EasyChair Preprint

№ 3149

Artificial Superintelligence : A Model for Self-Improving / Self-Modifying Programs

Poondru Prithvinath Reddy

April 12, 2020

Artificial Superintelligence : A Model for Self-Improving / Self-Modifying Programs

Poondru Prithvinath Reddy

ABSTRACT

Self-Improvement or Self-Modification is any behavior of a system where a program gets better at achieving goals as it receives input. An example of a self-improving program would be a program that gets better at playing chess by playing games against itself. In this paper, we provide a formal definition of self-improvement systems and then we present a self-modification model by two different approaches. The first one is to find an optimal program defined by given scores and program generation probabilities using Markov Chain. The second one describe a method of Applying Genetic Algorithm on Multilayer Artificial Neural Network for updating and optimizing the neural network weights. GA creates multiple solutions and evolves them through a number of generations, and each solution holds all weights in all layers to help achieve higher accuracy. The evolutionary algorithm(i.e.GA) was used as an optimization approach that mimics the concept of natural evolution for creating fitter individuals that have higher chance of survival through natural selection. The GA processes integrated with Artificial Neural Network model and the network improves itself by learning to optimize its own weights. We observe from the test results that the networks were able to self-improve through natural selection with good accuracy and also it is observed that self-modification mechanism for artificial intelligence is convenient.

INTRODUCTION

If research into strong AI produced sufficiently intelligent software, it would be able to reprogram and improve itself. It would then be even better at improving itself, and could continue doing so in a rapidly increasing cycle, leading to a superintelligence. This scenario is known as an intelligence explosion. Such an intelligence would not have the limitations of human intellect, and may be able to invent or discover almost anything.

Thus, the simplest example of a superintelligence may be an emulated human mind that's run on much faster hardware than the brain. A human-like reasoner that could think millions of times faster than current humans would have a dominant advantage in most reasoning tasks, particularly ones that require haste or long strings of actions. This also raises the possibility of collective superintelligence : a large enough number of separate reasoning systems, if they communicated and coordinated well enough, could act in aggregate with far greater capabilities than any sub-agent.

The technological singularity – is a hypothetical future point in time at which technological growth becomes called intelligence explosion, an upgradable intelligent agent (such as a computer running software-based artificial general intelligence) will eventually enter a "runaway reaction" of self-improvement cycles, with each new and more intelligent generation appearing more and more rapidly, causing an "explosion" in intelligence and resulting in a powerful superintelligence that qualitatively far surpasses all human intelligence.

If it is possible for a system to improve itself, for example, for a program to rewrite its own source code to learn faster, or to store more knowledge in a fixed space, without being given any information except its own source code. This is a different problem than learning, where a program gets better at achieving goals as it receives input. An example of a self improving program would be a program that gets better at playing chess by playing games against itself. Another example would be a program with the goal of finding large prime numbers within t steps given t. The program might improve itself by varying its source code and testing whether the changes find larger primes for various t.

Is it possible for a computer program to write its own programs? While this kind of idea seems far-fetched, it may actually be closer than we think. Researchers conducted an experiment to produce an AI program, capable of developing its own programs, using a genetic algorithm implementation with self-modifying and self-improving code. However, artificial intelligence, if programmed, in an attempt to write a functioning program that can, itself, write programs.

METHODOLOGY

Evolution does modify its own source code and does that by manipulating DNA from generation to generation. A genetic algorithm is a type of artificial intelligence, modeled after biological evolution, that begins with no knowledge of the subject, aside from available tools and valid instructions. The AI picks a series of instructions at random (to serve as a piece of DNA) and checks the fitness of the result. It does this with a large population size, of say 100 programs. Surely, some of the programs are better than others. Those that have the best fitness are mated together to produce offspring. Each generation gets a bit of extra diversity from evolutionary techniques such as roulette selection, crossover, and mutation. The process is repeated with each child generation, hopefully producing better and better results, until a target solution is found. Genetic algorithms are programmatic implementations of survival of the fittest. They can also be classified as artificially intelligent search algorithms, with regard to how they search an immense problem space for a specific solution.

The methodology essentially consists of two parts :-

1. To find an optimal program defined by given scores and program generation probabilities using Markov Chain.

2. Applying Genetic Algorithm on Multilayer Artificial Neural Network – The best solution for a self-improving network by using genetic algorithm(**Genetic algorithms have collections of solutions that are collided with each other to make new solutions, eventually returning the best solution.)**

ARCHITECTURE

OPTIMAL PROGRAM FOLLOWING RSI

Recursive Self Improvement : Define an improving sequence with respect to G as an infinite sequence of programs P1, P2, P3,... such that for all i > 0, Pi+1 improves on Pi with respect to goal G and G be the identity goal.

Definition: P1 is a recursively self improving (RSI) program with respect to G if and only if Pi(-1) = Pi+1 for all i > 0 and the sequence Pi, i = 1, 2, 3...is an improving sequence with respect to G.

Definition (RSI system).Given a finite set of programs P and a score function S over P. Initialize p from P to be the system's current program. Repeat until certain criterion satisfied, generate $p' \in P$ using p. If p' is better than p according to S, replace p by p'.

From this definition, one needs to decide how $p \in P$ generates a program. In general, we should allow the RSI system to generate programs based on the history of the entire process. The way a program generates a new program is independent, and each program defines a fixed probabilistic distribution over P. This procedure defines a homogeneous Markov chain. We will see that even with this restriction, with some score function, the model is able to achieve a desirable performance.

We illustrate the proposed formulation by an example. Consider a set of programs P={p1, p2, p3, p4} and a score function S over P such that S(pi) =i. According to our formulation, each program can be abstracted as a probabilistic distribution over P. To specify the distributions, let wi be a vector of probabilistic weights of length 4 that represents the probabilistic distribution over P corresponding to pi. In this example we set w1= [0.97,0.01,0.01,0.01], w2= [0.75,0,0.25,0], w3= [0.25,0.25,0.25,0.25], w4= [0,0.58,0,0.42].Then a possible RSI procedure may do the flowing. It starts from p3. First p3 generates p4. Since S(p4)> S(p3), the current program is not updated. Then p3 generates p2. The current program is updated to p2 because S(p2)< S(p3). Next p2 generates p1, and the current program updates to p1. Since p1 has the lowest score (highest order), no future program will be updated. Figure 1 shows the corresponding Markov chain.

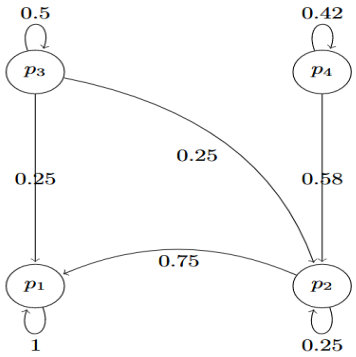


Figure 1 The Markov chain corresponding to the RSI

Fig. 1: The Markov chain corresponding to the RSI procedure defined by given scores and program generation probabilities.

A reasonable utility measure is the expected numbers of steps starting from a program to find the optimal program following our RSI definition. Furthermore, the score function needs to be consistent with the expected numbers of steps from programs to the optimal program following the process defined by itself. We mean that a score function S is consistent if for all p, p′∈P, S(p)> S(p′)implies that the expected number of steps to reach the optimal program from p is greater than starting from p′. More generally, if one takes some measure for a programs' ability to generate future programs, the score function needs to be consistent with this measure.

Two nice properties hold for this construction. First, the programs are added in a non-decreasing order of scores. Second, the score function equals the expected numbers of steps to reach the optimal program defined by this score function. We will prove the first property. The second property and the consistency of the score function are straightforward from the first property. We describe an example of how such score function is computed given the distributions to generate programs of each program and the optimal program. Consider the same abstraction of programs as the above example, where P={p1, p2, p3, p4} with corresponding probabilistic weights w1= [0.97,0.01,0.01,0.01], w2= [0.75,0,0.25,0], w3= [0.25,0.25,0.25,0.25], w4= [0,0.58,0,0.42]. Fix p1 to be the optimal program. Initially set S(p1) = 0 and S(pi) =∞, i=2,3,4. The transition function of initial Markov chain is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.75 & 0.25 & 0 & 0 \\ 0.25 & 0 & 0.75 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

At the first step, the expected number of steps from p2, p3, p4 following the current Markov chain are 4/3,4,∞. Hence we update S(p2) = 4/3. Because of the change of score, transition of the Markov chain change to

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.75 & 0.25 & 0 & 0 \\ 0.25 & 0.25 & 0.5 & 0 \\ 0 & 0.58 & 0 & 0.42 \end{bmatrix}$$

Then we compute the expected number of steps from p3 and p4 following the updated Markov chain. By some arithmetic we get the expectation are 8/3 for p3 and (approximately) 3.057 for p4. Since 8/3<3.057, update S(p3) = 8/3. By similar procedures, one can compute the score for S(p4).

Applying Genetic Algorithm on Multilayer Artificial Neural Network

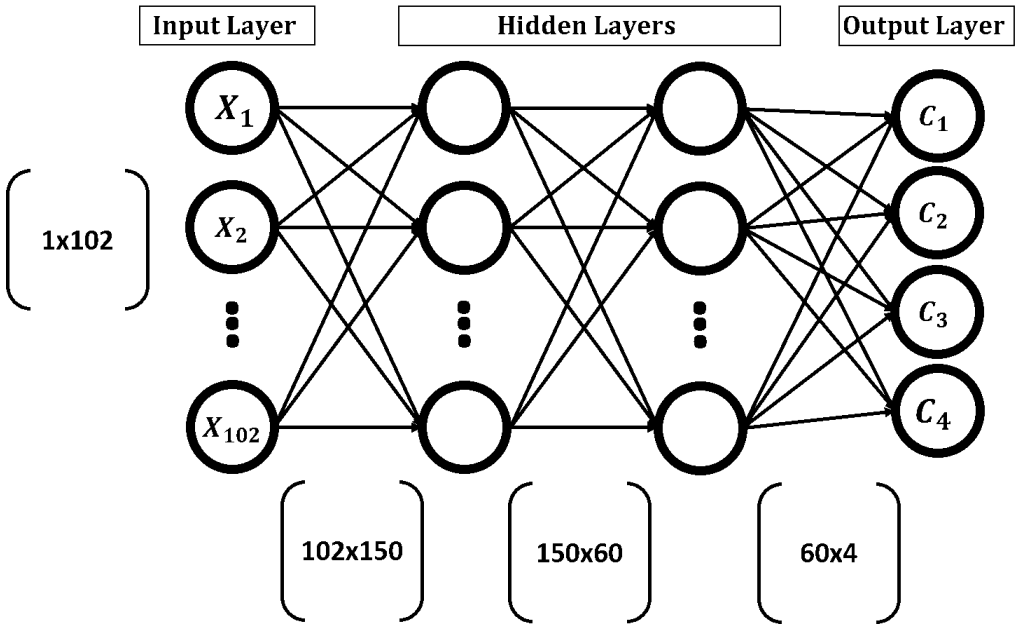
Genetic algorithms are stochastic search algorithms which act on a population of possible solutions. **Genetic algorithms** are used in artificial intelligence like other search algorithms are used in artificial intelligence — to search a space of potential solutions to find one which solves the problem. Thus a genetic algorithm(GA) is a type of artificial intelligence, modeled after biological evolution, by applying operations analogous to natural genetic processes to the population of programs.

Genetic algorithms have collections of solutions that are collided with each other to make new solutions, eventually returning the best solution. Since optimization and intelligence are deeply linked, using Genetic Algorithm to optimize Machine Learning or AI algorithm performances which would include ‘genetic algorithm’ as a numerical optimization technique.

In this paper , we use the genetic algorithm (GA) for optimizing the ANN network weights as the solution to the problem of very low accuracy in view of the fact that no backward pass for updating the network weights is used.

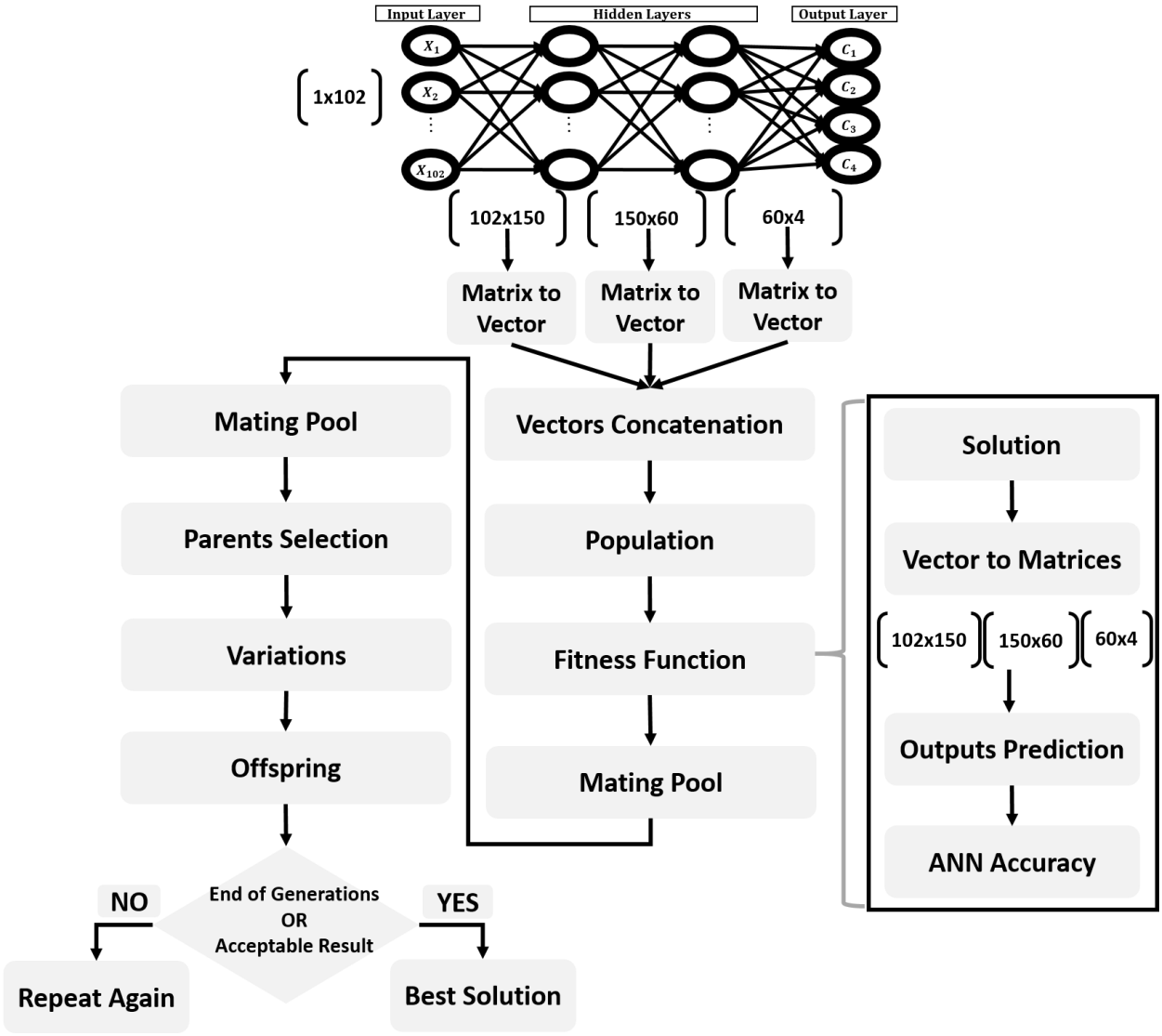
Using GA with ANN

GA creates multiple solutions to a given problem and evolves them through a number of generations. Each solution holds all parameters that might help to enhance the results. For ANN, weights in all layers help achieve high accuracy. Thus, a single solution in GA will contain all weights in the ANN. According to the network structure given in the figure below, the ANN has 4 layers (1 input, 2 hidden, and 1 output). Any weight in any layer will be part of the same solution. A single solution to such network will contain a total number of weights equal to 102x150+150x60+60x4=24,540. If the population has 8 solutions with 24,540 parameters per solution, then the total number of parameters in the entire population is 24,540x8=196,320.



Looking at the above figure, the parameters of the network are in matrix form because this makes calculations of ANN much easier. For each layer, there is an associated weights matrix. Just multiply the inputs matrix by the parameters matrix of a given layer to return the outputs in such layer. Chromosomes in GA are 1D vectors and thus we have to convert the weights matrices into 1D vectors.

Because matrix multiplication is a good option to work with ANN, we will still represent the ANN parameters in the matrix form when using the ANN. Thus, matrix form is used when working with ANN and vector form is used when working with GA. This makes us need to convert the matrix to vector and vice versa. The next figure summarizes the steps of using GA with ANN.

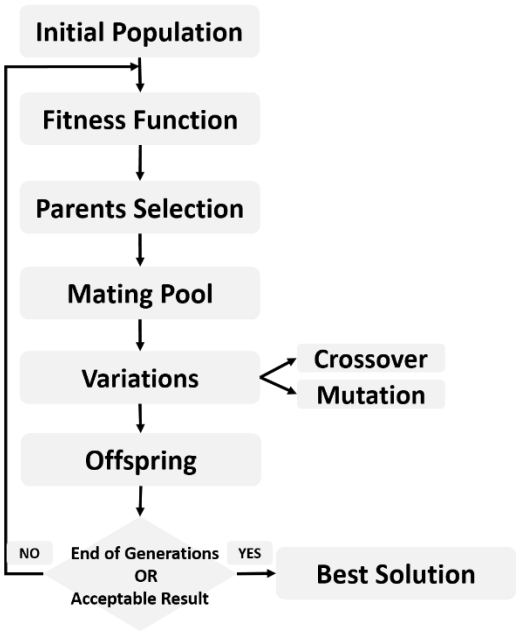


Weights Matrices to 1D Vector

Each solution in the population will have two representations. First is a 1D vector for working with GA and second is a matrix to work with ANN. Because there are 3 weights matrices for the 3 layers (2 hidden + 1 output), there will be 3 vectors, one for each matrix. Because a solution in GA is represented as a single 1D vector, such 3 individual 1D vectors will be concatenated into a single 1D vector. Each solution will be represented as a vector of length 24,540.

Implementing GA Steps

After converting all solutions from matrices to vectors and concatenated together, we are ready to go through the GA steps. The steps are presented in the figure above and also summarized in the next figure.



Remember that GA uses a fitness function to return a fitness value for each solution. The higher the fitness value the better the solution. The best solutions are returned as parents in the **parents selection** step.

One of the common fitness functions for a classifier such as ANN is the accuracy. It is the ratio between the correctly classified samples and the total number of samples. It is calculated according to the following equation. The classification accuracy of each solution is calculated according to steps in the above figure.

$$Accuracy = \frac{NumCorrectClassify}{TotalNumSamples}$$

The single 1D vector of each solution is converted back into 3 matrices, one matrix for each layer (2 hidden and 1 output).

The matrices returned for each solution are used to predict the class label for each of the samples in the used dataset to calculate the accuracy. This is done using 2 functions. The first function accepts the weights of a single solution, inputs, and outputs of the training data, and an optional parameter that specifies which activation function to use. It returns the accuracy of just one solution not all solutions within the population. It order to return the fitness value (i.e. accuracy) of all solutions within the population, the **second** function loops through each solution, pass it to the **first** function, store the accuracy of all solutions into an array, and finally return such an array.

After calculating the fitness value (i.e. accuracy) for all solutions, the remaining steps of GA as shown in the above figure are applied. The best parents are selected, based on their accuracy, into the mating pool. Then mutation and crossover variants are applied in order to produce the offspring. The population of the new generation is created using both offspring and parents. These steps are repeated for a number of generations. We can also try different values for the GA parameters such as a number of solutions per population, number of selected parents, mutation percent, and number of generations.

RESULTS

The test results of the proposed RSI procedure (Wenyi Wang) in simulation with randomly generated abstraction of programs where a fixed number of programs is chosen from $n= 2^l$, $l = 1,2,…..20$. The first program is designed to generate programs uniformly over all programs. Other programs generate programs follow a weighted distribution over a subset of programs. With 10 repeats for each $l = 1,2,……20$, the expected number of steps for the first program to reach the optimal program has been calculated and the results suggest a linear relation between l (Number of Programs) and expected number of steps.

GA-ANN

Based on 50 generations, and using visualization library that shows how the accuracy changes across each generation. It is observed that after 50 iterations, On the MNIST dataset, we are able to find an accuracy that is more than 50%. This is compared to 25% with no backward pass for updating the network weights and without using an optimization technique. This is an evidence about why results might be bad not because there is something wrong in the model or the data but because no optimization technique is used. However, using different values for the parameters such as 100 generations might increase the accuracy.

CONCLUSION

Self-Improvement where a program gets better at achieving goals as it receives input. In this paper, we first find an optimal program defined by given scores and program generation probabilities using Markov Chain. The second, we proposed hybrid genetic algorithm-artificial neural network predictive model as an optimization approach that mimics the concept of natural evolution / natural selection. This allowed us to create an artificial neural network which optimizes its own weights by self-improving and returning the best solution. The test results are encouraging with good accuracy.

REFERENCES

1. Wenyi Wang " A Formulation of Recursive Self-Improvement and Its Possible Efficiency". https://arxiv.org/pdf/1805.06610.pdf

2. Kory Becker, Justin Gottschlich “AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms “ https://arxiv.org/abs/1709.05703
3. Ahmed Gad “ Artificial Neural Networks Optimization using Genetic Algorithm with Python” Towards Data Science