Packet Classification Using Growing Hierarchical Self-Organized Map

BY

MARCO MONTAGNA
B.S., Politecnico di Torino, Turin, Italy, 2017

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

ACKNOWLEDGMENTS

Firstly, I would like to thank my thesis advisor Professor Amit Trivedi for giving me the opportunity to work with him on this challenging problem; from our first meeting to last one he has always been available to help me and he always gave me the push to do better every time.

I am deeply thankful to my second advisor Professor Mariagrazia Graziano, Her determination and confidence helped me to see from a different point of view when I faced difficulties.

Besides my advisors, I would also like to thank Professor Vamanan Balajee for his time and effort on this study, it was important for me to have the opportunity to work with someone who knew the subject so well.

I am also grateful to Rahul for his great help when I needed someone to confirm my ideas and for his composure that has always helped me to see things more calmly.

I am extremely grateful to my friends in Chicago who have been like a family during this period abroad making this experience something that I will never forget.

My deepest gratitude goes to my girlfriend that endured and supported me even 4,504 miles away. Her patience has made it all easier for me.

I would like to thank one by one all my friends in Italy, they showed me that even if we don't see each other for a year, nothing change.

ACKNOWLEDGMENTS (continued)

Last but not least, I must express my very profound gratitude to my family. My mum, my dad and my brother are the reason of who I am today and this accomplishment would not have been possible without them and their everyday support.

MM

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SOM	Self-Organized Map
GHSOM	Growing Hierarchical Self-Organized Map
G-SOM	Growing Self-Organized Map
FPGA	Field-Programmable Gate Array
SDN	Software-Defined Networking
TCAM	Ternary Content Addressable Memories
IP	Internet Protocol
IPv6	Internet Protocol version 6
ANN	Artificial Neural Network
mqe	Mean Quantization Error
DCU	Distance Computing Unit
MDD	Minimum Distance Detector
FIFO	First In First Out
ACL	Access Control List
FW	Firewall
IPC	IP Chain

SUMMARY

The main goal of this thesis titled "Packet Classification using Growing Hierarchical Self-Organizing Map" is to validate the possibility of using a new approach for a well know problem. The core idea is to present a possible implementation of a Growing Hierarchical Self-Organized Map (GH-SOM) for use in packet classification in the context of software-defined networking (SDN). SDN applications are characterized by frequent network configuration updates on-the-fly and a large number of rules thereby requiring a highly flexible packet classification mechanism. Moreover, as network bandwidths continue to grow, a high-performance packet classification is imperative.

Today's technology adopted for packet classification does not scale well in throughput and power consumption as number of rules increase. The main change proposed in this implementation is to move all the complexity to an offline phase, train a modeled neural network and exploit its properties to provide a more efficient solution for packet classification during run time. Different input space and different parameters have been tested for training the network; multiple simulations have been performed in order to find the best solution to this problem. The work proposed consists of a trained neural network that classifies the rules of a given data-set, then input packets are applied and rule with highest priority is determined.

Multiple simulation has been carried out for finding the best feature map for the different layer of the GH-SOM structure, in order to provide the best accuracy in every application. The MATLAB code was designed to give the possibility to the designer to easily adjust the training parameters of the neural network in order to provide a general implementation of the structure which can be adapted in multiple situations.

CHAPTER 1

INTRODUCTION

1.1 Packet Classification problem

From the beginning of the digital era, the amount of network traffic to be handled has grown exponentially and large datacenters and enterprise networks have become popular and necessary in order to manage this exponential growth. There exist multiple paradigms useful for managing the networks within these datacenters and one of the most popular is the Software-Defined Networking (SDN). In traditional networks, control-plane functions (e.g., routing, load balancing, access control) are distributed among network switches and routers, while in SDNs control-plane functions are managed by a logically centralized controller. The interface between the SDN controller and switch data-plane is handled by OpenFlow [1].

Although SDNs have changed how we manage large networks, the architecture and algorithms used to handle packet classification have remained largely unchanged; the goal of this thesis is to propose a new approach that can be suitable for improving packet classification.

Packet classification is defined as the management of the data flow (represented by data packet) in a network router. This topic has been widely studied in the recent years in order to accelerate network applications. Packets can be classified by source address, destination

[1]OpenFlow is a communication protocol through which is possible to access the data plane of a network switch over the network [1].

address, source port, destination port, mask and protocol type; field format will be discussed in more detail in the next chapters.

The packet classification problem attempts to identify the highest-priority rule that matches a given network packet out of a set of rules. Each rule specifies a collection of packets identified by a combination of packet header fields. A priority is assigned to each rule in order to avoid possible conflicts in case of multiple matching rules. A rule's packet header format depends on the protocol used: in this work the Internet Protocol version 4 (IPv4) protocol has been considered.

In this work we will show that a hierarchical tree of SOMs is fast, flexible, and suitable to fulfill the requirements of packet classification in SDN applications. We aim to present an approach to segregate rules based on priority and header field which are filtered into different SOMs and stored in bins at the bottom of the SOM-based tree. The main goal is to overcome limitations of current state-of-the-art solutions by exploiting simple neural networks.

1.2 Classification of existing solutions

Before starting with the analysis of the proposed solution, it is important to provide a background on existing solutions in order to fully understand the outcomes of the work.

Most of today's solutions applied for matching incoming packet against a set of flow rules employ Ternary Content Addressable Memories (TCAM). TCAM is special high performance memory which searches its entire content in a single clock cycle performing a brute force search. In other words, for every incoming packet the whole contents of the memory is searched in order to find the best match for the given input. Although TCAMs are characterized by a high search speed, they do not scale well in throughput and power as the number of rules increases. Moreover, SDNs are characterized by frequent updates to rules and because of the fact that TCAMs physically order rules based on priority, they perform poorly with high speed dynamic changes.

There are three key limitations of TCAMs as described below:

1. In traditional networks, rules are grouped manually by network administrators and each of these groups contain hundreds of similar rules. In Software-Defined Network applications, this rule aggregation process is automated by SDNs and rules are added or updated on demand during the lifetime of the SDN. As stated in [2], OpenFlow allows packets to be matched against a large set of header fields which increases the complexity. Limited TCAM capacity is a serious limitation to fully exploit an SDN's potential [3].
2. The rate of updates supported by TCAM-based switches is about 50-100 updates per second [4]. Therefore, if the rule update rate of an SDN application is higher than this threshold, the update performance is decreased dramatically [5].
3. The last reason has been already hinted in previous sections: as datacenters grow in scale, poor scalability of TCAMs becomes an ever more critical bottleneck.

The approaches currently adopted to address the packet classification problem can be divided in two categories: TCAM and algorithmic.

- *TCAM* works at the hardware level, exploiting parallel brute-force search of all rules for every incoming packet; this leads to high energy and space overhead in the hardware. As already mentioned, TCAMs physically sort rules based on highest priority and because of this, frequent updates are slow and complex. Considering the worst case all the rules should be moved to make space for the top priority rule updated.

- *Algorithmic schemes* try to reduce the search complexity of the TCAM protocol exploiting multiple methods. The current state-of-the-art algorithmic approaches exploit variants of decision trees.

Another solution currently adopted to address the packet classification problem exploits the implementation of deeply pipelined SRAMs instead of TCAMs; in fact, SRAMs offer higher throughput despite a larger number of memory accesses. In any case, the number of memory accesses required per packet match is a significant hurdle to their scalability because line rates grow faster than SRAM speeds.

1.3 Geometrical interpretation of the problem

Before starting, it is important to point out that each rule can be defined by its different components; in IPv4 this data is made up of six items, each of which is defined by a field of the packet header. In our investigation, four among the six fields have been used. A more in depth investigation into IPv4 and packet header has been presented in 2.1.

The idea of a geometric interpretation of classification is due to Lakshman and Stiliadis [6]. They stated that each rule can be considered as a hyper-rectangle with dimensions specified by the rules header fields. For a better comprehension of the geometrical analysis it is possible to consider a 2D example, remembering that in IPv4 the rules have six fields and thus six dimension (only four of them have been considered for the implementation of the system).

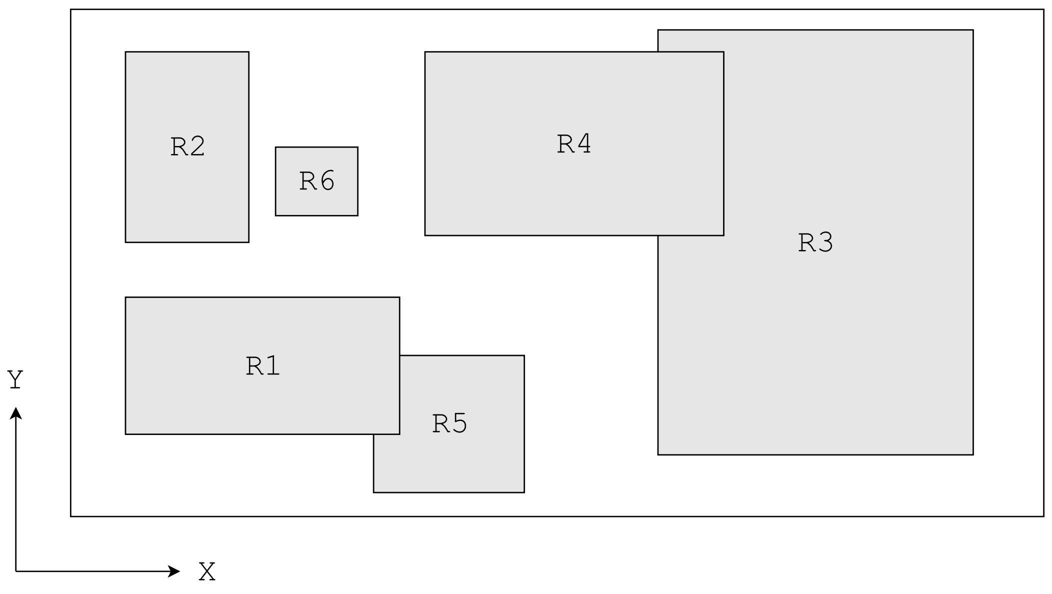

Figure 1: 2D geometrical representation of packet classification.

In the example above, the dataset is composed by six rules and each header's field define a range of values in the space. The intersection of two ranges in a 2D space forms a rectangle. As already stated presenting the problem, the goal of packet classification is to find the highest priority rule that matches the incoming packet. This process, from a geometrical point of view, can be seen in this way: the incoming packet represent a point in the 2D space and the highest priority matching rule is the rectangle with the smallest area that contain the point. This geometrical interpretation was fundamental in the first steps of the project as it was possible to analyze the problem from a previously studied, and better understood point of view.

State-of-the-art solution exploit decision tree to split rules into different bins in order to face the packet classification problem . Each node of the mentioned tree represents a cut in

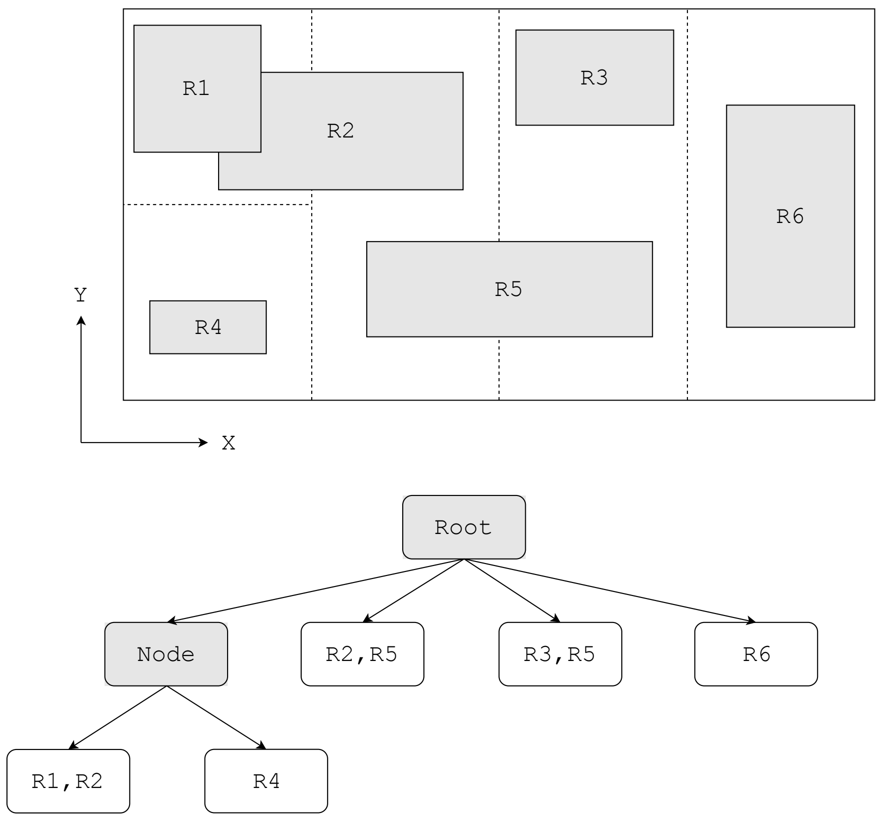

Figure 2: 2D rule-set and decision tree.

the space and the leaves represent the rules. From a geometrical point of view, decision trees exploit the idea created in [6] and represent rules as hyper-cubes in the multi-dimensional space; cuts divide the space into sub-spaces which partition rules into smaller subsets. The algorithms recursively partition the smaller spaces until the sub-spaces only contain a handful of rules that can be manually searched (brute force method using TCAMs).

Figure 2 shows a sample of a two-dimensional rule-set and the corresponding decision tree. Every node divided into leaves is referred to as a *parent*, while each leaf node is called *child.*

In the example shown, no more than two rules need to be searched at any leaf node and they match mutually exclusive sets of packets. During classification of new packets, the input starts at the root node of the decision tree and traverses the parent nodes towards its appropriate leaf node where a matching rule could be found.

Even though decision trees decrease search complexity, they do not scale well with increasing number of rules. One critical drawback of using decision trees is the high update complexity. Frequent updates are one of the main characteristic of packet classification in SDN applications. In [5] is stated that rule updates may increase tree-imbalance to the point where the decision tree needs to be re-balanced to keep the search time under control. One of the main problem regarding decision trees is the re-balancing process. In state-of-the-art solutions, large rules are replicated across multiple sub trees, increasing memory overhead. In a GHSOM-based system, the stored rules are not repeated in multiple locations and therefore the number of movements of rules is lower.

1.4 State-of-the-art solutions

State-of-the-art solutions regarding packet classification are represented by presented decision tree algorithms. [7] can be considered as a good starting point for the analysis of these algorithms.

The best algorithms exploited for packet classification in current systems are:

- HiCuts
- HyperCuts
- EffiCuts

EffiCuts represent an evolution of HyperCuts which themselves are an evolution of HiCuts.

1.4.1 HiCuts

The first idea for partitioning the multi-dimensional rule space exploit HiCuts. HiCuts performs the presented operation through a decision tree with the goal of separating the rules in different leaves. Starting from the root node (unit 0) that contain the whole dataset, HiCuts cuts the space among a single dimension creating multiple equi-sized sub-spaces. Child nodes are represented by these sub-spaces. This split process is iterated until the number of rules in a child node is below a certain threshold. For classifying an incoming packet, we traverse down the decision tree in accordance with the decisions taken at each level, and finally the rules at the leaves of the tree are searched linearly for finding the best matching rule for the incoming packet.

In the paper [7], it is presented an example of how HiCuts works. Modifying slightly the Figure 2 it is possible to obtain the Figure 3, useful for understanding how HiCuts works. Initially, the 2D plan is divided along the first dimension (Cut 0); the first layer of leaves of the tree contain the sub-space generated by the first cut. Subsequently, a second cut is carried out (Cut 1): the first sub-space is divided in two more child along the other dimension. Multiple cut can be performed if a leaf contain too many rules.

The critical drawbacks related to HiCuts are (1) the fact that the tree depth is dependent on the distribution of the rules in the rule space and (2) due to the large amount of redundancy in the tree. The first limitation comes from the inability of HiCuts to perform multi-dimensional cuts on the rule space; this problem is solved with the improved Hypercuts. On the other

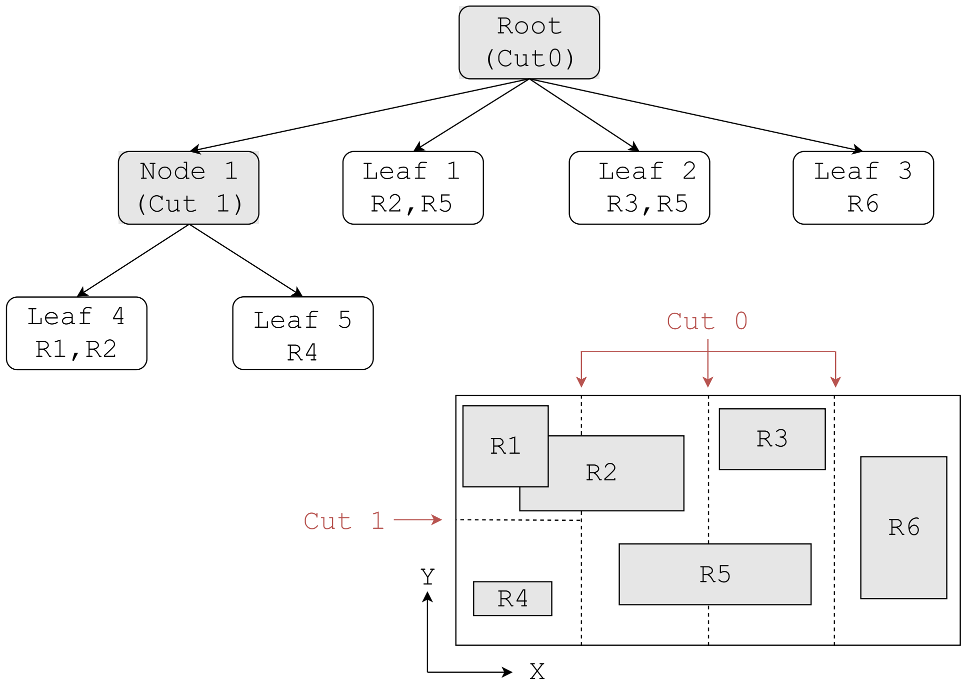

Figure 3: HiCuts in a 2D rule space.

hand, redundancy cannot be captured if siblings do not share the totality of rules (partial redundancy).

1.4.2 HyperCuts

HyperCuts is a decision tree-based algorithm and it has been proposed to address HiCuts' shortcomings. It represents an evolution of HiCuts and is based on simultaneous multiple dimension cuts. Hypercuts also addresses the partial redundancy problem: it recognize when a group of rules is common to multiple siblings and moves it to the parent.

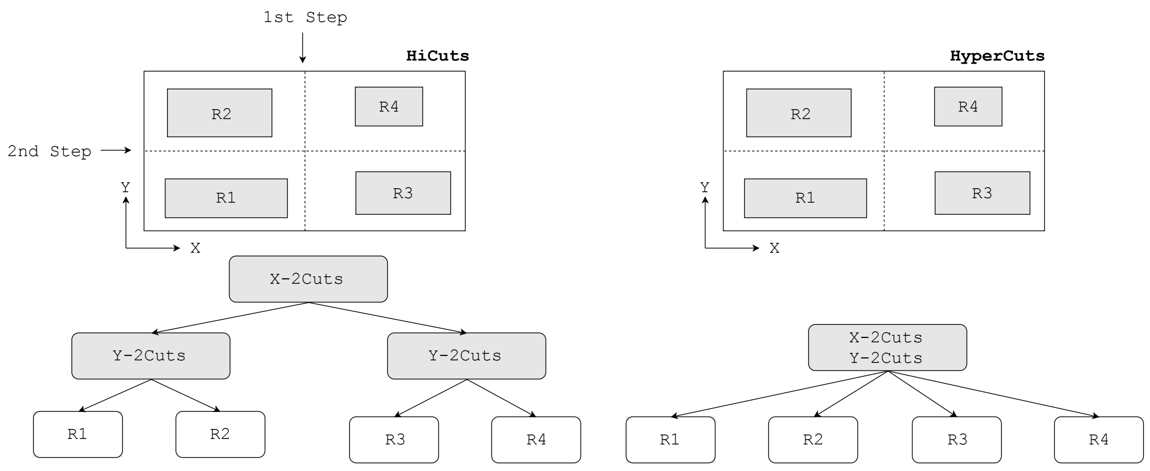

Figure 4: HiCuts vs. HyperCuts.

In Figure 4 from is presented an optimal example of the differences between HiCuts and HyperCuts: the height of the decision tree is reduced by applying HyperCuts which cuts the plane into four squares with one division.

HyperCuts overcomes some but not all critical problems of HiCuts: regarding redundancy, is it not possible to completely eliminate it when the repeated rules are not present in all the siblings.

1.4.3 EffiCuts

Because of its sub-optimal solution regarding redundancy, HyperCuts still incurs memory overhead with large test sets. For this reason, Efficuts represents the real state-of-the-art in packet classification. As explained in the paper *"EffiCuts: Optimizing Packet Classification for Memory and Throughput"* [7], EffiCuts is a feasible solution to the previously mentioned overhead of other algorithms. The new ideas used in this approach comprehend separable trees combined with selective tree merging. These tree-based algorithms are employed in order to face the variation in size of overlapping rules, while a particular type of cuts (equi-dense cuts) are used to face the change in the rule-space density. EffiCuts represent an improvement on HyperCuts, but still employs all of HiCuts and HyperCuts heuristics.

In this solution, redundancy is lower that the previous solutions, but is still present; one of the biggest advantages of GHSOM based solutions is the fact that rule redundancy is completely eliminated.

CHAPTER 2

THEORETICAL BACKGROUND

2.1 Internet Protocol version 4

As already stated in 1.1, the goal of Internet Protocol is to provide a location of a computer systems on networks; the last protocol released is the Internet Protocol version 4 (IPv4). Another result of the mentioned code is that the identification of multiple systems on the network is easier then before its application.

Each message in an IPv4 (or IPv4) packet consist of a *header* that controls information for addressing and routing, and a *payload* consisting of user data.

Source 32 bits	Destination 32 bits	Source Port 16 bits	Destination Port 16 bits	Mask	Protocol

Figure 5: Simplified IPv4 header format.

The *Mask* and *Protocol* fields have not been considered in this work and therefore the input space comprises of the first four fields only: source address and destination address, source port and destination port.

2.2 Self-Organized Map

The following sections of this chapter provide a necessary background of the tool (GHSOM) used for implementing the solution.

Self-organized Map, introduced by Kohonen [8], is a type of artificial neural network (ANN) that is trained using unsupervised training [1].

SOMs exploit dimensionality reduction in order to provide a low-dimensional, discretized representation of the space of the training inputs; this representation is called *map.* The elements which form the maps are called *units* or *nodes.* Patterns that are close in the original (higher dimension) space are also close in the reduced feature space (lower dimension). In general, a SOM's output space is typically one or two dimensional. One of the main feature fundamental for packet classification is the absence of redundancy in classification. This means that if a data has been classified in a certain node, it will not appear again in another unit.

The input-output relationship of the neural network can be expressed by the following equation:

$$y_i = x^{\mathrm{T}} * w_i \tag{2.1}$$

[1]Unsupervised training is based on a learning algorithm that finds (by itself) similarities between different training samples and classifies them accordingly (usually this classification is called clustering when talking about unsupervised learning). In certain contexts, the process is also referred to as self-organization. A training dataset is used to train the network, but there is no output information for the members of the input dataset.

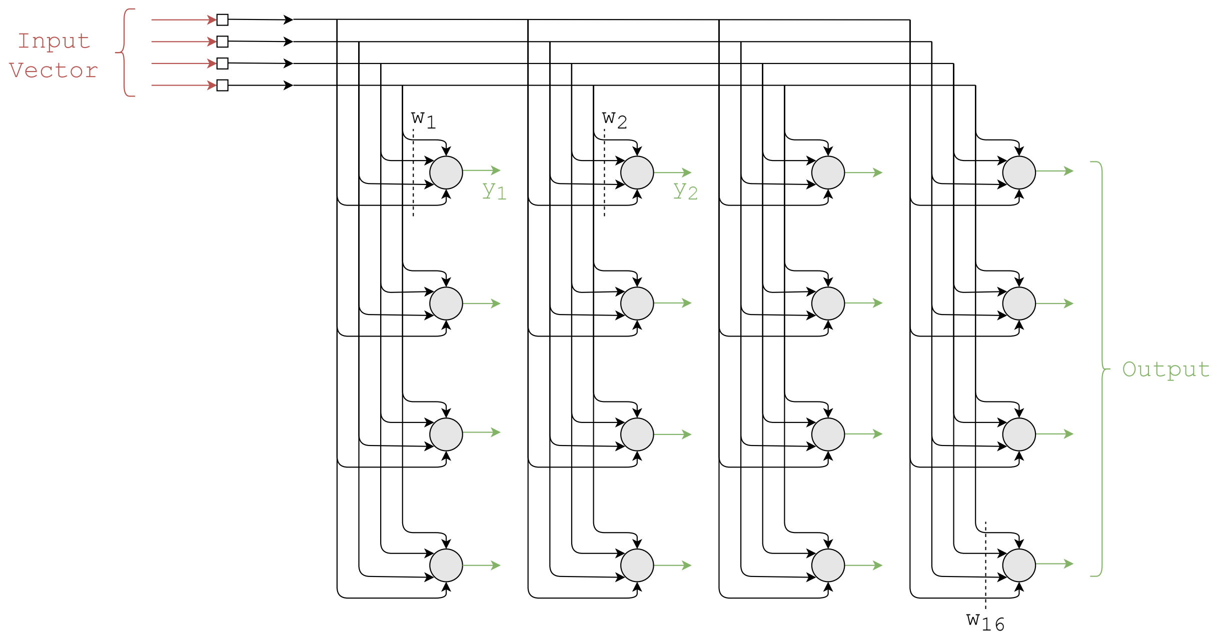

Figure 6: Two-dimensional lattice of neurons.

In the Figure 6 the input vector could be seen as $x = [x_1, x_2, x_3, x_4]^T$ and i indicates the number of the neuron unit. In the example shown above, a two-dimension SOM has been shown. Each unit has its own weights vector which will be used for determining the winning neuron during the calculation of the output given certain inputs. No activation function has been used in the SOM shown in the figure above or in the implemented SOM-based tree.

2.2.1 Training Self-Organized Map

As every neural network, the self-organized map needs to be trained against a group of training input in order to be able to work properly.

The Self-Organizing Map consists of a set of nodes i, which are arranged according to some topology: generally, the nodes are positioned to form a two-dimensional grid (Figure 6). A weight vector w_i of the same dimension as the input data is assigned to each of the node i, $w_i \in R^{\mathrm{d}}$. Generally, in the initial step the weights are randomly initialized. Training a self-organized map requires repeated learning iterations: t represents the current training iteration. During each learning step t, an input x_i is taken from the training dataset and presented to the map. The winning unit for the presented input is called $\mu(x)$ and is chosen by computing the highest activity level of units. Activity level depends on the Euclidean distance between the input pattern and units weight vector; thus, the winning neuron is represented by the node associated to the smallest Euclidean distance between its weight vector and the applied input vector.

The general formula for computing the distance at each iteration is:

$$\mu(x_i) = \underset{k}{argmin} \|x_i - w_k\| \tag{2.2}$$

Where k represents the number of neurons of the SOM.

The winning unit $\mu(x)$ is adapted in a way that, at future presentation of the same input pattern, it will exhibit an even higher activity level. The adaptation process is performed at every learning iteration with the goal to reduce the differences between input patterns and unit's weights. The update formula of weights is generally the following (applied to all neurons):

$$w_k = w_k + \eta * h_{k,\mu(x_i)}(x_i - w_k), \quad k = 1, ..., k \tag{2.3}$$

The other elements of the formula are η that is the learning parameters and $h_{k,\mu}$ that is the topological neighborhood function around the winning neuron μ. Typically the learning parameter is monotonically decreasing in the course of time and takes values between 0 and 1:

$$\eta(t) = \mu_0 * exp(\frac{-t}{\mu_1}) \tag{2.4}$$

for certain parameters μ_0 and μ_1. The topological neighborhood function $h_{k,l}$ is chosen to be a monotonically decreasing function of $d_{k,l}$ that represents the distance between the neuron k and the neuron l on the map grid.

The general effect of the learning rule is that it pulls all weights toward the input pattern x_i. However, the winning neuron is pulled the most due to the nature of the topological neighborhood function. The pulling effect is reduced for neurons that are further away from the winning neuron.

2.3 Growing Hierarchical Self-Organized Map

Classical SOMs presented in the previous section show two major limitations: they are static structures and therefore offer limited capabilities for the representation of hierarchical relations of the data. Also, since SOMs are unsupervised learning structures, no prior knowledge of the underlying data distribution means that there is no recourse in cases where the initial number of neurons is insufficient to adequately segment all the data. For addressing these two limitations, the *Growing Hierarchical Self-Organized Map* has been introduced. Packet classification applications require an high dynamism in modifying structures based on the input dataset. The GHSOM is a neural network with a hierarchical structure and is composed by independent growing SOMs. The GHSOM is characterized by the ability to grow both vertically and horizontally; it means that, based on the heuristic chosen for the training, the size of each map adapts itself to the characteristic of the input space.

Each map of the structure grows in size (depth and breadth) in order to represent a collection of data at a particular level of detail. The level of detail is chosen before the training, through heuristics parameters that will be more deeply discussed in section 2.3.2.

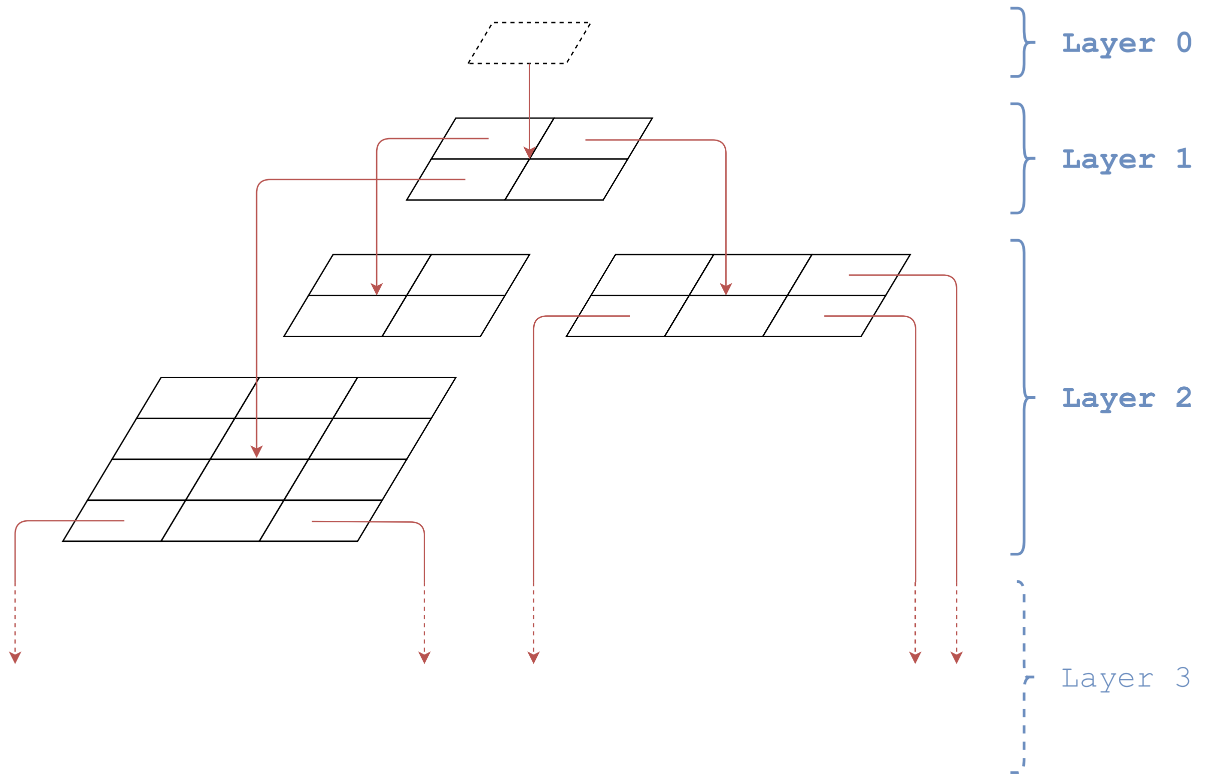

Figure 7: General GHSOM structure.

A graphical representation of a GHSOM is given in Figure 7. In this example the first layer comprises of a 2 by 2 map. Each unit of this first SOM contains a subset of input data and provides a rough organization of the main clusters in the input data. In other words, data points contained in the same unit are more similar to each other than data points in other nodes. The three maps in the second layer classify the data with more detail compared to the first map; they will be expanded as well, until the granularity level decided is reached. In the example considered, as the authors of the paper [9] confirm, one of the unit of the first unit has not been expanded into a map on a lower level because it has already reached the desired level of accuracy. The maps have different sizes based on the distribution of the input dataset; differently from the classical SOMs, the designer does not have to decide the size of the map a priori.

2.3.1 Growing Hierarchical Self-Organized Map mechanism

Before starting with the actual algorithm used for training GHSOM structure, it is important to define the mathematical tools used for this purpose. The basic principle of the GHSOM is to adapt its structure to the distribution of the input dataset. For doing this it is important that the growth of the map is correlated to the values of the *mean quantization errors* computed. Formally, the mqe_i of a node i is computed as the mean Euclidean distance of:

- the weight vector w_i of the unit i;
- the input vectors x_j that are components of the group of input vectors C_i mapped onto the aforementioned node.

n_c indicates the number input vectors. The general formula is the following one:

$$mqe_i = \frac{1}{n_c} \cdot \sum_{x_j \in C_i} \|w_i - x_j\|, \quad n_c \in C_i, C_i \neq \emptyset \tag{2.5}$$

The first step of the training begins with the calculation of the *mean quantization error* of the single unit at the layer zero. In the following equation, n_I is the number of all input vector x of the input dataset I:

$$mqe_0 = \frac{1}{n_c} \cdot \sum_{x_j \in I} \|w_0 - x_j\|, \quad n_I \in I \tag{2.6}$$

The value of mqe_0 measure the degree of overall dissimilarity of the input data. Afterwards, it will be discussed the reason why this value is a crucial parameter regarding the growth process of the artificial neural network.

2.3.1.1 Training and growth process of a map

Once we have the necessary parameters, it is possible to begin with the actual training of the structure. Beneath the layer 0, a new map is created. The size of this dimension depends on the heuristics that will be treated in 2.3.2. The map at the first layer is trained using a certain procedure (for example the one shown in 2.2.1). After a predefined number of epochs of iteration and input fed to the map, the *mean quantization error* of each unit is analyzed. At this point there are two possible scenarios:

- if *mqe* is high it means that for the selected unit the level of accuracy desired for the representation of the input space has not yet been reached.
- a low *mqe* shows that the level of granularity desired for the input space has been reached.

The quantitative value of high and low *mqe* will be discussed in the section dedicated to the choice of the heuristic parameters 2.3.2. In the first scenario, it is necessary to generate new units in order to better represent the input space. First of all it is necessary to select the two nodes among which a new row or column will be inserted: the *error unit e* is associated to the node which posses the highest *mean quantization error*, while the unit *d* represents its most unlike neighbor. After computing the two elements, a new column or row of nodes is inserted between *e* and *d*. The weights of the created units are initially set as the average of their corresponding neighbors. The insertion process is shown in Figure 8.

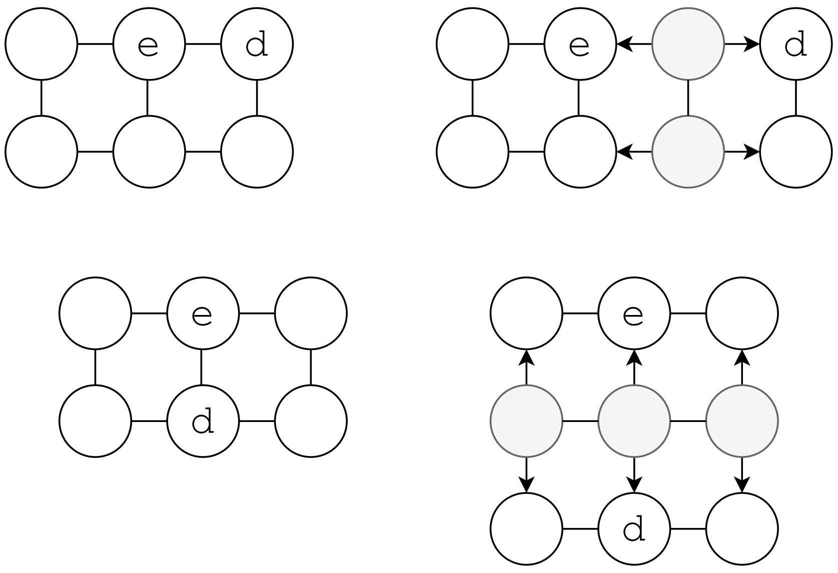

Figure 8: Insertions of units.

The inserted units are marked in gray.

Specifically, keeping in mind the assumption made previously it is possible to describe the growth process as presented. Consider C_i as defined for Equation 2.5, i.e. $C_i \in I$; let than w_i be the weight vector of unit i. At this point it is possible to calculating the *error unit e* as follows:

$$e = \underset{i}{argmax}\{\frac{1}{n_c} \cdot \sum_{x_j \in c_i} \|w_i - x_j\|\}, \quad n_C \in C_i \tag{2.7}$$

The distances between the weight vector of error unit and the weights vectors of the neighboring nodes is computed in order to find the most dissimilar neighbor d, the maximum value indicates the unit d.

The second scenario described in the analysis of the *mean quantization error* occurs one the value of *mqe* can be considered low. The enlargement process stops when the *map's mean quantization error*[1] reaches a certain value τ_1 of the mqe_u of the corresponding node in the level above the one considered.

$$MQE_m = \frac{1}{n_U} \cdot \sum_{i \in U} mqe_i, \quad n_U \in U \tag{2.8}$$

[1]The map's mean quantization error is commonly named *MQE* in capital letters.

The formula for computing the *MQE* [1] of a map is reported in Equation 2.8. As we can see, it is defined by the mean of all nodes' mqe_i of the subgroup U of the maps' nodes where data is mapped.

Specifically, *MQE* is used to delineate the stopping criterion of the growth process of a single map m:

$$MQE_m < \tau_1 \cdot mqe_u \tag{2.9}$$

It is clear from the previous formula that the smaller the parameter τ_1 is, the longer the training will last; longer training leads to larger map.

Once that the training of a map is finished, each unit is checked for further expansion. More formally, if a certain unit contains too diverse set of input a new map will be created in the next layer. The criterion for stopping the growth of depth of the structure depends on a second parameter τ_2; this value decides if the quality of the data representation is enough or it will be necessary to create a new map in the next layer. This is described in the following formula:

$$mqe_i < \tau_2 \cdot mqe_0 \tag{2.10}$$

It is notable that this second stopping principle is based on mqe_0 that represents the *mean quantization error* of layer 0, for each unit on all maps. The new map generated will be trained

[1] *map's mean quantization error* MQE is indicated in capital letters so as not to be confused with the standard *mean quantization error*.

using the input vectors mapped to the unit which has just been expanded. The process is iterated until all the lower leaves of the structure meet the criterion given in Equation 2.10.

2.3.2 Heuristics

As it has been pointed out several times, one of the main advantages of GHSOM structures is the possibility of a node to grow in depth independently from the other units. This feature allows the GHSOM to represent every dataset adapting the structure to the distribution of the input elements. The growth process - vertical and horizontal - mainly depends from two parameters:

- the parameters τ_1 controls the actual growth process of the GHSOM; thus, it controls the final size of each map.
- τ_2 controls the data representation granularity (the detail of all units. It means that each map has to posses a minimum level of quality of data representation. GHSOMs perfectly suit this feature providing the required number of units for ensuring that the representation detail level meets a minimum threshold for all parts of the input dataset.

All this discussion about SOMs and GHSOMs has been already treated by prof. Andreas Rauber, prof. Dieter Merkl and prof. Michael Dittenbach in their paper [9] and [10].

CHAPTER 3

OVERALL SYSTEM AND VALIDATION OF THE IDEA

Before starting with the description of the system, it is right to explain from where the basic of the MATLAB code came from. From the beginning, it has been decided to not use any pre-built neural network functions; this is because it was necessary to have complete control over all the parameters, training algorithms and functions for reaching the best performance offered from this type of implementation. Thus, the basic of the MATLAB code used for the GHSOM is the one developed by the Department of Software Technology of the Vienna University of Technology [11]. The code has been modified to make it suitable for packet classification applications.

The output grid size in the traditional SOM is fixed. Therefore, the traditional SOM does not adapt to the characteristics of input sample space and it is not suitable for problems characterized by frequent update of the training rule-set. Moreover, dataset in packet classification can present rules with different distributions in the space (dataset's characteristics are diverse, based on the type of packets 4.1) and, as already stated, the distribution may dramatically change over time due to rule updates in SDN applications. To handle rule updates, the Growing Hierarchical Self Organized Map has been adopted and implemented.

The goal of this thesis is to exploit learning architectures to solve in a clever way the packet classification problem presented in the first chapter. The core idea is to train a GHSOM on different rule-sets to find the best match rule for each input. A reinforcement learning style system has been used to find the best heuristics for building the GHSOM based on the characteristic of the dataset. Hence, the overall system shown in Figure 9, is made up of two blocks:

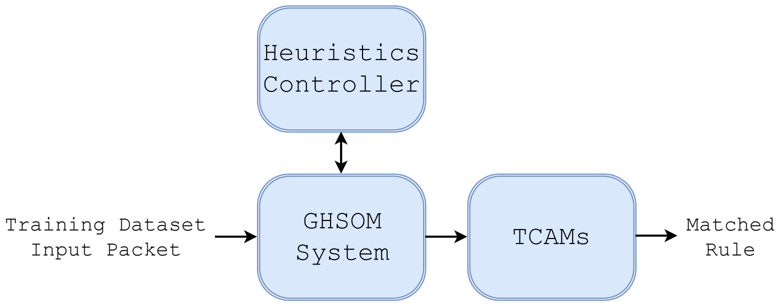

Figure 9: Overall system.

- The **GHSOM system** is the main block of the system and it is composed by the SOM-based structure.
- The **Heuristics Controller** which, through a reinforcement learning style network, select the best parameter for building the GHSOM.
- The **TCAMs** at the last leaves of the SOM-based trees are used to perform a brute force search on a handful of rules, in order to find the highest priority match. Furthermore, to increase the accuracy of the general system an additional memory has been implemented for storing the most critical rules (**additional TCAM**).

Before starting with the explanation of the solution, an aspect must be underlined. During the study, tree structure, leaves and bins will be mentioned several times and, from now on, when trees are mentioned we are referring to the SOM-based tree that represent the GHSOM system. The leaves of the tree are the children of a certain parent node and each of them is composed by a Growing-SOM (GSOM). The bins indicate the last leaves of the trees; at a hardware level, the bins are mapped to TCAMs, each of which contain a small number of rules. This is used to perform a brute force search for finding the highest priority match for the incoming packets within rules for a leaf nodes. As already mentioned in 1, TCAM is a specialized type of high-speed memory that is able to search the entire contents in a single clock cycle. TCAM provides the addresses where the requested data can be found.

The core idea of this study is to create a hybrid system that exploits the high performances of TCAMs, decreasing their drawbacks which are mainly due to bad scalability in throughput and to high power consumption as the number of rules increase. The structure presented in the following sections is composed of two separate levels: the higher level is implemented by means of GHSOM and aims to classify the input space based on the distribution of the input parameters, while the lower level is employed in the search of the highest priority matching rule and it is implemented by TCAMs which perform a brute force search against small subsets of rules (decreasing in this way the drawbacks of the mentioned memories).

3.1 Overall structure

As already stated in 2.2, a SOM produces a low-dimensional (usually 1-D or 2-D), discretized representation of the space of the training inputs. The first step for building a system able to

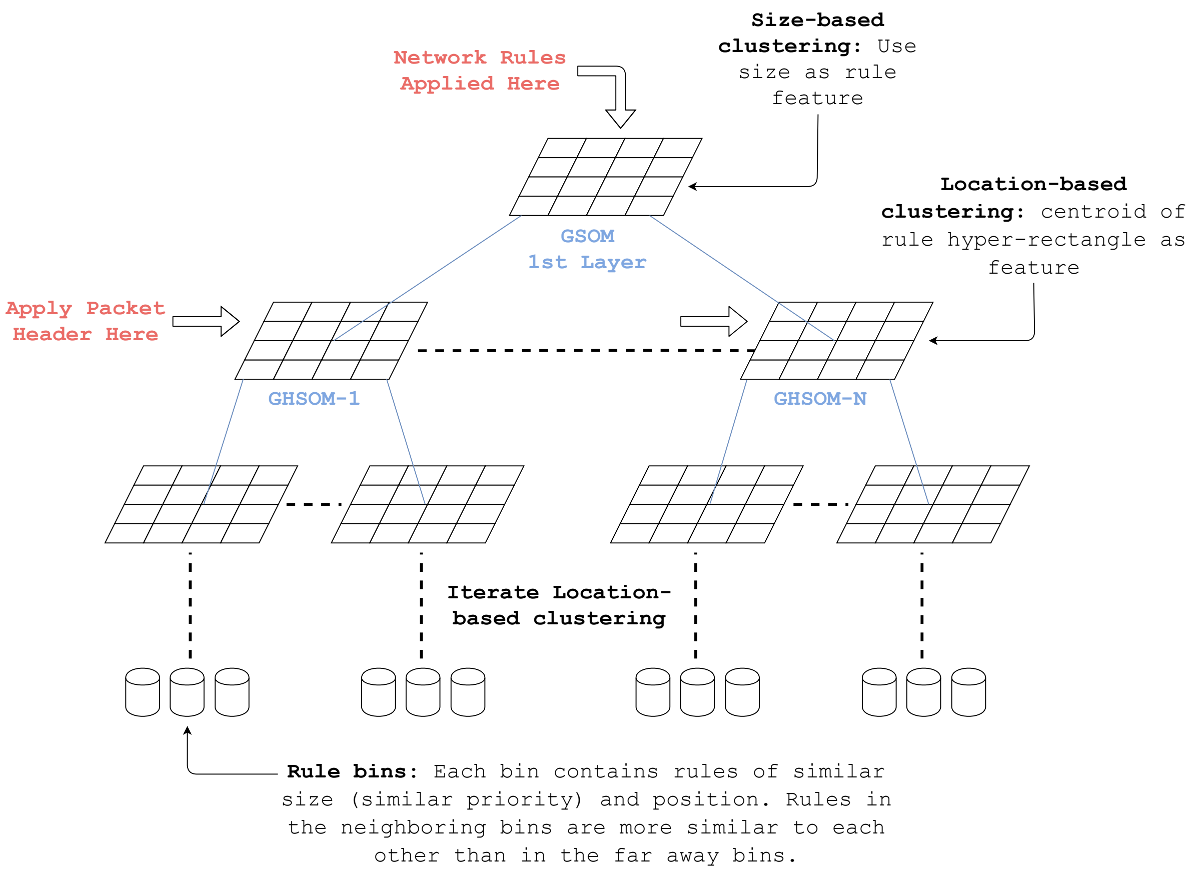
Network Rules Applied Here
Size-based clustering: Use size as rule feature
GSOM 1st Layer
Location-based clustering: centroid of rule hyper-rectangle as feature
Apply Packet Header Here
GHSOM-1
GHSOM-N
Iterate Location-based clustering
Rule bins: Each bin contains rules of similar size (similar priority) and position. Rules in the neighboring bins are more similar to each other than in the far away bins.

Figure 10: SOM-based tree used for Packet Classification.

perform a correct packet classification is to choose the input space wisely. The idea followed in this work is to use a first layer of GHSOM that can only grow in width and whose feature map depends on the log_{10} scaled value of two parameters:

- **Volume of the hyper-cube**: the volume of the four dimensional hyper-cube has been computed as the multiplication of the four edges of the figure. In other words, the vertices of the hyper-cubes are determined by the rule ranges across various header's fields and these values have been used as edges of the figure.

- **Hyper-cube surfaces area**: these areas are computed multiplying the permutations of the different edges of the hyper-cube.

The first layer classifies the whole set of rules in multiple subsets based on the parameters explained in the list above. For each subset, the locations of the hyper-cubes are computed. These locations are represented by the centroids of the hyper-cubes and are used as input parameters for the second layer of GHSOMs. The locations of the rules in the space are therefore used to develop finer clusters of the dataset. Thus, centroid-based clustering is iterated from the second layer to the end of the tree. Moreover, it is important to remember that the second layer of GHSOMs can grow both in width and in depth (unlike the first level of the tree); the growth depends on the training parameters τ_1 and τ_2.

During classification, the input packet headers (which geometrically represent a point in the 4D space) are applied from the second layer of the SOM-based tree. This because these input packet headers are only characterized by their position in the rule space (they do not have a volume or surfaces) and are composed of four fields. Another reason is related to the intrinsic characteristic of SOMs: the input dimension of the training samples has to be the same as the input dimension of the test inputs.

The process develops in the following steps:

- during the training phase, the feature map of the second layer of the GHSOM is composed of four points which represent the centroids (location) of the hyper-cube of each rule.

- during the test phase, the four fields of the feature map are composed just by the packet's position in each dimension of the input space as determined from the packet header.

At the lower leaves of the tree, rules are associated with an edge bin. The network' rules in a certain bin match each other with respect to their shape and position. Moreover, SOM-based rule clustering is also ordered from bin-to-bin regarding rule similarity; it means that rules in the adjacent bins match each other more (in priority and header fields) than in the bins farther away.

3.1.1 Additional TCAM

This part of the project will be further clarified in the 4 section, but it is worth mentioning while presenting the overall structure. During the training of the GHSOM where the best heuristics are chosen (the process is explained in 3.7.1, the map is tested with a subset of incoming packets randomly generated by Classbench 4.1. During the training phase is possible to find the most critical rules; in other words, it is possible to detect in the dataset the rules that most likely will generate mismatches or will not be found during the actual use phase. These rules are stored in a separate TCAM (named *additional TCAM*). Experimental results show an increase in accuracy of the system with the addition of this memory. The results of this assumption are shown in Section 4.

3.2 Functioning

As mentioned before, during the training phase the generated structures are tested against a bunch of packet headers randomly generated by Classbench tool 4.1. The number of rules tested in the simulations carried out is equal to 10% of the space of the dataset. This process helps to find the most critical rules present in the dataset and therefore increase the accuracy of the structure. The pseudo-code of this process is the following:

```
end_training = 10 % of the dimension of the dataset;
for i = 1:end_training
    chose random input(i);
    compute matched rule with GHSOM system;
    compute matched rule with accurate system (slower);
    if comparison == 1 % the matched rules are the same;
        do nothing % correct match;
    else
        add correct matched rule to the additional TCAM;
    end
end
```

The results in terms of performances and memory accesses will be presented in 4.5. Regarding the packet classification phase, a network packet is correlated across hierarchical SOM nodes in the GHSOM-based tree and eventually mapped to a bin at the leaf of the tree.

In the training phase, the size of the GSOM at the first layer is variable and its size depends on the heuristics presented in 2.3.2 and on the distribution of the input parameters. Each unit of the map is assigned a certain subset of rules which location represents the feature map for the second layer of GHSOMs. On the other hand, during training is possible that some of the units are left empty or with a number of rules not sufficient for creating a second layer map;

in these scenarios there are two possible solutions: if the number of rules is insufficient for creating a map obtaining some improvements, the subset of rules is directly stored in a TCAM. In the other case, if the node created has no rules stored into it, a TCAM is left empty. This operation allows to have free space available in view of a possible dataset's update. In both cases, as soon as the number of rules exceeds a certain threshold [1], it is possible to re-train the selected GHSOM without changing the rest of the structure.

3.3 Initialization of Self-Organized Map

There exist different ways for initializing a Self-Organizing Map. Generally a random initialization of the weights is used, but, in this study, a linear initialization has been chosen; this is because for the implemented training algorithm the linear initialization produces better results [2]. In other words, the computation of the SOM can be made orders of magnitude faster since a SOM is than already approximately organized in the beginning.

First of all a map structure is created; then, the eigenvalues and the eigenvectors of the training data are calculated. The weights of the map are initialized along the dimension of the map grid characterized by the greatest eigenvectors of the training data.

[1]The threshold depends on the number of rules that the designer decide to store in a single TCAM, is an hardware constriction.

[2]It is possible to start the training with a narrower neighborhood function and smaller learning-rate factor [8].

3.4 From header to binary

In the IPv4, the header of each information packet is composed by six fields, as already shown in 2.1. For understanding the mentioned fields, we can consider the following rule:

```
@147.12.227.34/32  76.188.171.59/32  0:65535  1717:1717  0x06/0xFF  0x0000/0x0200
```

The example above is used for explaining the different fields which are source address, destination address, source port, destination port, protocol and mask.

Geometrically is possible to consider each field as a range of valid value in a specific dimension; in the presented work, only the first four field have been considered and this means that each rule is represented in the space by a four dimensions hyper-cube.

1. **@147.12.227.34/32** the first field identifies the **source address** of the rule. First of all it is necessary to translate the number in binary: the source is represented on 32 bits, 8 bits for each section of the field.

TABLE I: FROM DECIMAL TO BINARY

Decimal	Binary
147	10010011
12	00001100
227	11100011
34	00100010

In this example, the source in binary on 32 bits is 10010011000011001110001100100010 and represents the concatenation of the four sections of the number. \32 is the number of bit to consider. In this case all the bits are counted and therefore the range in the first dimension is only a point. There are other cases in which this number is not 32: for a better comprehension consider now the same source address, but instead of 32 bits, only 27 are considered (this is just an example, on the datasets in A is possible to see a handful of rules from a dataset): @147.12.227.34/27. In this case the range of values considered in binary is 100100110000110011100011001XXXXX in which only the first 27 bits are important. Here, the range of value goes from 10010011000011001110001100100000 to 10010011000011001110001100111111.

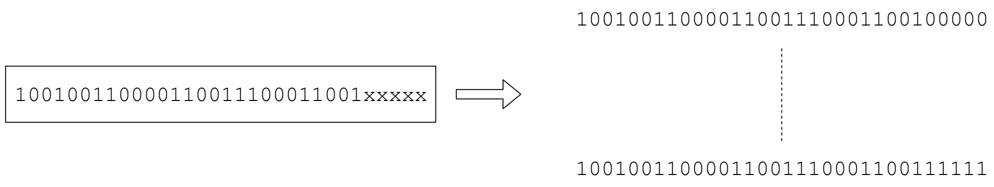

Figure 11: Translation of the range.

Geometrically this range represents an edge in a certain dimension.

2. **76.188.171.59/32** is the the **destination address**. The same consideration carried out for the source address can be done for this field.

3. **0 : 65535** the third field of the rule is the **source port**. The range in binary is expressed on 16 bits instead of 32 and is easier to understand the value considered for the edge from the geometrical point of view: the first number represents the beginning, while the second is the end of the range.
4. **1717 : 1717** same consideration of the third field can be used for the fourth which represents the **destination port**.
5. **0x06/0xFF** and **0x0000/0x0200** are the **mask** and the **protocol** of the rule and they have not been considered for defining the rule space.

3.5 Output

Each memory selected by the algorithm of the GHSOM perform a brute force search on the proper subset of rules. Thus, the selected TCAMs provide the matched rules in output (if they exist). The additional memory will perform the search as well. Geometrically, all the matched rules represent the hyper-cubes which contain the input point. The last step is to compare the rules and find the one with the highest priority: the selected rule will represent the correct match and therefore be the output of the system.

3.5.1 Priority Checker

The only goal of this unit is to compare the possible rule matched by the multiple TCAMs and define the one with the highest priority. Generally the priority is defined based on the dimension of the rule (in this study, the volume of the hyper-cube is considered).

In Figure 12 is given a graphical representation of the route of the incoming packet from the input to the output. Each blue and green block represented in the figure has an address

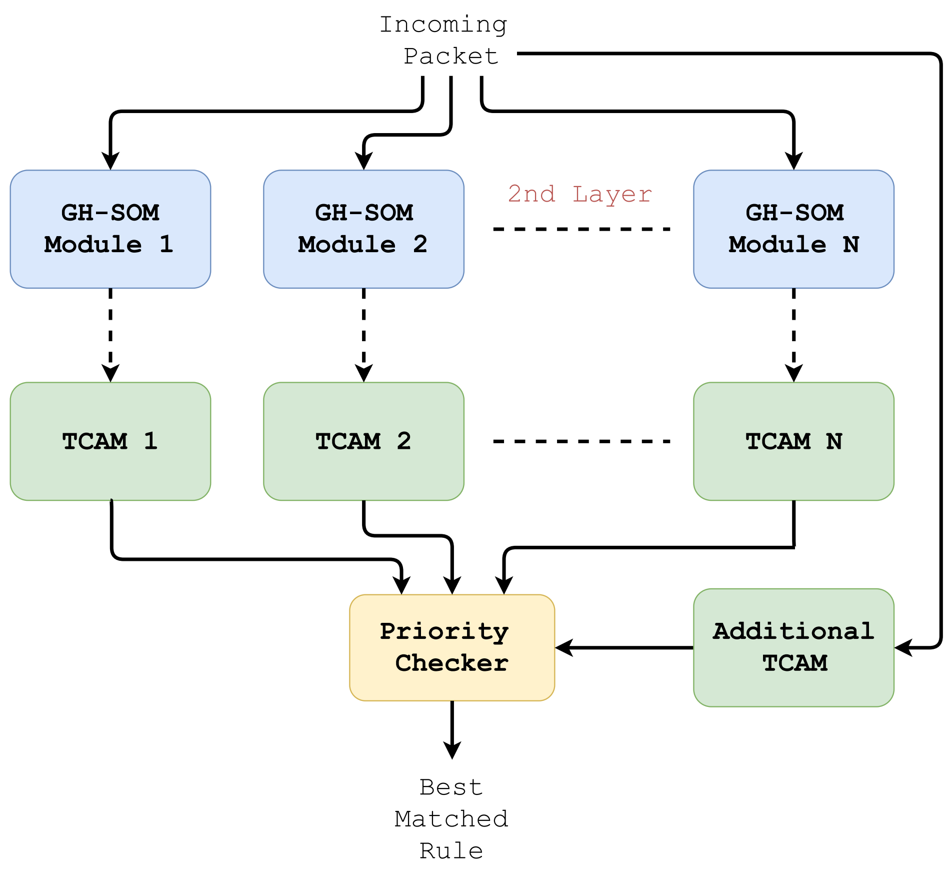

Figure 12: From input to output.

register that is helpful for the actual implementation of the GHSOM tree. In every block we compute the address of the next GHSOM until eventually reach an associated leaf level TCAM.

3.6 Updates management

One of the main feature of SDN applications is continuous rule updates. The SOM-based structure offers a partial solution to this problem: once the GHSOM has been trained with the initial dataset, the weights are decided and it is possible to use them even if it is necessary to update or remove new rules by simply removing the rules from one TCAM and re-classify them by running them through the structure and adding them to the newly matched leaf level TCAM. In other words, the proposed solution does not require to be fully re-trained when small updates are performed on the dataset.

3.6.1 Add rule function

When the user decides to add to the GHSOM structure new rules, it is necessary to find the correct bin for each of them. The process regarding the first layer is analyzed at first.

```
% Find input parameters:
Compute volume of the hyper−cube;
Compute surface areas of the hyper−cube;
Update dataset memory and labels adding new rules; % Updates
Find Best Match Unit 1st layer % Test the 1st layer GSOM
    input: log10(volume), log10(areas);
Add Rule to the found unit and update tree's labels;
```

The first layer is tested with a new rule and the GSOM gives back the most similar node in the map. Therefore, the rule is added to the corresponding unit. At hardware level the memory corresponding to the selected unit will be updated storing the new rule.

The weights of the GSOM do not need to be re-computed. Thus, we only need to select the best match unit and add the rule to the selected subset. As we know, these subsets are used for training the second layer's GHSOMs; in the following pseudo-code it is possible to see how to handle rule updates for the GHSOMs of the second layer of the tree.

```
Compute centroid of the hyper-cube; % Find input parameters;
Find Best Match Unit of the 2nd layer GHSOM; % Test the GHSOMs selected by the 1
    st layer;
    input: centroid;
Add rules to the each unit of the selected tree;
Add Rule to the found bin and update tree's labels;
```

These two functions are then combined and used as shown below:

```
for i=1:number_input
    Find Best Match Unit and update memories adding new rule;
    Add rules to the mapped tree to matched bins;
end
```

3.6.2 Delete rule function

For deleting a group of rules the actions taken on the system are similar to the ones performed for the adding new rules. The inputs of the deleting function are the GHSOM structure, the memories and the group of rules to delete; the output will be the exact same GHSOM (the weights are not modified) with updated rules stored in the memories.

The pseudo-code is the following:

```
for i=1:number_input
    Update dataset's memory and labels deleting selected rules;
    Search the rule in the 1st layer GHSOM;
    Delete the rule in the unit selected by the search before
        Delete rule from subset memory;
    Find the path from the 2nd layer of the tree to the bins;
        Delete rule from leaves of the tree and TCAM;
end
```

3.6.3 Retraining function

One of the function added to our implementation is the ability to retrain the network considering the changes made on the dataset. This additional function can be useful if, after several updates of the dataset, the GHSOM becomes imbalanced. The first consequence of the disbalance of the structure is the increase in the computation time. This effect can be generated from adding a large group of rules in the same leaf of the tree: the dimension of the TCAM for the mentioned node would explode, loosing the benefits obtained splitting the dataset.

3.7 How to chose the heuristics

As already stated in 2.3.2 section, in the SOM-based tree, each parent unit can expand both horizontally and vertically depending on two controlling parameters τ_1 and τ_2 as well as the mean quantization error (MQE) of input data [9]. The expansion of the SOM can affect the accuracy on a certain dataset as well as the number of rules contained in the lower leaves of the tree; for this reason, is important to find the best parameters based on the provided dataset. The number of rules contained in the leaf bins affects the cost of the algorithm: the TCAMs perform a brute force search in each bin in order to find the matching rules and, obviously, a larger set of rules means an higher cost per research. After understanding the importance of an accurate regulation of the parameters τ_1 and τ_2, two steps were undertaken to find the best solution. In the first place, a trial and error method has been performed: through this technique it was possible to understand the general trend for reaching the highest possible accuracy for each dataset. In the second step, a reinforcement fashion learning has been implemented for finding the best heuristics to avoid having to search manually the best training parameters.

3.7.1 Reinforcement Learning style algorithm

In this section is presented an innovative approach which goal is to increase the accuracy of the GHSOM in packet classification applications and build the best tree for a provided dataset. The idea is to perform multiple training iterations of the complete structure modifying the heuristics according to the trend of the accuracy test performed. The general idea of this approach has been taken from the classical reinforcement learning method introduced below. It is important to underline that the approach used is not the standard reinforcement learning, but a method whose basic idea is based on mentioned learning.

Figure 13 is a graphical representation of a reinforcement learning (RL) scheme. Briefly, an RL system consists of an **agent** that repeatedly interacts with an **environment**. The agent observes the state of the environment and consequently take an **action** that could change the environment's state.

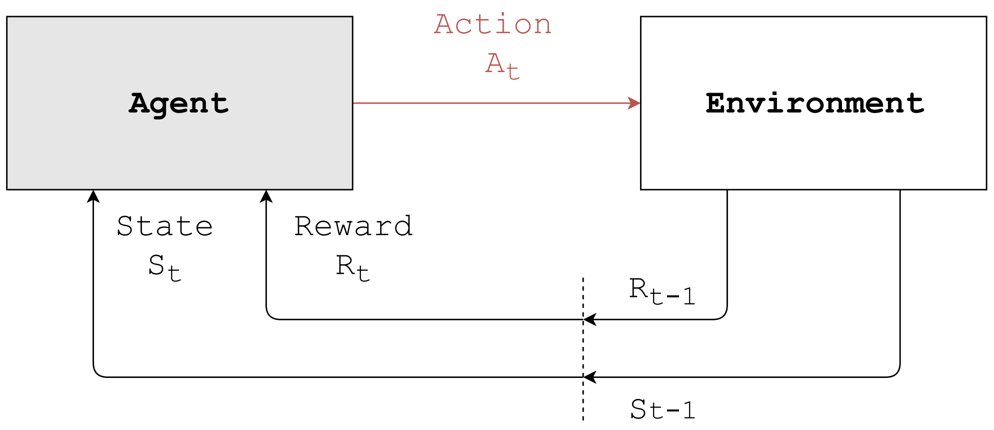

Figure 13: Classic RL system.

It is possible to see the same key elements of the reinforcement learning in our approach: the **agent** consists in the controller that choose the heuristics (**action**) which are responsible for the construction of the GHSOM (**environment**). After the modification of the training parameters, the structure is retrained changing consequently its state. The **state** can be seen as the accuracy value generated by the current setup; once the GHSOM is created, it is tested using a small set of packet headers generated by the Classbench tool. Thus, it is possible to obtain an estimation of the accuracy of the system; in parallel, the most critical rules of the dataset are identified and stored in the additional TCAM as explained in 3.1.1. The **reward** parameter come out from the analysis of the present and past accuracies computed; when the accuracy grows at new iterations it means that the parameter are updated with positive results on the environment. For this reason it is necessary to store the value of the accuracy at every epoch of training.

Let t represents the number of the current epoch: if accuracy in t is higher than accuracy in t-1, it means that the action that was performed had a positive result, the reward is positive and therefore the heuristics will be updated following the same trend.

A graphical representation of the approach is shown in Figure 14. It noticeable that the general structure of the classical reinforcement learning system has been maintained.

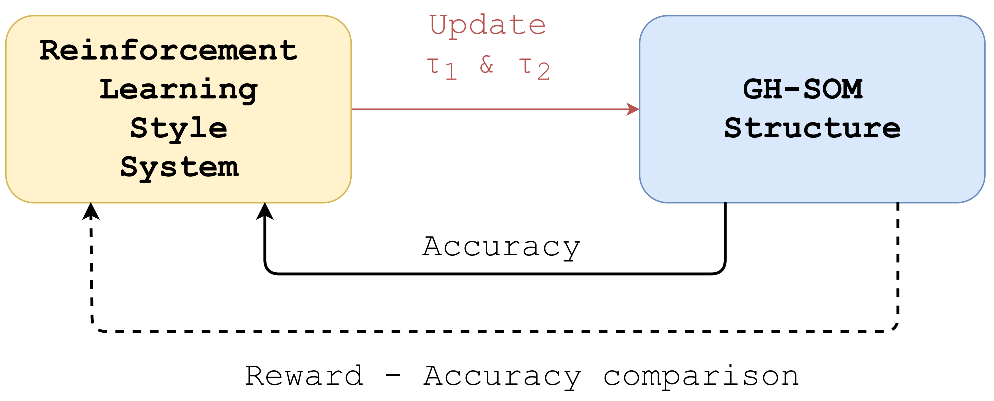
Reinforcement Learning Style System
Update
τ_1 & τ_2
GH-SOM Structure
Accuracy
Reward - Accuracy comparison

Figure 14: RL style in GHSOM system.

3.7.1.1 Early stopping

To avoid the situation in which the accuracy between t-1 and t decrease, a technique called **early stopping** has been used. This approach is wildly used in machine learning to avoid overfitting when training a network with an iterative method (e.g. as gradient descent). In this work, the test of the structure continues until there are three consecutive epochs in which the accuracy drops or until the accuracy remains the same for five consecutive iterations. In the first scenario, the heuristics where the accuracy where maximum are restored and the SOM-based tree is retrained for the last time and ready for the actual usage; in the second case, the heuristics are restored to the epoch where the accuracy where the maximum value before the five equal accuracies in row.

The pseudo code for the reinforcement training style algorithm is presented in the next page.

```
Set default heuristics parameters:
    breadth1stSOM(0),
    depth1stSOM ALWAYS = 1,
    breadth2ndSOM(0),
    depth2ndSOM(0);
flag_training = 0;
t = 0;
while flag_training == 0
    Train the network with the chosen heuristics;
    Test the network with corners of hyper-cubes (worst case);
    Compute accuracy(t);
    if t >= 3
        if accuracy(t)<accuracy(t-1)<accuracy(t-2)
            Stop Training: flag_training = 1;
            Restore heuristics where max(accuracy);
            Re-train network;
        end
    end
    if t >= 5
        if accuracy(t)=accuracy(t-1)=accuracy(t-2)=...
        ...=accuracy(t-3)=accuracy(t-4)
            Stop Training: flag_training = 1;
            Restore heuristics where max(accuracy);
            Re-train network;
        end
    end
    t = t + 1;
    Update Heuristics;
end
```

CHAPTER 4

RESULTS AND COMPARISONS

The first purpose of the testing process is to debug the system and bring it to its correct functioning. Several simulations have been carried out on different feature maps before finding the best suitable solution; the comparison between these simulations is presented in the following sections. After finding the best option for the specific application and verifying the correct working principle of the system, we tested its accuracy.

In order to test performance of the system on real rules it was necessary to generate datasets similar to the ones used in real life application. In this thesis, the standard benchmark Classbench [12] has been used to generate rules for different usage categories with different characteristics and sizes.

4.1 Classbench & dataset format

Classbench is a tool that helps generate rules and packet traces for different usage patterns. It is also helpful to use its output for benchmarking packet classification devices and algorithms. The *Filter Set Generator* included in Classbench produces rule-sets that are similar to the ones used in real applications. A *Trace generator* is used for producing the incoming packet applied to the system for testing it.

TABLE II: CLASSBENCH SEEDS

ACL1_seed ACLl2_seed ACL3_seed ACL4_seed ACL5_seed	Access Control List (ACL)
FW1_seed FW2_seed FW3_seed FW4_seed FW5_seed	Firewall (FW)
IPC1_seed IPC2_seed	IP Chain

From the Classbench tool is possible to generate datasets in different formats Table II. The differences between the formats are mainly due to the distribution of the data in the rule space and various ranges in the four analyzed dimensions.

From the comparison reported in the following section, it can be shown that the algorithm shows different accuracy on different datasets leading to variation in accuracy across the datasets.

4.2 Testing Growing Hierarchical Self-Organize Map

Once trained, the network is tested against a new set of packet headers, created by means of the Classbenchtool as well. These simulated sets of packet headers are as similar as possible to real life applications based on the previously generated rule sets, in order to present realistic results.

The pseudo-code for testing the network is the following one:

```
for i = 1:number of inputs
    For each 1st layer GHSOM structure, compute the address bin;
    Search the rule in selected TCAM + Additional TCAM;
    if rule matched != 0
        compare results for finding highest priority rule;
    else if no rule matched
        compute rule with accurate method (slower);
        update additional TCAM;
    end
    output highest priority rule matched;
end
```

The incoming packet is applied at the second level of the network and eventually the output is provided. A 100% accurate application is used to evaluate the number of errors and misclassifications when testing the system for accuracy. In fact, the proper classification of the input does not mean everything. In this study, a brute force search has been used as 100% accurate method. The output of the comparison between the two implementation can lead to three different scenarios:

- *Correct Rule Found* the optimal classification is provided in output;
- *Rule Not Found* this situation is tolerated, since it is possible to immediately perform the search with the back-up system presented in 5.2.1 (i.e. Efficuts). In this scenario the

loss in performance is equal to the time necessary for searching the rule with the back-up system.

- *Rule Mismatched* this is the worst scenario and it has to be avoided. The GHSOM system is not able to recognize this error and it is necessary a feedback signal from the external routers to report these problems. The additional TCAM is useful for decreasing this type of error during run time. The back-up system is used to solve mismatches of rules.

4.3 Choosing the feature map

As already mentioned, from the beginning of the project the idea was to implement the first layer of the structure for classifying the rules based on their hyper-cubes' geometrical features and the second layer for the classification based on the position. The main problem was finding a proper feature map which was suitable for classification by exploiting GHSOMs. The feature map under analysis is the one related to the first layer's Growing-SOM which is employed for classifying the rules based on their size. From the simulations performed, it has been proven that the accuracy of the structure depends significantly on how rules are classified by the first layer GSOM.

The choice of the feature map used for the classification of the second layer is quite straightforward (it is only necessary to compute the centroids of the hyper-cubes), on the other hand the problematic decisions has been made on the first feature map because multiple choices were possible. It was clearly necessary to include the volume as one of the parameters for the classification because, in this study, it represents the priority of the rule. Even though volume represented a good starting point for the feature map, it was not enough for a correct classification. Therefore, the area of the surfaces of the hyper-cubes has been considered as well. After these decisions, it was possible to define the complete feature map of the first GSOM as composed by the volume and the six areas of the surfaces of the hyper-cubes.

In the final realization of the system, the logarithm function has been applied to the first layer input parameters (volume and area of the surfaces of the hyper-cubes) in order to ensure that the considered input space was suitable for classification; it was necessary to apply the $\log_{10}$ in order to be able to classify the rule-sets.

The histogram plots reported in Figure 15 show that the analyzed datasets have a small number of rules with a volume not comparable with the other rules. Hence, for classifying this parameter was necessary to apply a function for getting a better distribution of the values.

Everything explained for the volume has to be applied to all the other parameters of the feature map.

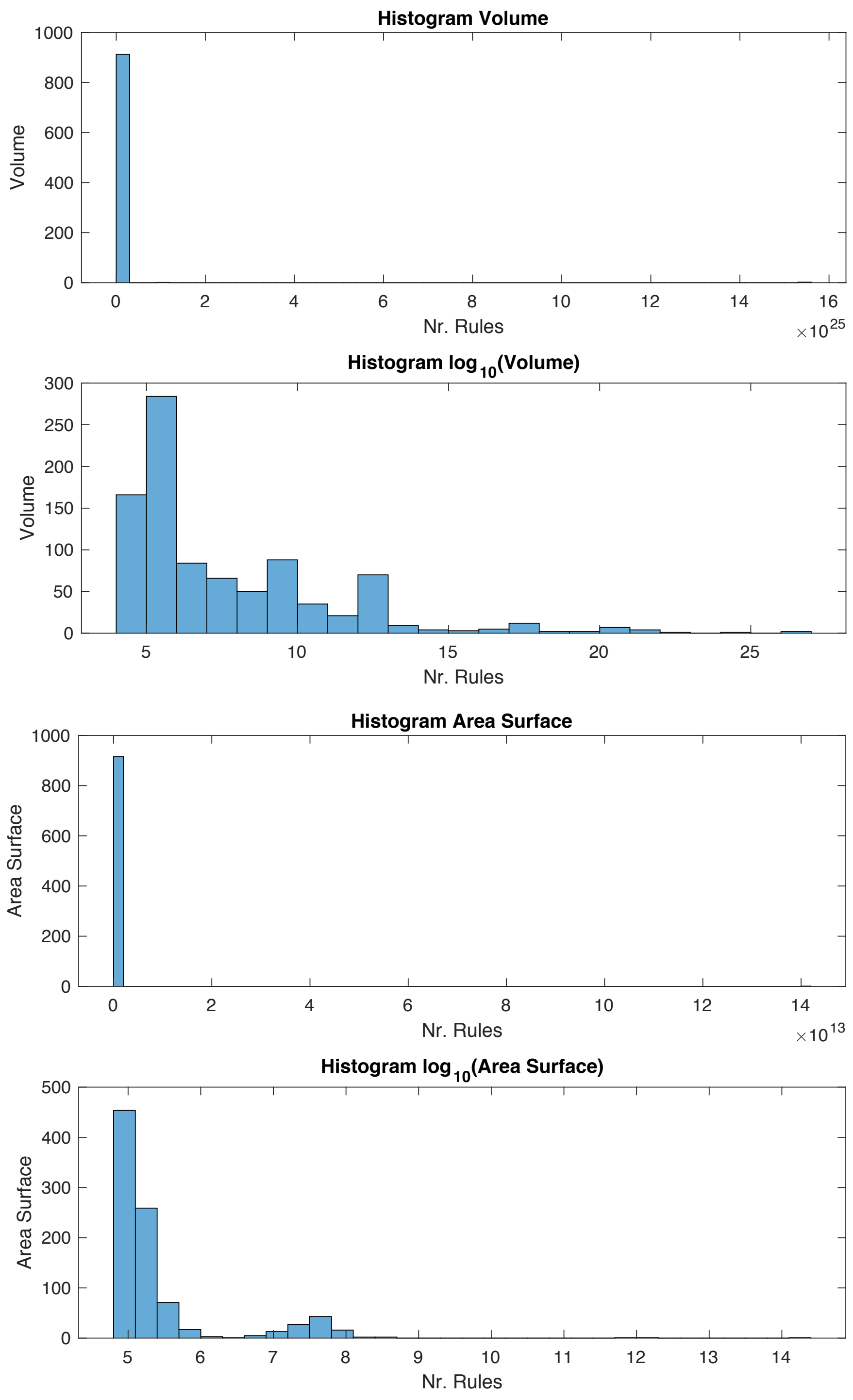

Figure 15: Distributions of input parameters of the first layer of the tree.

4.4 Accuracy test

The accuracy tests have been used to evaluate the number of misclassification and rules not found in the system. This value was useful to quantify the quality of the system.

In order to compute the accuracy, a 100% accurate system has been implemented in MATLAB in order to ensure the correctness of the analysis. The output of the GHSOM structure and the brute force search are compared for assuring that the values are the same.

The table and the graph below show the performances obtained by the system implemented:

TABLE III: ACCURACY TEST PERFORMED ON DIFFERENT DATASETS

dataset	**1000**	**5000**	**10000**
ACL1	100	100	99.8
ACL2	100	96.6	94.2
ACL3	100	95.4	95.9
ACL4	98.5	97.5	94
ACL5	100	100	100
FW1	100	93.3	93.6
FW2	94.3	95.2	92.9
FW3	97.3	95.4	96.9
FW4	99.65	98.5	93
FW5	98.75	97.3	96.5
IPC1	98.5	97.4	98.5
IPC2	98	97.4	94

Figure 16 presents packet classification accuracy for 36 different rule-sets created by ClassBench. The largest rule-set contains around 10000 rules, and each rule-set is tested against

2000 packets. High accuracy is observed across all the tested rule-sets and that means that the system will have to face a performances loss only in a low percentage of executions (second and third scenario 4.2).

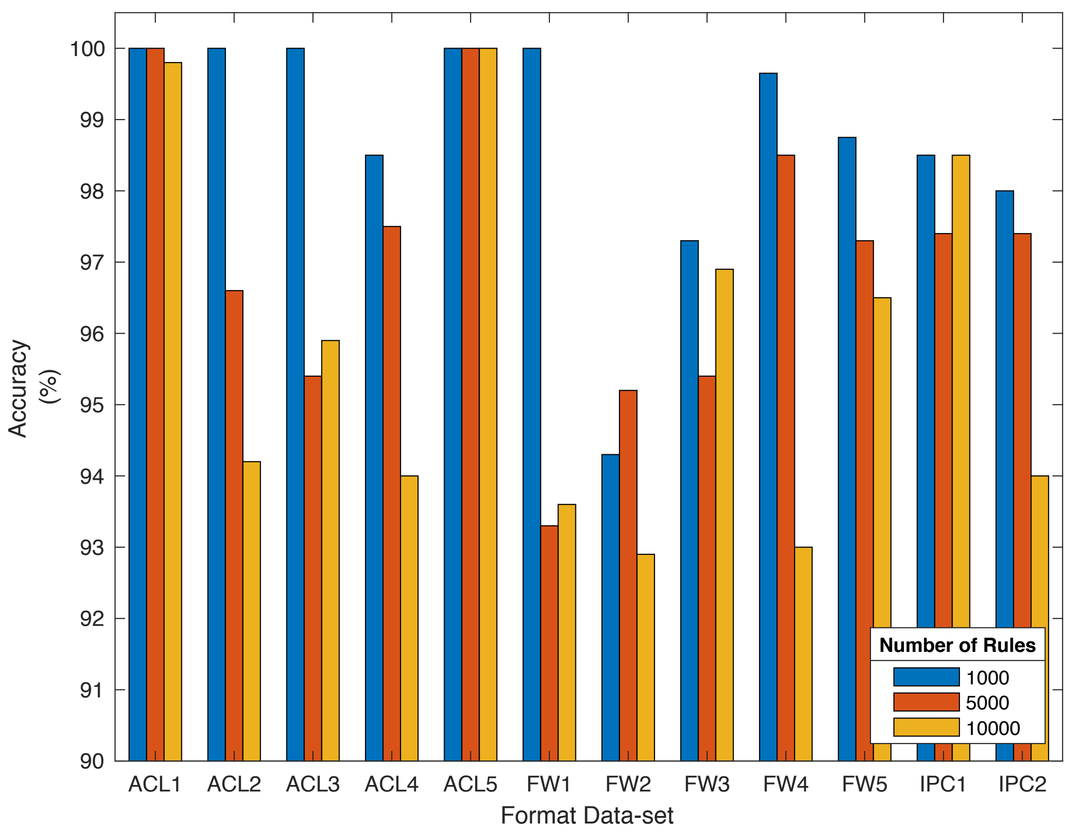

Figure 16: Accuracy with different datasets.

The accuracy results presented above do not consider the performance boost due to the back-up system that computes the matching rules when the incoming packet cannot be found using the GHSOM-based system. The accuracy has been re-calculated with the back-up structure and the success rate has increased to almost at 100% for every format. Only the worst cases will be shown in this table:

TABLE IV: ACCURACY TEST INCLUDING BACK-UP SYSTEM (SLOWER)

dataset	**1000**	**5000**	**10000**
ACL4	100	100	99.9
FW1	100	100	99.85
FW2	99.9	100	98.7
FW4	100	100	99.6
IPC2	100	100	99.8

If a dataset is not in Table IV, it means that it has reached 100% accuracy. The small percentages of error reported for the other formats are due to misclassifications that the system alone was not able to detect.

In certain situation is not tolerable to mismatch any rule so, for this reason, a solution has been presented in the chapter related to the hardware design. The system adopted for protecting the SOM-based tree is based on a feedback coming directly from the routers that indicates the correctness of the received packet; further analysis have been provided in 5.2.

The pseudo-code reported below is the implementation of the brute force method for packet classification. The same code has been exploited for simulating the TCAM search:

```
min_volume = infinite;
rule_found = 0;
for i=1:n_rules in the memory
    for i=1:4
        if input(i) is inside the range of the dimension(i)
            go on;
        else
            input not inside the rule;
            break
        end
    end
    if input is contained by the hyper-cube (rule)
        rule_found = 1;
        compute volume hyper-cube;
        if volume_hyper-cube < min_volume
            min_volume = volume_hyper-cube;
        end
    end
end
```

The output of the presented algorithm is the matching hyper-cube with the lowest volume and therefore it represents the highest priority matching rule.

4.4.1 Comparison between different dataset formats - Accuracy

In 4.1 the differences between the dataset formats have been mentioned. From the plot below, it is possible to see the trend of each format with increasing number of rules: *ACL1*, *FW1* and *IPC1* format have been selected as samples for this analysis:

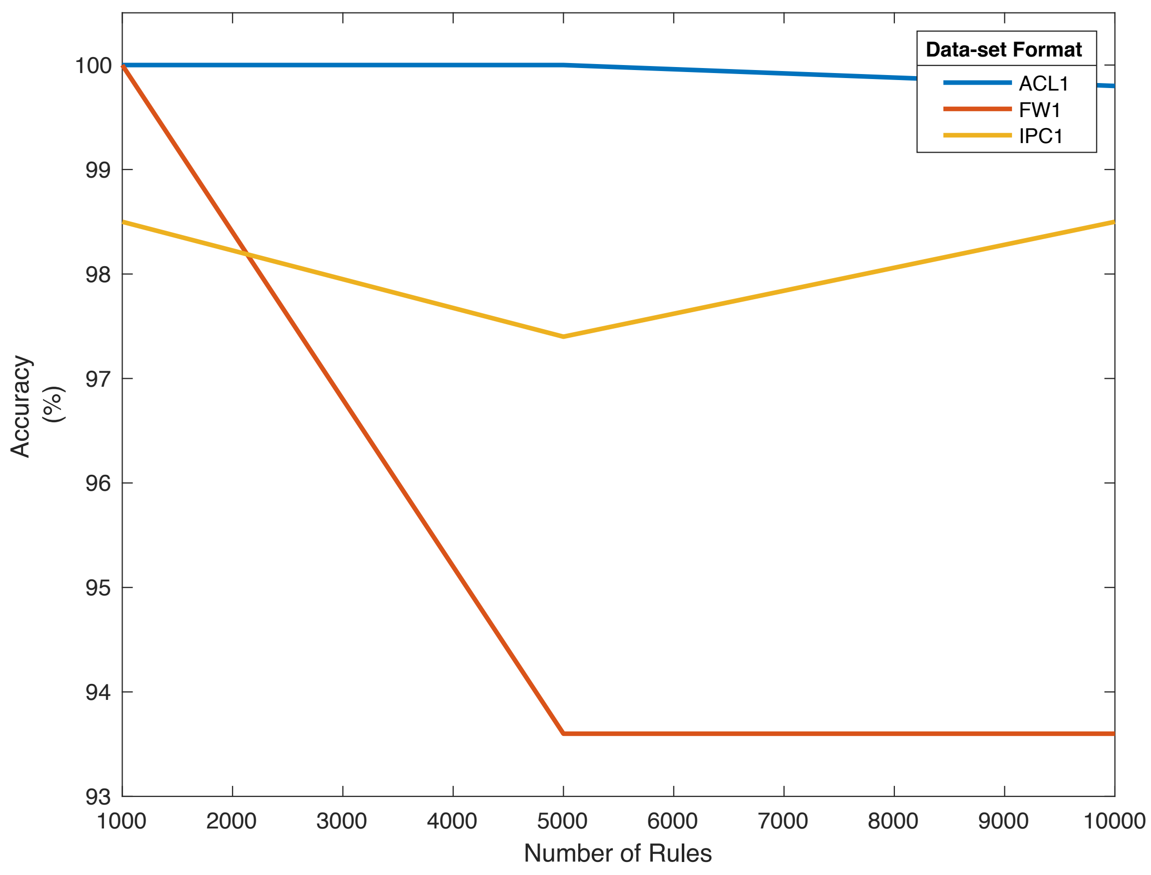

Figure 17: Accuracy comparison between different dataset formats.

It is important to mention that the *ACL* format, compared to *FW* and *IPC*, contains an higher percentage of rules with smaller sizes. In this scenario, it has been proved that finding the matching rule was easier due to the lower dependency on the size of the hyper-cubes.

From the graph we can see that despite the increasing number of rules in the dataset, the trend of the ACL1 remains almost constant around 100% of accuracy. FW1 is the format the shows the worst performances between the datasets taken in account; the accuracy decrease with increasing size of the rule-set, but remain practically constant after 5000 rules are exceeded. Regarding *IP Chain* datasets, the trend remains constant at an accuracy slightly lower than ACL.

From the multiple simulations carried out and reported in this study, it is possible to confirm that the behavior shown by ACL1, FW1 and IPC1 correctly represent the trends of the respective formats.

4.5 Testing additional TCAM

The idea behind the addition of a TCAM for checking the most critical rules [1], came after an accurate analysis of the distribution of the errors that occurred during the simulations. It is shown that each dataset contains a small number of rules which most likely will not be matched, decreasing the actual accuracy of the system. The possibility of rules not founded comes from the fact that some incoming packets are classified in some leaf bins based on their location, but then no hyper-cubes of the corresponding sub-space contains the incoming point. From a

[1]Most critical rules are the ones that most likely will generate mismatches or will not be found during the test phase.

rapid test (1000 randomly generated samples of packet headers are tested against the network) during the training of the GHSOM it was possible to accumulate the most critical rules and store them in an additional TCAM. The idea was considered valuable after proving that the dimension of the input space of the additional TCAM was limited compared to the others, in order to perform the search inside it in parallel to the GHSOM search.

TABLE V: SIZE AND PERFORMANCE WITH ADDITIONAL TCAM

dataset	**1000**	**5000**	**10000**
ACL1	4	8	13
FW1	70	80	96
IPC1	43	100	20

dataset	**1000**	**5000**	**10000**
ACL1	0.3%	0.1%	0.6%
FW1	4.3%	1.4%	3.6%
IPC1	2%	4.6%	1.2%

ACL1, *FW1* and *IPC1* formats have been selected as samples for this analysis because the results of the other corresponding type of rule-sets were similar.

4.6 Rules executed per incoming packet

As pointed out at the beginning of the thesis, one of the main problem of TCAM in packet classification was the poor scalability of throughput and power as the number of rules increase. In the implemented system, the problem has been addressed exploiting the intrinsic proprieties of GHSOMs: the number of rules executed [1] per search is, in the worst case, around 25%

[1]The executed rules are the rules searched by brute force inside the TCAMs to find the highest priority matching rule.

(Format FW3_5000). On average, the number of executed rules is around 15% as it is possible to see in Figure 18.

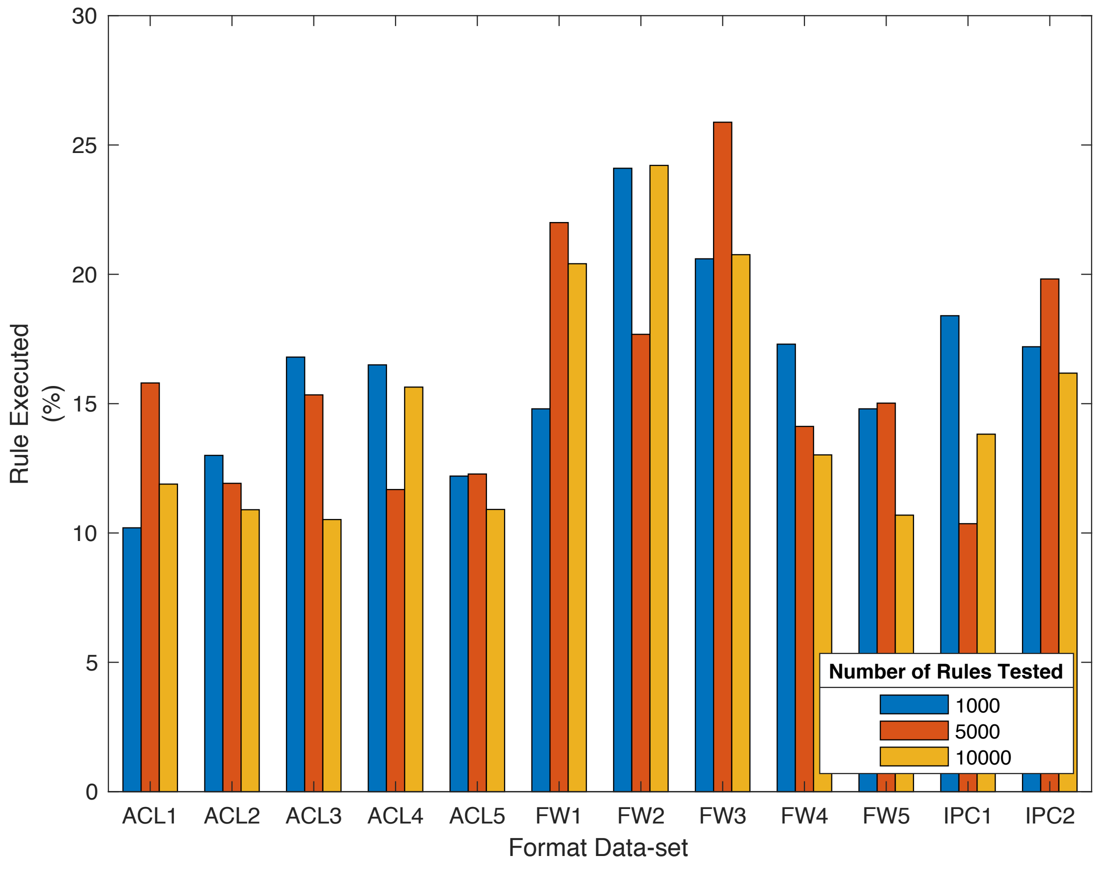

Figure 18: Average of executed rules per research.

The size of the TCAMs on which the searches are performed are contained and depends on the chosen heuristics.

4.6.1 Comparison between different dataset formats - Executed Rules

Just as pointed out for the accuracy analysis, the number of executed rules is different based on the rule-set format. From Figure 19 it is possible to see a lower bound around 10% of executed rules and an upper bound around 25% in the worst cases.

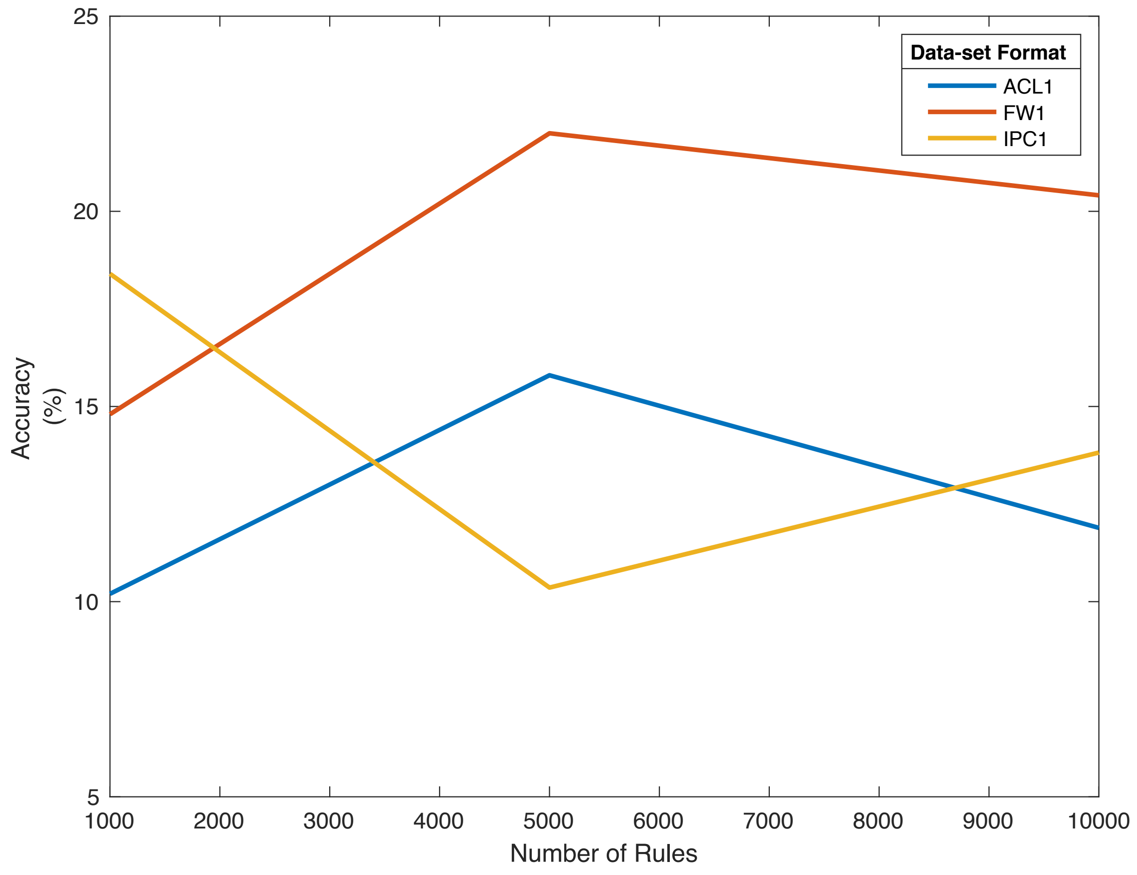

Figure 19: Comparison between executed rules in different formats.

In general, *ACL* seeds generate dataset that requires a lower number of searched rules for computing the correct matching rule, compared to *FW*. The behaviour of *IPC* seeds is similar to the trend found for *ACL*.

Because of the similarity between type of rule-sets of the same format, again the consideration obtained from the analysis of the first seed (ACL1, FW1 and IPC1) can be extended to the other datasets of the same category because the results follow the same pattern.

4.7 Update friendly

One of the main features of SDN applications is the frequent update of the dataset. The system used for addressing the packet classification in SDN needs to be able to handle fast update without affecting performance. The GHSOM structure represents a significantly adaptive and robust system for handling such dynamic rule updates. For example TCAM-based search requires quite a long time to update its content and therefore such dynamic updates adversely affect performances.

In general, it is not necessary to re-build the SOM-tree by retraining the structure with all the rules (including rules that did not change) when only a few new rules are added/modified in the rule-set; clustering new rules based on the hyper-cube's features and centroid locations into the existing GHSOM bin is sufficient.

An example on the ACL1 format has been used for showing the performances of the structure with updates: in particular Figure 20 shows the system's performance for *ACL1_5000*, but the same trend can be found in every format.

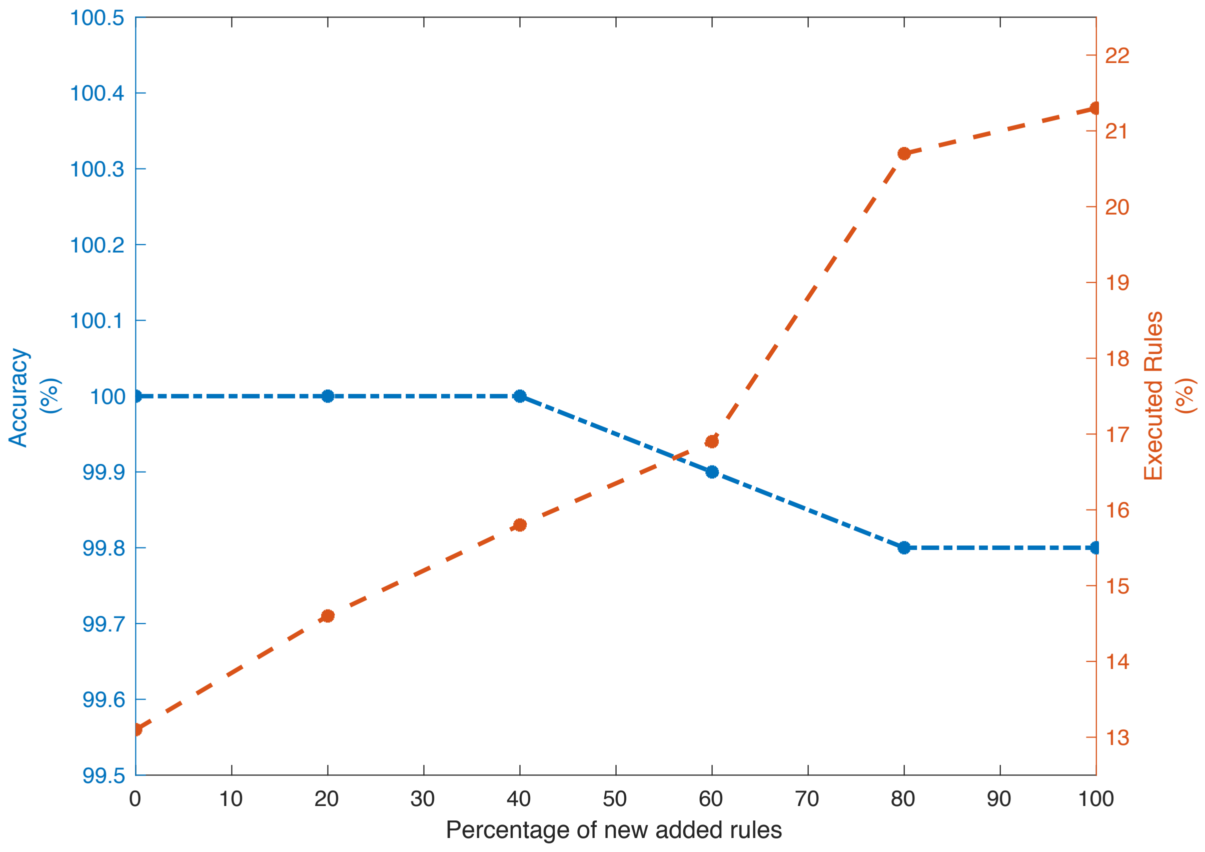

Figure 20: System performance with increasing added rules.

In Figure 20 are plotted the accuracy and the number of executed rules as a function of the percentage of rules added to the dataset. The added rules are generated from a similar distribution of rule-set *ACL1_10000*. The newly added rules are classified into existing SOM bin without regenerating new SOM-tree and neuron weights.

Accuracy remains constant as the number of new added rules increase and the low number of misclassification is taken care by the block shown in 5.2. As expected, adding new rules to existing bin means that the size of the TCAMs increases and the number of executed rules grows as well.

To sum up, the structure shows good results in both accuracy and number of executed rules even when the number of added rules are as many as the original dataset size.

CHAPTER 5

MOVING TO HARDWARE

The main idea of this thesis was to validate a possible alternative solution to the packet classification problem. First of all it was important to prove that GHSOM system was a feasible option for this application. Several tests on the possible feature maps have been carried out before finding the one presented in this study. The hardware implementation has not been developed, but in this chapter some suggestions on a possible architecture are provided to help future development. In particular the discussion will be concentrated on the GHSOM module that represents the core block of this architecture.

Moving to hardware requires to think about bits instead of full precision numbers. The idea followed for performing the translation is the same explained in the paper [13] that was to literally translate each information embodied in the neural network to hardware. From the simulation performed it is possible to estimate the number of bits and the type of representation needed for a correct implementation. In will be necessary to operate on weights and packet headers and, for a precise representation, it will be necessary to use 64 bits and adopt fixed point arithmetic.

Each of the leaf nodes in the SOM-tree eventually terminate in a TCAM module where a subset of rules is contained. This group of rules are classified size and positioned as already explained. Depth and breadth of each single GHSOM module are based on the parameters

τ_1 and τ_1. The way in which each map develops is independent from the construction of the others.

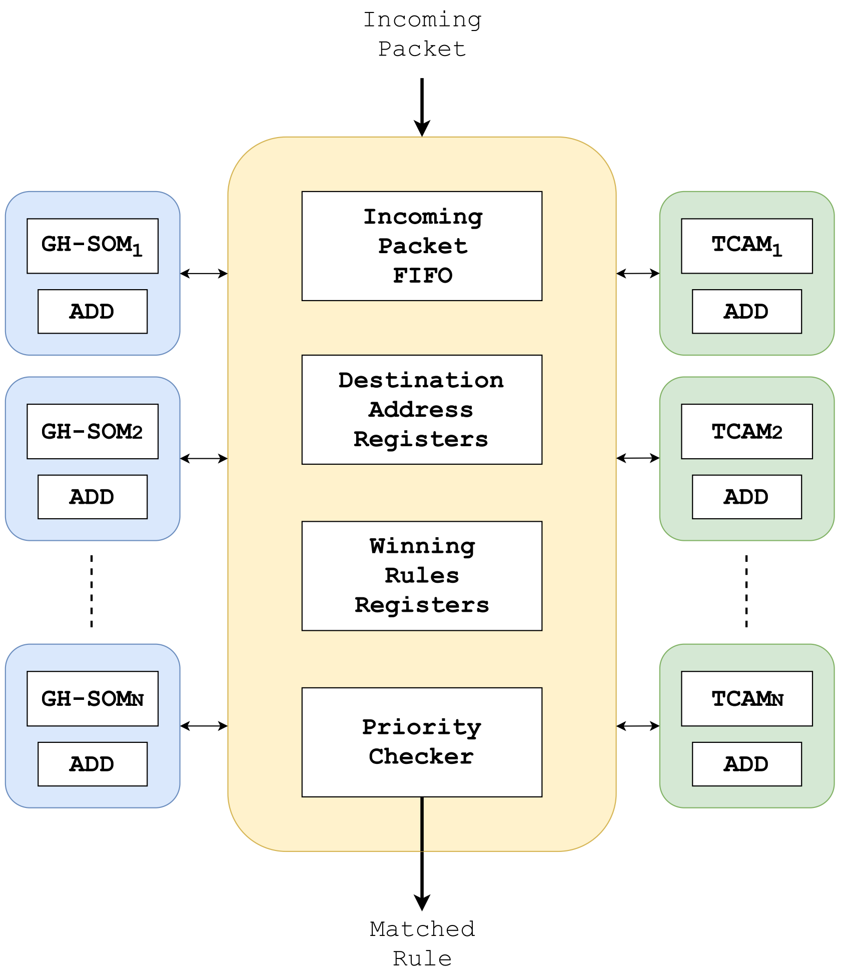

Figure 21: Hardware architecture of the structure.

The input of the hardware implementation are the weights and the corresponding addresses of the GHSOM modules, the packet headers and the feedback signal which comes from the external routers. The weights are computed at in software and stored in specific memories inside the system 5.1, the input packet headers are the points that have to be matched against the dataset which has been used for training the SOM-based tree. Finally, the feedback signal is the one used by the external routers to report the rules' mismatches.

The best way for increasing the throughput of this massively parallel structure is to insert a pipeline. In Figure 21 is presented the physical architecture of GHSOM tree for a pipelined packet processing. The first step for a parallel implementation is to store the incoming packets in a FIFO memory (First In First Out). Thus, is possible to provide to the system a new input every time the computation of the first step for the previous value is ended. When the first group of GHSOMs have computed the lower distance between the weights and the incoming packet, the destination address of the subsequently module is computed as well. The number of destination address registers have to be at least equal to:

$$n_{destination\_add} = n_{parallel\_GHSOM} * n_{layer} \tag{5.1}$$

Assuming that the number of destination address registers and the number of GHSOMs module available are known a priori, inverting Equation 5.1 is possible to find out the maximum number of layers that can be implemented in the system. Therefore the required granularity level of the GHSOM tree is adjusted by setting τ_2, in order to meet the hardware restrictions.

After a number of step equal to the maximum depth of the deepest GHSOM structure, the incoming packet is addressed to the correct TCAM where theoretically the matching rule should be found (if it exists). From the assumption made before, the GHSOMs can grow in depth independently of each other; every tree that is shorter than the deepest one delays the address of the final TCAM location passing it from one destination register to the next one, but without actually computing any new address. In this way, is possible to compute all the TCAM search in parallel and then perform the priority check when all the rules are available. The procedure of these last steps is the following: after the TCAMs have performed their searches, the matching rules are stored inside a set of registers called *Winning Rule Registers*. The number of registers is equal to the number of the parallel GHSOMs of the structure available in the hardware implementation. The winning rules are then sent to the *Priority Checker*: in this work, this unit compares the volume of each matched rule and selects the smallest. In other applications, the priority may not be based on the volume; in such cases, it is only necessary to adjust the block in order to select the rule based on the wanted parameter [1].

5.1 Growing Hierarchical Self-Organized Map module

The GHSOM modules are used for storing the weights and computing the best unit match for every incoming packet. It is important to remember that the first layer of the SOM-tree is only used for generating the division of the dataset based on the size and other characteristics of the hyper-cubes.

[1] In some application the priority is given as a field of the incoming packet.

The weights of the maps which are stored include also the ones of the first layer because they will be useful when new rule is added, even they are not necessary for actual packet classification [1]. The additional constrain regarding the storage of GHSOM modules is that the number maps of the structure has to be lower than N (the total number of GHSOM modules available in the physical architecture). For every rule-set, the higher N is the smaller is the number of rules that each leaf bin has. This facilitate the TCAM-based search; moreover N is also limited by the allocated physical modules of SOM-based tree.

Figure 22 shows the block diagram of a GHSOM module.

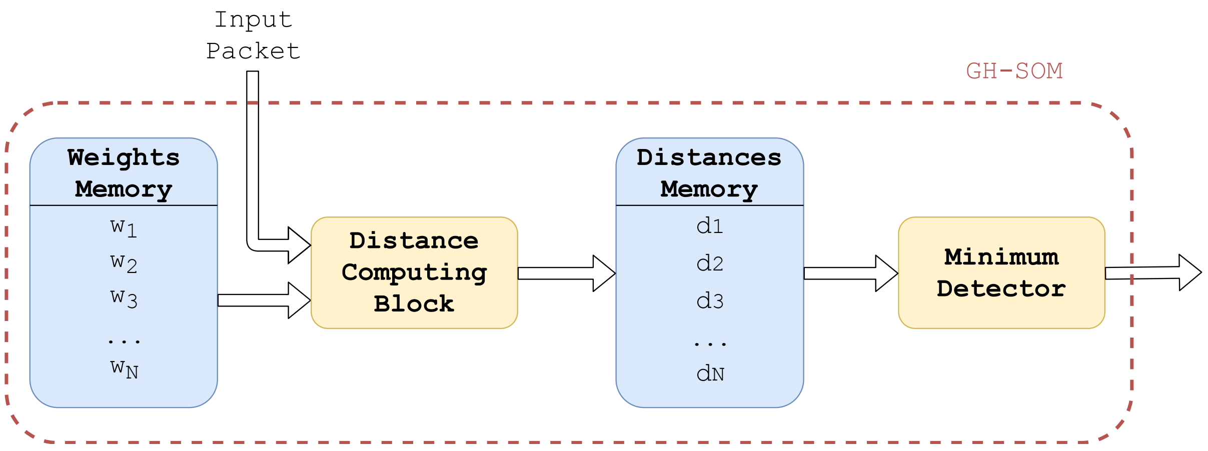

Figure 22: Physical implementation of the GHSOM Block.

[1]From Figure 7 we see that the incoming packet is applied from the second layer.

As can be seen from the figure above, it is composed by four units:

- *Weights Memory* stores the weights of each unit present on the map considered. Together with the weights, the child address of each unit is stored as well (the number of weights that could be stored depends on the size of the memories of the GHSOM modules). In this way, after finding the winner unit, is possible to carry on the search of the best match rule onto the next the module addressed.
- *Distance Computing Block* is used for performing the calculation of the distance between the unit's weights and the incoming packet. This block will be the subject of further analysis in section 5.1.1.
- *Distances Memory* stores all the distances computed by the previous block. The distances are computed sequentially and saved inside the memory one by one.
- *Minimum Detector*, this block is responsible for finding the minimum value between all the distances computed. After this, is possible to move forward to the address of the child (i.e. GHSOM module or TCAM).

The number of weights stored in each GHSOM module can be computed as follows:

$$n_{weights} = 4 * n_{rules\_subset} \tag{5.2}$$

The number 4 in equation Equation 5.2 represents the dimension of the input space. Each field of the incoming packet can be associated to a geometrical point in one of the four dimensions. From the background explained in the Section 2, it is possible to conclude that the number of weights of each unit is equal to the size of the input space.

5.1.1 Distance Computing Unit

The goal of this element is simple; compute the distance between the weights stored and the incoming packet. The weights of a unit can be seen as a matrix because each node is characterized by four weights, one for each dimension. The pseudo-code regarding the *Distance Computing Unit* is provided below:

```
for i=1:number of units
    distance(i) = 0;
    for k = 1:4
        compute distance between the packet and the weight on the i-th dimension:
            partial = weight(i, k) + (- packet(k));
        if partial >= 0
            do nothing;
        else if partial < 0
            2's complement of the partial dimension;
        end
        distance(i) = distance(i) + partial;
    end
    store distance(i);
end
```

At a hardware level, the necessary blocks for the *distance computing unit* are three 64-bits adders and two block of 64 parallel XOR gates (one for each bit). The first wall of XOR gates is used for performing the 2's complement of the input packet header, then, the first adder perform the addition of the two input with the carry fixed at 1. The second segment of the system perform the 2's complement of the result whenever the distance computed is negative. The last adder is used for summing the four distances computed on the four dimension. The sum of the four single dimensional distances represents the total distance in the 4D space. A possible hardware implementation about the previous description is shown in Figure 23.

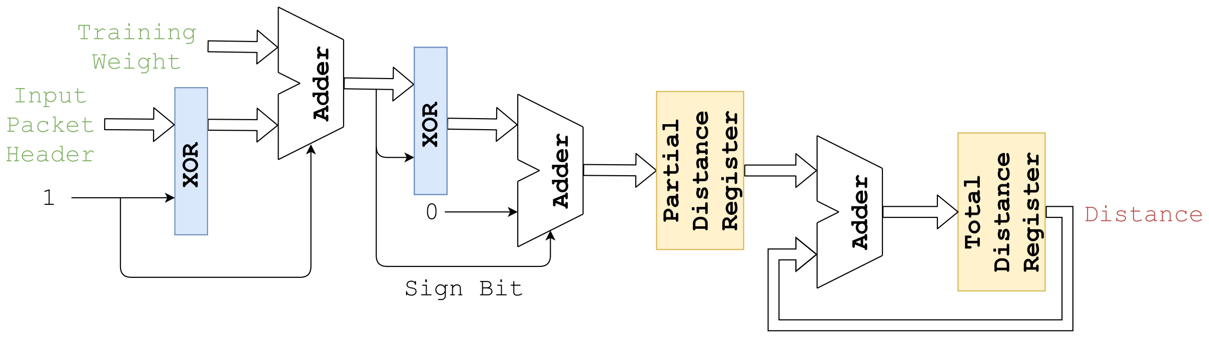

Figure 23: Distance Computing Unit.

The other option would be to use four parallel distance computing unit and then sum the four results. This solution would be faster, but it would have larger area and it would consume more power.

5.1.1.1 Adder

In the DCU there are two adders, each with a different purpose. The first adder computes the difference between the weights stored in memory and the incoming packet. The XOR wall is used for inverting the sign of the input value in order to perform the subtraction using an adder. The second XOR and adder is used for performing the 2's complement whenever the result of the previous operation is negative. If the distance computed is already positive, the value will pass through the second adder unchanged.

The input parameters of the first adder are the weights of the GHSOM module and the fields of the incoming packet [1]. Because of the fact that the input values have different signs, overflow and underflow have not been considered for the implementation of the adder.

Addition is performed between two number represented in fixed point on 64 bits so there is no need to used floating point which would have been more difficult to implement.

5.1.2 Minimum Distance Detector

After the distances are evaluated by the DCU, the minimum has to be detected in order to declare the winning neuron. The general structure is shown in Figure 24 and is composed by:

- *Minimum Detector*
- *Address Register*
- *Minimum Distance Register*

[1]As already mentioned, the packet classification is applied from the second layer of the structure. Geometrically the incoming packets represent a point in the space and therefore they cannot assume negative values.

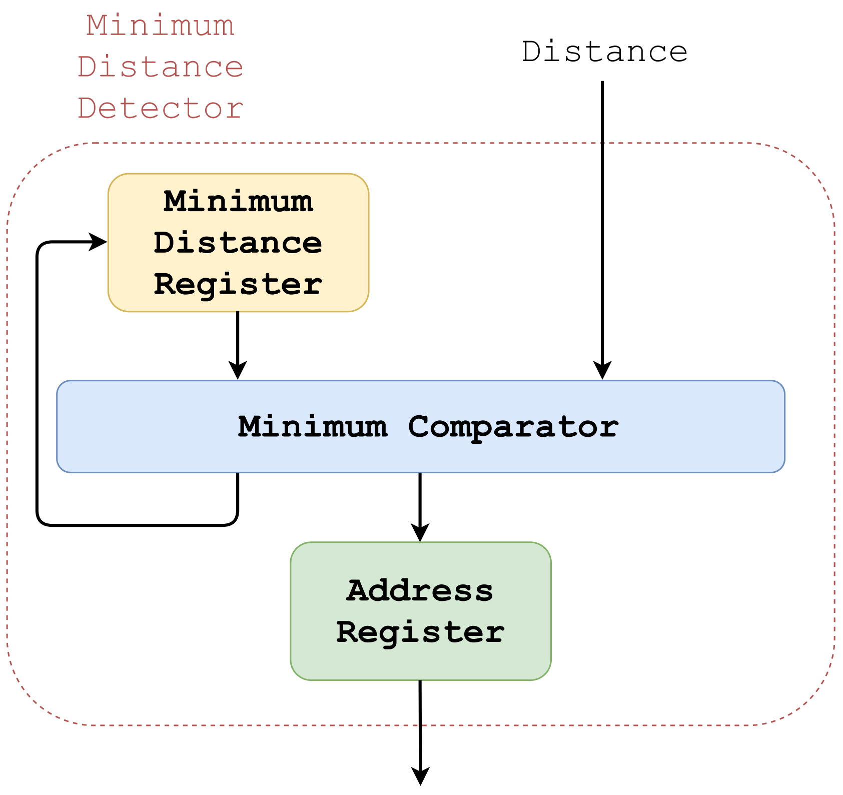

Figure 24: Minimum Distance Detector (MDD).

Based on the availability of a parallel *Minimum Detector* unit, the *minimum distance register* may or may not be necessary. There are two possible scenarios: in the first case, if the number of distances in input is more than the input space of the minimum detector, the minimum distance register will be necessary because the minimum distance has to be stored and compared at every iterations until all the distances have been checked. In the second scenario, if the number of distances to compare is equal or less than the input space of the minimum detector, all the distances could be compared at the same time and saving the temporary minimum distances would not be necessary. Further analysis of both scenarios will take place in next sections.

5.1.2.1 Minimum Comparator

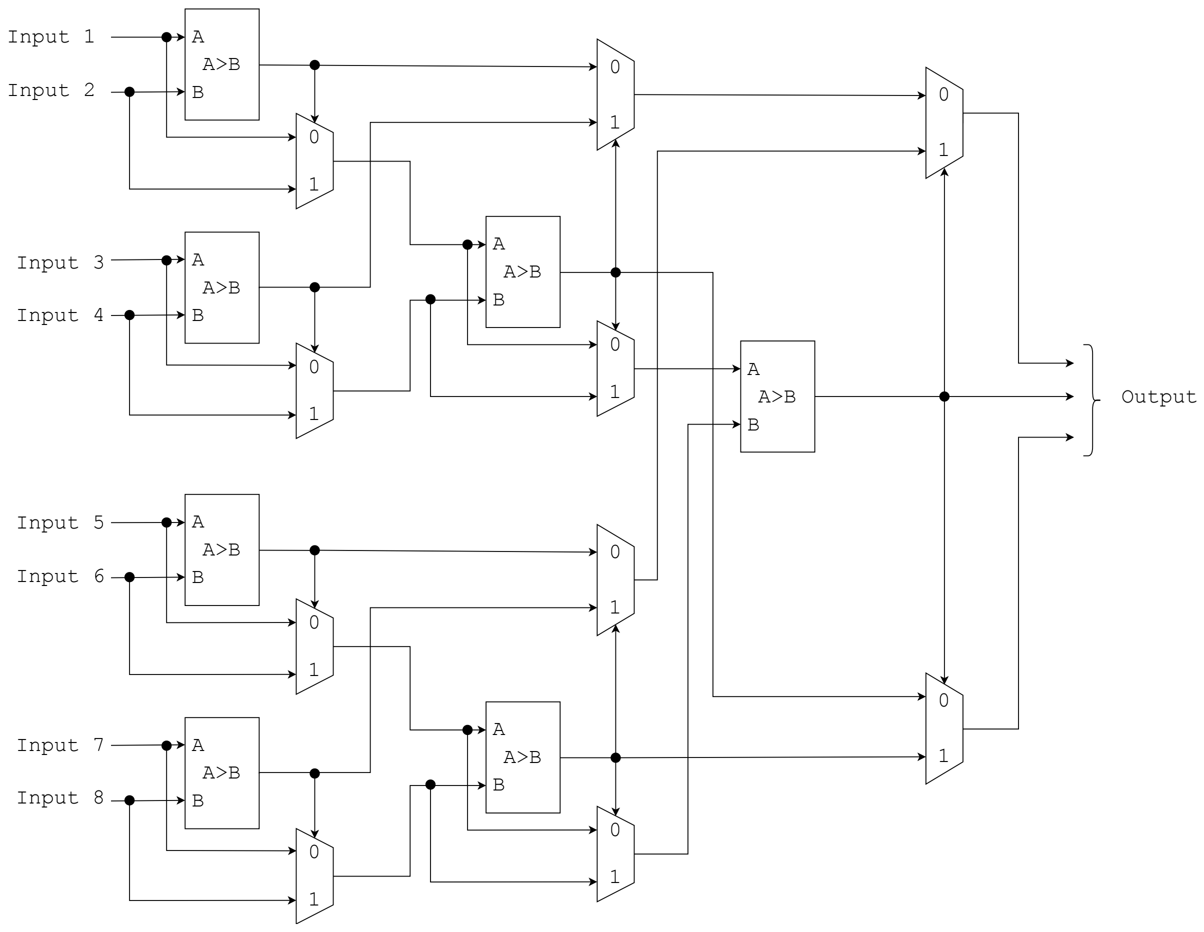

Figure 25: Distances Comparator.

The gate level implementation of a minimum comparator is presented in Figure 24. As mentioned in the previous paragraph, there are two possible implementations based on the hardware availability and the decision made by the designer. If it is not possible to implement the cascade version of the minimum comparator [1], the idea is to perform serially the computation of the minimum distance, storing the minimum value at each iteration and using it as future input. The other reasons for not implementing the cascaded version of this unit are around the trade-off between area, power consumption and speed. Implementing a cascade version means more area and more dynamic power (all the gates switch together); on the other hand, a serial implementation involves less area and less dynamic power, but it will slow down the search of the minimum.

Figure 25 represents a single stage of a *minimum distances comparator*. By means of a handful of multiplexers and single comparators it is possible to obtain the ID of the minimum distance. As already mentioned, if the number of distances to compare is higher than the input space it will be necessary to implement a serial search or to expand the system using a cascade these units. The cascade tree has $\log_2$(number of inputs) layers. During the design phase it will be necessary to take this decision, in order to make the system more suitable for the application's requirement.

The single comparator used in the minimum detector is presented in Figure 26; it compares the two inputs bit-by-bit.

[1]There are several reason why it would be preferable to implement the serial version of the comparator, e.g. there is not enough area avaiable.

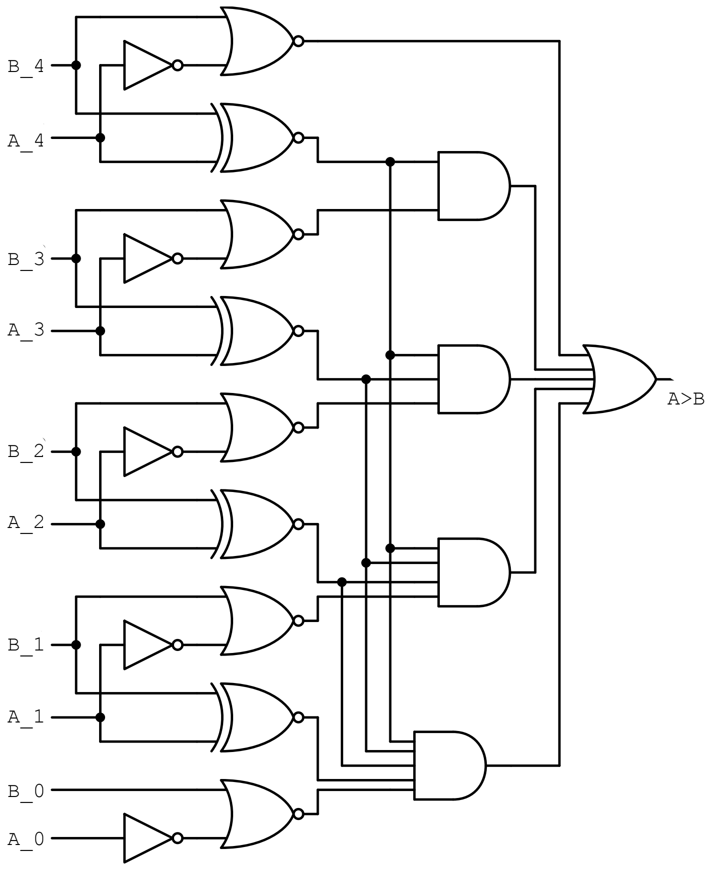

Figure 26: Single Minimum Comparator.

5.1.2.2 Address Registers

The minimum distance detector also consists of an address register necessary for storing the value of the next module responsible for computing the future classification for the incoming packet. It would be possible to analyze the address register as divided in two parts: the first one consists of the address of the actual GHSOM on which the present operations are performed, the second part of the address will be updated based on the results of the winning neuron (the closer unit). Specifically, the first part of the address is the parent address in which the best match unit is computed and the second part of the address indicate the child, the winning node of the map. Therefore, these two parts generate the address of the child which will perform the subsequent operations; the address calculated could point both to another GHSOM module or final TCAM.

5.2 Error Detection Architecture

The main goal of this system is to improve the accuracy of the system during run time and compute the correct rule when the SOM-based tree system cannot find a match. In order to achieve this target, the structure in Figure 27 is used.

The blocks used for this purpose are:

- *GHSOM Hardware Structure* that represents the core of this work: it includes the GHSOM structure presented and the additional TCAM where the most critical rules are stored.
- *100% Accurate Method* represents the parallel system used for support the system when it cannot find the rule or the corrent rule is misclassified.
- *Output Multiplexer* select the correct matching rule which has to be sent to the external routers.

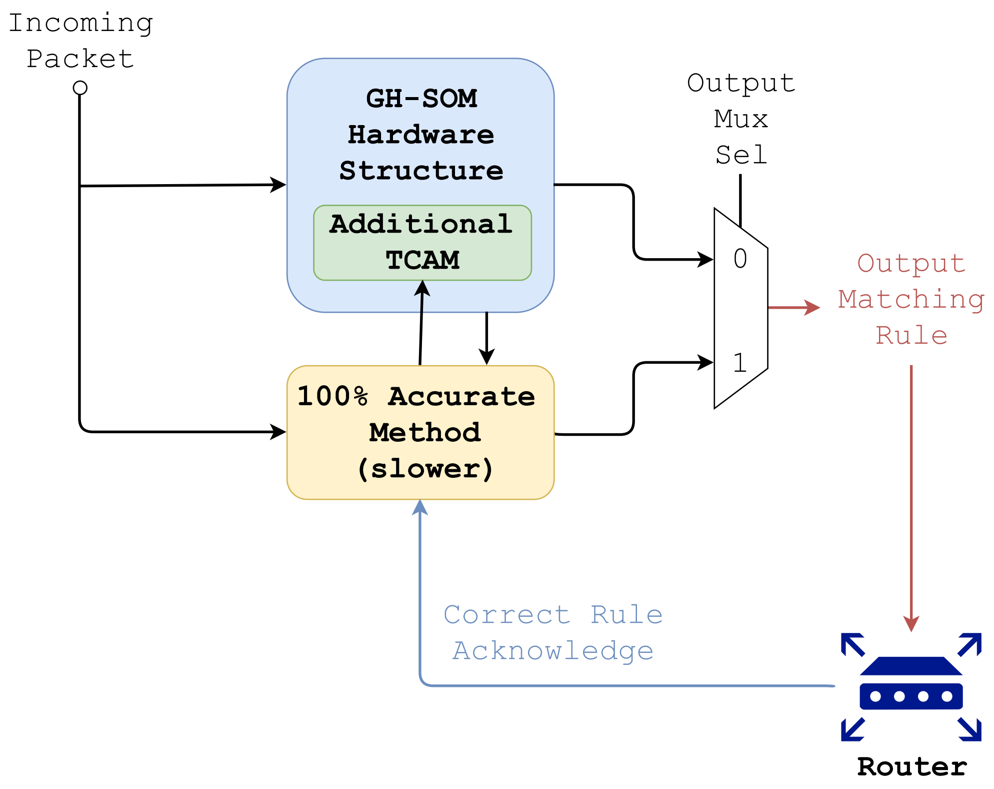

Figure 27: Error Detector.

5.2.1 Back-up system

The back-up system represents the core of the unit used for increasing accuracy of the system at run time. A 100% accurate method has to be used in this block (i.e. brute forces approaches). This unit, being generally purely algorithmic, is slower than the hardware implementation of the GHSOM system. If a TCAM is used as back-up system it means that the search will be still fast, but there will be a loss in terms of power consumed. In any scenario, when the GHSOM system cannot find a matching rule, there will be a loss in terms of performances.

The parallel system is used in two scenarios:

- when the GHSOM system and the additional TCAM do not find any matching rule for the incoming packet, the back up system receives a signal that tells it to start searching the matching rule for the incoming packet. Once the highest priority rule has been found, the matching rule is selected by the output multiplexer and it is sent to the external routers; in parallel, the found rule is stored in the additional TCAM with the other most critical rules of the dataset. In this way it is possible to increase the accuracy of the system almost to 100% for every format of rule-sets.
- this second scenario is the most critical, and it has already been mentioned in 4.2. When a rule is misclassified it means that a wrong rule has been matched to the incoming packet. From the results of simulation performed it is possible to state that those cases are very rare, but still present. Moreover, in some situation is necessary to assure a 100% accuracy. For addressing this problem, a feedback system from the routers can be implemented: when the router receives a matched rule that is wrong [1], it reports it back to the system that compute the correct matching with the back up system. The computation in these cases the computation can be done in parallel unless the system is already busy. The back up system will update the TCAM memory with the mismatched rule in order to prevent the error in the future.

[1]The external routers can recognize the mismatches because they will receive rules that were not meant for their addresses.

The implementation will use a small memory for storing the last match between incoming packet and outgoing rule: in this way, if a rule is sent back from the routers is possible to recognize the mismatched packet and recompute it again with the more accurate method.

The output of the final system is chosen by a multiplexer addressed by a signal that indicates if the GHSOM system found or not a matching rule. The same flag can be used for reporting to the back up system that it need to became the research of the highest priority rule when no match has been found by the SOM-based system.

CHAPTER 6

CONCLUSIONS

The outcome of the project meets the goals set at the beginning; it is proven that Growing Hierarchical Self-Organized Map are a suitable solution for SDN applications and that the system implemented is able to handle large datasets with low performance losses. The intrinsic characteristics of the GHSOM implemented address the redundancy problem related to the known algorithm used for packet classification. It has been proven that the system can handle small updates without the need to re-train the neural network. The approach to this problem is different from earlier published works because it applies the increasingly used SOMs in an application where they had not yet been seen.

There are some advantages of the discussed GHSOMbased packet classification over the current packet classification methods (e.g. TCAM search): first, for an incoming packet, the average number of rules executed by the memories in the SOM-tree algorithm is significantly smaller than in a brute-force search method against all rules. Second, as compared to the software-based algorithm and pure TCAM-based searching scheme, the discussed SOM-tree algorithm is more efficient in dealing with rule updates. Moreover, thanks to the features of the SOM, it has been possible to eliminate the replication of rules that characterizes all the state-of-the-art solutions.

6.1 Future Work

This work represents a revolutionary way of solving the packet classification problem, then a lot of future development can be carried out.

The first step to follow is the implementation of the algorithm from the hardware point of view. With the provided suggestions, a hardware architecture can be easily developed. Moreover, for the purpose of this study, a real profiling of the applications needs to be undertaken. Since the goal of this thesis was to validate the idea, a precise work on the profiling of the application has not been carried out. This can be seen as a future work to further increase the confidence in this structure and it is a needed step before using the algorithm in real-world problems.

Although an hardware implementation of the system is one of the first work to be undertaken in the future, some improvement can be made from an algorithmic point of view; the most important is related to the managing of the unbalancement of the tree due to a large number of rule updates. This complication is associated with both, the SOM-base tree and state-of-the-art solutions (e.g. Efficuts) and is one of the most critical issue related to the approaches applied in packet classification. This disbalance in the structure happens when heavy updates are iterated on a single branch of the structure; consequently the TCAM at the end of the branch will explode in size, eliminating the performance improvement of the structure and forcing a retraining of the network. Thus, one of the first algorithm improvement needed is to find a possibility to take care of big updates without retraining the structure every time.

APPENDIX

EXAMPLE DATASET

This is an example of how a dataset looks like, the dataset showed is *acl1_seed_1000.filter*:

```
@191.147.101.52/31 122.228.222.0/24 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.96.0/22 5.179.96.127/32 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.54/32 167.79.88.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@5.179.100.0/22 216.26.120.0/22 0 : 65535 0 : 65535 0x06/0xFF 0x1000/0x1000

@191.147.101.8/31 23.220.164.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.8/31 5.56.120.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.8/31 116.71.124.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.52/31 122.228.216.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.48/30 122.228.220.0/23 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@5.179.96.176/29 217.238.10.0/23 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.12/30 5.179.100.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.101.48/30 216.26.120.0/22 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x0000

@191.147.100.0/24 122.228.128.0/18 0 : 65535 0 : 65535 0x06/0xFF 0x1000/0x1000

@191.147.101.128/25 122.228.128.0/18 0 : 65535 0 : 65535 0x06/0xFF 0x0000/0x0000

@191.147.100.0/24 49.33.0.0/18 0 : 65535 0 : 65535 0x06/0xFF 0x0000/0x0000

@5.179.97.0/24 5.179.96.128/25 0 : 65535 0 : 65535 0x00/0x00 0x0000/0x000
```

CITED LITERATURE

1. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J.: Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, March 2008.

2. Consortium, O. S.: Openflow switch specification, v. 1.4.0. Technical report, ONF TS-025, Open Networking Fundation, 2013.

3. Kang, N., Liu, Z., Rexford, J., and Walker, D.: Optimizing the "one big switch" abstraction in software-defined networks. In Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

4. Katta, N., Alipourfard, O., Rexford, J., and Walker, D.: Cacheflow: Dependency-aware rule-caching for software-defined networks. In Proceedings of the Symposium on SDN Research, SOSR '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

5. Vamanan, B. and Vijaykumar, T. N.: Treecam: decoupling updates and lookups in packet classification. In CoNEXT, 2011.

6. Lakshman, T. V. and Stiliadis, D.: High-speed policy-based packet forwarding using efficient multi-dimensional range matching. SIGCOMM Comput. Commun. Rev., 28(4):203–214, October 1998.

7. Vamanan, B., Voskuilen, G., and Vijaykumar, T. N.: Efficuts: optimizing packet classification for memory and throughput. SIGCOMM Comput. Commun. Rev., 41(4):–, August 2010.

8. eds. T. Kohonen, M. R. Schroeder, and T. S. Huang Self-Organizing Maps. Berlin, Heidelberg, Springer-Verlag, 3rd edition, 2001.

9. Dittenbach, M., Rauber, A., and Merkl, D.: Uncovering hierarchical structure in data using the growing hierarchical self-organizing map. Neurocomputing, 48(1):199 – 216, 2002.

CITED LITERATURE (continued)

10. Rauber, A., Merkl, D., and Dittenbach, M.: The growing hierarchical self-organizing map: exploratory analysis of high-dimensional data. IEEE Transactions on Neural Networks, 13(6):1331–1341, Nov 2002.

11. Pampalk, E. and Chan, A.: Gh-som matlab toolbox, `http://www.ifs.tuwien.ac.at/~andi/ghsom/description.html`, 2002. [Online; accessed 01/15/2019].

12. Taylor, D. E. and Turner, J. S.: Classbench: A packet classification benchmark. IEEE/ACM Trans. Netw., 15(3):499–511, June 2007.

13. Li, Z., Huang, Y., and Lin, W.: Fpga implementation of neuron block for artificial neural network. In 2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC), pages 1–2, Oct 2017.

14. Singh, S., Baboescu, F., Varghese, G., and Wang, J.: Packet classification using multidimensional cutting. In Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM.

15. Hung, S.-C., Iliev, N., Vamanan, B., and Trivedi, A. R.: Self-organizing maps-based flexible and high-speed packet classification in software defined networking. in (to appear) *Proceedings of the International Conference on VLSI Design and Embedded Systems (VLSID)*, 2019.

VITA

NAME	Marco Montagna
EDUCATION	
	B.S. in Electronic Engineering, Jul 2017, Politecnico of Turin, Italy
	M.S. in Electrical and Computer Engineering, University of Illinois at Chicago, May 2019 (Expected), USA
	Specialization Degree in Electronic Systems - Electronic Engineering, Jul 2019 (Expected), Politecnico of Turin, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2017 - IELTS examination
	A.Y. 2018/19 One Year of study abroad in Chicago, Illinois
	A.Y. 2017/18. Part of lessons and exams attended exclusively in English
TECHNICAL SKILLS	
Basic level	Python
Average level	C#, Arduino, Simulink, MPI
Advanced level	C, C++, VHDL, Matlab, TeX
Hardware Design	MentorGraphics Modelsim, Synopsys, Altera Quartus, Cadence Encounter, Simulink, LTspice, ORCad, VLSI and Low Power Design
Analog Electronics	Mixers, Amplifier, PLL, VCO, synthesizer and ADC/DAC design
Laboratory Instruments	Oscilloscope, Multimeter, Spectrum Analyzer, Power Supply, Function and Signal Generator

VITA (continued)

WORK EXPERIENCE AND PROJECTS

Mar 2017 - Jun 2017	*Academic Internship at Instituto Superiore Mario Boella, Turin, Italy.* Curricular internship with the Politecnico of Turin. Designed and tested a wireless charging device for a GPS traker. Gained good OrCAD and Eclipse skills; solder and test phases were performed too.
Apr 2014 & Apr 2018	*Teaching Assistant for Universit degli Studi of Turin, Italy.* For two different years I taught to a group of high school students basics of semiconductors characteristics and photovoltaic panels.
Jan 2019 - May 2019	*Master Thesis researcher at AEON LAB, Chicago, USA.* Design of a Growing Hierarchical Self Organized Map for Packet Classification. Main goal: prove the feasibility of the implementation of neural network to the packet classification problem.
2017	Wireless charging device Hardware and software design of a wireless charging device for GPS tracker using OrCAD. The main components have been designed and soldered. Gained strong C++ coding skills on Eclipse. Test phase performed on the prototype using the oscilloscope.
2018	Top-down design of a custom DLX processor Developed a personalized version of the DLX for the Microelectronics class. Hardwired Control Unit and fully pipelined structure. Industrial design flow followed: (1) pen and paper project of each component and the whole system; (2) VHDL description; (3) Advanced Testbench system developed VHDL; (4) Synthesis with Synopsys; (5) Physical design (by means of Cadence Encounter). Developed a processor for high-speed performances, with an eye on Low-power.
2018	Butterfly for FFT Designed an ASIC for Fast Fourier Transform (FFT) Computation by means of Kooley-Tukey algorithm. Improved knowledge on most advanced design techniques, such as variable lifetime, micro-programming and pipelining. Realization of the project included schematics, optimizations and tests through Modelsim. Control Unit and Datapath designed through VHDL.
2018	Digital State Analyzer

VITA (continued)

A digital state anayzer was designed in VHDL and tested on FPGA Altera DE2. Data could be inserted from PC and sent to FPGA by serial line connection; the analyzer could detect ’1’, ’0’ and glitches.