Can a neural network manipulate symbols and perform calculations?

Darius Barbano

Abstract

Abstract here.

1 Preface

1. What is symbolic computation?

   A symbolic computation is a calculation performed with symbolic representations of values and operations. A simple example would be the expression $(x+1)(x-1)$ which would evaluate to x^2-1, rather than to some numerical result.

2. What is calculation?

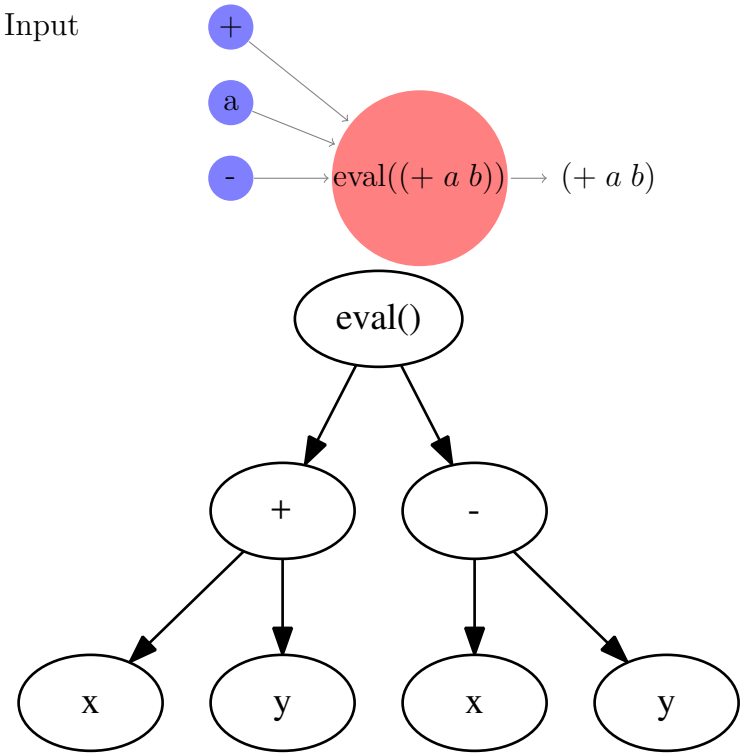
   A calculation is a process by which one or more inputs is transformed into one or more results. One may calculate that the product of 5 and 4 is 20.

3. What is meant by a computational class? - do you mean complexity class?

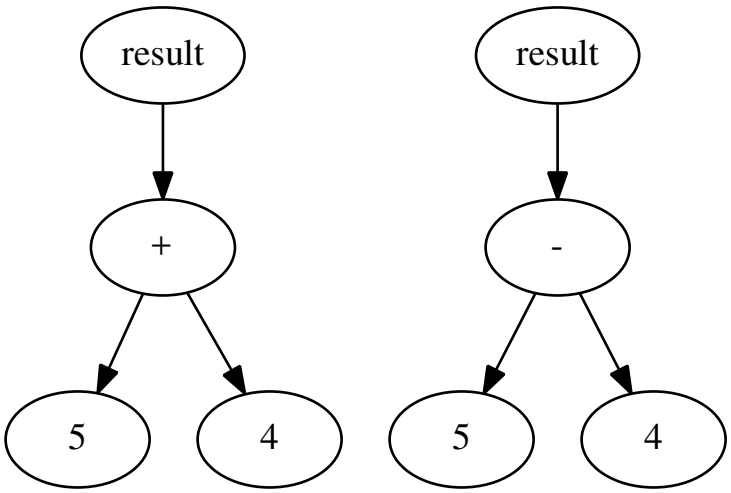
Code for each subsection of this document can be found at `https://github.com/DariusBxsci/NeuralNetworkResearch/tree/master/NeuralNets`.

2 Simple Addition/Subtraction Network

Consider the construction of a neural network to alternately add and subtract numbers. It is useful to represent all binary operations in list form, $(+\ a\ b)$. The neural network, then, becomes a representation of a stack.



This tree-like structure closely mimics the abstract syntax tree which is generated by compilers when processing code in a program. For example, to evaluate the results of the expressions "+ 5 4" and "- 5 4", a compiler may generate the following trees:



To improve this sentence don't use *weasel words*. I don't know what *very similar* means to you. Neither a compiler nor interpreter *view* code. Neither are (yet) sentient beings.

The best piece of writing advice I can give is:

> Mean exactly what you write and write exactly what you mean.

3 Addition/Subtraction Network with input permutations

In the previous network, inputs were configured such that the operation to perform was the first element in the list, the first input the second element, and the second input the third. We represented $a + b$ as $[+, a, b]$, and $a - b$ as $[-, a, b]$.

Suppose we want to evaluate the expression $a + b + c - d$ in the same way that was done in the last network. Since the expression is evaluated using prefix notation, using this expression as it is as input would result in an error. Instead, we must transform the expression into this:

$$- + +abcd$$

Keep in mind, though, that the expression which would be passed into the eval() function would look like this:

$$(-(+(+ab)c)d)$$

since the LISP interpreter is incapable of inferring the order of evaluation of the expression without the help of parenteses.

First, let us clearly define the desired transformation

$$[a, +, b, +, c, -, d] \rightarrow [-, +, +, a, b, c, d]$$

Which we may represent more generally as

$$[e_1, e_2, e_3, e_4, e_5, e_6, e_7] \rightarrow [e_6, e_4, e_2, e_1, e_3, e_5, e_7]$$

or by a seventh order permutation matrix:

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

Now we will generalize this permutation matrix so that we can generate one every time we need to reformat a simple mathematical expression into prefix notation.

Consider another transformation

$$[a, +, b, -, c] \rightarrow [-, +, a, b, c]$$

We can express it as

$$[e_1, e_2, e_3, e_4, e_5] \rightarrow [e_4, e_2, e_1, e_3, e_5]$$

Let us now define a function which, given an input of an element's position in the first expression, will output it's position in the resulting expression

$$f(p) = (n + \frac{n-1}{2}) + \sum_{i=1}^{n} (-1)^p * p$$

Where n is the number of elements in the input vector.

We can apply this function to the expression to get

$$[e_1, e_2, e_3, e_4, e_5] \rightarrow [f(1), f(2), f(3), f(4), f(5)] \rightarrow [3, 2, 4, 1, 5]$$

Finally, we can define our permutation matrix of dimensions $n * n$ such that each element $a_{r,c}$ is zero except for those where $f(r) = c$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,5} \\ a_{2,1} & a_{2,2} & \dots & a_{2,5} \\ a_{3,1} & a_{3,2} & \dots & a_{3,5} \\ a_{4,1} & a_{4,2} & \dots & a_{4,5} \\ a_{5,1} & a_{5,2} & \dots & a_{5,5} \end{bmatrix}^T$$

$$a_{r,c} = \begin{cases} 0, & f(r) \neq c \\ 1, & f(r) = c \end{cases}$$

When we use this procedure to generate a permutation matrix of size $n = 5$, we get

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

When we apply P to the expression $[a, +, b, -, c]$, we get $[-, +, a, b, c]$
To further confirm that our algorithm works, lets generate a permutation matrix of size $n = 7$

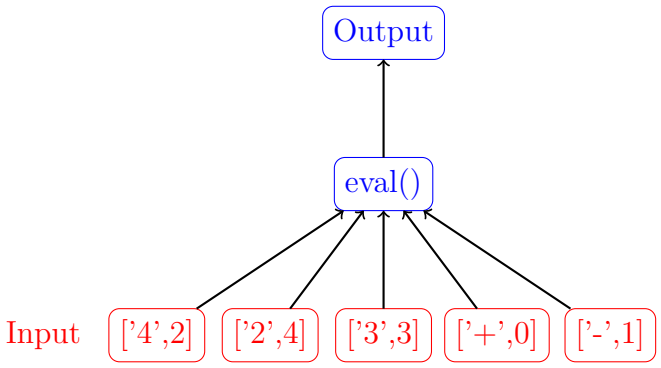
$$P = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Which is identical to the matrix we got for the first transformation.

When we apply this matrix to the expression $[a, +, b, +, c, -, d]$, sure enough, we get $[-, +, +, a, b, c, d]$

4 'Lollipop' Model of a Neuron

When constructing Neural Networks, we generally consider the order of inputs to neurons or the order of inputs in a layer, to be irrelivent. However, there is no reason that the order of inputs should be self-evident, and in many cases, the best order of inputs may not be known at all. To examine this, let us create a simple model for a single neuron, where it accepts any number of inputs and has a single output which is the result of applying the 'eval' function to the inputs.



Each input is a tuple where the first index is either an operator or operand, and the second index contains the input's position in the eval function. When the neuron's value is evaluated, all inputs are positioned according to their second index's value. If the tuples in the above diagram were passed into the neuron, it would result in 'eval(+ - 4 3 2)', or simply '3'.

What if we don't know the correct position for the operator and operands?
Typically, the order of inputs into a neural network isn't given much consideration, particularly when tuning the neurons to a yield a desired result. In this section, we will design a simple algorithm to tune the second index of each input tuple (the index which represents the input's position in eval()) using only the unordered inputs and a few output examples given the inputs.

Let's examine the above diagram, but eliminate all information which requires us to know the order of inputs, aside from the output: