EasyChair Preprint

№ 1981

Self-Replicating AI : Integrating New Neural Networks Through a Natural Selection Process

Poondru Prithvinath Reddy

November 18, 2019

Self-Replicating AI : Integrating New Neural Networks Through a Natural Selection Process

Poondru Prithvinath Reddy

ABSTRACT

Self-replication is any behavior of a system that yields construction of an identical copy of itself. Biological cells, given suitable environments, reproduce by cell division. In this paper, we present a self-replicating neural network that integrates new neural networks through natural selection process. The network replicates itself by integrating other neural networks and by learning to output identical copies of its own components for which it is trained. Also we describe a method of an optimization technique using Genetic Algorithm (GA) 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 the ANN predictive model (GA-ANN) and the network replicates itself by learning to optimize its own weights. We observe from the test results that the networks were able to replicate through natural selection with good accuracy. It is observed that self-replication mechanism for artificial intelligence is convenient because it establishes the possibility of persistent improvement through natural selection.

INTRODUCTION

Self-replication is any behavior of a dynamical system that yields construction of an identical copy of itself. Biological cells, given suitable environments, reproduce by cell division. During cell division, DNA is replicated and can be transmitted to offspring during reproduction.

The ability to pass on successful traits is a defining characteristic of biological organisms. Earlier this year, two researchers from Columbia University found a way to apply this principle to artificially intelligent systems — creating self-replicating neural networks called “quines.” The idea of self-replicating, self-evolving AI that can automatically take on the most successful traits of previous generations is a pretty tantalizing one, with lots of potentially useful applications.

While self-replication has been studied in many automata, it is notably absent in neural network research, despite the fact that neural networks appear to be the most powerful form of AI known to date. In this paper, we identify and attempt to solve the challenges involved in building and training a self-replicating neural network that integrates new networks. Specifically, we propose to view a neural network as a differentiable computer program composed of a sequence of tensor operations. Our objective then is to construct a neural network that integrates with other neural networks to replicate components and also a neural network that optimizes its own weights. The best solution for a self-replicating network was found by alternating optimization steps.

METHODOLOGY

The methodology essentially consists of two parts :-

1. Integrating new neural networks through natural selection process
2. Optimization steps –The best solution for a self-replicating network by using genetic algorithm.

ARCHITECTURE

Integrating New Neural Networks

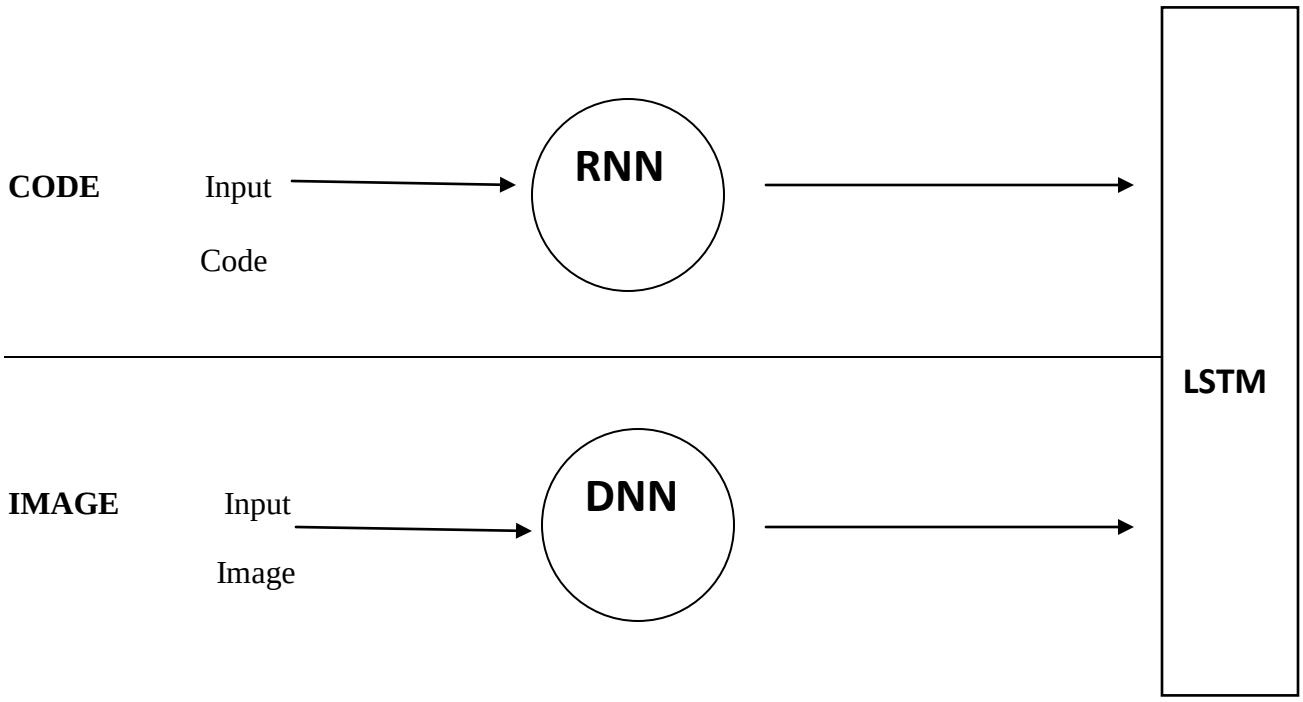


Figure 1: The INNN(Integrating New Neural Networks) architecture. It contains three subsystems that are trained separately.. In the image subsystem, the encoder can transfer an input (or predicted) image into a population representation vector I at the DNN layer (mimicking the Deep Neural Network for 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 1, the INNN 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.

CODE SUBSYSTEM

The First problem is to represent our data.

A neural network treats only numbers. Everything else is unknown to the network. Thus, each character of our dataset should be represented in this form (a number / characters).

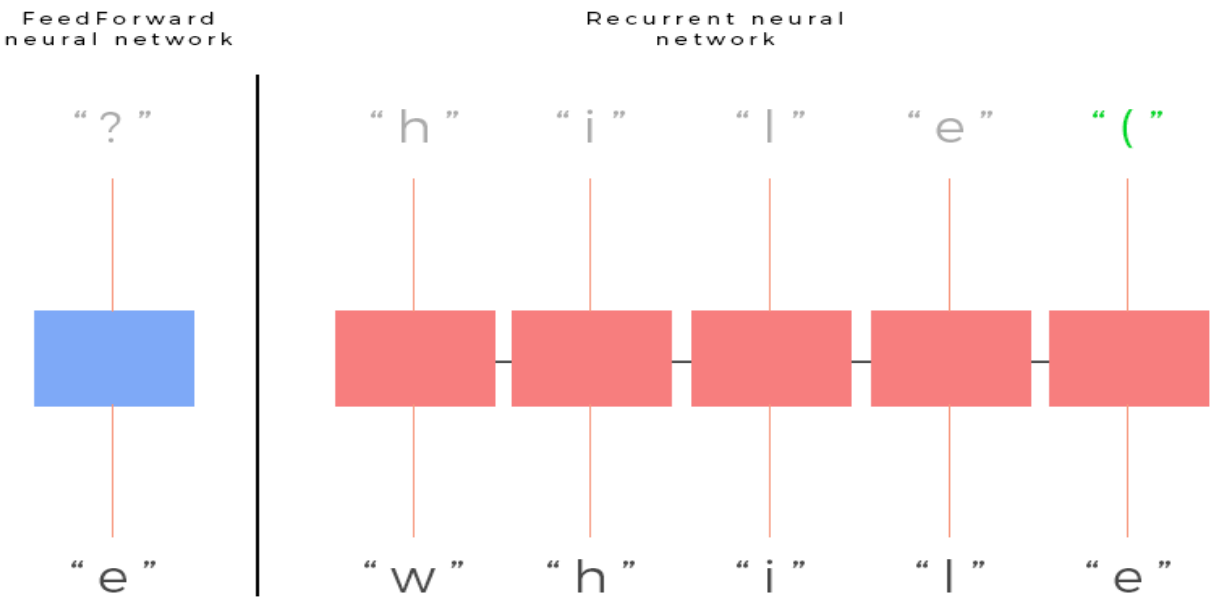
First we need to Load the text file and create character to integer mappings. The entire text file is read, we would be mapping each character to a respective number and all characters are converted to numbers. This is done to make the computation part of the RNN easier.

For example, if the character "=" is assigned to the number 7, we will then represent each number in one hot encoding in order to better converge during the backpropagation.

The three important variables to remember here are **vocab_to_int**, **int_to_vocab** and **encoded**. The first two allow us to easily switch between a character and an int and vice versa. The last is the representation of all our dataset in an encoder format. (Only int instead of characters)

We therefore create a neural network taking into account the temporal space of the characters type. To do this, we need to use a reccurent neural network.

Recurrent neural network



In order to illustrate, a classic classifier (on the left of the diagram) takes the preceding letter; it’s passed by the hidden layer represented in blue in order to deduce an output. A recurring neural network is architecturally different. Each cell (represented in red) is not only connected to the inputs, but also to the cell of the instant t-1. In order to represent our code subsystem, we will use RNN (Recurrent Neural Network) cells.

Building the model

We will describe this with 5 main parts. Placeholder serving as an entry to our model. The initialization of our cells used to create the RNN. The output layer connected to each cell. The operation used to measure the model error. Finally, we will define the training operation.

Graph inputs

We define a placeholder for the input, and the shape expected for our input is therefore of size [number, size]. Each entry of the input batch being associated with a single output, we can define the same shape for our target. Finally we define a placeholder for the value of the probability used for the future dropout.

RNN

- **create_cell()** is used to create an RNN cell composed of neurons. This function also adds a dropout to the cell output.
- **tf.contrib.rnn.MultiRNNCell** is used to easily instantiate our rnn. We give as a parameter an array of **create_cell()** because we want an RNN consisting of several layers.
- **initial_state**: Knowing that each cell of an RNN depends on the previous state, we must instantiate an initial state filled with zero that will serve as input to the first entries.
- **cell_outputs** gives us the output of each cell of our RNN.
- **final_state** returns the state of our last cell which can be used during training as a new initial state for a next batch.

Graph outputs

The values at the output of our cells are stored in a three-dimensional table [number of sequences, sequence size, number of neurons] or [2, 10, 4]. We no longer need to separate the outputs by sequences. We then resize the output to get an array of dimension [20, 4] stored in the **seq_out_reshape** variable.

Finally, we apply a simple linear operation: **tf.matmul (..) + b.** This followed by a softmax in order to represent our outputs in the form of probability.

Loss

In order to apply our error operation, the targets of our batch must be represented in the same way and in the same dimension as the output values of the model. We use **tf.one_hot** to represent our outputs under the same encoding as our inputs. Then we resize the array (**tf.reshape ()**) to the same dimensions of the linear output: **tf.matmul (..) + b**. We can now use this function to calculate the error of the model.

Training

We simply apply an AdamOptimizer to minimize our errors.

Results

It‘s finally the results of the training. We have for this one used the following parameters:

- Size of a sequence: 50
- Size of a batch: 40
- Number of neurons : 256
- Depth of RNN: 2
- Learning rate: 0.0005
- Dropout: 0.5

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

Finally, let’s look at what type of code our model is capable of generating . 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

This 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 start with a pretty simple analysis with the help of the ndim and size attributes of the images array: Note that the images and labels variables are lists, so we might need to use np.array() to convert the variables to an array.

As guessed the 40 labels that are included in this dataset, the components are different from each other. Also These images are not of the same size.

Let's start first with extracting some features - we'll rescale the images, and we'll convert the images that are held in the images array to grayscale. We'll do this color conversion mainly because the color matters less in classification.

To tackle the differing image sizes, we're going to rescale the images; We can do this with the help of the skimage or Scikit-Image library, which is a collection of algorithms for image processing.

In this case, the transform module will come in handy, as it offers a resize() function; We'll see that we make use of list comprehension to resize each image to 28 by 28 pixels. Once again, for every image that we find in the images array, we'll perform the transformation operation that is borrowed from the skimage library. Finally, we store the result in the images28 variable:

. Next we'll also go through the trouble of converting the images to grayscale. Just like with the rescaling, we again count on the Scikit-Image library to help out; In this case, it's the color module with its rgb2gray() function that we need to use to get where we need to be.

However, we need to convert the images28 variable back to an array, as the rgb2gray() function does expect an array as an argument.

We checked the result of grayscale conversion by plotting some of the images;

Now that we have explored and manipulated the data, it's time to construct neural network architecture, layer by layer with the help of the TensorFlow package.

- Next, we build up the network. We first start by flattening the input with the help of the flatten() function, which will give an array of shape [None, 784] instead of the [None, 28, 28], which is the shape of our grayscale images.

- Activation function :The activation function of a node defines the output given a set of inputs. A common activation function is a Relu, Rectified linear unit.
- After we have flattened the input, we construct a fully connected layer that generates logits of size [None, 40]. Logits is the function operates on the unscaled output of previous layers, and that uses the relative scale to understand the units is linear.
- With the multi-layer perceptron built out we can define the loss function. Loss function - after we have defined the hidden layers and the activation function, we need to specify the loss function and the optimizer. The loss function is a measure of the model's performance. We make use of

  sparse_softmax_cross_entropy_with_logits()

- This computes sparse softmax cross entropy between logits and labels. In other words, it measures the probability error in discrete classification tasks in which the classes are mutually exclusive. This means that each entry is in exactly one class. Here, a user element can only have one single label.
- The optimizer will help improve the weights of the network in order to decrease the loss. In this case, 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.

Now that we have built up our model layer by layer, it’s time to actually run it! To do this, we first need to initialize a session with the help of Session(). Next, we can use this initialized session to start epochs or training loops. In this case, we pick 201 because we want to be able to register the last loss_value; In the loop, we run the session with the training optimizer and the loss (or accuracy) metric that we defined. We also pass a feed_dict argument, with which we feed data to the model. After every 10 epochs, we’ll get a log that gives us more insights into the loss or cost of the model.

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

We still need to evaluate our neural network. In this case, we try to get a glimpse of how well our model performs by picking 10 random images and by comparing the predicted labels with the real labels.

We can first print them out, by using matplotlib to plot the components themselves and to make a visual comparison.

However, by looking at random images give us many insights into how well our model actually performs. Then we loaded in the test component data and run predictions , and found that images were classified with good accuracy.

LSTM SUBSYSTEM

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 $\mathbf{c}(t) = [\mathbf{C}(t),\mathbf{I}(t)]$ at time t, and gives a prediction output $\mathbf{a'}(t) = [\mathbf{C'}(t),\mathbf{I'}(t)]$, which is expected to be identical to $\mathbf{a}(t + 1) = [\mathbf{C}(t + 1),\mathbf{I}(t + 1)]$ 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 INNN 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.

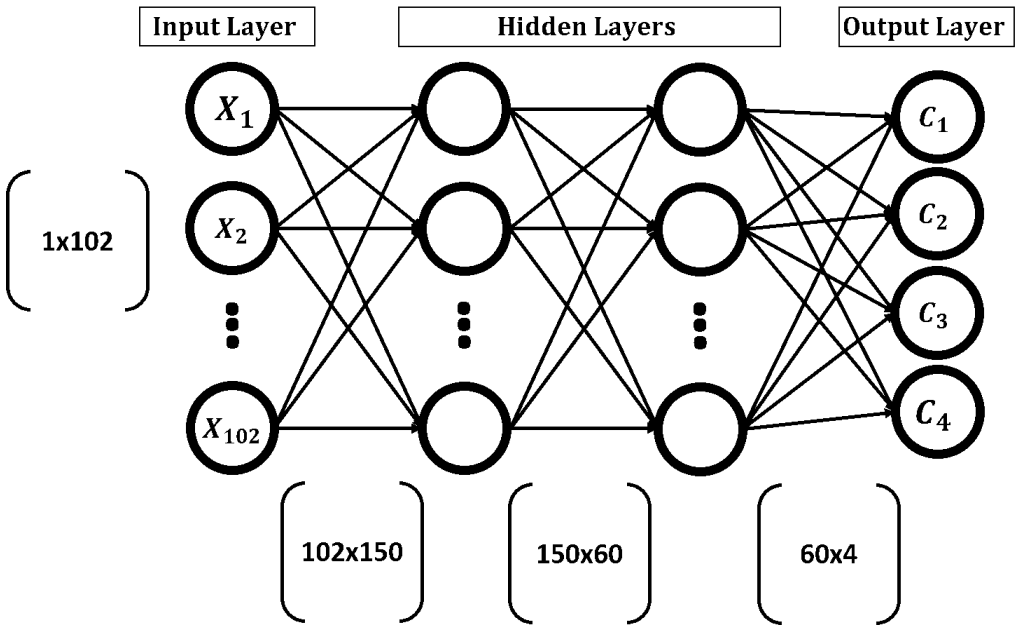
The LSTM layer serves as working memory, that takes the concatenated input [**C**,**I**] from both code and image subsystems, and output the predicted next component representation that could be fed back into both subsystems to form a guided loop.

Optimizing Artificial Neural Network using Genetic Algorithm

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.

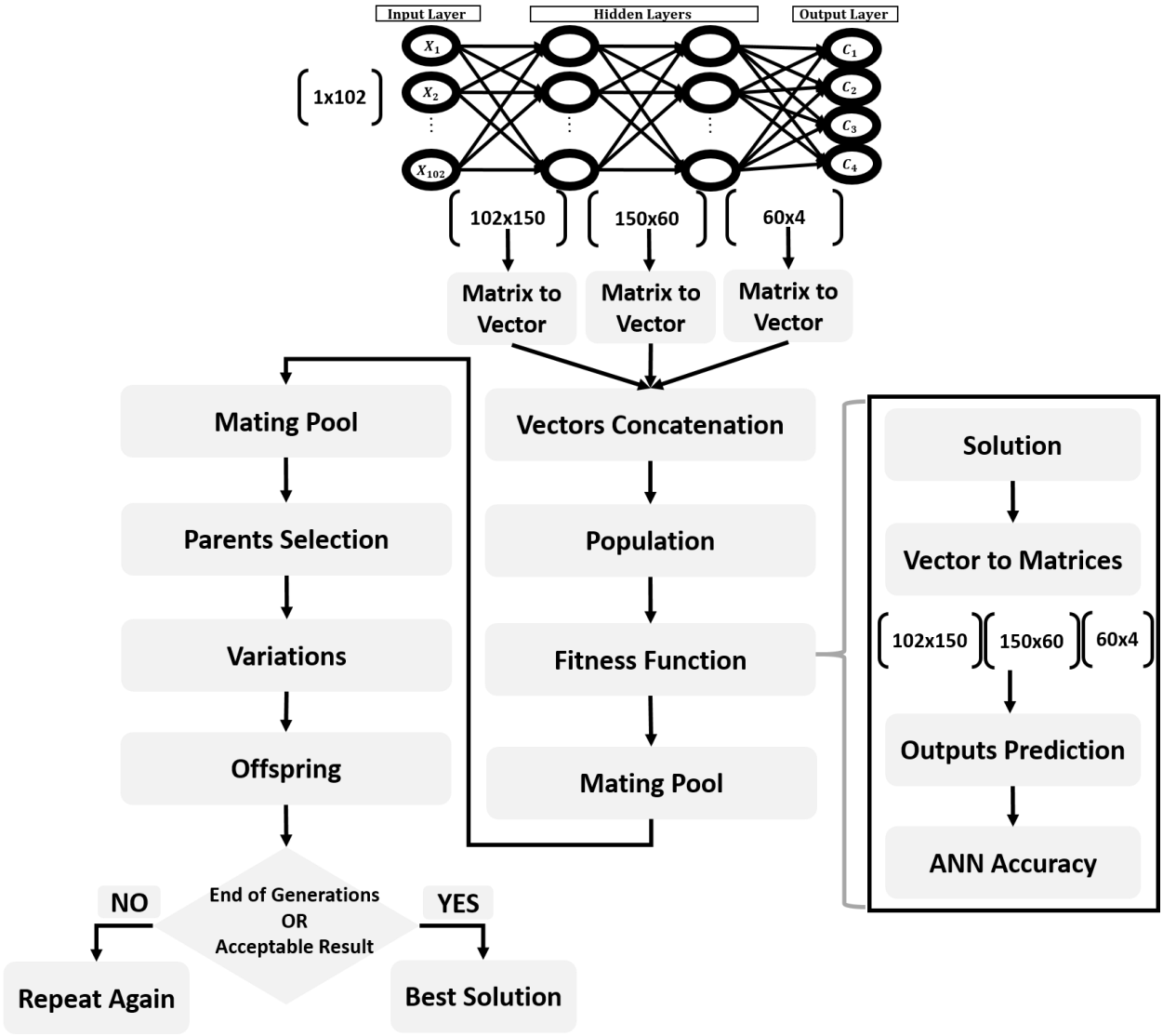
Using GA with ANN

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.

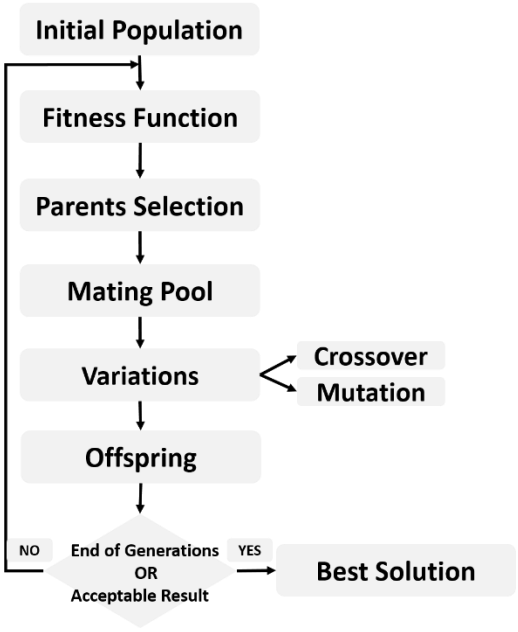


Weights Matrices to 1D Vector

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 returns 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

GA-ANN

Based on 100 generations, and using visualization library that shows how the accuracy changes across each generation. It is observed that after 100 iterations, On the MNIST dataset, we are able to find an accuracy that is

more than 70%. This is compared to 35% 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 1,000 generations might increase the accuracy.

CONCLUSION

In this paper, we have described two methods of building and training a self-replicating neural network through natural selection process. Firstly, we proposed to integrate new neural networks to achieve self-replication in a neural network as a method of creating identical copies of output. Secondly, we also proposed hybrid genetic algorithm artificial neural network (GA-ANN) predictive model as an optimization approach that mimics the concept of natural evolution / natural selection. This allowed us to create a neural network which optimizes its own weights. The test results are encouraging with good accuracy.

REFERENCES

1. Oscar Chang and Hod Lipson “Neural Network Quine” https://arxiv.org/pdf/1803.05859.pdf
2. Poondru Prithvinath Reddy “ Artificial Intelligence that Learn to Write Code : Memory Guided Programming” Google Scholar
3. Ahmed Gad “ Artificial Neural Networks Optimization using Genetic Algorithm with Python” Towards Data Science