EasyChair Preprint

№ 2860

# Artificial Superintelligence: The Recursive Self-Improvement in NLP

Poondru Prithvinath Reddy

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

March 4, 2020

# Artificial Superintelligence : The Recursive Self-Improvement in NLP

**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 phenomenon referred to as superintelligence. In this paper, we provide a formal definition of RSI systems and then we present a recursive self-improving model which takes a program as an argument and return a suggested improvement of the given program. With this concept, we have built a recursive neural network(RNN) which is a word vectorization useful for natural language processing( NLP ) and also which can be accomplished with a recursive algorithm known as Word2vec. We have converted a corpus of words into vectors by both recursive neural network using parse tree & recursive Word2vec method without tree structure which are then thrown into vector space to structure text data hierarchically for outputting parse trees for a sentence and measure the cosine distance between word-context pairs i.e. their similarity or lack of, and the test results are encouraging with a possibility of more comprehensible 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.

Recursive neural networks (RNNs) are neural nets useful for natural-language processing. They have a tree structure with a neural net at each node. We can use recursive neural networks for boundary segmentation, to determine which word groups are positive and which are negative. The same applies to sentences as a whole.

Word vectors are used as features and serve as the basis of sequential classification. They are then grouped into subphrases, and the subphrases are combined into a sentence that can be classified by sentiment and other metrics.

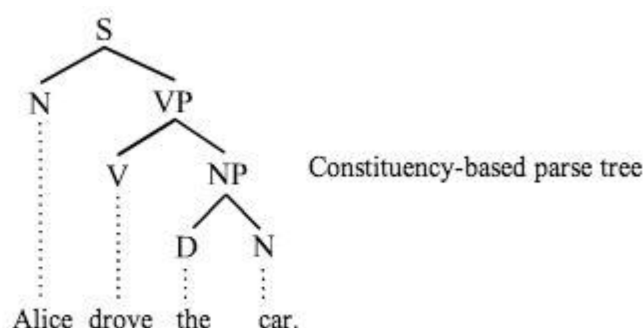
Recursive neural networks require external components like Word2vec. To analyze text with neural nets, words can be represented as continuous vectors of parameters. Those word vectors contain information not only about the word in question, but about surrounding words; i.e. the word's context, usage and other semantic information.

The first step toward building a working RNN is word vectorization, which can be accomplished with an algorithm known as Word2vec. Word2Vec converts a corpus of words into vectors, which can then be thrown into a vector space to measure the cosine distance between them; i.e. their similarity or lack of.

Word2vec is a separate pipeline from NLP. It creates a lookup table that will supply word vectors once we are processing sentences. Whereas Natural-language-processing pipeline will ingest sentences, tokenize them, and tag the tokens as parts of speech.

To organize sentences, recursive neural networks use constituency parsing, which groups words into larger subphrases within the sentence; e.g. the noun phrase (NP) and the verb phrase (VP). This process relies on machine learning, and allows for additional linguistic observations to be made about those words and phrases. By parsing the sentences, we are structuring them as trees.

The trees are later binarized, which makes the math more convenient. Binarizing a tree means making sure each parent node has two child leaves. Sentence trees have their a root at the top and leaves at the bottom, a top-down structure that looks like the following :




The entire sentence is at the root of the tree (at the top); each individual word is a leaf (at the bottom).

Finally, word vectors can be taken from Word2vec and substituted for the words in our tree, we then combine those word vectors with neural nets.

The methodology essentially consist of the following :

- To model the programs taking a **program as an argument** and return a suggested improvement of the given program. We implement recursive neural network by combining word vectors with neural nets for natural-language processing. Also we implement word vectors without tree structure recursively.

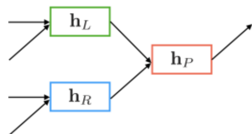
## ARCHITECTURE

### Recursive Neural Network with Tree Structure

Recursive Neural Network is one of Recurrent Neural Networks that extended to a tree structure. As both networks are often written as RNN, but in natural language

processing it sometimes refers to the Recursive Neural Network. Recursive Neural Network uses a tree structure with a fixed number of branches. In the case of a binary tree, the hidden state vector of the current node is computed from the hidden state vectors of the left and right child nodes, as follows:

$$\mathbf{h}_P = a\left(\mathbf{W}\begin{bmatrix}\mathbf{h}_L \\ \mathbf{h}_R\end{bmatrix} + \mathbf{b}\right)$$




Recursive Neural Networks are natural mechanisms to model sequential data. This is so because language could be seen as a recursive structure where words and sub-phrases compose other higher-level phrases in a hierarchy. In such structure, a non-terminal node is represented by the representation of all its children nodes. The figure 1 below illustrates a simple recursive neural network .

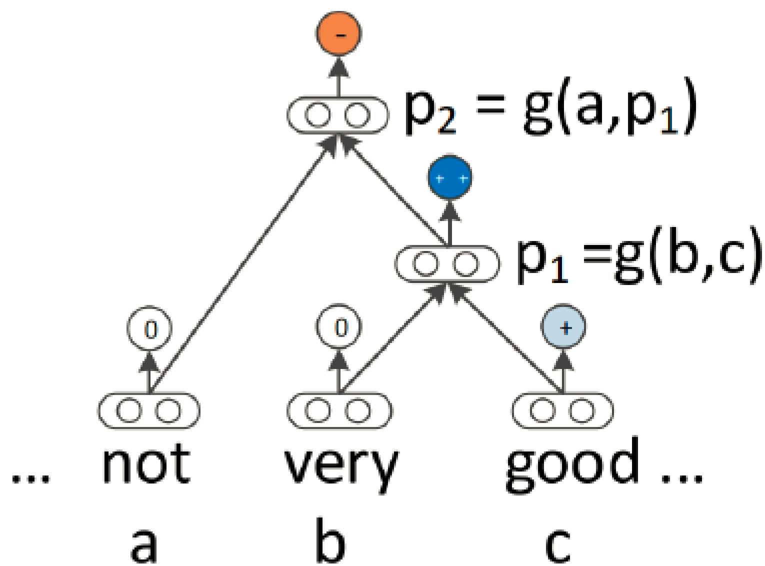



Figure 1: Recursive Neural Network

In the basic recursive neural network form, a compositional function (i.e., network) combines constituents in a bottom-up approach to compute the representation of higher-level phrases (see figure above).

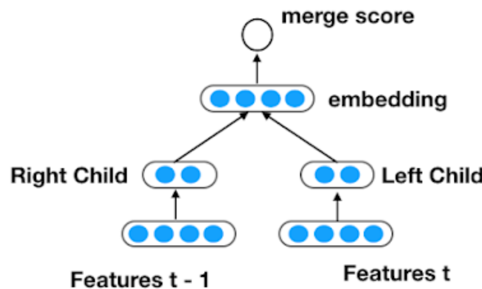
In the standard recursive neural network, It is first determined which parent already has all its children computed. In the above figure, *p1* has its two children's vectors since both are words. RNNs use the following equations to compute the parent vectors:

$p_1 = f(W\begin{bmatrix} b \\ c \end{bmatrix})$, $p_2 = f(W\begin{bmatrix} a \\ p1 \end{bmatrix})$,

where f = tanh is a standard element wise nonlinearity, $\mathbf{W} \in \mathbf{R}^{\mathbf{dx2d}}$ is the main parameter to learn and we omit the bias for simplicity. The parent vectors must be of the same dimensionality to be recursively compatible and be used as input to the next composition. Each parent vector *pi*, is given to the same softmax classifier of Eq.( $y^a$ = softmax($W_s a$), where $Ws \in R^{5xd}$ is the sentiment classification matrix ) to compute its label probabilities.

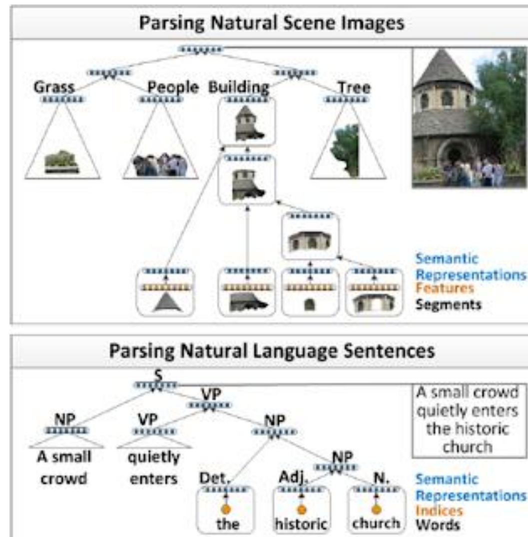
Recursive neural networks are used for various applications and in this paper, we use them for Parsing.

Recursive Neural Networks with merging score is given as below :




A Recursive Neural Network

Recursively Neural Networks can learn to structure data such as text or images hierarchically. The network can be seen to work similar to a context free grammar and the output is a parse t. The figure below shows examples of parse trees for an image (top) and a sentence (bottom).




A recursive neural network for sequences always merges two consecutive input vectors and predicts if these two vectors can be merged. If so, we replace the two vectors with best merging score with the hidden vector responsible for the prediction and this is done in a recursive manner by constructing a parse tree. We have used a recursive neural network which is replicated for each pair of possible input vectors. We predict parse trees and compute their scores with RNN, and this network is a different RNN formulation in that it predicts a score for being a correct merging decision.

Formally we construct the network as follows :

Given an input sequence $x = x_1 \ldots x_T$, $x_i \in \mathbb{R}^D$ then for two neighboring inputs $x_t, x_{t+1}$ we predict a hidden representation:

$$h_t = \sigma(x_t W_l + b_l)$$
$$h_{t+1} = \sigma(x_{t+1} W_r + b_r)$$

The hidden layer used for replacement is computed from the concatenation of the two previous hidden layers: $h = \sigma([h_t, h_{t+1}] W_h + b_h)$. Finally, the prediction is simply another layer predicting the merging score.

As a part of the implementation, we first defined node class which also handles the parsing, by greedily merging up to a tree. A node in the tree holds it's representation as well as a left and a right child. In other words, each node represents a merging decision with the two children being the nodes merged and the parent representation being the hidden layer in the merger grammar. Parsing a tree involves merging the vectors with the highest score and replacing the nodes with their parent node. Also as a part of the implementation, we defined the neural network with the merging decision. Given a node

with the parent representations (for leaf nodes that are the input vectors) we build the hidden representations and predict the label from the hidden representation, and then compute the merging score.

We have used a text file containing sentences for training the recursive neural network and during learning, we collected the scores of the subtrees and used labels to decide if a merge was correct or not. Further, training is done with an aim to increase scores of segment pairs with the same label and decrease scores of pairs with different labels, unless no more pairs with the same labels are left.

The Recursive Neural Network has been implemented in Python using PyTorch.

**Word Vectors Without Tree Structure Recursively**

**Word2Vec**

Word2vec is a two-layer neural net that processes text by “vectorizing” words. Its input is a text corpus and its output is a set of vectors: feature vectors that represent words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand.

The purpose and usefulness of Word2vec is to group the vectors of similar words together in vectorspace. That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words. It does so without human intervention.

Given enough data, usage and contexts, Word2vec can make highly accurate guesses about a word’s meaning based on past appearances. Those guesses can be used to establish a word’s association with other words (e.g. “man” is to “boy” what “woman” is to “girl”), or cluster documents and classify them by topic. Those clusters can form the basis of search, sentiment analysis and recommendations in such diverse fields as scientific research, legal discovery, e-commerce and customer relationship management.

The output of the Word2vec neural net is a vocabulary in which each item has a vector attached to it, which can be fed into a deep-learning net or simply queried to detect relationships between words. Measuring cosine similarity, no similarity is expressed as a 90 degree angle, while total similarity of 1 is a 0 degree angle, complete overlap; i.e. Sweden equals Sweden, while Norway has a cosine distance of 0.760124 from Sweden.

The word vectors tend to embed syntactical and semantic information and are responsible for  a wide variety of NLP tasks such as sentiment analysis and sentence compositionality. Distributed representations were heavily used in the past to study

various NLP tasks, but it only started to gain popularity when the continuous bag-of-words (CBOW) and skip-gram models were introduced to the field. They were popular because they could efficiently construct high-quality word embeddings and because they could be used for semantic compositionality (e.g., ‘man’ + ‘royal’ = ‘king’).

**Word2Vec** consists of models for generating word embedding. These models are shallow two layer neural networks having one input layer, one hidden layer and one output layer. Word2Vec utilizes two architectures :

1. **CBOW (Continuous Bag of Words) :** CBOW model predicts the current word given context words within specific window. The input layer contains the context words and the output layer contains the current word. The hidden layer contains the number of dimensions in which we want to represent current word present at the output layer.

2. **Skip Gram :** Skip gram predicts the surrounding context words within specific window given current word. The input layer contains the current word and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word present at the input layer.

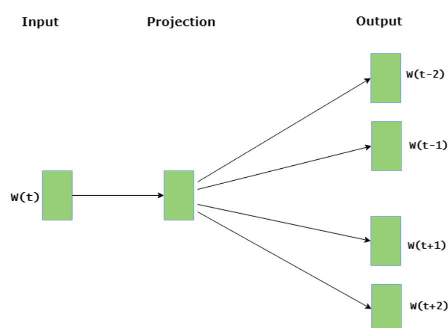



Figure 2. Skip Gram Model

The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space.

For generating word vectors in Python using word2vec, modules needed are **nltk** and **gensim**.

For generating word vectors, we have used a text file.

In this paper, we have implemented the **Skip-gram** architecture recursively. The implementation essentially consisted the following :-

1. Importing all necessary modules from nltk and gensim
2. **Data Preparation** — Reading text file for tokenize the sentence into words
3. **Generate Training Data** — Build vocabulary and one-hot encoding for words

4. **Creating Skip Gram model recursively by using most similar word from word2vec model and returning similarity function which calls itself**. However recursion will terminate, when the supplied text contains no known words.
5. **Model Training** — Passing encoded words through forward pass and calculate error rate
6. **Inference** — Get word vector and find similar words by measuring Cosine Similarity for Skip Gram model.

## RESULTS

Initial experiments showed that the Recursive Neural Network model worked with lower accuracy at the sentence level. This is due to the fact that the learning was performed with smaller dataset and also performance varied at larger or smaller phrases and batch sizes.

In word2vec Skip Gram model experiment, we look up the vector for the given words and also to find similar words by calling the function for computing the cosine similarity between words. The recursive skip gram model achieves lower performance on word similarity. This is mainly due to training with smaller dataset and batch size thus affecting accuracy.

## CONCLUSION

Recursive self-improvement( RSI ) systems create new software iteratively using the current system and this process leads to phenomenon referred to as superintelligence. We presented a recursive self-improvement model where a program is taken as an argument and return a suggested improvement. We built a recursive neural network for natural language processing and achieved word vectorization with a recursive algorithm known as Word2vec. The results are encouraging with a measured cosine distance in vector space and return parse trees for a sentence in NLP, and suggest a possibility of more comprehensible recursive self-improvement.

## REFERENCES


1. Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank; Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng and Christopher Potts; 2013; Stanford University.
2. Parsing Natural Scenes and Natural Language with Recursive Neural Networks; Richard Socherrichard, Cliff Chiung-Yu Linchiungyu, Andrew Y. Ngang, Christopher D. Manningmanning; Stanford University.
3. www.github.com