Artificial Superintelligence: A Recursive Self-Improvement Model

Poondru Prithvinath Reddy

January 30, 2020

Artificial Superintelligence : A Recursive Self-Improvement Model

Poondru Prithvinath Reddy

ABSTRACT

Recursive self-improving(RSI) systems create new software iteratively. The newly created software iteratively generates a greater intelligent system using the current system, then this process leads to a phenomenon referred to as superintelligence. However, many existing studies on RSI systems lack clear mathematical formulation or results. In this paper, we provide a formal definition of RSI systems and then we present a recursive self-improvement model by three 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 is to model by embedding histories when generating a new program. And the third is to model the programs taking a program as an argument and return a suggested improvement of the given program. We use simulation to show that we achieve logarithmic runtime complexity with respect to the size of the search space and realize good accuracy to a AI model of embedding histories. The results suggest that it is possible to achieve an efficient recursive self-improvement.

INTRODUCTION

If research into strong AI produced sufficiently intelligent software, it would be able to reprogram and improve itself – a feature called "recursive self-improvement(RSI)". 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.

Recursive self-improving(RSI) systems create new software iteratively. The newly created software should be better at creating future software. With this property, the system has potential to completely rewrite its original implementation, and take completely different approaches. Chalmers' proportionality thesis hypothesizes that an increase in the capability of creating future systems proportionally increases the intelligence of the resulting system. With this hypothesis, he shows if a process iteratively generates a greater intelligent system using the current system, then this process leads to a phenomenon many refer to as super-intelligence. However, many existing studies of RSI systems remain philosophical or lack clear mathematical formulation or results.

METHODOLOGY

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.

In this paper, we provide a mathematical and other formulations for a class of RSI procedures and show that there are such different computable RSI systems or approaches.

The methodology essentially consist of the following :

- To find the **optimal program following RSI** procedure defined by given scores and program generation probabilities using Markov chain.
- To model by **embedding histories** when generating a new program.
- To model the programs taking a **program as an argument** and return a suggested improvement of the given program.

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'∈ 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 ∈ 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.

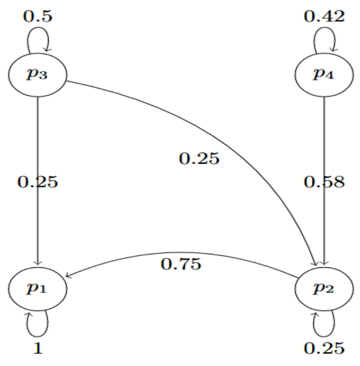
$$\begin{tikzcd}
p_3 \arrow[loop above, "0.5"] \arrow[d, "0.25"] \arrow[rd, bend left, "0.25"] & p_4 \arrow[loop above, "0.42"] \arrow[d, "0.58"] \\
p_1 \arrow[loop below, "1"] & p_2 \arrow[l, bend right, "0.75"'] \arrow[loop below, "0.25"]
\end{tikzcd}$$

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).

EMBEDDDING HISTORIES

Recursive self-improvement describes software that writes its own code in repeated cycles of improvement. It is associated with artificial intelligence as self-improving software has potential to develop superintelligence.

Recursion can be seen as an elegant 'architectural factorization' - building complexity by combining the results of smaller, similar patterns previously encountered. Computationally, recursion can always be converted into iteration so this form of elegance is mainly of use in helping to make designs more comprehensible.

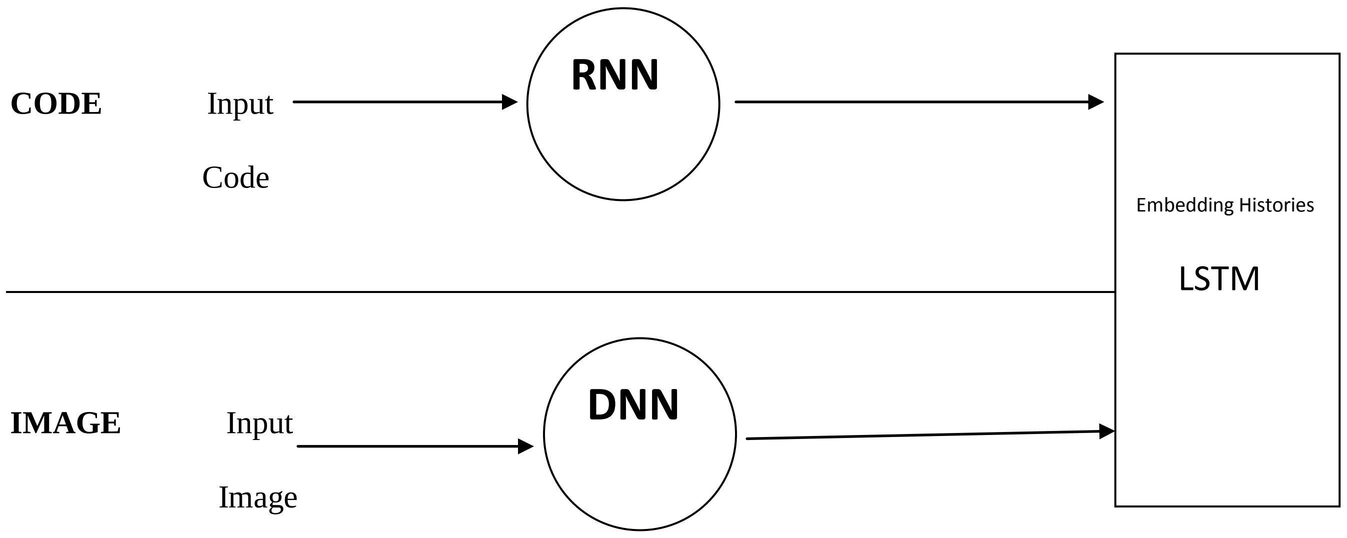

Figure 2: The EHM(Embedding History Model) architecture. It contains three subsystems that are trained separately. In the image subsystem, the encoder can transfer an input (or predicted) image into a population representation vector I at the DNN layer (mimicking the Deep Neural Network for high-level image representation), and the decoder can reconstruct a vector output from LSTM to a predicted image, which can be fed into the encoder to form the guided loop. In the code subsystem, The coding system which consists of a mapping to transfer symbol texts into respective numeric and a RNN to extract the sequence dependencies from the input texts, and an output encoder to convert numeric values into text symbols. There is a memory layer

implemented by a RNN to extract sequence information from the vector C. The LSTM layer serves as working memory, that takes the concatenated input [C, I] from both code and image subsystems, and output the predicted next element representation that could be fed back into both subsystems to form a guided loop.

As is shown in Figure 2, the EHM network contains three main subsystems including the code, image and LSTM subsystems. The image encoder network was trained separately. After training, the encoder is separated into two parts: the encoder (or recognition) part ranges from the image entry point to the final encoding layer, to provide the high-level abstract representation of the input image; the decoder part ranges up to image prediction point. The activity vector of the encoding layer are concatenated with code activity vector as input signals to the LSTM. Finally, the predicted image is fed back to the encoder network for the next iteration. The code processing component first converts the input text symbol into a sequence of binary vectors [C(t = 0), . . . ,C(T)], where T is the text length. To improve the code recognition, we added one RNN layer to generate the sequence dependencies of the text. The LSTM training based on the next component prediction (NCP). The LSTM is trained by the NCP principle, where the goal of the LSTM is to output the representation vectors (including both code and image) of the next component which required the understanding of the previous text code and observed images. The LSTM subsystem contains a LSTM and a full connected layer. It receives inputs from both code and image subsystems in a concatenated form of c(t) = [C(t),I(t)] at time t, and gives a prediction output a a'(t) = [C'(t), I'(t)], which is expected to be identical to a(t + 1) = [C(t + 1), I(t + 1)] at time t+1. This has been achieved with a next component prediction (NCP). So given an input image, the LSTM can predict the corresponding code description. The strategy of learning by predicting its own next component is essentially an unsupervised learning. Our LSTM subsystem was trained separately after code and image components had completed their functionalities. Finally, we demonstrate how the network forms a thinking loop with text code and predicted images.

DATASET

User Interface Elements

When designing the user interface, the following Interface elements are considered but are not limited to:

- **Input Controls**: pointer, checkboxes, radio buttons, dropdown lists, list boxes, buttons, toggles, text fields, date field, frames, combo boxes, timer, hscrollbar, vscrollbar, drivelistboxes, dirlistboxes, filelistboxes, shape, line, pictureboxes, data, ole, labels, charts
- **Navigational Components**: breadcrumb, slider, search field, pagination, slider, tags, icons

- **Informational Components**: tooltips, icons, progress bar, notifications, message boxes, modal windows, links
- **Containers**: accordion

A total of 40 user interface components / elements along with C programming language scripts / code associated with each visual component has been selected as dataset.

We have for the Code Subsystem and used the following parameters with Recurrent Neural Network :

• Size of a sequence: 50
• Size of a batch: 40
• Number of neurons : 256
• Depth of RNN: 2
• Learning rate: 0.0005
• AdamOptimizer to minimize our errors
• Dropout: 0.5

The results were obtained after training the model on CPU and the model is fit over 100 epochs.

It's interesting to see that this model has clearly understood the general structure of a program related to visual components; A function, parameters, variables, conditions, etc.

Image Subsystem is an implementation of building a deep neural network with TensorFlow for Image Classification in user interface component dataset.

We used 40 images of different visual components / elements from User Interface elements dataset.

We have constructed neural network architecture, layer by layer with the help of the TensorFlow package.

- Next, we build up the network.
- Activation function : Relu, Rectified linear unit.
- We constructed a fully connected layer that generates logits of size [None, 40].
- With the multi-layer perceptron built out we define the loss function and the loss function we make use of is sparse softmax cross entropy.
- We pick the ADAM optimizer, for which we define the learning rate at 0.001.

The above has been implemented with Python and TensorFlow as a backend.

We have now successfully trained our model with all the visual components.

Then we loaded in the test component data and run predictions , and found that images were classified with good accuracy.

The LSTM subsystem contains a LSTM and a fully connected layer. It receives inputs from both code and image subsystems in a concatenated form of **c**(*t*) = [**C**(*t*),**I**(*t*)] at time t, and gives a prediction output **a'**(*t*) = [**C'**(*t*),**I'**(*t*)] , which is expected to be identical to **a**(*t* + 1) = [**C**(*t* + 1),**I**(*t* + 1)] at time t+1. This has been achieved with a next component prediction (NCP) . So given an input image, the LSTM can predict the corresponding code description. The strategy of learning by predicting its own next element is essentially an unsupervised learning.

The Training is based on the next component prediction (NCP). The LSTM-FC is trained by the NCP principle, where the goal of the LSTM-FC is to output the representation vectors (including both code and image) of the next component / element. At time T, the LSTM of EHM generated the guided digit instance, which required the understanding of the previous code language and observed images.

The LSTM subsystem was trained separately after vision and code components had completed their functionalities. We have trained the network to accumulatively learn different components, and the related code results. Finally, it is demonstrated how the network forms a thinking loop with code language and observed images.

The LSTM layer serves as working memory, that takes the concatenated input [**C**,**I**] from both code and image subsystems, and output the predicted next component representation that could be fed back into both subsystems to form a guided loop.

PROGRAM AS AN ARGUMENT

Recursive procedures are functions that invoke themselves either directly (call themselves from within themselves) or indirectly (calls another method that calls original method).

Recursion:

. An alternative to iteration

. Recursion can be very elegant at times,

. Not inexpensive to implement

. Ease of writing and maintaining procedures.

Classic examples of recursion :

. Recursive calculation of a string length, factorials, divide and conquer, towers of Hanoi, binary searches, and more

Recursive functions are used in many applied areas :

. In artificial intelligence.

. In searching data structures that are themselves "recursive" in nature, such as trees.

The Expense of Recursive Functions :

. RECURSION has a significant overhead.

. "DEEPER" WE GO, Need ADDITIONAL COPIES OF THE DATA

. These 'copies' are stored in "STACK FRAMES" which contain current values and outstanding function calls, and more.

RECURSIVE FUNCTION IS WRITTEN AS:

```
int public factorial (int n)

{

        if (n==1)

                return (1)                  /*  THIS IS THE BASE CASE  */

        else

                return (n * factorial (n-1) );     /* RECURSIVE CASE */

}end factorial ()
```

RECALL:

When a function calls itself, this implies **development** of a **new set of local variables** and same "variable **names**" but very **different** variables (different memory addresses) and different **values.** There may also be some other **temporary** variables during the life of a particular 'call.'

Base case is a very important notion and determining the base case **ensures** the recursive function will **terminate** someday.

SIMULATING RECURSION

- logical simplicity and self-documentation of recursive solutions.
- define binary tree and a LNR traversal as a ‘tree’ is a recursive data structure.
- a problem solved naturally recursively.

Let us write a Recursive program to print all permutations of a given string.

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.

The permutations of string “ABC” - consisting of six possible arrangement or order(Ex. ABC ACB BAC BCA CBA CAB).

The Recursion Tree for permutations of string “ABC” and the string is represented as a data structures that are themselves "recursive" in nature, such as trees.

A program to output all permutations of string has been implemented in Python and uses function “permute” recursively with three parameters.

Algorithm Paradigm**:** Backtracking
Time Complexity: There are n! permutations and it requires O(n) time to print a permutation.

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.

After 200 steps training, EHM could not only reconstruct the input image but also predict the element / component with associated program script / code, correct parameters and variables just after the image classification. After training of 200 steps, EHM could classify various visual components with correct code (accuracy = 15%). Note that, the classification process is not performed by large dataset, but by small number of training steps or iterations which is resulting in less accuracy. However, accuracy can be improved if a program gets better at achieving goals as it receives input and a self-improving program would build complexity by combining the results of smaller, similar patterns previously encountered. The program improves itself by varying its source code or model and testing whether the changes find larger set of visual components.

The results of Recursion Tree for permutations of string show that there are n! permutations and it requires O(n) time to output a permutation.

CONCLUSION

Recursive self-improvement(RSI) systems create new software iteratively using the current system and this process leads to a phenomenon referred to as superintelligence. We presented a recursive self-improvement model by three different approaches. We used simulation to achieve logarithmic runtime complexity with respect to size of the search space and realize good accuracy with AI model of embedding histories. However, embedding histories can be seen as factorization – building complexity by combining the results of smaller, similar patterns previously encountered. The results suggest a possibility of more comprehensible recursive self-improvement.

REFERENCES

1. Wenyi Wang   “ A Formulation of Recursive Self-Improvement and Its Possible Efficiency”. https://arxiv.org/pdf/1805.06610.pdf
2. Matt Mahoney “A Model for Recursively Self Improving Programs”. http://mattmahoney.net/rsi.pdf
3. Poondru Prithvinath Reddy “Artificial Intelligence that Learn to Write Code : Memory Guided Programming”. Google Scholar