UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

Alex Midwar Rodriguez Ruelas

Studies on the Applicability of Neural Networks for Load Balancing in OpenFlow-based Data Center Networks

Estudos de aplicabilidade de redes neurais para balanceamento de carga em redes de data centers baseados em OpenFlow

CAMPINAS

2020

Alex Midwar Rodriguez Ruelas

Studies on the Applicability of Neural Networks for Load Balancing in OpenFlow-based Data Center Networks

Estudos de aplicabilidade de redes neurais para balanceamento de carga em redes de data centers baseados em OpenFlow

Dissertation presented to the Faculty of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering.

Dissertação apresentada à Faculdade de Engenharia Elétrica e Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Eletrica, na Àrea de Engenharia de Computação.

Orientador: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Alex Midwar Rodriguez Ruelas, e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

CAMPINAS

2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

R618s

Rodriguez Ruelas, Alex Midwar, 1980-
Studies on the applicability of neural networks for load balancing in OpenFlow-based data center networks / Alex Midwar Rodriguez Ruelas. – Campinas, SP : [s.n.], 2020.

Orientador: Christian Rodolfo Esteve Rothenberg.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Redes neurais (Computação). 2. Redes definidas por software (Tecnologia de rede de computadores). 3. Redes de computadores. I. Esteve Rothenberg, Christian Rodolfo, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Estudos de aplicabilidade de redes neurais para balanceamento de carga em redes de data centers baseados em OpenFlow
Palavras-chave em inglês:
Artificial neural network
Software-defined networking
Computer network
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Rafael Pasquini
Tiago Fernandes Tavares
Data de defesa: 19-06-2020
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-5363-8652
- Currículo Lattes do autor: http://lattes.cnpq.br/5950257121828711

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Alex Midwar Rodriguez Ruelas RA: 151594
Data da Defesa: 19 de Junho 2020
Título da Tese:
"Studies on the Applicability of Neural Networks for Load Balancing in OpenFlow-based Data Center Networks"
"Estudos de aplicabilidade de redes neurais para balanceamento de carga em redes de data centers baseados em OpenFlow"

Prof. Dr. Christian Rodolfo Esteve Rothenberg - (Presidente, FEEC/UNICAMP)
Prof. Dr. Rafael Pasquini (UFU) - Membro Titular
Prof. Dr. Tiago Fernandes Tavares (FEEC/UNICAMP) - Membro Titular

Acknowledgements

I would like to thank my advisor prof. Dr. Christian Rothenberg, for his support, valuable comments, and optimistic approach throughout the thesis, besides guidance he provided during the completion of this dissertation.

At the State University of Campinas (Unicamp), I have had the opportunity to learn from the best professors. I would like to acknowledge Prof. Dr. Mauricio Magalhães, Prof. Dr. Léo Pini, Prof. Dr. Yuzo Iano, Prof. Dr. Marco Aurelio and Prof. Dr. Romis Ribeiro for sharing their knowledge and experience, which served as a constant motivation throughout this work.

I had the good fortune to meet very talented people within the INTRIG and LCA groups. My special thanks go to Mateus, Samira, Gyanesh, Danny, Luis, Raphael, Anderson, Hirley, Ramon, Alaelson, Claudio, Talita, and Javier. We were the best group that anyone wants to have.

I would like to thank the committee members Rafael Pasquini, Tiago Fernandes Tavares, and Christian Rothenberg for the corrections that contributed to enrich this work.

I would especially like to thank my amazing family for the love, support, and constant encouragement I have gotten over the years.

This work was carried out with the support of the Desenvolvimento Científico e Tecnológico (CNPq), bolsa processo No 134031/2015-6.

“Failure is the stepping stone to success.”

Japanese proverb

Abstract

The growth of cloud application services delivered through data centers with varying traffic demands unveils the limitations of traditional load balancing study. Aiming at attending the evolving scenarios and improving the overall network performance. This research proposes a load-balancing study based on an Artificial Neural Network (ANN) in the context of Knowledge-Defined Networking (KDN). KDN seeks to leverage Artificial Intelligence (AI) techniques for the control and operation of computer networks. KDN extends Software Defined Networking (SDN) with advanced telemetry and network analytics introducing a so-called Knowledge Plane. The ANN is capable of predicting the network performance according to traffic parameters by creating a model of traffic behavior using the available bandwidth and latency measurements over different paths. The study includes training the ANN model to choose the least loaded path routing. We conduct a series of experiments to verify the proposed study. The experimental results show that the performance of the KDN-based data center has been greatly improved.

Keywords: Artificial Neural Network; Software Defined Networking; Knowledge Defined Networking, sFlow.

Resumo

O crescimento dos serviços de aplicativos em nuvem fornecidos por os data centers com demandas de tráfego variáveis revela limitações dos métodos tradicionais de balanceamento de carga. Visando em atender aos cenários em evolução e melhorar o desempenho geral da rede. Esta pesquisa propõe um estudo de balanceamento de carga baseado em uma Rede Neural Artificial (ANN) no contexto da Rede Definido por Conhecimento (KDN). A KDN busca alavancar as técnicas de Inteligência Artificial (AI) para o controle e operação de redes de computadores. O KDN amplia o Redes Definidas por Software (SDN) com telemetria avançada e análise rede, introduzindo o chamado Plano de Conhecimento. A proposta da ANN é capaz de prever o desempenho da rede de acordo com os parâmetros de tráfego, criando um modelo de comportamento de tráfego baseado em medições de largura de banda e latência sobre diferentes caminhos. O estudo inclui o treinamento do modelo ANN para escolher o roteamento de caminho menos carregado. Realizamos uma série de experimentos em um ambiente emulado para validar o estudo proposto. Os resultados experimentais mostram que o desempenho do data center baseado em KDN foi bastante aprimorado.

Palavras-chaves: Redes Neurais Artificial; Redes Definidas por Software; Redes Definidas por Conochecimento; sFlow.

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence.
ANN	Artificial Neural Network.
API	Application Programming Interface.
App	Application.
ASICs	Application Specific Integrated Circuits.
ATCA	Advanced Telecommunications Computing Architecture.
AUC	Area Under the ROC Curve.
BGP	Border Gateway Protocol.
BGP-LS	BGP protocol with Link-State Distribution.
CPU	Central Processing Unit.
DDoS	Distributed Denial of Service.
ECMP	Equal-Cost Multi-Path.
FPGA	Field Programmable Gate Array.
GUI	Graphical User Interface.
HTTP	Hypertext Transfer Protocol.
ICMP	Internet Control Message Protocol.
IP	Internet Protocol.
IPFIX	Internet Protocol Flow Information Export.
IT	Information Technology.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
KDN	Knowledge-Defined Networking.
KVM	Kernel-based Virtual Machine.
MD-SAL	Model-Driven Service Abstraction Layer.
MLP	Multilayer Perceptron.
MPLS	Multi-Protocol Label Switching.
NA	Network Analytic.
NAT	Network Address Translator.
NB	Northbound.
NETCONF	Network Configuration Protocol.
NOS	Network Operating System.
ODL	OpenDaylight.

OF	OpenFlow.
ONF	Open Networking Foundation.
ORAN	OpenFlow Routers for Academic Networks.
OS	Operating System.
OSGi	Open Service Gateway initiative.
OSPF	Open Shortest Path First.
OVS	Open vSwitch.
OVSDB	Open vSwitch Database.
OXM	OpenFlow Extensible Match.
PBB	Provider Backbone Bridges.
PC	Principal Component.
PCA	Principal Component Analysis.
QoS	Quality of Service.
ReLU	Rectified Linear Unit.
REST	Representational State Transfer.
ROC	Receiver Operating Characteristic.
RPC	Remote Procedure Call.
SAL	Service Abstraction Layer.
SB	Southbound.
SCTP	Stream Control Transmission Protocol.
SDN	Software-Defined Networking.
sFlow	sampled Flow.
sFlow-RT	sFlow Real Time.
SGD	Stochastic Gradient Descent.
SMOTE	Synthetic Minority Over-sampling Technique.
SSH	Secure Shell.
TCP	Transmission Control Protocol.
ToS	Type of Service.
UDP	User Datagram Protocol.
URL	Uniform Resource Locator.
VLAN	Virtual LAN.
VM	Virtual Machine.
VTN	Virtual Tenant Network.
XML	eXtensible Markup Language.
YANG	Yet Another Next Generation.

Contents

1 Introduction

Data centers are the leading hosting infrastructures of Internet applications and services (i.e., multimedia content, Internet banking, and social networks). Traditional load balancing methods in such data center networks use dedicated hardware devices to distribute the network traffic in different server replicas. Although this approach can achieve high performance in general, it is expensive, and it lacks flexibility in its configuration, which cannot be dynamically adjusted based on real-time network state or other information.

As an innovative networking technology that offers centralization (logical) of network control and introduces the ability to program the network. Software-Defined Networking (SDN) has been applied to many load balancing systems (AL-FARES *et al.*, 2008), (AKYILDIZ *et al.*, 2014), (KREUTZ *et al.*, 2015), based on decoupling of the data and control planes, in which the forwarding state in the data plane is managed by a remote control plane. OpenFlow version 1.3 (ONF, 2013) is currently the most well-known SDN protocol. OpenFlow-based SDN can be used for monitoring network switches, sending policy commands to each switch, programming routes with flow tables that can define the planned path, and allowing the dynamic reprogramming of network devices through an external controller that contains the control logic with a global knowledge of the network state.

Knowledge-Defined Networking (KDN)(MESTRES *et al.*, 2017) is a recent networking paradigm that allows the application of Artificial Intelligence (AI) techniques for controlling and operating the network. KDN relies on SDN, telemetry, and network analytics, and introduces a so-called Knowledge Plane for network control and management operations.

1.1 Research Proposal

The growth of cloud application services delivered through data centers with varying traffic demands unveils the limitations of traditional data center load balancing methods. Aiming to attend evolving scenarios and improve the network performance of cloud data centers, this work proposes a load balancing study based on an Artificial Neural Network (ANN) in the context of Knowledge-Defined Networking (KDN). Therefore, our main research question can be stated as follows: Is it possible to apply KDN using standardized SDN protocols (OpenFlow) to gather knowledge of the data center network, and employ that knowledge to improve network control using Artificial Intelligence techniques?

1.2 Research Objective and Methodology

The main objective of this work is to study the applicability of implementing an ANN inside an SDN controller to provide a load balancing method for dynamic path selection of individual data flows to improve the network performance. In order to evaluate the load balancing methods based on ANN, we compare our strawman proposals with two alternative methods, namely (random) static load balancer, and equal-cost multi-path (ECMP) routing (HOPPS, 2000).

1.2.1 Specific Objectives

The specific objectives to carry this research are:

- Design and implement an ANN-based load balancing method in an open source commercial-grade SDN/OpenFlow controller (OpenDaylight) responsible for the datacenter routing.
- Train the ANN based on a monitoring system that collects network-level metrics (bandwidth and latency) of each OpenFlow switch in the data center.
- Experimentally evaluate the proposed ANN-based SDN load balancing methods using multiple paths between source and destination nodes in a fat-tree data center topology.

1.3 Thesis Structure

In addition to this introductory chapter, the dissertation is organized with the other chapters as follows.

Chapter 2 presents background information to understand the different components of the design and implementation of our work, covering relevant aspects of SDN, KDN, sFlow-RT, Fat-Tree Network, and ANN.

Chapter 3, we explain our new architecture of the proposed ANN-based SDN load balancing. We present the design per component to break down the complexity and provide focus on each subtask.

Chapter 4 presents the experiments carried out, as well as the results of two case studies for the prototype built on the KDN architecture.

Chapter 5 concludes the work with a presentation of the main conclusions of the research, final remarks, and future work.

2 Background and Related Work

In this chapter, we provide a general background which is considered appropriate for the understanding of research design. The first section provides the background on Artificial Neural Networks (ANNs). The next sections examine the concepts of Software Defined Networking (SDN), OpenFlow (OF) protocol, the OpenDaylight controller, the Mininet emulation environment,. We explain how to monitor network traffic using sFlow-RT in OpenFlow-enabled networks. This chapter finishes with a review of load balancing methods, and related works.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are inspired in the behavior of biological neurons and their interconnections, Thanks to their characteristics, neural networks are the protagonists of conceptual innovations in machine learning systems and more specifically in the artificial intelligence. The neurophysiologist Warren McCulloch and the mathematician Walter Pitts inspired by the biological neuron, they were proposed the first mathematical model in 1943, this was the first artificial neural architecture. Since then many other architectures have been invented. In 1975, the development of the backpropagation algorithm was first introduced, which has emerged as the most popular learning algorithm for the training of multiple layers networks (HAYKIN *et al.*, 2009).

A sudden advance in the performance of the ANN was in the 21th century. A group of computer scientists and engineers at the University of Toronto demonstrated a way to significantly advance computer vision using deep neural nets running on GPUs. Therefore, it was possible to achieve huge performance gains in computing and GPU made possible to train complex network.

2.1.1 Models of a Neuron

A *neuron* is the basic unit of a neural network that performs the information processing. The block diagram of Figure 1 shows the model of a *neuron* (HAYKIN *et al.*, 2009).

The following are the parameters that describe a neuron based on Figure 1:

- Input signals $\{ x_1, x_2, ..., x_n \}$

  These are the signals or measures coming from the external medium and that represent the values assumed by the variables of a specific application. The input signals

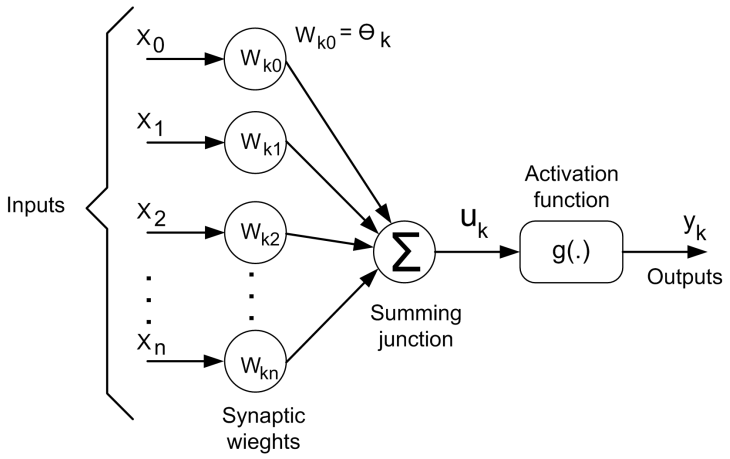


Figure 1 – Model of a neuron; w_{k0} accounts for the bias θ_k. Source (HAYKIN *et al.*, 2009)

are usually normalized in order to increase the computational efficiency of the learning algorithms.

- The synaptic weights { $w_1, w_2, ..., w_n$ }

  These are the values that served to weight each of the input variables of the network, allowing to quantify their relevance to the functionality of the respective neuron.

- Linear combiner { Σ }

  Its function is to aggregate all the input signals that were weighted by the respective synaptic weights in order to produce an activation potential value.

- Bias (Offset) { θ }

  The bias θ has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively.

- Activation potential { μ }

  It is the result produced by the difference of the value produced between the linear combiner and the bias, for instance, if $\mu \geq \theta$ then the neuron produces an excitatory potential, otherwise the potential will be inhibitory.

- The activation function { g }

  The activation functions are used to introduce non-linearity in neural networks, with the aim of limiting the amplitude of the output of a neuron.

- The output signal { y }

  It consists of the final value produced by the neuron in relation to a certain set of input signals and may also be used by other neurons that are sequentially interconnected.

In mathematical terms, we may describe the neuron k depicted in Figure 1 by writing the pair of equations:

$$\mu_k = \sum_{i=1}^{n} x_i w_i + b_k \tag{2.1}$$

and

$$y_k = g(\mu_k) \tag{2.2}$$

In Equation 2.1, we have added a new synapse. Its input is :

$$x_0 = +1 \tag{2.3}$$

and its weight is

$$w_{k0} = b_k \tag{2.4}$$

Where the output of neuron k can be described by:

$$y_k = f(u_k) = f(\sum_{j=0}^{n} w_{kj} x_j) \tag{2.5}$$

2.1.2 Types of Activation Function

There are many activation functions at our disposal to change the output of our neuron. Remember: An activation function is simply a mathematical function that transforms x in the output *f(x)*. The activation function introduces non-linearity into network, allowing it to learn complex non-convex functions.

- Sigmoid (Logistic) Function:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.6}$$

- Hyperbolic Tangent Function:

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.7}$$

- Rectified Linear Unit (ReLU) Function:

$$f(x) = max(0, x) \tag{2.8}$$

2.1.3 Network Architectures

There are different types of neural networks, which can be distinguished by the manner to connect the nodes (structure) and directions of signal flow generally, neural networks may be differentiated as follows:

1. **Feedforward network** is a non-recurrent network which incorporates inputs, outputs, and hidden layers. Where information flows from inputs to outputs in only one direction.

2. **Recurrent network** has feedback paths, meaning they can have signals traveling in both directions using loops. Thus, some of the information flows not only in one direction from input to output, but also in the opposite direction.

2.1.4 MultiLayer Perceptron

The multilayer perceptron (MLP) has of one or more hidden layers, where the neuron output in one layer has direct connections to the neurons of the next layer. The architectural in Figure 2 illustrates the design of a multilayer feedforward neural network (HAYKIN *et al.*, 2009). The function of hidden neurons is to intervene between the external input and the network output in some useful manner :

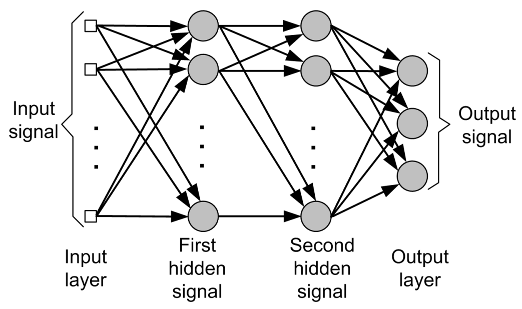


Figure 2 – Architectural of a multilayer perceptron with two hidden layers. Source (HAYKIN *et al.*, 2009)

In particular, a typical learning algorithm for MLP networks is the so-called back propagation's algorithm. It is important to note that in MLP networks, although you don't know the desired outputs of the neurons of the hidden layers of the network, it is always possible to apply a supervised learning method based on the minimization of an error function via the application of gradient-descent techniques.

2.1.5 Loss Function

A function that measures the difference, or loss, between a predicted label and a true label. Denoting the set of all labels as y and the set of possible predictions as y', a loss function L is a mapping L*:* $y \times y'$ $\mathbb{R}+$. In most cases, $y = y'$ and the loss function is bounded, but these conditions do not always hold.

Root Mean Square Error (RMSE):

$$RSME = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - y_i')^2} \tag{2.9}$$

2.2 Software Defined Networking

In traditional networking, the control system is distributed, and the network protocols are running inside the network devices (e.g., routers and switches). Each network element is a separate entity with its control plane and data plane implemented in a local control plane on embedded hardware. Hence, all the devices in the network make their own decision by using information that is shared (see Figure 3).

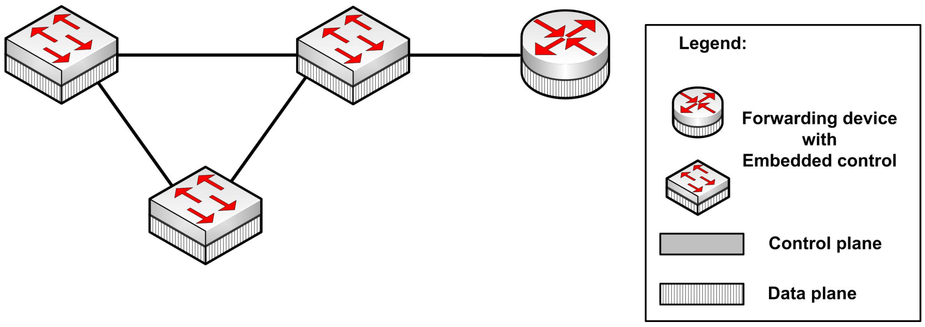


Figure 3 – Traditional networking (with distributed and middleboxes). Source: (KREUTZ *et al.*, 2015)

The control plane decides how to handle the network traffic and device configurations, and the data plane handles input and output control, such as traffic shaping and policing whenever it is necessary. The decisions made by the control plane are informed to the data plane (KREUTZ *et al.*, 2015). This makes it difficult to add new functionality to traditional networks, reducing flexibility and hindering innovation and evolution of the networking infrastructure (SHENKER *et al.*, 2011), (NUNES *et al.*, 2014).

In contrast, Software Defined Networking (SDN) is an innovative approach to networking in which the data plane (forwarding hardware) and the control plane (control decisions) are decoupled (see Figure 4), as opposed to traditional networking. The SDN can simplify network management, allowing innovation through network programmability and offers a logically centralized control plane. The control plane is extracted from network devices to an external entity, the so-called SDN controller or the network operating system (NOS). The controller is installed on a general-purpose server, providing more processing power than a local control plane on embedded hardware. Hence, the network devices will become simple (packet) forwarding elements that can be programmed via an open interface (e.g., ForCES (DORIA *et al.*, 2010), OpenFlow (MCKEOWN *et al.*, 2008), etc.). Depending on the rules installed by the controller, a forwarding element can behave like a router, switch, firewall, or even a middlebox such as a firewall, Network Address Translator (NAT) or load balancer (ANDERSSON; TERMANDER, 2015), (SHENKER *et al.*, 2011).

The SDN architecture consists of three main layers (or planes), and there is a communication between layers, as shown in the following figure:

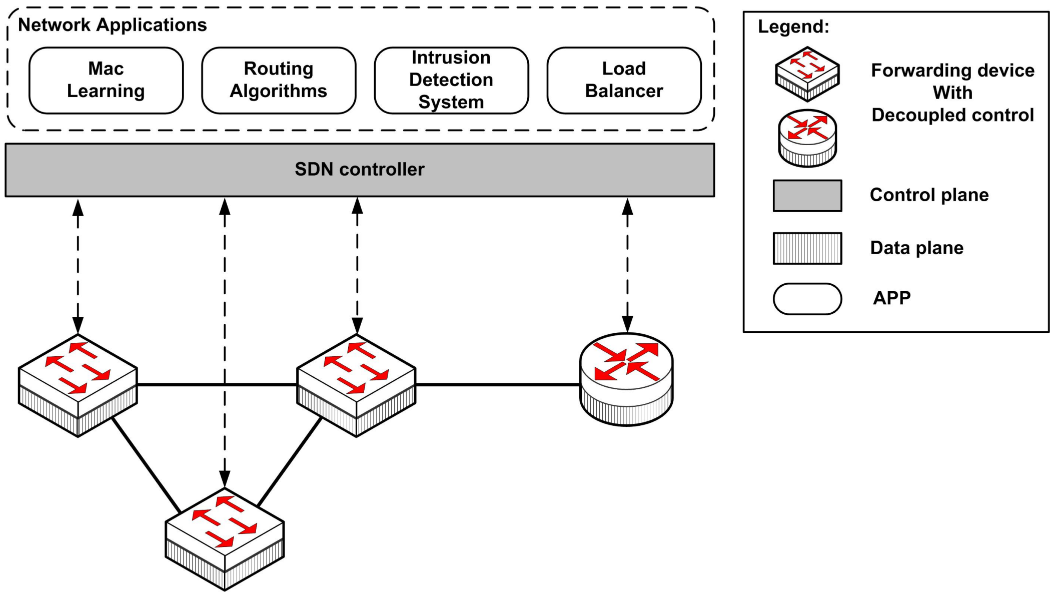


Figure 4 – SDN networking (with decoupled control). Source: (KREUTZ *et al.*, 2015)

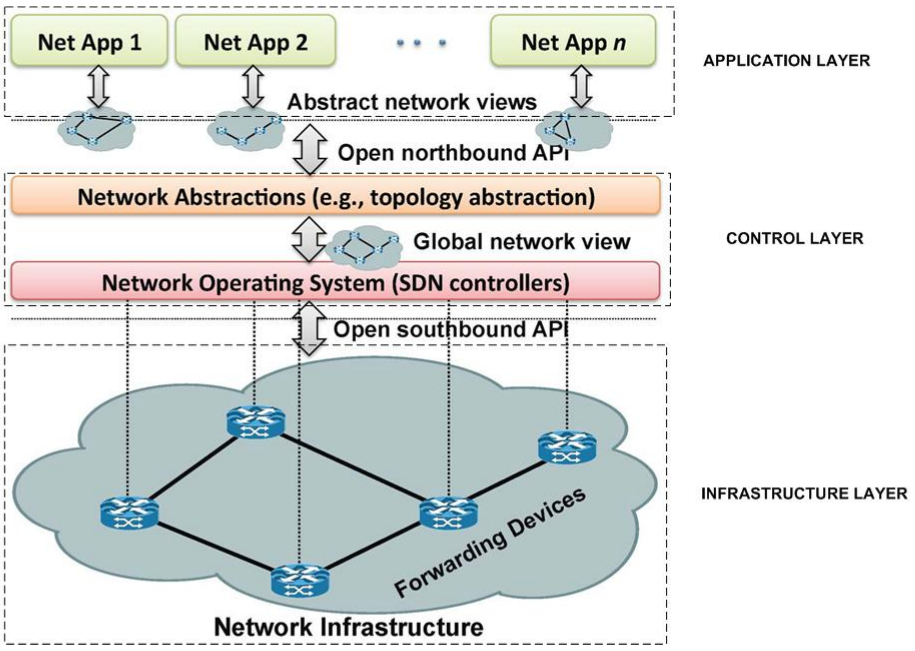


Figure 5 – SDN arquitecture. Source: (KREUTZ *et al.*, 2015)

Following is a description of the specific functions of each layer, namely an infrastructure layer, a control layer, and an application layer, stacking over each other (XIA *et al.*, 2015),(KREUTZ *et al.*, 2015).

A) *Infrastructure Layer*: It is called as the forwarding/data plane. This layer is built up from OpenFlow-based forwarding devices (e.g., switches, routers, etc.) and provides connectivity. The connections among forwarding devices are through different transmission media, such as wireless (e.g., 802.11 and 802.16), wired (e.g., Ethernet) and optical networks. The Forwarding Devices do not have an internal control logic. It is necessary to configure it via a Southbound API (e.g., OpenFlow protocol) with the control plane, which allows processing packets based on rules provided by a controller. Besides, each network device can store its current state temporarily (e.g., the port statistical information: the send and receive packet count, the send and receive byte count, etc.) in order to be "sent" to the controller.

B) *Control Layer*:

Also known as the control plane. Here is where the SDN controller resides (network intelligence), as illustrated in Figure 5. The SDN controller can provide the abstractions, essential services, and common APIs to developers. The software-based SDN controller for interacting with the infrastructure layer, via the southbound interface, where some specific functions are reporting network status and importing packet forwarding rules. Besides, the controller can communicate with the application layer via the northbound interface. It provides service access points in various forms, for example, an Application Programming Interface (API). The control layer allows a logically centralized and global view of the network, which provides a control platform in network management over traffic flows.

C) *Application Layer*: The top layer in the SDN architecture has one or more applications. Applications and services can control all the forwarding devices through an abstract view of the network via the Representational State Transfer (REST) API Functions. The REST API is used by the programmer to create their applications and services with the controller.

2.2.1 OpenFlow Protocol

The OpenFlow protocol started as academic research (MCKEOWN *et al.*, 2008). It is the most deployed interface on forwarding devices and has come under control and development by the Open Networking Foundation (ONF) (ONF, 2013). This protocol is helping the company to improve operational efficiency and save operational costs. In each version of the OpenFlow specification introduced new match fields including Ethernet, IPv4/v6, MPLS, TCP/UDP, etc., as detailed in Table 1.

In later years, the implementation of SDN with the OpenFlow protocol has gained significant interest in the industry, including equipment with support for the OpenFlow API. There is confusion in terms of SDN and OpenFlow, which is about whether they are synonymous. But they are unlike. SDN is the overlaying idea and concept, while OpenFlow is one of several alternative protocols that enables the use of SDN in a network (ANDERSSON; TERMANDER, 2015).

An OpenFlow switch is a software program or hardware device that supports the OpenFlow Protocol and uses the Software Defined Network (SDN) techniques to forward

Table 1 – Different match fields, statistics, and capabilities have been added on each OpenFlow Protocol Revision. Source: (KREUTZ *et al.*, 2015).

OpenFlow Version	**Match fields**	**Statistics**
v 1.0	Ingress Port Ethernet: src, dst, type, VLAN IPv4: src,dst, proto, ToS TCP/UDP: src port, dst port	Per table statistics Per flow statistics Per port statistics Per queue statistics
v 1.1	Metadata, SCTP, VLAN tagging MPLS: label, traffic class	Group statistics Action bucket statistics
v 1.2	OpenFlow Extensible Match (OXM) IPv6: src, dst, flow label, ICMPv6	
v 1.3	PBB, IPv6 Extension Headers	Per-flow meter Per-flow meter band
v 1.4	–	Optical port properties

packets in a network. The OpenFlow protocol defines a set of messages that can be exchanged between the SDN controller and switch, over a secure channel, as shown in Figure 6.

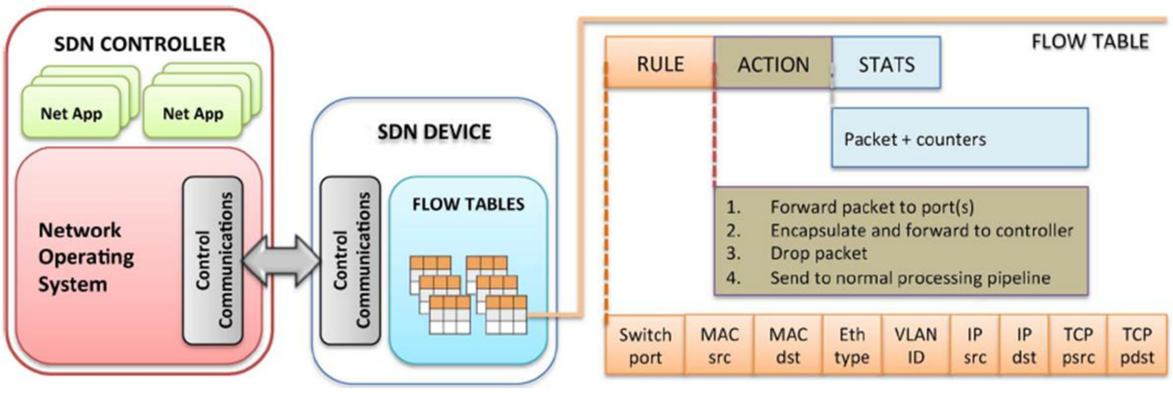


Figure 6 – OpenFlow-enabled SDN devices. Source: (KREUTZ *et al.*, 2015)

The forwarding device or OpenFlow switch supports one or more flow tables. OpenFlow switch maintains a flow table that contains a set of flow entries. Each flow entry determines how packets belonging to the flow will be processed and forwarded. Figure 7 illustrates the main components of a flow entry.

Match Fields	**Priority**	**Counters**	**Instructions**	**Timeouts**	**Cookie**

Figure 7 – Components of a flow entry in a flow table (OpenFlow). Source: (ONF, 2013).

As can be seen in Figure 7, each flow entry consists of:

- **Match fields or matching rules:** used to match incoming packets; match fields may contain information found in the packet header, ingress port, and metadata.

- **Priority:** matching precedence of the flow entry.
- **Counters:** used to collect statistics for the particular flow, such as number of received packets, number of bytes, and duration of the flow.
- **Instructions or actions:** to be applied upon a match; they dictate how to handle matching packets.
- **Timeouts:** maximum amount of time or idle time before the switch expires a flow.
- **Cookie:** opaque data value chosen by the Controller. May be used by the Controller to filter flow statistics, flow modification, and flow deletion. Not used when processing packets.

Upon a packet arrives at an OpenFlow switch. Packet match fields are extracted from the packet. For example, Ethernet address, IP address, Transmission Control Protocol (TCP) port, or a tag value (e.g., Multi-Protocol Label Switching (MPLS) tag). After, when a match is found in one of the flow tables, a corresponding action is performed, e.g., forward, push/pop tags, drop the packet, etc.

The meter and group table are also other tables in an OF enabled switch. The Meter tables are used to provide Quality of Service (QoS) functionality, such as rate-limiting. The Group tables are used to perform actions on a group type (ONF, 2013). For instance, to forward on the first live port in a group of ports (fast failover), or to forward on all ports in a group of ports (multicast).

An OpenFlow switch can be hardware-based or software-based:

- **Hardware-based:** Open network hardware platform based switches allow a vendor-independent. They are utilized to build networks (SDN prototypes) for research and classroom experiments since they are more flexible than the vendor?s switches and getting a better result with the performance metrics (higher throughput). For example, SDN switch implemented with NetFPGA based implementations such as SwitchBlade and ServerSwitch, and Advanced Telecommunications Computing Architecture (ATCA) based implementations such as OpenFlow Routers for Academic Networks (ORAN) (XIA *et al.*, 2015).
- **Software-based:** SDN switches can be performed as software running on a host operating system (OS), generally Linux. Software switches such as Open vSwitch (OVS) (PFAFF *et al.*, 2015), ofsoftswitch (FERNANDES; ROTHENBERG, 2014), and Pica8, can be installed in a general-purpose computer, adding switch functionality into it. A significant advantage of software implemented SDN switches is that they can provide virtual switching for virtual machines in the popular paradigm of server virtualization and cloud computing.

In OpenFlow networks, switches come in two varieties:

- **Pure:** OpenFlow switches have no legacy features or on-board control, and entirely rely on a controller for forwarding decisions.
- **Hybrid:** Switches support OpenFlow in addition to traditional operations and protocols.

2.2.2 SDN Controllers

The SDN controller is a software platform that provides the necessary resources and abstractions to expedite the programming of forwarding devices based on a logically centralized, such as communicates can be used to implement management tasks (flow table) and provide new functionalities, as depicted in Figure 6.

There are two principal ways in which a controller can add a new rule in the flow table of an OpenFlow switch:

1. **Proactively mode:** The flow tables are statically prepopulated, thereby reducing latency.
2. **Reactively mode:** The controller can dynamically insert entries in response to switch requests.

There are several open-source SDN controller frameworks available for a wide range of languages. The following is a list of open-source controller based on their programming language:

- **C**: Trema (also Ruby), and MUL
- **C++**: NOX (also Python).
- **Java**: Beacon, Floodlight, Open IRIS, Maestro, and OpenDaylight/Helium
- **Python**: POX, Pyretic, and RYU

2.2.2.1 OpenDaylight Controller

OpenDaylight (ODL) is an open-source software project that seeks to create an SDN controller that can be employed to accelerate the adoption of SDN. The OpenDaylight Controller is written in Java, runs within Java Virtual Machine (JVM) on the Open Service Gateway initiative (OSGi) framework. Hence, it can be utilized on any operating system platform that supports Java.

The controller uses these tools (SDX CENTRAL, 2018):

- **Maven:** It is a software tool used for the management and construction of Java projects. OpenDayLight uses Maven for more straightforward build automation.
- **Open Service Gateway Interface (OSGi):** for dynamically loading bundles and packaged JAR files.
- **Karaf:** is a small OSGi based runtime which provides a lightweight container for loading different modules.
- **Java interfaces:** for event listening, specifications, and forming patterns.

The OpenDaylight Controller supports external access to applications and data using following model-driven protocols:

- **NETCONF:** XML-based RPC protocol, which provides abilities for a client to invoke YANG-modeled RPCs, receive notifications, and read, modify and manipulate YANG modeled data.
- **RESTCONF:** HTTP-based protocol, which offers REST-like APIs to manage YANG modeled data and invoke YANG modeled RPCs, using XML or JSON as payload format.

The multiprotocol and modular (bundles) of ODL allow users can develop an SDN controller to fit their specific needs. This modular and multiprotocol approach gives IT admins the ability to pick a singular protocol or to select multiple protocols to resolve complex problems as they emerge. Each module is a service offered by the controller, and it is developed under a multi-vendor sub-project following the idea of SDN.

The OpenDaylight project is home to several sub-projects which combined, provide a vast array of features under a single package. Some examples of such projects are OpenFlow Plugin, Virtual Tenant Network (VTN), L2 Switch, and YANG Tools. These sub-projects are available in the Project section of the OpenDaylight Wiki[1]

OpenDaylight has REST APIs which are used by external applications. Applications that exist inside of OpenDaylight make use of the Service Abstraction Layer (SAL) to communicate with different types of devices. The platform includes support for various communication protocols, including OpenFlow, OVSDB, NETCONF, and BGP.

Figure 8 shows the architecture of the OpenDaylight release, Helium, its structure is constituted of several blocks (OpenDaylight Foundation, 2018):

The modules of ODL can be viewed in Figure 8 as green boxes above the Service Abstraction Layer (SAL) in the controller platform. Using the OSGi framework makes

[1] https://wiki.opendaylight.org/view/Project_list

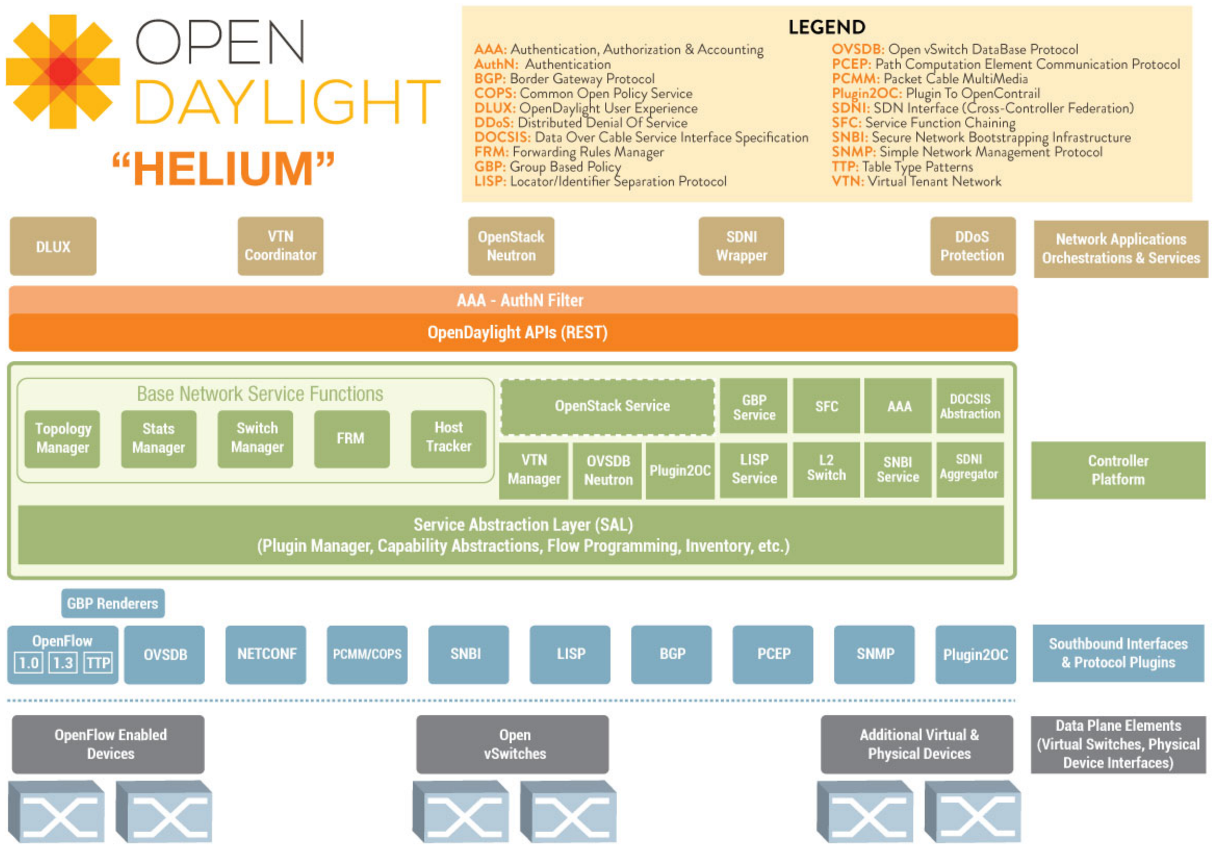

OPEN DAYLIGHT
"HELIUM"
LEGEND
AAA: Authentication, Authorization & Accounting
AuthN: Authentication
BGP: Border Gateway Protocol
COPS: Common Open Policy Service
DLUX: OpenDaylight User Experience
DDoS: Distributed Denial Of Service
DOCSIS: Data Over Cable Service Interface Specification
FRM: Forwarding Rules Manager
GBP: Group Based Policy
LISP: Locator/Identifier Separation Protocol
OVSDB: Open vSwitch DataBase Protocol
PCEP: Path Computation Element Communication Protocol
PCMM: Packet Cable MultiMedia
Plugin2OC: Plugin To OpenContrail
SDNI: SDN Interface (Cross-Controller Federation)
SFC: Service Function Chaining
SNBI: Secure Network Bootstrapping Infrastructure
SNMP: Simple Network Management Protocol
TTP: Table Type Patterns
VTN: Virtual Tenant Network
DLUX
VTN Coordinator
OpenStack Neutron
SDNI Wrapper
DDoS Protection
Network Applications Orchestrations & Services
AAA - AuthN Filter
OpenDaylight APIs (REST)
Base Network Service Functions
Topology Manager
Stats Manager
Switch Manager
FRM
Host Tracker
OpenStack Service
GBP Service
SFC
AAA
DOCSIS Abstraction
VTN Manager
OVSDB Neutron
Plugin2OC
LISP Service
L2 Switch
SNBI Service
SDNI Aggregator
Controller Platform
Service Abstraction Layer (SAL)
(Plugin Manager, Capability Abstractions, Flow Programming, Inventory, etc.)
GBP Renderers
OpenFlow 1.0 1.3 TTP
OVSDB
NETCONF
PCMM/COPS
SNBI
LISP
BGP
PCEP
SNMP
Plugin2OC
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 8 – Detailed architecture of the OpenDaylight Controller. Source: (OpenDaylight Foundation, 2018)

it possible to add, remove, and update bundles/modules without having to reboot the complete system. OpenDaylight exposes open Northbound APIs used by applications.

The Southbound protocol support multiple protocols (as plugins), such as OpenFlow version 1.0, OpenFlow version 1.3, and BGP-LS protocols. These modules are linked dynamically into a Service Abstraction Layer (SAL). The SAL separates southbound protocol plugins (SB) from northbound service/application plugins (NB). The SAL exposes the infrastructure layer to the applications north of it and determines how to fulfill the requested services independently of the underlying protocol used and the network devices.

The OpenFlow plugin project intends to develop a plugin to support implementations of the OpenFlow specification as it develops and evolves. Specifically, the project has developed a plugin aiming to support OpenFlow 1.0 and 1.3.x. It can be extended to add support for subsequent OpenFlow specifications. The plugin is based on the Model-Driven Service Abstraction Layer (MD-SAL) architecture.

The OpenFlow Plugin functionality can be cleanly divided into four areas, which have clearly-defined interactions. In the interest of providing naming continuity, we refer to them as Managers. The four managers are:

1. Connection Manager is responsible for early session negotiation, including determining switch features and identity.

2. Device Manager is responsible for handling low-level interactions with the switch. The Iterations allow us to ensure access to the commutator, populating inventory with physical properties (such as port status), tracking outstanding requests, and performing response dispatch to the proper requesting entity.

3. Statistics Manager is responsible for maintaining synchronization between the on Switch counters and their representation in MD-SAL datastore. This process is asynchronous to the normal processing, such that it can back off if the network cannot keep up the data rate needed to maintain polling frequencies requested by applications.

4. RPC Manager is responsible for routing application requests from MD-SAL towards the device. It performs translation between high-level semantic requests to low-level protocol messages, both in terms of data and in terms of the lifecycle.

OpenFlow plugin collects the following statistics from the OpenFlow enabled node (switch):

1. Individual Flow Statistics
2. Aggregate Flow Statistics
3. Flow Table Statistics
4. Port Statistics
5. Group Description
6. Group Statistics
7. Meter Configuration
8. Meter Statistics
9. Queue Statistics
10. Node Description
11. Flow Table Features
12. Port Description
13. Group Features

14. Meter Features

At a high-level statistics, collection mechanism is divided into the following three parts:

1. Statistics related YANG models, service APIs, and notification interfaces are defined in the MD-SAL.
2. Service APIs defined in yang models are implemented by the OpenFlow plugin. Notification interfaces are wired up by OpenFlow plugin to MD-SAL.
3. Statistics Manager Module: This module use service APIs implemented by the OpenFlow plugin to send statistics requests to all the connected OpenFlow enabled nodes. The module also implements notification interfaces to receive statistics response from nodes. Once it receives a statistics response, it augments all the statistics data to the relevant element of the node (like node-connector, flow, table, group, meter) and stores it in the MD-SAL operational data store.

In an SDN architecture, the Northbound Application Program Interfaces (APIs) are used to communicate between the SDN Controller and network applications. The Northbound APIs can be used to facilitate innovation and enable efficient orchestration and automation of the network to align with the needs of different applications (Apps) via SDN network programmability.

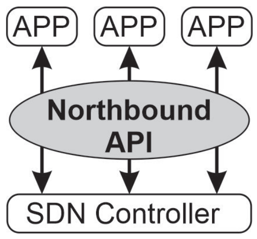


Figure 9 – A controller with the northbound APIs

2.2.3 Traffic Monitoring for SDN

Network traffic monitoring is the process of reviewing, analyzing, and managing network traffic that can affect network performance. For instance, link failure detection that causes a bottleneck leading to poor network performance.

Several different technologies have been employed to monitor network traffic. Technologies such as sFlow, Cisco NetFlow, Juniper J-Flow, NetStream, and IPFIX. They perform similar functions but are supported by different network equipment vendors.

- **NetFlow** is a network protocol developed by Cisco Systems. It is a Cisco proprietary technology for the collection of traffic data on switches/routers, which uses probe methods that are installed at switches as special modules to collect either complete or sampled traffic statistics and send them to a central collector.
- **sFlow** (SFLOW, 2003) is a flow sampling method developed by InMon, which uses time-based sampling for capturing traffic information.
- **JFlow** is a flow sampling method developed by the Juniper Networks. JFlow is quite similar to NetFlow. However, these approaches may be not efficient solutions to be applied in SDN systems, such as large-scale data center networks, because of the significantly increased overhead incurred by statistics collection from the whole network at the central controller.

2.2.3.1 sFlow Overview

In a data center, the efficient place to monitor traffic is within the switch/router. The sFlow is a packet sampling technology embedded inside switches and routers and does not impact forwarding or network performance. Figure 10 shows the growth of use over packet sampling technologies, which was available since 2001. The packet sampling with sFlow in high-speed networks has become recognized as the scalable, accurate, and general solution for network monitoring (SFLOW, 2003).

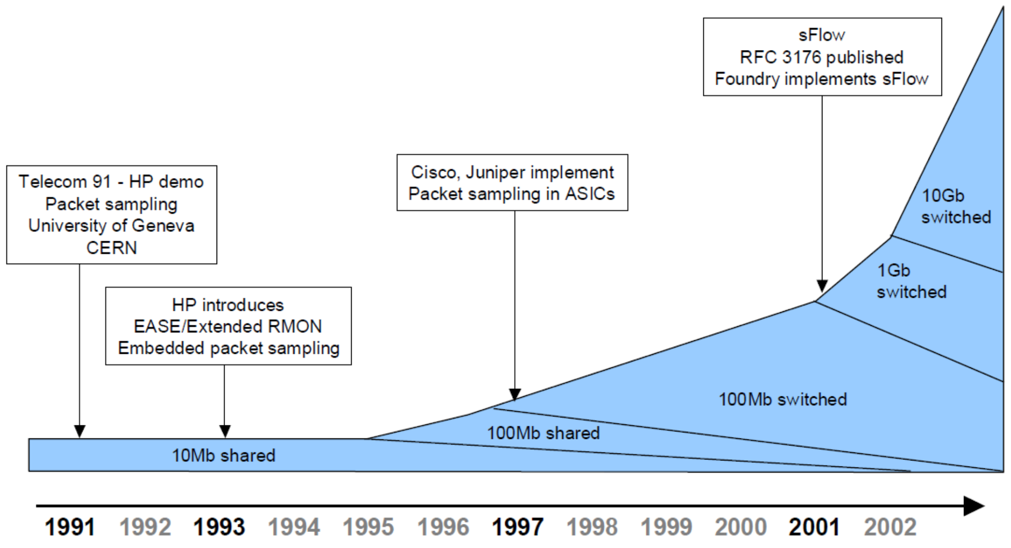


Figure 10 – Packet sampling timelines. Source: (SFLOW, 2003)

The sFlow monitoring system (see Figure 11) be composed of a sFlow Agent (embedded in a switch or router) and a central sFlow Collector:

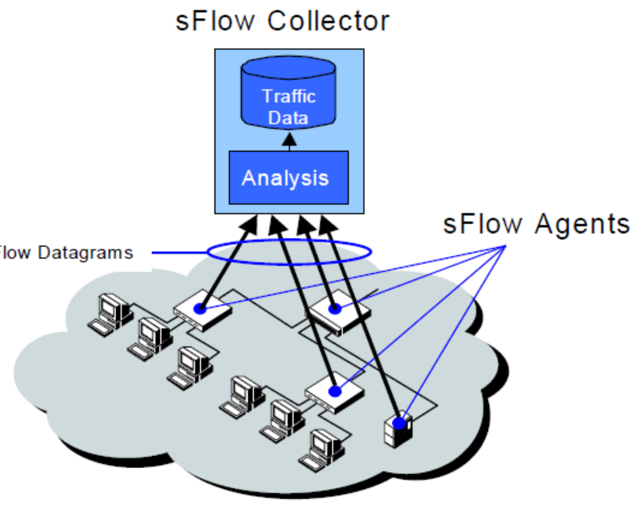


Figure 11 – sFlow Agent and Collector. Source: (SFLOW, 2003)

- **The sFlow Agent** uses sampling technology to capture traffic statistics from the device it is monitoring. sFlow Datagrams are used to immediately forward the sampled traffic statistics to a sFlow Collector for analysis.
- **The sFlow Collector** receives sFlow Datagrams from one or more sFlow Agents (see Figure 11). The sFlow Collector may also configure sFlow instances using the configuration mechanisms provided by the sFlow Agent.

The sFlow Agent is a software process that runs as part of the network management software within a device (see Figure 12). It combines interface counters and flow samples into sFlow datagrams that are sent across the network to a sFlow Collector. Packet sampling is typically performed by the switching/routing ASICs, providing wire-speed performance. The state of the forwarding/routing table entries associated with each sampled packet is also recorded.

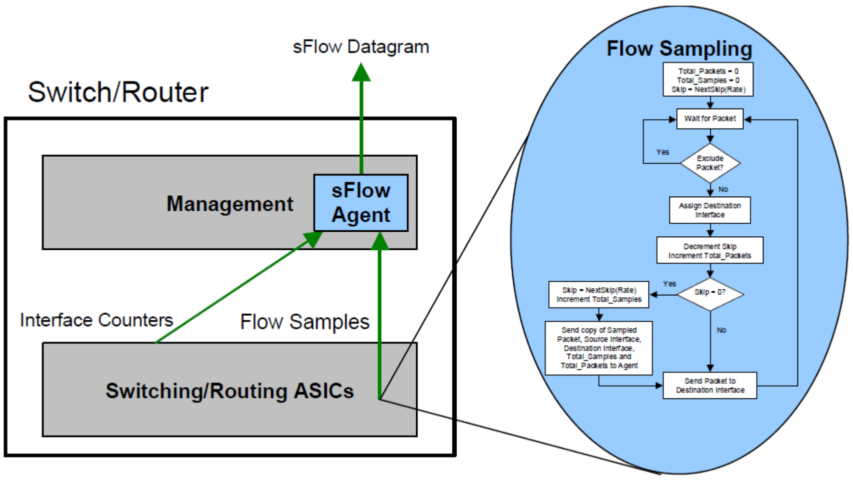


Figure 12 – sFlow Agent embedded in switch/router. Source: (SFLOW, 2003)

sFlow separates traffic analysis from traffic sampling, therefore the sampling logic is embedded in the network device (e.g., the router or switch), and traffic analysis is executed on a separate machine (typically a server). This allows a greater response capacity on a larger scale in real time. This design specifically addresses issues associated with:

- Accurately monitoring network traffic at Gigabit speeds and higher.
- Scaling to monitor tens of thousands of agents from a single sFlow Collector.
- Extremely low cost sFlow Agent implementation.

sFlow is a sampling technology that meets the key requirements for a network traffic monitoring solution (SFLOW, 2003):

- **sFlow provides a network-wide view** of usage and active routes. It is a scalable technique for measuring network traffic, collecting, storing, and analyzing traffic data. This enables tens of thousands of interfaces to be monitored from a single location.
- **sFlow is scalable**, enabling it to monitor links of speeds up to 10Gb/s and beyond without impacting the performance of core internet routers and switches, and without adding significant network load.
- **sFlow is a low cost solution.** It has been implemented on a wide range of devices, from simple L2 workgroup switches to high-end core routers, without requiring additional memory and CPU.
- **sFlow is an industry standard** with a growing number of vendors delivering products with sFlow support.

Accelerating adoption of virtual switching is helping to drive sFlow growth since support for the standard is integrated in virtual switches:

- Open vSwitch, integrated in the mainstream Linux kernel and an integral part of many commercial and open source virtualization platforms, including: VMware/Nicira NSX, OpenStack, Xen Cloud Platform, XenServer, and KVM.
- Hyper-V Virtual Switch, part of Window Server 2012
- IBM Distributed Virtual Switch 5000V
- HP FlexFabric Virtual Switch 5900v

2.2.3.2 The sFlow-RT Analytics

The sFlow-RT analytics engine collects a constant telemetry stream from sFlow Agents embedded in network devices, hosts, and applications and converts them into actionable metrics, accessible through APIs (The RESTflow). The RESTflow API makes it easy for SDN application developers to configure customized measurements, retrieve metrics, set thresholds, and receive notifications.

Figure 13 shows sFlow-RT's role that can be combined with a wide variety of on-site and cloud, orchestration, DevOps and Software-Defined Networking (SDN) tools.

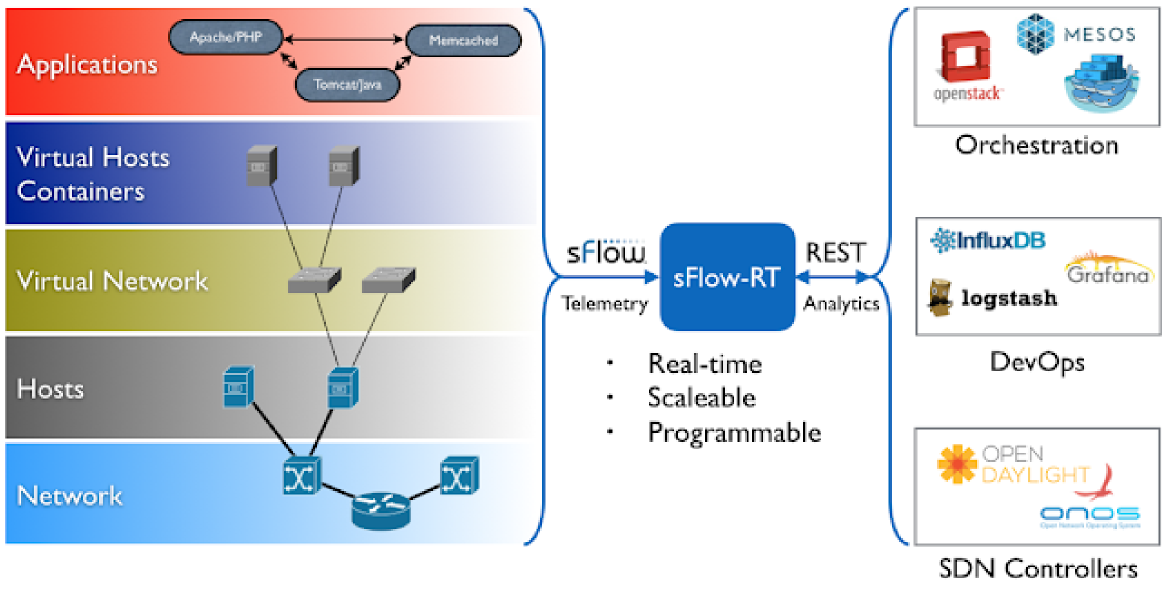


Figure 13 – The sFlow-RT analytics

Writing Applications [2] describes how to use sFlow-RT's APIs to extend or modify existing applications or develop new applications. For example, Arista eAPI describes how to automatically push controls based on flow measurements, describing automated DDoS mitigation as a use case. Other use cases include: traffic engineering, traffic accounting, anomaly detection, intrusion detection, targeted packet capture, etc.

2.2.4 Mininet

Mininet is an emulation environment that creates a realistic virtual network, running real kernel, switch and application code on a single machine (virtual machine (VM), cloud or native), in seconds. Mininet is useful for development, teaching, and research on OpenFlow and SDN.

[2] https://sflow-rt.com/writing_applications.php

2.2.4.1 Mininet Flow Analytics

This article [3] shows how standard sFlow instrumentation built into Mininet can be combined with sFlow-RT analytics software to provide real-time traffic visibility for Mininet networks. Augmenting Mininet with sFlow telemetry realistically emulates the instrumentation built into most vendor's switch hardware, provides visibility into Mininet experiments, and opens up new areas of research (e.g. SDN and large flows).

The following papers are a small selection of projects using sFlow-RT:

- Network-Wide Traffic Visibility in OF@TEIN SDN Testbed using sFlow (REHMAN *et al.*, 2014)
- OrchSec: An Orchestrator-Based Architecture For Enhancing Network-Security Using Network Monitoring And SDN Control Functions (ZAALOUK *et al.*, 2014)
- Utilizing OpenFlow and sFlow to Detect and Mitigate SYN Flooding Attack (NUGRAHA *et al.*, 2014)
- Implementation of Neural Switch using OpenFlow as Load Balancing Method in Data Center (RUELAS; ROTHENBERG, 2015)

2.3 Knowledge-Defined Networking

The research community has considered in the past the application of Artificial Intelligence techniques to control and operate networks. A notable example is the Knowledge Plane proposed by D.Clark et al. (CLARK *et al.*, 2003). However, such techniques have not been extensively prototyped. We describe a new paradigm that accommodates and exploits SDN, Network Analytic (NA) and Artificial Intelligence (AI) We refer to this new paradigm as Knowledge-Defined Networking (MESTRES *et al.*, 2016). The KDN paradigm operates by means of a control loop to provide automation, recommendation, optimization, validation and estimation. Figure 14 shows an overview of the KDN paradigm and its functional planes.

- **The Data Plane** is responsible for storing, forwarding and processing data packets.
- **The Control Plane** exchanges operational state in order to update the data plane matching and processing rules.
- **The Management Plane** ensures the correct operation and performance of the network in the long term. It defines the network topology and handles the provision and configuration of network devices.

[3] https://blog.sflow.com/2016/05/mininet-flow-analytics.html

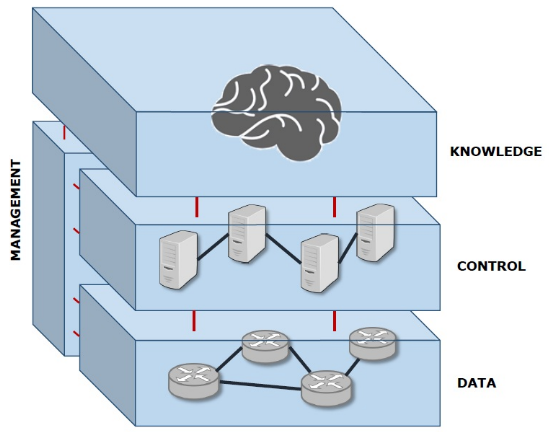


Figure 14 – KDN planes. Source: (MESTRES *et al.*, 2016)

- **The Knowledge Plane**, in the KDN paradigm, the KP takes advantage of the control and management planes to obtain a rich view and control over the network. It is responsible for learning the behavior of the network and, in some cases, automatically operate the network accordingly. While parsing the information and learning from it is typically a slow process, using such knowledge automatically can be done at a time-scales close to those of the control and management planes.

Figure 15 shows the basic steps of the KDN control. In what follows we describe these steps in detail.

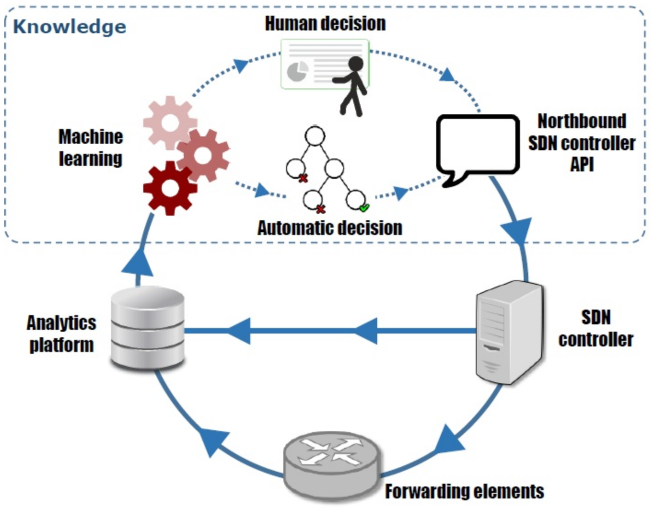


Figure 15 – KDN operational loop. Source: (MESTRES *et al.*, 2016)

1. **Forwarding Elements & SDN Controller → Analytics Platform** The Analytics Platform aims to gather enough information to offer a complete view of the network. To that end, it monitors the data plane elements in real time while they forward packets in order to access fine-grained traffic information.The most relevant data collected by the analytics platform is summarized below.

    - Network state
    - Control and management state
    - Service-level telemetry

2. **Analytics Platform → Machine Learning** ML algorithms (such as Deep Learning techniques) are the heart of the KP, which are able to learn from the network behavior. The current and historical data provided by the analytics platform are used to feed learning algorithms that learn from the network and generate knowledge (e.g., a model of the network). We consider three approaches:

    - **Supervised learning**, the KP learns a model that describes the behavior of the network, i.e., a function that relates relevant network variables to the operation of the network (e.g., the performance of the network as a function of the traffic load and network configuration).
    - **Unsupervised learning** is a data-driven knowledge discovery approach that can automatically infer a function that describes the structure of the analyzed data or can highlight correlations in the data that the network operator may be unaware of. As an example, the KP may be able to discover how the local weather affects the link?s utilization.
    - **Reinforcement learning** a software agent aims to discover which actions lead to an optimal configuration. As an example the network administrator can set a target policy, for instance the delay of a set of flows, then the agent acts on the SDN controller by changing the configuration and for each action receives a reward, which increases as the in-place policy gets closer to the target policy.

    Please note that learning can also happen offline and applied online. In this context knowledge can be learned offline training a neural network with datasets of the behavior of a large set of networks, then the resulting model can be applied online.

3. **Machine Learning → Northbound controller API** The KP eases the transition between telemetry data collected by the analytics platform and control specific actions.

    Depending on whether the network operator is involved or not in the decision making process, there are two different sets of applications for the KP. We next describe these potential applications and summarize them in Table 2.

Table 2 – KDN applications. Source: (MESTRES *et al.*, 2016)

	Close Loop	**Open Loop**
Supervised	Automation Optimization	Validation Estimation What-if analysis
Unsupervised	Improvement	Recommendation
Reinforcement	Automation Optimization	N/A

4. **Northbound controller API → SDN controller** The northbound controller API offers a common interface to, human, software-based network applications and policy makers to control the network elements. The API offered by the SDN controller can be either a traditional imperative language or a declarative one. In the latter case, the users of the API express their intentions towards the network, which then are translated into specific control directives.

5. **SDN controller → Forwarding Elements** The parsed control actions are pushed to the forwarding devices via the controller southbound protocols in order to program the data plane according to the decisions made at the KP.

2.4 Load-Balancing Methods

We define load-balancing as the distribution of traffic across multiple resources such as computers or network links. The goal of the load-balancing is to optimize some property of the network, such as minimizing response time, maximizing throughput, or maximizing the utilization of the network links.

In (QADIR *et al.*, 2015), load balancing and flow splitting approaches can be either static/quasi-static or dynamic/load-aware. The following Table 3 has summarized major load balancing works.

1. *Static/ Quasi-Static Load Balancing:* In these approaches, the traffic split is either static or is managed offline based on a predicted traffic demand matrix with changes occurring after a significantly long period of time.

2. *Dynamic (Load-Aware) Load Balancing:* In these approaches, the traffic split is dynamically adapted to the changes in observed load.

Table 3 – Summary of flow splitting/ Load balancing techniques. Source: (QADIR *et al.*, 2015)

Approach		Main Idea
Static/ Quasi-Static Approaches	Weighted round robin or Deficit round robin	Adopts an efficient fair queuing method to schedule contending traffic flows in an almost perfectly fair fashion with efficient running time and a complexity of *O(1)* per-packet processing time.
	ECMP	Divides traffic evenly over multiple equal-cost paths. There are three splitting methods in ECMP: 1) Per-flow hashing; 2) Per-packet round robin; and 3) Dividing destination prefixes among the available next hops in forwarding table.
	Valiant Load Balancing (VLB)	Performs a decentralized 2-phase routing technique, which is agnostic to the traffic matrix. In the first phase, VLB-independent of the destination-redirects traffic to arbitrary intermediate routers. The intermediate routers then forward the traffic to the destination in the second phase.
	OSPF-OMP	Extends OSPF, which can form multiple equal-cost paths and uses ECMP techniques to divide traffics equally over these paths, so that it utilizes opaque LSAs to distribute loading information to facilitate uneven splitting.
	MPLS Optimized Multipath (MPLS-OMP)	Proposes a TE procedure to achieve load balancing across multiple label-switched paths in MPLS. In MPLS-OMP, interior gateway protocol (IGP) floods traffic load information according to OSPF-OMP and ISIS-OMP specifications.
Dynamic (Load-Aware) Approaches	MPLS-based Adaptive Traffic Engineering (MATE)	Uses probing messages and explicit congestion notification packets to solve multipath routing which is formulated as optimization problem with the objective function of minimizing congestion on highly utilized links in the network.
	TeXCP	Uses explicit congestion notification packets to perform flow splitting in order to balance load in real-time manner in response to actual traffic demands/failures. It is an online distributed protocol.
	Common-case Optimization with Penalty Envelope (COPE)	Establishes optimal multiple paths efficiently for expected network scenarios while providing worst-case guarantees. It is a hybrid of oblivious routing and prediction-based optimal routing?a class of TE algorithms.
	Hedera	Provides a centralized load-aware scheduler for datacenters.
	MPTCP	Shifts traffic from one path to another; hence, it is a load balancing scheme for end hosts.

2.5 Related Work

Many methods and mechanisms have been proposed to implement load balancing in SDN environment using OpenFlow protocol (AKYILDIZ *et al.*, 2014), (ANDERSSON; TERMANDER, 2015),(AL-FARES *et al.*, 2010), (LI; PAN, 2013), (YANG *et al.*, 2014),(ROTSOS *et al.*, 2012). In Equal-Cost MultiPath (ECMP) (HOPPS, 2000) and Valiant Load Balance (VLB) (ZHANG-SHEN, 2010), the controller analyzes response information from OpenFlow switches and modify the flow-tables following specific load balancing strategy, in order to efficiently plan the data transmission path and achieve load balancing in SDN. However, these strategies belong to the static load balancing method unable to make a dynamic routing plan, according to real-time network load condition.

Al Sallami et al. (SALLAMI; ALOUSI, 2013) discussed a proposal to obtain an

ANN-based load balancing technique in cloud computing environments. The ANN predicts the distribution of the overload demand of each user. Then the resources are allocated according to the predicted demand to distribute equal load among all the servers. But the active servers at any given time depend on the demand of users for a specific time. As a result, busy servers are minimized, which leads to low energy consumption. These predictions are based on current workload metrics from the N servers. Metrics such as throughput, fault tolerance, response time, and resource utilization.

Semong et al. (SEMONG *et al.*, 2020) have presented a survey and research challenges summary of emulators/mathematical tools commonly used in the design of intelligent load balancing SDN algorithms. SDN allows load balancers to be programmable and offer flexibility to design and implement their load balancing methods. A dynamic load balancer optimizes network parameters such as latency, resource utilization, throughput, and fault tolerance with minimal power consumption. ANNs have no limits on input vectors and are described as flexible mathematical structures capable of pinpointing complex nonlinear connections between input and output data sets. We indicate some inputs used in ANNs, such as response time, packet loss, latency, bandwidth ratio, and hop. The neural model is found to be efficient, particularly in situations where the characteristics of the processes are not easy to explain using physical equations. Therefore, the SDN services that are integrated with artificial neural networks (ANNs) propose new load balancing techniques.

The work in Andersson et al. (ANDERSSON; TERMANDER, 2015) proposes a design of a dynamic load-balancer in the context of microwave mobile backhauls using a Performance Management (PM) to decrease the drop level. Thus, the PM measures end-to-end latencies and packet loss of all network paths to detect these capacity fluctuations. Then we combine the resulting performance knowledge with an SDN controller to reroute traffic around microwave links with reduced capacity. With the goals of load balancing and improving the overall performance.

Yang et al. (YANG *et al.*, 2014) implements an wildcard-base load balance mechanism on cloud services and it was used different data mining algorithms to predict future traffic. The implemntation of a prediction mechanism allows to dynamically update the wildcard rule on switches or routers, in tihs way, enhance load balancing all over the whole service. The features used to make the predictions: The first input is each IP's percentage of possession on the time of check, the second input is each IP's percentage of packets on the time of check, the third input is the occurrence time ratio for each IP, and the last input is about how often an IP appears. The output node is the probability of appearance for each IP. To determine the number of neurons in the hidden layer for the neuronal model (BPNN) depending on the training's response time, where the greater number of neurons in the hidden layer would cost more time to give the result, and if there were few neurons, the response error is greater. The K-Means classification is more inaccurate than the BPNN prediction mechanism for lower usage rate users and better than the BPNN mechanism for higher usage rate users.

The dynamic load balancing algorithm proposed in (DU; ZHUANG, 2015) is based on OpenFlow and sFlow protocol to distribute traffic among servers of the cluster efficiently. The algorithm makes decisions based on real-time traffic statistics obtained via the OpenFlow and sFlow protocol. These protocols allow us to collect network status and export them to the controller. Furthermore, the proposed load balancer proactively installs wildcard rules on the switches to direct requests for a large group of clients without involving the controller, which will reduce the number of rules and reduce the network latency. The proposed load balancing algorithm considers the load on the server and the performance differences between servers in the heterogeneous cluster and different types of client requests. Some of the parameters are load computation of the server, processing ability computation of the server, and server weight computation for the traffic distribution.

3 Load-balancing based on Artificial Neural Networks

This chapter contains a description of the different components to study load balancing based on Artificial Neural Networks. Let us begin by describing the general design after we provided further details on the implementation. We also present pseudocode to facilitate the understanding of the different parts and their interaction with each other. Throughout this chapter, we will describe the architecture, which will serve the experimental evaluation.

3.1 Concept

Traditional load balancing methods on datacenter networks use dedicated hardware devices to distribute network traffic across different server replicas. Although this approach achieves high performance in general, it is expensive and lacks flexibility in its configuration. Therefore, load-balancing methods based on an Artificial Neural Network (ANN), allows creating a dynamic control of the traffic flow from a source to destination through multiple paths of the data center.

The proposed solution is divided into 4 components:

a) OpenFlow-based data center

b) SDN controller

c) Data collection

d) Load balancing

3.2 Components

3.2.1 Implementation Choice

In order to test and evaluate the performance of the proposed model, this section describes the chosen implementation. A sketch of the implementation can be seen in Figure 16, and Table 4 lists the different versions and build numbers for the software and packages used.

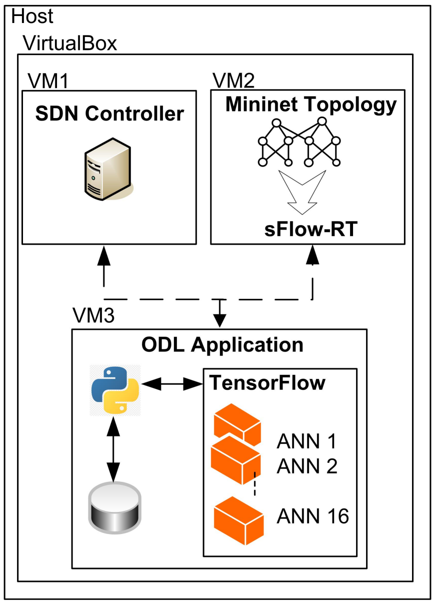


Figure 16 – Testbed

We have used three virtual machines. One will run the Mininet emulated network, next one the OpenDaylight controller, and the other will run the Tensorflow. We will connect the virtual machines (VMs) to a host-only network. Besides, they can communicate with each other, and with programs running on the host computer, such as *Secure Shell (SSH)* protocol and the *X11 client*.

We will use VirtualBox to run the Mininet VM. The Mininet project team[1] provides an Ubuntu 14.04 LTS VM image with Mininet 2.1.0, Wireshark, and OpenFlow dissector tools already installed and ready to use.

Table 4 – Software Versions used in implementation and experiments.

Item	**Version**
Open vSwitch in Mininet	2.0.0
Mininet	2.1.0
OpenDaylight Controller	Helium
OS VM1	Ubuntu 13.04
OS VM2	Ubuntu 13.04
OS VM3	Ubuntu 14.04
Python	3.05
Virtualization env.	VirtualBox
Client OS	Ubuntu 16.04
TensorFlow	2.00

[1] https://github.com/mininet/mininet/wiki/Mininet-VM-Images

3.2.2 OpenFlow-Based Data Center

We begin to create the fat-tree topology, using a network emulator, which creates a network of virtual hosts, switches, controllers, and links. We simulate the data center network in Mininet v2.1.0, which includes support for running Open vSwitch in user-space mode.

3.2.2.1 Topology

The Fat-Tree topology (AL-FARES *et al.*, 2008), (CHO *et al.*, 2001) contains multiple paths among hosts. Thus, it can provide higher available bandwidth and fault-tolerant networking. It is typically a 3-layer hierarchical tree that consists of switches on the core, aggregation, and edge layers. The switches in each pod have two types: edge switches on the bottom, aggregation switches on the medium. There are *k* pods, each containing two layers of *k/2* switches. Each *k*-port switch in the lower layer is directly connected to *k/2* hosts. Each of the remaining *k/2* ports is connected to *k/2* of the *k* ports in the aggregation layer of the hierarchy. Figure 3.2.2.1 shows the emulated 4-ary fat-tree topology as a custom topology in Mininet using the mininet.topo API.

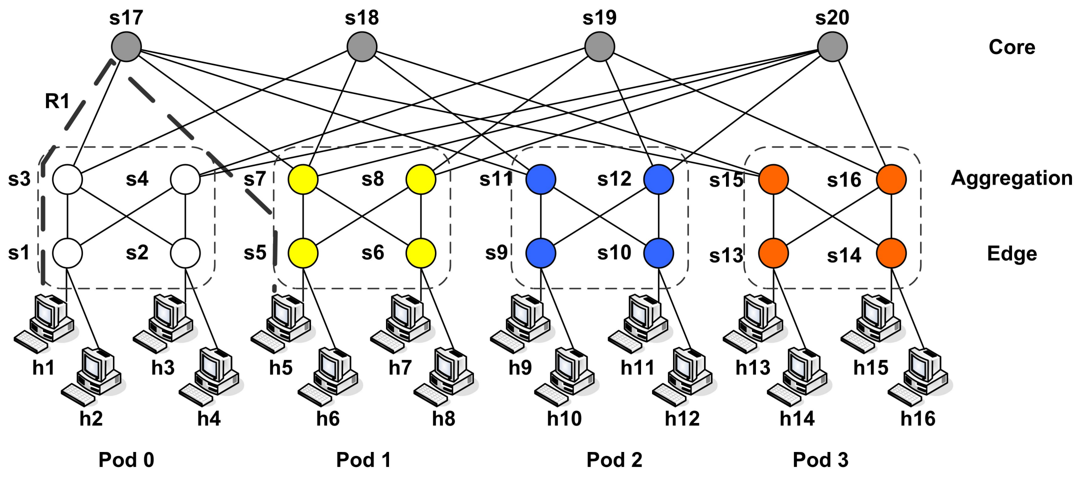


Figure 17 – Experimental data center topology. Symple k-ary Fat-Tree Topology(k = 4) . Source: (AL-FARES *et al.*, 2008)

The fat-tree topology has the following parameters: 4 pods, 8 edge switches, 8 aggregation switches, 4 core switches, and 16 hosts. Table 5 shows the name of the switches and hosts to create the simulation used with the source code of Appendix A.

Figure 3.2.2.1 shows a multiple-path network topology. Dijkstra's algorithm is implemented in order to find multiple paths of same length. This topology contains four paths from the source to the destination node. Example of *Pod 0 to Pod 1*:

Table 5 – List of the switches and hosts.

	Switch				**Host**			
Pod 0	s1	0_0_1	s3	0_2_1	h1	0_0_2	h3	0_1_2
	s2	0_1_1	s4	0_3_1	h2	0_0_3	h4	0_1_3
Pod 1	s5	1_0_1	s7	1_2_1	h5	1_0_2	h7	1_1_2
	s6	1_1_1	s8	1_3_1	h6	1_0_3	h8	1_1_3
Pod 2	s9	2_0_1	s11	2_2_1	h9	2_0_2	h11	2_1_2
	s10	2_1_1	s12	2_3_1	h10	2_0_3	h12	2_1_3
Pod 3	s13	3_0_1	s15	3_2_1	h13	3_0_2	h15	3_1_2
	s14	3_1_1	s16	3_3_1	h14	3_0_3	h16	3_1_3
Core	s17	4_1_1	s19	4_2_1				
	s18	4_1_2	s20	4_2_2				

Table 6 – The shortest paths between h1 and h5.

Host pairs	**Paths (switch-port)**
Pod 0: **h1** - Pod 1: **h5**	R1: s1-eth1, s3-eth1, s17-eth2, s7-eth2, s5-eth2
	R2: s1-eth1, s3-eth3, s18-eth2, s7-eth2, s5-eth2
	R3: s1-eth3, s4-eth1, s19-eth2, s8-eth2, s5-eth2
	R4: s1-eth3, s4-eth3, s20-eth2, s8-eth2, s5-eth2

3.2.3 SDN Controller

We assume an SDN control platform that could be implemented as a distributed system, obtaining the global view of the SDN network and discover all paths between the network device. The SDN controller uses the OpenFlow protocol as an open standard to communicate control decisions to data plane devices.

OpenDaylight Helium was chosen as the SDN controller for use in the testbed implementation. The controller was installed in a separate Linux Virtual Machine (VM) to not interfere with the network emulation in any way. Specific instructions on how to install the ODL controller can be found on the wiki page of the ODL project [2].

3.2.3.1 Installing OpenDaylight Features

Karaf is a modern and polymorphic container that allows the developers to put all the required software in a single distribution folder. These facilitate the installation or re-install OpenDaylight when needed because everything is in one folder. Furthermore, Karaf allows programs to be bundled with optional modules that can be installed when required.

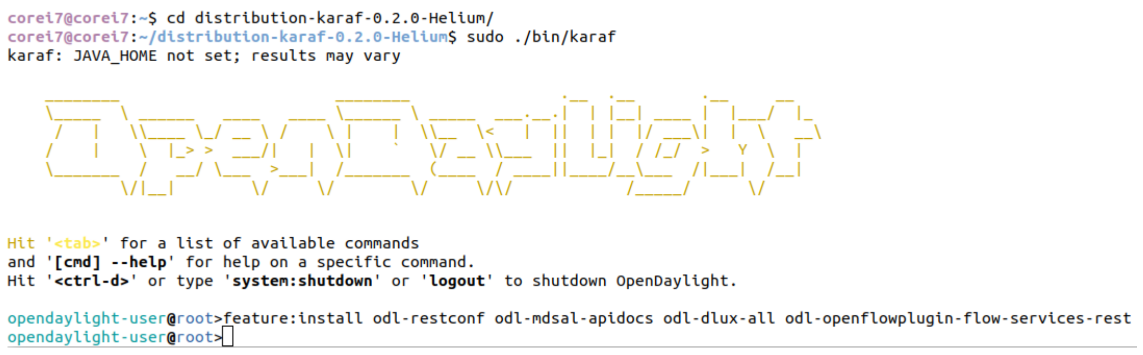
To run and use the OpenDaylight controller with the implementation, run the following command.

[2] https://wiki.opendaylight.org/view/OpenDaylight_Controller:Installation.html

```
>  cd distribution-karaf-0.2.0-Helium
> ./bin/karaf
```

Listing 1 – Run the ODL controller

The above command starts all the OSGi bundles installed as jar files in the plugins directory. After the ODL console is opened, we will install the karaf features:



```
corei7@corei7:~$ cd distribution-karaf-0.2.0-Helium/
corei7@corei7:~/distribution-karaf-0.2.0-Helium$ sudo ./bin/karaf
karaf: JAVA_HOME not set; results may vary

    ________                       ________                .__  .__       .__     __
    \_____  \ ______   ____   ____ \______ \ _____  ___.__.|  | |__| ____ |  |___/  |_
     /   |   \\____ \_/ __ \ /    \ |    |  \\__  \<   |  ||  | |  |/ ___\|  |  \   __\
    /    |    \  |_> >  ___/|   |  \|    `   \/ __ \\___  ||  |_|  / /_/  >   Y  \  |
    \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|___|  /__|
            \/|__|        \/     \/        \/     \/\/            /_____/      \/


Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>feature:install odl-restconf odl-mdsal-apidocs odl-dlux-all odl-openflowplugin-flow-services-rest
opendaylight-user@root>
```

Figure 18 – Karaf used to run ODL

The Karaf distribution has no features enabled by default. However, all of the features are available to be installed. By installing the features with the *feature:install* command on the ODL console. We entered the following features:

```
>feature:install odl-restconf odl-apidocs odl-dlux-all
odl-openflowplugins-flow-services-rest
```

Listing 2 – Installing multiple features

Once installed, these features are permanently added to the controller and will run every time it starts. We describe the installed features in this work:

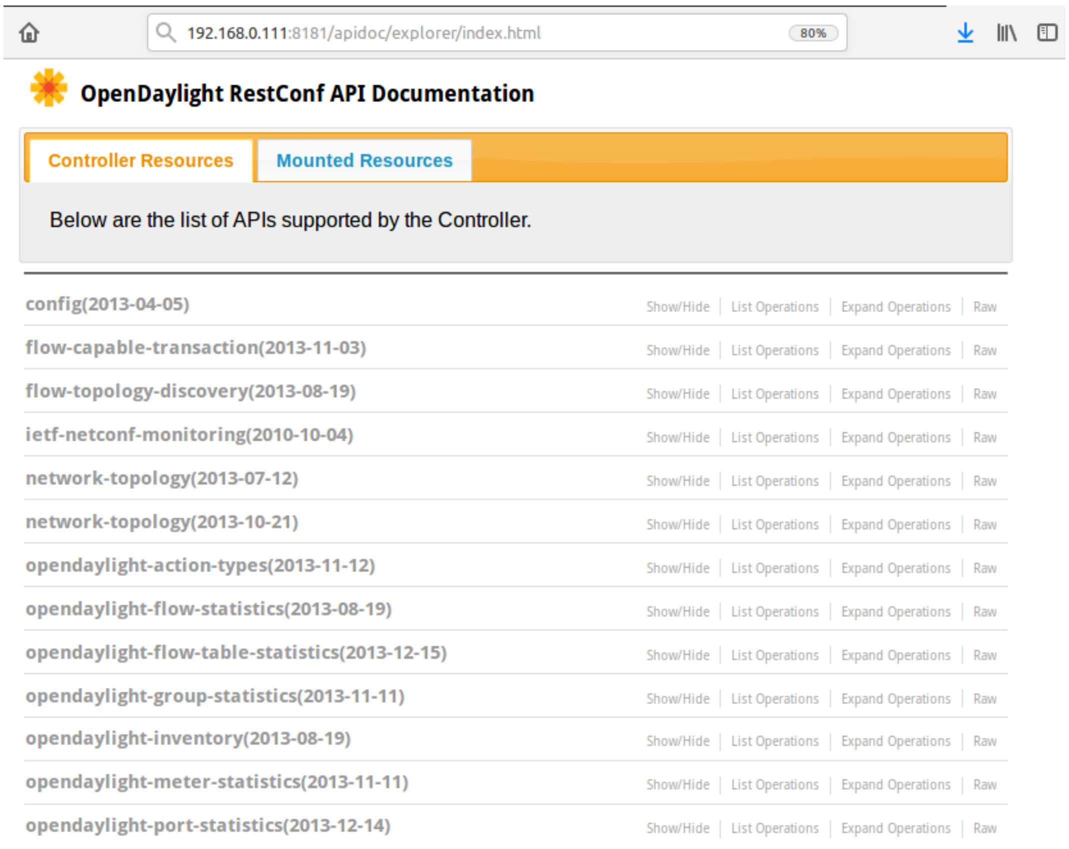
- **odl-restconf**: Allows access to RESTCONF API
- **odl-mdsal-apidocs**: Allows access to Yang API
- **odl-dlux-all**: OpenDaylight graphical user interface
- **odl-openflowplugins-flow-services-rest**: Wrapper feature for standard applications with REST interface

3.2.3.2 Northbound REST APIs

The Helium distribution has a chart user interface in the form of a web, that can be accessed after installing the feature *odl-mdsal-apidocs*, and entering the address:

```
http://<controller-ip>:8181/apidoc/explorer/index.html
```

Figure 19 shows the access from the machine with the OpenDaylight installed, where we can observe some installed plugins that allow getting data from the switches connected to the SDN controller.


192.168.0.111:8181/apidoc/explorer/index.html

OpenDaylight RestConf API Documentation

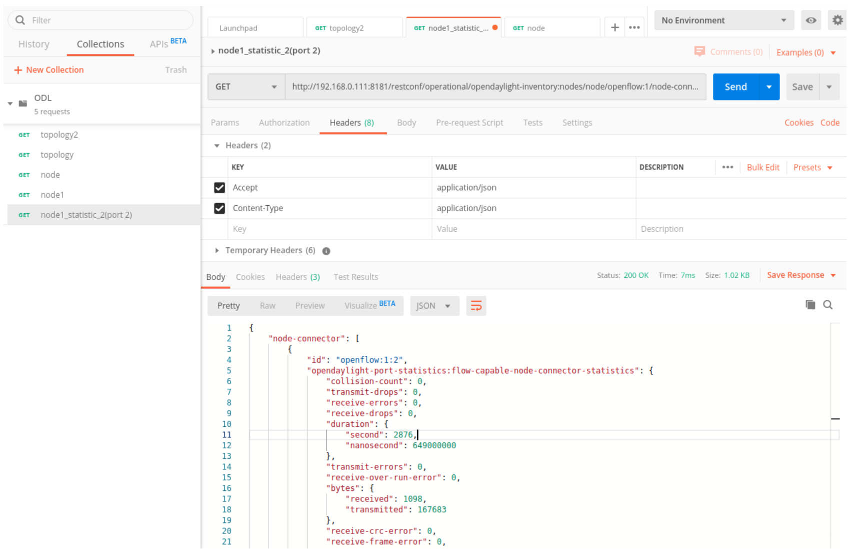
Controller Resources | Mounted Resources

Below are the list of APIs supported by the Controller.

config(2013-04-05)	Show/Hide	List Operations	Expand Operations	Raw
flow-capable-transaction(2013-11-03)	Show/Hide	List Operations	Expand Operations	Raw
flow-topology-discovery(2013-08-19)	Show/Hide	List Operations	Expand Operations	Raw
ietf-netconf-monitoring(2010-10-04)	Show/Hide	List Operations	Expand Operations	Raw
network-topology(2013-07-12)	Show/Hide	List Operations	Expand Operations	Raw
network-topology(2013-10-21)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-action-types(2013-11-12)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-flow-statistics(2013-08-19)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-flow-table-statistics(2013-12-15)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-group-statistics(2013-11-11)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-inventory(2013-08-19)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-meter-statistics(2013-11-11)	Show/Hide	List Operations	Expand Operations	Raw
opendaylight-port-statistics(2013-12-14)	Show/Hide	List Operations	Expand Operations	Raw

Figure 19 – REST API explorer seen in a web browser

Each element of the list is a different set of instructions of the REST API, that can be explored using *Postman*. Postman is a collaboration platform for API development. It offers a very intuitive GUI and has some interesting features, such as the possibility to save groups of commands into collections.



node1_statistic_2(port 2)

GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/node/openflow:1/node-conn...

Headers (2)

KEY	VALUE	DESCRIPTION
✓ Accept	application/json	
✓ Content-Type	application/json	
Key	Value	Description

Status: 200 OK Time: 7ms Size: 1.02 KB

```
{
    "node-connector": [
        {
            "id": "openflow:1:2",
            "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
                "collision-count": 0,
                "transmit-drops": 0,
                "receive-errors": 0,
                "receive-drops": 0,
                "duration": {
                    "second": 2876,
                    "nanosecond": 649000000
                },
                "transmit-errors": 0,
                "receive-over-run-error": 0,
                "bytes": {
                    "received": 1098,
                    "transmitted": 167683
                },
                "receive-crc-error": 0,
                "receive-frame-error": 0,
```

Figure 20 – Postman overview

To interact with any REST API, there are three elements that we may have to provide:

- Headers: They can specify options such as the language in which we have written the body or we want to receive the response.
- Address: It will specify which is the command we are trying to send. The most common methods are GET, POST, PUT, and DELETE.
- Body: For certain kind of instructions such as GET, we need to provide the data.

3.2.3.3 Implementation of the Northbound Interface

The Northbound REST APIs (also called RESTCONF) are easy to access through HTTP requests from most programming languages or applications. Most of the bundles in the controller have Northbound REST APIs implemented. They give an essential functionality for communicating with the controller. A full reference for the different REST APIs can be found in Seccion 2.2.2.1.

Details of statistics collection:

- The implementation collects the statistics mentioned in Section 3.2.4 at a periodic interval of 15 seconds.
- Whenever any new element is added to the node (like flow, group, meter, and queue), it sends statistics to request immediately to fetch the latest statistics and store it in the operational data store.

- Statistics Manager stores flow statistics as unaccounted flow statistics in a functional data store if there is no corresponding flow exist in the configuration data store. ID format of unaccounted flow statistics is as follows :

  *[#UF$TABLE*<table-id>*Unaccounted-flow-count - e.g #UF$TABLE*2*1]*

- Statistics Manager only entertains statistics response for the request sent by itself. Users can write their own statistics collector using the statistics service APIs and notification defined in yang models. It won't affect the functioning of the Statistics Manager.

RESTCONF to access statistics of various node elements:

1. Access the Controller GUI using the following URL.

```
http://<controller-ip>:8181/dlux/index.html
```

2. Use RESTCONF to see the topology information.

```
GET http://<controller-ip>:8181/restconf/operational/network-topology:network-
topology/
```

3. Aggregate Flow Statistics & Flow Table Statistics.

```
GET http://<controller-ip>:8181/restconf/operational/opendaylight-inventory:nodes/
node/{node-id}/table/{table-id}
```

4. Node Connector Statistics.

```
GET  http://<controller-ip>:8080/restconf/operational/opendaylight-inventory:nodes
/node/{node-id}/node-connector/{node-connector-id}
```

3.2.3.4 Flow Rule Example

The flow example was tested to work with Open vSwitch. To add flow rules on Open vSwitch via the RESTCONF. The knowledge plan will create flow rules and send these to the controller using a PUT request. After the SDN controller will push the flow rules to the corresponding network devices (switches). We use the POSTMAN with the following parameters:

Headers:

- **Content-type:** application/xml
- **Accept:** application/xml
- **Authentication:** admin:admin

URL:

```
http://<ctrl-addr>:8181/restconf/config/opendaylight-inventory:nodes/node/<Node-id>
  /table/<Table-#>/flow/<Flow-#>
```

Method: PUT

For example:

```
PUT http://<ctrl-addr>:8181/restconf/config/opendaylight-inventory:nodes/node/<Node-id>
  /table/<Table-#>/flow/<Flow-#>
```

Listing 3 – Set up the POSTMAN

This example programs a flow that matches IPv4 packets (ethertype 0x800) with a source address (*msg_src[host]*) and destination address (*msg_dst[host]*). After sending them to port *msg_src[out]*. The flow rule is installed in table 0 of the switch with datapath ID *id_flow*. The flow rule created contains the following body:

In Listing 4 we see that:

a) Priority number for that flow, where 2 is a number between 1 and 500.

b) Unique name of flow on that switch.

c) ID of flow on that switch.

d) ID of the table in the pipeline that stores that flow.

e) Start of action fields.

f) A forward action to output port.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
<priority>2</priority> /* (a) */
<flow-name>Load Balance ' +str(id_flow)+ '</flow-name> /* (b) */
  <match>
        <ipv4-destination>'+ msg_src['host'] +'</ipv4-destination>
        <ipv4-source>'+ msg_dst['host'] +'</ipv4-source>
        <ethernet-match>
            <ethernet-type>
                <type>2048</type>
            </ethernet-type>
        </ethernet-match>
    </match>
    <id>'+ str(id_flow) +'</id> /* (c) */
    <table_id>0</table_id> /* (d) */
    <instructions> /* (e) */
        <instruction>
            <order>0</order>
            <apply-actions>
                <action>
                    <order>0</order>
                    <output-action> /* (f) */
                     <output-node-connector>'+ msg_src['out'] +'</output-node-connector>
                    </output-action>
                </action>
            </apply-actions>
        </instruction>
    </instructions>
</flow>
```

Listing 4 – Flow rule in XML format

3.2.4 Data Collection: Data Gathering and Processing

This process consists in to collect the metrics (bandwidth and transmission latency) of each link by the sFlow-RT network analyzer and the SDN controller (see Figure 24). These data are collected periodically and stored in a database.

3.2.4.1 Path Features Extraction Using sFlow-RT

The sFlow agent uses the statistical packet-based sampling of switched packet flows to capture traffic statistics from the switch. Thus, the traffic can be accurately identified and monitored (PHAAL; LEVINE, 2004). In order to acquire the load condition of each path, sFlow-RT is used to obtain the link load condition between two switches. We implemented the sFlow-RT application "Real-time network weather map example"[3] to estimate the available bandwidth in each network link.

3.2.4.2 Bandwitdh

Bandwidth (BW) reflects the load condition over each link. sFlow-RT measures the BW of each connection then sends this data using the REST API to the ANN module. We can calculate the bits per second from every port by taking successive sFlow counter

[3] https://github.com/sflow-rt/svg-weather

samples and subtracting the values of *ifinoctets*/*ifoutoctets* counters using the following Equation (3.1) :

$$BW = \frac{(ifoutoctets - ifinoctets) * 8}{t} \tag{3.1}$$

where t is the time between the collect samples made by the counter samples.

To measure bandwidth between nearby neighbor switches, we implement the following applications for the sFlow-RT:

a) **The sFlow-RT: Configuration in Open vSwitch**

To configure the sFlow-RT, we implement the *Large flow detection* [4] that measures the links between neighboring switches with link bandwidths of 10 Mbit/s and allows reading measurements in JSON format using REST API.

The following command configures the sFlow-RT on the virtual switch (Open vSwitch) with a 1-in-10 sampling probability and a 20 second counter export interval:

```
    ovs-vsctl -- -- id=@sflow create sflow agent=eth0 target=127.0.0.1  \
    sampling=10 polling=20 --  \
    -- set bridge 0_0_1 sflow=$@$sflow  \
    -- set bridge 0_1_1 sflow=$@$sflow  \
    -- set bridge 0_2_1 sflow=$@$sflow  \
    ...
    -- set bridge 4_2_2 sflow=$@$sflow
```

Listing 5 – Configuration sFlow-RT

Figure 21 shows a second-by-second representation of the stepped test pattern that was used to evaluate the responsiveness of the detection script. The test consisted of 20 second constant rate traffic flows ranging from 2 Mbit/s to 10 Mbit/s, representing 20% to 100% of the link bandwidth.

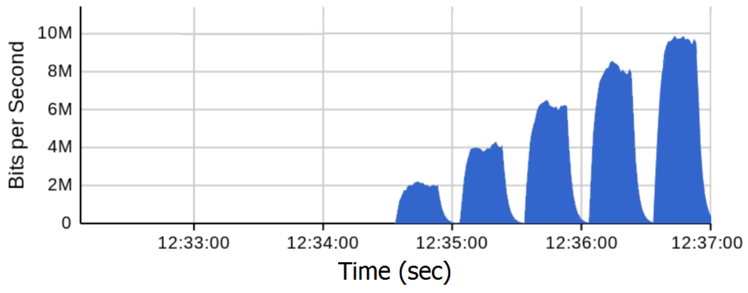


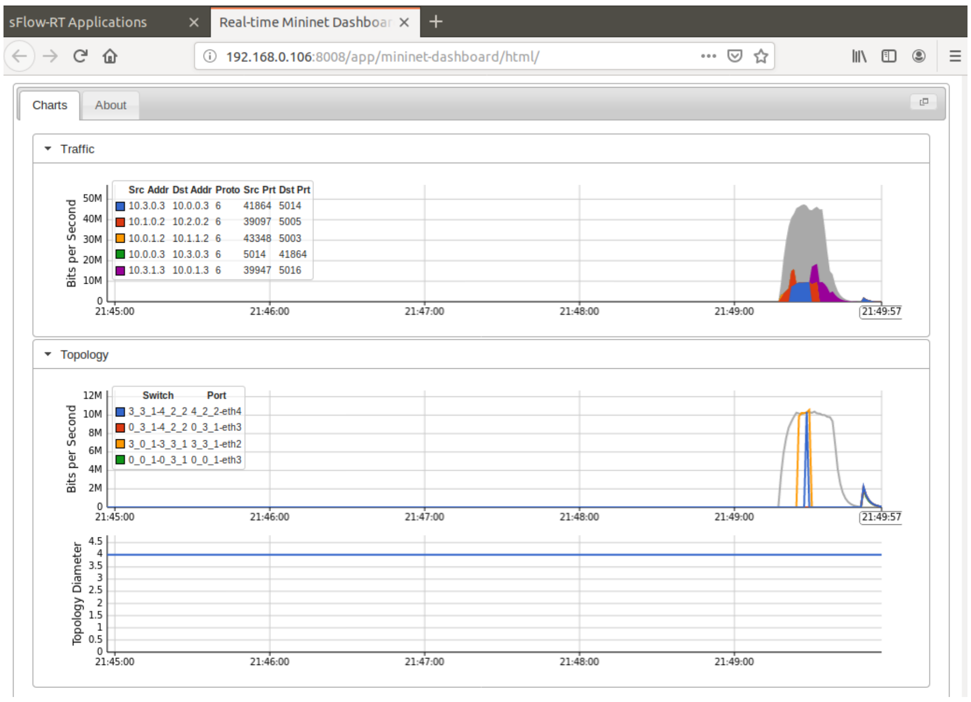
Figure 21 – Test of configuration in sFlow program.

[4] https://blog.sflow.com/2013/06/large-flow-detection.html

b) **sFlow-RT: Measurements in JSON format**

We implement Mininet Dashboard [5] that is a real-time dashboard displaying traffic information from Mininet virtual networks. Figure 22 shows the real-time bandwidth measurements by IP address and output port.

The dashboard (see Figure 22) has three time series charts that update every second and show five minutes worth of data. Switch to the Charts tab to see traffic trend charts. In this case, the trend charts show the results of five iperf tests. The Traffic chart shows the top flows and the Topology charts show the busy links and the network diameter.



	Src Addr	Dst Addr	Proto	Src Prt	Dst Prt
🟦	10.3.0.3	10.0.0.3	6	41864	5014
🟥	10.1.0.2	10.2.0.2	6	39097	5005
🟧	10.0.1.2	10.1.1.2	6	43348	5003
🟩	10.0.0.3	10.3.0.3	6	5014	41864
🟪	10.3.1.3	10.0.1.3	6	39947	5016

	Switch	Port
🟦	3_3_1-4_2_2	4_2_2-eth4
🟥	0_3_1-4_2_2	0_3_1-eth3
🟧	3_0_1-3_3_1	3_3_1-eth2
🟩	0_0_1-0_3_1	0_0_1-eth3

Figure 22 – Dashboard displaying real-time traffic information from Mininet.

To get the measurements in JSON format, we implement *Real-time network weather map example* [6], performing the following modifications:

- We have created an SVG image using the diagram of Figure 3.2.2.1 after it is stored in path: *app/svg-weather/html/map.svg*

[5] https://github.com/sflow-rt/mininet-dashboard
[6] https://github.com/sflow-rt/svg-weather

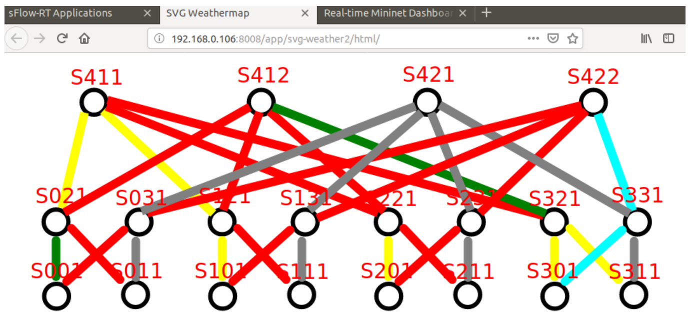


Figure 23 – sflow-RT in operation.

In Figure 23, the links between switches have different colors due to the bandwidth measurement. If it is less than 0.5 Mbps the line color is gray; if it is less than 2 Mbps the line color is blue. The following code was made to get the color of the links:

```
function linkProperties(utilization) {
  var color;
  if(utilization === -1) color = 'gray';
  else if(utilization < 500000) color = 'gray';
  else if(utilization < 2000000) color = 'blue';
  else if(utilization < 4000000) color = 'cyan';
  else if(utilization < 6000000) color = 'green';
  else if(utilization < 8000000) color = 'yellow';
  else color = 'red';
```

Listing 6 – Assign link properties based on utilization the bandwidth

- We have modified the following code of the *sflow-rt/app/svg-weather/scripts/status.js* in the variable *links*, see Appendix C.1. This code we let know the bandwidth measurements in the JSON format, for each ingress and egress port:

```
{
 "links": {
 "s001-eth2": {    /* (a) */
   "color": "red", /* (b) */
   "width": 10,    /* (c) */
   "meter": 9829733.170511438 /* (d) */
  },
  "s001-eth4": {
   "color": "yellow",
   "width": 10,
   "meter": 6997028.101058965
  },
  "s011-eth2": {
   "color": "yellow",
   "width": 10,
   "meter": 6641743.560652587
   },
  ...
  "s311-eth4": {
   "color": "yellow",
   "width": 10,
   "meter": 377.75578773341715
   }
```

Listing 7 – Bandwidth measurement in JSON format

In Listing 7 we see that:

a) Indicates the bandwidth measurement on the switch and its output port for transmitting data. Measurement belongs to switch s1-eth2.

b) According to Listing 6, it is red if the bandwidth is less than 8 Mbps.

c) Line width (links between switches).

d) The value of the bandwidth measurement.

3.2.4.3 Latency

Latency (Latency) is the time spent by host switch on data transmission required to push all the packet's bytes into the wire until they are received by the end system.

The transmission latency can indicate the congestion status of a link and the load situation of the switch in some way. The SDN controller can collect the transmitted bytes Num_Byte in this period and the transmission rate tx_{Rate} at corresponding OpenFlow switches port. Then, the transmission latency can be calculated by Equation (3.2):

$$Latency = \frac{Num\_Byte}{tx_{Rate}} \tag{3.2}$$

For several links L_1, L_2,..., L_n with its transmission latency $Latency_1$, $Latency_2$,..., $Latency_n$, respectively, then the total latency of this path is as follows:

$$Latency\_path_i = \frac{1}{n}\sum_{i=1}^{n} Latency_i \tag{3.3}$$

To calculate the latency of a link between the switches/routers. We will use Equation 3.3. Thus, it is necessary to know the transmitted bytes and the transmission rate where the number of bytes sent is provided by the SDN controller using the REST API (RESTCONF) function and the transmission rate by the sFlow-RT.

For example, we measure the Latency in path Pod 0: h1 - Pod 1: h5 using route R1 (see Figure 21). Follow the following steps:

- The switches that we have in the route R1: s1 (openflow: 1), s3 (openflow: 513), s17 (openflow: 262401), s7 (openflow: 66049), and s5 (openflow: 65537). Thus, the command to read the byte amount of each port on route R1 using the SDN controller in JSON format is:

```
GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/
node/openflow:1/

GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/
node/openflow:513/

GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/
node/openflow:262401/

GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/
node/openflow:66049/

GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/
node/openflow:65537/
```

Listing 8 – Command sent to controller SDN

The request using the *GET* method for reading data from switch s1 in the JSON format:

```
GET http://192.168.0.111:8181/restconf/operational/opendaylight-inventory:nodes/
node/openflow:1/

{
    "node": [
    {
    "id": "openflow:1",
    "flow-node-inventory:ip-address": "192.168.0.106",
    "flow-node-inventory:hardware": "Open vSwitch",
    "flow-node-inventory:software": "2.3.90",
    "flow-node-inventory:table": []
    "flow-node-inventory:manufacturer": "Nicira, Inc.",
    "node-connector": [
    {
        "id": "openflow:1:1", {
        "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
        "collision-count": 0,
        "transmit-drops": 0,
        "receive-errors": 0,
        "receive-drops": 0,
        "duration": {
            "second": 11032,
            "nanosecond": 963000000
        },
        "transmit-errors": 0,
        "receive-over-run-error": 0,
        "bytes": {
            "received": 690781,
            "transmitted": 682141
        },
        "receive-crc-error": 0,
        "receive-frame-error": 0,
        "packets": {
            "received": 4358,
            "transmitted": 4358
        }},
        "flow-node-inventory:hardware-address": "22:EC:72:A1:1F:47",
        "flow-node-inventory:port-number": "1",
        "flow-node-inventory:name": "0_0_1-eth1",
        "flow-node-inventory:state": {
        "link-down": false,
        "live": false,
        "blocked": false
        },},
    {
        "id": "openflow:1:2",{...},
    },
        {
        "id": "openflow:1:3",{...},
    },
        {
        "id": "openflow:1:4",{...},
    }]}]}
```

Listing 9 – How to get the TX/RX bytes of each port of a switch specified in JSON format

Listing 9 only shows part of the data readings. The full answer is found in appendix D.

- The transmission rate in the input and output ports on switches. We do the reading using the command:

```
GET http://192.168.0.106:8008/app/top_flows_1/scripts/status.js/topology/json

{
"value": 789785.7163159391,
"agent": "192.168.0.106",
"key": "1_2_1-eth2"
},
{
"value": 747670.5886217935,
"agent": "192.168.0.106",
"key": "0_0_1-eth1"
},
{
"value": 746795.1898133202,
"agent": "192.168.0.106",
"key": "0_2_1-eth1"
},
{
"value": 699042.1515834683,
"agent": "192.168.0.106",
"key": "1_0_1-eth2"
},
{
"value": 691332.9161517477,
"agent": "192.168.0.106",
"key": "4_1_1-eth2"
},
{
"value": 20071.34802618815,
"agent": "192.168.0.106",
"key": "0_2_1-eth2"
},...}
]
```

Listing 10 – Command sent to sFlow-RT

Listing 10 only shows part of the data readings. The full answer is found in appendix C2.

Then, it was made 5 replications of the experiment to calculate latency. It are listed in Table 7.

Table 7 – Test to calculate latency.

Sample	Time (msec)
1	83.10
2	115.00
3	91.20
4	75.30
5	125.00

where the average was 97.92 msec.

3.2.5 Load Balancing

This component is the main focus of the investigation. The proposed solution predicts the path with less traffic, chosen by the neural network allowing the distribution of traffic flow over the available routes in the system.

The system architecture for the proposed load balancing in the KDN-based data center is shown below:

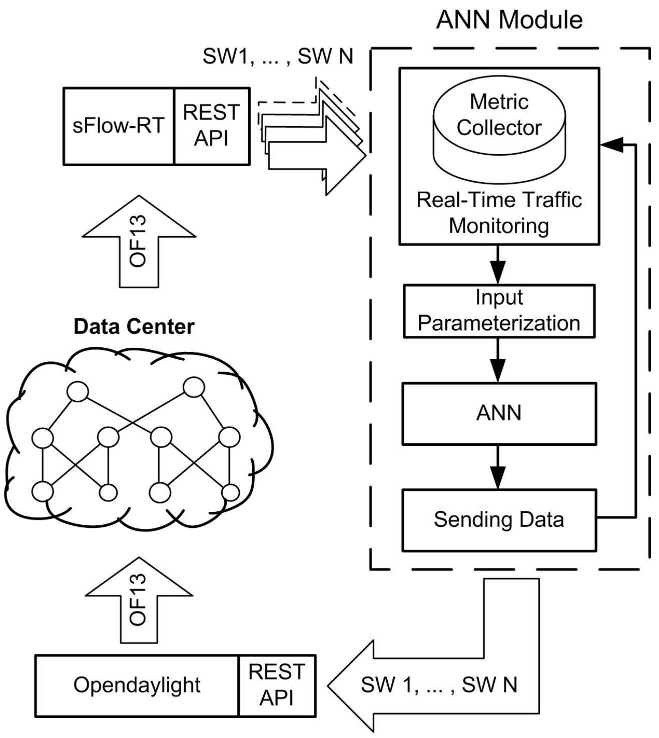


Figure 24 – System architecture

Figure 24 shows an overview of the methodology. The sFlow-RT network analyzer gathers the metrics of multiple paths of the data center. Next, the data is sent to the ANN module to be processed by the multilayer perceptron. Then the result will be a route with the least load and sent to the OpenDaylight controller. Following the SDN controller starts to program the flow tables of the network devices with the installation of the flow rules for each OpenFlow switch on the chosen path to reach the data flow transmission plan.

The proposed load balancing algorithm is as follows:

a) If the SDN controller finds only a single path for data transmission, then the SDN controller will create a flow-tables and allocate them to OpenFlow switches for active data transmission.
b) If the SDN controller finds multiple paths for data transmission, then the SDN controller should transmit multiple path load information to the ANN module. Besides the sFlow-RT gathering the metrics (bandwidth and transmission latency) of multiple paths of the data center.
c) The ANN module processes the parameters and chooses the least loaded path. The result is sent to the SDN controller.
d) The SDN controller receives the selected route from the ANN Module and creates flow-tables to allocate to OpenFlow switches.
e) The procedure is repeated every 10 seconds.

3.2.6 The ANN Module

We used a Multilayer Perceptron Network (MLP) (BISHOP *et al.*, 1995), a class of the feedforward Artificial Neural Network. An MLP has three types of layers: input, hidden, and output. The number of input and output units in a neural network is generally determined by the dimension of the data set, while the number of hidden units is a free parameter adjustable to give the best generalization performance, corresponding to the balance between under-fitting and over-fitting.

The configuration of the inputs for the multiclass classifier (MLP) depends on traffic matrix of the 32 links within the implemented data center. Thus, we get 64 inputs (ingress and egress ports) while the output is the four possible paths, where the classifier chooses a path with less traffic load (latency) from the source to destination node. Consequently, the structure of the classifier can be depicted in Figure 25, where each node is a neuron that uses a nonlinear activation function:

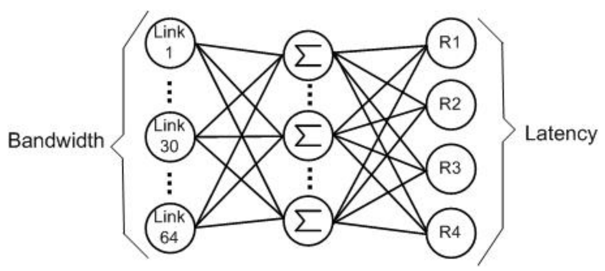


Figure 25 – Neural Network Architecture.

The MLP utilizes a learning technique called backpropagation for training. In our experimental setup, we set the value of 1000 as the number of training and the learning rate is set to 0.01. For this work, the input and output dataset was divided randomly into two subsets:

- Training set (70 %)
- Validation set (30 %)

There are many rule-of-thumb methods (HEATON, 2008) for determining the correct number of neurons to use in the hidden layers, such as the following:

- The number of neurons should be between the size of the input layer and the size of the output layer.
- The number of neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of neurons should be less than twice the size of the input layer.

This module has four parts that are:

a) Database: We store each feature in a time series database (InfluxDB) to collect and store the metrics. These values form the traffic matrix (dataset).

b) Reformatting of data: It would be problematic for the neural network to process the input data with different ranges. Feature scaling is a method used to normalize the range of independent variables or features of data, often between zero and one. We use the *Min-max scaling* (also called normalization)

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.4}$$

where the elements are rescaled so that they end up ranging from 0 to 1, x is a particular sample, x_{max} and x_{min} are the correspondingly minimum and maximum return in the training data, respectively.

c) Build an Artificial Neural Network:

We implement *the MNIST digits classification with stochastic gradient descent (SGD) using TensorFlow* (GÉRON, 2019). This classifier has a configuration of 64 inputs and 16 labels. Every label output has 4 classes.

We have built an MLP with the following architecture (see Seccion 3.2.3.1). The source code can be found in Appendix B.:

- First, we need to import appropriate libraries and the Tensorflow framework.

- After we are created a name scope using the name for every layer, here we define the input data (tensor) for the MLP network with 64 neurons, 2 hidden layers, and 4 output neurons. Therefore, it will contain all the computation nodes for this neuron network.
- Next, we need to calculate a logit (a linear transformation) $\widehat{y}_k$ for each class using Equation 2.5 :

$$\widehat{y}_k = f\left(f\left(\vec{x} \cdot \vec{w_k} + b_k\right)\right), k = 1,2,3,4 \tag{3.5}$$

- *The loss function* should match the type of problem we're trying to solve. In this case, we choose *binary_crossentropy* as the cross-entropy cost function, because it penalizes wrong predictions, producing larger gradients and converging faster.
- After calculating the scalar loss, we will take a step accordingly with the help of SGD using default Adam Optimizer.
- We use a learning rate of 0.01.
- Finally, using Tensorboard we can present the program nodes.

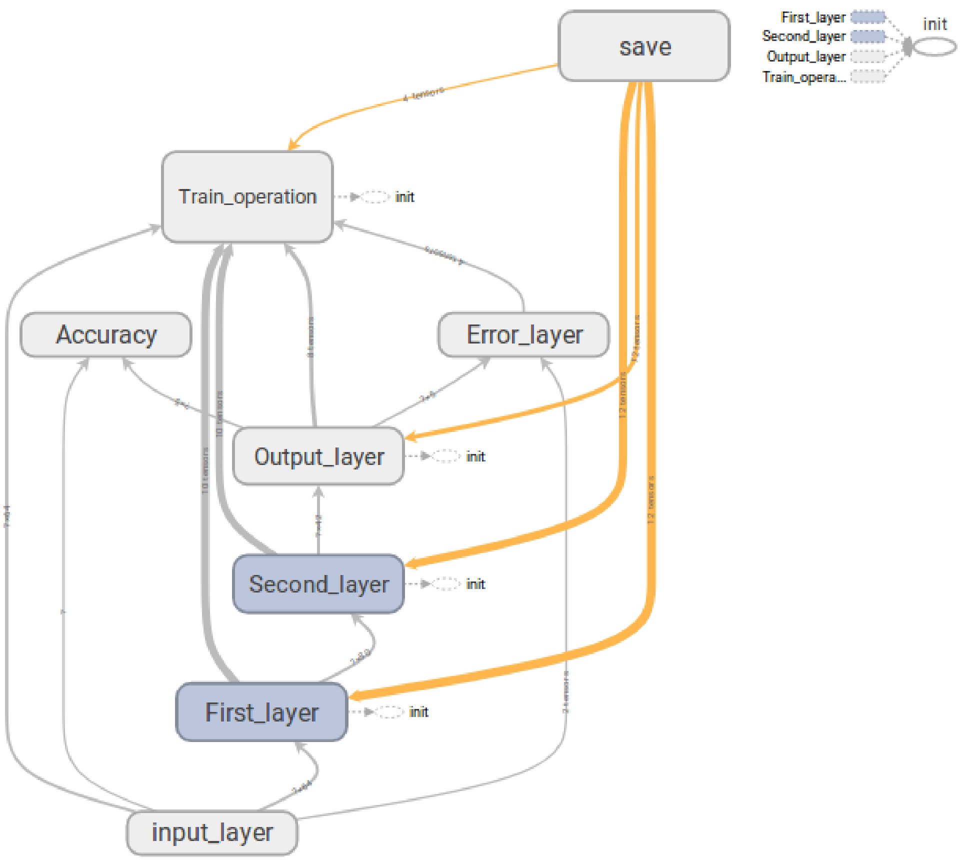


Figure 26 – Visualizing the graph using TensorBoard

Figure 26 shows the graph of inputs, two hidden layers, and outputs, as well as layer error, train operation, save, and accuracy.

d) Finally, we build the flow rules to forward the packets along the route chosen by ANN, then send them to the SDN controller using a RESTCONF. For example:

   The Listing 11 shows a flow that matches IPv4 packets (ethertype 0x800) with a source address ("h1:10.0.0.1") and destination address ("h5:10.1.0.2"). After sending them to port "2" . The flow rule is installed in table 0 of the OVS with datapath ID 1. The flow rule created contains the following body:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
<priority>2</priority>
<flow-name>Load Balance 1</flow-name>
  <match>
        <ipv4-destination>10.1.0.2</ipv4-destination>
        <ipv4-source>10.0.0.1</ipv4-source>
        <ethernet-match>
            <ethernet-type>
                <type>2048</type>
            </ethernet-type>
        </ethernet-match>
    </match>
    <id>1</id>
    <table_id>0</table_id>
    <instructions>
        <instruction>
            <order>0</order>
            <apply-actions>
                <action>
                    <order>0</order>
                    <output-action>
                     <output-node-connector>2</output-node-connector>
                    </output-action>
                </action>
            </apply-actions>
        </instruction>
    </instructions>
</flow>
```

Listing 11 – Flow rule in XML format send SDN controller

4 Experimental Evaluation

In this chapter, a proof of concept for our controller design is presented. We evaluate a prototype that includes use cases. First, an experiment is conducted to see how the KDN balances the 16 traffic flows in a fat-tree topology, where each route of traffic data, there are four probable paths between the sender and receiver. In the next section, we will see how reducing the input variables to the neural network can increase load balancing performance. Experiments also show that our ANN-based SDN controller design is working as expected.

4.1 First Scenario: ANN-based Load Balancing

This first experiment serves to illustrate how the load balancing delivers the data plane to program the switches in the resulting route with less load, as predicted by the neural network.

The knowledge acquired from other load balancers helps to create a neural network model to improve load balancing. The neural network model establishes the knowledge plan created for the proposed load balancing method in the context of KDN.

4.1.1 Dataset

The dataset is composed of the traffic matrix (the egress and ingress ports). In order to train and validate a neural network, we use two datasets where data has been collected from the static load balancer and the equal-cost multi-path (ECMP) load balancer.

The input traffic matrix: [*Input*, *Label*]

- **Input**: They are formed by the available bandwidth of the output and input ports of each switch, for example, some of the inputs of the neuronal network to predict the less load path using the TR1 traffic flow, are shown in Table 6, which are: { s1-eth1, s1-eth3, s3-eth1, s3-eth3, s5-eth2, s7-eth2, s8-eth2, s17-eth2, s18-eth2, s19-eth2, s20-eth2 }

  The total number of inputs to the neural network is 64.
- **Label**: The neuronal network's output is multiclass (R1, R2, R3, and R4), with four possible exit routes between source and destination node.

All data collected from the load balancing methods were normalized in range 0 to 1.

The distribution of the load traffic can be observed in Table 8. To generate the traffic was used the iPerf tool.

Table 8 – The distribution of traffic flows in the datacenter.

Traffic number	**Source - Destination**
TR1	h1 → h5
TR2	h2 → h6
TR3	h3 → h7
TR4	h4 → h8
TR5	h5 → h9
TR6	h6 → h10
TR7	h7 → h11
TR8	h8 → h12
TR9	h9 → h13
TR10	h10 → h14
TR11	h11 → h15
TR12	h12 → h16
TR13	h13 → h1
TR14	h14 → h2
TR15	h15 → h3
TR16	h16 → h4

The simulation parameters used in load balancers are:

Load balancing 1: The static (random) load balancer

Table 9 – Experiment parameters of the load balancer 1.

Name	**Description**
Topology	Fat-tree, k=4 (POD)
Links	64 (egress and ingress ports)
Bandwidth	10 Mbps
Network simulation tool	Mininet 2.1.0
SDN controller	OpenDaylight Controller (Helium)
Open vSwitch in Mininet	2.0.0
Data Collector	sFlow-RT
Experiment time	400 seconds

Load balancing 2: The equal-cost multi-path (ECMP) load balancer.

Table 10 – Experiment parameters of the load balancer 2.

Name	**Description**
Topology	Fat-tree, k=4 (POD)
Links	64 (egress and ingress ports)
Bandwidth	10 Mbps
Network simulation tool	Mininet 2.1.0
SDN controller	OpenDaylight Controller (Helium)
Open vSwitch in Mininet	2.0.0
Data Collector	sFlow-RT
Experiment time	400 seconds

We collect 1600 data from the load balancers.

4.1.2 Training and Evaluating of MLP

Before utilizing the MLP, it is indispensable to train the neural network with a massive amount of dataset to achieve the least error in the prediction made by ANN.

Inside the ANN module (see Seccion 3.2.6), the number of neurons in the hidden layer can not be easily determined. In the training procedure for the neuronal model choosing the number of neurons with different amounts for each hidden layer, then according to the results, we choose the neural model. Therefore, we created the five models with the value of [12,16], [20,6], [30,12], [50,22], and [80,42] that is the number of neurons used for the first and second hidden layer, respectively. Consequently, we can get these different neural network models to evaluate the performance of every MLP.

For example, we used the TR1 dataset to create the *MLP 1* model; the experimental results of these five neural networks are shown in Table 11.

Table 11 – Training results with different number of neurons in the hidden layer of the model (MLP 1).

No. of Test	**Number of hidden node**		**Error**	**Training time (sec)**
	1 Layer	**2 Layer**		
1	12	6	0,45	13.86
2	20	6	0,22	14.19
3	30	12	0,002	14.55
4	50	22	0,00085	15.95
5	80	42	0,0000032	17.81

In Table 11, we can choose the last result that has fewer errors, but the computational cost is higher to select the least loaded path, due to the number of neurons used to create the neural network model. Then the fourth result can be preferred with a reasonable error of 0.00085 and a training time of 15.95 sec.

Therefore, to control the 16 traffics through the proposed load balancing. The knowledge plan needs 16 neural network models. Each neural network model controls one traffic. The neural network model is an MLP, and its code is in Appendix B. The ANNs have been trained, validated, and stored for later use in the knowledge plan.

Table 12 shows the simulation parameters for the neural networks:

Table 12 – Model training information.

Name	**Description**
ANN	MLP
Layer Input	64 nodes
Layer Hidden	50 and 22 nodes
Layer Output	4 nodes
Matrix Features and Label	Bandwidth and Latency
Learning algorithm	Backpropagation
Learning rate	0.01
Maximum training epoch	1000
Bach size	50
Data set	1600
TensorFlow	2.00

With the configuration parameters in Table 12, we performed the training and validation for the 16 models of neural networks where we achieved the loss and accuracy of each model created. Figures 27 and 28 show the loss and accuracy of MLP 1 created with the TR1 dataset. According to this, the training loss decreases, and the training accuracy increases with each epoch. That's what we would assume when running gradient descent optimization. Although that isn't the problem of the validation loss and accuracy, a model that performs better on the training data isn't necessarily a model that will do better on data of validation.

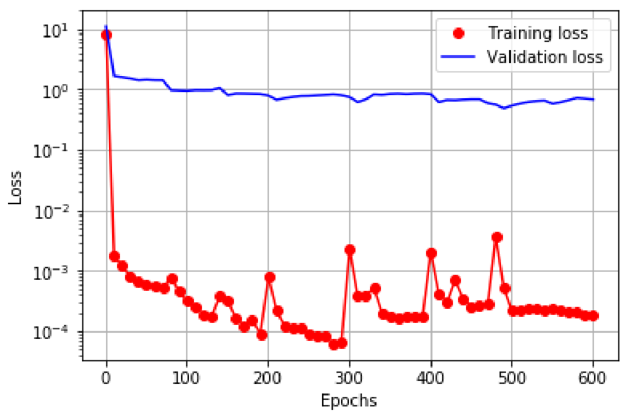


Figure 27 – Training and validation loss

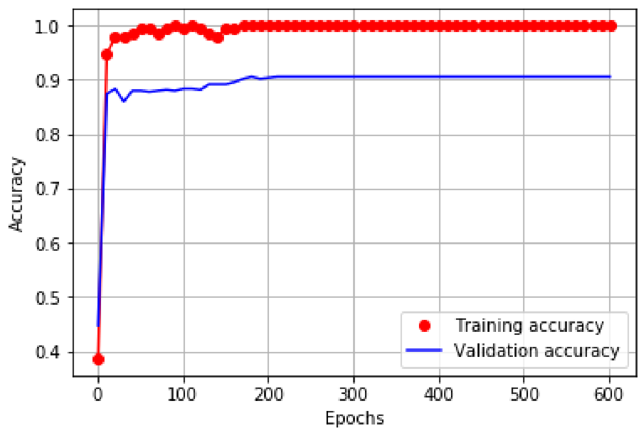


Figure 28 – Training and validation accuracy

The Table 13 is the summary of the 16 neural network models:

Table 13 – ANN accuracy.

Name	**Test accuracy (%)**
MLP 1	91.6
MLP 2	90.2
MLP 3	88.2
MLP 4	91.6
MLP 5	90.8
MLP 6	89.8
MLP 7	91.2
MLP 8	90.2
MLP 9	93.1
MLP 10	90.6
MLP 11	89.8
MLP 12	91.1
MLP 13	91.4
MLP 14	90.0
MLP 15	92.4
MLP 16	89.8

4.1.3 Evaluation Based on Classifier Performance

A common way to evaluate the performance of a classifier is to look at *the confusion matrix*. The rows represent the actual classes and the columns of the predicted classes.

4.1.3.1 MLP 1

The confusion matrix for the four-class:

As we have four classes, we create a four by four matrix. The first row in the matrix considers that 436 $(323 + 56 + 43 + 14 = 436)$ samples are of class R1, where 323

Table 14 – The confusion matrix for neural network (MLP 1)

		Predicted Class			
		R1	R2	R3	R4
Actual Class	R1	323	56	43	14
	R2	108	382	21	25
	R3	36	52	454	14
	R4	14	26	15	81

of them were classified as class R1, 56 to be a class R2, 43 to belong to the class R3, and 14 to be R4. Then, we see that of 436 samples, 74.08 % (365 samples) have been correctly classified as R1, and 25.94 % (102 samples) were misclassifications. In the case of class R2: 71.26 % was correctly classified, and 28.74 % was incorrectly classified. For class R3: 81.65 % was correctly classified, and 18.35 % was incorrectly classified. Finally, for class R4: 59.55 % was correctly classified, and 40.45 % was wrongly classified.

The neural network classifier is not able to predict class R4 with high precision. Therefore, the model could not distinguish class R4 because there are few samples of class R4 in the dataset.

In Python's scikit-learn[1], we can easily calculate the F1-score, precision and recall for each class in a multi-class classifier. A convenient function to use here is *classification_report*. The result is mentioned in Table 15.

Table 15 – Performance metrics for neural network (MLP 1)

	Precision	**Recall**	**F1-score**
R1	0.67	0.74	0.70
R2	0.74	0.71	0.72
R3	0.85	0.81	0.83
R4	0.60	0.57	0.60

One way to compare classifiers is to measure the area under the curve (AUC). A perfect classifier will have a ROC Area Under the Curve (ROC AUC) equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5 (GÉRON, 2019). In Figure 29, we can see the AUC values, and class R4 has a value of 0.78, which represents a lower level compared to the other classifiers. Therefore, we should increase the number of samples for class R4 or perform data balancing.

[1] https://scikit-learn.org/stable/

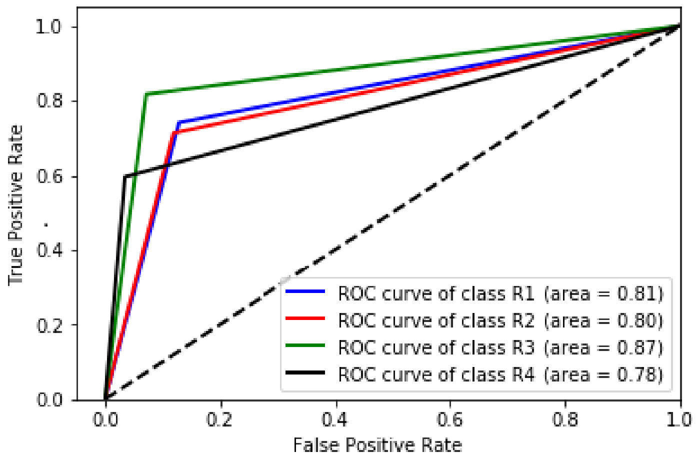


Figure 29 – Receiver operating characteristic for multi-class data (MLP 1)

Table 16 summarizes the performance of the ROC curve of each classifier with the four output classes. These classifiers are described in Appendix 5.

Table 16 – Performance result with the neural network model (MLP-PCA).

ANN	**AUC**			
	R1	R2	R3	R4
MLP 1	0.81	0.80	0.87	0.78
MLP 2	0.73	0.83	0.80	0.70
MLP 3	0.78	0.75	0.83	0.78
MLP 4	0.87	0.90	0.90	0.76
MLP 5	0.86	0.88	0.87	0.83
MLP 6	0.74	0.73	0.80	0.84
MLP 7	0.77	0.81	0.59	0.92
MLP 8	0.83	0.84	0.85	0.70
MLP 9	0.71	0.76	0.76	0.82
MLP 10	0.87	0.85	0.75	0.84
MLP 11	0.73	0.86	0.88	0.81
MLP 12	0.88	0.84	0.69	0.88
MLP 13	0.70	0.88	0.89	0.85
MLP 14	0.87	0.87	0.87	0.77
MLP 15	0.89	0.80	0.81	0.95
MLP 16	0.84	0.75	0.79	0.77

We describe only three main classifiers, which were classified as good, regular, and poor, using the ROC curve's values.

4.1.3.2 Good: MLP 4

The performance evaluation of the neural network (MLP 4) is summarized in Table 17 and Figure 30. The MLP 4 can be used to map 64 features and predict any of the four targets. Besides, it is possible to observe in Table 17 that the main diagonal of the confusion matrix gathers more samples. Therefore the values of the classifier are higher. The neural network classifier has high precision in the four classes, although there are few samples of classes R1, R3, and R4 in the dataset.

Table 17 – The confusion matrix for neural network (MLP 4)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	211	50	0	15	0.76
	R2	22	817	23	22	0.90
	R3	18	15	251	12	0.81
	R4	5	35	53	115	0.62
Precision		0.82	0.89	0.76	0.70	

The neural network classifier has high precision in the four classes, although there are few samples of classes R1, R3, and R4 in the dataset.

The AUC values in Figure 30 are close to 1, and it indicates the classifiers are good, besides the R4 class has a value of 0.76, representing a lower level than the other classes.

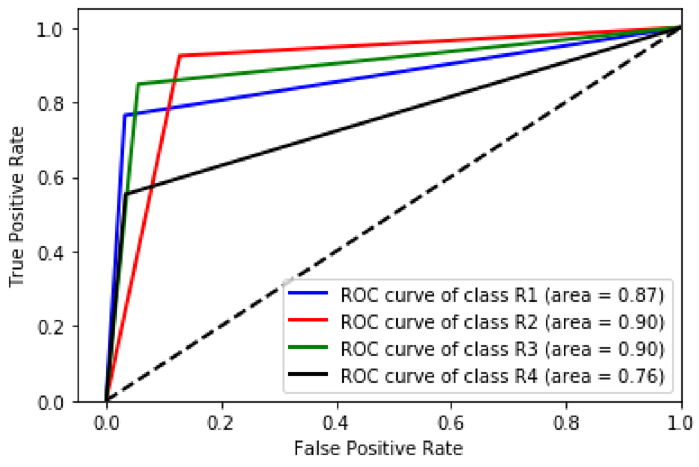


Figure 30 – Receiver operating characteristic for multi-class data (MLP 4)

4.1.3.3 Regular: MLP 11

The performance evaluation of the neural network (MLP 11) is summarized in Table 18 and Figure 31. It is possible to observe in Table 18 that the confusion matrix's main diagonal gathers more samples, but the precision is low for classes R2, R3, and R4. Therefore the classifier has regular behavior when classifying its classes. The neural network classifier has few samples for classes R1, R3, and R4 in the dataset.

The neural network classifier is not able to predict class R4 with high precision, because there are few samples of class R4 in the dataset.

Table 18 – The confusion matrix for neural network (MLP 11)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	150	85	50	39	0.46
	R2	2	472	14	72	0.84
	R3	6	26	387	37	0.85
	R4	3	32	52	237	0.73
Precision		0.93	0.77	0.77	0.62	

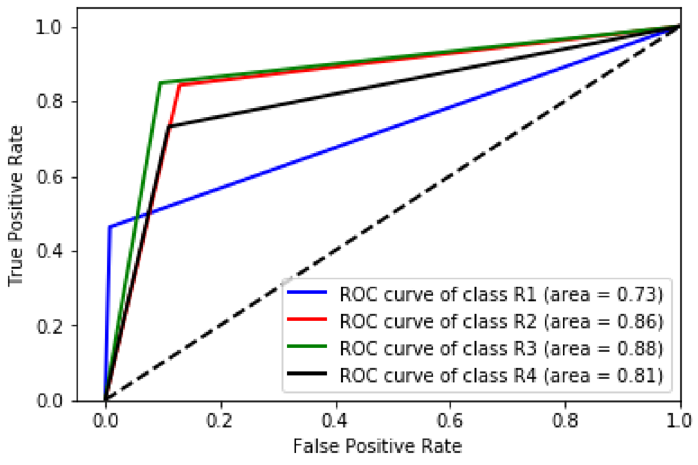


Figure 31 – Receiver operating characteristic for multi-class data (MLP 11)

We can see the AUC values in Figure 31, where the R1 class has a value of 0.73, which represents a lower level compared to the other classifiers.

4.1.3.4 Bad: MLP 9

The performance evaluation of the neural network (MLP 11) is summarized in Table 19 and Figure 32. It is possible to observe in Table 19 that the confusion matrix's main diagonal does not group the most significant samples. Besides, the precision and recall are low for classes. Therefore the classifier has bad behavior when classifying its classes. The neural network classifier is not able to predict class R3 with high accuracy, because there are few samples of class R3 in the dataset.

Table 19 – The confusion matrix for neural network (MLP 9)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	172	49	54	93	0.45
	R2	28	269	107	12	0.65
	R3	31	41	195	17	0.68
	R4	3	66	79	448	0.75
Precision		0.74	0.63	0.45	0.79	

We can see the AUC values in Figure 32, where the R1 class has a value of 0.71, which represents a lower level compared to the other classifiers.

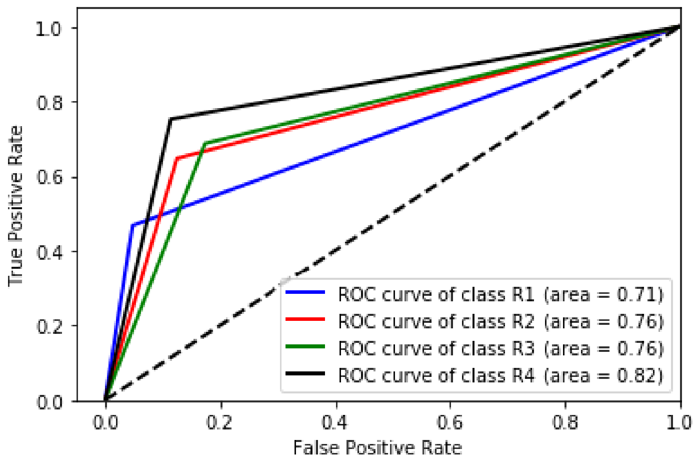


Figure 32 – Receiver operating characteristic for multi-class data (MLP 9)

4.1.4 Load Balancing Experiments

In order to evaluate the load balance study based on MLP, all datasets must be used to train the ANN using the exact same methodology. There are 16 traffic, and each one generates an MLP, according to the data entered by the load balancers (ECMP and Random). Table 13 shows the validation accuracy for each dataset. The 16 classifiers were created, and we evaluated the performance (see Section 4.1.3). These classifiers were used in the knowledge plan for load balancing.

Finally, we compare it with two other load balancing methods, namely (random) static load balancer, and equal-cost multi-path (ECMP) load balancer. Therefore we have 16 MLP, which choose a path with less traffic between the four possible routes.

The following tables show the simulation parameters for the neural network and the software defined network:

Table 20 – Model training information.

Name	Description
ANN	MLP
Layer Input	64 nodes
Layer Hidden	50 and 22 nodes
Layer Output	4 nodes
Matrix Features and Label	Bandwidth and Latency
Learning algorithm	Backpropagation
Learning rate	0.01
Maximum training epoch	1000
Bach size	50
Data set	1600
TensorFlow	2.00

Table 21 – Experiment parameters in the datacenter.

Name	Description
Topology	Fat-tree, k=4 (POD)
Links	64 (egress and ingress ports)
Network simulation tool	Mininet 2.1.0
SDN controller	OpenDaylight Controller (Helium)
Open vSwitch in Mininet	2.0.0
Data Collector	sFlow-RT
Experiment time	400 seg.

Figure 33 shows the results of the proposed study compared with the other two methods used for the training of the ANN. The proposed load balancing has the highest bandwidth in the 16 controlled traffic.

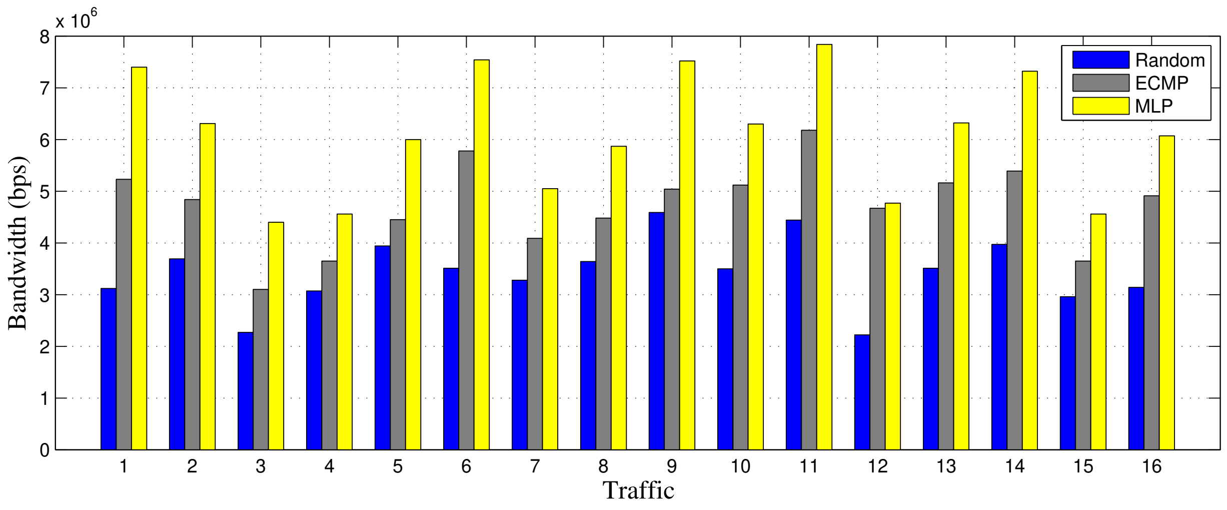


Figure 33 – Results obtained with the three load balancing methods (MLP).

4.2 Second Scenario: Reduction of Features and Dataset Balance

Unbalanced classification problems produce problems in various learning algorithms. These problems are characterized in several datasets used to train MLP's with an uneven proportion of cases that are available for each class. Therefore, It is possible to improve the classifier by balancing the data between the four classes and not having dominant classes. Besides, a reduction in the input variables to the classifier will allow fewer features to choose the less loaded path.

4.2.1 SMOTE for Balancing Data

It shows the imbalanced dataset for the four classes in Figure 34. Then class R4 has 136 samples, which means our models will learn more about types R1, R2, and R3, and very little in class R4.

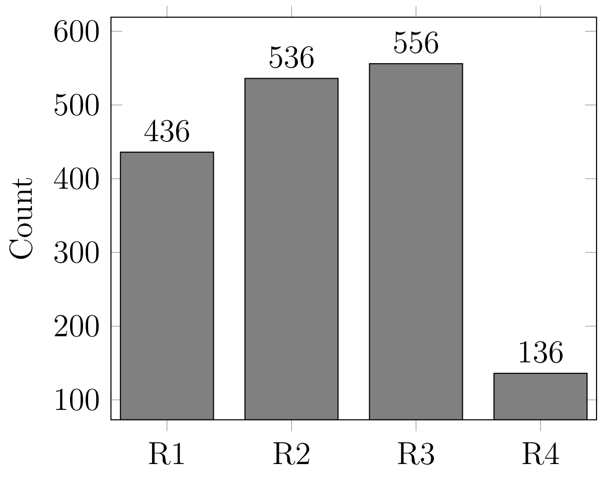


Figure 34 – TR1 dataset

An unbalanced dataset causes the classifier to have a high recognition rate (sensitivity) for the dominant classes R1, R2, and R3. The F1 score of the model could be unreliable in an imbalanced dataset.

There are several approaches for dealing with class imbalance include upsampling the minority class, downsampling the majority class, and the generation of synthetic training samples. There's no universally best solution. A technique for dealing with class imbalance is the generation of synthetic training samples. The most widely used algorithm for synthetic training sample generation is Synthetic Minority Oversampling Technique (SMOTE). SMOTE is configured to create data synthetically for the minority class to match the dominant class. In this case, an additional 420 samples will be created and added to class R4, providing 2224 samples in total.

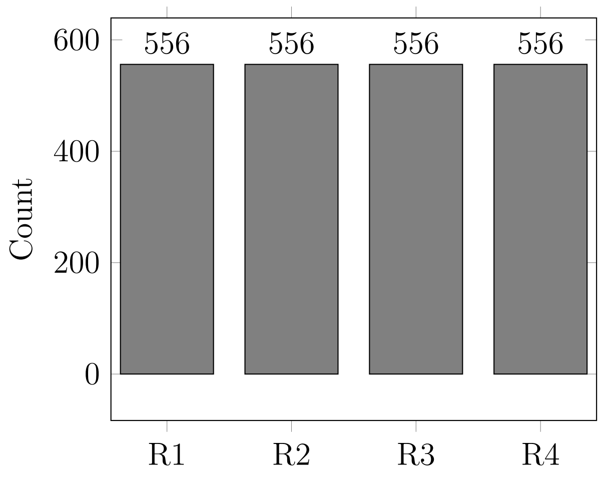


Figure 35 – TR1 dataset (SMOTE)

Summary table of the unbalanced and balanced datasets:

Table 22 – The unbalanced and balanced dataset

Dataset	**Unbalanced**				**Balaced**			
	R1	R2	R3	R4	R1	R2	R3	R4
TR1	436	536	556	136	556	556	556	556
TR2	180	432	248	804	804	804	804	804
TR3	356	316	332	660	660	660	660	660
TR4	276	884	296	208	884	884	884	884
TR5	304	512	192	656	656	656	656	656
TR6	440	456	368	400	456	456	456	456
TR7	504	656	272	232	656	656	656	656
TR8	604	504	432	124	604	604	604	604
TR9	368	416	284	596	596	596	596	596
TR10	652	304	140	568	652	652	652	652
TR11	324	560	456	324	560	560	560	560
TR12	728	292	332	312	728	728	728	728
TR13	456	356	224	628	628	628	628	628
TR14	520	664	316	164	664	664	664	664
TR15	596	360	532	176	596	596	596	596
TR16	224	320	268	852	852	852	852	852

4.2.2 Feature Space Dimension Reduction to Train ANN

In dimensionality reduction, we can simplify the data without losing too much information. It is often possible to considerably reduce the number of features, turning an intractable problem into a tractable one. There are many ways to achieve dimensionality reduction. Here are some of the most important algorithms:

- Principal Component Analysis (PCA)
- Kernel PCA

- Locally-Linear Embedding (LLE)
- t-distributed Stochastic Neighbor Embedding (t-SNE)

To analyze and build new feature subspace of lower dimensionality than the original one by PCA. Following steps are used in general:

a) Standardizing the data.

b) Constructing the covariance matrix.

c) Obtaining the eigenvalues and eigenvectors of the covariance matrix.

d) Select the optimal number of principal components (PC).

e) Construct new featured dataset from chosen components.

The optimal number of principal components is determined by looking at the cumulative explained variance ratio as a function of the number of components. The resulting plot (see Figure 36) indicates that the first 6 principal components correspond to approximately 60 % of all variance. Also, We see that these 21 components account for just over 95 % of the variance in the dataset. That would lead us to use these 21 components. We would recover most of the essential characteristics of the data.

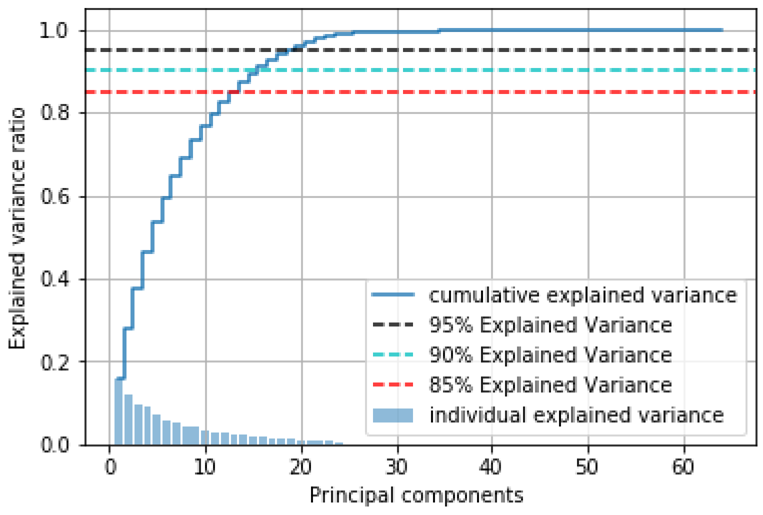


Figure 36 – Principal component index (MLP 1)

The 21 principal components are: { 1_3_1-eth1, 1_2_1-eth1, 4_2_1-eth3, 4_1_1-eth2, 2_3_1-eth1, 4_2_1-eth2 , 4_2_1-eth1, 4_1_1-eth4, 1_0_1-eth1, 4_1_1-eth1 , 0_2_1-eth2, 3_2_1-eth1, 1_2_1-eth2 , 1_2_1-eth3, 1_0_1-eth3, 4_1_2-eth1, 1_3_1-eth3 , 4_1_2-eth2, 2_2_1-eth1, 3_1_1-eth1 }

In Figure 37, we can see our TR1 dataset in its 2D feature subspace. The samples of the four routes are on the same diagonal line because we only predict a variable that is latency. PCA can often perform well even with a small percentage of outliers in the training set.

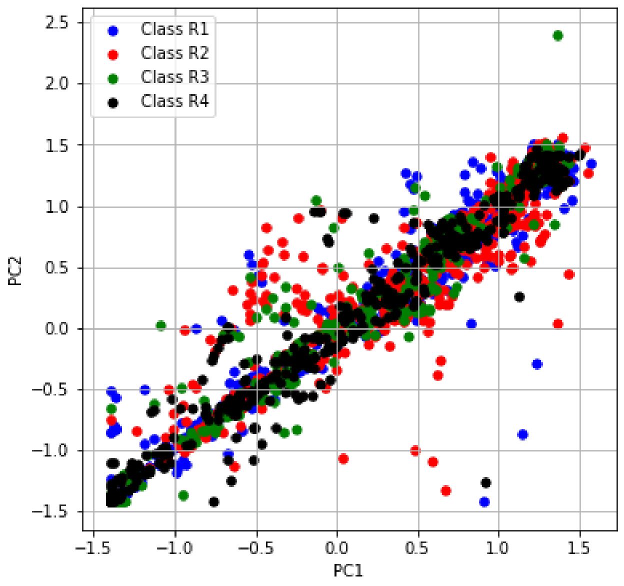


Figure 37 – Principal component by TR1 dataset

Finally, we can use the PCA for training the neural networks using 21 principal components. These principal components are different in every neural network model.

For example, we used the TR1 dataset to create the *MLP-PCA 1* model, using five neural models and we chose only one that has the best performance; the experimental results of these five neural network models are shown in Table 23.

Table 23 – Training results with different number of neurons in the hidden layer of the *MLP-PCA 1* model.

No. of Test	**Number of hidden node**		**Error**	**Training time (sec)**
	1 Layer	**2 Layer**		
1	4	2	0,55	10.07
2	8	5	0,11	10.10
3	14	7	0,05	9.86
4	16	9	0,04	9.97
5	22	15	0,04	10.29

In Table 23, the number of neurons in hidden layers of an ANN has a strong impact on error. If the number is insufficiently small (i.e., four neurons), the ANN model cannot accurately characterize a prediction the route less load. An increase in the number of neurons improves the accuracy of the model. However, when the number of neurons is 14, there's no more improvement in training time for the following neural models. Therefore, model 4 can be chosen for its average training time.

With the configuration parameters in Table 24, we performed the training and validation for the 16 models of neural networks where we achieved the loss and accuracy of each model created.

Table 24 – Model training information.

Name	**Description**
ANN	MLP
Layer Input	21 nodes
Layer Hidden	14 and 7 nodes
Layer Output	4 nodes
Matrix Features and Label	Bandwidth and Latency
Learning algorithm	Backpropagation
Learning rate	0.01
Maximum training epoch	300 Aprox.
Bach size	50
Data set	1600
TensorFlow	2.00

Figures 38 and 39 show the loss and accuracy of *MLP-PCA 1* created with the TR1 dataset.

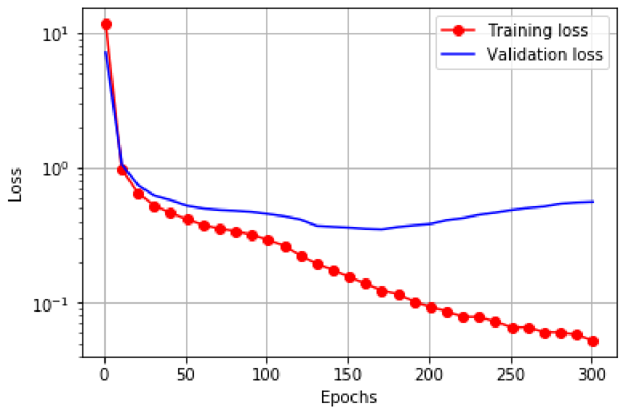


Figure 38 – Training and validation loss

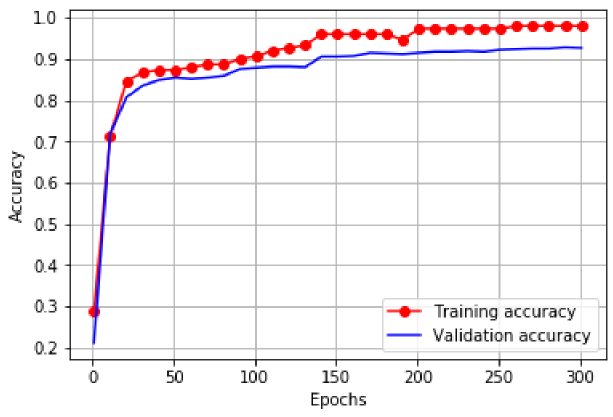


Figure 39 – Training and validation accuracy

The Table 25 is the summary of the 16 neural network models:

Table 25 – ANN accuracy.

Name	**Test accuracy (%)**
MLP 1	92.5
MLP 2	94.1
MLP 3	93.3
MLP 4	93.7
MLP 5	94.4
MLP 6	95.1
MLP 7	94.5
MLP 8	93.7
MLP 9	95.3
MLP 10	93.1
MLP 11	92.3
MLP 12	94.6
MLP 13	94.9
MLP 14	93.5
MLP 15	95.9
MLP 16	92.1

4.2.3 Evaluation Metrics

The performance evaluation of the *MLP-PCA 1* model with the reduction of features by PCA and dataset balance are summarized in Table 26 and Figure 40. In Table 26, as the classifier has a balance of data to train the neural network, it is possible to observe an improvement in the classification. The confusion matrix results allow us to observe that the classified samples are more concentrated on the diagonal, thus achieving a high level in the prediction than the MLP 1, as shown by the ROC curve (see Figure 40).

Table 26 – The confusion matrix for neural network (MLP-PCA 1)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	514	27	11	4	0.88
	R2	9	528	4	15	0.80
	R3	10	44	490	12	0.82
	R4	0	8	2	546	0.93
Precision		0.92	0.94	0.73	0.87	

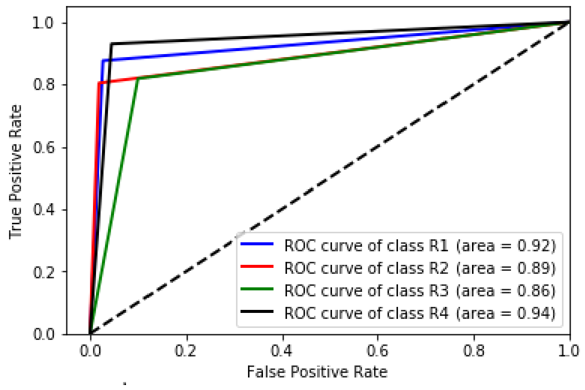


Figure 40 – Receiver operating characteristic for multi-class data (MLP-PCA 1)

In Figure 41, the PCA result for the TR1 dataset, it is still possible to observe some outliers present. A majority of the samples are on the same diagonal line because the only output is the latencies of routes R1, R2, R3, and R4.

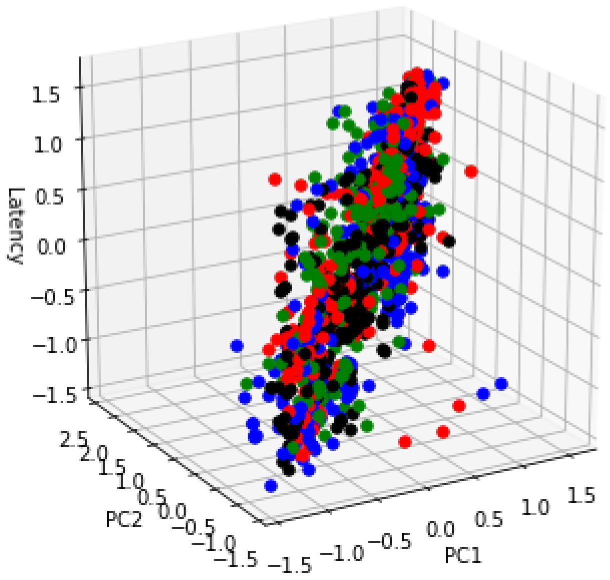


Figure 41 – Result MLP-PCA 1 for multi-class data

Table 27 shows the AUC score summary:

Table 27 – Performance result with the neural network model (MLP-PCA).

ANN	**AUC**			
	R1	R2	R3	R4
MLP-PCA 1	0.92	0.89	0.86	0.94
MLP-PCA 2	0.92	0.86	0.85	0.90
MLP-PCA 3	0.88	0.87	0.85	0.84
MLP-PCA 4	0.95	0.94	0.94	0.90
MLP-PCA 5	0.88	0.86	0.87	0.86
MLP-PCA 6	0.88	0.86	0.87	0.86
MLP-PCA 7	0.85	0.84	0.86	0.81
MLP-PCA 8	0.85	0.84	0.86	0.81
MLP-PCA 9	0.88	0.90	0.94	0.90
MLP-PCA 10	0.84	0.88	0.93	0.87
MLP-PCA 11	0.87	0.86	0.89	0.88
MLP-PCA 12	0.91	0.89	0.86	0.94
MLP-PCA 13	0.92	0.92	0.93	0.93
MLP-PCA 14	0.79	0.86	0.87	0.97
MLP-PCA 15	0.88	0.80	0.83	0.86
MLP-PCA 16	0.86	0.74	0.80	0.85

4.2.4 Load Balancing Experiments

To evaluate the load balancing based on the neural network model (MLP-PCA). We compare it with three load balancing methods, namely Static (Random) load balancer, ECMP load balancer, and MLP. There is 16 traffic, and each one generates an MLP-PCA, according to the data entered by the ECMP and Random load balancers. Therefore we have 16 MLP-PCA, which choose a path with less traffic between the four possible routes.

The following tables show the simulation parameters for the neural network and the software defined network:

Table 28 – Model training information.

Name	Description
ANN	MLP-PCA
Layer Input	21 nodes
Layer Hidden	14 and 7 nodes
Layer Output	4 nodes
Matrix Features and Label	Bandwidth and Latency
Learning algorithm	Backpropagation
Learning rate	0.001
Maximum training epoch	Aprox. 300
Bach size	50
Data set	1600
TensorFlow	2.00

Table 29 – Experiment parameters in the datacenter.

Name	Description
Topology	Fat-tree, k=4 (POD)
Links	64 (egress and ingress ports)
Network simulation tool	Mininet 2.1.0
SDN controller	OpenDaylight Controller (Helium)
Open vSwitch in Mininet	2.0.0
Data Collector	sFlow-RT
Experiment time	400 seg.

Figure 42 shows the results of the proposed study compared with the two methods used for the training of the ANN and the neural network model (MLP). The proposed load balancing has the highest bandwidth compared to traditional load balancing methods and MLP.

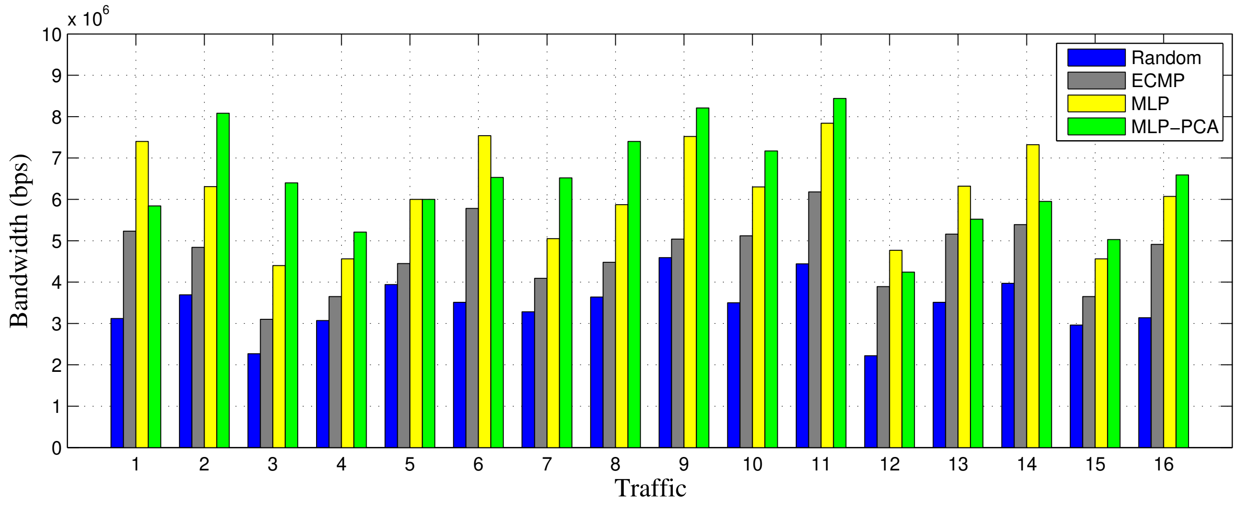


Figure 42 – Results obtained with the three load balancing methods (MLP-PCA).

We can emphasize that it was very successful to use the PCA and balanced dataset to create the neural network models.

The following traffics {2, 3, 4, 7, 8, 9, 10, 11, 15, and 16} show greater bandwidth. For traffic five, the *MLP-PCA 5* model has the highest bandwidth along with MLP because the two classifiers are close in the same AUC score.

The cases where the neural network model (MLP-PCA) was the second-best positioned in bandwidth was in traffic {4, 6, 12, 13, and 14}. The cause may be a lack of data to be able to map all the cases using the neural network.

4.3 Three Scenario: The Neural Network Model vs Multiple Linear Regression

Regression models are mathematical models that relate the behavior of one variable Y with another X, variable X is the independent variable, which is the model inputs, variable Y is the dependent variable, which is output from the model. The model is called simple when it involves only two variables and multiple when it has more than two variables.

Regression analysis is a statistical methodology that uses the relationship between two or more quantitative variables in such a way that one variable can be predicted from another.

The objective of linear regression is to search for the equation of a regression line that minimizes the sum of the squared errors (MSE), the difference between the observed value of Y and the predicted value.

The basic linear regression formula is shown below:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + +\theta_n x_n + e \tag{4.1}$$

where y is the target, θ are the linear regression parameters, and x are the features. The predicted data is called the target, and the data used to make predictions are called the features. We have built an linear regression (LR). The source code can be found in Appendix F.:

Now we will create a model considering all the features in the dataset using the egress and ingress ports as the linear regression parameters, we can now construct the linear regression model that can be used to predict Latency. The model would look something like this:

$$Latency = \theta_1 * (s1 - eth1) + \theta_2 * (s1 - eth3) + ... + \theta_{63} * (s20 - eth2) + \theta_{64} * (s20 - eth4) \tag{4.2}$$

The evaluation model with the TR1 dataset, we will predict the y_test from the x_test dataset using the selected model (i.e. LinearRegression()).

The coefficient of determination (R^2) is used to describe how well our models predicted.

R^2 score is 0.57, and MSE is 0.44. R^2 suggests that 57% of the dependent variable is predicted by the independent variable. The value of R^2 means that it is still possible that there is a non-linear relationship between the observed and predicted values.

In figure 43 we use a single input variable to be able to view it in 2D, to do that the port s9-eth1 is a variable that has a more significant correlation with the output (latency), this was analyzed in the PCA. In this graph, data that are closer to the diagonal can be predicted with a lower MSE. Therefore, as little data is diagonal, so the linear regression (LR) will not predict latency with high precision.

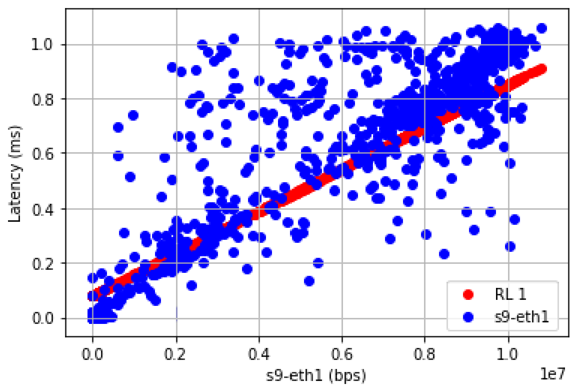


Figure 43 – Latency vs the s9-eth1 port

In Table 30, the performance analysis with the confusion matrix shows that the classification is not right, and the main diagonal does not concentrate most of the data, so the classifier does not have a high level to predict latency values. Also, the values of precision and recall are low.

Table 30 – The confusion matrix for lineal regression (LR 1)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	139	284	7	6	0.32
	R2	22	421	91	2	0.78
	R3	37	99	412	8	0.74
	R4	11	43	13	69	0.50
Precision		0.66	0.49	0.73	0.81	

The ROC curve for this linear regression presents low values for the four routes, which indicates the predicted values are not good.

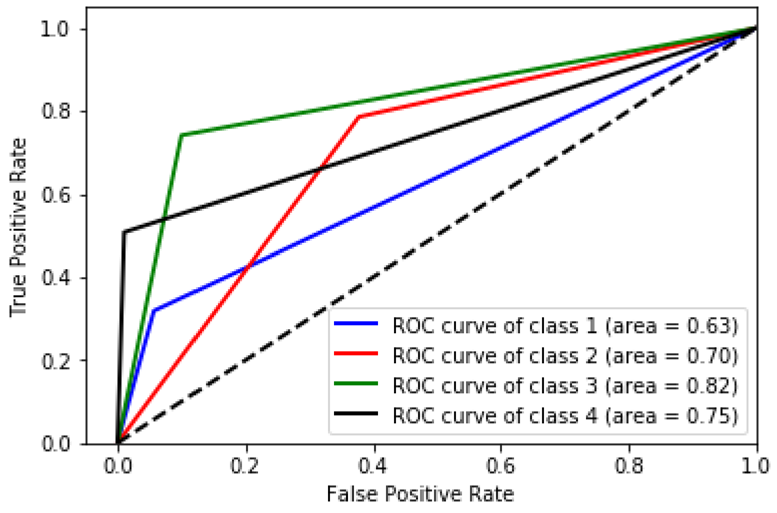


Figure 44 – Receiver operating characteristic for multi-class data (LR 1)

The linear regression model is put inside the knowledge plan after we start to control the TR1 traffic flow and other traffic.

Table 31 compares MLP-PCA and RL, where the neural network can have the highest bandwidth. The performance was also better on the ROC curve and the confusion matrix.

Table 31 – MLP-PCA vs RL.

Name	**MLP-PCA (Mbps)**	**RL (Mbps)**
TR 1	5.8	4.3
TR 2	8.1	3.8
TR 3	6.3	2.5
TR 4	5.2	3.1
TR 5	6.0	4.3
TR 6	6.5	3.3
TR 7	6.5	3.7
TR 8	7.2	3.9
TR 9	8.2	4.2
TR 10	7.1	3.3
TR 11	8.5	4.2
TR 12	4.2	2.9
TR 13	5.4	3.3
TR 14	5.9	4.2
TR 15	5.0	3.7
TR 16	6.7	3.9

We will highlight some disadvantages of the linear regression model and leave recommendations for when to use it and when not:

- A significant limitation of the linear regression model is that the machine does not learn the shape of nonlinearity. This part is the responsibility of the person who is assembling the model. In other words, it is necessary to force-code the nonlinearity in the algorithm, and this is done by forging new variables.

- They only consider a linear relationship;
- It takes the average of the dependent variable as a basis
- Sensitive to Outliers;
- Linear regression assumes that the data is independent.

Thus, it is recommended to use linear regression only in cases of simpler nonlinearities.

5
Conclusions and Future Work

This research has explored different load balancing estudies using SDN as a base technology. OpenFlow-based SDN can be used to monitor network switches and allow the dynamic reprogramming of network devices through an external controller. We leverage the concept of Knowledge-Defined Networking architecture, a novel networking paradigm that combines Software-Defined Networking, Network Analytics, and Artificial Intelligence (AI) techniques, and add a so-called Knowledge Plane for network control and management operations. These technologies provide robustness and stability for the network.

Our proposed design is a load-balancing based on MLP classifiers to change the routes according to the proposed traffic metrics. Traffic metrics as Bandwidth utilization ratios and path latencies are collected and integrated into the Artificial Neural Network to represent the path load condition. Thus, The MLP chooses the least loaded path between the four possible routes. The selected route is sent to the SDN controller, which creates the forwarding rules and installs on each OpenFlow switch.

Experimental results using Mininet and the OpenDaylight controller point to the performance opportunities of applying the KDN-based networks in data center scenarios. The load balancing method proposed achieves higher performance by reducing features with PCA and dataset balance with SMOTE, by comparing it with (random) static load balancer and equal-cost multi-path (ECMP) load balancer. Therefore, the results showed that network performance has increased after running the load balancing algorithm, which increases throughput and improves network utilization.

In our future research, we plan to run larger scale experiments with diverse topologies, network size, and explore different routing configurations. We propose to change the neural network model by the ML techniques, such as Q-learning techniques, LSTM, convolutional neural networks, and deep learning. In addition to applying KDN to data center scenarios, we will also consider video streaming scenarios over wireless access networks using Mininet-WiFi.

Bibliography

AKYILDIZ, I. F.; LEE, A.; WANG, P.; LUO, M.; CHOU, W. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, Elsevier, v. 71, p. 1–30, 2014.

AL-FARES, M.; LOUKISSAS, A.; VAHDAT, A. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 4, p. 63–74, 2008.

AL-FARES, M.; RADHAKRISHNAN, S.; RAGHAVAN, B.; HUANG, N.; VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In: *NSDI*. [S.l.: s.n.], 2010. v. 10, p. 19–19.

ANDERSSON, J.; TERMANDER, E. Performance management in software defined networking. 2015.

BISHOP, C. M. *et al. Neural networks for pattern recognition.* [S.l.]: Oxford university press, 1995.

CHO, S. L.; YANG, M. K.; LEE, J. Analytical modeling of a fat-tree network with buffered switches. In: IEEE. *Communications, Computers and signal Processing, 2001. PACRIM. 2001 IEEE Pacific Rim Conference on.* [S.l.], 2001. v. 1, p. 184–187.

CLARK, D. D.; PARTRIDGE, C.; RAMMING, J. C.; WROCLAWSKI, J. T. A knowledge plane for the internet. In: ACM. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* [S.l.], 2003. p. 3–10.

DORIA, A.; SALIM, J. H.; HAAS, R.; KHOSRAVI, H.; WANG, W.; DONG, L.; GOPAL, R.; HALPERN, J. *Forwarding and control element separation (ForCES) protocol specification.* [S.l.], 2010.

DU, Q.; ZHUANG, H. Openflow-based dynamic server cluster load balancing with measurement support. *Journal of Communications*, v. 10, n. 8, p. 16–21, 2015.

FERNANDES, E. L.; ROTHENBERG, C. E. Openflow 1.3 software switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC*, p. 1021–1028, 2014.

GÉRON, A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* [S.l.]: " O'Reilly Media, Inc.", 2019.

HAYKIN, S. S.; HAYKIN, S. S.; HAYKIN, S. S.; HAYKIN, S. S. *Neural networks and learning machines.* [S.l.]: Pearson Upper Saddle River, 2009. v. 3.

HEATON, J. *Introduction to neural networks with Java.* [S.l.]: Heaton Research, Inc., 2008.

HOPPS, C. *Analysis of an Equal-Cost Multi-Path Algorithm.* 2000. <https://tools.ietf.org/html/rfc2992>. [accessed: 09 Aug. 2019].

KREUTZ, D.; RAMOS, F. M.; VERISSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015.

LI, Y.; PAN, D. Openflow based load balancing for fat-tree networks with multipath support. In: *Proc. 12th IEEE International Conference on Communications (ICC'13), Budapest, Hungary*. [S.l.: s.n.], 2013. p. 1–5.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

MESTRES, A.; RODRIGUEZ-NATAL, A.; CARNER, J.; BARLET-ROS, P.; ALARCÓN, E.; SOLÉ, M.; MUNTÉS, V.; MEYER, D.; BARKAI, S.; HIBBETT, M. J. *et al.* Knowledge-defined networking. *arXiv preprint arXiv:1606.06222*, 2016.

MESTRES, A.; RODRIGUEZ-NATAL, A.; CARNER, J.; BARLET-ROS, P.; ALARCÓN, E.; SOLÉ, M.; MUNTÉS-MULERO, V.; MEYER, D.; BARKAI, S.; HIBBETT, M. J. *et al.* Knowledge-defined networking. *ACM SIGCOMM Computer Communication Review*, ACM, v. 47, n. 3, p. 2–10, 2017.

NUGRAHA, M.; PARAMITA, I.; MUSA, A.; CHOI, D.; CHO, B. Utilizing openflow and sflow to detect and mitigate syn flooding attack. *Journal of Korea Multimedia Society*, Korea Multimedia Society, v. 17, n. 8, p. 988–994, 2014.

NUNES, B. A. A.; MENDONCA, M.; NGUYEN, X.-N.; OBRACZKA, K.; TURLETTI, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, IEEE, v. 16, n. 3, p. 1617–1634, 2014.

ONF. *Open Networking Foundation (ONF), "OpenFlow switch specification", Tech. Rep., Oct. 2013.* 2013. [Online]. Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf>. [accessed: 09 Aug. 2019].

OpenDaylight Foundation. *OpenDaylight Controller:Architectural Framework.* 2018. [Online]. Available: <https://wiki.opendaylight.org/view/OpenDaylight_Controller:Architectural_Framework>. [accessed: 09 Aug. 2019].

PFAFF, B.; PETTIT, J.; KOPONEN, T.; JACKSON, E. J.; ZHOU, A.; RAJAHALME, J.; GROSS, J.; WANG, A.; STRINGER, J.; SHELAR, P. *et al.* The design and implementation of open vswitch. In: *NSDI*. [S.l.: s.n.], 2015. p. 117–130.

PHAAL, P.; LEVINE, M. *sFlow version 5. sFlow. org.* [S.l.]: July, 2004. [Online]. Available: <https://sflow.org/sflow_version_5.txt>. [accessed: 09 Aug. 2019].

QADIR, J.; ALI, A.; YAU, K.-L. A.; SATHIASEELAN, A.; CROWCROFT, J. Exploiting the power of multiplicity: a holistic survey of network-layer multipath. *IEEE Communications Surveys & Tutorials*, IEEE, v. 17, n. 4, p. 2176–2213, 2015.

REHMAN, S. U.; SONG, W.-C.; KANG, M. Network-wide traffic visibility in of@ tein sdn testbed using sflow. In: IEEE. *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*. [S.l.], 2014. p. 1–6.

ROTSOS, C.; MORTIER, R.; MADHAVAPEDDY, A.; SINGH, B.; MOORE, A. W. Cost, performance & flexibility in openflow: Pick three. In: IEEE. *Communications (ICC), 2012 IEEE International Conference on.* [S.l.], 2012. p. 6601–6605.

RUELAS, A. M.; ROTHENBERG, C. E. Implementation of neural switch using openflow as load balancing method in data center. *Campinas, Brasil: University of Campinas*, 2015.

SALLAMI, N. M. A.; ALOUSI, S. A. A. Load balancing with neural network. *(IJACSA) International Journal of Advanced Computer Science and Applications*, Citeseer, v. 4, n. 10, 2013.

SDX CENTRAL. *What is an OpenDaylight Controller?* 2018. [Online]. Available: <https://www.sdxcentral.com/sdn/definitions/sdn-controllers/opendaylight-controller/>. [accessed: 09 Aug. 2019].

SEMONG, T.; MAUPONG, T.; ANOKYE, S.; KEHULAKAE, K.; DIMAKATSO, S.; BOIPELO, G.; SAREFO, S. Intelligent load balancing techniques in software defined networks: A survey. *Electronics*, Multidisciplinary Digital Publishing Institute, v. 9, n. 7, p. 1091, 2020.

SFLOW. 2003. [Online]. Available: <https://www.sflow.org/sFlowOverview.pdf>. [accessed: 09 Aug. 2019].

SHENKER, S.; CASADO, M.; KOPONEN, T.; MCKEOWN, N. *et al.* The future of networking, and the past of protocols. *Open Networking Summit*, v. 20, 2011.

XIA, W.; WEN, Y.; FOH, C. H.; NIYATO, D.; XIE, H. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, IEEE, v. 17, n. 1, p. 27–51, 2015.

YANG, C.-T.; SU, Y.-W.; LIU, J.-C.; YANG, Y.-Y. Implementation of load balancing method for cloud service with open flow. In: IEEE. *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on.* [S.l.], 2014. p. 527–534.

ZAALOUK, A.; KHONDOKER, R.; MARX, R.; BAYAROU, K. Orchsec: An orchestrator-based architecture for enhancing network-security using network monitoring and sdn control functions. In: IEEE. *Network Operations and Management Symposium (NOMS), 2014 IEEE.* [S.l.], 2014. p. 1–9.

ZHANG-SHEN, R. Valiant load-balancing: Building networks that can support all traffic matrices. In: *Algorithms for Next Generation Networks.* [S.l.]: Springer, 2010. p. 19–30.

Appendix

APPENDIX A – Mininet

A.1 Topology

```
'''
@author: Milad Sharif (msharif@stanfor.edu)
'''

from mininet.topo import Topo
from mininet.node import Controller, RemoteController, OVSSwitch, CPULimitedHost
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.util import custom
from mininet.log import setLogLevel, info, warn, error, debug

from dctopo import FatTreeTopo, NonBlockingTopo

from DCRouting import Routing

from subprocess import Popen, PIPE
from argparse import ArgumentParser
import multiprocessing
from time import sleep
from monitor.monitor import monitor_devs_ng
import os
import sys

import json
import re

import termcolor as T

import requests # funtion GET

# Number of pods in Fat-Tree
K = 4

# Queue Size
QUEUE_SIZE = 100

# Link capacity (Mbps)
BW = 10

parser = ArgumentParser(description="ECMP routing")
```

```

parser.add_argument('-d', '--dir', dest='output_dir', default='log',
        help='Output directory')

parser.add_argument('-i', '--input', dest='input_file',
        default='inputs/all_to_all_data',
        help='Traffic generator input file')

parser.add_argument('-t', '--time', dest='time', type=int, default=30,
        help='Duration (sec) to run the experiment')

parser.add_argument('-p', '--cpu', dest='cpu', type=float, default=-1,
        help='cpu fraction to allocate to each host')

parser.add_argument('-n', '--nonblocking', dest='nonblocking', default=False,
        action='store_true', help='Run the experiment on the noneblocking topo')

parser.add_argument('--iperf', dest='iperf', default=False, action='store_true',
        help='Use iperf to generate traffics')

parser.add_argument('--hedera',dest='hedera', default=False,
        action='store_true', help='Run the experiment with hedera GFF scheduler')

parser.add_argument('--ecmp',dest='ECMP',default=False,
        action='store_true',help='Run the experiment with ECMP routing')

parser.add_argument('--random',dest='RANDOM',default=False,
        action='store_true',help='Run the experiment with RANDOM routing')


args = parser.parse_args()

def dumpTopology(net,agent='192.168.1.106'):
    topoData = {'nodes': {}, 'links': {}}
    for s in net.switches:
        topoData['nodes'][s.name] = {'name': s.name, 'dpid': s.dpid, 'ports': {},'agent': ag
    path = '/sys/devices/virtual/net/'
    for child in os.listdir(path):
        parts = re.match('(.*)-(.*)', child)
        #print parts.group(1)
        print child

        if (parts == None) or (child == 'ovs-system') : continue
        ifindex = open(path+child+'/ifindex').read().split('\n',1)[0]
        topoData['nodes'][parts.group(1)]['ports'][child] = {'name': child, 'ifindex': ifind
   # print json.dumps(topoData)
    #sleep(2)
    i = 0
    for s1 in net.switches:
        j = 0
        for s2 in net.switches:
            if j > i:
                intfs = s1.connectionsTo(s2)
                for intf in intfs:
                    s1ifIdx = topoData['nodes'][s1.name]['ports'][intf[0].name]['ifindex']
                    s2ifIdx = topoData['nodes'][s2.name]['ports'][intf[1].name]['ifindex']
                    linkName = '%s-%s' % (s1.name, s2.name)
                    info('topology link %s: %s %s %s %s %s %s\n' % (linkName, s1, intf[0].na
                    topoData['links'][linkName] = {'node1': s1.name, 'port1': intf[0].name,
            j += 1
        i += 1


    topofile = 'topology.json' #Store the topology in JSON format

    #Now identify the leaf/edge switches
    print 'host************************************************'
    sleep(2)

    for h in net.hosts:
        for s in net.switches:
            intfs = h.connectionsTo(s)
            if intfs:
                topoData['nodes'][s.name]['tag'] = 'edge'

    f = open(topofile, 'w')
    f.write(json.dumps(topoData, indent=4))
    f.flush()
    f.close

    #Send the data to sFlow
    os.system('sudo /home/mininet/hedera/sflow_topology_load.sh')


def FatTreeNet(k=4, bw=10, controller='hashed', dir = -1):
    ''' Create a Fat-Tree network '''

    pox_c = Popen("~/pox/pox.py --no-cli riplpox.riplpox --topo=ft,%s --routing=%s --mode=re

    info('*** Creating the topology')
    topo = FatTreeTopo(k)

    link = custom(TCLink, bw=bw)

        net = Mininet(topo, link=link, switch=OVSSwitch, controller=RemoteController)

    return net

```

```python
def progress(t):
    while t > 0:
        print T.colored('  %3d seconds left  \r' % (t), 'cyan'),
        t -= 1
        sys.stdout.flush()
        sleep(1)
    print '\r\n'

def start_tcpprobe():
    ''' Install tcp_probe module and dump to file '''
    os.system("rmmod tcp_probe; modprobe tcp_probe full=1;")
    Popen("cat /proc/net/tcpprobe > ~/hedera/tcp.txt" , shell=True)

def stop_tcpprobe():
    os.system("killall -9 cat")

def iperfTrafficGen(args, hosts, net):
    ''' Generate traffic pattern using iperf and monitor all of thr interfaces

    input format:
    src_ip dst_ip dst_port type seed start_time stop_time flow_size r/e
    repetitions time_between_flows r/e (rpc_delay r/e)

    '''

    host_list = {}
    for h in hosts:
        host_list[h.IP()] = h

    port = 5001

    data = open(args.input_file)

    start_tcpprobe()

    info('*** Starting iperf ...\n')
    for line in data:
        flow = line.split(' ')
        src_ip = flow[0]
        dst_ip = flow[1]
        if src_ip not in host_list:
            continue
        sleep(0.2)
        server = host_list[dst_ip]
        server.popen('iperf -s -p %s > ~/hedera/server.txt' % port, shell = True)

        client = host_list[src_ip]
        #client.popen('iperf -c %s -p %s -t 45 > ~/hedera/client.txt' % (server.IP(), port )
        client.popen('iperf -c %s -p %s -t 35 > ~/hedera/client.txt &' % (server.IP(), port

        port += 1

    monitors = []
    monitors.append(multiprocessing.Process(target = monitor_devs_ng, args =
                ('%s/rate.txt' % args.output_dir, 5.0))) #cada 5 segundos


    #Collect the Bandwidth
    for m in monitors:
        m.start()

    #Collect data from sFlow
    os.system('sudo /home/mininet/hedera/dados_flow_port.sh %s &' % args.output_dir)

    t=args.time
    temp = 0
    while t > 0:
        print T.colored('  %3d seconds left  \r' % (t), 'cyan')
        t -= 1
        sys.stdout.flush()
        sleep(1)
        print '\r\n'
        temp +=1
        print T.colored('  %3d seconds left  \r' % (temp), 'red')

        if (int(temp)%55) == 0:
            port = 5001
            data = open(args.input_file)
            info('*** Starting iperf refresh ...\n')
            for line in data:
                flow = line.split(' ')
                src_ip = flow[0]
                dst_ip = flow[1]
                if src_ip not in host_list:
                    continue
                sleep(0.2)
                server = host_list[dst_ip]
                #server.popen('iperf -s -p %s > ~/hedera/server.txt' % port, shell = True)

                client = host_list[src_ip]
                #client.popen('iperf -c %s -p %s -t 45 > ~/hedera/client.txt' % (server.IP()
                client.popen('iperf -c %s -p %s -t 35 > ~/hedera/client.txt &' % (server.IP(

                port += 1

    for m in monitors:
        m.terminate()

```

```
    info('*** stoping iperf ...\n')
    stop_tcpprobe()

    Popen("killall iperf", shell=True).wait()

def trafficGen(args, hosts, net):
    ''' Run the traffic generator and monitor all of the interfaces '''
    listen_port = 12345
    sample_period_us = 1000000

    traffic_gen = 'cluster_loadgen/loadgen'
    if not os.path.isfile(traffic_gen):
        error('The traffic generator doesn\'t exist. \ncd hedera/cluster_loadgen; make\n')
        return

    info('*** Starting load-generators\n %s\n' % args.input_file)
    for h in hosts:
        tg_cmd = '%s -f %s -i %s -l %d -p %d 2&>1 > %s/%s.out &' % (traffic_gen,
                args.input_file, h.defaultIntf(), listen_port, sample_period_us,
                args.output_dir, h.name)
        h.cmd(tg_cmd)

    sleep(1)

    info('*** Triggering load-generators\n')
    for h in hosts:
        h.cmd('nc -nzv %s %d' % (h.IP(), listen_port))


    monitor = multiprocessing.Process(target = monitor_devs_ng, args =
        ('%s/rate.txt' % args.output_dir, 0.01))

    monitor.start()

    sleep(args.time)

    monitor.terminate()

    info('*** Stopping load-generators\n')
    for h in hosts:
        h.cmd('killall loadgen')

def FatTreeTest(args,controller):

    net = FatTreeNet( k=K, bw=BW, controller=controller, dir = args.output_dir)

    net.start()

    #Create topology.json for sflow
    dumpTopology(net=net)

    #Wait for the switches to connect to the controller
    info('** Waiting for switches to connect to the controller\n')
    sleep(1)
    info('*Waiting for switches to connect to the Sflow***')
    os.system('sudo /home/mininet/Downloads/comandos/seed/multipath_riplpoxhedera2.sh')
    sleep(3)

    hosts = net.hosts

    if args.iperf:
        iperfTrafficGen(args, hosts, net)
    else:
        trafficGen(args, hosts, net)


    net.stop()


def clean():
    ''' Clean any the running instances of POX '''

    p = Popen("ps aux | grep 'pox' | awk '{print $2}'",
            stdout=PIPE, shell=True)
    p.wait()
    procs = (p.communicate()[0]).split('\n')
    for pid in procs:
        try:
            pid = int(pid)
            Popen('kill %d' % pid, shell=True).wait()
        except:
            pass

if __name__ == '__main__':

    setLogLevel( 'info' )
    if not os.path.exists(args.output_dir):
        print args.output_dir
        os.makedirs(args.output_dir)

    clean()

    if args.nonblocking:
        NonBlockingTest(args)
    elif args.ECMP:
        FatTreeTest(args,controller='hashed')
    elif args.RANDOM:
        FatTreeTest(args,controller='random')
    else:
        info('**error** please specify either hedera, ecmp, or nonblocking\n')

    clean()

    Popen("killall -9 top bwm-ng", shell=True).wait()
    os.system('sudo mn -c')
```

APPENDIX B – Artificial Neural Network

B.1 The MNIST digits classification with TensorFlow

```python
n_inputs = 64 # Number of variables
n_l1 = 50 # Number of first layer neurons
n_l2 = 22 # Number of second layer neurons
n_outputs = 5 # Number classes

graph = tf.Graph() # Create a graph
with graph.as_default(): # The following code creates the graph represented in Figure ..:

    # Input Layers
    with tf.name_scope('input_layer'): # Input layer name scope
        x_input = tf.placeholder(tf.float32, [None, n_inputs], name='images')
        y_input = tf.placeholder(tf.int64, [None], name='labels')

    # Layer 1
    with tf.name_scope('first_layer'): # First layer name scope
        # Layer variables
        W1 = tf.Variable(tf.truncated_normal([n_inputs, n_l1]), name='Weights')
        b1 = tf.Variable(tf.zeros([n_l1]), name='bias')

        l1 = tf.add(tf.matmul(x_input, W1), b1, name='linear_transformation')
        l1 = tf.nn.relu(l1, name='relu')

    # Layer 2
    with tf.name_scope('second_layer'): # Second layer name scope
        # Layer variables
        W2 = tf.Variable(tf.truncated_normal([n_l1, n_l2]), name='Weights')
        b2 = tf.Variable(tf.zeros([n_l2]), name='bias')

        l2 = tf.add(tf.matmul(l1, W2), b2, name='linear_transformation')
        l2 = tf.nn.relu(l2, name='relu')

    # Output layer
    with tf.name_scope('output_layer'): # Output layer name scope
        # Layer variables
        Wo = tf.Variable(tf.truncated_normal([n_l2, n_outputs]), name='Weights')
        bo = tf.Variable(tf.zeros([n_outputs]), name='bias')

        scores = tf.add(tf.matmul(l2, Wo), bo, name='linear_transformation') # Logits
    #Cross entropy - Training loss
        error = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y_input, logits=scores),
            name='error')

    # Calculate accuracy
    # This returns a 1D tensor full of boolean values, so we need to cast these booleans to
    correct = tf.nn.in_top_k(scores, y_input, 1)
    # Convert from bool to float32
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
    # Optimizer
    optimizer = tf.train.AdamOptimizer(learning_rate=lr).minimize(error)

    # Initializer
    init = tf.global_variables_initializer()

    # To save the trained model
    saver = tf.train.Saver()
```

APPENDIX C – sFlow-RT

C.1 Modification of the variable: links

```
// Assign link properties based on utilization
function linkProperties(utilization) {
  var color;
  if(utilization === -1) color = 'gray';
  else if(utilization < 500000) color = 'gray';
  else if(utilization < 2000000) color = 'blue';
  else if(utilization < 4000000) color = 'cyan';
  else if(utilization < 6000000) color = 'green';
  else if(utilization < 8000000) color = 'yellow';
  else color = 'red';
//Send data with positive value
  if(utilization < 0) utilization = 0;

//Send data to svg
  var props =  {'color':color,'width':10,'meter':utilization};
  return props;
}

// Edit table of links to match SVG map IDs with agents and data sources

setHttpHandler(function(req) {

var topp;

topp = getTopology();

var links = {

  '0_0_1-eth1': [{agt:'192.168.0.106', ds:topp.nodes["0_0_1"]["ports"]["0_0_1-eth1"].ifindex}],
'0_2_1-eth2': [{agt:'192.168.0.106', ds:topp.nodes["0_2_1"]["ports"]["0_2_1-eth2"].ifindex}],
'0_0_1-eth3': [{agt:'192.168.0.106', ds:topp.nodes["0_0_1"]["ports"]["0_0_1-eth3"].ifindex}],
'0_3_1-eth2': [{agt:'192.168.0.106', ds:topp.nodes["0_3_1"]["ports"]["0_3_1-eth2"].ifindex}],
'0_1_1-eth1': [{agt:'192.168.0.106', ds:topp.nodes["0_1_1"]["ports"]["0_1_1-eth1"].ifindex}],
'0_2_1-eth4': [{agt:'192.168.0.106', ds:topp.nodes["0_2_1"]["ports"]["0_2_1-eth4"].ifindex}],
'0_1_1-eth3': [{agt:'192.168.0.106', ds:topp.nodes["0_1_1"]["ports"]["0_1_1-eth3"].ifindex}],
'0_3_1-eth4': [{agt:'192.168.0.106', ds:topp.nodes["0_3_1"]["ports"]["0_3_1-eth4"].ifindex}],

'0_2_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["0_2_1"]["ports"]["0_2_1-eth1"].ifindex}],
'4_1_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_1"]["ports"]["4_1_1-eth1"].ifindex}],
'0_2_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["0_2_1"]["ports"]["0_2_1-eth3"].ifindex}],
'4_1_2-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_2"]["ports"]["4_1_2-eth1"].ifindex}],
'0_3_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["0_3_1"]["ports"]["0_3_1-eth1"].ifindex}],
'4_2_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_1"]["ports"]["4_2_1-eth1"].ifindex}],
'0_3_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["0_3_1"]["ports"]["0_3_1-eth3"].ifindex}],
'4_2_2-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_2"]["ports"]["4_2_2-eth1"].ifindex}],

'1_0_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["1_0_1"]["ports"]["1_0_1-eth1"].ifindex}],
'1_2_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["1_2_1"]["ports"]["1_2_1-eth2"].ifindex}],
'1_0_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["1_0_1"]["ports"]["1_0_1-eth3"].ifindex}],
'1_3_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["1_3_1"]["ports"]["1_3_1-eth2"].ifindex}],
'1_1_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["1_1_1"]["ports"]["1_1_1-eth1"].ifindex}],
'1_2_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["1_2_1"]["ports"]["1_2_1-eth4"].ifindex}],
'1_1_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["1_1_1"]["ports"]["1_1_1-eth3"].ifindex}],
'1_3_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["1_3_1"]["ports"]["1_3_1-eth4"].ifindex}],

'1_2_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["1_2_1"]["ports"]["1_2_1-eth1"].ifindex}],
'4_1_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_1"]["ports"]["4_1_1-eth2"].ifindex}],
'1_2_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["1_2_1"]["ports"]["1_2_1-eth3"].ifindex}],
'4_1_2-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_2"]["ports"]["4_1_2-eth2"].ifindex}],
'1_3_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["1_3_1"]["ports"]["1_3_1-eth1"].ifindex}],
'4_2_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_1"]["ports"]["4_2_1-eth2"].ifindex}],
'1_3_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["1_3_1"]["ports"]["1_3_1-eth3"].ifindex}],
'4_2_2-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_2"]["ports"]["4_2_2-eth2"].ifindex}],

'2_0_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["2_0_1"]["ports"]["2_0_1-eth1"].ifindex}],
'2_2_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["2_2_1"]["ports"]["2_2_1-eth2"].ifindex}],
'2_0_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["2_0_1"]["ports"]["2_0_1-eth3"].ifindex}],
'2_3_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["2_3_1"]["ports"]["2_3_1-eth2"].ifindex}],
'2_1_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["2_1_1"]["ports"]["2_1_1-eth1"].ifindex}],
'2_2_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["2_2_1"]["ports"]["2_2_1-eth4"].ifindex}],
'2_1_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["2_1_1"]["ports"]["2_1_1-eth3"].ifindex}],
'2_3_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["2_3_1"]["ports"]["2_3_1-eth4"].ifindex}],

'2_2_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["2_2_1"]["ports"]["2_2_1-eth1"].ifindex}],
'4_1_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_1"]["ports"]["4_1_1-eth3"].ifindex}],
'2_2_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["2_2_1"]["ports"]["2_2_1-eth3"].ifindex}],
'4_1_2-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_2"]["ports"]["4_1_2-eth3"].ifindex}],
'2_3_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["2_3_1"]["ports"]["2_3_1-eth1"].ifindex}],
'4_2_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_1"]["ports"]["4_2_1-eth3"].ifindex}],
'2_3_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["2_3_1"]["ports"]["2_3_1-eth3"].ifindex}],
'4_2_2-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_2"]["ports"]["4_2_2-eth3"].ifindex}],

'3_0_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["3_0_1"]["ports"]["3_0_1-eth1"].ifindex}],
'3_2_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["3_2_1"]["ports"]["3_2_1-eth2"].ifindex}],
'3_0_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["3_0_1"]["ports"]["3_0_1-eth3"].ifindex}],
'3_3_1-eth2' : [{agt:'192.168.0.106', ds:topp.nodes["3_3_1"]["ports"]["3_3_1-eth2"].ifindex}],
'3_1_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["3_1_1"]["ports"]["3_1_1-eth1"].ifindex}],
'3_2_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["3_2_1"]["ports"]["3_2_1-eth4"].ifindex}],
'3_1_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["3_1_1"]["ports"]["3_1_1-eth3"].ifindex}],
'3_3_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["3_3_1"]["ports"]["3_3_1-eth4"].ifindex}],
    '3_2_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["3_2_1"]["ports"]["3_2_1-eth1"].ifindex}],
    '4_1_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_1"]["ports"]["4_1_1-eth4"].ifindex}],
    '3_2_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["3_2_1"]["ports"]["3_2_1-eth3"].ifindex}],
    '4_1_2-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["4_1_2"]["ports"]["4_1_2-eth4"].ifindex}],
    '3_3_1-eth1' : [{agt:'192.168.0.106', ds:topp.nodes["3_3_1"]["ports"]["3_3_1-eth1"].ifindex}],
    '4_2_1-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_1"]["ports"]["4_2_1-eth4"].ifindex}],
    '3_3_1-eth3' : [{agt:'192.168.0.106', ds:topp.nodes["3_3_1"]["ports"]["3_3_1-eth3"].ifindex}],
    '4_2_2-eth4' : [{agt:'192.168.0.106', ds:topp.nodes["4_2_2"]["ports"]["4_2_2-eth4"].ifindex}],

};
```

C.2 The transmission rate in the input and output ports.

```
[
 {
  "value": 789785.7163159391,
  "agent": "192.168.0.106",
  "key": "1_2_1-eth2"
 },
 {
  "value": 747670.5886217935,
  "agent": "192.168.0.106",
  "key": "0_0_1-eth1"
 },
 {
  "value": 746795.1898133202,
  "agent": "192.168.0.106",
  "key": "0_2_1-eth1"
 },
 {
  "value": 699042.1515834683,
  "agent": "192.168.0.106",
  "key": "1_0_1-eth2"
 },
 {
  "value": 691332.9161517477,
  "agent": "192.168.0.106",
  "key": "4_1_1-eth2"
 },
 {
  "value": 20071.34802618815,
  "agent": "192.168.0.106",
  "key": "0_2_1-eth2"
 },
 {
  "value": 18448.330913986203,
  "agent": "192.168.0.106",
  "key": "1_2_1-eth1"
 },
 {
  "value": 17009.575149372824,
  "agent": "192.168.0.106",
  "key": "0_0_1-eth2"
 },
 {
  "value": 16701.59207744502,
  "agent": "192.168.0.106",
  "key": "4_1_1-eth1"
 },
      {
       "value": 16412.999380434692,
       "agent": "192.168.0.106",
       "key": "1_0_1-eth1"
      },
      {
       "value": 4.538829872768871E-37,
       "agent": "192.168.0.106",
       "key": "0_2_1-eth4"
      },
      {
       "value": 4.130379519497393E-37,
       "agent": "192.168.0.106",
       "key": "4_1_2-eth1"
      },
      {
       "value": 3.993174509831626E-37,
       "agent": "192.168.0.106",
       "key": "3_2_1-eth3"
      },
      {
       "value": 3.5964122777503437E-37,
       "agent": "192.168.0.106",
       "key": "0_1_1-eth4"
      },
      {
       "value": 3.5413279153475595E-37,
       "agent": "192.168.0.106",
       "key": "0_1_1-eth2"
      },
      {
       "value": 3.0598538807422476E-37,
       "agent": "192.168.0.106",
       "key": "3_1_1-eth1"
      },
      {
       "value": 2.868489040191135E-37,
       "agent": "192.168.0.106",
       "key": "4_1_2-eth3"
      },
      {
       "value": 2.6105249355333748E-37,
       "agent": "192.168.0.106",
       "key": "3_2_1-eth1"
      },
      {
       "value": 2.282801661370805E-37,
       "agent": "192.168.0.106",
       "key": "3_0_1-eth1"
```

```
 },
 {
  "value": 2.241492771840971E-37,
  "agent": "192.168.0.106",
  "key": "3_2_1-eth4"
 }
]
```

APPENDIX D – OpenDaylight

D.1 Reading input and output ports of switches

```
{
    {
"node": [
{
    "id": "openflow:1",
    "flow-node-inventory:ip-address": "192.168.0.106",
    "flow-node-inventory:description": "None",
    "flow-node-inventory:hardware": "Open vSwitch",
    "flow-node-inventory:serial-number": "None",
    "flow-node-inventory:software": "2.3.90",
    "flow-node-inventory:table": [
        {
            "id": 5,
            "opendaylight-flow-statistics:aggregate-flow-statistics": {
                "packet-count": 0,
                "flow-count": 0,
                "byte-count": 0
            },
            "opendaylight-flow-table-statistics:flow-table-statistics": {
                "active-flows": 0,
                "packets-matched": 0,
                "packets-looked-up": 0
            }
        },
        {
            "id": 39,
            "opendaylight-flow-statistics:aggregate-flow-statistics": {
                "packet-count": 0,
                "flow-count": 0,
                "byte-count": 0
            },
            "opendaylight-flow-table-statistics:flow-table-statistics": {
                "active-flows": 0,
                "packets-matched": 0,
                "packets-looked-up": 0
            }
        },

    ],
    "flow-node-inventory:manufacturer": "Nicira, Inc.",
    "flow-node-inventory:switch-features": {
```

```
                "max_tables": 254,
                "capabilities": [
                    "flow-node-inventory:flow-feature-capability-flow-stats",
                    "flow-node-inventory:flow-feature-capability-group-stats",
                    "flow-node-inventory:flow-feature-capability-queue-stats",
                    "flow-node-inventory:flow-feature-capability-port-stats",
                    "flow-node-inventory:flow-feature-capability-table-stats"
                ],
                "max_buffers": 256
            },
            "opendaylight-meter-statistics:meter-features": {
                "max_meter": 0,
                "max_color": 0,
                "max_bands": 0
            },
            "opendaylight-group-statistics:group-features": {
                "max-groups": [
                    4294967040
                ],
                "group-capabilities-supported": [
                    "opendaylight-group-types:select-weight",
                    "opendaylight-group-types:select-liveness",
                    "opendaylight-group-types:chaining"
                ],
                "actions": [
                    134217729
                ],
                "group-types-supported": [
                    "opendaylight-group-types:group-ff",
                    "opendaylight-group-types:group-select",
                    "opendaylight-group-types:group-all",
                    "opendaylight-group-types:group-indirect"
                ]
            },
            "node-connector": [
        {
            "id": "openflow:1:4",
            "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
                "collision-count": 0,
                "transmit-drops": 0,
                "receive-errors": 0,
                "receive-drops": 0,
                "duration": {
                    "second": 11032,
                    "nanosecond": 926000000
                },
                "transmit-errors": 0,
                "receive-over-run-error": 0,
                "bytes": {
```

```
                "received": 796,
                "transmitted": 682532
            },
            "receive-crc-error": 0,
            "receive-frame-error": 0,
            "packets": {
                "received": 10,
                "transmitted": 4357
            }
        },
        "flow-node-inventory:hardware-address": "F6:36:2C:A8:38:FA",
        "flow-node-inventory:maximum-speed": 0,
        "flow-node-inventory:peer-features": "",
        "flow-node-inventory:current-feature": "copper ten-gb-fd",
        "flow-node-inventory:port-number": "4",
        "flow-node-inventory:supported": "",
        "flow-node-inventory:name": "0_0_1-eth4",
        "flow-node-inventory:configuration": "",
        "flow-node-inventory:state": {
            "link-down": false,
            "live": false,
            "blocked": false
        },
        "flow-node-inventory:advertised-features": "",
        "flow-node-inventory:current-speed": 10000000
    },
    {
        "id": "openflow:1:3",
        "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
            "collision-count": 0,
            "transmit-drops": 0,
            "receive-errors": 0,
            "receive-drops": 0,
            "duration": {
                "second": 11032,
                "nanosecond": 939000000
            },
            "transmit-errors": 0,
            "receive-over-run-error": 0,
            "bytes": {
                "received": 692204,
                "transmitted": 682975
            },
            "receive-crc-error": 0,
            "receive-frame-error": 0,
            "packets": {
                "received": 4356,
                "transmitted": 4355
            }
        },
        "flow-node-inventory:hardware-address": "12:64:B4:A2:7E:80",
        "flow-node-inventory:maximum-speed": 0,
        "flow-node-inventory:peer-features": "",
        "flow-node-inventory:current-feature": "copper ten-gb-fd",
        "flow-node-inventory:port-number": "3",
        "flow-node-inventory:supported": "",
        "flow-node-inventory:name": "0_0_1-eth3",
        "flow-node-inventory:configuration": "",
        "flow-node-inventory:state": {
            "link-down": false,
            "live": false,
            "blocked": false
        },
        "flow-node-inventory:advertised-features": "",
        "flow-node-inventory:current-speed": 10000000
    },
    {
        "id": "openflow:1:LOCAL",
        "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
            "collision-count": 0,
            "transmit-drops": 0,
            "receive-errors": 0,
            "receive-drops": 0,
            "duration": {
                "second": 11032,
                "nanosecond": 972000000
            },
            "transmit-errors": 0,
            "receive-over-run-error": 0,
            "bytes": {
                "received": 0,
                "transmitted": 0
            },
            "receive-crc-error": 0,
            "receive-frame-error": 0,
            "packets": {
                "received": 0,
                "transmitted": 0
            }
        },
        "flow-node-inventory:hardware-address": "6A:C6:83:10:BD:4C",
        "flow-node-inventory:maximum-speed": 0,
        "flow-node-inventory:peer-features": "",
        "flow-node-inventory:current-feature": "",
        "flow-node-inventory:port-number": "LOCAL",
        "flow-node-inventory:supported": "",
        "flow-node-inventory:name": "0_0_1",
        "flow-node-inventory:configuration": "PORT-DOWN",
        "flow-node-inventory:state": {
            "link-down": true,
            "live": false,
            "blocked": false
        },
        "flow-node-inventory:advertised-features": "",
        "flow-node-inventory:current-speed": 0
    },
    {
        "id": "openflow:1:1",
        "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
            "collision-count": 0,
            "transmit-drops": 0,
            "receive-errors": 0,
            "receive-drops": 0,
            "duration": {
                "second": 11032,
                "nanosecond": 963000000
            },
            "transmit-errors": 0,
            "receive-over-run-error": 0,
            "bytes": {
                "received": 690781,
                "transmitted": 682141
            },
            "receive-crc-error": 0,
            "receive-frame-error": 0,
            "packets": {
                "received": 4358,
                "transmitted": 4353
            }
        },
        "flow-node-inventory:hardware-address": "22:EC:72:A1:1F:47",
        "flow-node-inventory:maximum-speed": 0,
        "flow-node-inventory:peer-features": "",
        "flow-node-inventory:current-feature": "copper ten-gb-fd",
        "flow-node-inventory:port-number": "1",
        "flow-node-inventory:supported": "",
        "flow-node-inventory:name": "0_0_1-eth1",
        "flow-node-inventory:configuration": "",
        "flow-node-inventory:state": {
            "link-down": false,
            "live": false,
            "blocked": false
        },
        "flow-node-inventory:advertised-features": "",
        "flow-node-inventory:current-speed": 10000000
    },
    {
        "id": "openflow:1:2",
        "opendaylight-port-statistics:flow-capable-node-connector-statistics": {
            "collision-count": 0,
            "transmit-drops": 0,
            "receive-errors": 0,
            "receive-drops": 0,
            "duration": {
                "second": 11032,
                "nanosecond": 951000000
            },
            "transmit-errors": 0,
            "receive-over-run-error": 0,
            "bytes": {
                "received": 706,
                "transmitted": 683065
            },
            "receive-crc-error": 0,
            "receive-frame-error": 0,
            "packets": {
                "received": 9,
                "transmitted": 4357
            }
        },
        "flow-node-inventory:hardware-address": "D2:CC:DE:7B:C8:A2",
        "flow-node-inventory:maximum-speed": 0,
        "flow-node-inventory:peer-features": "",
        "flow-node-inventory:current-feature": "copper ten-gb-fd",
        "flow-node-inventory:port-number": "2",
        "flow-node-inventory:supported": "",
        "flow-node-inventory:name": "0_0_1-eth2",
        "flow-node-inventory:configuration": "",
        "flow-node-inventory:state": {
            "link-down": false,
            "live": false,
            "blocked": false
        },
        "flow-node-inventory:advertised-features": "",
        "flow-node-inventory:current-speed": 10000000
    }
    ]
    }
    ]
    }
```

APPENDIX E – The ROC curve's values

E.1 MLP 2

The performance evaluation of the neural network (MLP 2) is summarized in Table E.1 and Figure E.1. The MLP 2 is used to map 64 features and four targets.

Table E.1 – The confusion matrix for neural network (MLP 2)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	141	22	5	12	0.78
	R2	53	307	42	30	0.71
	R3	62	7	159	20	0.64
	R4	361	41	18	384	0.45
Precision		0.41	0.81	0.71	0.86	

The neural network classifier is not able to predict class R1 with high precision because there are few samples of class R1 in the dataset.

We can see the AUC values in Figure E.1, and class R4 has a value of 0.70, which represents a lower level compared to the other classifiers.

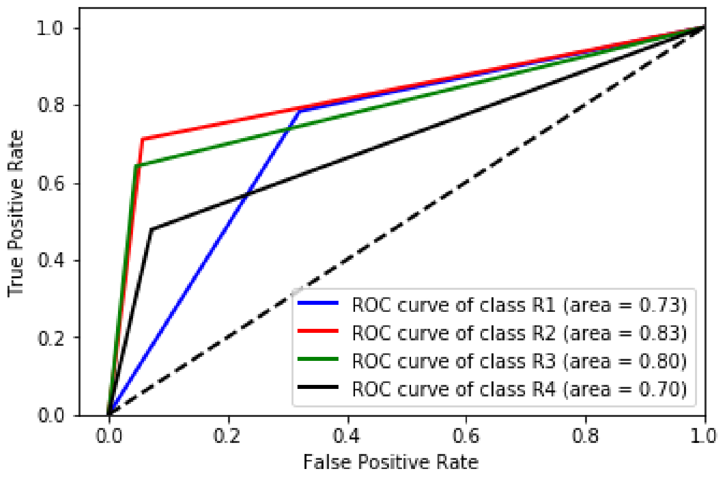


Figure E.1 – Receiver operating characteristic for multi-class data (MLP 2)

E.2 MLP 3

The performance evaluation of the neural network (MLP 3) is summarized in Table E.2 and Figure E.1. The MLP 3 is used to map 64 features and four targets.

Table E.2 – The confusion matrix for neural network (MLP 3)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	287	46	6	17	0.80
	R2	113	195	0	8	0.62
	R3	61	23	226	22	0.68
	R4	147	102	13	398	0.60
Precision		0.47	0.53	0.92	0.89	

The neural network classifier is not able to predict the R1 and R2 classes with high precision, because there are few samples of the R1 and R2 classes in the data set.

We can see the AUC values in Figure E.2 , and the R2 class has a value of 0.75, which represents a lower level compared to the other classifiers.

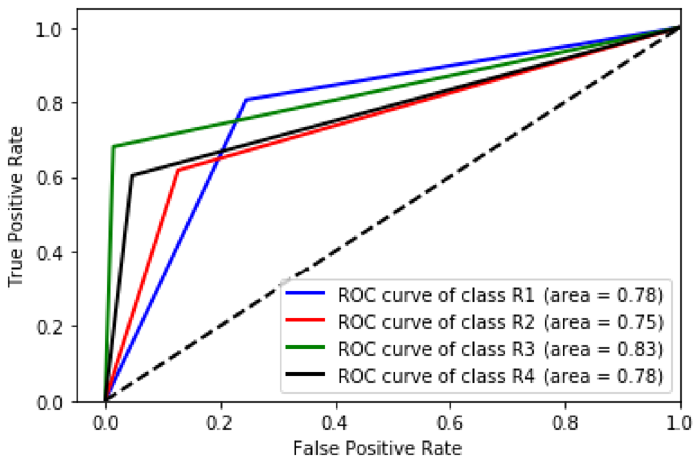


Figure E.2 – Receiver operating characteristic for multi-class data (MLP 3)

E.3 MLP 5

The performance evaluation of the neural network (MLP 5) is summarized in Table E.3 and Figure E.3. The MLP 5 is used to map 64 features and four targets.

Table E.3 – The confusion matrix for neural network (MLP 5)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	271	6	10	17	0.89
	R2	83	410	14	5	0.80
	R3	30	2	154	6	0.80
	R4	115	30	65	446	0.68
Precision		0.54	0.91	0.63	0.94	

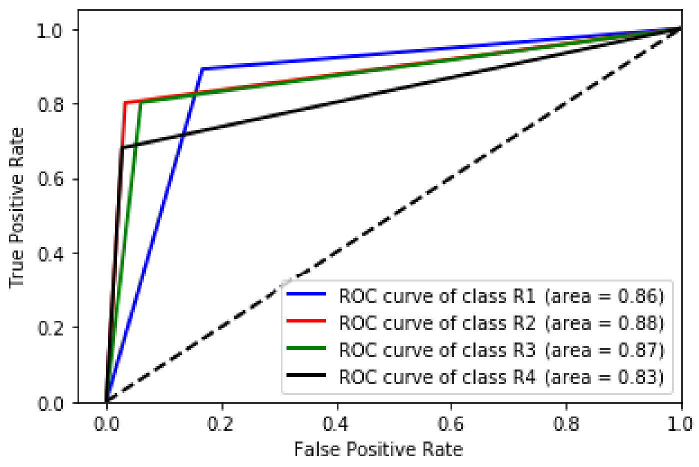


Figure E.3 – Receiver operating characteristic for multi-class data (MLP 5)

The neural network classifier is not able to predict the R1 and R3 classes with high precision, because there are few samples of the R1 and R3 classes in the dataset.

We can see the AUC values in Figure E.3, and the R4 class has a value of 0.83, which represents a lower level compared to the other classifiers.

E.4 MLP 6

The performance evaluation of the neural network (MLP 6) is summarized in Table E.4 and Figure E.4. The MLP 6 is used to map 64 features and four targets.

The neural network classifier is not able to predict class R3 with high precision, because there are few samples of class R3 in the data set.

We can see the AUC values in Figure E.4, and the R2 class has a value of 0.73, which represents a lower level compared to the other classifiers.

Table E.4 – The confusion matrix for neural network (MLP 6)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	221	1	162	56	0.50
	R2	3	220	104	129	0.48
	R3	28	4	307	29	0.83
	R4	7	13	40	340	0.85
Precision		0.85	0.92	0.50	0.61	

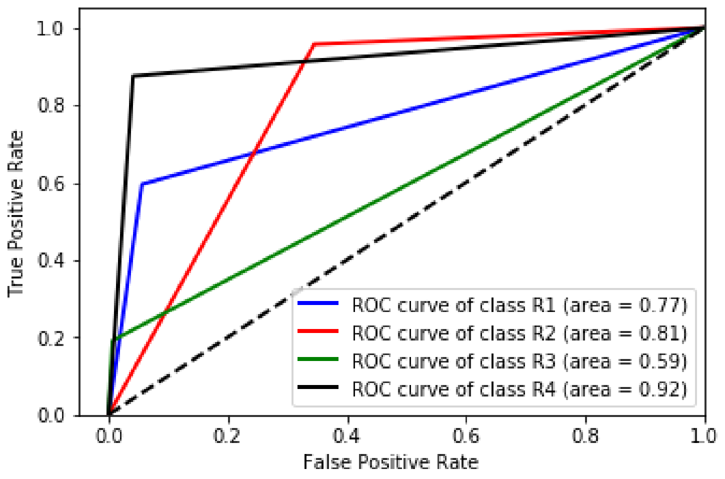


Figure E.4 – Receiver operating characteristic for multi-class data (MLP 6)

E.5 MLP 7

The performance evaluation of the neural network (MLP 7) is summarized in Table E.5 and Figure E.5. The MLP 7 is used to map 64 features and four targets.

Table E.5 – The confusion matrix for neural network (MLP 7)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	300	155	6	43	0.60
	R2	19	628	3	6	0.86
	R3	32	178	52	10	0.69
	R4	14	15	0	203	0.86
Precision		0.82	0.64	0.85	0.77	

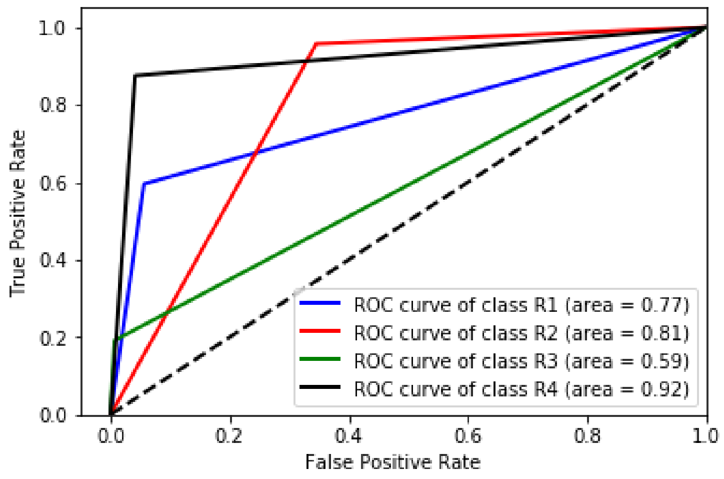


Figure E.5 – Receiver operating characteristic for multi-class data (MLP 7)

The neural network classifier is not able to predict the R3 and R4 classes with high precision, because there are few samples of the R3 and R4 classes in the dataset.

We can see the AUC values in Figure E.5, and the R3 class has a value of 0.59, which represents a lower level compared to the other classifiers.

E.6 MLP 8

The performance evaluation of the neural network (MLP 8) is summarized in Table E.6 and Figure E.6. The MLP 8 is used to map 64 features and four targets.

Table E.6 – The confusion matrix for neural network (MLP 8)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	503	93	7	1	0.83
	R2	64	417	14	9	0.83
	R3	92	31	307	2	0.71
	R4	33	40	0	51	0.51
Precision		0.73	0.72	0.94	0.81	

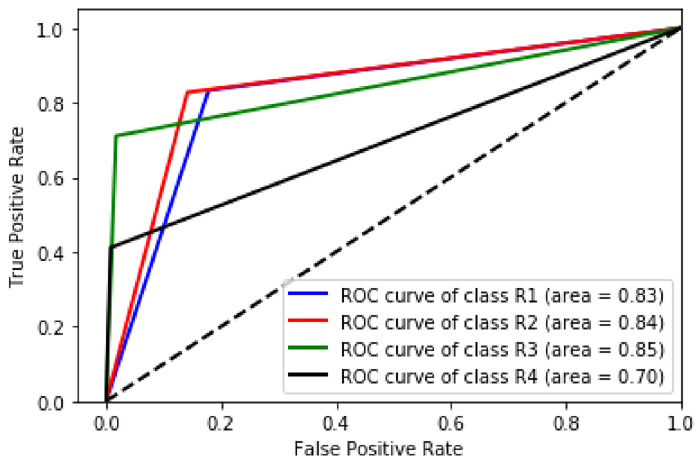


Figure E.6 – Receiver operating characteristic for multi-class data (MLP 8)

The neural network classifier has high precision to class R4, although there are few samples of class R4 in the dataset.

We can see the AUC values in Figure E.6, and the R4 class has a value of 0.70, which represents a lower level compared to the other classifiers.

E.7 MLP 10

The performance evaluation of the neural network (MLP 10) is summarized in Table E.7 and Figure E.7. The MLP 10 is used to map 64 features and four targets.

Table E.7 – The confusion matrix for neural network (MLP 10)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	627	15	1	9	0.96
	R2	57	223	2	22	0.73
	R3	61	1	72	6	0.52
	R4	113	33	19	403	0.71
Precision		0.73	0.82	0.77	0.92	

The neural network classifier has high precision in the four classes, although there are few samples for the R2, and R4 classes in the dataset.

We can see the AUC values in Figure E.7, and the R3 class has a value of 0.75, which represents a lower level compared to the other classifiers.

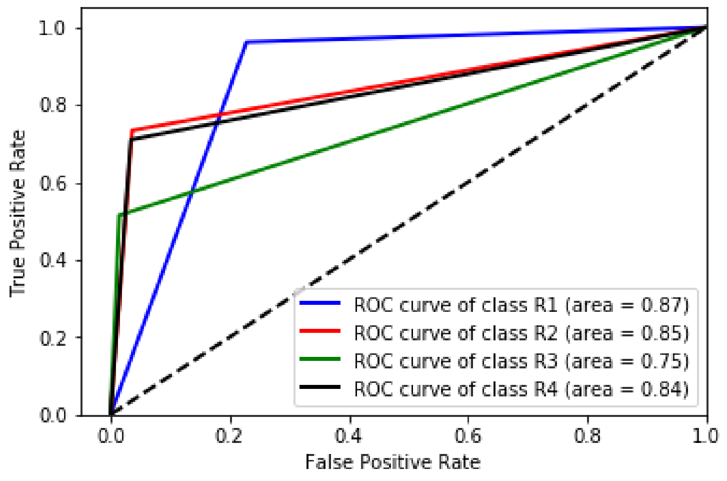


Figure E.7 – Receiver operating characteristic for multi-class data (MLP 10)

E.8 MLP 12

The performance evaluation of the neural network (MLP 12) is summarized in Table E.8 and Figure E.8. The MLP 12 is used to map 64 features and four targets.

Table E.8 – The confusion matrix for neural network (MLP 12)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	717	237	80	4	0.98
	R2	37	202	32	21	0.69
	R3	123	10	135	64	0.41
	R4	46	7	4	255	0.82
Precision		0.77	0.89	0.79	0.74	

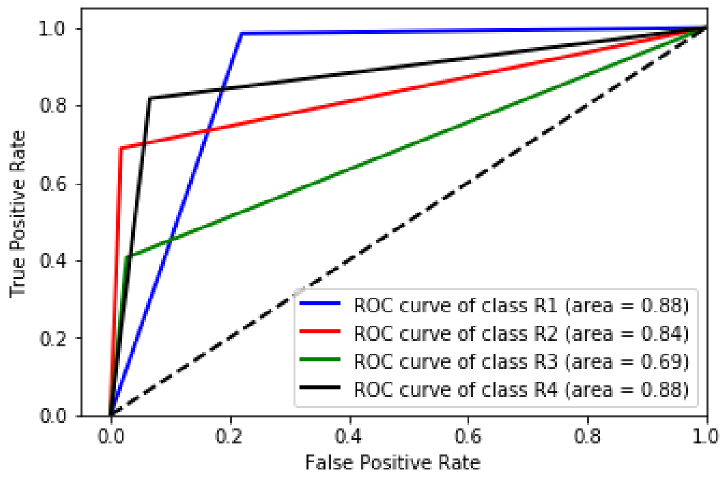


Figure E.8 – Receiver operating characteristic for multi-class data (MLP 12)

The neural network classifier has high precision in the four classes, although there are few samples for the R2, R3, and R4 classes in the dataset.

We can see the AUC values in Figure E.8, and the R3 class has a value of 0.69, which represents a lower level compared to the other classifiers.

E.9 MLP 13

The performance evaluation of the neural network (MLP 13) is summarized in Table E.9 and Figure E.9. The MLP 13 is used to map 64 features and four targets.

Table E.9 – The confusion matrix for neural network (MLP 13)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	188	49	52	167	0.41
	R2	1	291	12	52	0.82
	R3	1	11	187	25	0.84
	R4	12	14	10	592	0.94
Precision		0.93	0.80	0.72	0.71	

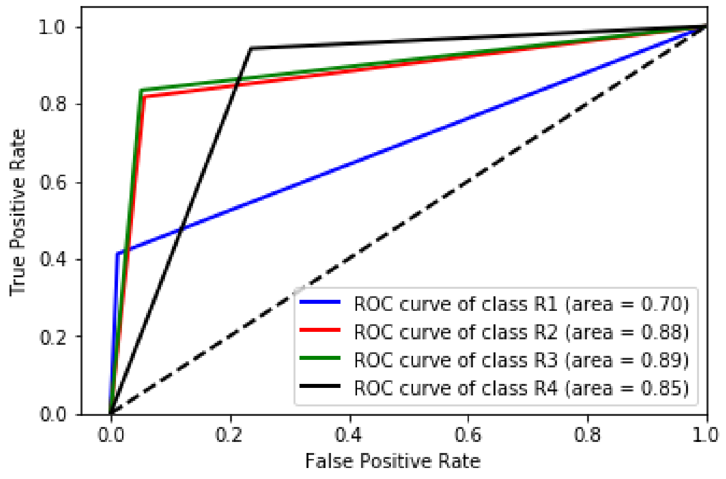


Figure E.9 – Receiver operating characteristic for multi-class data (MLP 13)

The neural network classifier has high precision in the four classes, although there are few samples for the R2, R3, and R4 classes in the dataset.

We can see the AUC values in Figure E.9, and the R1 class has a value of 0.70, which represents a lower level compared to the other classifiers.

E.10 MLP 14

The performance evaluation of the neural network (MLP 14) is summarized in Table E.10 and Figure E.10. The MLP 14 is used to map 64 features and four targets.

The neural network classifier is not able to predict the R2 and R4 classes with high precision, because there are few samples of the R2 and R4 classes in the dataset.

Table E.10 – The confusion matrix for neural network (MLP 14)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	496	23	1	0	0.95
	R2	125	524	6	9	0.79
	R3	61	20	234	1	0.74
	R4	66	5	2	91	0.56
Precision		0.66	0.92	0.96	0.90	

We can see the AUC values in Figure E.10, and the R4 class has a value of 0.76, which represents a lower level compared to the other classifiers.

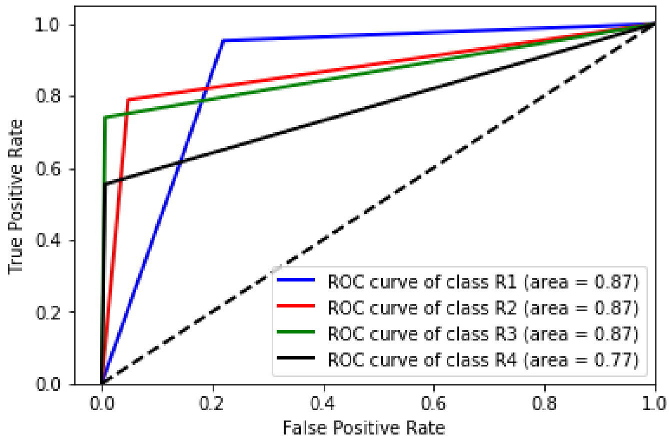


Figure E.10 – Receiver operating characteristic for multi-class data (MLP 14)

E.11 MLP 15

The performance evaluation of the neural network (MLP 15) is summarized in Table E.11 and Figure E.11. The MLP 15 is used to map 64 features and four targets.

Table E.11 – The confusion matrix for neural network (MLP 15)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	587	4	2	3	0.98
	R2	105	224	15	16	0.62
	R3	105	20	339	68	0.64
	R4	3	4	1	168	0.95
Precision		0.73	0.89	0.95	0.66	

The neural network classifier is not able to predict class R4 with high precision, because there are few samples of class R4 in the data set.

We can see the AUC values in Figure E.11, and the R2 class has a value of 0.80, which represents a lower level compared to the other classifiers.

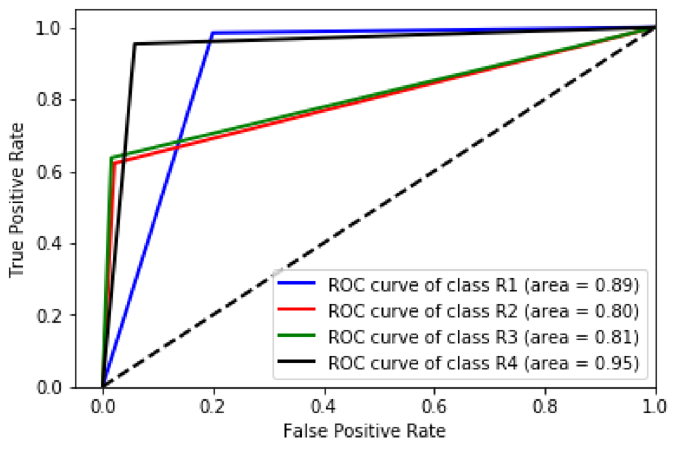


Figure E.11 – Receiver operating characteristic for multi-class data (MLP 15)

E.12 MLP 16

The performance evaluation of the neural network (MLP 16) is summarized in Table E.12 and Figure E.12. The MLP 16 is used to map 64 features and four targets.

Table E.12 – The confusion matrix for neural network (MLP 16)

		Predicted Class				
		R1	R2	R3	R4	Recall
Actual Class	R1	158	3	1	62	0.71
	R2	4	161	18	137	0.50
	R3	1	1	170	96	0.63
	R4	34	10	45	763	0.90
Precision		0.80	0.92	0.73	0.72	

The neural network classifier has high precision in the four classes, although there are few samples for the R1, R2, and R3 classes in the dataset.

We can see the AUC values in Figure E.12, and the R2 class has a value of 0.75, which represents a lower level compared to the other classifiers.

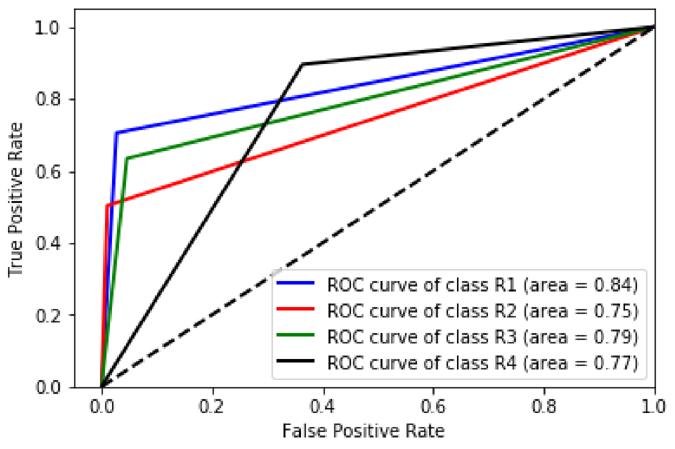


Figure E.12 – Receiver operating characteristic for multi-class data (MPL 16)

APPENDIX F – Linear Regression

```python
import pandas as pd
from sklearn import linear_model
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

df_x = pd.DataFrame(dataset.data, columns = dataset.feature_names)
df_y = pd.DataFrame(dataset.target)

# Feature Scaling
"""from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train)"""

# Fitting Multiple Linear Regression to the Training set
model = linear_model.LinearRegression()

# Training and testing the model
x_train, x_test, y_train, y_test = train_test_split(df_x, df_y,
                                                    test_size = 0.3, random_state = 4 )

# Fitting Multiple Linear Regression to the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Predicting the Test set results
y_pred = regressor.predict(X_test)

regressor = LinearRegression()
regressor.fit(X_train, y_train)
y_pred = regressor.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
rmse = (np.sqrt(mse))
r2 = round(regressor.score(X_train, y_train),2)

print("The model performance for training set")
print("--------------------------------------")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))
```