Neural Networks and Their Theoretical Capabilities

Overview of Neural Networks

A neural network is a system in which many similar functions, called *neurons*, are composed together in order to classify inputs, perform some computation, or approximate some function. Neural networks are used today in various machine-learning applications such as handwriting- or speech-recognition, and have several interesting mathematical properties.

This paper will introduce two mathematical models of neurons, the basic workings of neural networks, and then will proceed to examine their theoretical capabilities. First, the ability of certain neurons to compute various boolean functions will be demonstrated. This leads to the intuition that recurrent neural networks, are Turing-complete, which will then be proved.

Definition of a Neuron

The idea for neural networks is borrowed from the field of biology; the human brain uses cells called neurons for cognition. Neurons are joined with one another via connections called synapses. Neurons create voltages based upon the voltages on certain synapses that act as inputs. Thus when the collective voltage on the input synapses exceeds a certain level the neuron becomes excited and produces a higher voltage on the output synapses.

A neuron is modeled mathematically as a function $f(x_1, x_2, \ldots, x_n) = \phi(\Sigma_{i=1}^{n} w_i x_i)$, where $\{w_1, w_2, \ldots\ w_n\}$ is a set of weights, with each weight w_i corresponding to an input x_i, and where ϕ is the activation function that determines the output of the neuron based on the sum of the products of the weights and inputs. For the sake of brevity we will also notate the weight and input vectors as $\vec{w}$ and $\vec{x}$, respectively. $\Sigma_{i=1}^{n} w_i x_i$ calculates the collective weighted input; this total is then passed to the activation function ϕ which uses the collective weighted input to generate an output.

A neuron could be also thought of more generally as a partial application of ϕ and $\vec{w}$ over the function $F(\phi, \vec{w}, \vec{x})$. Such a definition may be useful for implementation of a neural network in software.

There are two popular activation functions, the perceptron and the sigmoid.

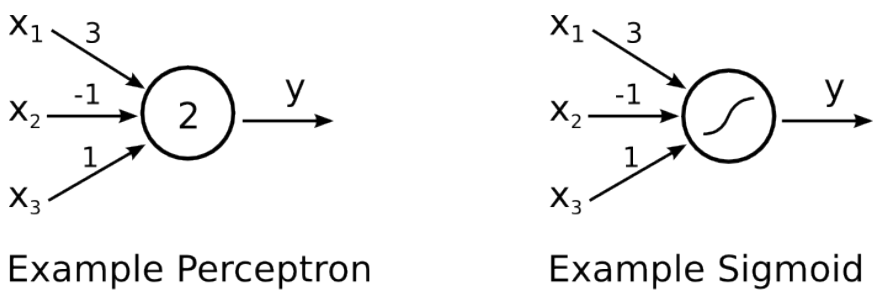

Figure 1: Example perceptron and sigmoid neurons

Having been invented in 1957 by Frank Rosenblatt, the perceptron was among the first activation functions to be implemented. It can be used as a standalone classifier on linearly separable data. The perceptron can be defined as the function

$$\phi_P(x) = \begin{cases} 0 & : x + b \leq 0 \\ 1 & : x + b > 0 \end{cases} \tag{1}$$

where b is a bias that is specific to the neuron. Notice that the bias b of the perceptron is sometimes denoted by a number written inside neuron if displayed in an image.

The sigmoid function can be defined as

$$\phi_S(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Though bearing some resemblance to the perceptron, the sigmoid function has the advantage of being smooth and thus differentiable. These properties make training a neural network composed of sigmoid neurons much easier then a similar network composed of perceptron neurons.

Example. *Consider the neurons in Figure 1. Note that Let* $x_1 = 3, x_2 = 2, x_3 = -0.5$. *Both of the neurons will have the same aggregate input passed to their activation functions. To determine the aggregate input, multiply the inputs by their respective weights and then sum the products:* $w_1 * x_1 + w_2 * x_2 + w_3 * x_3 = 3 * 3 + -1 * 2 + 1 * -0.5 = 6.5$ *The perceptron will output* 1 *since* $\phi(6.5) = 1$ *(see Equation 1). The sigmoid will then output* $\sigma(6.5) = \frac{1}{1+e^{-6.5}} \approx 0.9985$.

Construction of Neural Networks

More complex behavior can be created by linking the outputs of some neurons into the inputs of other neurons. This is how the human brain is constructed. An example of a neural network is shown in Figure 2. The network pictured is considered

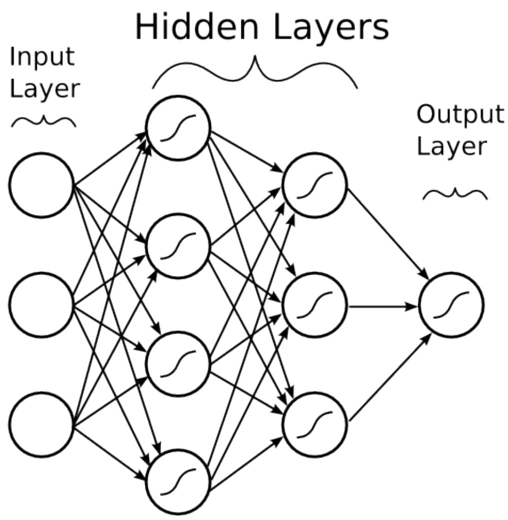

Figure 2: A four-layer neural network of sigmoid neurons.

a four-layer sigmoid network because each layer besides the input layer consists of sigmoid neurons as denoted by the sigmoid shape inscribed in the representation of each neuron. The input layer does not contain actual neurons but rather is drawn by convention to show the inputs being fed to the second layer. The hidden layers encompass all of the neurons between the input layer and the output layer.

The output layer contains the neurons whose output is interpreted as the output of the network. In the figure shown the output layer only contains one output neuron. It is possible to have more than one output from a neural network; for example, a binary number could be encoded in the outputs.

If no neuron takes as input an output that the neuron affected (i.e. there are no cycles in the graph) then the network is called a *feed-forward* network. If loops do exist then the network is referred to as *recurrent*.

Demonstration of the Equivalence of Various Perceptron Networks to Certain Boolean Logic Functions

Neural networks can be used to compute boolean logic functions. In this section the construction of various logic gates (devices that compute logical functions) from perceptron networks will be demonstrated. Since traditional computers are constructed from logic gates, predominantly NAND[1] gates, it stands to reason that a neural network could be constructed to simulate a traditional computer;[2] intuitively this leads to the hypothesis that recurrent neural networks are Turing-complete. This will be proved in the section following the current one.

Consider the following network consisting of one perceptron-type neuron, with inputs $\binom{x_1}{x_2}$.
Let $\vec{w} = \binom{1}{1}$ and $b = -1$. Compare the behavior of the network to that of the AND function:

x_1	x_2	$\vec{x} \cdot \vec{w}$	$\vec{x} \cdot \vec{w} + b$	Output	x_1 AND x_2
0	0	0	-1	0	0
0	1	1	0	0	0
1	0	1	0	0	0
1	1	2	1	1	1

Note how the network is equivalent to the AND function over the inputs for which AND is defined.

Now let $\vec{w} = \binom{1}{1}$ and $b = 0$. Compare the behavior of the network to that of the OR function:

x_1	x_2	$\vec{x} \cdot \vec{w}$	$\vec{x} \cdot \vec{w} + b$	Output	x_1 OR x_2
0	0	0	0	0	0
0	1	1	1	1	1
1	0	1	1	1	1
1	1	2	2	1	1

Now, simply by changing the weights, the network output becomes equivalent to the OR function over the domain of OR.

Let's also consider NAND. Now let $\vec{w} = \binom{-1}{-1}$ and $b = 2$. Compare the behavior of the network to that of the NAND function:

[1] NAND is the composition of NOT and AND.

[2] It should be noted, however, that most implementations of neural networks do just the opposite: neurons are simulated in software. This is due to the ease of construction of digital circuitry i.e. transistors over something that requires analog signals like a sigmoid neuron.

x_1	x_2	$\vec{x} \cdot \vec{w}$	$\vec{x} \cdot \vec{w} + b$	Output	x_1 NAND x_2
0	0	0	2	1	1
0	1	-1	1	1	1
1	0	-1	1	1	1
1	1	-2	0	0	0

Thus a single two-input perceptron can compute NAND.

Computer scientists met with some difficulty when attempting to weight a single perceptron to compute the exclusive-or (XOR) function. In fact, it is impossible to weight a single perceptron to do so.

Proof. Let P be a perceptron with two inputs x_1 and x_2, with associated weights w_1 and w_2, respectively, and a bias b. Assume for the sake of contradiction that P properly computes the XOR function. This implies the following behavior:

	x_1	x_2	Output $\begin{cases} 0 & : w_1x_1 + w_2x_2 + b \leq 0 \\ 1 & : w_1x_1 + w_2x_2 + b > 0 \end{cases}$
1)	0	0	0
2)	0	1	1
3)	1	0	1
4)	1	1	0

Then the following must be true:
Line (1) implies that $b \leq 0$.
Line (2) implies that $w_2 + b > 0$.
Line (3) implies that $w_1 + b > 0$.
Line (4) implies that $w_1 + w_2 + b \leq 0$.

Lines (2) and (3) imply that $w_1 > -b$ and $w_2 > -b$.
Then $w_1 + w_2 + b > (-b) + (-b) + b = -b \geq 0$.
Combining this with line (4) implies that $w_1 + w_2 + b = 0$.
Adding lines (2) and (3) yield $w_1 + b + w_2 + b = w_1 + w_2 + b + b > 0$.
Then the previous result implies that $w_1 + w_2 + b + b = 0 + b > 0$.
This contradicts line (1).
Therefore the assumption is false and no perceptron can compute XOR.

□

However, a multilayer perceptron network can be designed to compute XOR. One way to demonstrate this is by constructing a network of perceptrons weighted to compute NAND. Then XOR can be computed because $\{NAND\}$ is functionally complete; that is, all boolean functions can be expressed as a composition of NAND.

Definition 1. Let S be a set of truth functions. Then S is functionally complete iff all possible truth functions are definable from S[3].

The following proof borrows heavily from ProofWiki[4].

Proof. Let $\{0, 1\}$ be the set of truth values, with 0 signifying a false value and 1 signifying a true value. Consider the following truth table of binary boolean functions. On the left side are the two inputs x_1 and x_2. Because there are two inputs, each of which can assume two values, there are four possible inputs, yielding $2^4 = 16$ possible outputs. Thus the listing on the right side of the table of binary functions is exhaustive.

For purposes of readability the table has been broken horizontally into two halves.

x_1	x_2	f_F	AND	$(\neg \Rightarrow)$	pr_1	$(\neg \Leftarrow)$	pr_2	XOR	OR
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

x_1	x_2	NOR	$\Leftrightarrow$	$(\neg\mathrm{pr}_2)$	$\Leftarrow$	$(\neg\mathrm{pr}_1)$	$\Rightarrow$	NAND	f_T
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

The following table will demonstrate that the set $S_1 = \{\neg, \Rightarrow, \mathrm{AND}, \mathrm{OR}\}$ is functionally complete. Each possible binary boolean function not in S_1 is listed down the first column. An equivalent expression for each is function is written in the second column. The third column contains the set of boolean functions that have been used so far in the chart without having an equivalent expression given for them.

Function	Equivalent Expression	Functions Used So Far
f_T	$x_1 \Leftrightarrow x_2$	$\{\Leftrightarrow\}$
f_F	$x_1 \text{XOR} x_1$	$\{\Leftrightarrow, \text{XOR}\}$
XOR	$\neg(x_1 \Leftrightarrow x_2)$	$\{\Leftrightarrow, \neg\}$
$\Leftrightarrow$	$(x_1 \Rightarrow x_2)\text{AND}(x_1 \Leftarrow x_2)$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow\}$
$(\neg \text{pr}_1)$	$\neg \text{pr}_1$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow, \text{pr}_1\}$
$(\neg \text{pr}_2)$	$\neg \text{pr}_2$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow, \text{pr}_1, \text{pr}_2\}$
pr_1	$x_1 \text{AND} x_1$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow, \text{pr}_2\}$
pr_2	$x_2 \text{AND} x_2$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow\}$
NAND	$\neg(x_1 \text{AND} x_2)$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow\}$
NOR	$\neg(x_1 \text{OR} x_2)$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow, \text{OR}\}$
$(\neg \Rightarrow)$	$\neg \Rightarrow$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow, \text{OR}\}$
$(\neg \Leftarrow)$	$\neg \Leftarrow$	$\{\neg, \text{AND}, \Rightarrow, \Leftarrow, \text{OR}\}$
$\Leftarrow$	$x_2 \Rightarrow x_1$	$\{\neg, \text{AND}, \Rightarrow, \text{OR}\}$

Thus S_1 is functionally complete.

Now consider another table similar to the one above; however, in this table S_1 is shown to be definable from $S_2 = \{\text{NAND}\}$.

Function	Equivalent Expression	Functions Used So Far
$\Rightarrow$	$\neg(x_1 \text{AND} \neg x_2)$	$\{\neg, \text{AND}\}$
OR	$\neg(\neg x_1 \text{AND} \neg x_2)$	$\{\neg, \text{AND}\}$
$\neg$	$x_1 \text{NAND} x_1$	$\{\text{AND}, \text{NAND}\}$
AND	$(x_1 \text{NAND} x_2)\text{NAND}(x_1 \text{NAND} x_2)$	$\{\text{NAND}\}$

Therefore S_1 can be expressed from S_2. Since S_1 is functionally complete this implies that S_2 is functionally complete.

$\square$

See Figure 3 for the design of a neural network that computes XOR.

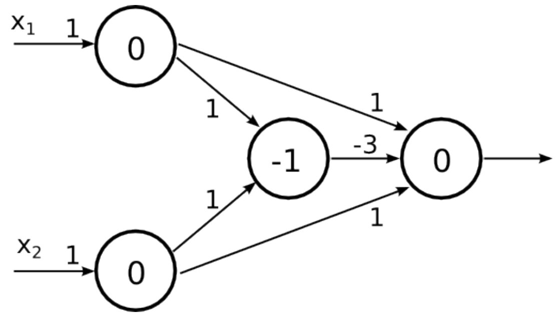

Figure 3: A neural network that computes XOR.

Statement and Proof of the Turing-completeness of Certain Recurrent Neural Networks

As was alluded to before, recurrent neural networks are Turing-complete.

The Turing machine is a conceptual machine proposed by Alan Turing to provide a definition for the idea of computability. The description of a Turing machince is based on Turing's 1936 paper[1].

The Turing machine has a finite number of conditions called m-configurations that it can be in. It has a "tape" of infinite length which is divided into "squares" containing "symbols" which can be read one at a time by a "reader". The machine can read and write symbols onto the square of the tape directly under the reader and can move the tape back and forth underneath the reader. In this manner the Turing machine can perform any computation; for a more detailed description and discussion see Turing's paper[1].

In lieu of expressing m-configurations and such in terms of neural networks we shall utilize a common strategy for proving Turing-completeness: implementing a language which has already been shown to be Turing-complete. If a language is Turing-complete, then it can express any algorithm since the Turing machine is universal. If we can implement the language using a recurrent neural network then it follows by transitivity that our network is also Turing-complete.

The following proof is largely based on a 1996 paper by Heikki Hyötyniemi[6].

Consider the following language $\mathbb{L}$, which consists of the following four instructions:

name	operation	description
inc	$V' \leftarrow V + 1$	increment
dec	$V' \leftarrow \max(0, V - 1)$	decrement
nop	$V' \leftarrow V$	no operation
goto	if $V \neq 0$ goto j	conditional branch

where $V, j \in \mathbb{Z}^+$.

It is known that $\mathbb{L}$ is Turing-complete[6]. Also note that the decrement operator precludes the possibility of negative variable values. This is due to the following activation function used for the purposes of this proof:

$$\phi(x) = \begin{cases} x & : x > 0 \\ 0 & : x = 0 \end{cases} \tag{3}$$

To elucidate the proof an example will be interspersed throughout the proof. The following program `example-prog` in $\mathbb{L}$ will be implemented using a neural network:

```
0: dec V0
1: inc V1
2: if V0 != 0 goto 0
```

The program has two variables, `V0` and `V1`. While `V0` $\neq 0$ the program loops.

Procedure for Creating Network

Given an input program P, create the following neurons:
For each variable i in P, create a variable neuron V_i.
For each line j in P, create a instruction neuron N_j.
For each `goto` in P, create two transition neurons N'_j and N''_j, given that j is the line number of the `goto`.

Then connect the neurons as follows:

- Create a connection with a weight of 1 between each variable neuron V_i and V_i. This feedback loop keeps the value of V_i consistent between iterations unless otherwise modified.
- Create a connection with a weight of 1 between each `inc` instruction neuron N_j and the variable V_i that is being incremented.
- Create a connection with a weight of -1 between each `dec` instruction neuron N_j and the variable V_i that is being decremented.
- Create a connection with a weight of 1 between each non-`goto` instruction neuron N_j and N_{j+1}.
- Create a connection with a weight of 1 between each `goto` instruction neuron N_j and the transition neurons $N_{j'}$ and $N_{j''}$.

- Create a connection with a weight of 1 between each $N_{j'}$ transition neuron and the variable neuron referenced by the N_j `goto`.
- Create a connection with a weight of -1 between each $N_{j''}$ transition neuron and the variable neuron referenced by the N_j `goto`.

Thus the aggregate input to each neuron is as follows:

- V_i has an aggregate input of

$$V_i + \Sigma_{j \in x} N_j - \Sigma_{j \in y} N_j + \Sigma_{j \in z} N_{j'} - \Sigma_{j \in z} N_{j''} \tag{4}$$

  where x is the set of `inc` neurons referencing that variable, y is the set of `dec` neurons referencing that variable, and z is the set of `goto` neurons refrencing that variable.

- N_j has an aggregate input of

$$N_{j-1} + \Sigma_{l \in z} N_{l'} - Nl'' \tag{5}$$

  where z is the set of `goto` neurons referencing that instruction neuron. If N_{j-1} does not exist then that term is zero.

- $N_{j'}$ has an aggregate input of

$$N_j \tag{6}$$

- $N_{j''}$ has an aggregate input of

$$N_j - V \tag{7}$$

  where V is the variable referenced by the `goto`.

A realization of `example-prog` is given in Figure 4. The output of the network is run to the bus on the right side of the diagram, through the unit delay, and fed back into the network on the left side. For example, the output of N_0 runs to position 0 on the bus, which is connected to N_1. The network is said to *iterate* each time the network output is run through the unit delay. The table below gives the value of each position at each iteration. Note that the 0^{th} iteration contains the initial settings of the network. Also note that the box labeled "Initial State" outputs 1 during the calculation of the first iteration and outputs 0 afterward.

	Iteration									
Position on bus	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	1	0	0	0	0
1	2	1	1	1	1	0	0	0	0	0
2	0	0	1	0	0	0	1	0	0	0
3	0	0	1	1	1	1	2	2	2	2
4	0	0	0	1	0	0	0	1	0	0
5	0	0	0	0	1	0	0	0	1	0
6	0	0	0	0	0	0	0	0	1	0

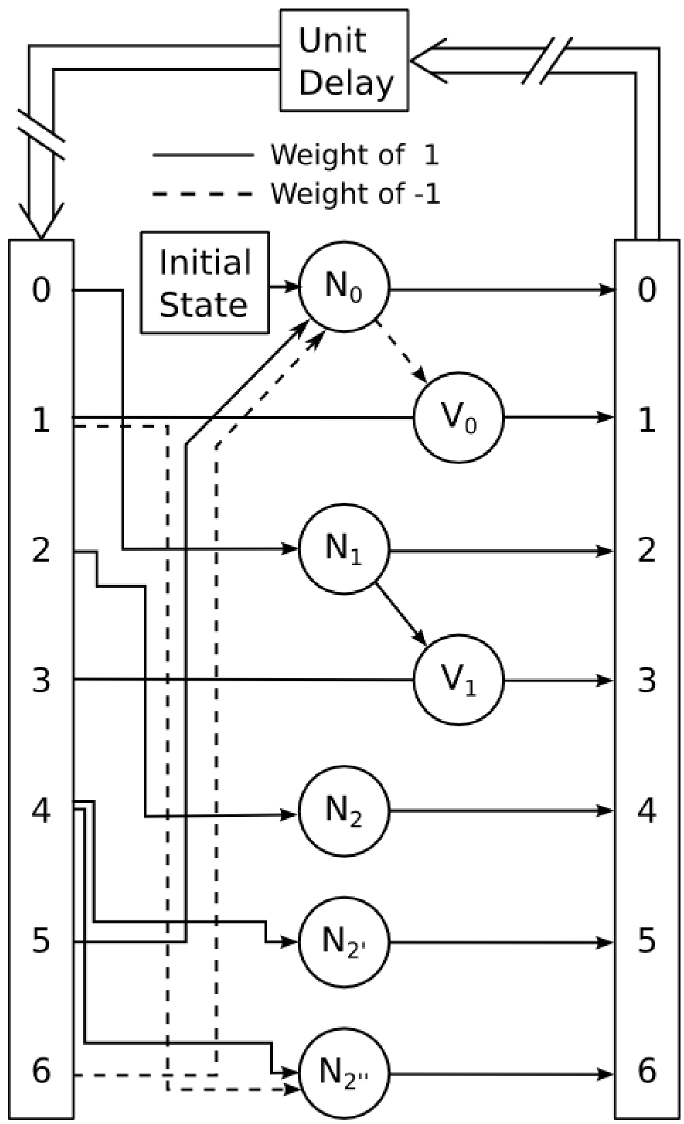

Figure 4: The neural network for `example-prog`.

Note that all the iterations but the 0^{th} and 9^{th} share some common properties which make them *legal states*.

Definition 2. A neural network's state is *legal* provided that the following conditions hold:
- At most one instruction neuron N_j has an output of 1, the others output 0.
- All transition neuron $N_{j'}$ and $N_{j''}$ have an output of 0.

The network is in the *final* state provided that all transition and instruction nodes have an output of 0.

We assume that the network is initialized so that the first iteration is a legal state, after which we will induct on the current instruction neuron to show that after each instruction execution the network is either in a legal or final state and that the instruction was carried out and that therefore the network properly implements the program.

Proof. Let the network constructed according to the directions above be in a legal state at the k^{th} iteration and have j such that the activated instruction neuron is N_j. Let V_i be any variable referenced by the instruction at line j. The output of a neuron N at iteration k will be denoted as $N(k)$.

Consider the following exhaustive list of cases for the instruction associated with N_j:

`nop`:

$$\begin{cases} N_j(k) = 1 \\ N_{j+1}(k) = 0 \end{cases} \text{by the definition of a legal state and}$$

$$\begin{cases} N_j(k+1) = 0 \\ N_{j+1}(k+1) = 1 \end{cases} \text{by Equation 5.}$$

`dec`:

$$\begin{cases} V(k) = v \\ N_j(k) = 1 \\ N_{j+1}(k) = 0 \end{cases} \text{for } v \in \mathbb{Z}^+ \text{ by the definition of a legal state and}$$

$$\begin{cases} V(k+1) = max(0, V_i - 1) \\ N_j(k+1) = 0 \\ N_{j+1}(k+1) = 1 \end{cases} \text{by Equation 4.}$$

`inc`: This is similar to `dec`.

`goto`:
Let m be the line referenced by the `goto` (i.e. `if V0 != 0 goto m`).

$$\begin{cases} V(k) = v \\ N_j(k) = 1 \\ N_{j'}(k) = 0 \\ N_{j''}(k) = 0 \\ N_m(k) = 0 \\ N_{j+1}(k) = 0 \end{cases} \text{by the definition of a legal state and}$$

$$\begin{cases} V(k+1) = v \\ N_j(k+1) = 0 \\ N_{j'}(k+1) = 1 \\ N_{j''}(k+1) = max(0, 1-v) \\ N_m(k+1) = 0 \\ N_{j+1}(k+1) = 0 \end{cases} \text{by Equations 6 and 7 and}$$

$$\begin{cases} V(k+2) = v \\ N_j(k+2) = 0 \\ N_{j'}(k+2) = 0 \\ N_{j''}(k+2) = 0 \\ N_m(k+2) = max(0, 1 - max(0, 1-v)) = \begin{cases} 1 & : v > 0 \\ 0 & : v = 0 \end{cases} \\ N_{j+1}(k+2) = max(0, 1-v) = \begin{cases} 0 & : v > 0 \\ 1 & : v = 0 \end{cases} \end{cases} \text{by Equation 6 and}$$

the network is back in a legal state (or final state, if N_{j+1} does not exist) at iteration $k+2$.

Thus all instructions are carried out and maintain a legal state throughout the duration of the program execution. Since the network can then implement any program in $\mathbb{L}$, it follows that recurrent neural networks are Turing-complete. □

Conclusion

In this paper the theoretical capabilities of neural networks were explored. Not only can a neural network compute the boolean functions, but recurrent neural networks are Turing-complete. The Turing-completeness is promising for the future of neural networks; however, to utilize this potential new learning algorithms and network designs will need to be developed in order to train recurrent networks as efficiently as the algorithm currently used to train feed-forward networks.

References

[1] Alan M. Turing, "On Computable Numbers, With An Application To The Entscheidungsproblem", Princeton University, 1936

[2] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2014

[3] https://proofwiki.org/wiki/Definition:Logical_NAND

[4] https://proofwiki.org/wiki/Functionally_Complete_Logical_Connectives/Negation,_Conjunction,_Disjunction_and_Implication

[5] Simon Haykin, "Neural Networks: A Comprehensive Foundation", Maxwell Macmillan International, 1994

[6] Heikki Hyötyniemi, "Turing Machines are Recurrent Neural Networks", Publications of the Finnish Artificial Intelligence Society, pp. 13-24, http://lipas.uwasa.fi/stes/step96/step96/hyotyniemi1/