Genetic evolution of neural networks

Charles-Henri Gros
Masters Student
Computer Science Department
Stanford University

ABSTRACT

The purpose of this project will be to study several ways to evolve neural networks. I will compare several methods of creating neural networks using GP. The function set in the GP will be used to create a directed graph containing the inputs, the output and any number of intermediate nodes. As a test, I will use the parity function (which is a good test since changing a single input will always change the result). I will then compare the performance of the neural nets to that of functions evolved with regular GP.

1 Introduction

Neural nets are a way of doing computation. It is usually used for tasks where the function is not entirely known, and needs to be approximated from sample data. However neural networks are very complex and difficult to program directly. But despite this complexity, their structure usually has to be entered by hand.

Genetic programming is a kind of search whose main goal is to focus on the performance of the objects it tries to create, instead of their components or structure. They are therefore well suited for the creation of neural networks.

2 Neural Networks

Neural networks are ways to define functions inspired by the human brain. They consist of a set of *neurons*, including *input* neurons and *output* neurons. Each neuron is connected to one or several others, in a directed fashion. Each neuron has an *activation value* and an *activation function*, and each link has a *propagation coefficient*. All the values are real numbers, either positive or negative.

The way neural networks act as functions is the following: The input is entered as the input neurons' activation value. All other neurons have an initial activation value of 0.

Then there is a number of propagation rounds, in which the activation value of each neuron is changed according to the weighted sum of its predecessors' activation functions' results, with regard to the weights on the links. The activation function's result is computed by applying the function to the activation value.

The output of the network is the activation value of the output neurons after a specified number of rounds, or until convergence.

The most common type of neural network is a *layered* network (see Figure 1) In these there is an input layer, an output layer and a number of intermediate layers, consisting of *hidden nodes*, and placed between the input layer and the output layer. Each neuron may only have edges to neurons in the layer directly after theirs.

The advantage of this method is that, if the number of rounds is set to the number of layers minus one, the evaluation time is linear in the number of neurons: it is possible to evaluate the activation levels one layer at a time. It also avoids the need to consider the previous activation level of each neuron, since it's always zero.

One of the most common type of activation functions is a threshold function. This function, for a given threshold, returns 0 if the activation level is below this threshold, and 1 if above. This is roughly what happens in real neurons, which send a signal through their axon if they reach a certain excitation level.

In this example, I decided not to use layered networks, since their expressiveness is limited, and they are hard to build with genetic programming. I used the threshold activation function, with a threshold of 1. Also, since the network

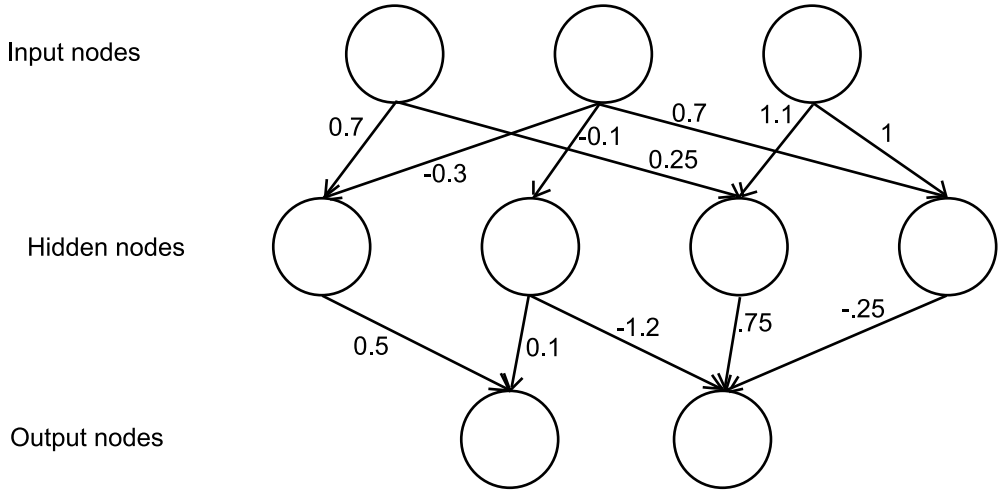

Figure 1: 3-layer neural network

is not layered, to deal with the activation level of previous rounds, I used an exponentially decreasing influence: the previous activation level is divided by 2, and added to the external influence. The activation level of inputs is only set at the first round, after which it evolves like any other neuron's. The number of propagation rounds is set to be the longest acyclic path from any input to the output.

The inputs used are the bits of the input to parity, plus an additional always-on input. Set bits give an initial activation level of 1, while cleared ones leave it to 0. The output of the network is the activation function of the output node.

3 Genetic programming

Genetic programming is a way to conduct search among a set of trees, which are usually used to represent functions.

The trees consist of nodes, which correspond to *functions*, and leaves, which correspond to *terminals*. A node corresponding to a given function always has a predetermined number of children, which represent the function's arguments. Terminals are actually just functions with no arguments (which may be constants, or return some state).

The goal is to find a tree that optimizes a given function which is called the *fitness measure*.

The search is conducted in the following way: First, a predetermined number of random *individuals* is created, forming the first *generation*. Then they are evaluated according to the fitness measure.

From these individuals, a new generation, consisting of the same number of individuals, is created by picking some the best ones (*selection*), and then mixing them together (by exchanging compatible subtrees). The latter operation is called *crossover*.

Another operation that may happen between generations is *mutation*, which randomly changes a subtree of an individual. It is usually used rarely, as it is akin to a random search.

This process is repeated for a predetermined number of generations, or until an ideal individual is found.

There are lots of ways to conduct selection, one of which is *fitness proportionate* selection, in which the probability of an individual being picked to create the next generation is proportional to its fitness.

Another popular one is *tournament* selection, where, to pick an individual from a generation to the next, a small number of individuals is selected uniformly at random from the previous, and the best one is kept. This is the method I used, with a size of 7 individuals.

Table 1: Random graphs with initial random constants

Objective	Find a neural network that produces the same values as the parity function
Terminal set	`Stop`
Function set	`LinkTo`, `LinkFrom`
Fitness cases	Output value on all numbers from 0 to 2^{n-1}
Fitness	Number of hits minus a thousandth the size of the network
Hits	Number of points at which the network successfully computes parity
Wrapper	Computation of the network from the description given by the functions
Parameters	M = 5000 ; G = 50
Success predicate	Fully matches parity (2^n hits)

4 First method: random graphs

The first method I used to create neural networks was with functions allowing to create random directed graphs.

In this method, the functions allow to create edges between an existing node and either another existing one or a new one, with a random propagation coefficient.

More precisely, there are three functions: `LinkFrom`, `LinkTo` and `Stop`. The `Link` functions are assigned a random coefficient (from -10 to 10) and a random integer (from 0 to 10000) when the node is created.

During the construction of the graph, a current node is kept, which allows to specify one end of each link added. This ensures that the graph stay connected.

`LinkFrom` creates a link from the current node to the one specified by the integer. To restrict the range of the integer, it is taken modulo the current number of neurons, plus one (to allow adding a new neuron). The coefficient is used as the propagation coefficient of the link. If this link already exists, the coefficients are added. Then the current node is set to the destination of the link.

`LinkTo` does the same thing, except that it creates an edge in the reverse direction.

To create a random number of links, both `Link` functions have two children. The `Stop` function is used as a terminal, and doesn't do anything.

The GP evolves one tree for each input (which is set as the current node when each tree is evaluated). This has the advantage of forcing each input to be used.

The initial network consists of each of the inputs, as well as the output, with no link. They are all available as parameter of the `Link` functions, i.e. they can be reached after the modulo operation.

The fitness measure is the number of hits (inputs for which the output of the network matched the output of the parity function), minus a small coefficient times the size (number of neurons) of the network (for an equal number of hits, this favors smaller networks)

I started with a population size of 3000 and 150 generations, but most of the times the number of hits stopped progressing after a relatively short number of generations, so I changed it to 5000 individuals and 50 generations.

This technique gives surprisingly good results, as I was able to get a perfect fit for 5-way parity in about 30 generations (Figure 2).

5 Second method: DAGs

The previous method worked rather well, but I wanted to do better. For that, I decided to concentrate on acyclic graphs, since back-feed (cycles in the graph) is not necessary for something as simple as parity. I also wanted to make subgraphs more stable after crossovers.

Also, the fact that all coefficients were created at the first generation was a problem, since the available coefficient pool was only decreasing generation after generation, so I decided to replace them with genetically evolved ones.

Lastly, for simplicity, I replaced one tree per input with a single tree representing the output.

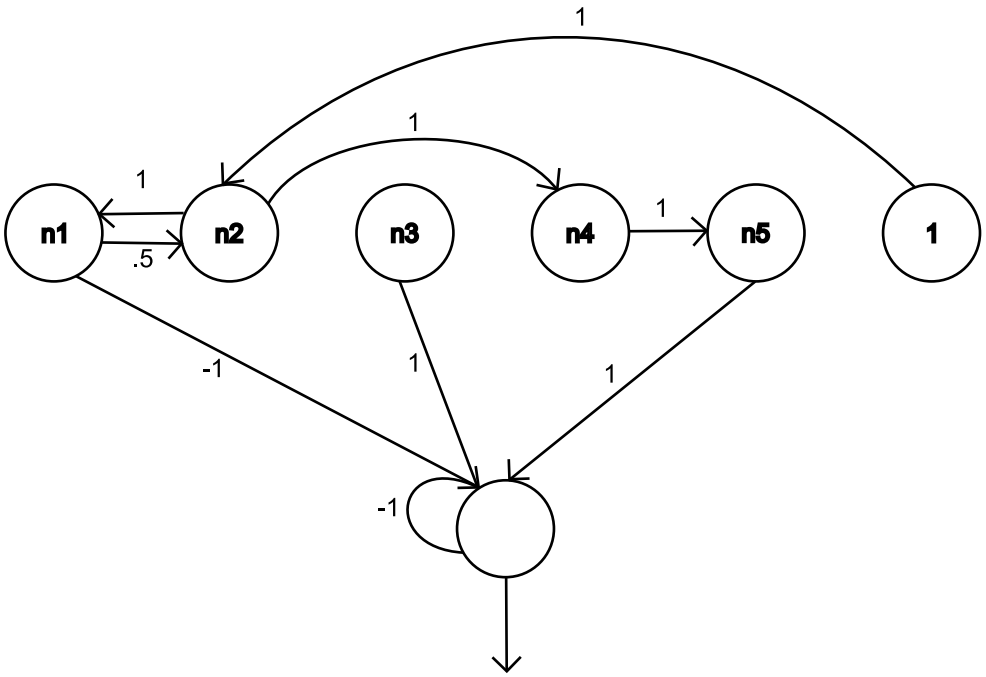

Figure 2: Solution to 5-parity

The new method uses typed genetic programming, with two types of values, *numbers* and *graph functions*. The new function set can be divided into two subsets:

1. Functions on numbers: `Add`, `Neg`, `X2`, `D2` and `Cons`. `Add` takes two numbers as children, and evaluates the addition of these two's values. `Neg` has one child number, and represents the negation of the child's value. `X2` and `D2` also have one child number, and represent respectively multiplication and division by 2. Lastly, `Cons` is a terminal representing a constant. I used 1 and -1 as constants. These functions allow to create numbers biased toward 1.

2. Graph creation functions: `Input`, `Link`, and `Fork`. `Input` represents the input nodes. One such terminal is used for each of the inputs. `Link` creates a link to the current node from another. This time, it is no more acceptable to limit it to either an existing one or a single new one. To maintain a DAG structure, the target node has to be ranked higher than the current one. To do that, and also to allow more stability of structure during crossover operations, the target is specified relatively to the current node, as a strictly positive number. To maintain connectivity as much as possible, while keeping expressiveness, this number is chosen with a probability distribution favoring small numbers. More precisely, a real number d is chosen uniformly at random between 0 and 1, and then the number is taken to be the integer part of $-\log(1-d)$. Since a node with only one incoming edge is practically useless, each `Link` has two children, whose destination will be the origin of this link. Lastly, `Fork` is used to allow a random number of inputs to a given node: it has two children, and before each one resets the current node to the one it was at the beginning of its evaluation.

Alas, this technique didn't work as expected; it actually performed much worse than the previous one: I was never able to get a perfect individual for more than 2 inputs!

I first blamed this lack of results as a consequence of using only one tree, since any functions that doesn't use all the inputs matches exactly half the cases of parity. So I went back to one tree per input, but this actually didn't change anything: the fitness stopped progressing after a relatively short number of generations.

Then I thought of another possible source of bad results: the destination of links is fixed at the first generation, so variety is lost generation after generation. So I changed that by adding yet another type of functions, which generated integer numbers.

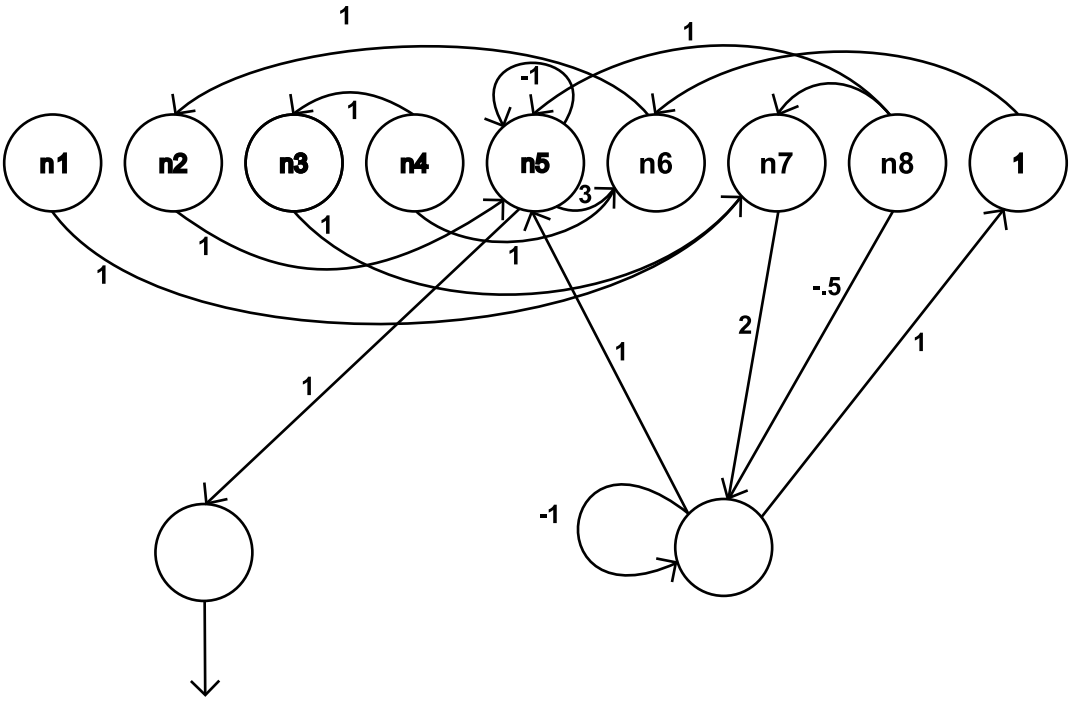

Figure 3: Solution to 8-parity

This improved the results a bit, as I was usually able to get a higher number of hits after some generations, but I still couldn't get any perfect individual, even for very low arity.

All the same, augmenting the population size didn't really help.

After looking more closely at the result of the first method, I realized that restricting the networks to DAGs is actually a bad idea: although they can solve parity, they're far from being as concise as the result I got from the random graph method, which makes a lot of uses of back feed.

6 Third method: Random networks with relative links

To try and combine the features of the last two methods, I decided to go back to random networks. But since I wanted to keep the structural stability, I kept the idea of relative links; except this time they're allowed to go back. To try not to favor some inputs with regard to the others, I added terminals for both the inputs and the output. However, to still allow links among inputs, and since the corresponding functions are terminals, the relative links are still allowed to land on inputs or output.

This method worked much better, with 5-parity solved in as little as 10 generations, and 8-parity in less than 30 generations (Figure 3).

The results are also surprisingly concise, with only one node beside the given input and output nodes, and no more than a few outgoing links per node.

7 Fourth method: ADFs

Since parity is very regular, and relative links allow structural stability, I tried to see if I could use ADFs (automatically defined functions).

So I added one tree to the set, which represents a two-argument function. I added this function to the other trees' function set, and the ADF's function set was set to consist of the `Link`, `Fork` and `Nil` functions, as well as terminals

representing its arguments, and of course the numeric functions required by `Link` for its coefficients.

However this didn't improve the search, as in most searches, the ADF tree quickly became just a node with one of its arguments, rendering it useless.

This isn't too surprising either, considering how on the networks found as result by the previous methods, there are almost no hidden nodes, which are the only ones which can be used inside of an ADF.

8 Fifth method: separate links and nodes

Since ADFs didn't work when links and their destination were combined, because it only allowed ADFs to create links going through hidden nodes, I decided to make separate functions for links and nodes.

So I separated the graph function type into two types, *link* and *node*.

Also, I allowed links taking two nodes (not necessarily preexisting) as parameters. This may allow for unconnected graphs, but the network size part of the fitness should eliminate them quickly.

The new function set included the previous functions on numbers, and the following functions on graphs:

1. `Link`, which takes as parameter a node and a number, and creates a link from the current node to the parameter node, with the parameter number as coefficient, in a predetermined random direction;
2. `Link2`, which takes as parameter two nodes and a number, and creates a link between these nodes (both of which are evaluated with the same current node), with, as in the case of `Link`, the parameter number as coefficient, and a predetermined random direction;
3. `Fork`, which takes as parameters two links, which are set to have the same current node;
4. `Node`, which takes as parameter a link, and returns a node; it is specified relatively to the previous current node (as in the previous construction methods), and becomes the current node for the child link;
5. `Input` and `Output`, which are terminals of type node, that represent the input and output nodes;
6. `Stop`, which is a terminal of type link, which does nothing (the software I used actually requires terminals for each type).

But, as many other methods I tried, this didn't work well. I was still able to find a solution for 4-parity, but it was much bigger than even the solution to 8-parity found by the previous method.

This is probably because separating the links and the nodes somehow dissolves the structure, and not enough of it is kept during crossovers.

9 Comparison: parity with logic operators

To compare the efficiency of neural networks, I made a quick experiment to evolve simple propositional logic expressions for parity, using only `OR`, `AND` and `NOT` operators. I also added a 2 argument ADF.

As could be expected, this works better, with 8-parity solved in less than 10 generations. However it is not that much better, and boolean functions are much more limited than neural networks, especially for cases where the values for the functions are not entirely known, in which case neural networks are much better at generalizing training data than boolean functions.

10 Conclusion

Neural networks are very difficult to code by hand. It is therefore almost always better to let them learn in some way. This is especially true of back-feed neural networks.

Although it is very difficult to find a good method to evolve working neural networks, genetic programming allows to build very concise networks to solve the very sensitive parity problem, without needing to give any constraints to the structure of the network, providing the "hands free" method which is the greatest strength of genetic programming.

Although this result is only valid for a toy problem, it still shows the versatility of neural networks combined with genetic programming. It may be reasonable to think that it could solve some much more complicated problem, where the generalization capability of neural networks could be put to use.