# Formal verification of deep neural networks using learning cellular automata

by
Shrohan Mohapatra
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar
sm32@iitbbs.ac.in

Dr. Manoranjan Satpathy
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar
manoranjan@iitbbs.ac.in

April 16, 2019

**Abstract**

Deep neural networks (DNNs) have found diverse applications such as image processing, video processing, text classification, computer vision, safety-critical systems such as controllers for autonomous vehicles etc. But for DNNs in safety-critical systems, formal verification becomes really essential before actual deployment. There have been several algorithms, like the Reluplex algorithm, which are limitedly scalable. It has been claimed that the formal verification can be made significantly more scalable by means of intelligent parallelisation. Cellular automata (CAs) have also analysed to have some computational power and universality apart from highly scaling data parallelism. Recent literature reveals that cellular automata have been studied as a black box for neural networks to study their temporal evolution and predict the transition rules. In this article, we propose a formal verification system for deep neural networks by using equivalent learning cellular automata (LCA), a new discrete structure that incorporates all the properties of a CA associated with some learning algorithm. We provide necessary formal definitions of CAs, DNNs, and LCAs, and prove that the emulation complexity of an equivalent LCA for a given DNN is NP-complete. Finally, we describe the overall layout of the verifier based on a polynomial-time approximation of the emulation, illustrated by extensive experimental results.

# 1 Introduction

Deep neural networks consist of many nodes with some computational power, arranged in layers, that pass data from one layer to another. Input neurons get activated through sensors perceiving the environment, other neurons get activated through weighted connections from previously active neurons [16]. The computational power that the nodes have is the activation function that characterise the network. The commonly used activation functions include sinusoidal [12], linear, logarithmic, exponential functions [13], and rectified linear units [11]. DNNs find a lot of applications in neural network controllers, image recognition, text classification [14], safety-critical systems like controllers for autonomous vehicles [8, 9, 10, 17], simulation of land use [15] etc. Applications like those employed in safety-critical systems demand rigorous testing of the DNNs, which is met by formal verification. *Linear programming (LP)*, *SMT solvers* [18, 19] and Reluplex solvers [8, 10] are some of the approaches suggested in this direction. But they have been experimentally seen to be scalable to a limited extent with respect to the size of the neural network and the size of the specification. It has been claimed that the formal verification can be made significantly more scalable by means of intelligent parallelisation.

Cellular automata (CA) [1] are machines consisting of a grid of cells, each in one of a finite number of states. The grid evolves in terms of the states of its cells, all following a common transition rule. All the cells evolve simultaneously, thus exhibiting extremely high degree of data parallelism. Some elementary cellular automata [1] show class 4 behaviour [2], i.e. they display globally complex behaviour with locally interacting structures. Such CA were conjectured to be Turing-complete. Rule 110 CA has been proven to be Turing-complete [4] but the emulation complexity is too high to be practically implementable on a digital computer. Also, two-dimensional CA like the Game of Life has also been proven to be Turing-complete [3] by showing some seeds to be emulating logic gates like AND, OR, and NOT gates. But even this emulation is practically expensive and time-consuming. Recent applications of CA in deep learning ventures have correlated the two somewhat different concepts in different ways. Cellular automata like Game of Life have been modelled as convolutional neural networks [20] to learn the transition rules from the video of the temporal evolution of a given cellular automata. There are also suggestions on applications of DNN by means of a CA to simulate land use changes [15], where the DNN can be used as a base of the CA model transition rule.

This article discusses a formal verification system for deep neural networks by converting it into an approximate learning CA, a new discrete structure that *learns* its own transition function from the incoming data. The CA we present here shows the outcome of all possible values of the input data in the same grid and keeps evolving its transition rules as per the learning algorithm, so it performs the verification of all possible inputs. The following are the features of our proposed system:

1. The overall system considers all possible inputs simultaneously in a single

cell, so the verification happens in constant time. The convergence of the system is completely governed by the learning algorithm, so the focus lies mostly on this other than the structure of the neural network.

2. It would be easy to determine the failing inputs directly from the grid itself, as the contents of the cells are readily available and may be passed through a digital logic implementing the post-conditions of the specification.

3. The overall structure can be easily reconfigurable to higher precision as our algorithm is merely a truncation and runs in polynomial time, but still behaves as a subclass of the overall solution that runs in exponential time. The neural network is first converted to an equivalent function, and is then approximated using the nultivariate Taylor series. For higher precision, one can use higher orders of Taylor series approximation.

4. Changing the seed of the CA easily allows the same verification system to emulate another completely different DNN without much difficulty, thus facilitating ease of reuse.

The rest of the article is organised as follows. In section 3, we introduce some formal notions of cellular automata, deep neural networks and learning cellular automata. Also we present an algorithm to convert a DNN to its equivalent LCA, and proved that it is in NP-complete. In section 4, we describe our proposed LCA-based formal verifier. In section 5, we present various experiments on the performance of this system and compare it with existing methods such as SMT solvers and LP solvers. We finally conclude in section 6.

## 2 Related work

The problem of formally verifying any deep neural network is NP-complete [8]. Existing formal verification systems capable of handling various kinds of constraints include SMT solvers [18, 19] and linear programming solvers [9]. But these scale to a very limited extent [19]. An improvement in this direction was the development of the Reluplex algorithm [8] that considered a limited subset of all possible constraints. To improve further in terms of scalability, it has been suggested that constraints might be represented in a better way, and some smart parallelisation may be used. Cellular automata possess inherent data parallelism which can be highly scaling. Also, the interaction of the various 'cells' in a CA by means of its transition function, as defined in section 3.2, produce interesting informational variation. But as of now, their applicability is limited to domains like convolutional neural networks [20], detection of land use patterns [15]. In this article, we have provided both general (which is in NP-complete as shown in section 3.5) as well as approximate conversion of a neural network to an equivalent cellular automaton that *learns* from the ambient environment and changes its transition rules to emulate the learning behaviour over the entire span of the input space in a largely parallelised fashion. The

algorithms presented here do not get heavily influenced by the scaling behaviour of the constraints, but show good performance in terms of time and propagation error as shown in section 5.

# 3 Background

In this section, we formally define general cellular automata and then introduce the notion of learning cellular automata. We also formally define deep neural networks with one single activation function in all the layers. We finally prove that the emulation complexity of a DNN into a learning CA is NP-complete, by presenting an algorithm to fulfil the same.

## 3.1 Formal definition of a first-order cellular automata

Any $n$-dimensional $r$-neighbourhood first-order **cellular automata (CA)** can be defined as a 4-tuple $(S, Q_t, N, f)$ where $S \subseteq \mathbb{N}$, known as the set of states; $Q_t$ is an $l_1 \times l_2 \times l_3 \cdots \times l_n$ matrix for an instance of time $t$, where $\forall k \in \{1, 2, 3, 4, \dots n\} \forall i_k \in \{1, 2, 3, 4, \dots l_k\}, Q_t[i_1, i_2, i_3, \dots i_n] \in S$; $N$ is an $r \times n$ matrix known as the neighbourhood matrix; and finally $f : S^{r+1} \to S$, known as the transition function. Here the future state of a cellular automaton is defined in the following way

$\forall k \in \{1, 2, 3, 4, \dots n\} \forall i_k \in \{1, 2, 3, 4, \dots l_k\},$

$Q_{t+1}[i_1, i_2, i_3, \dots i_n] = f(Q_t[i_1, i_2, i_3, \dots i_n], h_1, h_2, h_3, \dots h_r)$

where $h_m = Q_t[\{(i_k + N[m, k]) \mod l_k) \forall k \in \{1, 2, 3 ... l_m\}\}]$
$\forall m \in \{1, 2, 3, \dots r\}$.

A special case where $n = 1, r = 2, S = \{0, 1\}, N = (1, -1)^T$, the cellular automaton becomes an elementary cellular automaton. The decimal equivalent of the binary number formed by an ordered sequence of the images of the function $f$ of an elementary cellular automaton is said to be its 'rule', or the 'Wolfram rule'. An example running of rule 110 cellular automaton is graphically illustrated in figure 1 [4]. Also, I hereby would refer to the sequence $Q_1, Q_2, Q_3, Q_4 \dots$ as the temporal evolution of a cellular automaton, the matrix $Q_1$ as the seed of the machine. Another classical example of a two dimensional cellular automaton is shown in figure 2, known as the 'Conway's Game of Life' [3], whose transition function is verbally described as follows.

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.

2. Any live cell with two or three live neighbours lives on to the next generation.

3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Figure 1: A demonstration of the rule 110 cellular automaton [4]. The first row shows the transition rules. It is noteworthy of the binary number formed of the outputs, whose decimal equivalent is rule 110. The subsequent grid shows the temporal evolution, where the initial seed consists of a single one and rest all zeroes.

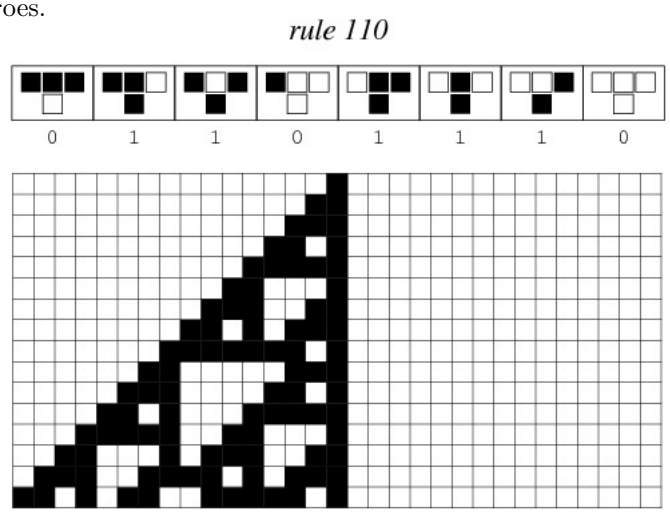
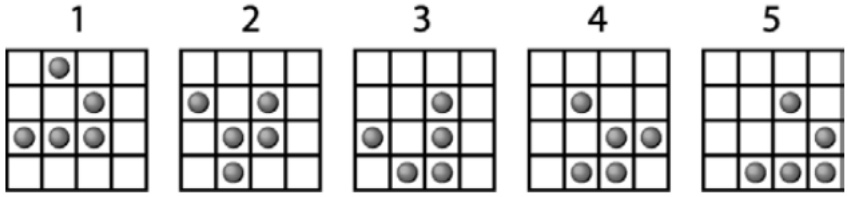



Figure 2: A demonstration of the Game Of Life cellular automaton [3]. The first grid shows the seed, and the subsequent grids show the temporal evolution of the same. This is known as a 'glider'.




## 3.2 Formal definition of a general cellular automata

An $n$-dimensional $r$-neighbourhood $g$-order cellular automata can be defined as a 4-tuple $(S, Q_t, N, f)$ where $S \subseteq \mathbb{N}$, known as the set of states; $Q_t$ is an $l_1 \times$

$l_2 \times l_3 \cdots \times l_n$ matrix for an instance of time $t$, where $\forall k \in \{1, 2, 3, 4, \ldots n\} \forall i_k \in \{1, 2, 3, 4, \ldots l_k\}, Q_t[i_1, i_2, i_3, \ldots i_n] \in S$; $N$ is an $r \times n$ matrix known as the neighbourhood matrix; and finally $f : S^{(g+1)(r+1)} \to S$, known as the transition function. Here the future state of a cellular automaton is defined in the following way

$\forall k \in \{1, 2, 3, 4, \ldots n\} \forall i_k \in \{1, 2, 3, 4, \ldots l_k\},$

$Q_{t+1}[i_1, i_2, i_3, \ldots i_n] = f(Q_t[i_1, i_2, i_3, \ldots i_n], h_1, h_2, h_3, \ldots h_r)$

where $h_m = \{$
$Q_{t-u}[\{(i_k + N[m, k]) \mod l_k) \forall k \in \{1, 2, 3 ... l_m\}\}] \forall u \in \{0, 1, 2, 3 \ldots g\}$
$\}$
$\forall m \in \{1, 2, 3, \ldots r\}.$

### 3.3 Learning cellular automata

An $n$-dimensional first-order **learning cellular automaton (LCA)** as a 6-tuple $(S, Q_t, R_t, N, f_t, A)$ where $S \subseteq \mathbb{N}$; $Q_t$ is an $l_1 \times l_2 \times l_3 \cdots \times l_n$ matrix for an instance of time $t$, where $\forall k \in \{1, 2, 3, 4, \ldots n\} \forall i_k \in \{1, 2, 3, 4, \ldots l_k\}$, $Q_t[i_1, i_2, i_3, \ldots i_n] \in S$; $N$ is an $r \times n$ matrix; $f_t : S^{r+1} \to S$, the transition function for the instance of time $t$. Here the future state of a cellular automaton is defined in the following way

$\forall k \in \{1, 2, 3, 4, \ldots n\} \forall i_k \in \{1, 2, 3, 4, \ldots l_k\},$

$Q_{t+1}[i_1, i_2, i_3, \ldots i_n] = f_t(Q_t[i_1, i_2, i_3, \ldots i_n], h_1, h_2, h_3, \ldots h_r)$

where $h_m = Q_t[\{(i_k + N[m, k]) \mod l_k) \forall k \in \{1, 2, 3 ... l_m\}\}]$
$\forall m \in \{1, 2, 3, \ldots r\}.$

Here $R_t$ is the observed data matrix at time instance $t$ (which is of the same dimension as $Q_t$), that is meant to be coming from the ambient environment, $f_{t+1} = A(f_t, R_t, Q_t)$, where $A$ is the learning algorithm. The emulation of an NN to an LCA would lead to the learning parameters that show up in the learning algorithm.

Similarly, an $n$-dimensional $r$-neighbourhood $g$-order cellular automata can be defined as a 6-tuple $(S, Q_t, R_t, N, f_t, A)$ where $S \subseteq \mathbb{N}$, known as the set of states; $Q_t$ is an $l_1 \times l_2 \times l_3 \cdots \times l_n$ matrix for an instance of time $t$, where $\forall k \in \{1, 2, 3, 4, \ldots n\} \forall i_k \in \{1, 2, 3, 4, \ldots l_k\}, Q_t[i_1, i_2, i_3, \ldots i_n] \in S$; $N$ is an $r \times n$ matrix known as the neighbourhood matrix; and finally $f : S^{(g+1)(r+1)} \to S$, known as the transition function. Here the future state of a cellular automaton is defined in the following way

$\forall k \in \{1, 2, 3, 4, \ldots n\} \forall i_k \in \{1, 2, 3, 4, \ldots l_k\},$

$Q_{t+1}[i_1, i_2, i_3, \dots i_n] = f(Q_t[i_1, i_2, i_3, \dots i_n], h_1, h_2, h_3, \dots h_r)$

where $h_m = \{$
$Q_{t-u}[\{(i_k + N[m,k]) \mod l_k) \forall k \in \{1, 2, 3 ... l_m\}\}] \forall u \in \{0, 1, 2, 3 \dots g\}$
$\}$
$\forall m \in \{1, 2, 3, \dots r\}$.

### 3.4 Deep neural networks

Formally, for a deep neural network (DNN) with an activation function $\Theta : \mathbb{R} \rightarrow \mathbb{R}$, along the lines of [8],

1. We use $n$ to denote the number of layers.
2. $s_i$ denotes the size of $i^{th}$ layer (i.e. the number of nodes in the $i^{th}$ layer), $1 \leq i \leq n$,
   (a) Layer 1 is the input layer.
   (b) Layers $2, 3, \dots n-1$ are the hidden layers.
   (c) Layer $n$ is the output layer.
3. The value of $j^{th}$ node of the layer $i$ is denoted by $v_{ij}$ and the column vector $[v_{i,1}, v_{i,2}, \dots v_{i,s_i}]^T$ is denoted by $V_i|_{s_i \times 1}$.
4. Each layer $i, 2 \leq i \leq n$ has a weight matrix $W_i$ of $s_i \times s_{i-1}$ and a bias vector $B_i|_{s_i \times i}$ such that

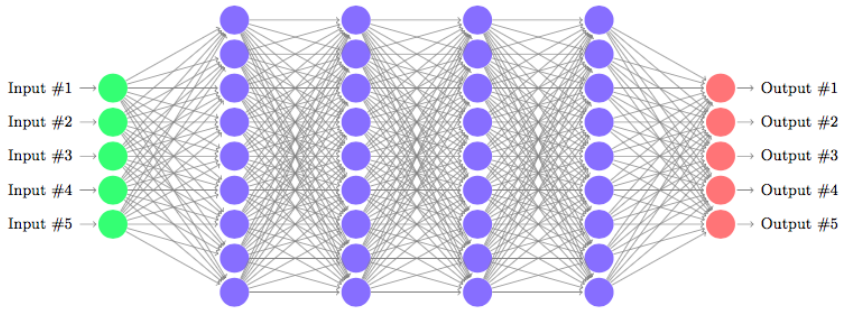
$$V_i|_{s_i \times 1} = \Theta(W_i|_{s_i \times s_{i-1}} \times V_{i-1}|_{s_{i-1} \times 1} + B_i|_{s_i \times 1}) \tag{1}$$

An architectural example of the DNN is illustrated in figure 3, where any activation function can be incorporated. One commonly used activation function is the *rectified linear unit* (ReLU), defined as $\Theta(x) = max(0, x)$, used in a lot of applications [11], consequently calling for a large number of solutions to the formal verification problem specifically for ReLU DNNs [8, 9, 10, 14]. Other activation functions include sinusoidal function ($\Theta(x) = sin(x)$) [12] and some other prominent functions such as linear, logarithmic, and exponential functions [13].

### 3.5 Conversion of neural network to LCA is NP-complete

Here we are trying to present an emulation of a neural network consisting in an equivalent cellular automaton based on algorithms used in symbolic computation. The emulation consists of the following phases.

Figure 3: A fully connected DNN with 5 input nodes (in green), 5 output nodes (in red), and 4 hidden layers containing a total of 36 hidden nodes (in blue) [10].




**Phase 1: Extraction of an equivalent function from the neural network** Here we can treat the input neural network as a parse tree with the nodes consisting of activation functions and the edges. Therefore, applying bottom up parsing through this network allows the generation of the expression by supplementing an appropriate S-attributed grammar. So this phase of conversion is in P, the time consumed is linear with respect to the number of nodes in the neural network, as the proposed algorithm is essentially a modified breadth-first search from the leafs of the equivalent directed acyclic graph.

**Phase 2: Finding a differential equation satisfied by the function** To analyse this phase, we first try to pose the problem as, 'Given a function $F$ and a differential equation $D$, does $F$ satisfy $D$?'. Guenter [7] suggests a graph-theoretic method of symbolic differentiation of a given function, which is proven and observed to be faster than some commercially viable computer algebra system. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with an expression with $\nu$ nodes, the time to perform symbolic differentiation is $O(mn\nu^3)$. Also, the evaluation of differential expression is also in P, from the algorithm discussed in phase 1. Since the posed problem is in P, from the formal definition, finding at least one such $D$ for a given $F$ is in NP.

Say function $f$ is differentiated $n$ times with respect to $x$, resulting in a sequence $S = \{f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, \frac{\partial^3 f}{\partial x^3}, \cdots \frac{\partial^n f}{\partial x^n}\}$, and an algebraic expression $E(\{x_1, x_2, \cdots x_n\})$. We can always find a finite set of functions $F = \{f_1, f_2, f_3, \cdots f_n\}$ and another algebraic expression $E'(\{x_1, x_2, \cdots x_n\})$, satisfying

$$E(S) = 0 \implies E'(F) = 0 \tag{2}$$

Separating the coefficients for each of the elements $f_i \in F$, there will be $n$ different equations $E_1'', E_2'', \cdots E_n''$, each with $c$ coefficients (not necessarily non-zero), described by the coefficient matrix $C$ as follows.

$$C = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1c} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2c} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nc} \end{bmatrix} \tag{3}$$

The $i^{th}$ row of the matrix $C$ shows the coefficients in the equation $E_i''$. Each such equation $E_i''$ is to be solved for the unknown coefficients $\{a_{i1}, a_{i2}, \cdots a_{ic}\}$ associated with the terms $T_i = \{t_{i1}, t_{i2}, t_{i3} \cdots t_{in}\}$ as an instance of the subset sum problem for the multi-set $(T_i, m_i)$, where $m_i(t_{ik}) = max(\{a_{i1}, a_{i2}, \cdots a_{ic}\}) \quad \forall 1 \leq k \leq n$. Since we know that subset sum problem is in NP-complete, using the above polynomial-time reduction, one can claim that finding a differential equation $D$ satisfied by a given function $f$ is NP-complete.

**Phase 3: Converting a differential equation into recurrence relation** Again we can define an S-attribute grammar that can be applied over LR-parsing of the differential equation, that substitutes each differential term by the recurrence relation obtained by first principle derivative. A similar approach is used in Risch's algorithm [6] where known integrals are substituted in accordance with the Liouville theorem. So this conversion is also in P.

**Phase 4: Expression of a recurrence relation using a cellular automaton** Omohundro's algorithm [5] suggests that the transition function of the converted CA can be calculated using the recurrence relation itself. The boundary condition of the differential equation appears in the initial seed of the CA. So this final phase is also in P.

For example, consider the DNN in figure 4 with $n = 2$ layers, activation function $\Theta(x) = tan^{-1}(x)$, $s_1 = 3, s_2 = 2$, and

1. $V_1 = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$
2. $W_2 = \begin{bmatrix} 0.3 & 0.7 & 0 \\ 0 & 0.8 & 0.2 \end{bmatrix}$
3. $B_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$

In phase 1, we get $V_2 = tan^{-1}(W_2V_1 + B_2)$, which is,

$$V_2 = tan^{-1}(\begin{bmatrix}0.3 & 0.7 & 0\\ 0 & 0.8 & 0.2\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix})$$
$$= \begin{bmatrix}tan^{-1}(0.3x + 0.7y + 1)\\ tan^{-1}(0.8y + 0.2z + 2)\end{bmatrix}$$
$$= \begin{bmatrix}f_1(x,y,z)\\ f_2(x,y,z)\end{bmatrix}$$
$$\implies f_1(x,y,z) = tan^{-1}(0.3x + 0.7y + 1), f_2(x,y,z) = tan^{-1}(0.8y + 0.2z + 2)$$

In phase 2, specifically for function $f_1(x, y, z)$, an ideal symbolic manipulation after the first differentiation with respect to the variables leads us to

$$tan(f_1) = 0.3x + 0.7y + 1$$
$$\implies \frac{\partial f_1}{\partial x} = \frac{0.3}{1 + (0.3x + 0.7y + 1)^2} = 0.3cos^2(f_1)$$
$$\frac{\partial f_1}{\partial y} = \frac{0.7}{1 + (0.3x + 0.7y + 1)^2} = 0.7cos^2(f_1)$$
$$\frac{\partial f_1}{\partial z} = 0$$
$$\implies \frac{\partial f_1}{\partial x} + \frac{\partial f_1}{\partial y} + \frac{\partial f_1}{\partial z} = cos^2(f_1)$$

In phase 3, the differential equation is converted to a recurrence relation by replacing each differential term by the derivative by first principle, as is explained below.

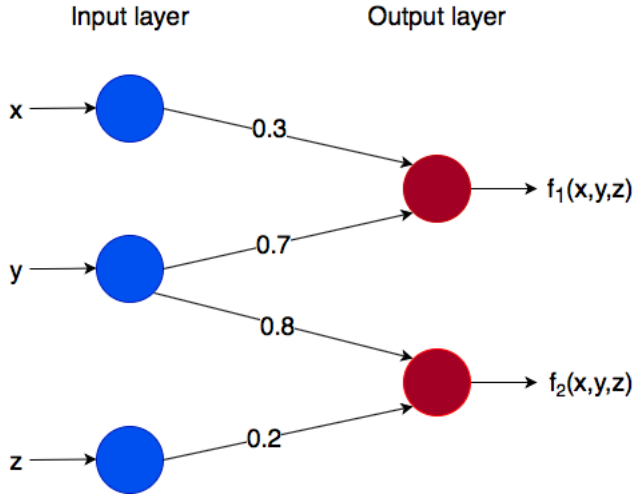
$$\frac{\partial f_1}{\partial x} + \frac{\partial f_1}{\partial y} + \frac{\partial f_1}{\partial z} = cos^2(f_1)$$
$$\implies \lim_{\Delta x \to 0} \frac{f_1(x + \Delta x, y, z) - f_1(x, y, z)}{\Delta x} + \lim_{\Delta y \to 0} \frac{f_1(x, y + \Delta y, z) - f_1(x, y, z)}{\Delta y}$$
$$+ \lim_{\Delta z \to 0} \frac{f_1(x, y, z + \Delta z) - f_1(x, y, z)}{\Delta z} = cos^2(f_1(x, y, z))$$

By choosing appropriate values of $\Delta x, \Delta y$, and $\Delta z$, we can safely claim that

$$\frac{f_1(x + \Delta x, y, z) - f_1(x, y, z)}{\Delta x} + \frac{f_1(x, y + \Delta y, z) - f_1(x, y, z)}{\Delta y}$$
$$+ \frac{f_1(x, y, z + \Delta z) - f_1(x, y, z)}{\Delta z} = cos^2(f_1(x, y, z))$$
$$\implies f_1(x, y, z + \Delta z) = f_1(x, y, z) +$$
$$\Delta z\left[cos^2(f_1(x, y, z)) - \left(\frac{f_1(x + \Delta x, y, z) - f_1(x, y, z)}{\Delta x} + \frac{f_1(x, y + \Delta y, z) - f_1(x, y, z)}{\Delta y}\right)\right]$$

In phase 4, the above recurrence relation is converted into the corresponding CA using the Omohundro's algorithm [5]. In general, due to phase 2, automated emulation of a neural network into an equivalent cellular automaton is in NP-complete. Other than the performance, uniqueness of the solution is also under question. There can be more than one differential equation satisfied by a function. If one DE $D$ is found satisfied by a function $F$, another DE $D'$ obtained by differentiating both sides by the same variable, but is also satisfied by $F$. So there can be infinitely many differential equations fulfilling the purpose. Working in the reverse direction, integration with respect to any one variable introduces an integration constant, which can take any value. So this shows that an emulating CA can represent infinitely many neural networks. This shows that the equivalent CA forms the basis of a class of neural networks.

Figure 4: The exemplary DNN with two layers with edges labelled with weights. The activation function considered here is $\Theta(x) = tan^{-1}(x)$.




# 4 Overall framework for the proposed formal verification

Here we present the design of the overall framework for the formal verification of the neural networks using learning cellular automata. We begin with a polynomial time approximation of the algorithm presented in section 3.5. We use this to present stage-by-stage analytical architecture of our system.

## 4.1 A polynomial time approximation for LCA from neural networks

A general NP-complete algorithm to obtain an equivalent cellular automata from a given neural network has been shown in section 3.5. Precisely, the NP-completeness arises from the phase 2 of the algorithm, where one needs to obtain a differential equation satisfied by a given function. Here, a polynomial time approximation of this phase is shown here for a general function, $f : \mathbb{R}^n \to \mathbb{R}$, of $n$ variables.

Using a simple multivariate Taylor expansion we can obtain $n + 1$ constants $\alpha_1, \alpha_2, \ldots, \alpha_n, \beta$ from the function $f$ such that

$$f(x_1, x_2, \ldots x_n) = \beta + \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n + O(x_1 x_2 \ldots x_n) \tag{4}$$

where,

$$\beta = f(0, 0, 0 \ldots 0)$$
$$\alpha_i = \frac{\partial f}{\partial x_i}|_{(0,0,0\ldots 0)} \forall i \in \{1, 2, 3, \ldots n\}$$

One of all the PDEs satisfied by $f$ is given as,

$$\frac{\partial f}{\partial x_1} + \frac{\partial f}{\partial x_2} + \cdots + \frac{\partial f}{\partial x_n} = \alpha_1 + \alpha_2 + \cdots + \alpha_n \tag{5}$$

where $f(0, 0, 0 \ldots 0) = \beta$ and $f(x_1, x_2 \ldots x_{n-1}, 0)$ are defined for $x_i \neq 0, 1 \leq i \leq n$. Here $x_n$ may be treated as a temporal variable, and $\alpha_i$s may vary with time. Since function simplification and derivative computation can happen in polynomial time, obtaining the differential equation with this approximation can also be done in polynomial time.

## 4.2 Formal verification system

Now we present the design of the formal verification system for the deep neural network using the polynomial time heuristic sketched in section 4.1 that converts the DNN into an equivalent LCA. We begin with phase 1 explained in section 3.5, where we obtain the function from neural network. Using equation 1 described in section 3.4 for a neural network with $n-1$ hidden layers and activation function $\Theta$, we obtain subsequent final formulas for the output from the corresponding layers as,

$$
\begin{aligned}
V_2 &= \Theta(W_2V_1 + B_2) \\
V_3 &= \Theta(W_3V_2 + B_3) \\
&= \Theta(B_3 + W_3\Theta(B_2 + W_2V_1)) \\
V_4 &= \Theta(B_4 + W_4\Theta(B_3 + W_3\Theta(B_2 + W_2V_1))) \\
&\cdots\cdots\cdots \\
V_n &= \Theta(B_n + W_n\Theta(B_{n-1} + W_{n-1}\Theta(\ldots\Theta(B_2 + W_2V_1)\ldots)))
\end{aligned}
$$

Ultimately, the $n^{th}$ layer vector $V_n$ can be represented as a set of functions $F = \{f_1, f_2, f_3 \ldots f_n\}$ such that

$$v_{n,i} = f_i(v_{1,1}, v_{1,2}, \ldots, v_{1,s_1})$$

Now we are going to obtain the differential equations corresponding to all the functions $f \in F$. Rearranging equation 5, we obtain

$$\frac{\partial f}{\partial x_n} = \sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n-1} \frac{\partial f}{\partial x_i} \tag{6}$$

Following phase 3 of the general algorithm in section 3.5,

$$\lim_{\Delta x_n \to 0} \frac{f(x_1, x_2, x_3 \ldots, x_{n-1}, x_n + \Delta x_n) - f(x_1, x_2, x_3 \ldots, x_{n-1}, x_n)}{\Delta x_n} =$$

$$\sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n-1} \lim_{\Delta x_i \to 0} \frac{f(x_1, x_2, \ldots x_{i-1}, x_i + \Delta x_i, x_{i+1}, \ldots x_n) - f(x_1, x_2, \ldots x_{i-1}, x_i, x_{i+1}, \ldots x_n)}{\Delta x_i}$$

An appropriate choice of the incrementals $\Delta x_i, \forall i \in \{1, 2, \ldots n\}$, leads us to,

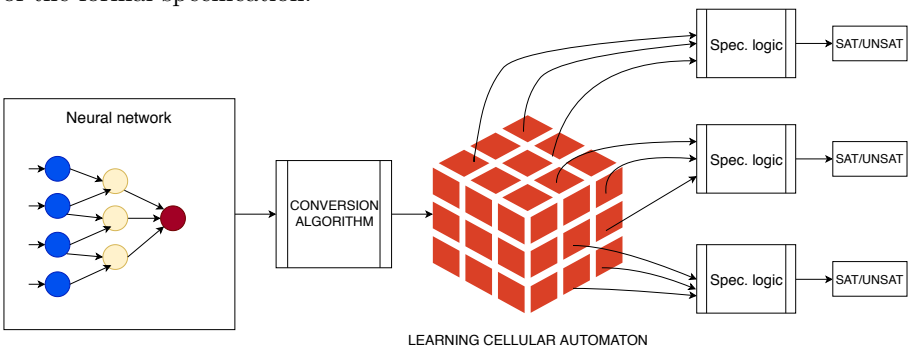
$$f(x_1, x_2, x_3 \ldots x_n + \Delta x_n) = f(x_1, x_2, x_3, \ldots x_n) +$$

$$\Delta x_n \left( \sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n-1} \frac{f(x_1, x_2, \ldots x_{i-1}, x_i + \Delta x_i, x_{i+1}, \ldots x_n) - f(x_1, x_2, \ldots x_{i-1}, x_i, x_{i+1}, \ldots x_n)}{\Delta x_i} \right) \tag{7}$$

From this, we can define the equivalent LCA $L$ as the 6-tuple $(S, Q_t, R_t, N, f_t, A)$ where $S = \mathbb{R}$, $Q_{x_n}(x_1, x_2, \ldots x_{n-1}) = f(x_1, x_2, x_3 \ldots x_n)$, $R_t$ comes from the actual data coming from the environment, $f_t$ is corresponding CA rule for the recurrence relation defined in equation 7, $A$ is the learning algorithm and

$$N = \begin{bmatrix} \Delta x_1 & 0 & 0 & \cdots & 0 \\ 0 & \Delta x_2 & 0 & \cdots & 0 \\ 0 & 0 & \Delta x_3 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \Delta x_{n-1} \end{bmatrix} \tag{8}$$

Figure 5: The overall design of the formal verification system. The blocks containing 'Spec. Logic' have a common first-order logic based representation of the formal specification.




This is one such LCA for the corresponding function $f \in F$. There are $s_n$ such simultaneous LCAs that would approximately emulate the entire DNN. This framework can be used for formal verification of the DNN in the following way:

1. The indices $x_i, \forall i \in 1, 2, 3, \ldots n$ of the grid of the LCA, i.e. $Q_t$ are chosen so that they satisfy the pre-conditions of the specification.

2. The cells of the grid $Q_t$ of the LCA, along with their neighbours defined by the neighbourhood matrix $N$, are passed through the digital logic according to the post-conditions of the specification.

Figure 5 shows the overall design of the formal verification system. For example, consider the DNN in figure 6 with $n = 2$ layers, activation function $\Theta(x) = tan^{-1}(x)$, $s_1 = 3, s_2 = 1$, and

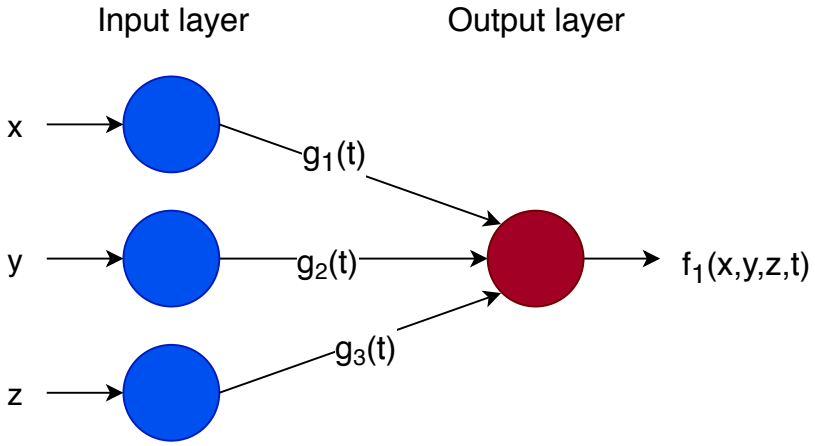
1. $V_1 = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$

2. $W_2 = \begin{bmatrix} g_1(t) & g_2(t) & g_3(t) \end{bmatrix}$

3. $B_2 = \begin{bmatrix} 0 \end{bmatrix}$

In phase 1, we get $V_2 = tan^{-1}(W_2 V_1 + B_2)$, which is,

Figure 6: Another exemplary DNN with two layers with edges labelled with weights. The activation function considered here is $\Theta(x) = tan^{-1}(x)$.




$$
\begin{aligned}
V_2 &= tan^{-1}\left( \begin{bmatrix} g_1(t) & g_2(t) & g_3(t) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \right) \\
&= \left[ tan^{-1}\Big( g_1(t)x + g_2(t)y + g_3(t)z \Big) \right] \\
&= \begin{bmatrix} f_1(x,y,z,t) \end{bmatrix} \\
\implies f_1(x,y,z,t) &= tan^{-1}\Big( g_1(t)x + g_2(t)y + g_3(t)z \Big)
\end{aligned}
$$

where $g_1(t) = 0.5^{0.447\left(1.62^t - (-0.62)^t\right)}$, $g_2(t) = 0.3^{0.447\left(1.62^t - (-0.62)^t\right)}$ and $g_3(t) = 0.2^{0.447\left(1.62^t - (-0.62)^t\right)}$.

Computing $\alpha_i, i \in \{1,2,3,4\}$ as described in equation 4,

$$
\begin{aligned}
\alpha_1 &= \frac{\partial f_1}{\partial x}|_{0,0,0,0} = 1 \\
\alpha_2 &= \frac{\partial f_1}{\partial y}|_{0,0,0,0} = 1 \\
\alpha_3 &= \frac{\partial f_1}{\partial z}|_{0,0,0,0} = 1 \\
\alpha_4 &= \frac{\partial f_1}{\partial t}|_{0,0,0,0} = 0
\end{aligned}
$$

$$\beta = f_1(0,0,0,0) = \frac{\pi}{4}$$

Following the lines, we obtain the differential equation,

$$\frac{\partial f_1}{\partial t} = 3 - \left(\frac{\partial f_1}{\partial x} + \frac{\partial f_1}{\partial y} + \frac{\partial f_1}{\partial z}\right)$$

Assuming $\Delta x = \Delta y = \Delta z = \Delta t = 1$, we obtain the recurrence relation as,

$$\begin{aligned} & f_1(x,y,z,t+1) - f_1(x,y,z,t) = \\ 3 - \Big( & f_1(x+1,y,z,t) - f_1(x,y,z,t) + f_1(x,y+1,z,t) \\ & - f_1(x,y,z,t) + f_1(x,y,z+1,t) - f_1(x,y,z,t) \Big) \\ \implies & f_1(x,y,z,t+1) = 4f_1(x,y,z,t) + 3 - f_1(x+1,y,z,t) \\ & - f_1(x,y+1,z,t) - f_1(x,y,z+1,t) \end{aligned} \tag{9}$$

with the boundary conditions $f_1(0,0,0,0) = \frac{\pi}{4}$, and $f_1(x,y,z,0) = tan^{-1}(x+y+z), x \neq 0, y \neq 0, z \neq 0$. So we can define the equivalent LCA $L$ as the 6-tuple $(S, Q_t, R_t, N, f_t, A)$ where $S = \mathbb{R}$, $Q_t(x,y,z) = f_1(x,y,z,t)$, $R_t$ comes from the actual data coming from the environment, $f_t$ is the corresponding CA rule for the recurrence relation defined in equation 9, $A$ is the learning algorithm and

$$N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

For example, $Q_1(1,1,1) = 0.784735$ and the original neural network results in $f_1(1,1,1,1) = 0.784739$, which renders approximately 0.0007% error. The variation of this error against various parameters has been adjudged in section 5. As an example of verification for a time instance $t_0$, we consider the following specification of the *local adversarial robustness* around a given point $(x_0, y_0, z_0)$ [8].

$$\begin{aligned} & \forall (x,y,z) \in \mathbb{R}^3, \|(x,y,z) - (x_0,y_0,z_0)\| < \delta \\ & \implies \|f_1(x,y,z,t_0) - f_1(x_0,y_0,z_0,t_0)\| < \epsilon \end{aligned} \tag{11}$$

Let $(x_0, y_0, z_0) = (1,2,3), \delta = 1, \epsilon = 3$. For time instances $t_0 = 1,2,3,4$, Table 1 shows the points where the specification 11 is met or not. The time taken to check these points for the satisfiability of the formal specification is not much, but are dependent on some factors as shown in section 5.

Table 1: The specification verification table for the considered neighbourhood points at the given time instances.

| | $(x_0, y_0, z_0)$ | (0,2,3) | (2,2,3) | (1,1,3) | (1,3,3) | (1,2,2) | (1,2,4) |
|---|---|---|---|---|---|---|---|
| $t_0$ | Robustness | | | | | | |
| 1 | UNSAT | F | T | T | T | T | T |
| 2 | SAT | T | T | T | T | T | T |
| 3 | UNSAT | F | T | T | T | T | T |
| 4 | UNSAT | T | T | T | T | T | T |

# 5 Experimental results

In this section, we show some experimental results that exhibit the efficiency of the proposed formal verification system in terms of time complexity. There are two aspects of this: the time taken to convert the given neural network to the CA, and time taken to verify the properties using that CA. The dependence of the time taken on various characteristics of a DNN such as number of layers, number of nodes per layer, choice of activation etc. are illustrated through the results shown in section 5.1. The other aspect, i.e. the verification time complexity, has been dealt with and its variation with the number of boolean conditions/predicates and the number of layers in DNN has been shown in section 5.2. Also, as seen in equation 4, the truncation error of approximation is $O(\Delta x_1 \Delta x_2 \dots \Delta x_n)$, where $n$ is the number of nodes in the input layer and $\Delta x_i$s are the resolution differences. This error gets propagated to the verification stage as well, when the values are borrowed from the LCA grid. This has been experimented upon and shown in section 5.3. Finally, the existence of classes of formal verification methods such as satisfiability modulo theories (SMT) solvers and linear programming (LP) solvers has also been considered, and some experimental comparison with our proposed methodology has been made in section 5.4. All the implementations have been done in Mathematica 11.2, where well-optimised in-built functions such as *NetGraph* (used for constructing deep neural networks), *CellularAutomaton* (used for emulation of learning CA), *FindInstance* (which makes use of SMT to find a satisfiable instance for a predicate) and *LinearProgramming* (which solves LP problems using some compile-time optimisations) are effectively used.

## 5.1 Experiments on conversion of DNN to LCA

The time taken to convert a neural network into its learning cellular automata using the four-phase method explained in section 4 varies with the choice of activation function, number of layers and number of nodes per layer. Figure 7 illustrates that the variation of time is dictated by how complex the activation function is. Figure 8 and Figure 9 show that the time increases steadily with the increase in total number of nodes in the DNN.

Figure 7: A plot of the time taken by the conversion algorithm against various activation functions. Each DNN considered here has 5 layers having 4 nodes each (except for the output layer that has only one node). The function 'LogSigmoid' represents logistic sigmoid function.

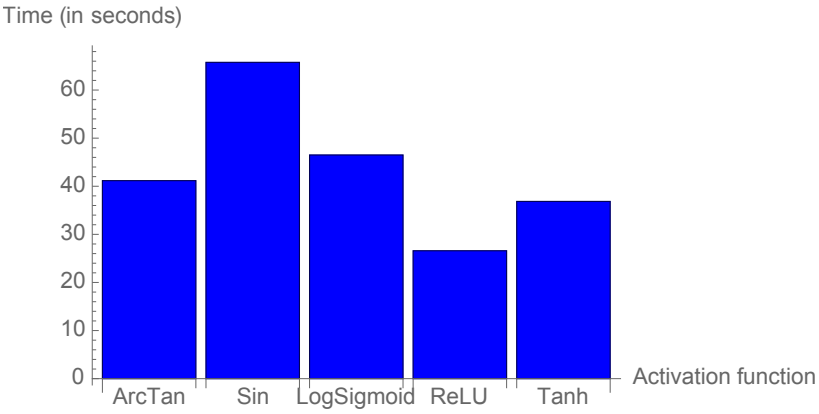
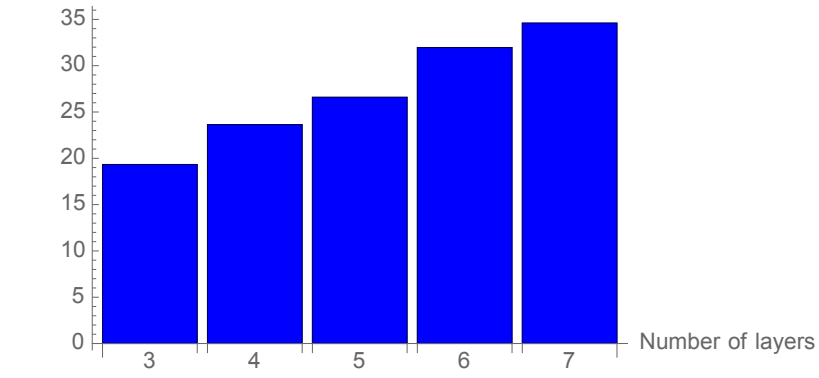


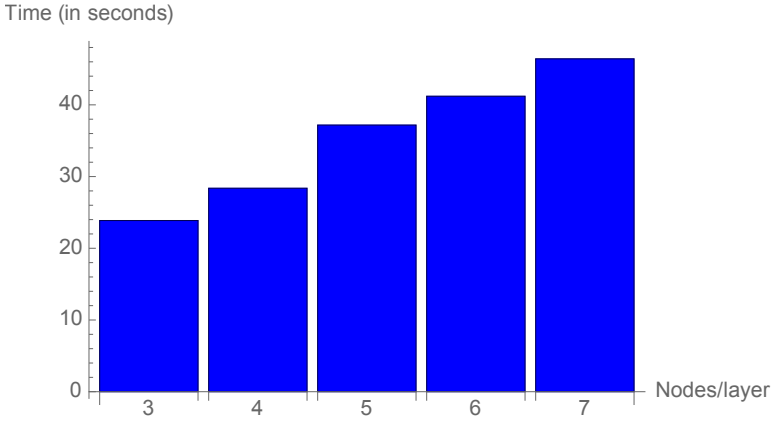

Figure 8: A plot of the time taken by the conversion algorithm against the number of layers. Each DNN considered here has 4 nodes per layer (except for the output layer that has only one node) with ReLU as the activation function.




## 5.2 Experiments on verification using LCA

The actual verification phase for the DNN, as explained in section 4.2, involves only the equivalent LCA, specifically the grid values. A first-order logic formula representing the formal specification for the DNN takes into account these grid values for the verification. So the overall verification time solely depends on the specification (in fact, it increases with the number of conditions) and not on the size of the neural net itself. This is clearly visible from figures 10 and 11. It is also noteworthy that the time taken for the conversion part (several

Figure 9: A plot of the time taken by the conversion algorithm against the number of nodes per layer. Each DNN considered here has 5 layers, one node in the output layer, and all the nodes are activated by the ReLU function.




tens of seconds) is about a thousand times that taken for the verification (a few milliseconds). This is so because the output data for all possible inputs to the DNN is present on the complete grid, and every cell there evolves simultaneously at a single time instance. So once we have an equivalent learning CA generated from the DNN, the actual formal verification is extremely fast.

Figure 10: A plot of the time taken to verify the DNN using the LCA against the number of Boolean conditions/predicates in the specification. The DNN considered here has 5 layers with 4 nodes each (except for the output layer which has one node) with ReLU as the activation function.

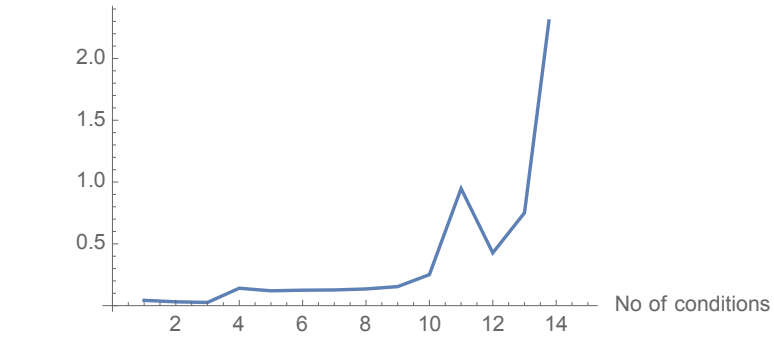
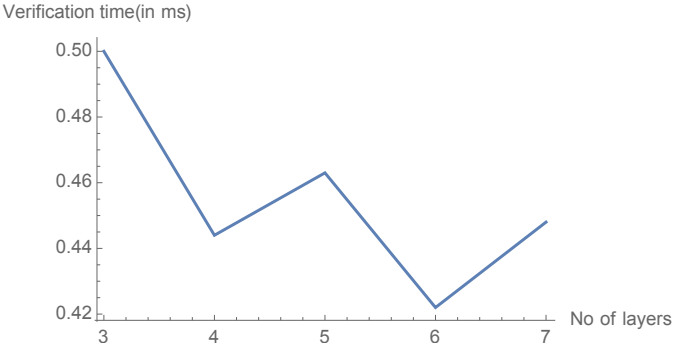


Figure 11: A plot of the time taken to verify the DNN using the LCA against the number of layers. The DNN considered here has 4 nodes in each layer (except for the output layer which has one node) with ReLU as the activation function. A Boolean predicate-based specification containing 12 Boolean predicates was chosen randomly among all possible ($2^{2^{12}}$) specifications.




## 5.3 Experiments on finding truncation errors

As is visible in equation 4, the error of truncation involved in our polynomial-time approximation algorithm is $O(\Delta x_1 \Delta x_2 \Delta x_3 \dots \Delta x_n)$, where $n$ is the number of input variables in the original DNN, and $\Delta x_i$s are the resolution differences. Assuming $\Delta x_i = \delta, \forall i, 1 \leq i \leq n$, and $k = \frac{\partial^n f}{\partial x_1 \partial x_2 \dots \partial x_n}|_{\{0,0,0\dots 0\}}$, $f$ being the equivalent function of the DNN, the error complexity is $O(k\delta^n)$. The variation of this propagated error is shown in figures 12 and 13. The value of $k$ is dictated by the choice of activation function. Also, as expected, the larger the input layer, lesser is the error, and lesser the resolution, the lesser is the error. But lesser resolution implies larger number of cells in the LCA grid, where one needs to adhere to some appreciable trade-off. Also, with the apt choice of resolution and the number of the input variables, as seen more prominently in figure 12, the propagated error is really small. So the convergence of the learning cellular automata towards the corresponding value produced by the DNN is restricted to one or two time instances.

## 5.4 Comparison with existing formal methods

There exist other formal verification methods such as those based on SMT solvers, that optimise on the predicate-variant of the original DPLL algorithm, and LP based algorithms, applicable to neural nets having linear activation functions, such as ReLU. Several LP based approaches exist in literature such as the simplex algorithm, Reluplex algorithm (specifically for ReLU based DNNs) [8] etc. In figure 14, the proposed LCA-based verification system has been compared with optimised SMT solvers and LP solvers available in the Mathematica libraries in terms of the total verification time taken. Clearly our system takes much less time in comparison to the others, and this does not scale much with

Figure 12: The plot of propagated error of truncation against the number of input variables to the DNN. Here each of the four neural nets (for different activation functions) contains 5 hidden layers with 4 nodes per layer. The value of $\delta$ is fixed at 0.01. The function 'Ramp' is the rectified linear unit (ReLU).

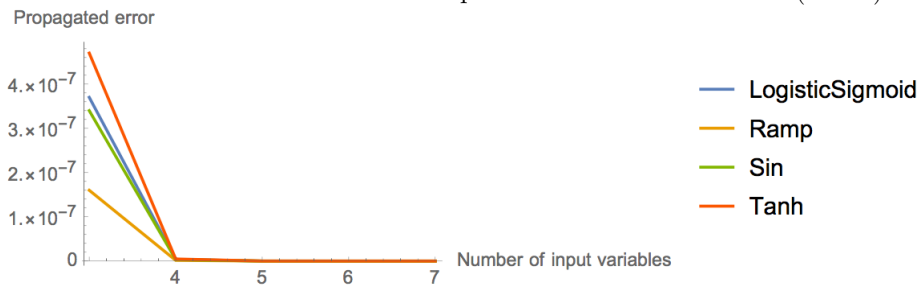
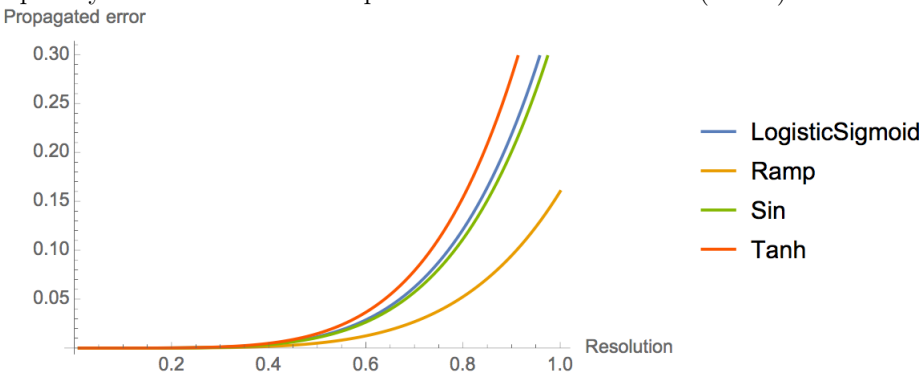



Figure 13: The plot of propagated error of truncation against the resolution, i.e. the value of $\delta$. Here each of the four neural nets (for different activation functions) contains 5 hidden layers with 4 nodes per layer, with 5 nodes in the input layer. The function 'Ramp' is the rectified linear unit (ReLU).

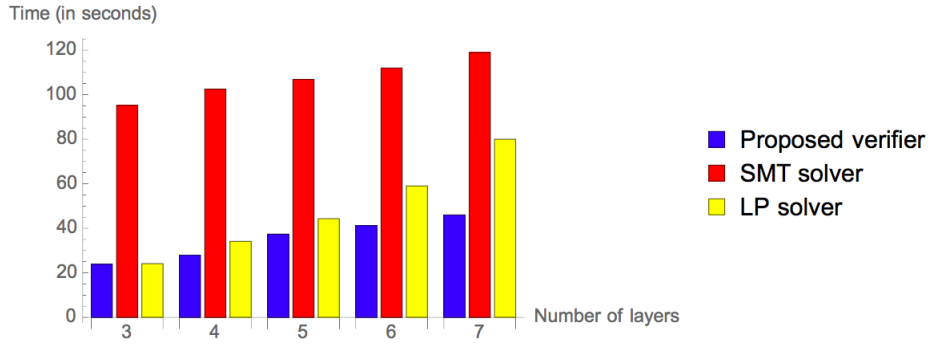



the increasing size of the neural net. This can be explained by the high level of data parallelism present inherently in the definition of a cellular automata.

# 6 Conclusion

In this paper, we present formal definitions of first-order and general cellular automata, and deep neural networks, and introduce the notion of learning cellular automata. Also, we prove that the conversion of deep neural network to an equivalent LCA is in NP-complete, and present an LCA-based formal verification system based on a Taylor series based polynomial time approximate of the

Figure 14: A plot of the time taken by the formal verification system compared with the SMT solver and LP solver available in the libraries of Mathematica 11, and its variation against the number of layers in the DNN. The neural net has 4 nodes per layer (except for the output layer that has only one node) with ReLU as the activation function.




conversion. Finally, we show the performance of the system, in terms of time consumption and error rate, and compare the performance with some existing methods which are implementable in the experimental environment. The magnitude of the propagation error allows programmability and faster convergence of the LCA. Also, the system is usable for most of the possible applications as it is not functionally influenced by the choice of activation function. But the most visible bottleneck in the total time consumption of the formal verification system is the conversion of DNN to LCA which consumes majority of the time. In future, work can be done to optimise upon the polynomial time approximation of the NP-complete algorithm, while ensuring less error complexity to facilitate the convergence.

# References


[1] J. Schiff, "Introduction to Cellular Automata", $1^{st}$ edition, *Wiley and sons*

[2] Wolfram, S., "Cellular automata as models of complexity", in *Nature*, Vol. 311, No. 5985, pp. 419-424, 1984

[3] Renard, J.P., "Implementation of logical functions in the Game of Life", in *A. Adamatzky (Ed.), Collision-Based Computing*, pp. 491-512, 2002

[4] Cook, M., "Universality in elementary cellular automata", in *Complex Systems*, vol. 1, pp. 1-40, 2004

[5] Omohundro, S., "Modelling Cellular Automata with Partial differential equations", in *Physica D: Nonlinear Phenomena*, 10.1016/0167-2789(84)90255-0

[6] Geddes, K.O., Czapor, S.R., Labahn, G. "Algorithms for Computer Algebra", in *Kiuwer Academic Publishers*

[7] Guenter, B.K., "Efficient symbolic differentiation for graphics applications", in *ACM Trans. Graph.*, 2007

[8] Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M., "Reluplex: An Efficient SMT solver for deep neural networks", in *International Conference on Computer Aided Verification*, pp., 97-117, 2017

[9] Sun, X., Khedr, H., Shoukry, Y., "Formal Verification of Neural Network Controlled Autonomous Systems", arXiv: 1810.13072, 2018

[10] Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M., "Towards Proving the Adversarial Robustness of Deep Neural Networks", arXiv:1709.02802, 2017

[11] Agarap, A., "Deep Learning using Rectified Linear Units (ReLU)", arXiv:1803.08375, 2018

[12] Gashler, M., Ashmore, S., "Training Deep Fourier Neural Networks To Fit Time-Series Data", arXiv:1405.2262, 2014

[13] Godfrey, L., Gashler, M., "A continuum among logarithmic, linear, and exponential functions, and its potential to improve generalization in neural networks", in *7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management: KDIR*, 2016

[14] Lomuscio, A., Maganti, L., "An approach to reachability analysis for feed-forward ReLU neural networks", arXiv:1706.07351, 2017

[15] Charif, O., Omrani, H., Basse, R.M., "Cellular automata based on artificial neural network for simulating land use changes", in *Proceedings of the 45th Annual Simulation Symposium*, Article No. 1, 2012

[16] Schmidhuber, J., "Deep Learning in Neural Networks: An Overview", in *Neural Networks*, 61:85-117, arXiv:1404.7828, 2015

[17] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B. et al, "End to End Learning for Self-Driving Cars", arXiv:1604.07316, 2016.

[18] Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A., "Measuring Neural Net Robustness with Constraints", in *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016

[19] Pulina, L., Tacchella, A., "Challenging SMT Solvers to Verify Neural Networks", in *AI Communications*, 25(2):117-135, 2012

[20] Gilpin, W., "Cellular automata as convolutional neural networks", arXiv:1809.02942, 2018