seen as Black-Box

[2] An ANN, seen as a black-box entity, is a feed-forward model consisting of a layer-based architecture. There are 3 types of layers: input layer, hidden layers and output layer. Feed-forward means that the computation happens in the order of the layers, while the error is propagated backwards. So here we have to deal with a layered architecture in which each layer is composed of neurons.

General idea of ANN As observed in the picture above, each neuron from one layer linked to all the neurons in the layer after (even though, is not really a constraint to be linked to all neurons). This is called a dense layer.

The supervised learning part means that we supervise the learning process by giving feedback to the performance of our ANN for each test. We tell to our ANN when is wrong, and more importantly, how wrong is it, so it can learn not to output the same wrong value again (through methods that we are going to describe later).

So, in our learning phase of the ANN, we are going to need training data, which consists of input values, and output values that are considered to be correct. Our ANN is going to learn to handle new inputs, and output values that are the desired ones, very much like how the human brain learns.

The quality of the output relies heavily on many factors, like:

* The architecture of the ANN used. (how many hidden layers we use, the size of each layer, activation functions, etc.)
* The quality of the training data. (training data needs to explore enough of the solution space if we want correct outputs. If not, we will deal with the phenomenon known as overfitting, which in Layman's terms means that our ANN will learn the training inputs "by heart" and not be able to handle new particular inputs properly)

### [2] Seen as White-Box

The state of an ANN consists of some values called weights and biases. Learning process comes down to finding the right values for the weights and biases.

Neuron For each neuron, we have its input values from the previous layer (in case of the input layer, we will have only one single input value). We have weight values (noted as w) for each link to the previous layer. It represents the importance we give to the input of the predecessor neuron.

Altogether, the value outputted by our neuron will be

Where b is the bias

Dependending on the type of the neutron, the output can be projected through different mathematical functions, such as sigmoid, softmax and linear rectifier. Such function is called activation function and also has impact on the overall performance of the ANN.

The adjustment of the weight and bias values is done in the learning phase, through a method called error back propagation. First, it is calculated how far (in distance) is the output from the actual solution, and then the error is propagated back to the layers it came from. The weights and biases are recalculated in such manner that on the next evaluation with the same input, the outputted error will be smaller (how smaller, it depends on how fast the network learns. Learning rate is tweaked in order to prevent overfitting).

For the error back propagation, a very used method is gradient descend.

## Integration

The two concepts described above go well hand in hand. We can define our phenotype as an ANN which is fit for having a better accuracy after a training session.

This way we are will be able to find the right ANN architecture for our problem, without much effort.

We need to define our genotype. As mentioned in the abstract, our genotype will consist of the ANN's hyperparameters. A hyperparameter is that value which is configured before the process of learning begins.

The genotype can consider:

* Number of the hidden layers
* Size of each layer
* Type of each layer along with activation function
* Initial weights
* Initial bias

In the current state of the implementation, our genotype will regard only the number of hidden layers and the size of each hidden layer.

## Classical Problems

Here we are going to mention the Machine Learning problem on which we tested our algorithm.

#### [2] Digit Recognition

This is the "hello world" of Machine Learning. In the testing, we used the [MNIST](http://yann.lecun.com/exdb/mnist/) database. It provides 60,000 train instances along with 10,000 test instances. Each train instance is a picture of one of the 10 digits, in a bitmap format. The size of each bitmap is 28x28 pixels and the chromatic range of each pixel is one byte monochrome (so a color value ranges between 0-255).

The way we are feeding this onto our ANN is: we have an input layer of 28\*28 = 784 neurons, one for each pixel. This is a classification problem, so the output layer have one neuron for each class, in our case 10 neurons. Our ANN will evaluate how likely (in terms of chance) is for our input to be classified as each class. So if we sum the outputs of each neuron of the last layer, we need to get 1. To achieve the translation between the 0-255 ranged input values and probabilistic subunitary values we need to set the activation function of the last layer to softmax.

# Applicabillity

As the practical part, I used for my case study different technologies and libraries:

* All the code is implemented in Python.
* For the ANN API I used Keras along with TensorFlow backend and for scalability concerns I used Tensor GPU to make tensor use specialized hardware for the learning workload.
* For plotting certain metrics I logged parts of the algorithm and then extracted the information and plotted it using Matplotlib.

To create a clear architecture of the code, I wrapped all the behaviour of the Keras library in some decorator classes so that I have a clear API which speaks Evolutive Algorithm terminology. For example, I wrapped the ANN which extends "Individual" class in such manner that the ANN is contracted to implement EA operations like crossover or mutate.

## Code architecture

Code architecture

*Code architecture*

# Fesability and Impact Analysis

In the implementation phase, I decided to use an iterative system. In each iteration, the algorithm was tested, observations were made based on some metrics, and improvement ideas were planned for implementation in the next iteration.

## Metrics and testing

For testing the algorithm, we used here the character recognition problem mentioned above, with MNIST database.

## Iteration 1

In the initial implementation, we have the natural selection function, which has as input a list of individuals, sorted descendingly by their measured fitness. For each 2 individuals indexed i and j, we calculate their chance to reproduce:

A selection for replacement was also implemented, which considers a probability, which is proportional to the fitness of the individual, and also the age of the individual (generation count). For each individual i, his chance to be replaced is:

Unde E = life expectancy and n = total number of individuals

### Parameters

Life expectancy: 3

### Crossover

The crossover of the neural nets was done through the combination of the hidden layers.

In the case that the crossover parties have different numbers of layers, a random sampling happens on the net with more layers. The size of the sampling is the number of layers of the one with fewer layers (lets call it A and the counterparty B). The bad thing here is that the genes of A can be proportionally dominant compared to B, as the layers of B, that are not included in the sampling, are discarded in the process.

### Fitness

The calculation of the fitness is (currently) based only on the accuracy of the neural network.

### Metrics

* Fitness trend

Fitness trend

*Fitness trend*

* Generation size (Blue - number of individuals in the population, Green - maximum fitness)

Generation size

*Generation size*

## Iteration 2

### Parameters

Life expectancy: 1.3

### Crossover

Unchanged.

### Fitness

Unchanged.

### Metrics

* Fitness trend

Fitness trend

*Fitness trend*

* Generation size (Blue - number of individuals in the population, Green - maximum fitness, Dark blue - average fitness)

We are now measuring also the average fitness of the population

Fitness trend

*Fitness trend*

## Observations

The main observations made throughout the first two iterations focuses on weak points of the algorithm:

* One major weakness is that the population grows quickly.
* This simply doesn't scale, because finding a good solution to our problem, we need to grow a certain amount of generations, approximately 10. To grow 4 generations, the program ran for about ~12 hours. The conclusion drew by Generation Size metric, this approach doesn't scale. To perform a crossover between the genotypes of the generation 4 would take an unreasonably huge amount of time.
* Another observations here is that the average fitness converges to a value. This means that the genotypes become homogenous. There is not enough new genetic material and this will not help us raise the maximum fitness either.
* We can see how the maximum fitness increases, but not significantly enough to justify the need of such algorithms to find a better solution. A significant fitness increase would be achieved by growing a larger number of generations.
* The crossover function can be also improved. When performing crossover, we can consider all the genes of the individuals, without too much domination from one party.
* We can consider other hyperparameters in finding the optimal solution, like initial weights and biases.
* The fitness function can also consider the physical running time as a quality of a genotype.
* [5] For the following iterations, we will use Tournament Selection for selections. This should give us a controllable number of genotypes and more generations to grow.
* We can consider trying the same test but for other problems (which can also be more complex) and other training data sets.
* The initial genotypes can be generated randomly.
* The input and the train data can be randomly sampled (k-fold)

# References

[1] [Let’s evolve a neural network with a genetic algorithm](https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164)

[2] [Using neural nets to recognize handwritten digits](http://neuralnetworksanddeeplearning.com/chap1.html)

[3] [What is an Evolutionary Algorithm](https://www.cs.vu.nl/~gusz/ecbook/Eiben-Smith-Intro2EC-Ch2.pdf)

[4] [Computational complexity](https://en.wikipedia.org/wiki/Computational\_complexity)

[5] [An Evolutionary Approach for Tuning Artificial Neural Network Parameters](https://pdfs.semanticscholar.org/7d7b/84986459ccc1de349342d44fde999c3de8e9.pdf)