Modelling and Analysis of Spiking Neural Networks

By Soham Banerjee

Modelling and Analysis of Spiking Neural Networks

Soham Banerjee

Modelling and Analysis of Spiking Neural Networks

Dissertation submitted in partial fulfillment of the requirements
for the degree of

Master of Technology
in
Computer Science

by

Soham Banerjee
[Roll No: CS2024]

under the guidance of

Ansuman Banerjee
Professor
Advanced Computing and Microelectronics Unit

Swarup Mohalik
Prinicipal Engineer - Research
Ericsson Reserach India

Indian Statistical Institute
Kolkata-700108, India

July 2022

CERTIFICATE

This is to certify that the dissertation titled **"Modelling and Analysis of Spiking Neural Networks"** submitted by **Soham Banerjee** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under our supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in our opinion, has reached the standard needed for submission.

Ansuman Banerjee
Advanced Computing and Microelectronics Unit(ACMU)
Indian Statistical Institute, Kolkata

Swarup K. Mohalik
Ericsson Research
Bangalore

Acknowledgement

I would like to thank my supervisors, *Dr. Ansuman Banerjee*, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata and *Dr. Swarup K. Mohalik*, Ericsson Research for their continuous guidance and unwavering support. For the entire duration of the thesis, I have had many opportunities to learn and improve myself and my work. Their guidance has given me a much better appreciation of the research sphere and the value of good quality work.

My deepest thanks to the faculties of Indian Statistical Institute and Ericsson Research, for their support and assistance throughout the duration of the thesis.

I would also like to thank *Dr. Sumana Ghosh*, ECSU, Indian Statistical Institute, Kolkata for the great amounts of insight and assistance provided that has helped me improve my work further.

Last but not the least, I would like to thank my family, friends and peers for their help and support. I thank all those, whom I have missed out from the above list.

Soham Banerjee
Roll No. CS2024
Indian Statistical Institute
Kolkata - 700108, India.

Abstract

The increasing demand for neural network based models for problem solving is due to it's simple operation and robustness. These models have been able to solve many problems that traditional solutions have a hard time solving. This increased demand for more efficient and robust models has caused the field of artificial intelligence to be one of the most popular fields of study. With the continuous and rigorous developments in this field, the need for systems and models capable of solving specific problems with high performance has also increased manifold. This has led to the introduction of larger, more complicated networks that have shown great results. One of the major drawbacks of these larger deep neural networks is that the amount of power consumed by the hardware that runs these models is extremely high. This drawback prevents these systems to be used for edge systems which cannot run hardware with high power requirements. To overcome this drawback of traditional deep neural networks, a new type of neural network called Spiking Neural Network (SNN) has been introduced. These are special neural networks that have great similarity in their architecture and operation with the human brain, unlike the traditional networks which only share the architecture. These spiking neural networks have shown similar performance when compared to traditional networks with a much lesser energy requirement.

This thesis predominantly focuses on formal methods for modelling and analysis of Spiking Neural Networks with an objective to model the behaviors and apply various frameworks like verification and simplification. We discuss the various properties of spiking neural networks and their functioning. We then use different mathematical objects to model the network and it's properties. One such model is the Linear Real Arithmetic (LRA). We express the network as a set of linear real arithmetic formulas, which can then be used to test, extend and modify the network. We propose a LRA based encoding for spiking neural networks and define a few frameworks that take advantage of the encoding. These frameworks are simplification, verification and neuron equivalence of spiking neural networks. Each of these frameworks is based on the proposed encoding.

We also discuss the modelling of SNNs using timed automata. We talk about how timed automata can be used to encode a SNN and how the the timed automata simulation and verification can be used for the sound modelling of SNNs. The frameworks discussed are based on the timed automata encoding and its consequent simulation and verification.

We have implemented our proposed encoding and have tested the proposed frameworks on various spiking neural networks. With the increasing performance and applications of spiking neural networks, the requirement for a modelling framework will increase. We believe that our thesis makes an important contribution in that space.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, problems like image recognition, text classification and pattern recognition have been extremely challenging to solve using traditional algorithms and strategies [1]. In order to tackle these problems and come up with solutions with higher performance in terms of simplicity and accuracy, we often use *Deep Neural Networks*. These are models that show exceptional accuracy and robustness, especially on inputs not encountered before.

These deep neural networks are based on the architecture of the brain in a way that, the processing units are isomorphic to the neurons of the brain and the connections between them are isomorphic to the synapses present in the brain. The synapses have real weights associated with them while the processing nodes have activation functions associated with them. The network is often structured as collections of these processing units called as *layers*. These layers are connected to each other and act as parallel computation subsystems which make up the entire model. The model, upon receiving an input propagates the input forward through the entire network using the *synapses* and the *nodes* while performing the associated operations. In a logical sense each layer is said to take an input vector from its previous layer and then pass an output vector which will act as the input for the next layer. The output of the entire model is the output vector generated by the last layer. There are no fixed structural constraints for deep neural networks and these are often determined by the problem and the data. In an abstract sense, complex problems require complex structures whereas simple problems can be achieved with smaller and simpler networks.

The weights of the synapses are parameters of the model that need to be learned. This learning is done by a process called as *training* which is done using the existing data on the model to generate outputs and then calculating the loss between the expected outputs and the actual outputs [14]. We try to optimize the network by minimizing the loss. This training process is often a one time process and is greatly affected by the size, quality and structure of the data available for training. A well trained network performs well on both seen and unseen data. This property makes these models appealing to use for most problems.

With increasing power of processing systems and reducing costs of memory, the popularity of deep neural networks is increasing by a great amount. Unlike its other alternatives, deep neural networks are more robust and can utilize the extra computational power well. These simple networks discussed are usually referred to as the *first generation of neural networks* [3]. Even though a large number of problems can be solved using these simple deep neural networks, they still lack the ability to solve complicated problems like machine translation and pattern recognition. For such problems there have been many new variants of deep neural networks that have been developed to further increase the scope of problems that can be solved by the same.

These neural networks are often referred to as the *second generation of neural networks* and have many architectural variations unlike their predecessors. These variations often yield better performance for specific problems and are often different for different types of problems. In the current spectrum of artificial intelligence and machine learning, these models are considered to be the best models available to solve complicated problems. Some of these variations of deep neural networks referred from [14] are mentioned in the following.

(a) *Convolutional Neural Networks*
These deep neural networks have a special convolutional layer along with the regular layers. These convolutional layers have specific filters that are used to filter out specific sections of the input data. This property of being able to extract specific parts of the input data for processing makes convolutional neural networks great for image processing and pattern recognition.

(b) *Recurrent Neural Networks*
These deep neural networks unlike other networks have recurrent layers, which are special layers which have synapses connecting to the next layer as well as itself. This structure allows the network to essentially remember the previous output (by propagating the output of the layer as input to the same layer). This property gives recurrent neural networks the ability to recognize sequential patterns which is often used for natural language processing problems.

(c) *Transformers*
These are special deep neural networks that are often a sequence of encoder layers and decoder layers. These layers encode the input into a sequence of numbers and decode them back to reconstruct the output. This structure allows the transformer to work well for problems like machine translation, where the encoding-decoding process can be used to successfully express the translation process mathematically.

These models are further improved by using different mechanisms and strategies like transfer learning, adaptive boosting, hyper-parameter tuning, attention, masking, negative sampling etc. These together with the discussed architectures lead to more complicated and powerful models.

There are many more variations of deep neural networks which are extensively used to solve increasingly challenging problems. But, all these networks also share a few common drawbacks. These are:

- Highly dependent on availability of data.
- Require high computational resources.
- Require large amount of power to run the hardware.

1.1 Spiking Neural Networks

Since the availability of the data is not part of the network, we can consider it to be beyond our control. Then considering the other two drawbacks, we need to consider a variation of deep neural networks which has comparable performance as the existing methods, however, can be run with low power and computational resources.

An observation that can be made is that, even though deep neural networks are inspired by the brain and have very similar structures, the operation of the networks does not match the brain. A neuron in the brain remains inactive until it receives a sizeable stimulus from its synapses. Upon receiving

such stimulus the neuron will propagate the stimulus further. This implies that at any given point of time, only a fraction of neurons will be active. This makes the whole structure very energy efficient.

The neurons also have a very simple operation. They send an output if they receive a significant input, otherwise they just wait for the inputs to increase. Such a simple structure allows the neurons to yield good results without complicated operations.

The observations can be combined to come up with an idea of a new type of network that can operate with high accuracy, low dependency on computational components and high energy efficiency. These types of networks are often categorised as neuromorphic systems and are often labelled as *third generation of neural networks*.

One of the more popular networks of this kind is *Spiking Neural Networks*. These networks are designed to be similar to the brain in the structural sense as well as operational sense. Spiking Neural Networks operate on *spike trains* instead of traditional vectors. The spikes are considered to be distinct events which occur independently. A spike train is a sequence of spikes. Each neuron of the network accepts these spikes as input and produces an output spike train which is fed to the next layer. This operation is similar to the traditional deep neural networks with an additional temporal nature of the operation. The neurons perform operations only upon receiving a spike from the previous layers and is in a waiting state when no spikes are received.

The spiking neural networks, unlike traditional neural networks, are run on special hardware. These are relatively cheaper due to being made for a specific set of computations instead of a general processing unit. The efficiency of the model comes from the fact that for any input, only a fraction of the neurons will be operational, which drastically reduces the energy requirement.

This low power operation and existence of special hardware enables these networks to take advantage of the edge computing paradigm. For example, let us consider a self operating drone. The drone takes images using cameras and sensors and then sends the data to a remote server. The server will then process the data and generate the output which will be sent back to the drone to make the decision. This method suffers from latency and dependency issues as, without the server communication it cannot operate. If we construct a spiking neural network for the same, it will be a simpler hardware with low power requirement. This will enable us to place the network on the drone itself, saving large amounts of power and time with almost no trade-off of accuracy.

Similarly, there exist many other problems where the portability and robustness offered by spiking neural networks can be utilized to get better solutions. Spiking neural networks have also been shown to provide better accuracy in the context of image and audio detection. This is due to the fact that the brain uses changes in images and audio as cues for detection. These cues are equivalent to spikes of the network, so the network is also good at the same. All of these applications make spiking neural networks a useful model to consider for many problems.

1.2 Need for Verification of Neural Networks

Due to the inference efficiency offered by the different neural networks, many safety critical systems like self driving vehicles, automatic guidance systems and assistance tools employ these neural networks. Being able to work well with complicated spacial relations are often requirements for such problems which, the deep neural networks provide. Due to the extreme safety requirements of such systems, failure of these systems can be catastrophic. To ensure that the model to be deployed is not susceptible to failure, verification of such systems is conducted. A primary objective of this thesis is to explore formal methods for verification and robustness analysis of Spiking Neural Networks.

Formal Verification of a system is a framework where a given system is analysed along with a set of properties to check which of the properties are satisfied by the system. These properties are behaviours of the system that guarantee safety of the system. Unlike traditional testing paradigms, instead of generating input test cases to test for the satisfiability of the properties, formal methods try to express the system mathematically and check the property on that mathematical encoding. The traditional testing paradigms are simpler to implement as they just require a few samples from the input space, but they lack the guarantee of satisfiability for all inputs. In order to achieve this guarantee, for safety critical systems, we use formal verification.

The properties to be checked are usually of the form, as mentioned below:

- System specific – These properties refer to the operation of the system without any constraints on the input space. Example of such a property is, "For all valid inputs, does the system always output a valid action?". These properties make sure that the system is sound.
- Problem specific – These properties refer to the expected output of the system for a specific subset of the input space. Example of such a property is, "If the camera is obstructed, does the system output the action halt?"
- Robustness – These properties refer to the system operating correctly after the introduction of some perturbation to the inputs. Example of such a property is, "Does the system give the same output for an input with and without a perturbation of 0.5?"

1.3 Formal verification of Spiking Neural Networks

The requirements of formal methods for verification not only apply to traditional neural networks but, to spiking neural networks as well, as they are also being used in safety critical systems. But unlike traditional neural networks, which have many tools and frameworks for verification, spiking neural networks suffer from a lack of such frameworks. This is mainly due to the recency of spiking neural networks.

Creating such frameworks for verification of spiking neural networks also poses a challenge as we cannot use the existing frameworks that exist for traditional neural networks. If we consider a single timestep of the entire execution of a spiking neural network, it is similar to a traditional neural network with inputs and outputs. This implies that the challenge of the problem stems from the temporal nature of the system. SNNs are temporal in nature in the way they process information over time. Thus, verification of SNNs is not limited to verifying correctness for a specific time step, but over all time steps of operation, which is hard to model, considering that time is continuous and the SNN is expected to run forever.

There are broadly two approaches to mathematically express the temporal nature of the system. These are:

- Using continuous time based modelling – These include automata that operate on timed words like Timed Automata [2]. The overall approach would be to model the operation of a network as a finite state timed automata and using the constructed automata for verification.
- Discretizing the operation of the system into finite timesteps which can be treated as interconnected discrete systems – Then we can use traditional verification paradigms on these discrete systems concurrently in order to create a verification framework.

These paradigms provide a mathematical base upon which further work can be done in order to encode the spiking neural networks and the verification properties. These encodings can be used to verify the encoded properties as well as create a framework that can be used for further structural and behavioural modification to the system. These modifications are often useful for the simplification of large networks. The encoding can also be used to check equivalence of two distinct networks as well as getting a better understanding of the model behaviours.

The only drawback with the above paradigms is that the size of the framework increases greatly with increasing complexity of the network and the simulation time. This poses a challenge as most verification tools start to fail at extremely large encodings. So the challenge is not just to come up with a verification framework, but a scalable verification framework.

1.4 Motivation of this dissertation

The lack of standard verification paradigms for spiking neural networks is a problem still not solved. This is a severe hurdle for the applications of these systems as, without a reliable verification framework, the systems cannot be used for validation of safety critical artifacts. The current frameworks that exist for traditional neural networks cannot be used for verification of spiking neural networks without some significant changes. In order to successfully utilize spiking neural networks and all the advantages it provides, we require a reliable modelling and verification paradigm.

Due to the infancy of these spiking neural networks, the primary works focus on getting better and more efficient systems both in terms of performance and portability. These works are a good depiction of the power and robustness offered by spiking neural networks but lack any discussion regarding the scalability and reliability of these networks.

Another orthogonal direction of work being done is hardware centric. These works focus on creating special processors and systems to improve the cost effectiveness of such systems as well as the hardware reliability. The hardware reliability parts of the works focus on system specific properties but still lack any notion of a framework for modelling such systems.

There also exist works which introduce some timed automata based encodings [6] for spiking neural networks along with a framework for verification of hardware properties. The work however lacks a scalable notion of models and suffers from a lack of robustness as well. These works however serve as a great starting point in order to tackle the modelling and verification frameworks for spiking neural networks.

The primary motivation of this thesis is to develop scalable and efficient encodings for Spiking Neural Networks by taking advantage of different paradigms. Specifically, this thesis has the following motivations:

- Propose a lossless encoding for spiking neural networks using timed automata which encapsulate the behaviours of spiking neural networks.
- Propose a lossless encoding for spiking neural networks as Satisfiability Modulo Theory (SMT) constraints that provide an expressive theory for symbolic encoding of the behaviours of spiking neural networks.
- Use these encodings in order to tackle various problems like verification, robustness analysis, simplification and equivalence of spiking neural networks.

1.5 Contributions of this dissertation

The objective of the thesis is to propose two encodings, based on timed automata and SMT for Spiking Neural Networks. Using these encodings, we tackle the problems of verification, abstraction-refinement, simplification and equivalence of the same. The contributions of this thesis are briefly described below:

- *SMT based Encoding*: Upon discretization of the spiking neural network operation, the functioning of the spiking neural networks can be encoded using linear real arithmetic. In this thesis we propose an encoding of an SNN in linear real arithmetic and demonstrate how the formulation of the systems mimics the behaviour of the original.
- *Timed automata based Encoding*: The temporal aspects of spiking neural networks are encapsulated well using timed automata. In this thesis we propose an encoding for spiking neural networks using timed automata. We encode the entire system as a large collection of smaller timed automata corresponding to each neuron and then demonstrate the identical behaviour of the encoding and the original network.
- *Applications of Encodings*: Using these encodings we can propose strategies for verification, adversarial robustness, simplification and equivalence checking of spiking neural networks. In this thesis we go over each proposed encoding and their equivalent frameworks for these applications.

1.6 Organization of the dissertation

This dissertation is organized into 5 chapters. A summary of the contents of the chapters is as follows:

Chapter 1: This chapter contains an introduction and a summary of the major contributions of this work.

Chapter 2: This chapter corresponds to the background and prerequisites of the work for all the topics discussed.

Chapter 3: This chapter describes the SMT based encoding of spiking neural networks.

Chapter 4: This chapter describes the timed automata based encoding of spiking neural networks.

Chapter 5: We summarize with conclusions on the contributions of this dissertation.

Chapter 2

Background and Related Work

In this chapter, we describe the background concepts and prerequisites related to the systems that are discussed and referenced throughout the thesis. We also demonstrate the working of the systems discussed to have a better understanding of the work described in the subsequent chapters. We finally describe the software tools and programming libraries used in the coming chapters.

2.1 Neural Networks

Neural Networks (NN), often called as Artificial Neural Networks or Deep Neural Networks, are the core of Deep Learning algorithms which is a prominent subset of Machine Learning [14]. A neural network is a model which is a collection of processing units called as *nodes* or *neurons*, interconnected by data transfer edges called as *synapses*. The etymology of neural networks comes from the similarity between the the architecture of the model and the human brain.

Neural Networks unlike traditional algorithmic models, rely on having a set of data available for training. This process of training is done before the model has been deployed. This process, in an abstract sense, is done by building the network on the training samples and updating the parameters of the network to reduce the errors and improve accuracy. These networks are powerful tools as they are very robust and can tackle problems that are difficult to solve with traditional algorithms.

There are many types of neural networks all of which have different architectures and applications. The most common one of these deep neural networks are feed-forward neural networks. A feed-forward neural network is made up of a collection of sequential layers each containing a number of nodes. Each of these layers is connected to the next layer and the previous layer. Upon receiving an input the first layer takes that input and after processing it passes it to the second layer. The process continues until the final layer outputs a vector. This vector is considered as the output of the entire network. The processing done by each layer of a feed-forward neural network is done in two steps as below.

- Weighted sum of the inputs : The synapses through which the data is transferred have an associated weight which is use to calculate the total input to every individual node. For a node v_i and previous layer with n nodes we have the input received at v_i as,

$$\text{Total input sum } = \sum_{j=1}^{n} w_{i,j} \cdot x_j$$

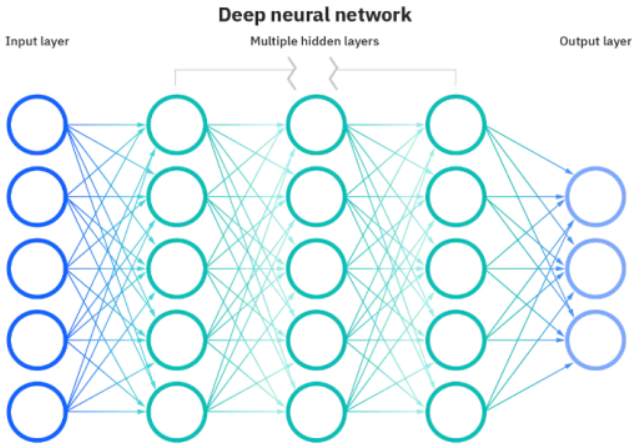


Figure 2.1: An abstract look of an Artificial Neural Network

where $w_{i,j}$ refers to the weight of the synapse between the neuron v_i and the j^{th} neuron of the previous layer and x_j refers to the input value received from the j^{th} neuron of the previous layer.

- Activation function : The neuron after receiving the inputs and performing a weighted sum, passes the total sum to an activation function. The output of these activation functions are sent to each neuron of the next layer. These activation functions can be linear or non-linear in nature and affect how the network behaves. Some activation functions are,
  - RELU (Rectified Linear Unit)
  $$y(x) = max(x, 0)$$
  - Absolute function
  $$y(x) = |x|$$
  - Sigmoid function
  $$y(x) = \frac{1}{1 + e^x}$$
  - Tanh function
  $$y(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Let us consider the small feed-forward network given in Figure 2.2. Let us assume that the activation functions of neurons are sigmoid activation functions. For an input vector $x = [2, 1]$ we have,

$$\begin{aligned} x_1 = 2,\ x_2 = 1 \\ a = sigmoid(-3x_1 + 3x_2) = sigmoid(-3) \approx 0.953 \\ b = sigmoid(-x_1 + 5x_2) = sigmoid(3) \approx 0.047 \\ c = sigmoid(2a + 4b) = sigmoid(2.094) \approx 0.89 \end{aligned}$$

So, for an input $[2, 1]$ the feed-forward network outputs the value 0.89.

These neural networks have many applications including but not limited to classification, pattern recognition, prediction of values as well as decision modelling. The study of these neural networks, it's variants and applications is one of the leading areas of research in the field of data science.

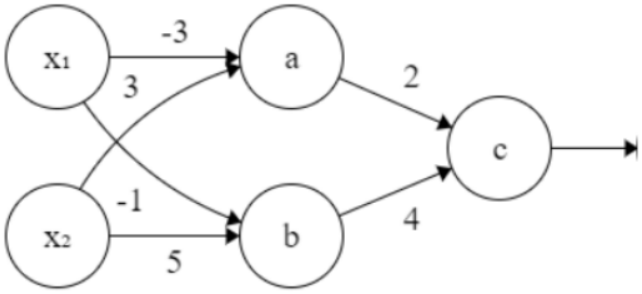


Figure 2.2: A small feed-forward network

2.2 Neural Network Verification

Due to the increasing popularity and power of neural networks over the past few years, neural networks are being considered for many safety critical problems [16]. These problems include but are not limited to, vehicular assistance systems, medical equipment operation, surveying and information retrieval for ecological data.

Failure of these systems can be catastrophic and cause destruction of property and endanger lives. This acts as the primary motivation for the introduction of formal verification. Formal verification is the process of mathematically modelling a hardware or software system in order to verify the correctness of the system with respect to some formal specifications or properties. This is often done, either by providing a formal proof or by mathematically modelling the system. Some of the many mathematical objects that are used for verification are, finite state machines, linear programs, graph structures, first-order logic, etc. Mathematically modelling a system, enables the exhaustive testing of properties, i.e. we can check the satisfiability of properties for all possible inputs without enumerating those inputs. Modelling complicated systems poses a challenge as encapsulating the nuanced behaviours of the system using mathematical objects may not always be possible. In these cases software and hardware testing paradigms yield better results.

2.2.1 Adversarial Robustness

Adversarial Robustness is a special subset of verification which deals with testing the robustness of a network against some adversarial noise added to the inputs [16]. A system is said to be adversarially robust if for any small perturbation of the input the output does not change. This is a very important part of the verification framework as networks need to be robust in order to avoid failures in it's execution like the instance shown in Figure 2.3.

Adversarial robustness of a system can be checked by using the same verification framework described above. Informally we can describe this property as, given an input vector x and network N, does the network output the same classification for all $\delta-$perturbations of x. The set of $\delta-$perturbations of x can be formally written as,

$$\{x' \ : \ -\delta \leq \|x' - x\|_\infty \leq \delta\}$$

where $\delta \in \mathbb{R}$. The set of $\delta-$perturbations of x are the set of vectors that are at most δ distance away from x.

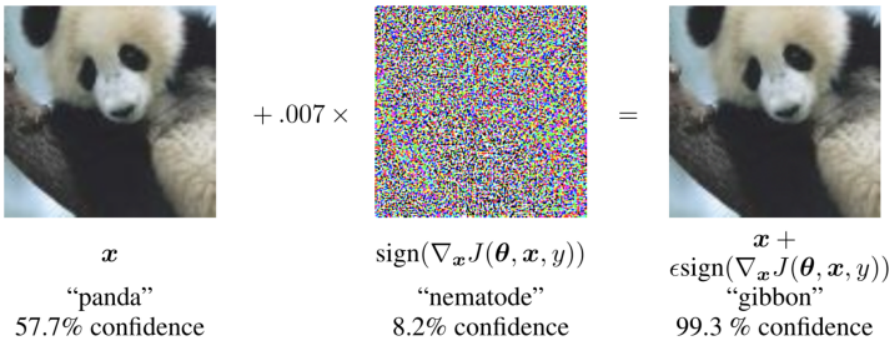


Figure 2.3: Adding a small amount of noise changes the output of a network

2.3 Spiking Neural Networks

Traditional neural networks suffer from a major drawback of high energy demand. This drawback makes these networks unsuitable for edge computing systems due to their lack of portability. This has motivated the development of a new variant of neural network that does not suffer from the high energy demand; Spiking Neural Networks (SNN).

A spiking neural network is similar to the human brain in not just the architecture but the functioning as well. Unlike traditional networks the data being propagated is not a real vector, instead it is a collection of timed events called as a *Spike train*. These spike trains are time based inputs, so the execution of an SNN occurs over a simulation time instead of occurring instantly.

The SNN receives the spike trains as input where the spikes are propagated forward through the network. The output of the SNN is also a spike train generated by the neurons of the last layer.

There are many different types of SNNs based on architecture and types of neurons. The architectures of SNN are similar to that of traditional neural networks, i.e. we can have Convolutional SNNs, Recurrent SNNs, Feed-forward SNNs, etc. The types of neurons an SNN has can be Leaky Integrate and Fire neurons, Synaptic, Lapicque or Alpha. All of these neurons have different operations and properties but all of them act on spike inputs.

One of the most common type of neuron for SNNs is Leaky Integrate and Fire neurons (LIFR) neurons. These neurons can be formally defined as below,

Definition 2.1 *An Integrate and Fire(IF) neuron is defined as a tuple* $\langle \theta, \lambda, \tau, p, y \rangle$ *where,*

- θ *is the firing threshold of the neuron.*
- τ *is the refractory period*
- λ *is the leak factor*
- $p : \mathbb{Q}^+ \to \mathbb{Q}^+$ *is the potential function defined as,*

$$p(t) = \begin{cases} \lambda \cdot p(t-1) + \sum_{i=1}^{n} w_i \cdot x_i(t) & ; \ p(t-1) < \theta \\ \sum_{i=1}^{n} w_i \cdot x_i(t) & ; \ p(t-1) \geq \theta \end{cases}$$

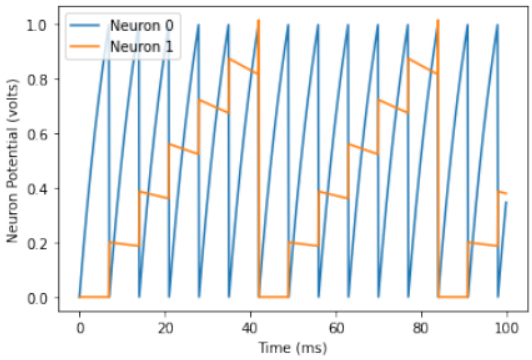


Figure 2.4: Neuron Potentials of 2 connected neurons

- $y : \mathbb{N} \to \{0,1\}$ *is the neuron output function given as,*

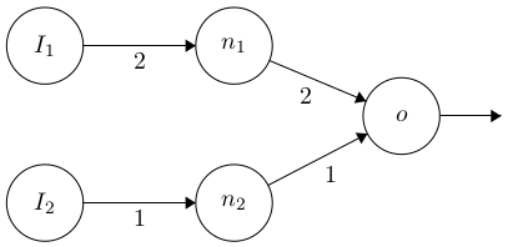
$$y(t) = \begin{cases} 1 & p(t) \geq \theta \\ 0 & p(t) < \theta \end{cases}$$

A LIFR neuron accumulates potential until a threshold is met, at which point it fires a spike. Upon receiving a spike from input neuron i at time-step t, the neuron will accumulate the potential corresponding to the weight of the synapse between the neuron and input neuron i (encapsulated by the potential function p). If the accumulated potential is greater than the firing threshold (θ) then the neuron outputs a spike (Output potential function y). After an output spike, the potential is reset and starts accumulating again after waiting for a small recovery period (Refractory period τ). If the LIFR neuron does not spike, the stored potential starts to reduce as time goes on (Leak factor λ).

Figure 2.4 demonstrates the working of these LIFR neurons. There are two LIFR neurons connected to each other through a synapse. Every time neuron 0 spikes the potential of neuron 1 goes up. When no inputs are received the potentials start to leak.

If we have $\lambda = 1$ then the neuron is called as Integrate Fire (IF) neuron. Most implementations of SNNs have very small refractory period(τ) which is mostly ignored during computations. Going forward we assume that $\tau = 0$.

As an example let us consider the following network,



I_1, I_2 are input spikes received, n_1, n_2, o are IF neurons all with the same firing threshold of $\theta = 3$. Also let $P(x)$ denote the potential of a node x. Now let us consider the input spikes as an array of timesteps when there is an input spike. For example,

$$I_1 = [0.6, 1.3, 2.4] \text{ and } I_2 = [0.4, 0.9, 1.8, 2.1]$$

Then we have the following execution for $t \in (0, 3)$ with $P(n_1), P(n_2), P(o)$ all initially 0.

- At $t = 0.4$, I_2 spikes,

$$P(n_2) = P(n_2) + 1 = 0 + 1 = 1$$

and since $P(n_2) < \theta$, n_2 does not spike.

- At $t = 0.6$, I_1 spikes,

$$P(n_1) = P(n_1) + 2 = 0 + 2 = 2$$

and since $P(n_1) < \theta$, n_1 does not spike.

- At $t = 0.9$, I_2 spikes,

$$P(n_2) = P(n_2) + 1 = 1 + 1 = 2$$

and since $P(n_2) < \theta$, n_2 does not spike.

- At $t = 1.3$, I_1 spikes,

$$P(n_1) = P(n_1) + 2 = 2 + 2 = 4$$

and since $P(n_1) \geq \theta$, n_1 does spike. This will imply that,

$$\begin{aligned} P(o) &= P(o) + 2 = 0 + 2 = 2 \\ P(n_1) &= 0 \end{aligned}$$

since $P(o) < \theta$, o does not spike.

- At $t = 1.8$, I_2 spikes,

$$P(n_2) = P(n_2) + 1 = 2 + 1 = 3$$

and since $P(n_2) \geq \theta$, n_2 does spike. This will imply that,

$$\begin{aligned} P(o) &= P(o) + 1 = 2 + 1 = 3 \\ P(n_2) &= 0 \end{aligned}$$

since $P(o) \geq \theta$, o does spike. So $P(o) = 0$.

- At $t = 2.1$, I_2 spikes,

$$P(n_2) = P(n_2) + 1 = 0 + 1 = 1$$

and since $P(n_2) < \theta$, n_2 does not spike.

- At $t = 2.4$, I_1 spikes,

$$P(n_1) = P(n_1) + 2 = 0 + 2 = 2$$

and since $P(n_1) < \theta$, n_1 does not spike.

So at the end o spikes once in the duration $t \in (0, 3)$ and at $t = 1.8$.

$$I_1, I_2 \xrightarrow{N} [1.8]$$

2.4 Spike trains

A spike train is a collection of timed events. These are often the data for neuromorphic problems. They are collected by using multiple sensors which detect changes on different stimuli.

These spike trains are often expressed as a sequence of timestamps for when the event occurs. For example, a spike train representation of an input that spikes every 1.3 secs would be $\{1.3, 2.6, 3.9, 5.1, \cdots\}$

We can encode a spike train as a fixed Boolean vector whose size is equal to the number of timesteps. These timesteps are the smallest unit of time for which no two events occur at the same timestep. We can do this by the following construction. Given a spike train I we define the Boolean vector I' as,

$$I'[i] = \begin{cases} 1 & i \in I \\ 0 & \text{otherwise} \end{cases}$$

The number of timesteps can be selected by calculating the number in a way that no two spikes of the same neuron that occur at different instances are mapped to the same timestep. Consider the spike trains,

$$I_1 = \{1.42, 2.16, 2.83, 4.94, 6.31\} \text{ and } I_2 = \{1.43, 2.82, 4.22\}$$

For the example we can infer that $\delta = 0.7$ and we get,

$$I_1 = \{0, 1, 1, 1, 0, 1, 0, 1\} I_2 = \{0, 1, 0, 1, 1, 0, 0, 0\}$$

2.5 Satisfiability Modulo Theories

Satisfiability Modulo Theories is a paradigm to determine if a given mathematical formula is satisfiable. It is essentially a generalization of the Boolean satisfiability problem (also known as SAT problem) by allowing more complex variables like integers and real numbers.

There exist many different SMT solvers that can solve formulas in Linear Real Arithmetic (LRA). A formula is said to be in LRA if it is of the form,

$$a_1 x_1 + a_2 x_2 + \cdots + a_k x_k \bowtie b$$

where $\bowtie$ can be $=, \neq, \leq, \geq, <, >$ and $\forall x_i,\ x_i, a_i \in \mathbb{R}$. These SMT solvers can compute satisfiability for formulas with extremely large number of variables. Here, $\mathbb{R}$ denotes the set of real numbers.

SMT solvers offer a new approach to verification of hardware and software systems. If we can successfully encode a system and it's properties into LRA formulas, we can then use these SMT solvers to check for satisfiability of the verification query. This method is used for verification of traditional neural networks with linear activation functions.

For the RELU activation function we have, we have,

$$y(x) = max(x, 0)$$

Since LRA does not support max as a function we encode this as,

$$(x \geq 0 \implies y = x) \wedge (x < 0 \implies y = 0)$$

We can observe that this formula encodes the behaviour of the RELU function. SMT solvers offer a great deal of verification capabilities and is also very scalable in nature.

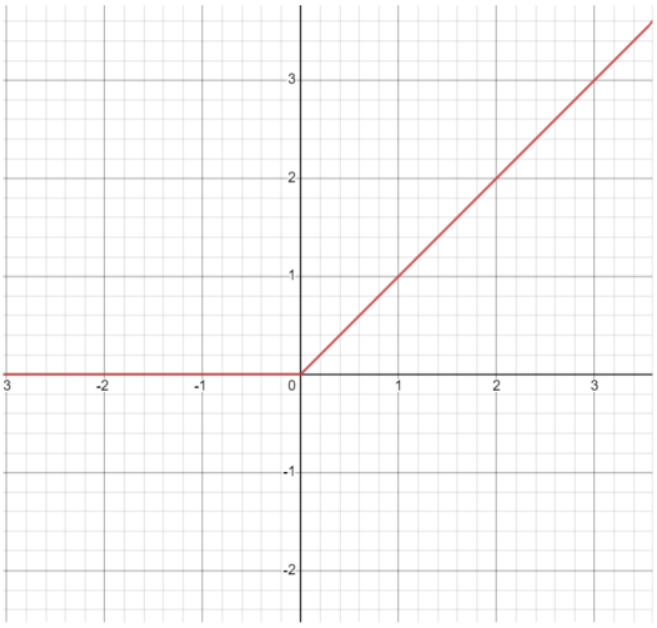


Figure 2.5: Graph for RELU activation function

2.6 Timed Automata

A timed automata is a mathematical object that operates on timed words. These timed words are words of the form,

$$\{(s,t) \ : s \in \sigma, \ t \in \mathbb{R}^+\}$$

It is structurally similar to a finite state automata and shares a lot of it's properties as well. Informally a timed automata is an extension of finite state automata with the additional functionality to handle clocks and clock-based constraints. Clocks are real value counters that are used to encapsulate the temporal aspect of the machine. Defining constraints on clocks allows the timed automata to accept timed words.

Formally a timed automata is defined as below,

Definition 2.2 *A **Timed Automata** TA is a tuple* $(L, l^0, X, \Sigma, Arcs, Inv)$*, where,*

- L *is the set of locations (states) with* l^0 *as initial location.*
- X *is a set of clocks.*
- Σ *is the set of communication labels (input symbols)*
- $Arcs \subseteq L \times (G \times \Sigma \times 2^{|X|}) \times L$ *is a set of transitions between locations.*
- $Inv : \ L \rightarrow G$ *assigns invariants to each locations.*

Here G is the set of guard conditions and $2^{|X|}$ refers to the reset conditions of all clocks. In other words, any subset of the clocks can be reset on an arc. The guard conditions are of the form,

$$g := c_1 \bowtie x \bowtie c_2$$

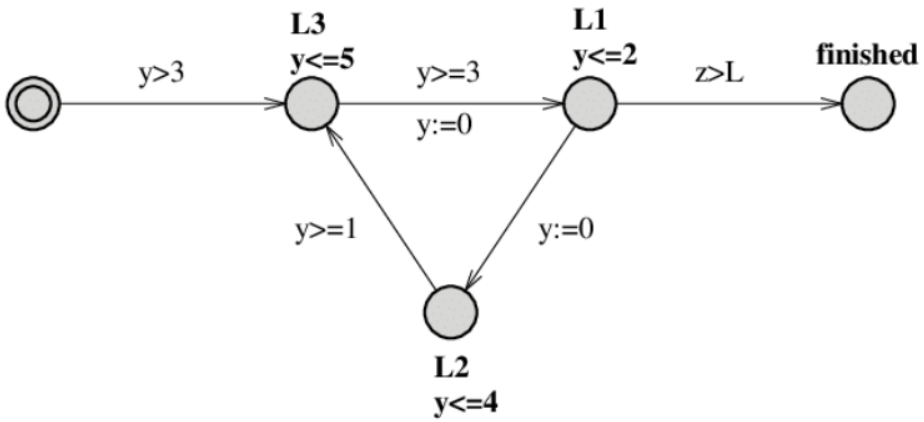


Figure 2.6: Example of a Timed Automata

where $c_1, c_2 \in \mathbb{R}^+$, $x \in X$ and $\bowtie$ can be $\leq, \geq, =, \neq, <, >$. The locations have their own clock invariants, which need to be satisfied while the machine is in that state. If the invariant of a state becomes false while the machine is in that state, it must take a transition to some other state or the execution dies if no such transition exists. Each transition (arc) has an associated clock guard, input symbol and clock resets. The clock resets refer to the subset of clocks that will be reset back to 0.

Since timed automata have the ability to encapsulate temporal nature of systems they are often good modelling tools for such systems. In Chapter 4, we use these to model SNNs.

2.7 Tools

In this section we describe the major tools and libraries used throughout the thesis work.

2.7.1 Z3

Z3r is an SMT solver developed by Microsoft Research [7]. Z3 is one of the state of the art solutions for SMT solving and can handle variables in the order of millions. Z3 has bindings in various programming languages like C,C++ and Python. We have used the Python library corresponding to Z3 which is named Z3py.

From Figure 2.7 we can see that Z3 can solve LRA formulas and returns *unsat* upon receiving a set of formulas.

2.7.2 Uppaal

Uppaal is an integrated tool environment that is used for the verification and modelling of real-time systems modelled as timed automata and extended timed automata [4]. The Uppaal tool has 4 components,

```python
from z3 import *

x = Int('x')   x: x
y = Int('y')   y: y
print("The solution to the formulas {x > 2, y < 10, x + 2*y == 7} is")
solve(x > 2, y < 10, x + 2*y == 7)
print("The solution to the formulas {x > 2, y > x, y < 1} is")
print(Solver().check(x > 2, y > x, y < 1))
```

(a) Sample z3 Python code

```
The solution to the formulas {x > 2, y < 10, x + 2*y == 7} is
[y = 0, x = 7]
The solution to the formulas {x > 2, y > x, y < 1} is
unsat
```

(b) Output of the sample code

Figure 2.7: Sample z3 execution

- GUI for making the automata and transitions along with their invariants and guards.
- A programming platform (C based) to model complex functionality that cannot be modelled using the GUI tool.
- A simulator that simulates the timed automata for a given timed word.
- A model checker that can run Linear temporal logic (LTL) and Computation tree logic (CTL) formulas in order to check properties on the modelled timed automata.

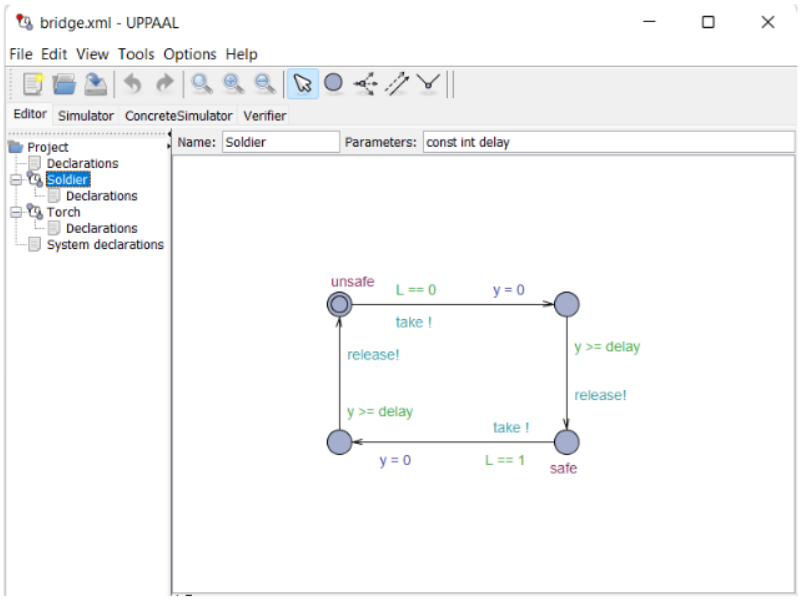
Figure 2.8 and Figure 2.9 provide a snapshot of the Uppaal framework.

2.7.3 snnTorch

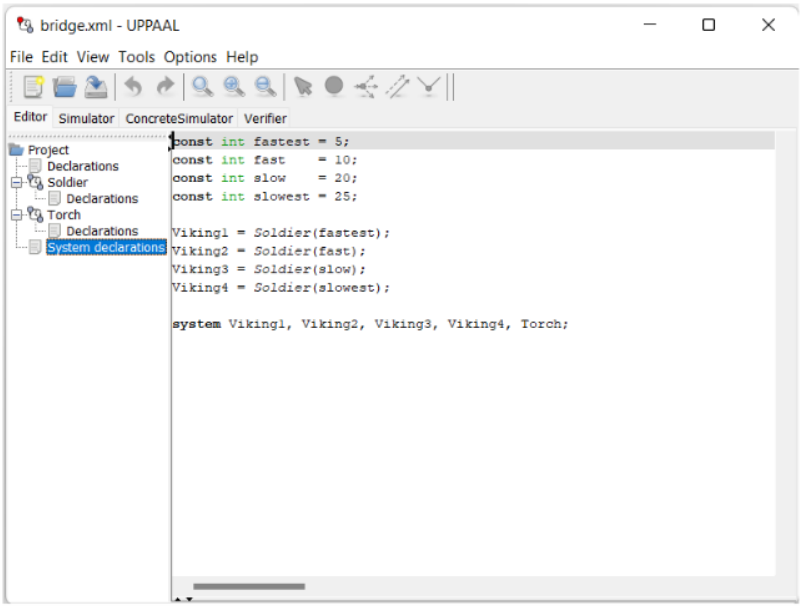
snnTorch is a Python framework built upon PyTorch in order to perform gradient based learning with spiking neural networks [11]. The library provides additional functionality to the existing PyTorch implementation allowing the user to create spiking neural networks.

The PyTorch package has support for many traditional neural network layers like, feed-forward layers, convolutions layers, recurrent layers but with snnTorch we have the additional functionality to have LIFR neurons, Lapicque neurons and Synaptic neurons.

The library can be used to simulate and train different types of SNNs. The simulation is done by breaking the entire simulation time into discrete steps, and then passing the spike trains as input one timestep at a time. The library also has additional functions for visualization and encoding of spike trains from non-spike train data.

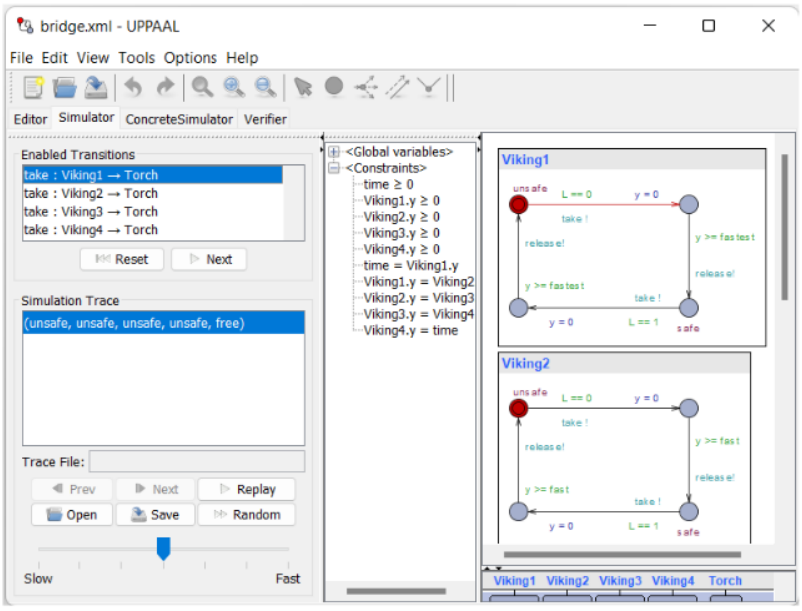


(a) Uppaal GUI

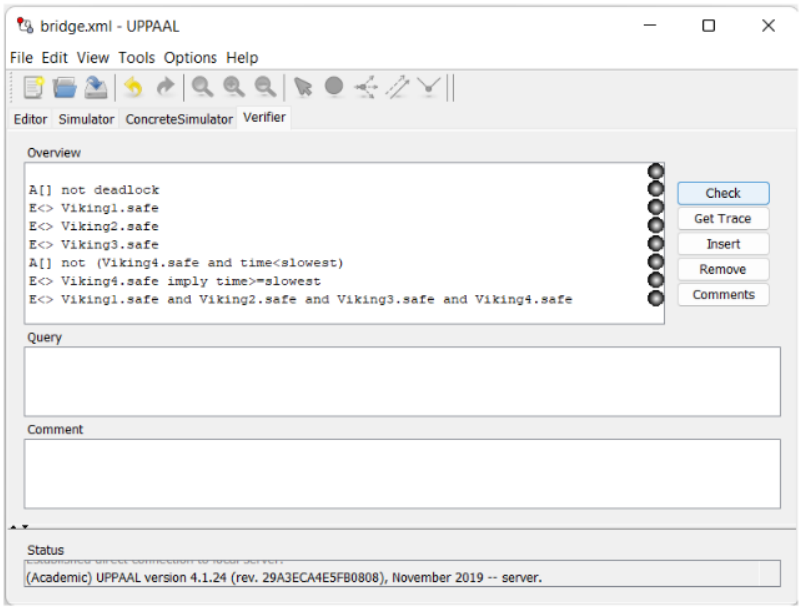


(b) Uppaal Codebase

Figure 2.8: Uppaal Modelling Components



(a) Uppaal Simulator



(b) Uppaal Verifier

Figure 2.9: Uppaal Verification Components

2.7.4 Brian2

Brian2 is a Python library that deals with the simulation of SNNs [20]. Unlike snnTorch or other simulators, Brian2 simulates SNNs from a hardware perspective. The simulation tool deals with physical quantities like voltage and current instead of dimensionless real values.

The simulation of Brian2 is done using these physical quantities as units and the functions which encapsulate the behaviour of the SNN are modelled using differential equations.

These functionalities enable a better understanding of SNN architectures and functioning.

2.8 Related Work

In this section we highlight some of the related work done for traditional neural networks as well as the works done for spiking neural networks.

2.8.1 Modelling of Traditional Neural Networks

There exists rich literature for modelling of traditional neural networks and modelling based frameworks for the same. In [18], Katz, et al, have discussed the modelling of neural networks using Linear Real Arithmetic. The work also discusses a verification framework based on the encoding proposed. Finally the paper discusses the notion of scalable verification and how the different architectures can enable more verification friendly networks.

In [10], an abstraction-refinement based verification for traditional neural networks is introduced. The paper talks about the use of over-approximation for property verification. The framework is an iterative process of constructing an over-approximation of the network followed by refining the abstracted network, while checking for the satisfiability of a property on the network. The over-approximation of the network is constructed in a way that if a property is not satisfied by the original system it will not be satisfied for the over-approximation either. The abstraction part of the framework is used for constructing smaller networks for verification and the refinement part of the framework is used to eliminate any spuriousness the model exhibits. The framework is a more efficient and scalable verification model for traditional neural networks.

There are many works that introduce different frameworks based on the verification of neural networks. One such work [19] discusses the pruning and slicing of neural networks with the help of the verification framework. The work defines the concept of redundant neurons. These are neurons that do not contribute to the network in a meaningful way. The work presents a sound framework to eliminate the redundant neurons using the verification framework.

Another framework for verification of neural networks introduced in [16] is the Reluplex method. The reluplex framework is a scalable verification tool that is based on the existing Simplex [16] framework used in Linear Programming. . Reluplex can handle the Rectified Linear Unit (ReLU) activation function, which is one of the most common activation functions used in traditional neural networks. The method applies the verification process to the entire network instead of applying it to some simplified network. The paper tests the model on neural networks designed for airborne collision avoidance for Unmanned Aerial Systems (ACAS Xu). The proposed framework is able to verify networks of a scale which could not be verified previously.

Another verification based framework is introduced in [12]. This framework is for simplification of

neural networks by the elimination of dead neurons. The paper defines a dead neuron as a neuron of the system which always outputs 0. This neuron does not contribute to the output of the network since the value output by the neuron is always zero. These neurons can be removed to create a smaller network without changing the behaviour of the original network. The framework uses network simulations in order to get a list of candidates that might be dead. Then it uses the verification framework to see if the candidates can ever output a non-zero value. The remaining neurons at the end are considered to be dead and can be removed from the network.

There also exists a framework for modification of a deep neural network [13]. Traditional neural networks need to be retrained every time there is a significant change in the inputs and outputs of a network. Since training of neural networks is not cheap, the paper presents a framework in order to modify the behavior of the network with minimal modifications. This framework uses formal verification to find the minimal amount of changes required to the network in order to account for the change in inputs and outputs.

2.8.2 Spiking Neural Networks

Spiking Neural Networks have gained popularity in the last few years due to the increasing applications of energy efficient neural networks. The physical hardware required for spiking neural networks are still under development with a few prototypes for non-commercial use. Despite this, there is an ever-growing literature base for spiking neural networks.

There is ample literature that describes the various spiking neural networks trained for different datasets with varying accuracy [21]. One such spiking neural network was trained for the handwritten digit recognition (MNIST) [8]. The paper demonstrates a learning method called as Spike-Timing-Dependent Plasticity (STDP). The network trained by this method on the MNIST dataset yields a model that has an accuracy of 95%. The work also uses unlabeled data for training (Unsupervised Learning).

There exist multiple works that compare the performance of spiking neural networks and different variations of the same. One such paper [17] talks about the tolerance of spiking neural networks and establishes a set of constraints for which the spiking neural network gives high accuracy, independent of non-ideal hardware operation.

With increasing popularity of SNNs, there are some methods to convert an ANN into an equivalent SNN. One such work [9], claims a higher accuracy of the converted network. In this work, the method for conversion of ANN to SNN is seemingly lossless. It introduces a different conversion technique that can be used to convert an existing ANN into an SNN with great accuracy.

Finally we look at a safety critical application of SNN. In [15], an SNN is constructed that takes images of signatures as input and labels them genuine or forged. A network like this can be used to automatically scan signatures on cheques and verify them. The paper discusses the architecture of the network along with its operation.

2.9 Novelty of this Dissertation Work

In this work we propose frameworks for the modelling and analysis of spiking neural networks. Since traditional neural network encodings cannot be used for spiking neural networks, we propose an SMT based encoding for spiking neural networks, that can successfully model SNNs. We uses this encoding to propose frameworks for simplification, verification and equivalence of SNNs.

We also discuss a timed automata encoding of SNNs. We finally introduce a simulation based verification framework for SNNs that utilizes the TA encoding of SNNs and the simulation tools for TA based systems.

Chapter 3

A Scalable framework for SNN verification based on Satisfiability Modulo Theories

This chapter presents our proposal on a SMT-based verification framework for SNN verification. Satisfiabilty Modulo Theories have offered a scalable verification framework for modeling, analysis and verification of a wide range of hardware and software systems, and are quite recently being used for modeling deep neural networks. Our exploration in this chapter intends to use SMT theories to model SNNs. In the discussion below, we begin with an intuitive example that illustrates the execution semantics of SNNs adopted in this chapter.

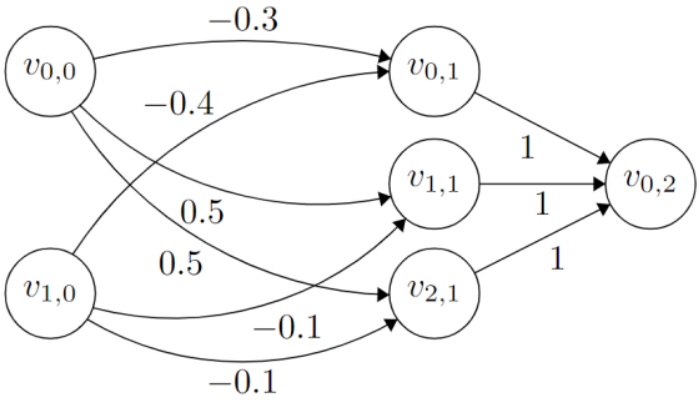


Figure 3.1: Small SNN

Example 3.1 *Consider the small SNN as shown in Figure 3.1 with the threshold of all neurons as 1 and initial potential as 0. Let us consider two input spike trains I_1 and I_2 as,*

$$I_1 = [1, 0, 1, 1]$$
$$I_2 = [1, 1, 0, 1]$$

As discussed before, a spike train depicts the firing of the inputs at specific time steps. The time axis is discretized into time steps, which correspond to the different positions in the spike train vector. An

input that fires at a specific time step is represented with a 1 at that specific position in the vector, with a 0 otherwise. Thus, for example, the input I_1 fires at time steps 1, 3 and 4 while the input I_2 fires at time steps 1, 2 and 4. The theory of this time discretization and spiking of neurons is discussed in Chapter 2.

As noted in Chapter 2, a spiking neuron transfers a potential equal to the weight of the edge to the neuron on the other side of the edge. A recipient neuron receives all such potentials from the incoming edges, computes the cumulative potential for itself and decides to spike or not based on its threshold. With these spike trains as inputs we will have the following simulation of the SNN.

- *At $t = 1$, both input neurons spike*
  - *Potential of $v_{0,1}$ = 0 - 0.3 - 0.4 = -0.7*
  - *Potential of $v_{1,1}$ = 0 + 0.5 - 0.1 = 0.4*
  - *Potential of $v_{2,1}$ = 0 + 0.5 - 0.1 = 0.4*
- *At $t = 2$, only $v_{1,0}$ spikes*
  - *Potential of $v_{0,1}$ = -0.7 - 0.4 = -1.1*
  - *Potential of $v_{1,1}$ = 0.4 - 0.1 = 0.3*
  - *Potential of $v_{2,1}$ = 0.4 - 0.1 = 0.3*
- *At $t = 3$, only $v_{0,0}$ spikes*
  - *Potential of $v_{0,1}$ = -1.1 - 0.3 = -1.4*
  - *Potential of $v_{1,1}$ = 0.4 + 0.5 = 0.9*
  - *Potential of $v_{2,1}$ = 0.4 + 0.5 = 0.9*
- *At $t = 4$, both input neurons spike*
  - *Potential of $v_{0,1}$ = -1.4 - 0.3 - 0.4 = -2.1*
  - *Potential of $v_{1,1}$ = 0.9 + 0.5 - 0.1 = 1.4*
    This neuron will spike at $t = 4$.
  - *Potential of $v_{2,1}$ = 0.9 + 0.5 - 0.1 = 1.4*
    This neuron will spike at $t = 4$.
  - *Potential of $v_{0,2}$ = 0 + 1 + 1 = 2*
    This neuron will spike at $t = 4$.

So the output spike train of the network is $[0, 0, 0, 1]$. ■

3.1 SNN encoding

In order to achieve a good encoding of any system, we need to ensure that the encoding is sound and complete. An encoding is,

- Sound – if every property of the encoding is also satisfied by the original system
- Complete – if every property of the original system can be encoded.

In order to encode an SNN we have to first model the execution semantics that can characterize the behaviors and properties of the network and then encode them. At the end we add additional properties if required to ensure soundness of the system.

For SMT based encodings of SNNs we use the theory of Linear Real Arithmetic (LRA), since the SNNs allow the edge weights to be real numbers. A collection of all LRA constraints encapsulate the behaviour of SNNs as shown in Figure 3.1. These formulas are often constructed for smaller units of the system and the conjunction of these formulas yields the encoding of the system. In the case of encoding neural networks, each neuron is encoded separately and the network encoding is the conjunction of all those individual encodings.

In this chapter we use a similar approach for the encoding of SNNs using LRA formulas. We consider the properties of each neuron and formulate them using LRA formulas. Then we create a conjunction of all properties in order to get our encoding of the entire SNN. However there arises the issue of the temporal aspect of the system. While writing LRA formulas we consider each neuron separately but, in this case, time is a continuous variable which cannot be expressed using LRA formulas. In order to work around this issue we divide the entire execution time of the system and divide it into discrete timesteps. These steps need to be small enough to ensure that no two events (spikes) can occur at one step, but also large enough such that it does not deeply impact the performance of the system. Often these steps can have the same length as the clock cycle of the hardware on which the SNNs are realized. A spike is a discrete event that occurs at a specific timestep. These spikes are the basic units for propagating information across a spiking neural network. A set of spikes can be called as a spike train, formally defined as below.

Definition 3.1 *A spike train I is a set of distinct spikes defined as,*

$$I = \{t \ : \ A \text{ spike occurs at } t\}$$

We can also have a Boolean representation of a spike train,

Definition 3.2 *A Boolean representation of a spike train is a vector whose size is equal to the number of steps and the values are given by,*

$$I[i] = \begin{cases} 1 & A \text{ spike occurs at } t \\ 0 & \text{otherwise} \end{cases}$$

Example 3.2 *Let us consider a spike train, $I_1 = \{0.8, 1.6, 2.0, 2.4, 3.6\}$. Here we divide the total simulation duration (4 sec) into timesteps of size 0.4 sec. So we can encode the array as a,*

$$I_1 = \{x_{0.4}, x_{0.8}, x_{1.2}, x_{1.6}, x_{2.0}, x_{2.4}, x_{2.8}, x_{3.2}, x_{3.6}, x_{4.0}\}$$

where x_i is 1 if I_1 spikes at timestep i, otherwise it is zero. So we get the Boolean encoding as, $I_1 = \{0, 1, 0, 1, 1, 1, 0, 0, 1, 0\}$. ■

In the further discussions, every reference to spike trains will refer to Boolean spike trains. For a given SNN with n layers, as in the Example 3.1, we define the following notations.

- The total time T.
- The total number of discrete steps, each of length δ as T/δ.

- The set $\{1, 2, \cdots, x\}$ as $[x]$.
- The number of nodes in layer i as m_i.
- The decay factor as λ.
- The i^{th} neuron of the j^{th} layer as $v_{(i,j)}$
- The threshold of $v_{(i,j)}$ as $\theta_{(i,j)}$
- The indicator variable for the spike generated by the i^{th} node of the j^{th} layer at timestep t as $x_{(i,j,t)}$
- The accumulated potential of node $v_{(i,j)}$ at time t as $P_{(i,j,t)}$
- The weight of the synapse between the i^{th} node of the j^{th} layer and the k^{th} node of the $(j-1)^{th}$ layer as $w_{(i,j,k)}$
- The potential gained by node $v_{(i,j)}$ at timestep t as $S_{(i,j,t)}$ where,

$$S_{(i,j,t)} = \sum_{k=1}^{m_{j-1}} x_{(k,j-1,t)} \cdot w_{(i,j,k)}$$

For a neuron $v_{(i,j)}$, if it receives a spike from a node $v_{(k,j-1)}$ then the weight of the synapse between $v_{(i,j)}$ and $v_{(k,j-1)}$ will be added to the potential of $v_{(i,j)}$. If no spike is received from $v_{(k,j-1)}$, no potential is added.

With all the notations defined, we can look at the properties of the neurons to be modelled. For each neuron $v_{(i,j)}$ and each timestep t we have,

- **A neuron spikes if the potential of the neuron crosses the threshold of the neuron**.
We know that the total potential of a neuron at any given point in time is the accumulated potential from the input synapses and the previously stored potential. So we have,

$$\text{Total Potential} : S_{(i,j,t)} + P_{(i,j,t-1)}$$

So we have the following LRA expression corresponding to this property,

$$\left(S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)}\right) \geq \theta_{(i,j)} \implies (x_{i,j,t} = 1)$$

In this formulation, the stored potential added is the decayed stored potential. As time passes, the neuron potential starts to leak causing the stored potential to decay. The leak factor denotes the fraction of potential retained after the previous timestep.

- **A neuron does not spike if the potential of the neuron does not cross the threshold of the neuron**.
Similarly for this property we have the LRA expression,

$$\left(S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)}\right) < \theta_{(i,j)} \implies (x_{i,j,t} = 0)$$

- **On crossing the threshold, the neuron potential is set to 0**.
If the threshold is crossed then the neuron potential is reset. This property can be expressed with the LRA formula,

$$\left(S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)}\right) \geq \theta_{(i,j)} \implies (P_{i,j,t} = 0)$$

- **If the potential of a neuron does not cross the threshold the potential is stored for the next timestep**
  This can be expressed as,

$$\left(S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)}\right) < \theta_{(i,j)} \implies \left(P_{(i,j,t)} = S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)}\right)$$

- **The initial potential of a neuron is 0.**
  The system starts with empty potentials. This can be expressed by the following LRA formula,

$$P_{(i,j,0)} = 0$$

Combining these formulas for a single node $v_{(i,j)}$ and single timestep t we can write the conjunction as,

$$\begin{aligned} F_{(i,j,t)} \triangleq \big(S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)} \geq \theta_{(i,j)} \implies (x_{i,j,t} = 1) \wedge (P_{(i,j,t)} = 0)\big) \wedge \\ \big(S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)} < \theta_{(i,j)} \implies (x_{i,j,t} = 0) \wedge (P_{(i,j,t)} = S_{(i,j,t)} + \lambda \cdot P_{(i,j,t-1)})\big) \end{aligned}$$

Consider a SNN with n layers, layer i having m_i nodes. For each layer i we have the encoding as,

$$F \triangleq \left(\bigwedge_{t \in [T/\delta]} \bigwedge_{j \in [n]} \bigwedge_{i \in [m_j]} F_{(i,j,t)} \right) \wedge \left(\bigwedge_{j \in [n]} \bigwedge_{i \in [m_j]} P_{(i,j,0)} = 0 \right)$$

At first glance the set of formulas can seem extremely large and complex, since the order of the number of variables is $O(n \cdot m \cdot t)$ where n refers to the number of layers, m refers to the maximum number of nodes in a layer and t refers to the number of timesteps. Even though the number of variables is in the order of 10^4 for small networks and 10^6 for large networks that are available in practice, the SMT solvers work seamlessly and show extremely good performances upto 10^8 variables.

It is intuitively easy to see that the SMT encoding is sound and complete for the time T, since the functionality of each SNN neuron is transferred to a lossless encoding as a collection of terms in the LRA theory. We thus have a sound and complete encoding and an associated decision procedure that employs a SMT solver to evaluate a given SNN for the time T against any given input spike train.

3.2 SNN Simplification

Modern Spiking Neural Network applications are increasingly growing to be large and complex. Thus, there is a continuous growing need for scalable procedures that can handle SNNs at scale. To this effect, we define a novel procedure for simplification of a given SNN that is driven by the observation that not all neurons in a given SNN fire, in other words, there may be neurons in a given SNN that may never cross their threshold. The challenge is in automatically identifying and pruning such neurons for a given SNN. Once such neurons are identified, we get an equivalent SNN which has lesser number of neurons and therefore, synapses. Before we talk about our proposal of dead neuron identification and elimination, we define the notion of equivalent SNNs.

Definition 3.3 *Two Spiking Neural Networks N_1 and N_2 are said to be **Equivalent** if $\forall x \in X$ we have, $N_1(x) = N_2(x)$.*

Where X is the input space given by, $\{0,1\}^{m\times t}$ where m denotes the number of input neurons and t denotes the number of timesteps. This means that if two SNNs have the same output spike train for all possible inputs, the networks are equivalent.

Equivalence of two SNNs N_1 and N_2 does not guarantee identical architecture, it just guarantees that the outputs of the networks are identical for all inputs. This indicates that two networks with different architectures and parameters may still be equivalent to each other. This notion of equivalence is useful for verifying the soundness and completeness of modification based frameworks applied to SNNs.

The equivalence of two networks can be checked by the following procedure,

- For the given two SNNs N_1 and N_2 we trivially check if the number of input neurons and output neurons are same for both the networks. If they are not we can conclude that the two networks are not equivalent, as they do not share the same input/output space.
- Then we construct the SMT encoding for both the networks. Then we construct the following formula, $\forall x \in X$,

$$Eq \triangleq (N_1(x) = y_1) \wedge (N_2(x) = y_2) \wedge (y_1 \neq y_2)$$

- We then use an SMT solver to check for satisfiability of Eq.
- If the SMT solver returns UNSAT we can say that the there exists no assignment of input x for which the two networks have different output (i.e. $y_1 \neq y_2$). So we can conclude that the two networks are equivalent.
- If the SMT solver returns SAT we can say that there exists an assignment of input x for which the two networks have different outputs. So we can conclude that the two networks are not equivalent. The solution returned by the SMT solver in this case will be the input for which the two outputs differ from each other.

3.2.1 SNN Simplification

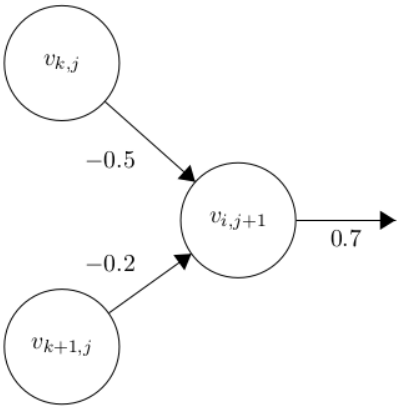
We now describe our approach towards SNN simplification. We begin with the definition of a dead neuron.

Definition 3.4 *A **Dead neuron** is a neuron $v_{i,j}$ of a network N where for every possible spike train,*

$$\forall t \in [T/\delta],\ P_{(i,j,t)} < \theta_{(i,j)}$$

This means that, for every possible input and all timesteps, the potential of the neuron does not cross it's threshold. This is equivalent to saying that a dead neuron is a neuron which never spikes for any possible input to the network.

Example 3.3 *Let us consider the following neuron with $\theta_{(i,j+1)} = 1$,*



In this example, whenever $v_{k,j}$ or $v_{k+1,j}$ spike, the potential of $v_{i,j+1}$ goes down by the formula

$$P_{(i,j,t)} = (-0.5)x_{(k,j,t)} + (-0.2)x_{(k+1,j,t)} + P_{(i,j,t-1)} \text{ and } P_{(i,j,0)} = 0$$

Since the potential $v_{i,j+1}$ is a decreasing function, the potential can never cross the threshold which implies that the neuron will never fire. So $v_{i,j+1}$ is a dead neuron. ■

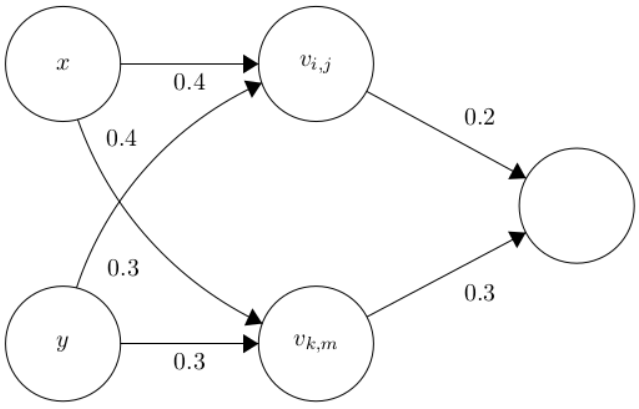
A dead neuron does not contribute to the network in any way since no output is ever generated by the neuron. During operation these dead neurons stay idle as the potential is never met and hence can be removed from the network to further simplify it. Removal of a dead neuron does not affect the output of the network as no part of the output was dependent on them. As a first step, we propose to remove such dead neurons. Next, we define the concept of equivalent neurons.

Definition 3.5 *Two neurons $v_{i,j}$ and $v_{k,m}$ are said to **equivalent** if, for every possible spike train,*

$$\forall t \in [T/\delta],\ (x_{(i,j,t)} = 1) \Leftrightarrow (x_{(k,m,t)} = 1)$$

This means that, for each possible input and all timesteps, equivalent neurons have the same output spike train. Equivalent neurons essentially act as copies of each other. In the context of a human brain each neuron acts as an indicator that contributes to the final output. Having two identical indicators for the same stimulus is redundant in nature and increases the size of the network without improving it's efficiency.

Example 3.4 *Let us consider the following example,*



Here, $v_{i,j}$ and $v_{k,m}$ have the same input weights and neurons. In this case if any one of them spikes, the other will also spike since the potentials of the two neurons are the same at every timestep. This can be observed by looking at the formula corresponding to the potentials of the neurons,

$$P_{(i,j,t)} = P_{(i,j,t-1)} + 0.4x + 0.3y$$
$$P_{(k,m,t)} = P_{(k,m,t-1)} + 0.4x + 0.3y$$
$$P_{(i,j,0)} = P_{(k,m,0)} = 0$$

Solving these relations we can observe that,

$$\forall t \in [T/\delta],\ P_{(i,j,t)} = P_{(k,m,t)}$$

So the neurons $v_{(i,j)}$ and $v_{(k,m)}$ are equivalent. ■

Merging duplicate neurons of the same layer into a single neuron will reduce the size of the network while still maintaining the original properties. However, we have to make some additional adjustments to the weights to ensure the equivalence, as discussed in the following subsection.

Now we define the simplification problem for SNNs.

Definition 3.6 *Given an SNN, N, we need to construct an SNN, N', such that $|N'| \leq |N|$ and $\forall x \in X,\ N(x) = N'(x)$.*

Here X denotes the input space which can be written as $\{0,1\}^{(m\times[T/\delta])}$ where m refers to the number of neurons in the input layer of N. For a given SNN N, $|N|$ stands for the number of neurons in N. We can construct a simplified network by removing the dead neurons and then merging the equivalent neurons. Since both these operations do not alter the output of the network, we are expected to get a network with a lesser (or same) number of neurons (in case there are no dead neurons or equivalent neuron pairs) which yields the same output as the original for every input.

3.2.2 Identifying Dead Neurons

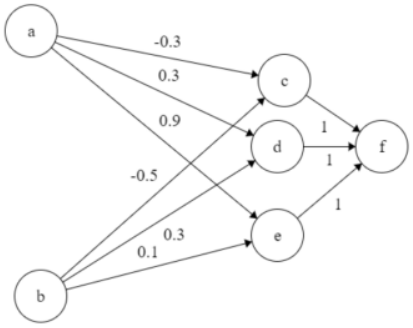
For the removal of dead neurons we have the following procedure,

- Randomly generate a fixed set of inputs for the network N. Let us denote the set of inputs as I. The size of I, i.e. the number of inputs should be proportional to the size of N where, a large network will require a large set of inputs whereas a smaller network will just require a small set.
- We initialize a list of possible dead neurons and insert all the neurons of N as candidates to be dead neurons.
- We run the inputs from I on N and note the spike trains of the neurons. If a neuron v has at least one spike, then we remove it from the candidate list since it cannot be a dead neuron if it spikes.
- After the simulations we get a list of candidate dead neurons. We use the following query to check if these neurons are dead. This can be done by invoking a solver to check for satisfiability of the following expression for neuron $v_{(i,j)}$:

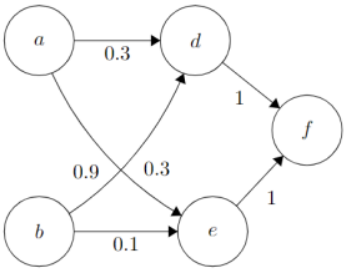
  $$\exists t \in [T/\delta],\ P_{(i,j,t)} \geq \theta_{(i,j)}$$

  along with the network formulas of the previous layers. The layers after the dead neuron do not impact the spikes of the candidate, hence they can be ignored.

- If the solver returns UNSAT, we can infer that there exists no input assignment for which the candidate neuron's potential crosses its threshold. This implies that the neuron never spikes and hence can be labelled as a dead neuron.
- If the solver returns SAT, we can infer that there exists some input for which the candidate neuron's potential crosses its threshold. This implies that the neuron can spike and hence cannot be labelled as a dead neuron. This candidate is removed from the list. This step ensures that no live neuron is labelled as dead because of unfavourable random inputs.
- We then construct a network N' from N but by removing all the dead neurons and their associated synapses.



(a) Sample SNN before dead neuron elimination



(b) Sample SNN after dead neuron elimination

Figure 3.2: Sample for dead neuron elimination

Example 3.5 *In order to demonstrate the working of the process of eliminating dead neurons, let us consider the network in Figure 3.2a. The execution will be as follows,*

- *Let us say we generate a random input* $I_1 = [0, 1, 0]$ *for neuron* a *and* $I_2 = [1, 0, 1]$ *for neuron* b.
- *We initialize the set of candidate dead neurons as* $\{c, d, e, f\}$
- *After simulating the input we can observe that neurons* e *and* f *did spike. So we eliminate them from the dead neuron candidate list.*
- *We now apply the SMT solver to check if the neuron is dead or not for the candidate list* $\{c, d\}$,
  - *For* d *the solver will return SAT as it's output. This is due to the fact that there exists an input spike train for which* d *spikes. The solver will return one such assignment, say,* $I_1 = [1, 1, 1]$ *and* $I_2 = [1, 1, 1]$. *So we remove* d *from the dead neuron candidate list.*

– *For c the solver will return UNSAT as its output. This implies that there is no input spike train for which c spikes. So we conclude that c is a dead neuron and it can be eliminated.*

On removal of the dead neuron we get the resulting network in Figure 3.2b. ■

Algorithm 1 presents our procedure for dead neuron handling. It is intuitively easy to see the following Theorem.

Theorem 3.1 *The SNN produced after dead neuron elimination is equivalent to the original SNN for all spike trains.*□

3.2.3 Handling Neuron Equivalence

For the merging equivalent neurons we have the following procedure,

- We generate the set of inputs I similar to the the ones used for eliminating the dead neurons.
- We initialize groups corresponding to each layer and insert all the nodes of a layer into their corresponding groups.
- We then check for equivalent neurons by comparing the spike trains obtained from running I through N. If two or more neurons of the same layer have different spike trains for any input we put them in separate groups.
- At the end of the simulation we have groups corresponding to each layer. If a neuron is the only member of its group, then it is a unique neuron without an equivalent neuron in the same layer. We consider each group containing more than one neuron.
- We then check the following formulas for all the neuron pairs in each group G,

$$\forall i_1, i_2 \in G, \exists t \in [T/\delta],\ x_{(i_1,j,t)} \not\Leftrightarrow x_{(i_2,j,t)}$$

- If for any formula the solver returns UNSAT we can say that for all possible inputs the output spike trains of the two neurons are identical. This would imply that the two neurons are equivalent.
- If the solver returns SAT we can say that, there exists an input for which the two neurons output different spike trains, which would imply that they are not equivalent.
- We consider the each of the final groups as G with neurons $\{v_{(i_1,j)}, v_{(i_2,j)}, \cdots, v_{(i_k,j)}\}$. We can merge them into a new neuron $v_{(i,j)}$ with threshold as follows,

$$\theta_{i,j} = \frac{\sum_{v_{(i',j)} \in G} \theta_{(i',j)}}{|G|}$$

and we merge all the input synapses for the neurons with the following weight assignment,

$$w_{i,j,k} = \frac{\sum_{v_{(i',j)} \in G} w_{i',j,k}}{|G|}$$

then we finally we merge all the input synapses for the neurons with the following weight assignment,

$$w_{i,j,k} = \sum_{v_{(i',j)} \in G} w_{i',j,k}$$

In an abstract sense, the neurons of a group are merged into a single neuron whose weights and thresholds are taken as the average of all the individual neurons of the group.

- We construct a new network N" from N' (obtained from the original network N after dead neuron elimination) by replacing the equivalent neurons with the associated merged neuron.

Theorem 3.2 *The merged neuron will behave identical to all the equivalent neurons for any spike train.* □

Proof 3.1 *Let us consider a set of equivalent neurons G containing k neurons to be merged. Let us also denote the merged neuron as v_G. In order to show that v_G encapsulates the behaviour of all neurons in G we show that whenever the neurons in G spike, v_G will also spike and vice versa. Let the set of timesteps for which the neurons in G spike are denoted by T_{spike} and the set of timesteps for which the neurons in G do not spike are denoted by T'_{spike}. We have the following, $\forall t \in T_{spike}$ and $\forall v_{i,j} \in G$,*

$$\sum_{l=1}^{m_{j-1}} w_{i,j,l} \cdot x_{l,j-1} \geq \theta_{i,j}$$

Adding the equations corresponding to all the nodes together and dividing by k we get,

$$\frac{1}{k} \sum_{v_{i,j} \in G} \sum_{l=1}^{m_{j-1}} w_{i,j,l} \cdot x_{l,j-1} \geq \frac{1}{k} \sum_{v_{i,j} \in G} \theta_{i,j}$$

Which is equivalent to,

$$\sum_{l=1}^{m_{j-1}} w_{(G,l)} \cdot x_l \geq \theta_G$$

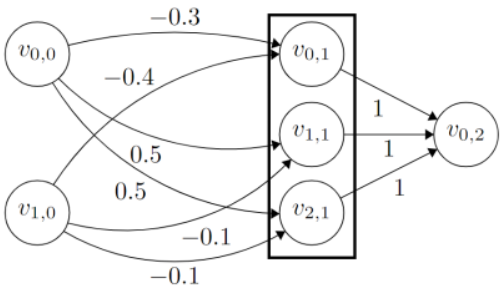
So whenever the neurons to be merged spike, the emerged neuron will also spike. By changing the sign of the inequality we can prove the other half of the proof as well. As for the

□

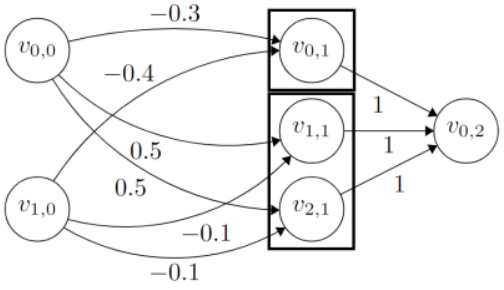
In order to demonstrate the working of the process of merging the equivalent neurons, let us consider an example.

Example 3.6 *Let us consider the example SNN in Figure 3.3a with thresholds of all neurons as 1 and the initial potentials as 0. The execution will be as follows,*

- *Let us say we generate a random input $I_1 = [1, 1, 1]$ and $I_2 = [1, 0, 1]$.*
- *We initialize the group as $\{v_{0,1}, v_{1,1}, v_{2,1}\}$.*



(a) Initial grouping



(b) Final groups

Figure 3.3: Sample SNN for the merging of equivalent nodes

- *After simulating the input we can observe that neurons $v_{0,1}$ and $v_{1,1}$ have different spike trains. We also notice that $v_{1,1}$ and $v_{2,1}$ have the same spike train. So we group them accordingly as shown in Figure 3.3b*

- *We now apply an SMT solver to check if the two neurons are equivalent or not. The model will return UNSAT. This implies that for all inputs the spike trains of the two neurons will always be the same, hence they are equivalent. So these neurons can be merged.*■

At the end of the procedure we get a new network N' for which the number of neurons is less than N. Once the neuron equivalence is established, it is easy to see that the network generated after merging of equivalent neurons is equivalent to the original one. For N and N' we have the following lemma,

Lemma 3.1 *The network N' is equivalent to the network N.* □

By observing algorithm 1, we can observe that initialization is $O(|N|)$, simulation is $O(|N| \cdot I)$ and the query execution using the solver and construction of the new network is $O(|N|)$ in the number of solver calls. So we can say that elimination of dead neuron makes $O(|N|)$ solver calls. For algorithm 2, the initialization is $O(|N|)$, simulation is $O(|N|^2 \cdot I)$ and the check using the solver and construction of the new network is $O(|N|^2)$ in the number of solver calls. So we can say that merging of equivalent neurons makes $O(|N|^2)$ solver calls. Combining the two we get the overall complexity of simplification to be $|N|^2$ in the number of solver calls.

Algorithm 1 Algorithm for removal of Dead Neurons

Input: The Spiking Neural Network N, Number of inputs for simulation I
Output: Reduced Spiking Neural Network N' without dead neurons.

```
candidates ← []
for i = 1 to i = |N| do
    candidates.add(i)                          ▷ Add the node i to candidate list
end for
c ← 0
while c < I do
    c ← c + 1
    ip ← generateRandomInput()                 ▷ Select a random input to simulate
    spiked ← simulate(N, ip)                   ▷ Simulate the input on the network
    for node in spiked do
        candidates.remove(node)                ▷ Remove neuron i from the candidate list
    end for
end while
for node in candidates do
    if checkSpike(N,node) == SAT then          ▷ Check if node can fire
        candidate.remove(node)
    end if
end for
newNetwork = removeNeurons(N, candidate)       ▷ Construct the new network
return newNetwork
```

3.3 Verification of SNNs

Formally we define the verification problem for Spiking Neural Networks as follows:

Definition 3.7 *Given a tuple $\langle P, Q, N\rangle$, where $N : \{0,1\}^{m\times t} \to \{0,1\}^{n\times t}$ is the SNN, $P : \{0,1\}^{m\times t} \to \{0,1\}$ is the input property and $Q : \{0,1\}^{n\times t} \to \{0,1\}$ is the output property the verification problem is to decide if there exists an $x_0 \in \{0,1\}^{m\times t}$ for which,*

$$P(x_0) \wedge (N(x_0) = y) \wedge \overline{Q(y)}$$

is satisfiable.

Here t refers to the number of timesteps, m refers to the number of input neurons of N and n refers to the number of output neurons of N.

If there exists such an x_0 we say that the system does not satisfy the given property. The input x_0 that was found, acts as the counter-example for the property. If there exists no such x_0 we can say that the system satisfies the property as there exists no inputs for which $\overline{Q(x)}$ is satisfied.

In this section, we describe the verification framework enabled by the SMT encoding of SNNs. The idea of verification is exactly the same as in case of DNNs as mentioned in Chapter 2. We show how the encoding in tandem with the solvers can yield a reliable framework for verification.

In order to utilize the SMT solvers, we need to encode each part of the formula using LRA formulas. Considering each part of the formula we have,

- $P(x_0)$ – input property
  The input property refers to the specific subset of inputs for which the property is to be checked.

Algorithm 2 Algorithm for merging of Equivalent Neurons

Input: The Spiking Neural Network N with n layers
Output: Reduced Spiking Neural Network N' without equivalent neurons

```
groups ← []
for i = 1 to i = n do
    for j = 1 to j = n_i do
        groups[i].add(j)                          ▷ Add the node i to candidate list
    end for
end for
c ← 0
while c < I do
    c ← c + 1
    ip ← generateRandomInput()                    ▷ Select a random input to simulate
    spikes ← simulate(N, ip)                      ▷ Simulate the input on the network
    for node1 in spikes do
        for node2 in spikes do
            if node1.spikes ≠ node2.spikes then
                splitGroups(node1, node2)         ▷ Put node1 and node2 in separate groups
            end if
        end for
    end for
end while
for group in groups do
    for node1 in group do
        for node2 in group do
            if checkEquiv(node1, node2) then      ▷ Use the SMT solver to check for equivalence
                splitGroups(node1, node2)         ▷ Put node1 and node2 in separate groups
            end if
        end for
    end for
end for
newNetwork ← N
for group in groups do
    if |group| ≠ 1 then
        newNeuron = mergeNeurons(group)                              ▷ Merge all neurons
        newNetwork = removeNeurons(newNetwork, group)   ▷ Remove neurons in group from
the network
        newNetwork = addNeuron(newNetwork, newNeuron)
    end if
end for
return newNetwork
```

These input properties can be encoded using LRA formulas as they are well defined subsets of the input space. Some of the common input properties and their respective encodings are as below.

- **All inputs** – These properties exhaustively search over all possible inputs. These are most commonly used for hardware property testing where, independent of the operation, the system should satisfy the hardware safety properties. The encoding for these properties is,

$$P(x_0) \triangleq \top$$

  Here $\top$ refers to a tautology, i.e. a statement that is always true.

- **Specific Inputs** – These properties focus on a specific input from the entire input space. These are commonly used for properties based on exceptions and corner cases. The encoding of such properties are assignments of the Boolean variables corresponding to the the timesteps. If a spike occurs at time t for neuron $v_{i,0}$ then the indicator variable $x_{(i,0,t)}$ would be assigned a 1. This can be shown using the formulation for a spike train I,

$$I' = \begin{cases} 1 & t \in I_i \\ 0 & otherwise \end{cases} \quad t \in \{1, 2, 3, \cdots, T/\delta\}$$

  Consider the two input spike trains,

$$I_1 = \{1, 3\}$$
$$I_2 = \{1, 2\}$$

  These spike trains can be interpreted as follows, at timestep $t = 1$ both inputs I_1 and I_2 spike. Similarly at $t = 2$, I_2 will spike again and at $t = 3$, I_1 will spike. We can encode the duration of $T = 3$ it into discrete steps of $\delta = 1$ and get the binary encoding of spike trains as,

$$I'_1 = \{1, 0, 1\}$$
$$I'_2 = \{1, 1, 0\}$$

  The LRA for this assignment would be of the form,

$$\begin{aligned} P(\{I_1, I_2\}) \triangleq & (x_{(0,0,1)} = 1) \wedge (x_{(0,0,2)} = 0) \wedge (x_{(0,0,3)} = 1) \\ & \wedge (x_{(1,0,1)} = 1) \wedge (x_{(1,0,2)} = 1) \wedge (x_{(1,0,3)} = 0) \end{aligned}$$

- **Inputs based on Spike counts** – These properties define a condition on the number of spikes in the input spike train. These can be lower or upper bounds on the number of spikes in the spike train. These can be generically written in the form,

$$P(x_0) \triangleq c_1 \leq \sum_{t \in [T/\delta]} x_{(i,0,t)} \leq c_2$$

  Here c_1, c_2 are non-negative integers.

- **Compound Properties** – these properties are conjunctions of multiple input properties. These are used to verify various problem specific properties. A generic formulation of these properties would be,

$$P(x_0) \triangleq P_1(x_0) \wedge P_2(x_0) \wedge \cdots \wedge P_k(x_0)$$

- $N(x_0) = y$ – network encoding
  The network's behaviour is encapsulated using the proposed SMT encoding. Since the encoding is sound and complete, any property of the original system is also a property of the encoding. This way when we use the SMT solver to check for a property we can guarantee that the properties satisfied by the encoding are also satisfied by the original system.

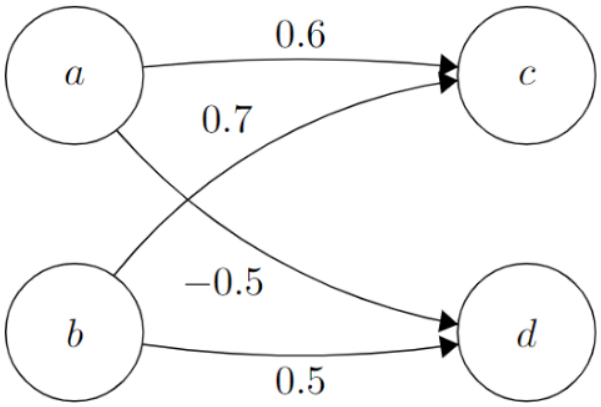


Figure 3.4: Example SNN (N) to demonstrate verification framework

- $Q(y)$ – output property
  The output property of the verification framework is similar to that of the input property. Several different types of properties can be expressed using LRA formulas. For output properties, most cases have count based properties since labelling and classification are all count based results for SNNs. The formulas we consider for these are often negations of required properties. This is due to the fact that, checking for existence of a counter example is much easier that exhaustive searching over all inputs. The following statements are equivalent to each other as they are contra-positives of one another.

$$\forall x_0,\ Q(y) \Leftrightarrow \neg(\exists x_0,\ \overline{Q(y)})$$

The set of all these formulas can be given to an SMT solver for verification. If the solver returns UNSAT we can say that the SNN satisfies the property as there exists no input for which the negation of the output property holds true. If the solver returns SAT, we can say that the SNN does not satisfy the property. Most solvers return a solution if the output is SAT. This solution is the counter-example for which the network failed the property.

The proposed framework is a reliable and scalable framework for verification of SNNs and is only limited by the expressibility of properties using LRA. This framework along with an efficient SMT solver can greatly increase the reliability of SNNs for safety critical systems. The robustness of the framework also unlocks the potential to verify more complicated properties and networks.

Example 3.7 *For the verification framework let us consider the example neural network in Figure 3.4. Let us consider the following two properties,*

- *P : the number of spikes generated by a is more than the number the spikes generated by b and Q : neuron d never spikes. This can be encoded as,*

$$P\ :\ \sum x_a \geq \sum x_b \text{ and } Q\ :\ \sum x_d = 0$$

where x_i are spike indicators for neuron i. We take the complement of the output property as, $\overline{Q}$: neuron d spikes at least once. If we give the network encoding N along with P and Q to the solver, it will return SAT along with the input spike train as, $I_a = [0,0,1,1,1]$ and $I_b = [1,1,0,0,0]$ for which the neuron d does spike. So the network N does not satisfy $\langle P, Q \rangle$.

- *P : neuron b never spikes and Q : neuron d never spikes. We take the complement of the output property as, $\overline{Q}$: neuron d spikes at least once. If we give the network encoding N along with P and Q to the solver, it will return UNSAT. So the network N satisfies $\langle P, Q \rangle$.*

So the network N satisfies the second property but it does not satisfy the first property. ■

Algorithm 3 Algorithm for Verification of SNNs

Input: The Spiking Neural Network N, Input property P and output property Q
Output: $UNSAT$ or Counter-example

```
Q' ← ¬Q
if Solver(N, P, Q') == SAT then
    counter = getSolution(N, P, Q')
    return counter
else if Solver(N, P, Q) == UNSAT then
    return UNSAT
end if
```

3.3.1 Hardness of SNN verification

In [16], it has been show using reductions that the verification problem of traditional neural networks is NP Hard. This is shown by reducing an instance of of the 3-SAT problem [16] to an instance of binary neural network verification. We use a similar process to prove hardness of SNN verification.

Theorem 3.3 *SNN verification is NP-Complete.* □

Proof 3.2 *Let us consider a general instance of the 3-SAT problem,*

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

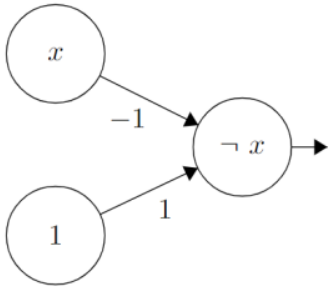
where each clause is of the form,

$$C_i \ : \ q_i^1 \vee q_i^2 \vee q_i^3$$

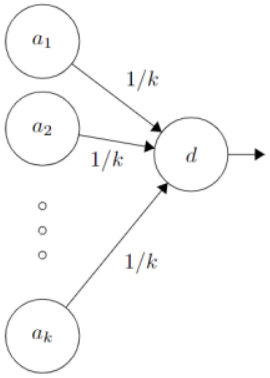
where each q_i is a Boolean variable or the negation of a Boolean variable. We will construct a verification problem of SNN that is satisfiable iff the 3-SAT formula is satisfiable. The constructed SNN verification problem will be done on a SNN which operates for one timestep and has the neuron thresholds as 1. Let us consider the following operations and their respective SNN interpretations for the Boolean operators,

- *Negation : $\neg x$.*
  We model this property using the SNN in Figure 3.5a. When x is 1 the potential of the output node is 0, and there is no output spike. If x is 0 the potential crosses the threshold and hence generates an output spike.

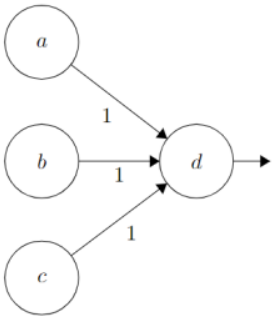
- *Conjunction : $C_1 \wedge C_2 \wedge \cdots \wedge C_k$.*
  We model this property using the SNN in Figure 3.5b. When all inputs are 1 the potential of the output node is 1 ($k \times 1/k$), so there is an output spike. If any of the inputs are 0 the potential does not cross the threshold and hence no output spike is generated.



(a) SNN encoding for negation of Boolean variable



(b) SNN encoding for conjunction



(c) SNN encoding for disjunction

Figure 3.5: SNN equivalents for logical operations

- *Disjunction : $a \vee b \vee c$.*
  We model this property using the SNN in Figure 3.5c. When a and b and c are 0 the potential of the output node is 0, so there is no output spike. If any one of them is 1 the potential crosses the threshold, so an output spike is generated.

For any instance of 3-SAT we can encode the formula using an SNN with the reductions mentioned above. The network is constructed in such a way that there will be only one output neuron (y). The reduction is polynomial in the number of clauses. Using this reduction we can now construct a SNN verification problem $\langle N, Q \rangle$, where N is the SNN constructed from the 3-SAT formula and Q is the output property "Does y spike?". If N satisfies the property Q we can say that the 3-SAT formula is also SAT. If the N does not satisfy the property Q we can say that the 3-SAT formula is UNSAT. Thus, we have a polytime reduction from 3-SAT to SNN verification, thereby allowing us to conclude on the hardness of the latter. □

3.3.2 Adversarial Robustness of SNNs using SMT encoding

We now talk about a special class of verification problems that is quite popular in recent literature, as defined in Chapter 2.

Definition 3.8 *An SNN N is said to be adversarially robust if $\forall x_1, x_2 \in X$ we have,*

$$\|x_1 - x_2\|_1 \leq \delta \implies N(x_1) = N(x_2)$$

where X is the input space given by $\{0,1\}^{m \times t}$, $\|a-b\|_1$ refers to the Manhattan norm, $N(x)$ corresponds to the output neuron with the most output spikes on input x and $\delta \in \mathbb{R}$.

In this section we take a look at a specific verification problem; the problem of adversarial robustness for SNNs. Adversarial robustness for traditional networks, as seen in Chapter 2, are defined as the ability of a network to give the same output for any input and every one of it's $\delta-$perturbations.

In the context of SNNs the same definition applies, where, for any given input x we ask if the SNN N gives the same output for x and all $\delta-$perturbations of x. Unlike traditional networks spiking neural networks cannot use the same notion of $\delta-$perturbation as the input of the SNN (i.e. spike trains) does not have real values to perturb.

We need to define a new notion of $\delta-$perturbation in the context of SNNs. Perturbations in traditional NNs are isomorphic by some bounded noise introduced to the inputs. We propose the following two notions of $\delta-$perturbations.

Definition 3.9 *$\delta-$perturbations on spike times refers to shifting the spike times of the spikes in the spike train by δ timesteps.*

For example, for δ and a spike $x_{(i,0,5)}$ the $\delta-$perturbation would mean that the spike can occur anytime between $5-\delta \leq t \leq 5+\delta$. In an abstract sense a $\delta-$perturbation of spike time is essentially shifting the time of the spikes of a spike train in any direction to have the spike events occur earlier or later than the original spike train. For example let us consider a spike train I given as,

$$I = [0,0,1,0,0,0,1,0,0,1,0]$$

Now let us consider a spike train I' given by,

$$I' = [1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0]$$

for $\delta = 2$ we can observe that each spike from I has been shifted by at most 2 timesteps, so we can say that I' is a $\delta-$perturbation (on spike time) of I.

Definition 3.10 *$\delta-$perturbations on spike counts refers to changing the spike counts of the spike trains so that there are at most δ changes.*

If $\delta = 10$ then, the sum of all the spikes that are present in the perturbed input but not in the original image plus the the sum of all spikes which are present in the original input but not in the perturbed input must be at most 10. The set of all δ-perturbed spike trains I_δ for a given spike train I can be formally written as,

$$I_\delta = \{I' \ : \ \|I' - I\|_1 \leq \delta\}$$

Where $\|X - Y\|_1$ refers to the first norm i.e. Manhattan distance between the two vectors.

For example let us consider a spike train I given as,

$$I = [0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0]$$

Now let us consider a spike train I' given by,

$$I' = [1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0]$$

for $\delta = 5$ we can observe that the Manhattan distance between I and I' is at most δ, so we can say that I' is a $\delta-$perturbation (on spike count) of I.

Comparing the original MNIST spike encoding (3.6b) and the $\delta-$perturbation of spike times (3.6c) we can observe that the variation is not drastic. There is some loss of spikes caused by spike times crossing the bounds of the simulation but, they are still very similar to each other. This similarity can also be intuitively explained. Since each neuron spike corresponds to a particular stimulus, delaying the stimulus will not affect the output drastically as long as the number of spikes is not greatly altered.

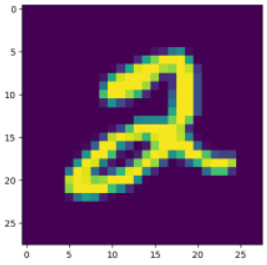
For most real life applications the spike trains generated by sensors are more susceptible to errors where, there are incorrect spikes or missing indented spikes. So we choose to use the notion of $\delta-$perturbation of spike counts for adversarial robustness.

3.3.3 Procedure

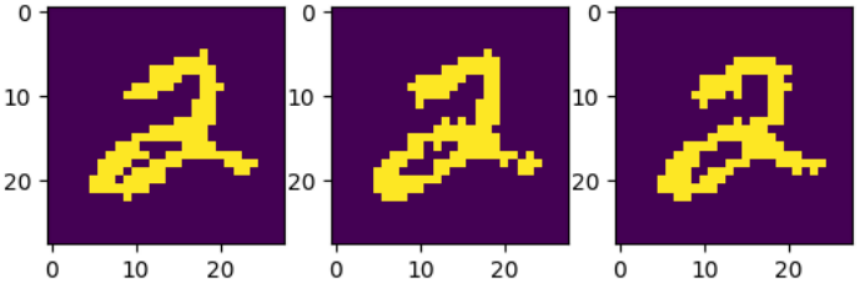
In order to check for adversarial robustness for SNNs we have the following procedure,

- Given an SNN N, we use the SMT encoding to get the set of formulas corresponding to the network
- We then create the set of formulas that cover the $\delta-$perturbations of a singe input x_0 as,

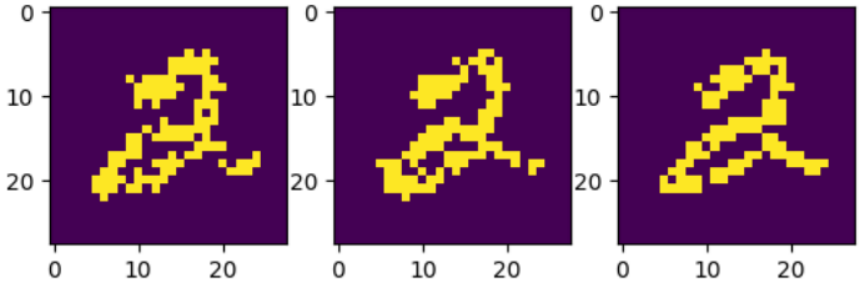
$$F_\delta \triangleq \|x - x_0\|_1 \leq \delta$$



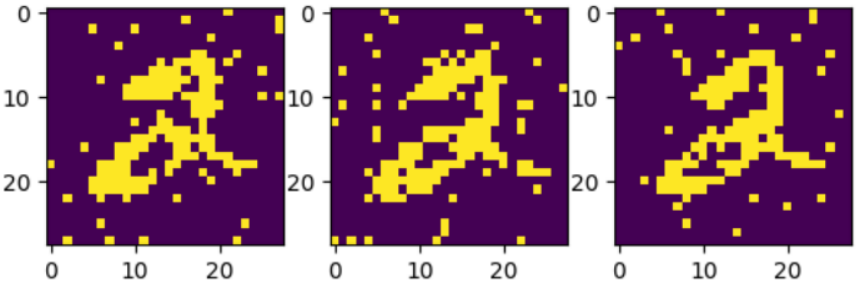
(a) A sample MNIST image



(b) Spike encoding of the image



(c) $\delta-$perturbation of spike times for $\delta = 1$



(d) $\delta-$perturbation of spike counts for $\delta = 50$

Figure 3.6: Different $\delta-$perturbations for a sample MNIST data generated using snnTorch

- Then we create the set of formulas which refer to the property that the original output ($y = max(y_1, y_2, \cdots, y_k)$) should match the output for all perturbations ($y' = max(y'_1, y'_2, \cdots, y'_k)$). The formula corresponding to the output of the network is given by,

$$F_y \triangleq \bigwedge_{i \in [k]} (\sum y_i \geq \sum y_1) \wedge \cdots \wedge (\sum y_1 \geq \sum y_k) \implies (y = i)$$

where $[y]$ refers to the set $\{1, 2, \cdots, k\}$. Informally the formula encapsulates the property that the output of the network should be the label corresponding to the output node which spiked the most.

Example 3.8 *In order to show this let us consider a network N with 3 output neurons and let us consider the output spike train generated by $N(x_0)$ for some input spike train x_0 are,*

$$y_1 = \{1, 1, 1, 1, 0, 1\}$$
$$y_2 = \{0, 0, 1, 0, 1, 0\}$$
$$y_2 = \{1, 0, 0, 1, 1, 1\}$$

The formulation would be,

$$\begin{aligned} F_y \triangleq & ((\sum y_1 \geq \sum y_2) \wedge (\sum y_1 \geq \sum y_3) \implies (y = 1)) \wedge \\ & ((\sum y_2 \geq \sum y_1) \wedge (\sum y_2 \geq \sum y_3) \implies (y = 2)) \wedge \\ & ((\sum y_3 \geq \sum y_2) \wedge (\sum y_3 \geq \sum y_1) \implies (y = 3)) \\ \triangleq & (y = 1) \end{aligned}$$

So for our example we will have $y = 1$. ■

The set of formulas corresponding to the matching outputs are,

$$F_{op} \triangleq F_y \wedge F_{y'} \wedge (y = y')$$

We take the negation of the intended property for the sake of the solver to get,

$$F'_{op} \triangleq F_y \wedge F_{y'} \wedge (y \neq y')$$

- We then combine all the formulas to get the set of formulas corresponding to adversarial robustness as,

$$F \triangleq F_\delta \wedge (N(x_0) = y_0) \wedge (N(x) = y) \wedge F'_{op}$$

We then pass these formulas to the SMT solver.

- If the SMT solver returns UNSAT, we can conclude that there exists no $\delta-$perturbation around x_0 for which the outputs of the two inputs do not match. This implies that for the given x_0 and δ, the network is adversarially robust.

- If the SMT solver returns SAT, we can conclude that there exists a $\delta-$perturbation of x_0 for which the outputs of the two inputs do not match. This implies that for the given x_0 and δ, the network is not adversarially robust. The assignment of x returned by the SMT solver is the $\delta-$perturbation of x_0 for which the outputs do not match.

Number of Nodes in SNN	Number of timesteps	Number of Variables for the solver	Time taken
10	25	~ 5000	0.12 sec
400	25	$\sim 2 \times 10^5$	0.79 sec
625	30	$\sim 3.75 \times 10^5$	0.86 sec
10000	30	$\sim 6 \times 10^6$	1.13 sec

Table 3.1: Relation between number of nodes and variables of the SMT encoding

3.4 Implementation and Results

We have written a program to encode a given SNN into its equivalent SMT encoding. We then construct random synthetic networks with varying scale and check the number of variables in the SMT encoding. The number of variables and time taken for encoding is given in Table 3.1

In order to test our method for SNN simplification, we create different SNNs with increasing number of nodes. We then apply the Simplification framework to these networks and analyse the resultant SNN. The results for the same are noted in the Table 3.2.

Nodes in the SNN	Inputs for simulation	Candidates after simulation	Simulation Time	Candidate neurons after query	Query Time	Percentage reduction
20	10	7	2.13 sec	6	2.45 sec	30%
20	20	6	2.34 sec	6	2.72 sec	30%
35	10	18	2.19 sec	13	4.52 sec	37.14%
35	20	15	2.39 sec	13	4.63 sec	37.14%
50	10	22	2.25 sec	17	6.89 sec	34%
50	20	17	2.41 sec	17	7.32 sec	34%

Table 3.2: Results of applying the simplification frameworks to networks with varying scales

In order to test the adversarial robustness framework we construct some networks and select some random inputs and note their respective outputs. Then for different values of δ we check for adversarial robustness.

Neurons of SNN	Number of inputs	Number of perturbations that are Robust			
		$\delta = 10$	$\delta = 20$	$\delta = 50$	$\delta = 100$
25	10	10	10	8	2
30	10	10	10	8	3
45	10	10	10	9	3

Table 3.3: Adverserial robustness for synthetic networks

In Table 3.3 the first column corresponds to the total number of neurons in the network. The second column describes the number of inputs for which adversarial robustness was checked. In the last column, the value for each δ denotes the number of perturbed inputs for which the the inputs were robust for that value of δ around the specific input under consideration.

Chapter 4

Timed Automata based encoding of SNNs

The temporal aspect of the spiking neural networks, makes it hard for the traditional mathematical tools to model the behaviours and their properties. In order to perfectly encapsulate the properties of a spiking neural network we need a mathematical object that can handle temporal behaviours. One of the most popular mathematical objects that is used to model time-based systems is Timed Automata.

In this chapter we first discuss the Timed Automata Based encoding of spiking neural networks and spike trains. We will then proceed to take a look at the verification framework which utilizes the timed automata encoding. We begin with a formal definition of a timed word, a Timed Automata and the notion of acceptance and rejection on timed words. Following that, we present the encoding of LIFR neurons using TAs and representation of a SNN as a collection of TAs encoding the constituent neurons and their interconnections.

4.1 Timed Automata

A timed word is a special type of word that has an associated positive non-decreasing sequence of numbers. These numbers refer to the time associated with each letter of the word. Formally we define a timed word as in Definition 4.1 below.

Definition 4.1 *A timed word w is a sequence,*

$$w = (x_1, t_1)(x_2, t_2), \cdots$$

where for all i, $x_i \in \Sigma$, $t_i \in \mathbb{R}^+$ and $t_i \leq t_{i+1}$.

Here Σ denotes an input alphabet. It can be a finite or an infinite sequence. A model that is often used in the context of timed words is a timed automata. A timed automata is formally defined as in 4.2 below.

Definition 4.2 *A **Timed Automata** TA is a tuple $(L, l^0, X, \Sigma, L_{acc}, Arcs, Inv)$, where,*

- *L is the set of locations (states) with l^0 as the initial location.*

- L_{acc} *is the set of accepting locations.*
- X *is a set of clocks.*
- Σ *is the set of communication labels (input symbols)*
- $Arcs \subseteq L \times (G \times \Sigma \times 2^{|X|}) \times L$ *is a set of transitions between locations.*
- $Inv: L \to G$ *assigns invariants to each locations.*

Here G is the set of guard conditions and $2^{|X|}$ refers to the reset conditions of all clocks. In other words, any subset of the clocks can be reset on an arc. The guard conditions are of the form,

$$g := c_1 \bowtie x \bowtie c_2$$

where $c_1, c_2 \in \mathbb{R}^+$, $x \in X$ and $\bowtie$ can be $\leq, \geq, =, \neq, <, >$. A timed automata can be used as an acceptance machine for timed words. If a timed word is passed to a timed automata the timed automata will either accept or reject the word. If a timed word, at the end of its transitions ends in one of the accepting states then we can say that the timed word is accepted, otherwise it is said to be rejected. If a timed word is accepted by a timed automata we say that the word is part of the language of the timed automata. If the word is rejected we say that it is not a part of the language of the timed automata.

A timed automata definition can be extended to include real variables and their update statements. This extended automata is often useful for mapping various real life applications. Most real work applications are made of smaller components that work together with each other. In order to model such systems it is not intuitive to model the whole system as a single automata. In these cases, it is better to model the individual modules as automata and then construct a composition of these automata. A composition of a timed automata is a collection of timed automata that function concurrently using a common global clock. In order for these disconnected automata to function with each other they need to communicate using some common means. This is achieved using common flags that are shared by the automata, called as communication labels or synchronization labels. On taking a transition an automata can either send a signal (denoted by the variable name followed by an !) or receive a signal (denoted by the variable name followed by an ?). We demonstrate the notion of composition of TAs with Example 4.1.

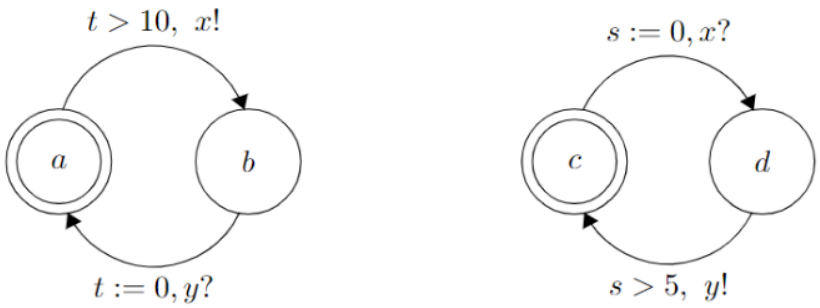


Figure 4.1: Example of composition of TA

Example 4.1 *The composition of automata in Figure 4.1 is a composition of two automata (with start states a and c respectively) that share a global clock and two communication labels x, y. If the first automata takes the transition $a \to b$ it will send a signal to the second automata using the communication label x. This means that when the transition $a \to b$ is taken, the transition $c \to d$ is also taken simultaneously. This way two independent automata achieve a notion of synchronization variables.* ■

4.2 TA encoding of SNN

In order to encode a spiking neural network as a timed automata we need to encode each neuron as a separate timed automata. We can then construct a larger timed automata using these smaller timed automata.

4.2.1 TA encoding of LIFR neuron

Since Leaky Integrate and Fire neurons constitute the most popular type of neurons used in SNNs, we consider it for the TA encoding. Formally we define LIFR neurons as done in Definition 4.3.

Definition 4.3 *An LIFR neuron $v \in V$ is defined as a tuple $(\theta_v, \tau_v, \lambda_v, p_v, y_v)$ where,*

- θ_v *is the firing threshold of* v.
- τ_v *is the refractory period of* v.
- $\lambda_v \in \mathbb{Q} \cap [0,1]$ *is the leak factor*.
- $p_v : \mathbb{N} \to \mathbb{Q}_0^+$ *is the membrane potential function defined as,*

$$p_v(t) = \begin{cases} \sum_i w_i x_i(t) & ; \text{if } p_v(t-1) \geq \theta_v \\ \sum_i w_i x_i(t) + \lambda_v \cdot p_v(t-1) & ; \text{otherwise} \end{cases}$$

- $y_v : \mathbb{N} \to \{0,1\}$ *is the neuron output function defined as,*

$$y_v = \begin{cases} 1 & p_v(t) \geq \theta_v \\ 0 & \text{otherwise} \end{cases}$$

Using this definition of LIFR neuron as base we define the TA encoding as follows.

Definition 4.4 *Given a neuron $v = (\theta, \tau, \lambda, p, y)$ we define the TA encoding of v as a tuple, $(L, A, X, Var, \Sigma, Arcs, Inv)$ where,*

- $L = \{A, W, D\}$
- $X = \{t\}$
- $Var = \{p, a\}$
- $\Sigma = \{x_i | i \in [1, \cdots, m]\} \cup \{y\}$
- $Arcs =$

$$\{(A, t \leq T, x_i?, \{a := a + w\}, A) | \forall i \in [1, \cdots, m]\} \cup$$
$$\{(A, t = T, \epsilon, \{p := a + \lfloor \lambda p \rfloor\}, D),$$
$$(D, p \leq \theta, \epsilon, \{a := 0\}, A),$$
$$(D, p \geq \theta, y!, \{\}, W),$$
$$(W, t = \tau, \epsilon, \{a, t, p := 0\}, A)\}$$

In this encoding of the LIFR neuron, the states of the automata encoding correspond to the Accumulation (A), Decision (D) and Waiting (W) states of the LIFR neuron. The variables p and a correspond to the total potential and accumulated potential of the LIFR neuron. Σ corresponds to the communication labels for receiving input spikes (x_i) and the communication label corresponding to output spike (y_i) of the neuron. The parameter T corresponds to the accumulation period of the neuron. Each neuron collects input spikes for the accumulation period and then upon completion checks if the neuron has enough potential to spike. The parameter τ corresponds to the refractory period of the neuron. For the arcs in the encoding we have the following informal descriptions.

- $A \rightarrow A$
  This transition corresponds to the accumulation of potential by receiving an input spike through the communication labels.
- $A \rightarrow D$
  This transition corresponds to the addition of the previous potentials to the accumulated potentials and goes to the state D, which will decide whether the neuron will spike or not.
- $D \rightarrow A$
  This transition corresponds to the reset of the accumulated potential and beginning of the accumulation period again after the neuron fails to cross the threshold to spike.
- $D \rightarrow W$
  This transition corresponds to the spiking of the neuron. This transition is taken if the total potential is greater than the threshold.
- $W \rightarrow A$
  This is the complete reset of the neuron to restart the accumulation period after the neuron has spiked.

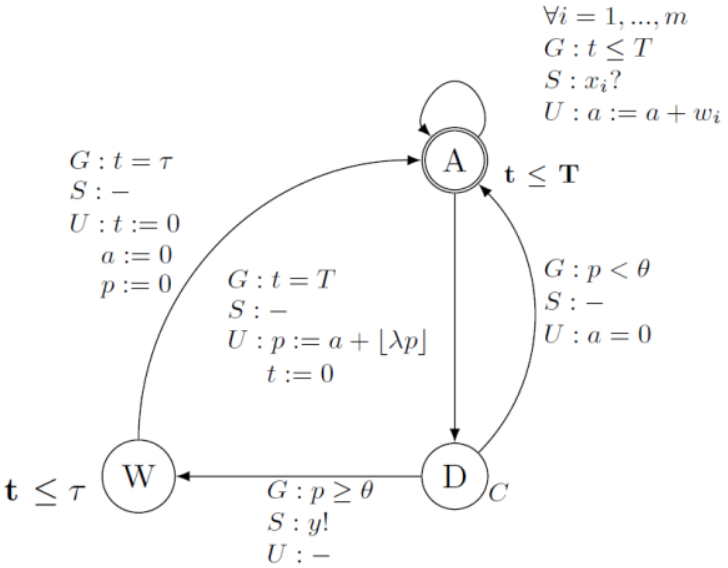


Figure 4.2: TA encoding of LIFR neuron [5]

The working of the TA given in Figure 4.2 can be explained as follows.

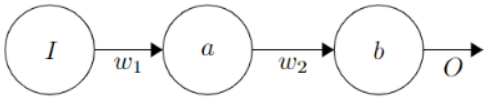


Figure 4.3: Sample SNN for TA conversion

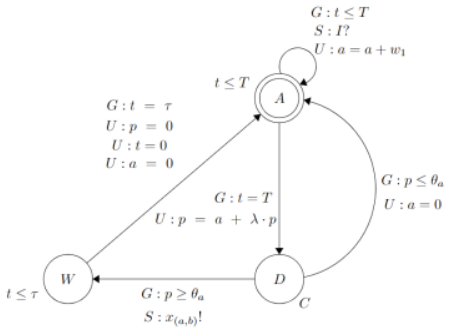
- The initial state of the automata is the state A. At this state the automata accumulates potential from the input neurons by taking the self transition corresponding to the input spike. If input neuron i spikes then the weight corresponding to x, w_i, is added to the accumulated potential of the neuron. $(a := a + w_i)$
- This process occurs until the accumulation period is completed $(t = T)$. At this point the transition from A to D is taken while adding the accumulated potential with the previous potential with some loss.
- D is a committed state. This means that if there is an output transition available, it must be taken. This is due to the decision process being instant. The clock cannot progress at this state and a transition must be taken immediately after arrival.
- If the total potential is less than the threshold $(p \geq \theta)$, the transition from D to A is taken while resting the accumulated potential. Here the next accumulation period will start.
- If the total potential is greater than or equal to the threshold, the transition from D to W is taken while sending an output spike $(y!)$.
- Then for τ duration the system will be idle at state W. Then at $t = \tau$ the transition from W to A is taken, while resetting all previous stored potentials and starting the next accumulation period.

An SNN can be considered as a collection of interconnected LIFR neurons. In a similar way, the TA encoding of an SNN can be considered as a composition of the TA encodings of the LIFR neurons. The synapses between two neurons is encoded by the communication label between the two TA corresponding to those neurons. Formally, if there exists a synapse from neuron v_i to neuron v_j, then, there will exist a corresponding communication label $x_{i,j}$ shared between them. The communication signal will be sent by the TA corresponding to v_i when, it takes the transition from its decision state to its waiting state. The communication signal will be received by the TA corresponding to v_j when, it takes the self transition from its accumulation state. We demonstrate this encoding with an Example 4.2

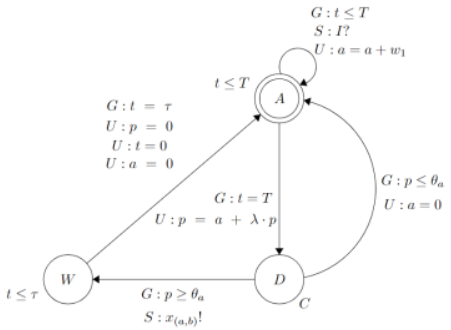
Example 4.2 *Let us consider a 2 state neural network given in Figure 4.3. The equivalent TA encodings of the two networks are given in Figure 4.4. When an input spike is given to the SNN, the TA corresponding to neuron a will take the self transition from state A to accumulate the potential. When the neuron a spikes it takes the transition from the state D to W, sending a communication signal through the label $x_{(a,b)}$ in the process. At this point the TA corresponding to the neuron b will take the self transition from state A. This way the synapse between the neurons a and b is encoded in the context of TA.* ■

Theorem 4.1 *The TA encoding of SNN is sound and complete.*□

Algorithm 4 refers to the algorithm to convert an SNN to its equivalent TA encoding.



(a) TA encoding of neuron a



(b) TA encoding of neuron b

Figure 4.4: TA encoding of the SNN in Figure 4.3

4.2.2 TA encoding of Spike Trains

The spike trains also need to be encoded in order to simulate the inputs along with the network. These spike trains can be formally described using the following grammar,

$$G ::= \Phi \cdot (\Phi)^\omega \mid P(d) \cdot \Phi \cdot (\Phi)^\omega$$
$$\Phi ::= s \cdot P(d) \cdot \Phi \mid \epsilon$$

where $P(d)$ denotes a pause of duration d, ϵ denotes empty alphabet (to indicate the end of a spike train) and $(a)^\omega$ corresponds to an infinite repetition of a. This grammar of spike trains works on continuous real values instead of discrete binary encodings, like the ones discussed in the previous chapters. Using this grammar we can construct a TA for the input spike train using the following recursive definition.

- If $I := \Phi_1 \cdot (\Phi_2)^\omega$
  - $L(I) = L(\Phi_1) \cup L(\Phi_2)$
    This means that the language accepted by the constructed TA I is the concatenation of the language accepted by the automata Φ_1 and Φ_2.
  - $first(I) = first(\Phi_1)$
    The initial state of I is the same as initial state of Φ_1.

Algorithm 4 Algorithm for getting the TA encoding of SNNs

Input: The Spiking Neural Network N with m layers with n_i nodes the $i^{th} layer$
Output: TA encoding of N

```
ta ← blankTA()                                   ▷ Create a TA with no states
for doi = 1 to i = n_1                           ▷ Adding all input neurons
    ta = composition(ta, encode(v_{i,1}))        ▷ Composition of the existing TA with the new TA
end for
for j = 2 to j = m do                            ▷ For each layer j
    for i = 1 to i = n_i do                      ▷ For each node i of layer j
        ta = composition(ta, encode(v_{i,j}))
        for k = 1 to k = n_{i-1} do              ▷ For each node i of layer j − 1
            ta.addCommLabels(i, k)               ▷ Connecting neuron v_{i,j} to the neuron v_{k,j-1}
        end for
    end for
end for
return ta
```

- $Arcs(I) = Arcs(\Phi_1) \cup Arcs(\Phi_2) \cup \{(last(\Phi_1), true, \epsilon, \emptyset, first(\Phi_2)), (last(\Phi_2), true, \epsilon, \emptyset, first(\Phi_2))\}$
  The arcs of I is the collection of all the arcs from Φ_1 and all the arcs from Φ_2 along with the arc between the last state of Φ_1 and the first state of Φ_2.
- $Inv(I) = Inv(\Phi_1) \cup Inv(\Phi_2)$
  The invariants of I is the collection of the all the invariants of Φ_1 and all the invariants of Φ_2.

This automata corresponds to the one in Figure 4.5a.

- If $I := P(d) \cdot \Phi_1 \cdot (\Phi_2)^\omega$
  - $L(I) = \{P_0\} \cup L(\Phi_1) \cup L(\Phi_2)$
    This means that the language accepted by the constructed TA I is the concatenation of a delay of d and the the language accepted by the automata Φ_1 concatenated with Φ_2.
  - $first(I) = P_0$
    The initial state of I is the state that encodes the delay.
  - $Arcs(I) = Arcs(\Phi_1) \cup Arcs(\Phi_2) \cup$
    $\{(P_0, t \leq d, \{t[0]\}, first(\Phi_1)), (last(\Phi_1), true, \epsilon, \emptyset, first(\Phi_2)), (last(\Phi_2), true, \epsilon, \emptyset, first(\Phi_2))\}$
    The arcs of I is the collection of all the arcs from Φ_1 and all the arcs from Φ_2 along with the arc between the last state of Φ_1 and the first state of Φ_2 and the arc between the delay state and first state of Φ_1.
  - $Inv(I) = \{P_0 \rightarrow t \leq d\} \cup Inv(\Phi_1) \cup Inv(\Phi_2)$
    The invariants of I is the collection of all the invariants of Φ_1 and all the invariants of Φ_2 along with the invariant of the delay state.

  This automata corresponds to the one in Figure 4.5b.

- If $\Phi := \epsilon$
  - $L(I) = \{\epsilon\}$
    The language accepted by the I is ϵ
  - $first(I) = E$
    The initial state is the only state.

- $Arcs(I) = \emptyset$
  No arcs.
- $Inv(I) = \emptyset$
  No invariants.

This automata corresponds to the one in Figure 4.5c.

- If $I := s \cdot P(d) \cdot \Phi'$
  - $L(I) = \{S, P_0\} \cdot L(\Phi')$
    The language accepted by I is the language formed with concatenating the timed word $\{S, P_0\}$ by the language accepted by Φ'
  - $first(I) = S$
    The initial state is S.
  - $Arcs(I) = Arcs(\Phi') \cup \{(S, true, b!, \emptyset, P), (P_0, t \leq d, \{t[0]\}, first(\Phi'))\}$
    The arcs of I are the collection of all the arcs in Φ' along with the arcs formed by concatenation.
  - $Inv(I) = \{P_0 \rightarrow t \leq d\} \cup Inv(\Phi')$
    The invariants of I is the set of all invariants of Φ' along with the invariant of the delay state.

  This automata corresponds to the one in Figure 4.5d.

For example, let us consider the spike train $\{1.3, 2.4, 2.9, 3.7\}$. Using the TA encoding for this spike train we get the TA in figure 4.6.

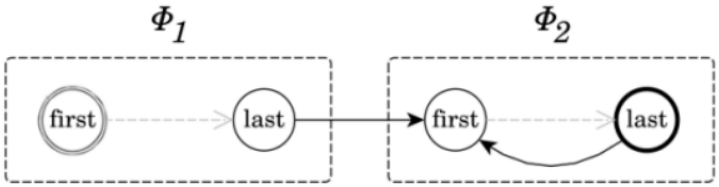
4.3 Verification of SNNs using TA encoding

Now that we have an encoding we can use it for verification of properties. We consider the definition 3.7 for verification of SNNs. In this definition we have an input tuple $\langle N, P, Q \rangle$ where N refers to the spiking neural network, P refers to the input property and Q refers to the output property.

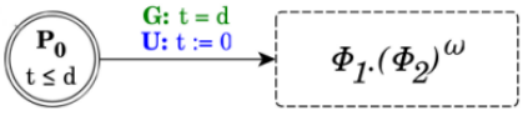
In order to utilize the encoding we construct the following,

- TA for P.
  We construct an automata corresponding to the property P such that every input accepted by the automata satisfies P. For example let us consider P :=minimum gap between two spikes is 1 second. This can be encoded as in the figure 4.7 The automata in Figure 4.7 has a single state and a single transition. The transition has a communication label associated which acts as the spike indicator. After the system takes the transition it must wait for at least 1 second before taking the transition again. So the set of all spike trains generated by the the automata satisfy the property P.

- TA for N.
  We encode the neural network using the TA encoding. We first encode each LIFR neuron in the automata using the TA encoding given in 4.4. We then encode the synapses by creating communication labels in between the appropriate neurons. The composition of all the neurons will together make up the TA encoding of the SNN.

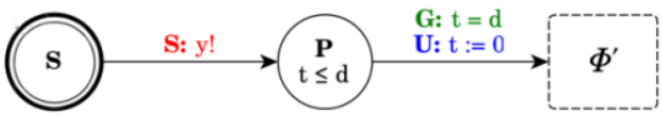


(a) $\Phi_1 \cdot (\Phi_2)^\omega$



(b) $P(d) \cdot \Phi_1 \cdot (\Phi_2)^\omega$

(c) ϵ



(d) $s \cdot P(d) \cdot \Phi'$

Figure 4.5: General TA corresponding to a spike train.

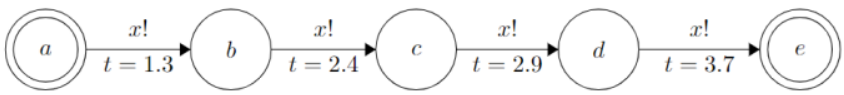


Figure 4.6: TA for the example

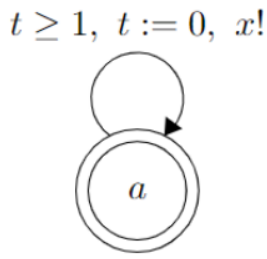


Figure 4.7: TA encoding of P

- TA for $\overline{Q}$.
  We write the property Q in the form of logic formulas that can be verified using the TA verification tools. These properties are usually written as reachability queries. For example let us consider Q :=the output neuron *op* never spikes. For this property we have a lemma,

  Lemma 4.1 *If a neuron never spikes, then the W state of the TA encoding of that neuron is not reachable.* □

  Using this lemma the property can be encoded as checking the reachability of the W state corresponding to the output neuron *op*. If *op* can spike then there exists an input for which the state W of *op* is reached.

We construct a composition of the TA associated with P and N. We use a TA verification tool to check the formula corresponding to the complement of Q on the composition of P and N. If the verifier returns a sample input trace, we can say that the property is not satisfied since, Q was false on the returned trace. If the simulator does not return a trace, we can say that the system satisfies the property.

4.4 Results

Number of Neurons	Time for TA construction	Time for verification
3	0.2 sec	0.6 sec
5	0.3 sec	1 sec
7	0.3 sec	2.4 sec
9	0.5 sec	6.3 sec

Table 4.1: Experiment results for the TA encoding of SNN

We construct some representative SNNs using Brian2 and encoded them as timed automata. We then use a reachability query and use a verifier to check if the encoded TA satisfies the reachability query. This query verification is done using the Uppaal tool. The implemented framework takes as input an SNN implemented using Brian2 libraries in Python. It then parses through the entire implementation to extract the required values and then constructs a file containing the timed automata encoding. This file is passes to Uppaal for verification. We also note the time taken for the construction of the file corresponding to the TA from an SNN and the verification time taken by Uppaal for verifying the query. These results are noted in the Table 4.1.

The verification for larger SNN networks and more complex output properties is left as future work.

Chapter 5

Conclusion and Future Work

In this thesis we have, focused on the spiking neural networks, their functions and its properties. Due to increasing number of use cases for spiking neural networks there is a proportional increase in the demand for modelling and verification frameworks for the same. A major hurdle for constructing these modelling and verification frameworks is the fact that traditional techniques cannot be applied directly in the context of SNNs. In this work we propose and highlight two frameworks based on two separate mathematical artifacts that can be used to model and analyse SNNs.

Initially, we describe the functionality of an SNN to get a better understanding of what functionalities have to be encapsulated by the encodings. We also discuss some additional frameworks that take advantage of the encodings in order to simplify and verify the SNNs. These frameworks utilize the soundness and completeness of the encodings to yield good results.

The first proposed encoding utilizes the LRA formulations to encode the SNN and encapsulate it's properties. The proposed encoding of SNNs can then be used of simulation, verification and simplification of SNNs. This encoding can also take advantage of various SMT solvers in order to create a scalable framework for property verification.

The second encoding uses timed automata in order to model the SNNs. This framework used for modeling SNNs can handle the temporal aspect of SNNs well and can achieve a very robust modelling for the same. The TA encoding can also take advantage of TA simulation and verification tools in order to simulate and verify SNNs.

With the increasing demand of power efficient systems for edge computing applications, we believe that the work done for the duration of this thesis serves as a good starting point for the research on encodings and verification of SNNs. For the future work, we plan to develop encodings for more complex SNN architectures like Convolutional SNNs and Recurrent SNNs. Additionally we also plan to apply the proposed frameworks to the various real world SNN networks. Finally we plan to propose additional frameworks analogous to the ones that exist in the context of traditional NNs like, Abstraction, Modification and many more.

Bibliography

[1] Albarghouthi, A. Introduction to neural network verification, 2021.

[2] Alur, R. Timed automata. In *Computer Aided Verification*. Springer Berlin Heidelberg, 1999, pp. 8–22.

[3] Ayuso-Martinez, A., Casanueva-Morato, D., Dominguez-Morales, J. P., Jiménez-Fernandez, A., and Jimenez-Moreno, G. Construction of a spike-based memory using neural-like logic gates based on spiking neural networks on spinnaker, 06 2022.

[4] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. Uppaal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control* (Berlin, Heidelberg, 1996), Springer-Verlag, p. 232–243.

[5] De Maria, E., Di Giusto, C., and Laversa, L. Spiking neural networks modelled as timed automata with parameter learning, 2018.

[6] De Maria, E., Di Giusto, C., and Laversa, L. Spiking neural networks modelled as timed automata with parameter learning, 2018.

[7] de Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008, pp. 337–340.

[8] Diehl, P. U., and Cook, M. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience 9* (Aug. 2015).

[9] Ding, J., Yu, Z., Tian, Y., and Huang, T. Optimal ann-snn conversion for fast and accurate inference in deep spiking neural networks, 2021.

[10] Elboher, Y. Y., Gottschlich, J., and Katz, G. An abstraction-based framework for neural network verification. In *Computer Aided Verification* (Cham, 2020), S. K. Lahiri and C. Wang, Eds., Springer International Publishing, pp. 43–65.

[11] Eshraghian, J. K., Ward, M., Neftci, E., Wang, X., Lenz, G., Dwivedi, G., Bennamoun, M., Jeong, D. S., and Lu, W. D. Training spiking neural networks using lessons from deep learning.

[12] Gokulanathan, S., Feldsher, A., Malca, A., Barrett, C., and Katz, G. Simplifying neural networks using formal verification. In *NASA Formal Methods* (Cham, 2020), R. Lee, S. Jha, A. Mavridou, and D. Giannakopoulou, Eds., Springer International Publishing, pp. 85–93.

[13] Goldberger, B., Katz, G., Adi, Y., and Keshet, J. Minimal modifications of deep neural networks using verification. In *LPAR23. LPAR-23: 23rd International Conference on Logic for*

Programming, Artificial Intelligence and Reasoning (2020), E. Albert and L. Kovacs, Eds., vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 260–278.

[14] Gurney, K. *An Introduction to Neural Networks*. Taylor Francis, Inc., USA, 1997.

[15] Jagtap, A. B., Sawat, D. D., Hegadi, R. S., and Hegadi, R. S. Verification of genuine and forged offline signatures using siamese neural network (SNN). *Multimedia Tools and Applications 79*, 47-48 (Apr. 2020), 35109–35123.

[16] Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design* (Jul 2021).

[17] Kim, T., Hu, S., Kim, J., Kwak, J. Y., Park, J., Lee, S., Kim, I., Park, J.-K., and Jeong, Y. Spiking neural network (SNN) with memristor synapses having non-linear weight update. *Frontiers in Computational Neuroscience 15* (Mar. 2021).

[18] Kuper, L., Katz, G., Gottschlich, J., Julian, K., Barrett, C., and Kochenderfer, M. Toward scalable verification for safety-critical deep networks.

[19] Lahav, O., and Katz, G. Pruning and slicing neural networks using formal verification, 2021.

[20] Stimberg, M., Brette, R., and Goodman, D. F. Brian 2, an intuitive and efficient neural simulator. *eLife 8* (aug 2019), e47314.

[21] Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., and Maida, A. S. Deep learning in spiking neural networks.

Modelling and Analysis of Spiking Neural Networks

ORIGINALITY REPORT

3%
SIMILARITY INDEX

PRIMARY SOURCES

1	library.isical.ac.in:8080 Internet	174 words — 1%
2	hal.archives-ouvertes.fr Internet	132 words — 1%
3	Elisabetta De Maria, Thibaud L'Yvonnet, Daniel Gaffé, Annie Ressouche, Franck Grammont. "Modelling and Formal Verification of Neuronal Archetypes Coupling", Proceedings of the 8th International Conference on Computational Systems-Biology and Bioinformatics - CSBio '17, 2017 Crossref	37 words — < 1%
4	Deng, Xiang Bo, Yue Xiang Lin, Ling Dong Bu, Li Zhou Zhang, and Zhe Liu. "Prediction Modeling of Maximum Dry Density of Coarse Grained Soil Using Improved Artificial Neural Networks", Applied Mechanics and Materials, 2013. Crossref	33 words — < 1%
5	amslaurea.unibo.it Internet	32 words — < 1%
6	"The effect of an exogenous magnetic field on neural coding in deep spiking neural networks", Journal of Integrative Neuroscience, 2018 Crossref	27 words — < 1%

7	Franck Cassez. "The Complexity of Codiagnosability for Discrete Event and Timed Systems", Lecture Notes in Computer Science, 2010 Crossref	26 words — < 1%
8	docplayer.net Internet	21 words — < 1%
9	"Algorithms and Computation", Springer Science and Business Media LLC, 2005 Crossref	18 words — < 1%
10	J. Green, S. Bhattacharyya, B. Panja. "Real-Time Logic Verification of a Wireless Sensor Network", 2009 WRI World Congress on Computer Science and Information Engineering, 2009 Crossref	18 words — < 1%
11	"Handbook of Natural Computing", Springer Science and Business Media LLC, 2012 Crossref	15 words — < 1%
12	Lecture Notes in Computer Science, 2012. Crossref	14 words — < 1%
13	www.grin.com Internet	14 words — < 1%

EXCLUDE QUOTES	ON	EXCLUDE SOURCES	< 14 WORDS
EXCLUDE BIBLIOGRAPHY	ON	EXCLUDE MATCHES	< 14 WORDS