Fakultät für Informatik
Technische Universität München

Master's Thesis in Informatics

Design and Analysis of a Novel Boolean Neuron Model

Sergey Nasonov

Fakultät für Informatik
Technische Universität München

Master's Thesis in Informatics

Design and Analysis of a Novel Boolean Neuron Model

Design und Analyse eines Neuartigen Boolschen Neuronenmodell

Author: Sergey Nasonov
Supervisor: Knoll, Alois Christian; Prof. Dr.-Ing. habil.
Advisor: Kohler, Matthias; M.Sc.
Submission Date: October 15, 2018

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, October 15, 2018

SERGEY NASONOV

Abstract

Neural Networks, which are the foundation of every human brain are very peculiar structures. Their functioning can produce amazing results. Learning about them and the way they work, however, is an uphill task.

To respond to this challenge, several models that represent the functionality of the neural networks were created, for instance, [6] and [14] . These models are successfully used in niche applications. However, performance of these models is lower than of a human brain in comparable tasks. Besides that, energy consumption is larger by several factors.

As a part of the Thesis a Novel Neuron Model was developed. It is motivated by the goal of extension of the current use of artificial neural networks that represent the ideas of how biological networks work. A model that works precisely and fast can be used in technical solutions that serve people. The Novel Neuron Model is based on principles of the neuron model described in [11] and [18]. It is extended with data structures as well as processes that optimize functioning of the network.

The Novel Model has dendritic trees in place which are prototypes of a biological version of these trees. In addition, it has a centralized control for boolean values in that tree and employs Boolean functions. In order to analyze the model, it was implemented as a software simulator of Neural activity. It is written in C++ which leads to swift execution. Moreover, it has a great potential to yet improve the performance by being ported and executed on a special programmable chip, such as a field-programmable gate array (FPGA).

Acknowledgements

I am grateful for the support and ideas to Dr. phil. Gabriele Scheler, Principal Research Scientist at Carl Correns Foundation for Mathematical Biology and Dr. rer. nat. habil. Johann Schumann who is a member of the Robust Software Engineering Group RSE at NASA Ames.

A special gratitude goes to M.Sc. Matthias Kohler for the ideas, guidance and meetings on a regular basis that allowed to have productive discussions, to structure the work well, and to follow through with the plan.

Contents

1 Introduction

Artificial neural networks (ANNs) are computing systems that reflect ideas behind biological neural networks that compose brains. The main goal for the use of ANNs is to allow electronic systems to control and handle objects, recognize patterns or sequences (gesture, speech) in a similar way as a human brain does. Efficiently solving these problems is especially important for the very promising domain of robotics [5].

ANN operate in two stages. First - "Learning", second - "Application". During the first phase ANN goes through examples and infers patterns. The result of this is a *model* that recaps the main features of the patterns. As an illustration, in image recognition, the network might learn to identify images that contain buildings. By analyzing images that have been marked as "a building"the network learns patterns such as shapes of windows and roofs. It then compares these patterns with the details of objects on images that were marked as "not a building"and adjusts the model accordingly. In the second phase the network would use the model to identify buildings in previously "unseen"images. ANN have been used to successfully solve many other tasks, such as flying an aircraft and detecting a credit card fraud [4].

There is one issue in applying ANN. It is that the training of the network to efficiently perform the tasks takes very substantial energy and memory resources [10].

The first reason is that most computational resources are based on *von Neumann* architecture. This requires storing and processing billions of bytes of information about the neurons and their connections along with billions of bytes more of auxiliary information.

The second reason is that many existing models and software are using a concept of Spiking Neural Networks (SNNs) suggested by Huxley and Hodgkin [13]. For instance, Izhikevich [14], FitzHugh-Nagumo [16] models, as well as NEURON and NEST simulators. Performance of every model can be seen on Fig. 2. in the paper by Izhikevich [15]. Their popularity is based on the fact that SNNs are close to biological analogs and utilize the notion of time. For instance, a neuron "fires"only when its electrical potential reaches a threshold value. The potential also decays over time. To model spikes in such networks, probabilities and differential equations to model neurons and synapses are used. Unfortunately, this requires a lot of energy and time to calculate.

Thirdly, it is often the case that with scale that is required to achieve meaningful results comes overhead. Namely, large SNNs require to transmit necessary data to other machines which significantly degrades the performance.

Notably, the inefficiency of the existing approaches can be clearly seen when huge data centers that try to model a human brain are compared to the biological alternative which is relatively small. Ultimately, the use of large-scale neural networks has been limited because the costs of using the networks have not been justified by proportionate advantages.

The Novel Neuron Model combines a unique idea of application of Boolean functions together with prototypes of dendritic trees. This brings a high precision of the modelling.

The reason is that Boolean functions are very versatile, and most known properties of dendritic trees can be encoded into them. In fact, the Boolean functions that were used as activation functions outlined in [11] and [18] produced realistic results. The simulator that is built around the new model also replicates spiking behaviour, cognate to the one in the SNNs. Every neuron "fires"*iff* its input contains at least a certain amount of *true* values.

Moreover, as it is described in the chapters that follow, the model can be used to deliver advanced processing performance.

2 Fundamentals

The software operates in several modes. In the mode for generating new data, a number of instances of *class Neuron* are created by a generation function that operates in that mode. Alternatively, a reading of previously created data is attempted. Next, the control proceeds to a time loop. The number of iterations of this loop is capped by the parameter MAX_TIME. Inside of this loop every neuron is sequentially processed. This processing includes:

- calculating the number of active synapses.
- executing a Boolean function which checks if a defined threshold for "firing"of the neuron is reached.
- in case the neuron "fires ", it sends a signal to the neighboring neurons. Specifically, it sets the values of synapses to which the current neuron is connected to the value of 1 and updates the time, starting which the synapse value is valid, as well as the time until which this value is valid. This is needed for calculations of active synapses. The end time and starting time are different by value of the parameter TIME_CONST.
- Writing the simulation results to the file "NeuronStatistics.csv ".

3 Related Work

3.1 TensorFlow Framework

TensorFlow is a software framework for machine learning. Its infrastructure and algorithms can be applied to understand speech, recognize objects using computer vision or infer patterns in financial data that can help to create a trading strategy.

The underlying core of every computation is a graph. Data that flows along edges is organized into arrays and is called tensors [6]. The framework typically uses *sigmoid* activation function for calculating the output of every neuron. This function has a recognizable "S"-shaped curve. It always produces values by computing the input in some specific way, usually with the use of the logistic function. This is in contrast to the case of spiking neural network, in which every neuron produces an output only if the incoming signal is greater than some threshold value.

Overall, this software is fast, however, its structures that represent neurons do not allow for tweaking in a thorough way.

3.2 NEURON simulation environment

NEURON is an environment for simulation of spiking neural networks. It is often used to solve problems that are based on experimental data, for instance, anatomical and biophysical properties [9]. Although it is possible to create very comprehensive neuron structures, it is practically slow in compare to the existing alternatives.

3.3 NEST simulation environment

NEST is a software for large-scale simulation. Its ultimate goal is to model the dynamics of spiking neural networks quickly. Additionally, its flexibility is increased through allowing different neuron and synapse models to work in the same network. This can increase accuracy of the modelling. Finally, the connectivity between the neurons can be implemented by an adjacency list that speeds up the performance [8].

It can be used to model visual and auditory parts of a brain [17, 7]. Overall, this simulator is very fast and has a variety of possibilities to adjust structures of neurons. However, it lacks as thorough precision to biological properties as the NEURON simulation environment.

4 Solution Approach

4.1 Description of a class Neuron.

The foundation of the software is the **class Neuron**.

Every instance of the class represents neuron as close to the biological concepts as necessary for the purpose of the research. For that reason, there are attributes and types that are defined as structures as well as functions that maintain every class instance. Specifically, the structure of a neuron is as follows. Every instance has a list of branches, some branches have child branches. Branches can have synapses. In the Bridge mode, which is discussed later, branches that do not have a synapse are either a root branch or host branches (which exist only in the Bridge mode). The attributes of the class are the following:

```
1 nid // Serves as a unique identification number for every neuron instance.
2 nNumBranches // Total number of branches that exist across the entire Neuron instance.
3 nNumActiveSignals // Total number of Synapses in the neuron instance that have value 1 (which means they are
      "active").
4 nOutDegree // Number of outgoing edges from the given neuron instance to other instances.
5 nNumSyns // Total number of Synapses in the neuron instance.
6 nNumBrInBush // Number of Branches in a Bush. A Bush is a set of branches that correspond to one expression,
      that consists of variables. Valid only in the Bridge mode.
```

Note. The attributes and functions that exist in the source code and not mentioned here are there to allow to extend the class more easily in the future or not mentioned because of their utility purpose.

Note. For example, equation $\bar{a}$, b, c corresponds to three synapses (that for simplicity of implementation are actually represented by branches in a Bush).

4.2 Data types that support the functionality of the class.

The software creates a dendritic tree using branches. Every branch has a defined number of synapses from 0 up to the parameter MAX_NUM_SYNAPSES or has a fixed value depending on the generation function (working mode). Also, every branch has a boolean type. The three possible boolean types for a branch are:

- AND which is encoded by value 0.
- OR by 1.
- NOT by 2.

During the execution of the Boolean function, depending the type of the branch, boolean values of all child branches are processed with all synapses sitting this branch. The processing performs as follows. On every boolean value operation of either conjunction or disjunction with the values processed earlier is applied. In case the branch type is NOT,

the final value of all previous operations is negated. When all child branches and all synapses of the current branch are processed, the current branch returns its value to the parent branch. The parent branch repeats the same procedure with all its own synapses and values returned by all child branches.

Additionally, every branch has a list (a vector) of IDs that correspond to the unique ID of a branch in every neuron, as well as a list (a vector) of unique synapse IDs that belong to the branch. Overall, the Branch structure has the following format:

```
typedef struct branch
{
    int numSyn; // nubmer of synapses on the branch itself.
    int boolType; // 0 − AND; 1 − OR; 2 − NOT.
    int numChildBr; // number of child branches

    vector<int> vChildBr; //IDs of child branches.
    vector<synapse> vsyns; //vector that stores values of synapses on the branch
} branch;
```

The following vector hosts all branches of the neuron instance:

```
vector<Neuron::branch> nvBran;
```

Every synapse sitting on a branch has an ID, as well as starting and ending times. If current time of the time loop is within values of these attributes, and if if it has a value 1, only then the synapse if considered to be "active". In other words, every synapse lives in time.

```
typedef struct synapse
{
    int synid; // id of a synapse;
    int fromTimeStamp; // to make sure signal is not available for calculation before the right time.
    int toTimeStamp; // signal lasts up to a defined time, but can be increased.

    bool sval; // value of the synapse: 0/1.

} synapse;
```

Every neuron has a list (a vector) of neurons with which it has a connection (outgoing edge, similar to an axon). This list contains a set of IDs, defined below. The purpose is to access all synapses of a neighbouring neuron to update the values in constant time.

```
typedef struct neigb
{
    int nid; //id of a neighboring neuron
    int bid; //id of a neighboring branch in that neuron
    int sid; //id of a synapse on that branch of that neuron.
} neigb;
```

Information about neurons to which a given neuron is connected through its Axon is stored in the vector:

```
vector<neigb> nvNeigb;
```

A special type that is necessary to operate in the Bridge mode (only) has the following structure:

```
typedef struct bridge
{
    bool synval; // value of the synapse: 0/1
    int fromTimeStamp; // to make sure signal is not available for calculation before the right time.
```

```
5     int toTimeStamp; // signal lasts up to a defined time, but can be increased.
6
7 } bridge;
```

Vector containing the bridge values:

```
vector<bridge> nvBridge;
```

Additionally, there exists a vector that stores the masks for synapse values:

```
vector<string> nvBrMasks;
```

Masks of variables are discussed in the subsequent chapters.

4.3 Operational modes of the software.

4.3.1 Read Configuration File.

Reads a file with parameters which was generated by one of the modes below (except mode 5, which has its own reading function).

4.3.2 Generate Random Data.

Generates a random number of branches and synapses up the MAX_NUM_BRANCHES and MAX_NUM_SYNAPSES parameters for every neuron instance respectively. Randomly defines the type of every branch (AND, OR, NOT). Assigns random number of child branches to each branch as well as values to the synapses (0/1). Specifically, every branch is assigned the child branches in the following way:

1. For the current branch generate a number that denotes how many children the branch will have.
2. Starting with the second branch (with ID = 1), assign sequentially these child branches to the current branch (in the very first iteration branch ID = 0).
3. Once the branches have been assigned, the next branch with ID = 1 (and later 2, 3, and so on) can have as many children as there are left unassigned from the previous draw.

After that, synapses are created for the branch. Next, everything is put into the vector and the processing of the next branch starts. Finally, when all neurons are processed, the function

```
ConnectNeuronsRand();
```

described below is executed. Essentially, it connects neurons between each other. It should be noted that during the execution of the generating function every parameter and every variable necessary to recreate the network is output to the file "NeuronOutDataRand.txt ".

4.3.3 Generate Layered Network.

Every Neuron has the same fixed structure: one root branch without synapses that hosts two branches that have two synapses each. Every neuron in every layer is connected to

every other neuron in the following layer. The parameter NUM_NEURONS_PER_LAYER defines the number of neurons in each layer (number is the same for every layer). The number of layers is defined by the parameter NUM_LAYERS. The approximate structure of this neural network can be seen on the picture 4.1 below. The function outputs every variable necessary to recreate the network to the file "NeuronOutDataLayered.txt ".

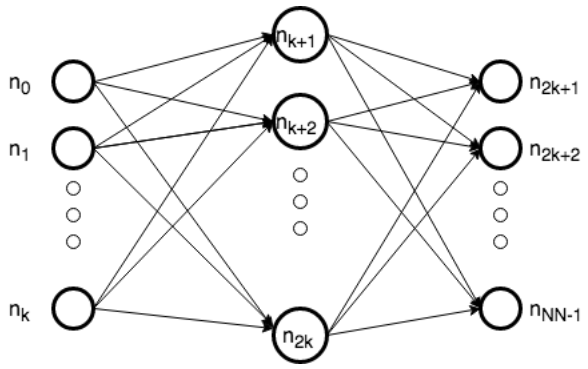

Figure 4.1: Example of a Layered Network.

4.3.4 Generate Circled Network.

Every neuron instance has the same fixed structure. It contains one branch and one synapse on that branch. The neurons are connected in a circle (thus, forming a one-directional graph). The approximate structure of this neural network can be seen on the picture 4.2 below. The function outputs every parameter and every variable necessary to recreate the network to the file "NeuronOutDataCircle.txt ".

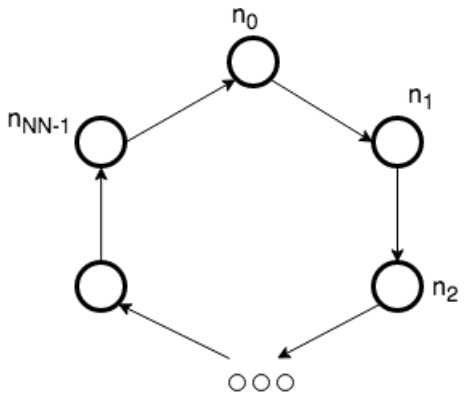

Figure 4.2: Example of a Circled Network.

4.3.5 Generate Bridged Network.

The function operates by creating neurons that have a specific branch structure and the types. The function outputs every variable necessary to recreate the network to the file "NeuronOutDataBridge.txt ". In this network there are three types of branches in each neuron. First type is a root branch, which has an ID = 0. Second type is a hosting branch. Every hosting branch is connected has the root branch as its parent. There can be several

branches of this type in each neuron with IDs, starting with 1. Also, this branch hosts branches of the third type forming a Bush of branches. The last type is a synaptic branch. These branches have synapses (only one per each branch). The first branch of this type gets an ID starting with the ID of the last hosting branch + 1. All following synaptic branches have an ID increased by 1 from the preceding branch.

The number of the hosting branches and the synaptic branches is defined as follows. Before the run-time of the program, a user defines parameters NUM_BRIDGE_SYNAPSES and NUM_BRIDGE_THRES. The first parameter is the number of variables that exists. For example, the expression a $\wedge$ b $\wedge$ c has three variables which correspond to three synapses (that for simplicity of the implementation are actually represented by the synaptic branches with a synapse). The second parameter defines the least amount of the variables that must have value 1 for the expression to output value 1. For instance, suppose the goal is to find all expressions that can be created out of the original expression a $\wedge$ b $\wedge$ c and that can be evaluated to 1, no matter the order of the boolean values that are supplied to the variables, while at least NUM_BRIDGE_THRES of these values are 1. Out of the 8 expressions that can be created, only 4 will result in the value 1, when the expressions are evaluated. That means that there will be 4 hosting branches in each neuron. Every expression out of these 4 contain masks for the variables. Specifically, some variable values will be used in the calculations of the expression as they are supplied, while others will be negated. The resulting expressions with the masks are shown in the table 4.1:

1	$\bar{a}$	b	c
2	a	$\bar{b}$	c
3	a	b	$\bar{c}$
4	a	b	c

Table 4.1: All possible masks that satisfy the condition.

Another example would be the case, when the threshold was set to 3. Then the only equation (mask) would be produced. It is in the last row of the same table. The boolean expressions that are represented by synapses on branches will evaluate to 1 at least once in the Neuron when all the equations are evaluated and disjunct ($\vee$).

On the picture 4.3 there is shown a Bridge network consisting of three neurons. In this network each neuron has a *root* branch with ID equal 0. Then, there are four *hosting* branches with IDs equaling sequentially from 1 to 4. Finally, there are *synaptic* branches on the bottom layer of every neuron. Every synaptic branch has a synapse that is depicted as a box with a variable.

As it can be seen on the picture, the masks are assigned to the branches within each hosting branch. Every branch within each bush (3 synaptic branches) are using one of the masks. Next bush of branches is using the next mask and this proceeds until all the masks are used. In the table 4.1, the first line contains masks $\bar{a}$, b, c so the function assigned the branch with ID = 5 a type NOT, branches 6 and 7 become type AND (they could also get assigned type OR, and that would make no difference, because there is only one synapse on the branch). Then, the next mask is taken and the value of AND is assigned to the branch with ID = 8, NOT to the branch with ID = 9 (because it is represented by value $\bar{b}$ and the branch 10 is assigned the value AND. This proceeds until all branches receive their types. It is important to note, that on the picture, values a, b, c represent not the actual values stored in that blocks, but the type of the synaptic branch that contains the

synapse. The values stored in these blocks will be values that were forwarded from the corresponding values from the bridges. The description of how forwarding works is described below.

In general, the algorithm can be described in the following way. In the beginning, masks for the branches are generated by the function *BranchMaskGeneration()*. The result is similar to what is shown in the table 4.1. These masks will be used to denote if some branch has a *negation* type, which means value of synapse during executions of a Boolean function on that branch gets inverted.

Following that, the function creates child synaptic branches for each hosting branch. Their number is equal to the parameter NUM_BRIDGE_SYNAPSES which defines how many variables (a, b, c, etc.) participate in the equations. The function also writes the number of host branches that is the size of the vector *brMasks* that stores all masks and number of synaptic branches NUM_BRIDGE_SYNAPSES to the output data file.

Generation of synapses follows, and every synapse gets current value *false* and its time variables receive values of 0. After that, a Bridge is created. The purpose of the Bridge is to manage synapse values in a coordinated and consistent way by forwarding them to corresponding synapses by the function

```
signalToOutBridgesAndBranches();
```

For instance, when there are 3 variables (a, b, c), then the bridge has the size 3. The first value represents variable a, the second b and so on. The values on the bridges will be forwarded to all corresponding synapses in the entire neuron. It can be seen on the picture 4.1. A small table which is a Bridge under each neuron contains three variables. The first variable has an arrow pointing to the branches that represent the same variable a. The same happens with b and c. Additionally, on the same picture it can be seen, that the neurons with the IDs = 0 and 1 "fired"and changed the values corresponding to variables a and b in the Bridge of neuron with ID = 2 to the value 1 while leaving the cell for the variable c untouched. Overall, the tables below sum up the representation of branches and synapses.

Branch ID	Type	Symbol
0	AND	$\wedge$
1	OR	$\vee$
2	NOT	$\bar{x}$

Synapse (variable)	Type	Symbol
1	AND	x
0	NOT	$\bar{x}$

4.3.6 Read the Bridge Network Data

Conceptually, it is not different from the regular reading mode. It is extended, however, with reading of a number of Host branches *numHostBr* and number of Synaptic branches *numSynVars* from the file "NeuronOutDataBridge.txt "and setting their values to every instance of neuron. Furthermore, in this mode the function fills the Bridge of every neuron (in the same way as in the generating function).

4.4 Functions that maintain the class and perform the work-flow.

1. `BranchMaskGeneration()`

   Generates a list (a vector) of masks for boolean types of the synaptic branches. It is

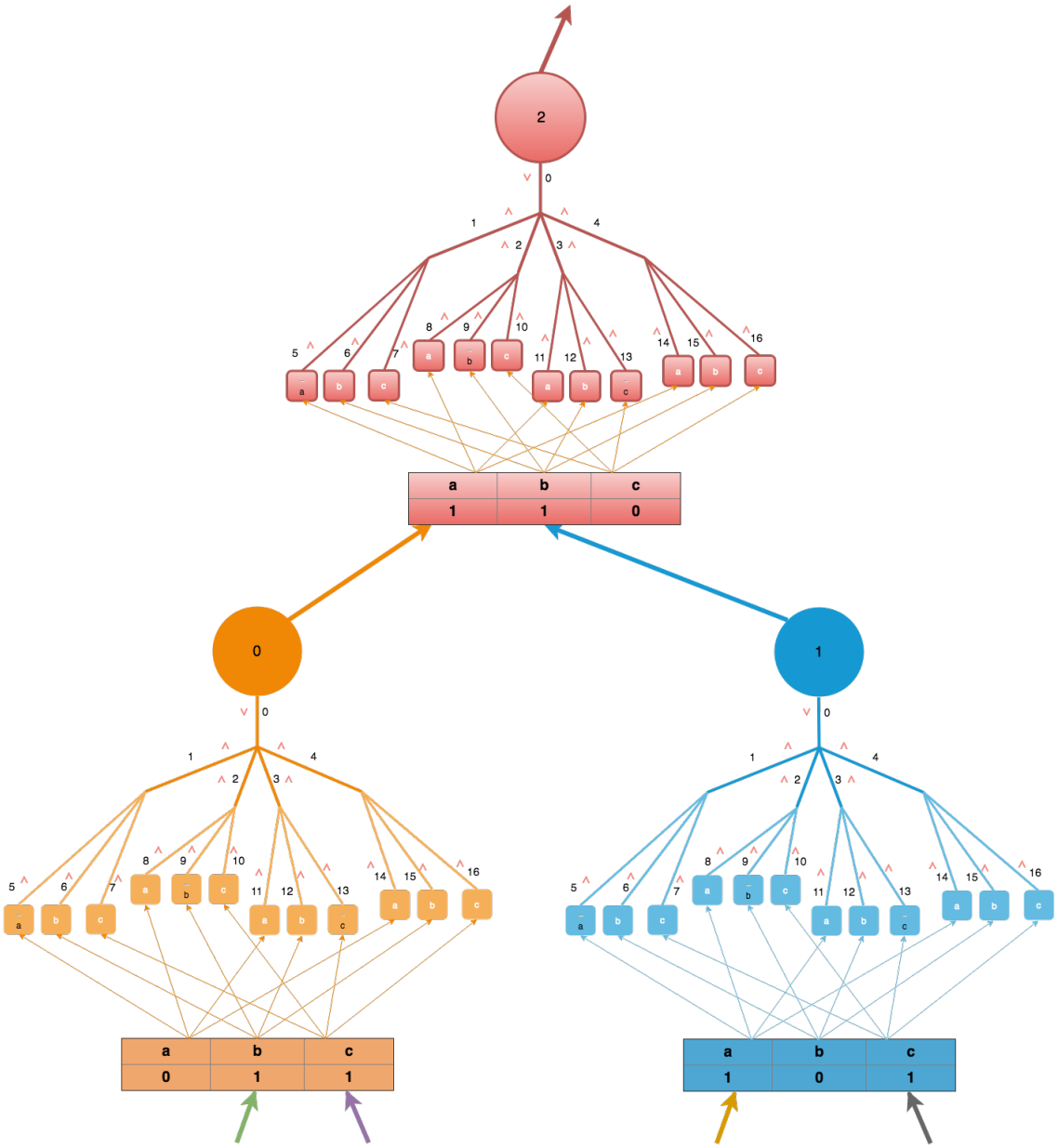

Figure 4.3: Example of a network of three neurons in the Bridge Mode.

appropriate to use only in the Bridge mode. It uses the parameters NUM_BRIDGE _SYNAPSES and NUM_BRIDGE_THRES. Every variable in the expression is represented by either 1 or 0. The value 1 means that the corresponding synaptic branch has a type AND. Meanwhile, a value 0 means that the type of the branch is NOT. The function will produce the expressions as in the table 4.1 in a very efficient way (with up to 20 variables quickly) and store them in the

```
vector<string> nvBrMasks
```

Permuting all combinations of zeros and ones has a worst-case performance of $\mathcal{O}(2^n)$. However, because this particular procedure utilizes the function

```
next_permutation();
```

from the Standard Library, the running time becomes much smaller. The reason is that this function rearranges the input items into the next lexicographically greater permutation. It is easy to see that the complexity of the next_permutation() is linear in half the distance between first and last actual swaps [3]. Therefore, the algorithm has a smaller upper bound.

2. `BooleanFunction(int bID, int CURRENT_TIME)`

   Boolean function that outputs *true*, *iff* the neuron for which this function was called "fires", otherwise outputs *false*. Accepts branch ID and current time from the time loop. Operates as follows. Recursively calculates boolean values by applying boolean operations specified for the branches. Specifically, the branch will call the same function on all child branches and will get boolean values, that it stores in

   ```
   vector<bool> vChildBrBoolResult(MAX_NUM_BRANCHES)
   ```

   If all child branches of a given branch produce value *true* and the branch type is AND, then the function will apply operation conjunction to all the values stored in the vector. Then it will apply the same boolean operation on all synapses sitting on that branch. In case at least one of the child branches or one of the synapses has a value *false*, the current branch will output *false*. Then, the parent branch of the current branch will evaluate its own child branches and apply its own boolean operation to all branches including the current as well as all synapses on that parent branch if any. In addition to that, all synapses live in time. Subsequently, this boolean function considers value of every synapse to be 1 (= *true*), only if the variable CURRENT_TIME of the time loop has a value between values of the variables *fromTimeStamp* and *toTimeStamp*. Otherwise, the function returns *false* to the parent branch (if the branch type is AND) and returns *true* (if the branch type is NOT).

   Note. It is assumed, that the actual operation is AND for the type NOT, thus returning *false*, when the AND would return *true*, and the other way around.

3. `signalToOutBridgesAndBranches(int CURRENT_TIME, vector<Neuron> &vNeurons)`

   The purpose of this function is to set values of bridges to 1 and update the time variables (*fromTimeStamp* and *toTimeStamp*). Once the bridge values that represent a set of synapses in the neuron are updated, the synapses are also updated - the values and the time variables. Overall, the operation sequence is the following:

   a) If the current neuron "fired", then it has to notify its neighbors (through the outgoing edges) and transmit the signal to them.

b) The signal gets first into the Bridge of every neighbor.

c) The synapse values that correspond to that Bridge value get updated (values and time variables).

For instance, suppose the Bridge of one of the neighbors has the structure as in the table 4.2:

a	**b**	**c**
1	0	0

Table 4.2: Masks for the child branches that represent variables.

If the "fired"neuron updates value of a cell in the Bridge that corresponds to all synapses a to 1, then all corresponding synapses a in that neuron are updated to the value 1 (along with the proper time attributes).

4. `signalToOutDegreeNeurons(int CURRENT_TIME, vector<Neuron> &vNeurons)`

   Same as above, with the difference that it does not set the values to the bridges, but only to synapses of the neighbors of the "fired"neuron.

5. ```
   void plastify(int plastFlag,
                 int CURRENT_TIME,
                 vector<Neuron> &vNeurons,
                 int nrDestID,
                 int brDestID,
                 int sDestID)
   ```

   This function can add and remove neighbors from neuron. Its purpose is to allow for *Neuroplasticity*. It can be used to increase the intensity of the connection between neurons as well as to weaken the one. When the variable *plastFlag* has a value 1, then the current neuron adds a new connection specified by the IDs which are passed as parameters to the function along the flag. When it has a value -1 then it removes the connection specified by the IDs. The result of work of this function can be seen on the picture 4.4 below. The second neuron must have fired, however, the connection between first and second was removed before the process began. Later, at time three, the function was called again to add this connection, and as a result, the neuron two fired starting from the next iteration of the time loop.
   ```

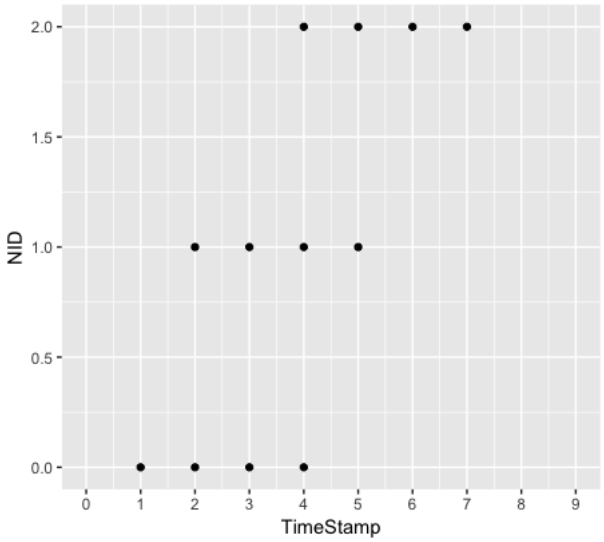

Figure 4.4: Example of execution of the Plastify function.

4.5 Functions that connect neurons.

1. `ConnectNeuronsRand(int *NN, vector<Neuron> &vNeurons)`

   The parameter CONNECTIVITY_RATIO defines how many neighbors (outgoing edges) every neuron instance will have. The exact number is a floored product between this parameter and the total number of neurons in the network. Proceeds as follows:

   a) For every neuron generate a random ID of some other neuron.

   b) Choose a random branch ID in that neuron.

   c) Choose a random synapse ID on that branch.

   d) Put these IDs into the vector that holds neighbors of current neuron.

   e) Repeat with the next neuron.

2. `ConnectNeuronsLayered(vector<Neuron> &vNeurons, int s, int d)`

   The goal is to connect every neuron with ID = **s** from the current layer to the neuron on the next layer given by ID = **d**. Proceeds as follows:

   a) For every neuron **d** choose a random branch ID in that neuron.

   b) Choose a random synapse ID on that branch.

   c) Put these IDs into the vector that holds neighbors of current neuron **s**.

3. `ConnectNeuronsCircle(vector<Neuron> &vNeurons, int s, int d)`

   Set that the neuron with ID = **s** has neighbor neuron with ID = **d**. Put these values

into the vector of neighbors stored at **s**.

4. `ConnectNeuronsBridge(vector<Neuron> &vNeurons, int s, int d)`

   Essentially, this function operates as a regular reading function discussed earlier. Initially, all neurons get connected to the neuron with ID = NUM_BRIDGE_NEURONS - 1. In other words, it is the last neuron that receives all the connections. A user can define other fashion of the connectivity.

4.6 How to use the Simulator.

User can set value for the parameter OPERATING_MODE from 1 to 6. A full list parameters with the description of each, as well as a relevant mode for each parameter is show in the table 4.3.

Once the user runs the simulator it will start processing the data and calculate the statistics. Statistics of the simulator will be in folder "cmake-build-debug"(when the simulator is run in the debug mode, otherwise in the same directory as the binary file) in the file "NeuronStatistics.csv".

Note. Once the data that was generated in several working modes ("Layered Mode", "Circle Mode", "Bridge Mode") is read from a file or generated anew, data for synapses that are active should also be applied (see examples in the *Invoke* functions in *main.cpp*).

Note. It is important to put the right file as a source of data in the reading function.

4.7 Detailed description of the Bridge Mode.

The motivation for describing this mode in detail is in the fact that it is the most promising mode to have a topnotch performance on field-programmable gate arrays (FPGAs). For this reason, it is important to have the best understanding of it.

1. Operation sequence.

   a) Once the program is run for execution, the function *int main()* takes control. After the check that the current operating mode is the Bridge mode (OPERATING_MODE == 5), the function

   `void invokeGenBridgeMode(vector<Neuron> &vNeurons)`

   is called, where the *vector<Neuron> vNeurons* contains all neurons in the network.

   b) Next, the function

   ```
   void genNeuronDataBridge(int *NN,
                            vector<Neuron> &vNeurons)
   ```

   creates the entire network given the parameters, the variable and the vector passed to the function.

   c) After that, the program creates connections between the neurons. It is done by the function

Parameter Name	Relevant Mode	Description	Default value
MAX_TIME	ALL	Duration of the Simulation	10
TIME_CONST	ALL	How long it takes for signal to decay completely	3
MAX_NUM_BRANCHES	Generate random	Maximum number of branches each branch in neuron has	4
MAX_NUM_SYNAPSES	Generate random	Maximum number of synapses each branch in neuron has	3
CONNECTIVITY_RATIO	Generate random	Degree of the number of neighbors each neuron is connected with others.	0.6
NUM_NEURONS_PER_LAYER	Generate layered, Generate Circle	Number of neurons that exist in one layer	5
NUM_LAYERS	Generate layered	Number of layers in the network	3
NUM_BRIDGE_NEURONS	Generate Bridged	Number of neurons in the network	3
NUM_BRIDGE_SYNAPSES	Generate Bridged	Number of variables (synapses, branches) in a Bush.	3
NUM_BRIDGE_THRES	Generate Bridged	Threshold on the number of active (= 1) variables (synapses) that trigger neuron to "fire".	2

Table 4.3: All parameters of the Simulator.

```
void ConnectNeuronsBridge(vector<Neuron> &vNeurons,
                          int d)
```

in the following for loop:

```
for (int iso = 0; iso < NN - 1; ++iso)
{
    ConnectNeuronsBridge(vNeurons, iso, NN - 1);
    vNeurons[iso].setNOutDegree(1);
}
```

Note. In the loop above all neurons in the network except the last one are connected to the last neuron. This can be changed depending on new requirements. Variables s and d are source and destination IDs of the neurons.

d) Inside of the Connect function, an instance of the neighbor structure for the source neuron will be created and filled. The code below illustrates the creation of the neighboring structure that will be used by the source neuron:

```
Neuron::neigb ngb;
```

The structure will be filled with the destination ID of the neuron giving it the value of d using the following code:

```
ngb.nid = d;
```

Since there are no branches in the bridges, the value assigned to this property will be -99, which means it is unavailable. The code below illustrates the assignment:

```
ngb.bid = -99;
```

Next, synapse ID for the neighbor will take the value of the next available cell ID in the bridge. If the destination neuron has all bridge cells filled, in other words, if the number of neurons that are connected to the destination neuron exceeds the value of the parameter NUM_BRIDGE_SYNAPSES which defines the number of cells in the bridge, then, all neurons that try to connect to the destination neuron will have an ID of the last cell of the bridge for the values of the attribute synapse of the neighboring structure. The code below shows this assignment:

```
ngb.sid = vNeurons[d].getAvailCellInBridge();
```

Note. There is a possibility to connect source neuron to random IDs of cells in the bridge to speed up the performance and decrease the complexity.

Finally, the entire structure is put into the vector that stores information about neigbors of the source neuron. This is done using the following operation:

```
vNeurons[s].nvNeigb.push_back(ngb);
```

e) The code below defines the initial values of the neurons with the goal to show that the neural network works and single neurons "fire".

```
int sBr = vNeurons[0].getnNumHostBranches() + 2;

vNeurons[0].nvBran[sBr].vsyns[0].sval = true;
vNeurons[0].nvBran[sBr].vsyns[0].fromTimeStamp = 1;
vNeurons[0].nvBran[sBr].vsyns[0].toTimeStamp =
                          1 + TIME_CONST;
```

```
vNeurons[0].nvBran[sBr + 1].vsyns[0].sval = true;
vNeurons[0].nvBran[sBr + 1].vsyns[0].fromTimeStamp = 1;
vNeurons[0].nvBran[sBr + 1].vsyns[0].toTimeStamp =
                          1 + TIME_CONST;


vNeurons[1].nvBran[sBr].vsyns[0].sval = true;
vNeurons[1].nvBran[sBr].vsyns[0].fromTimeStamp = 2;
vNeurons[1].nvBran[sBr].vsyns[0].toTimeStamp =
                          2 + TIME_CONST;

vNeurons[1].nvBran[sBr + 1].vsyns[0].sval = true;
vNeurons[1].nvBran[sBr + 1].vsyns[0].fromTimeStamp = 2;
vNeurons[1].nvBran[sBr + 1].vsyns[0].toTimeStamp =
                          2 + TIME_CONST;
```

It is easy to see, that when the *Number of Host Branches in the Network* is equal to 3 and having default parameters from the table 4.3 leads to the following results:

- Variable *sBr* has value 6.
- Neuron with ID = 0 will have its synapses activated from and until specific time. They are activated, however, only for two branches. They have IDs = 6 and 7. The reason these branches are chosen is that the first mask that was generated earlier for the branches will have value 011. That means that the first branch in the first Bush will have type NOT and the second and the third branches in the same Bush will have *bool* type AND. This will lead to a negation of value of the first synapse. It will also lead to taking plain values of the synapses that are sitting on the branches two and three while the Boolean function is executed in the time loop.
- The same assignment happens with the neuron with ID = 1 as with the neuron with ID = 0.

f) The control of execution goes to the time loop, which was described earlier.

2. Setting up a new network in the Bridge Mode.

   Let us consider an example when the network should have the following structure: 4 variables (a $\wedge$ b $\wedge$ c $\wedge$ d) while at least 3 any of them must have value 1 to produce 1 in the output after evaluating the expression. There are also 4 neurons in total. Then, the following modifications will be made:

   a) The function *BranchMaskGeneration()* will find all expressions that will give this result and produce the required masks for the branches. It will store them in the *vector<string> nvBrMasks* in the neuron instance. These masks are:

   0111
   1011
   1101
   1110
   1111

In total 5 masks.

b) Parameters NUM_BRIDGE_SYNAPSES and NUM_BRIDGE_THRES will have values 4 and 3 respectively.

c) It is necessary to comment a section of code in the function *void invokeGenBridgeMode(vector<Neuron> vNeurons)* that explicitly defines which synapses on which branches are activated. Also, it is necessary uncomment a *for loop* that activates synapses in this function. After making the changes a user can run the neural simulator. By plotting the results, using the R script it will be possible to see a picture as the illusion 4.5 below. The raster plot below illustrates activity of 4 neurons that spike. ID of each Neuron is along the ordinates axis, while along the axis of abscissas is the time dimension. Every dot on the picture means that some neuron "fired"at a given time.

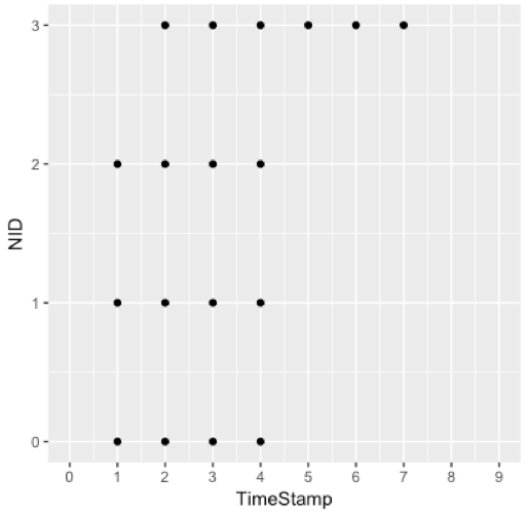

Figure 4.5: Example of Firing Activity of 4 Neurons.

4.8 Description of the generated data file.

The file that was generated as a result of execution of one of the generating functions and which defines each neuron has the following structure. In the first line, there is a number *NN* that defines the number of neurons in the network, *NN* blocks follow. Each block contains a number of Branches *Nbr* of each neuron on the first line, *Nbr* blocks follow within each *NN* block. In each of these *Nbr* blocks, on the first line there is a definition of a type of the branch (AND, OR, NOT), then a number *numBrChild* of child branches for this branch, followed by a list of length *numBrChild* of IDs of child branches. The next line contains a number of synapses *numSyn*, and a list of values of these synapses (0, 1).

After the *NN* blocks, there is a blank line after which, there is a definition of how neurons are connected. On the first line, there is an ID of a neuron that has an outgoing edge (axon) and a number of neighbors *neurNeigb*. Then, *neurNeigb* of lines follow, each containing ID of neighbor neuron, branch ID, to which the neuron is connected and synapse ID on that branch. The overall definition can be seen on the picture 4.6 below that illustrates the idea.

Note. When the file is generated in the Bridge mode, in addition to the number of neurons that is specified in the very first line of the file, there are also the number of Host branches and the number of synapses in the Bridge.

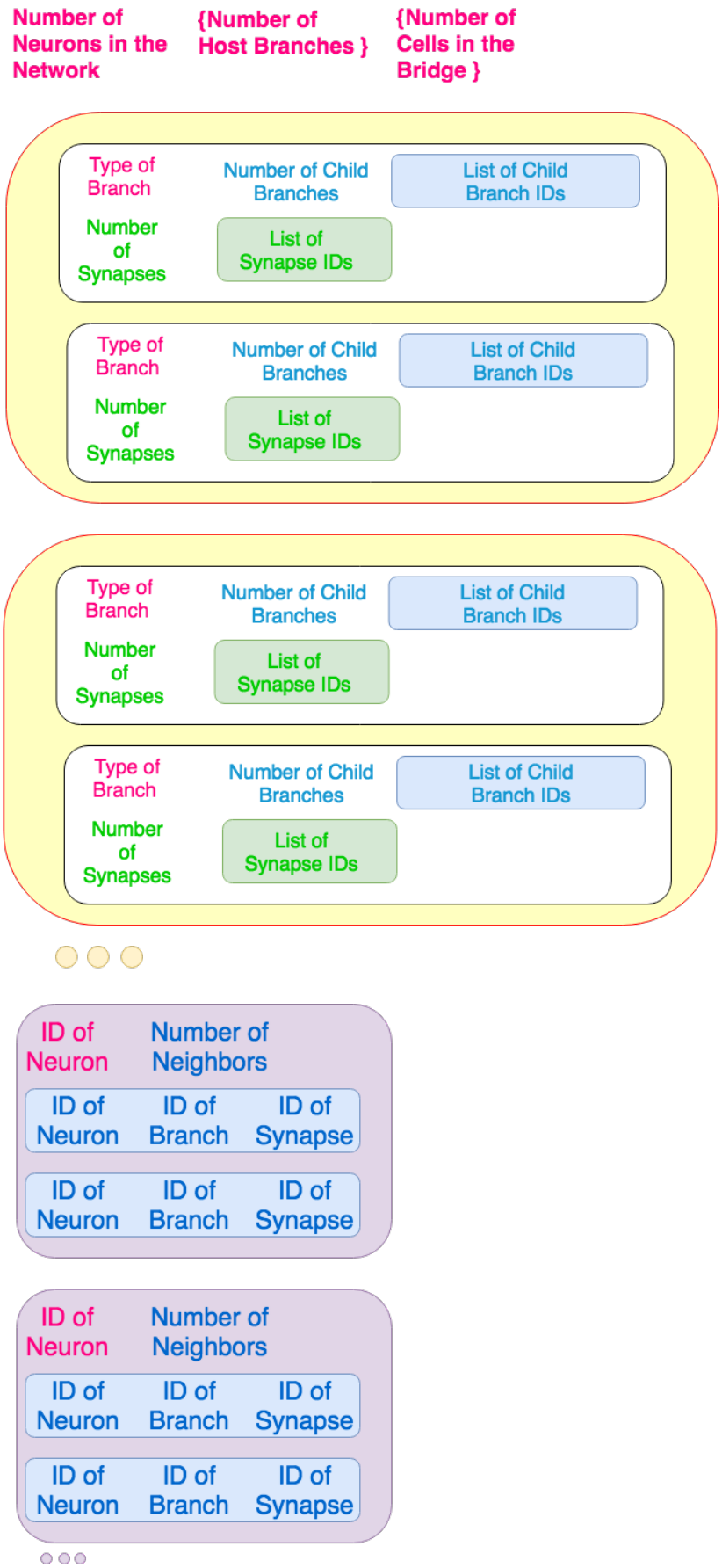

Figure 4.6: Structure of the Input Files.

4.9 Visualizing results.

R language was used to visualize the results of the Fire Trace. Specifically, to draw the plot a *ggplot* library was used. This library was designed for creating elegant plots and is widely used by the research community [19]. The following function call was invoked to draw the plot 4.7. There are two dimensions: neuron (Y axis) and time (X axis).

```
ggplot(dat, aes(x = TimeStamp, y = NID) ) +
  geom_blank() +
  geom_point(dat = subset(dat, Fires == 1)) +
  scale_x_continuous(breaks = F_int_time_breaks(1) )
```

The variable "dat" contains the data from the file of fire statistics. This file is created when the simulator finishes the execution.

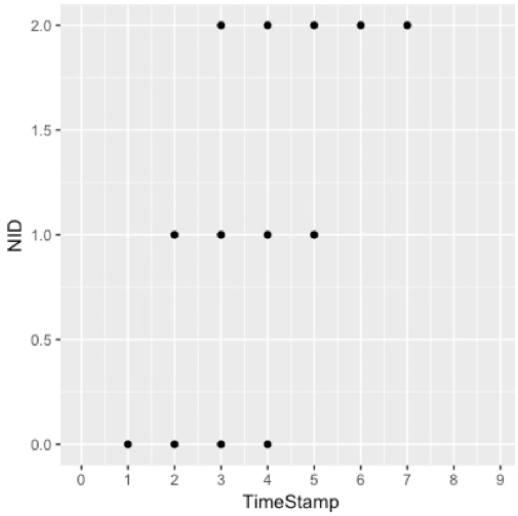

Figure 4.7: Example of Firing Activity of 3 Neurons.

5 Evaluation

5.1 Biological characteristics

It is widely accepted [9] that the NEURON simulation environment provides the highest closeness to biological concepts of neural networks, for instance, anatomical and biophysical properties of a neuron cell. It is designed to create single neurons with many details and to simulate small number of them.

Meanwhile, NEST Simulator has very wide range of possibilities to adjust network structures. It is extendable for new models of neurons and synapses. In general, it can have many biologically realistic properties.

With the Novel Neuron Model implemented during this Thesis it is possible to create different types of networks —from randomly structured neurons with random connections between them, to layered networks, and to bridged networks. Also, the simulator allows for neural plasticity during the run-time.

Finally, Tensor Flow incorporates only very basic ideas of biological neurons, thus, it is far from being comprehensive and close to the biological concepts.

5.2 Performance on traditional chips.

Several studies suggest that low complexity and high level of optimization to run on traditional hardware allow Tensor Flow to lead in this role. It can also run distributed training on multiple computers.

The Novel Neuron Model contains many low-level adjustments and heuristics to boost the performance. Implementation in C++ makes the simulator to work very fast on the traditional computational hardware.

A number of authors have recognized that Nest Simulator is fast, yet its grown complexity in compare to the other models makes the simulation slower.

At the same time, the NEURON environment is highly complex and very detailed. Thus, its performance is not very good relative to the alternatives.

5.3 Performance on FPGA.

The Novel Neuron Model based on Boolean functions is clearly the leader in the ability to be applied to an FPGA as its neuron structures were specifically designed and developed for that. Namely, on every programmable logic block of an FPGA it is possible to evaluate the values on the branches of the dendritic trees. The bridges will provide a high-speed access, a centralized control and modification of the synapses within the dendritic trees.

Present studies suggest that Tensor Flow was not created to show a top performance on FPGA. It is however, possible to port it, since it is an open source software [12]. Additionally, its simple default neuron structure should make it work fast.

It was reported in literature that Nest is fairly complex and was not designed to operate on non-traditional CPU. It is, however, simpler than the NEURON simulator and it is an open source software, so I can be adjusted.

Generally speaking, environment of NEURON is highly complex, and it is not expected that it could be efficiently implemented on a simple programmable chip.

6 Future Work

1. While there already exists a global neuron ID that allows to access each neuron in constant time, having a global synapse ID would allow for better Neuroplasticity and distributed computing.
2. A good improvement would be to have a backup of the initial data for all neurons. For instance, once one simulation that involved Neuroplasticity is done, it would be possible to quickly recover the initial state and to conduct another experiment.
3. Porting the model to an FPGA would bring a huge performance boost. These devices have logic blocks that can be programmed to execute the truth functions. This makes the Novel Neuron Model and the programmable array a perfect match.

   De facto these devices have been used for a variety of tasks already and produced significant improvements in performance in compare to traditional CPUs, including for similar tasks. A case in point is that there is already a product that uses Neural Networks and a custom programmable hardware. It increases the speed of the object recognition by up to six times in compare to alternatives that are based only on software [1]. The solution allows to quickly search through low-resolution videos to detect faces [2].

   Yet, because of the elegance of the model, a good design coupled with an efficient implementation, it is reasonable to assume that the Novel Neuron Model executed of an FPGA would outperform any other solution.

7 Conclusion

The Novel Neuron Model developed as a part of this Thesis employs dendritic trees, similar in structure to the trees in the biological neurons. The software that implements the model allows for network plasticity during run-time —a core functionality of the real neural networks. Equally important, it uses Boolean functions to calculate the binary values of synapses on these trees. The software can also process the data in a centralized manner using special data structures, *Bridges*, which boosts the performance. This makes the model fit exactly into the architectural ideas of a field-programmable gate array and can be efficiently ported to it.

Ultimately, with the new approach it will be possible to perform training of the model on a substantially larger number of instances. Therefore, it will be feasible to produce more precise results while taking the same amount of time and consume less energy.

Given these points, it is reasonable to expect that the Novel Neuron Model, implemented on a suitable hardware at scale is a strong competitor to the efficiency of neural networks of a brain. To put it differently, applying this model can allow for new functionality of technical solutions that serve people.

Bibliography

[1] Brainchip introduces world's first commercial hardware acceleration of neuromorphic computing. https://ir.brainchipinc.com/press-releases/detail/39/brainchip-introduces-worlds-first-commercial-hardware. [Online; accessed 27-July-2018].

[2] Civil surveillance solutions. https://www.brainchipinc.com/products/civil-surveillance-solutions/brainchip-studio. [Online; accessed 27-July-2018].

[3] Standard template library: Algorithms. http://www.cplusplus.com/reference/algorithm/next_permutation/. [Online; accessed 23-July-2018].

[4] Nasa neural network project passes milestone. https://www.nasa.gov/centers/dryden/news/NewsReleases/2003/03-49.html, 2003. [Online; accessed 23-July-2018].

[5] Artificial neural networks as models of neural information processing. https://www.frontiersin.org/research-topics/4817/, 2018. [Online; accessed 27-July-2018].

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Thomas E. Akam and Dimitri M. Kullmann. Oscillations and filtering networks support flexible routing of information. In *Neuron*, 2010.

[8] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3):349—398, December 2007.

[9] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, New York, NY, USA, 2006.

[10] Chris Edwards. Growing pains for deep learning. *Commun. ACM*, 58(7):14–16, June 2015.

[11] Scheler G and Schumann J. Boolean analysis of dendritic structure. *2nd Annual Symposium on Biological Complexity: Genes, Circuits and Behavior 2008. F1000Research 2014*, page 5:552 (poster).

[12] Google. Tensorflow. `https://github.com/tensorflow/tensorflow`, 2018. [Online; accessed 23-July-2018].

[13] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117(4):500–544, Aug 1952. 12991237[pmid].

[14] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, Nov 2003.

[15] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sept 2004.

[16] E. M. Izhikevich and R. FitzHugh. FitzHugh-Nagumo model. *Scholarpedia*, 1(9):1349, 2006. revision #123664.

[17] Jens Kremkow, Laurent U. Perrinet, Guillaume S. Masson, and Ad Aertsen. Functional consequences of correlated excitatory and inhibitory conductances in cortical networks. *Journal of Computational Neuroscience*, 28:579–594, 2010.

[18] G Scheler. Learning intrinsic excitability in medium spiny neurons [version 2; referees: 2 approved]. *F1000Research*, 2(88), 2014.

[19] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.