POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master in Computer Engineering

Master's Degree Thesis

An Architecture for Task and Traffic Offloading in Edge Computing via Deep Learning

Supervisor
prof. Flavio Esposito
Saint Louis University, St. Louis (USA)

Co-supervisor:
prof. Guido Marchetto
Politecnico di Torino, Torino (IT)

Candidate
Alessandro Gaballo
ID: 231587

Academic year 2017 – 2018

This thesis is dedicated to my parents,
who taught me the value of respect and
unconditionally believed in me, giving me
the support and the strength I needed.
From the bottom of my hearth, thank you.

Acknowledgements

I cannot find words to express my deepest gratitude to my supervisor, Professor Flavio Esposito, whose encouragement, guidance and support accompanied me from the first to the last day of work, allowing me to successfully complete this project.

I would also like to thank Professor Guido Marchetto, who made this experience in the United Stated possible and has always offered his guidance.

Finally, I want to thank all the close friends I've met in these past years, for making me realize my potential, for their support in good and bad times, and for their patience.

Abstract

The diffusion of smart mobile devices and the development of Internet of Things (IoT) has brought computational power in everyone's pocket, allowing people to perform simple tasks with their smartphones. There are some tasks, however, that are computationally expensive and therefore cannot be performed on a mobile device (e.g., a fleet of drones capturing multi-layered images to be processed with machine learning operations such as plate or face recognition); for these delay-sensitive applications, computation offloading represents a valid solution.

Computation offloading is the process of delegating computationally expensive tasks to servers located at the edge of the network; offloading is useful to minimize response time or energy consumption, crucial constraints in mobile and IoT devices. Offloading such tasks to the cloud is ineffective since cloud infrastructure servers are too distant from from the IoT devices. One of the fundamental mechanisms to reduce latency via edge computing is to choose a proper path to the destination; commonly used shortest path algorithms are performance (and so latency) agnostic: they ignore network conditions. One of the hypothesis that we validate in this work is that cooperative routing-based methods can steer (i.e. route or forward), edge traffic with (statistically) lower end-to-end delays than reaction-based methods, such as load balancers (1).

To forecast path metrics, we use machine learning techniques, which in the last few years have affected the way software is made (2). In particular, in this project, we present an architecture for edge offloading orchestration: the architecture design is modular so that the offloading policies could be easily plugged into the system. One effective path prediction policy for the offloading mechanisms that we have implemented is Long Short Term Memory (LSTM), a deep learning approach. Our evaluation shows that our method performs better than the traditional routing policies in terms of throughput.

Contents

List of Tables xi

List of Figures xiii

1 Introduction 1

2 Related Work 5

3 Background 9

3.1 Software-Defined Networking 9

3.2 Knowledge-Defined Networking 10

3.3 Routing 10

3.4 Machine Learning 11

3.4.1 Deep Learning 12

3.4.2 LSTM 12

4 Offloading Architecture 15

4.1 Task Offloading Architecture and Offloading Protocol 15

4.1.1 Offloading Architecture 15

4.1.2 Offloading Protocol 17

4.2 Path prediction via Deep Learning 20

4.2.1 Dataset 22

4.2.1.1 Topology 22

4.2.1.2 Routing information 23

4.2.1.3 Router packet counter 25

4.2.1.4 Dataset Generation 25

4.2.2 Deep Learning Model 27

4.2.2.1 Input/Output modeling 28

4.2.2.2 Neural Network Architecture 29

4.3 Implementation 33

5 Results 35
5.1 Learning from OSPF 35
5.2 Overwriting OSPF Routing 38
5.3 A critical scenario 41

6 Conclusions and Research Directions 43

A Protocol messages implementation 45

B Configure mininet to run the Quagga routing suite 49

Bibliography 51

List of Tables

4.1 Dataset generation parameters. 27
4.2 LSTM architectures comparison. 31
4.3 LSTM architectures average training time. 31

5.1 Path predictions different and equal to OSPF. 39
5.2 Routing strategies retransmission rate comparison. 40

List of Figures

3.1 LSTM cell overview.[1] 13

4.1 Offloading system architecture. 16
4.2 Offloading workflow: mobile devices and an offloading server orchestrate the request via an SDN controller 17
4.3 Model topology: R1-R6 are outer routers while R7-R10 are inner routers. Each router runs a next-hop predictor based on LSTM. 23
4.4 Quagga router implementation in MiniNext. 24
4.5 Router packet counter explanation 26
4.6 Traffic generation algorithm 27
4.7 Recurrent neural network and its unfolded version.[2] 28
4.8 Examples of paths connecting the validation target. 30
4.9 Comparison between training time and accuracy for 128 neurons. 32
4.10 Impact of input normalization on training accuracy and loss. . . 33

5.1 Training accuracy and loss progress on training and validation data. 37
5.2 LSTM and DNN performance comparison. 37
5.3 Routing policies comparison. 39
5.4 Routing policies retransmission comparison. 40
5.5 Comparison of the number of (severely) lossy links traversed by OSPF and LSTM. 41

Nomenclature

AI	Artificial Intelligence
BGP	Border Gateway Protocol
CNN	Convolutional Neural Network
DNN	Deep Neural Network
eBGP	External Protocol
ECMP	Equal Cost Multi-Path
iBGP	Internal Gateway Protocol
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
KDN	Knowledge-Defined Networking
KP	Knowledge Plan
LSTM	Long Short-Term Memory
ML	Machine Learning
NN	Neural Network
OSPF	Open Shortest Path First
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SDN	Software-Defined Networking

Chapter 1

Introduction

Data-intensive computing requires seamless processing power which is often not available at the network-edge but rather hosted in the cloud platforms. The huge amount of mobile and IoT devices that has become available in the past few years, is able to produce a massive quantity of data, introducing several (big) data challenges and opportunities. The majority of these devices do not have or can not handle the computational requirements to process the data they capture, outsourcing to the cloud the responsibility to perform (some or all) computations. This process of transferring computation tasks to another platform is called *computation offloading* and it is crucial to the mobile devices because it results in lower processing time and energy consumption [1]. In critical scenarios, such as natural or man-made disasters, where the physical network infrastructure may be highly unreliable or even unavailable, not only computation offloading becomes necessary, but latency requirements become more strict, making path management solutions, such as (mobile-generated) traffic steering, essential to satisfy application requirements. Current traffic steering or offloading solutions are usually performance-unaware (e.g., OSPF, ECMP), achieving therefore sub-optimal performance.

Data driven networking (3) is a recently adopted paradigm that has been used to fill the performance-unaware gap of many network decision problems.

[1] Despite being a less popular practice, the process of offloading can also be run “backwards“, that is, tasks may be offloaded from the cloud to other (mobile) devices to lighten an overwhelmed cloud server.

The idea behind Data-driven networking [2] is to wonder what data can do for networking, so that the network's control plane can be rethought and redesigned to overcome current limitations. These limitations, caused by *manually designed* control strategies are becoming more and more ineffective because of growing expectations of user-perceived Quality of Experience (4), a bigger decision space for control decisions and an increasing number of application operating conditions. Many researchers have proposed the use of machine learning techniques to solve networking problems, including traffic classification (5), latency prediction (6) and video streaming bitrate optimization (7); to our knowledge, this is the first attempt to use Long-Short Term Memory (LSTM) (8) to train network devices to steer traffic in a virtual network.

Routing is one of the most fundamental networking mechanism and, consequently, has been widely researched to be optimized in a variety of context, such as ISP networks and data centers. Usually route-optimization processes produce configurations based on previously observed traffic conditions, or configurations optimized for a range of feasible traffic scenarios, hoping to cover the entire range. The purpose of using machine learning is to leverage past traffic information to learn good routing configurations for future conditions. It is reasonable to assume that the history of traffic contains information about the future, for example, the traffic distribution at different times of the day. The idea behind data driven networking is therefore to observe traffic demands and adapt future routing decisions accordingly. In this work we develop an architecture to be deployed at the edge of the network to assist the offloading process and make use of machine learning techniques to perform path management. Our hypothesis in this work has been to evaluate whether or not classic (node or link) offloading policies can be outperformed in terms of latency and throughput by learning implicit patterns in network traces.

In particular, in this thesis we make the following contributions:

- we design of an edge computing architecture for (node and link) **offloading** management within edge computing, that is, identifying the mechanisms (i.e. macro blocks) required for such a system
- we propose a simple yet effective protocol for mobile computation (task) offloading

[2] An alternative term used for Data driven networking has been knowledge-defined networking (22, 19)

- we propose a deep learning based **path prediction system** as part of the offloading architecture; this system is meant to be used as an offloading policy in the proposed architecture
- we evaluate the performance of the proposed approach using different use cases: we first evaluate the prediction system on its ability to learn from an existing model, and then as a routing algorithm (replacing state of the art solutions such as OSPF and ECMP).

The rest of this thesis is organized as follows:

Chapter 1 explains the motivation and the purpose of this work.

Chapter 2 illustrates a summary of the related work.

Chapter 3 contains a brief background about machine learning and networking notions and overviews the main techniques used in this project.

Chapter 4 describes in details the architecture and the implementation of the path predictor system.

Chapter 5 shows considerations and results of the implemented system.

Chapter 6 presents comments about the outcome of the project and possible future developments.

Chapter 2

Related Work

In this chapter we describe the works related to this project, the relevant topics are cyber-foraging or task offloading, and machine learning applied to networking.

Cyber-foraging is a highly complex problem since it requires to take into consideration multiple issues. Lewis and Lago (9) dive into those issues and present several tactics to tackle them. Tactics are divided in *functional* and *non-functional*, with the former identifying the elements that are necessary to meet Cyber-foraging requirements and the latter the ones that are architecture specific. Functional tactics cover computation offload, data staging, surrogate provisioning and discovery; non-functional tactics deal with resource optimization, fault tolerance, scalability and security. They describe a simple architecture which include an *Offload Client* running on the edge device and an *Offload Server* running on the surrogate (cloud or local servers).

Wang et. al (10) report the state-of-the-art efforts in mobile offloading. More than ten architectures are described, each one in a different possible offloading situation. In particular the reported works face the single/multiple servers as offloading destination scenario, the online/offline methods for server load balancing, the devices mobility support, the static/dynamic offloading partitioning and the partitioning granularity.

In the last few years machine learning is being used to solve various challenges but it is not being widely adopted in networking problems; however, many people are trying to change this tendency. Malmos (11) is a mobile offloading scheduler that uses machine learning techniques to decide whether mobile computations should be offloaded to external resources or executed locally. A machine learning classifier is used to mark tasks for local or remote execution based on application and network information. To handle the dynamics of the network an online training mechanism is used so that the system is able to adapt to the network conditions. Malmos has proven to have higher scheduling performances than static policies under various network conditions.

The problem of resource management can be effectively addressed with machine learning (12). Generally speaking, resource management is a complex task that requires appropriate solutions depending on the workload. The authors implement a job scheduler based on Reinforcement Learning (RL); the results show a system that performs comparably to state-of-the-art heuristics, responsive to different conditions and with fast convergence.

Machine learning is also being used for computation offloading in mobile edge networks (13). Regression is used to predict the energy consumption during the offloading process as well as the time to for the access point to receive the payload. Available servers are represented in a feature space according to a hyper-profile, then K-NN (K-nearest neighbor) is used to determine the closest server based on metrics related to the hyper-profile. By using K-NN, if an application needs to partition a task into multiple parts onto multiple servers, one should simply vary the value of K.

Kato et. al (14) show the use of deep learning techniques for network traffic control. A DNN (deep neural network) is used for the prediction of a router next hop. The decision is based on the number of inbound packets in a router at a given time and OSPF paths are used for training. By combining the next hop decision for each router the system is able to predict the whole path from source to destination. Results show that the system is able to improve performances in terms of signaling overhead, throughput and average per hop delay with respect to the classic OSPF algorithm.

Another use of machine learning in networking is described in (6). Bui, Zhu, Pescapé & Botta designed a system for a long horizon end-to-end delay forecast. The idea is to use measured samples of end-to-end delays to create a model for long horizon forecast. Considering the set of samples as a discrete-time signal, wavelet transform is applied which results in two groups of coefficient. A NN (neural network) and a K-NN classifier are then used to predict the coefficients. Once again ML techniques seem to provide good results when applied to networking.

Valadarsky et al. (15) discuss the advantages of a data-driven approach to routing, using two different machine learning techniques, in an intradomain routing case study. Two scenarios are considered: learning future traffic demands from past patterns or learning optimal routing configuration. To predict traffic demands supervised learning is used, on the other hand, for the prediction of routing configurations, the idea is to use reinforcement learning. The results show that while supervised learning might be ineffective are irregular, reinforcement learning is much more promising to generate effective routing configurations.

Chapter 3

Background

Before describing our approach, we give a brief introduction of some of the methods and concepts that are needed to understand our work. Specifically, we first shortly describe the network technologies required to understand the aim of this project; next, we give an introduction about machine learning and its applications.

3.1 Software-Defined Networking

Software-Defined Networking (SDN) is paradigm that promises to break networks' vertical integration of control and data plane, separating the network's control logic from the underlying routers and switches, promoting (logical) centralization of network control, and introducing the ability to program the network. The separation of concerns introduced between the definition of network policies, their implementation in switching hardware, and the forwarding of traffic, is key to the desired flexibility: by breaking the network control problem into tractable pieces, SDN makes it easier to create and introduce new abstractions in networking, simplifying network management and facilitating network evolution (16). SDN is loosely built on four principles:

1. separate control and data planes: network devices are left responsible only of packet forwarding

2. generalized (or flow-based) forwarding: packets are forwarded by looking to a set of fields instead of only the destination, introducing greater flexibility (17)

3. external control logic: the control logic is moved to an external entity called SDN controller or Network Operating System (NOS), that is a software platform that provides abstractions to facilitate the programming of forwarding devices

4. programmable network: software running on top of the NOS allows to program the network; this is considered the main value of SDN.

SDN has successfully opened the way towards a next generation networking, creating an innovative research and development environment, promoting advances in switch and controller platform design, evolution of performance of devices and architectures, security and dependability (18).

3.2 Knowledge-Defined Networking

Knowledge-defined networking (KDN) (19) is a new paradigm that promotes the application of Artificial Intelligence (AI) to control and operate networks thanks to the rise of SDN. SDN provides a centralized control plane, a logical single point with knowledge of the network; moreover, current network devices have improved computing capabilities, which allow them to perform monitoring operations commonly referred to as network telemetry (20). Information provided by network telemetry are usually provided to a centralized Network Analytics (NA) platform (21), that combined with SDN can bring to light the Knowledge Plane proposed in (22). Knowledge-Defined Networking is the paradigm resulting from the combination of these tools, specifically Software Defined Networking, Network Analytics and Knowledge Plane. In the KDN paradigm, the knowledge plane has a rich view of the network; this view is transformed into knowledge via Machine Learning (ML) and used to make decisions. The ultimate goal of KDN is to combined SDN, Network Analytics and Machine Learning to provide automated network control.

3.3 Routing

In networking, routing is the process of selecting a path in or between networks. Routing directs network packets from their source toward their destination through intermediate network nodes by specific packet forwarding mechanisms. Forwarding is performed on the basis of routing tables, which maintain a record of the routes to various network destinations. Thus, constructing routing tables, is very important for efficient routing. Dynamic routing constructs routing tables automatically thanks to the information carried by the routing protocols, that are usually classified in *distance vector* and *link-state* algorithms.

BGP (Border Gateway Protocol) is a protocol designed to exchange routing and reachability information among or within Autonomous Systems (AS). An average AS is made by about 500 point of presences, and together ASes glue the Internet together. When used to route packets across ASes of the same Internet Service Provider (ISP) BGP is also referred to as Internal BGP, or iBGP. In contrast, to route across ASes BGP uses other "External BGP" protocols, or eBGP. BGP is typically used by ISPs to establish routing between one another, or in large private networks to join a number of large networks. Routing informations are exchanged between neighbor routers (or peers); to communicate, these peers often require a manual configuration.

OSPF is a link-state algorithm and it requires IP. OSPF falls in the family of iBGP and it is widely adopted in large networks. The majority of ISP use a version of Open Shortest Path First (OSPF). It works thanks to a map of the network, built by gathering link state information from available routers. The maps is used to compute the shortest-path tree for each route using a method based on Dijkstra's algorithm (23). The OSPF routing policies are dictated by link metrics associated with each routing interface, typically the interface speed.

3.4 Machine Learning

Machine learning is a field of computer science that studies the ability of making computers learn without explicitly programming them (24). In machine learning there are three main approaches: supervised learning, unsupervised learning and reinforcement learning. Supervised learning is used to classify labeled data, where the label is a sort of supervisor describing the class of an observation. In unsupervised learning, data is unlabeled and the goal is to find the hidden relation among data records. Reinforcement learning is learning what actions to apply so as to maximize a numerical reward signal without being told which actions to take but instead discovering which yield the most reward by trying them (25). Among the several branches of machine learning, neural networks and in particular deep learning, have recently attracted a lot of attentions.

3.4.1 Deep Learning

Teaching a computer to solve tasks that are hard to describe formally (e.g., speech recognition or routing) is challenging; the solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, that is, it has many layers (26). Deep learning can be described as the set of machine learning techniques that concatenate multiple layers of processing units (typically non-linear), for feature extraction and transformation (14).

The first working deep learning algorithm was a multilayer perceptrons developed by Ivakhnenko et al. (27). Later on, new techniques including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) were developed; these techniques dramatically increased Artificial Intelligence performance in tasks such as image recognition. More recently, deep learning has been used to successfully play games, see AlphaGo (28).

3.4.2 LSTM

Long Short-Term Memory (LSTM) is a further development of Recurrent Neural Networks (RNNs). They were developed by Hochreiter and Schmidhuber in 1997 and have been further improved ever since (29). The major flaw of a traditional neural network that RNN improves, is that it does not capture the relation between the input it currently looks at and the previous training example. Such relation is, however, crucial for example to develop a (human) language model. Making predictions about a subsequent word strongly depends on the semantic of preceding words. In practice, RNNs are also not scalable when trying to model long-term dependencies. This is due to numerical problems commonly referred to as the *vanishing/exploding gradient* (30) when weight updates are back-propagated through the time steps. During the backpropagation phase, the weights of the network are updated according to the computed gradients; the magnitude of gradients is affected by the weights and the activation functions. If either of these factors is smaller than one, then the gradients may vanish in time when propagating to the first layers of the network; similarly, if larger than one, then exploding might happen. LSTMs overcome this problem and enables capturing long-term temporal dependencies among the input elements. LSTMs are considered state-of-the-art in numerous sequential prediction tasks such as Speech Recognition, Handwriting Recognition, Language Translation and many others.

The novelty of LSTMs compared to conventional RNNs is the introduction of the *LSTM cell*. Figure 3.1 gives an overview over such a cell that is repeated three times, each receiving the current input as well as the output of the previous cell. For instance, the prediction h_{t+1} (through feed-forward) is based on the corresponding input x_{t+1} as well as on the output of the previous cell. The LSTM cell is responsible for maintaining and updating a state that keeps track of the input that has been processed over time. Which information precisely should be kept and which overwritten is decided during the training phase of the RNN. Each cell has associated weights that are updated during each backward pass such that the cell keeps the information that optimizes predictions. The major advantage of introducing this cell is that the *Back-propagation Through Time* does not need to flow through numerous activation gates between the hidden layers. The transfer from cell to cell only flows through pointwise multiplications and additions. This way the numerical problems of RNNs are avoided.

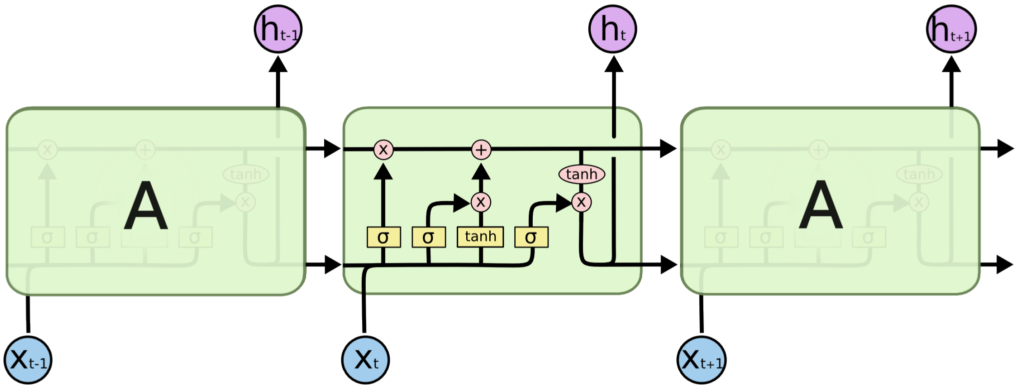

Figure 3.1. LSTM cell overview.[1]

[1] Taken from colah's blog: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Chapter 4

Offloading Architecture

This chapter is organized as follows: section 4.1 illustrates the task offloading architecture and its implementation, section 4.2 describes the path prediction system and its details and finally, section 4.3, summarizes the tools adopted in the development of the project.

4.1 Task Offloading Architecture and Offloading Protocol

Edge computing is a recent computing paradigm whose main idea is to push (i.e. delegate) data processing at the edge of the network for fast pre-processing within latency-sensitive applications. The goal is to reduce communication costs by keeping computations close to the source of data.

4.1.1 Offloading Architecture

Our proposed architecture is described in Figure 4.1; we consider a scenario in which mobile devices wish to offload tasks to the edge cloud in network that supports SDN.

The main components are:

- a mobile device interface: interface for communications between the mobile devices and the offloading system

- an edge cloud interface: interface for communications between the edge cloud and the offloading system

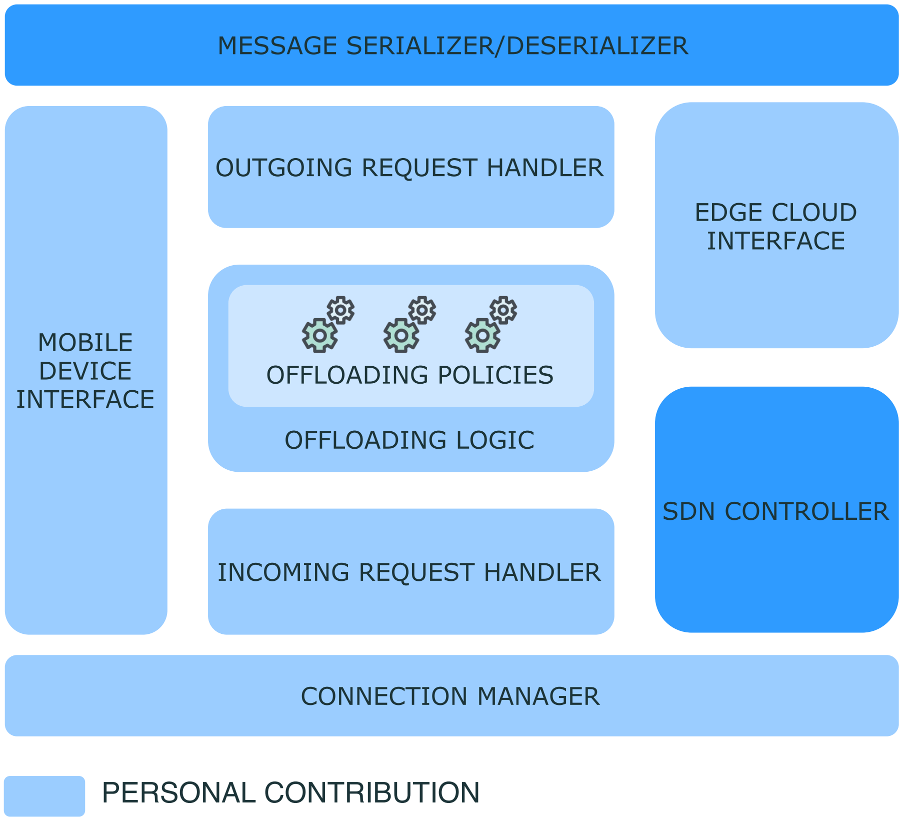

Figure 4.1. Offloading system architecture.

- offloading logic: offloading policy to be used to serve the edge client
- SDN controller: responsible of enforcing the offloading policies on the network devices.

The mobile device interface supports the communication between the mobile devices and the offloading mechanisms, providing a set of primitives necessary to the two parties to communicate efficiently and formalizes the offloading request requirements. The edge cloud interface has the same role of the mobile device interface, but the involved parties are different: in this case the primitives are meant for the communication between the offloading system and the edge cloud, but the objective remains the same. The offloading logic is the main part of the architecture, it contains the set of policies available as offloading criterion, allowing the mobile devices to specify which one they intend to use and users to implement their own. Finally, the architecture includes a SDN controller, that we believe, must be part of it, because of the numerous possibilities that SDN offers in terms of network management.

4.1.2 Offloading Protocol

Through the edge and cloud interfaces, the parties communicate in order to complete a task offloading process. For this communication to happen, a protocol is required. In this context, the communicating parties are:

- mobile device: sends request to the offloading server
- offloading server: accepts request from the mobile device, enforces the offloading policy by talking to the SDN controller and forwards the tasks to the edge server
- SDN controller: receives flows installation requests from the offloading server
- edge server: receives the tasks that need to be offloaded.

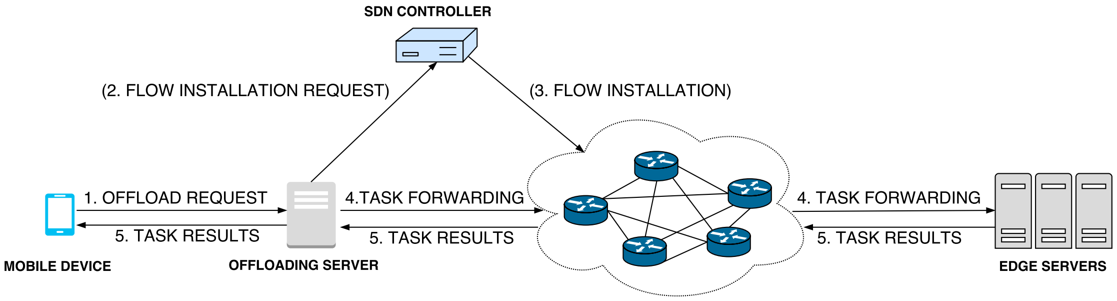

Figure 4.2. Offloading workflow: mobile devices and an offloading server orchestrate the request via an SDN controller

Figure 4.2 shows the typical message sequence needed to complete an offloading request; the required steps are the following:

1. the mobile device sends an offload request to the offloading server
2. the offloading server decides whether or not it is possible to accept the request and if so, where to offload it according to a set of configurable, i.e., programmable policies

3. the offloading server asks the SDN controller to install on the switches the flows required by the offloading policy [1]

4. the offloading server forwards the offloaded task to the edge server

5. the edge server sends the computation result back to the offloading server, that in turn forwards it to the device.

Task offloading is a complex problem, with a high number of factors involved in the final decision. In the literature, task offloading is also known as cyber-foraging (31). These factors include application requirements formalization, task retrieval, edge server/cloud discovery. Before explaining the details of our protocol (implementation), we describe some of these problems and how our protocol addresses them.

Application requirements formalization

We assume that the application requirements are expressed in terms of **CPU ratio** (average CPU used to execute the task on the mobile device), **memory footprint** (quantity of memory required by the application) and desired **latency** (specifying if the task, is urgent or it can wait).

Task retrieval

For the task retrieval problem we consider two different scenarios: in the first one the task is hosted on the server with the possibility to retrieve it with a unique identifier; in the second scenario, the task is sent to the server, wrapped in a container (e.g., a java package JAR or a python package EGG).

Edge servers discovery

The implementation of a server discovering protocol is out of the scope of this project: we assume that the offloading server is aware of the available edge servers. Our implementation over mininet uses a Link Layer Discovery Protocol (LLDP).

Abstract Syntax Notation With these assumptions in mind, we now describe the protocol messages definition using the Google Protocol Buffer (32)

[1] This step is optional because the required flows may be already installed

abstract syntax notation; Google Protocol Buffer is a library that allows to easily specify structured messages through a *.proto* file. The complete .proto file is available in the Appendix A at the end of the report; what follows is a brief overview of the protocol main messages. An abstract syntax notation is necessary to provide computer architecture independence as well as programming language independent, and so to abstract out the implementation details. This means that users may use our .proto files to write their own server or client applications using for example, Java with a big-endian architecture for a server, and C++ with a little-endian computer architecture on the client. Historically, abstract syntax notations have been using ASN 1, and more recently, developers use XML or JSON. XML or JSON are however text-based, while Google Protocol Buffer is binary based, and so serialize and deserialize messages is more efficient with Google Protocol Buffer. An alternative could have been BSON, but Google Protocol Buffer provides already compilers for many languages so it has been our choice. We now describe the specifics of each message.

OffloadRequest is the message sent from the mobile device to the offloading server and includes the requirements, the type of the task and optionally the task itself.

The **requirements** consist of the specification of the amount of cpu and memory necessary for the task to be executed and the level of latency needed. The idea of having a latency level allows a better support for real-time applications.

The **type** of the task is needed to support serverless computing (33) (34) and implies two possible scenarios. In case of serverless computing the type of the task is set to *LAMBDA* and it is assumed that the task is already on the server. If the type is set to *STANDARD* then the code to be executed is sent as payload of the request.

The **task** could be an identifier (*task_id*) of the task on the edge server (in case of a lambda application), or the task itself. In the latter case, the executable and its type are included in the request in a message called *TaskWrapper*.

Response is a message used for several purposes; it is used to confirm the reception of a message or to signal an error. `Response` is also used to send back to the mobile device the result of the computation. It is important to notice that protobuf messages are not self-delimiting, that means that it is necessary to know its size in order to receive it. Every time a new message needs to be sent, the sender starts with the message size first, if an OK response is received then the message is sent.

Message is a wrapper for all the messages used in the system, its purpose is to ease the parsing process by having a *type* field that can be one of the messages defined above.

4.2 Path prediction via Deep Learning

Why Deep Learning? Deep learning is currently one of the most active area of research. It has proven to be able to solve numerous problems in different fields, but it is not adopted to solve networking problem as much as in other contexts. In this work, we decide to use deep learning for two reasons: first of all, deep learning has shown great computation performance, especially for complex problems, furthermore, we want to understand if its poor adoption in networking is due to a lack of interest in the area or because it is not the right tool in this context. Before proceeding with this chapter, it is important to keep in mind that the word deep in deep learning, is used every time a neural network has more than one layer. The architecture described in Section 4.1.1 includes the *offloading logic* block, responsible of determining the criterion on which the offloading is based. As a first offloading policy, we implement a deep learning model capable of finding the path towards a destination, that should be used in place of traditional routing policies.

One of the problems of traditional routing algorithms is that they do not consider how the network load changes over time (they rely on TCP for congestion and flow control, and hence, by design they are performance-unaware): the path from a source to a destination is computed by taking into account static parameters, such as the nominal interface speed. This lack of consideration of the dynamic behavior of the network can cause traffic slowdown and packet loss in case of congestion, an undesired behavior in circumstances where latency is crucial. End-to-end congestion control when latencies are so important or when the network is so disruptive (in case of a natural disaster) is not enough. The intuition behind our project is that *collaborative traffic steering* should be able to identify and avoid congestion situations, without using TCP or other active queue management approaches such as Explicit Congestion Notification (ECN). A collaborative policy requires, in some way, the participation of all the parties in the decision process, however, achieving nodes collaboration in a network is a complicated task. A simplified version of this collaborative mechanism could be the following: instead of all nodes being directly involved in the decision process, they could limit their role to notifying a central node of their current status. This is made possible by the SDN paradigm: in our system the nodes are connected to a SDN controller responsible of retrieving information about the nodes status, in this case using the OpenFlow protocol (35). The information used in our system is the number of incoming packets on a node: the idea is that the packet distribution on the nodes, routers in this case, reflects the network conditions; a high packet count on a router is an indicator of a big load that is probably going to lead to packet loss and retransmission. Another point to clarify before proceeding with the implementation details, is that the distribution of packets on the routers is influenced by the routing algorithm: nodes that appear in multiple paths will probably have an higher count than less traversed nodes because they are responsible of forwarding packets for multiple source-destination pairs. If routers were able to see all possible outcomes of a routing protocol in a network and extract the consequent traffic patterns, they could try to choose the less busy path. This is exactly how we build our deep learning model: we simulate a small network with ten routers, we choose a routing algorithm (OSPF), we make it vary and record the traffic patterns. Afterwards, we use the collected data and the routing choices taken by the routing protocol to build a model capable of predicting each hop of the path, from the source to the destination. Learning from the algorithm we criticize and aim to replace may sound confusing and counterintuitive, but learning is different from copying. With our approach, we are correlating traffic patterns and routing decisions, replicating as many different scenarios as possible, so that the final model will have a complete view of the problem. The correlation between traffic patterns and routing allows are system to dynamically adapt to the network conditions, a behavior that wouldn't occur with a traditional routing algorithm.

For this problem, to create a working deep learning model, two elements are essentials: the dataset and the network architecture; the following sections describe the type of data, how we obtain them and what neural network architecture we select.

4.2.1 Dataset

The performance of a machine learning model depends heavily on the data used to train it. Creating a working model requires having good data in terms of quantity and quality, namely the number of samples available to train the model and how well these samples represent the domain you are trying to model. Training a model with a dataset too small usually results in poor performance, because there are not enough information for the model to learn from, causing the system to deal with unknown scenarios; on the other hand, even a big dataset that only covers a small portion of the domain of study, will perform poorly because the system, will not be able to learn all of the different scenarios, therefore losing the ability to generalize and creating an abstraction. Ideally, the perfect dataset would be representative of the whole domain we are modeling.

Recently a lot of effort is being put in making datasets available to the machine learning community, with the aim of promoting research. In the context of networks, there is plenty of dataset available (36)(37)(38); however those are mostly network captures of datacenters or small portions of networks, and they do not contain details about the underlying topology nor the routing strategies.

The elements needed to implement our path prediction system in a given a network are:

1. the network topology
2. the routing informations
3. the packet count on each node.

Since we could not find a dataset fit to our needs, we have created our own.

4.2.1.1 Topology

The first component of our system is the network: we need to emulate a network from which we can extract the data necessary to train the model and to use to test the final model. Mininet (39) is a software commonly used in the network virtualization research community, that can create a virtual network running real kernel, switch and application code with support for OpenFlow and Software-Defined Networking (35).

Figure 4.3 shows the topology used in our system: it is composed of ten routers, R1-R6 which we denote from now on as *outer routers*, and R7-R10, from now denoted as *inner routers*. The topology includes also fourteen switches, the reasons of their presence is explained in section 4.2.1.3. For simplicity we assume that traffic is being generated only by the outer routers, therefore the inners' are only responsible of forwarding other nodes packets. The topology is a partially connected network with multiple paths connecting the same source-destination pair; note that having multiple paths is crucial for the path prediction system: in this way, during the training phase, our path prediction algorithm will learn several different alternative paths from source A to a destination B.

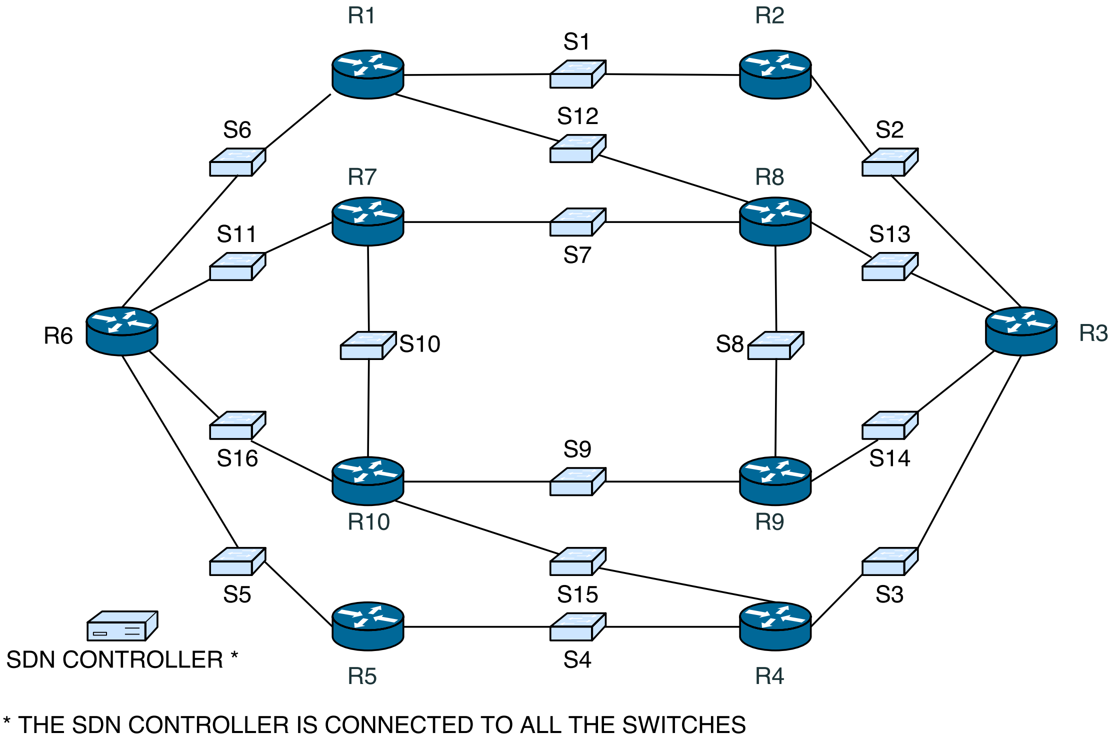

Figure 4.3. Model topology: R1-R6 are outer routers while R7-R10 are inner routers. Each router runs a next-hop predictor based on LSTM.

4.2.1.2 Routing information

The default behavior of Mininet is to create a layer 2 topology, with all the nodes acting as a switch or a host in a local network. Our objective is to build a path prediction system that learns from the routing algorithms, hence we need to build a fully functional level 3 network with level 3 routers. The easiest way to use Mininet as a layer 3 network is through MiniNext (40), a mininet extension layer that support routing engines and PID namespaces. As a routing engine we use Quagga (41), a routing suite providing implementation of routing protocols for Unix platforms. The Quagga architecture consists of a core daemon, zebra, that acts as an abstraction layer to the underlying Unix kernel and presents the Zserv API over a Unix or TCP stream to Quagga clients. It is these Zserv clients which typically implement a routing protocol and communicate routing updates to the zebra daemon. MiniNext allows us to place routers into a mininet topology; the way it does this is by instantiating a regular Mininet host node in a separate namespace and starting a routing process on it: different namespaces are necessary so that each host can run the routing process without interfering with the others. Figure 4.4 shows an outline of this architecture: each Mininet host runs in a separate namespace and runs the zebra and the routing daemons; each zebra daemon updates each router routing table with the routing informations exchanged through the virtual network built with the Linux Kernel.

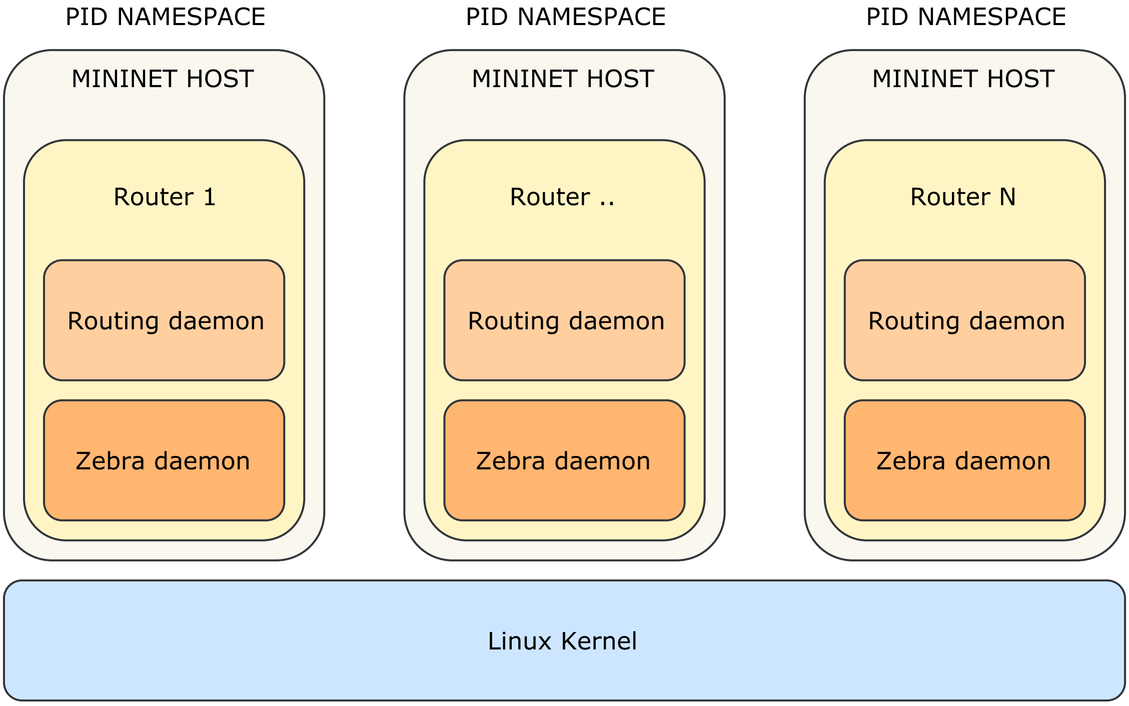

Figure 4.4. Quagga router implementation in MiniNext.

4.2.1.3 Router packet counter

In section 4.2.1.1 we have described the topology without giving an explanation of the presence of a switch between every pair of connected routers. The reason for the switches is that we want this system to work in a SDN context: the only way to use a SDN controller in Mininet is to use switches. There is also a secondary reason that justifies the switches presence: unlike the routers, switches do not run in a separate namespace, allowing traffic analyzer tools such as Wireshark to capture traffic on all the network interfaces at once. Because of the separate PID namespaces, if we wanted to capture traffic directly on the routers interfaces we would need a Wireshark instance for each router, that would make debugging extremely hard.

An SDN controller can retrieve switches information with the OpenFlow protocol; in this project we use Ryu (42), an SDN controller written in Python. Since the controller is connected to the switch, getting the packet count on the router is not trivial. On an OpenFlow enabled switch it is possible to get the number of packet received on each port; given that each router is directly connected to at least one switch, it is possible to infer the number of packets directed to the router by counting the number of packets coming from a certain port of the switches it is connected to. Figure 4.5 explains this concept with an example: in the figure, the number of incoming packets on R1 at a given time is the sum of the incoming packets on port1 of S1 and port2 of S2. By repeating this procedure for all the routers in the network, the controller is capable of retrieving, at any given time, the number of incoming packets on each router.

4.2.1.4 Dataset Generation

Up to this point we have a fully functional level 3 network, with a running instance of the OSPF routing protocol, connected to a controller capable of retrieving the packet count on each router when needed. To build our system we need a set of labeled data from which the model can learn. More specifically, we need a collection of samples containing at a given time, the packet counting information together with the routing information. The detail about the data format and how they are used are described in the following section 4.2.2; in the rest of this paragraph we describe how we generate these data.

The idea is to simulate the traffic conditions in a regularly functioning network, collect the traffic and routing information, and use it to train the network.

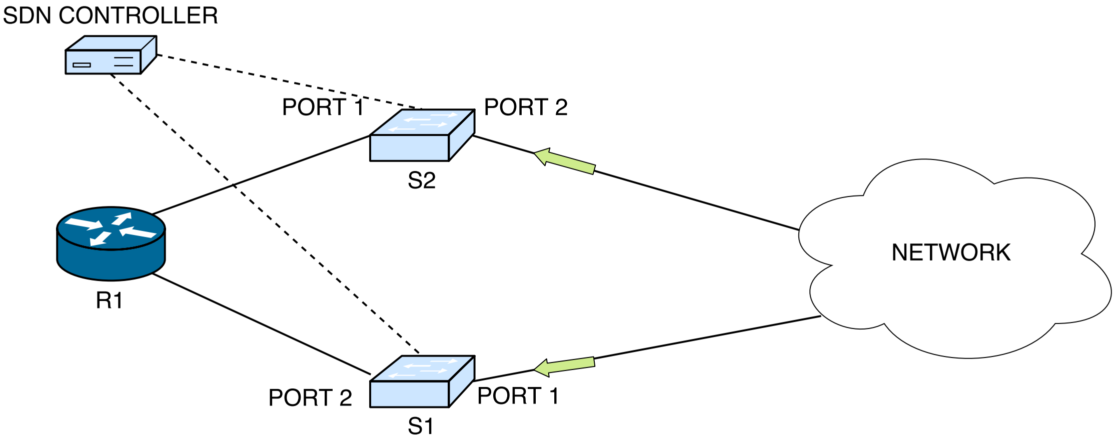

Figure 4.5. Router packet counter explanation

By default, a mininet network is static, in the sense that there is no traffic unless manually generated. One way to generate traffic is by using *iperf* (43), a tool for bandwidth measurement in IP networks. As a second step, we need to change the output of OSPF over time to explore the various paths in the network. OSPF computes each path cost according to the speed of the network interface; therefore, changing such speed is sufficient to force OSPF to compute new routes.

We generate our training dataset as follows: for *ospf_configurations* times, every *ospf_conf_duration* seconds, the network is teared down and rebuilt with a new link speed, producing a new OSPF configuration; for each OSPF configuration, traffic is generated in the network by running the *iperf* Linux utility between every source-destination pair, with a certain *transmission_probability* and a *transmission_time* duration. In the meantime, the controller retrieves the packets count every *sampling_time* seconds and saves it to a file, while another script saves the routing tables of each router, every time that the OSPF configuration changes. The pseudo-code of the algorithm is shown in Figure 4.6 while table 4.1 describes the algorithm parameters.

Using the parameter set of table 4.1, we obtain a dataset of 17696 samples; 85% of these samples form the training set, and the remaining 15% is used as test set.

Figure 4.6. Traffic generation algorithm

```
for all ospf_configurations do
    for all (src, dst) pairs do
        $p = random(0, 1)$
        if $p < transmission\_probability$ then
            $t = randint(0, transmission\_time)$
            run iperf for t seconds
        end if
    end for
end for
```

4.2.2 Deep Learning Model

The deep learning model chosen for this project is a LSTM, a Recurrent Neural Network (RNN): recurrent neural networks are a class of neural networks in which connections between nodes form a directed cycle (Figure 4.7). These connections allow the nodes to memorize information about what has been computed so far. The reason we choose RNNs is because of their ability to make use of sequential information and to exhibit a dynamic temporal behavior. We wanted our model to learn the correlation between changes in the packet distribution and routing decisions over time.

So far we have discusses our machine learning model as if a single instance would be able to predict every path for every possible source-destination pair; however, given the complexity of routing, this scenario did not seem feasible. Our solution is to train a separate model for every (s, d), resulting in several simpler models rather than a single, very complex one. To give an idea of the order of magnitude of this approach, a network with N nodes will result in $N(N-1)$ models. These numbers need to be adjusted if each node has, like

Parameter	Value	Description
ospf_configurations	15	number of different OSPF configurations
ospf_conf_duration	20 m	time after which a new ospf configuration is produced
transmission_probability	0.65	probability with which there is traffic between a pair of routers
transmission_time	0-5 s	time in seconds to transmit for (iperf)
sampling_time	1 s	sampling time of the packet count

Table 4.1. Dataset generation parameters.

in our case, more than one network interface; in this scenario, there will be more models with respect to the previous case, each one describing a particular $(src\_router, dst\_address)$ pair.

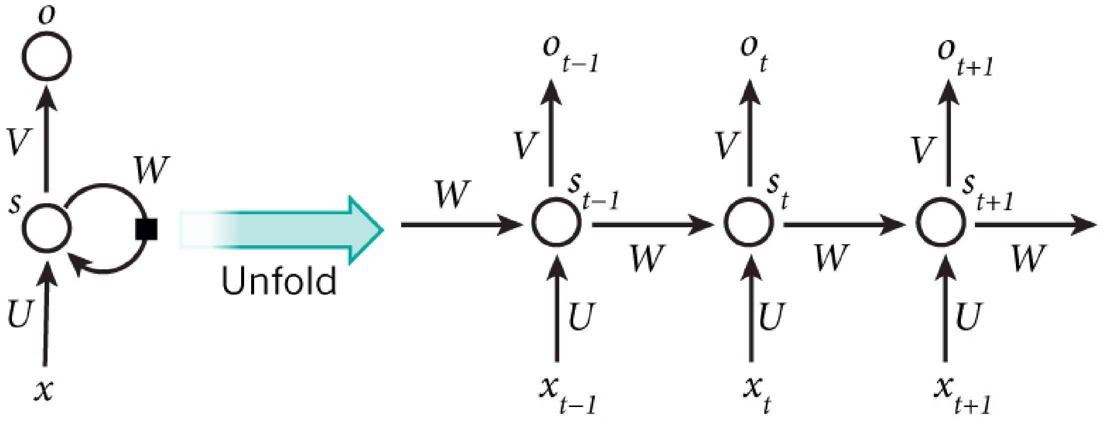

Figure 4.7. Recurrent neural network and its unfolded version.[2]

4.2.2.1 Input/Output modeling

A machine learning model requires a proper representation of the input and output; supervised learning involves a sample space X and a label space Y, with the network responsible of learning a mapping function from values in X to labels in Y, for each $(x_i, y_i) \in X \times Y$. Our input/output modeling follows the same approach described in (44): given O the set of outer routers, and R the set of all the routers in the network, for each $(s, d) \in R \times O$, the system learns the next hop for that particular target destination. Note that the fact that inner routers are included in the set of sources nodes is not a conflict with what stated in 4.2.1.1: even though these routers do not generate traffic, they are still responsible for forwarding packets coming from other sources.

Each model learns the next hop for a particular destination, given the packet count on each router at a given time ***t*****:** the easiest way to model the input is an *N-dimensional* array, with N being the number of

[2] Picture from: <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>

routers in the network. The array is indexed by the router number, so the *i-th* element of the array is the number of incoming packets on router *i*. The output is modeled as a one-hot encoded, router indexed array, with a 1 in the position indexed by the predicted next hop; the size of the output is again equal to the number of routers in the network.

4.2.2.2 Neural Network Architecture

The architecture of a deep neural network is determined by the number of layers, the number of processing units (neurons) per layer and the interconnections between the layers. Choosing these parameters once at the beginning, hoping to achieve good performance, is not feasible given the impossibility to derive them from a formal description of the problem; thus, these parameters need to be tuned in a preliminary phase. As a general rule, a network too small will not be able to solve the problem and a network too big will probably overfit on the training set; there is also consensus on the fact that for the majority of the problems, adding additional hidden layers does not significantly improve the performance.

To define the parameters of our network, we follow these three steps:

1. pick a source-destination pair that requires a complex model

2. cross-validate each combination of layer and neurons

3. choose the combination whose accuracy – loss pair is best.

It is important to clarify what we mean with *"source-destination pair that requires a complex model"*: cross-validating the different architectures on all the possible pairs would be excessively time-consuming, thus we decide to perform the validation on a single target. From now on, a target is simply a source-destination pair. For this validation to be meaningful, we need our target to be representative enough of the problem we are modeling: since we want our system to learn alternative paths, it would not make sense to choose a target of two directly connected nodes, because the resulting model would be too simple. As a consequence, we choose the target $(R1, R4)$; Figure 4.8 illustrates some of the 3-hop paths connecting the two routers.

As it is noticeable from the figure, there are several paths with the same number of hops connecting the two routers, so, in the context of choosing a representative model, the selected target seems a good candidate.

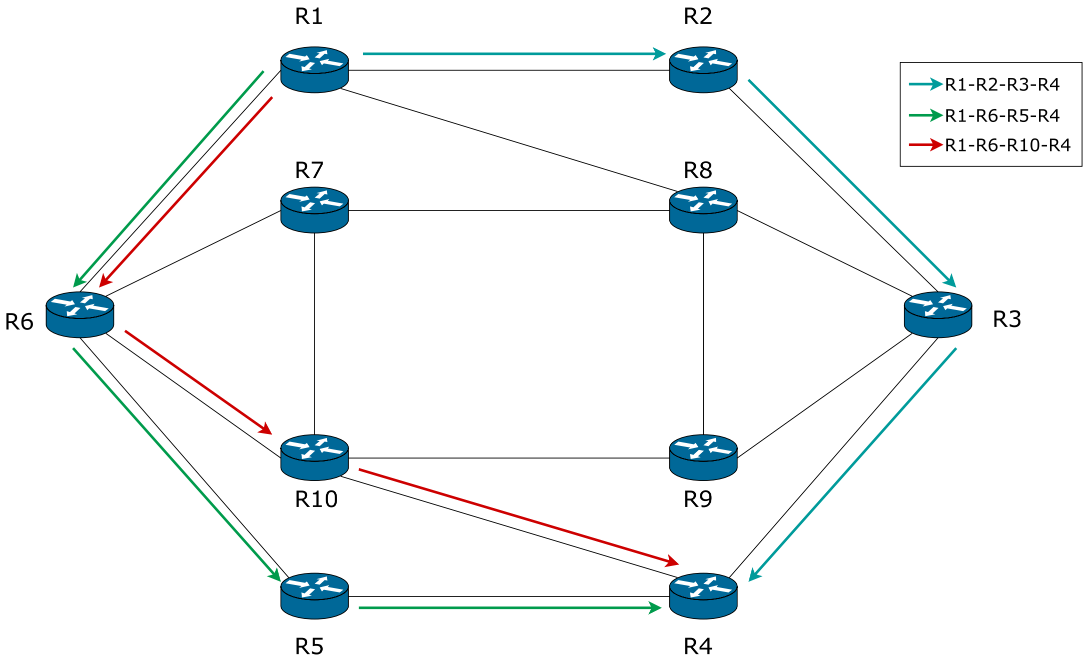

Figure 4.8. Examples of paths connecting the validation target.

In the cross-validation phase, we test 24 different configurations by trying all the combinations of the following parameters:

- $hidden\_layers = \{2,4,6,8\}$
- $neurons = \{4, 8,16,32,64,128\}$.

It is important to remind that each hidden layer is a recurrent layer with LSTM cell. Each configuration is tested 10 times on different partitions of the dataset, producing the results in table 4.2. The table shows two metrics: accuracy (percentage of samples correctly classified) and cross-entropy loss (distance between predicted and true label distribution); it is evident that what we described earlier about how adding layers does not significantly improve the performance, applies in this case. All the noticeable improvements in the results are caused by increasing the number of neurons in each layer (as you may notice by reading the table line by line); instead, if you read the table column wise (fixed number of neurons and increasing number of layers), you will not notice any performance gain. A 4-layers 128-neurons achieves the best performance in terms of accuracy whereas a 4-layers 128-neurons has the lowest loss; it is worth noticing that overall, given a fixed number of neurons, the performances typically differ by less than 1% in accuracy.

	Neurons											
Layers	4		8		16		32		64		128	
	Accuracy %	*Loss*	*Accuracy %*	*Loss*	*Accuracy %*	*Loss*	*Accuracy %*	*Loss*	*Accuracy %*	*Loss*	*Accuracy %*	*Loss*
2	86.59%	0.5147	88.82%	0.3946	92.95%	0.3026	94.85%	0.2159	95.60%	0.1659	96.08%	0.1368
4	76.48%	0.6416	87.69%	0.4644	92.99%	0.2435	94.90%	0.2045	95.70%	0.1554	**96.58%**	0.1214
6	64.90%	0.7727	88.82%	0.4450	92.72%	0.2515	95.25%	0.1663	95.38%	0.1541	96.14%	**0.1149**
8	65.84%	0.7953	87.42%	0.4914	91.01%	0.3340	95.08%	0.1718	95.83%	0.1374	95.73%	0.1361

Table 4.2. LSTM architectures comparison.

From this first analysis it is clear that to achieve the best accuracy, each layer of the network needs to have 128 processing units; however, deciding what is the optimal the number of layers is a challenging problem. Since the gain in performance with an increase of the number of layers is not noticeable, we decided to take into account other factors to choose the final architecture. Table 4.3 shows the average training time for the different configurations: in this case the impact of additional layers is evident. The training time seems to vary linearly with the number of layers: if we double the layers we basically double the training time. Figure 4.9 compares the different training times with the model performance in terms of accuracy; it is clear that while the training time increases noticeably with additional hidden layers, the gain in accuracy is barely noticeable and sometimes inexistent. Considering the limited computational power and time, and the number of models we need to train, we decide to use a network with 2 layers and 128 neurons, as a good trade-off between performance and time.

	Neurons					
Layers	4	8	16	32	64	128
2	301.20 s	304.82 s	314.56 s	326.72 s	388.46 s	623.84 s
4	487.49 s	499.91 s	516.06 s	557.60 s	675.94 s	1129.96 s
6	686.25 s	705.11 s	722.09 s	781.28 s	961.83 s	1649.99 s
8	905.18 s	931.16 s	972.52 s	1029.73 s	1275.34 s	2121.93 s

Table 4.3. LSTM architectures average training time.

As a result of this analysis, the final architecture is composed as follows:

- input layer (10 neurons)

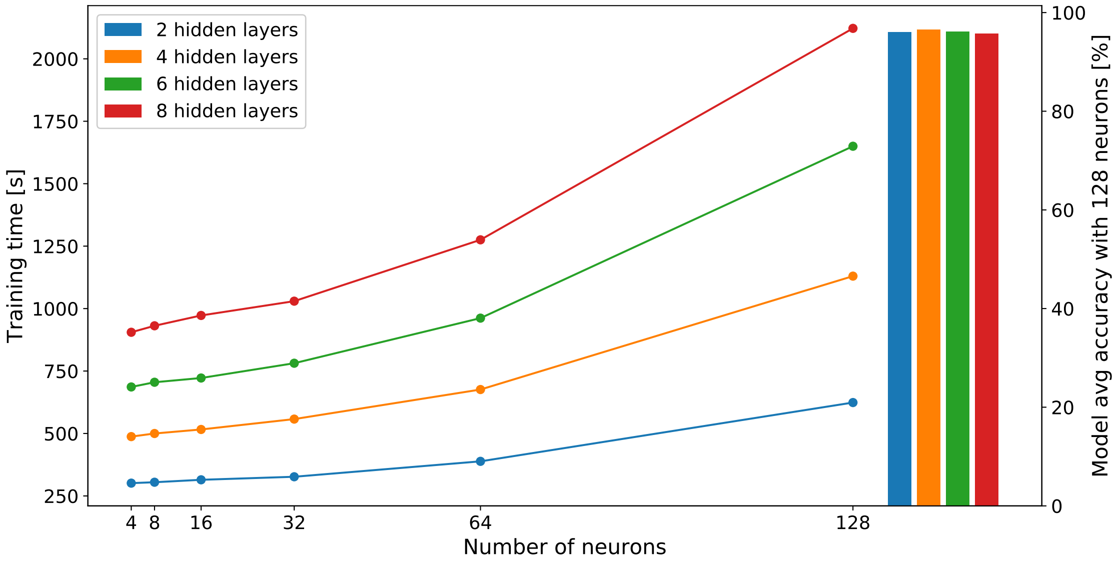

Figure 4.9. Comparison between training time and accuracy for 128 neurons.

- two hidden layers (128 neurons ea., hyperbolic tangent activation [3])
- output layer (10 neurons, sigmoid activation [3])

Challenges of training a neural network. When training a neural network, there are some precautions necessary to avoid problem such as overfitting and increase training efficiency, avoiding exploding and vanishing gradient problems. Input normalization is the process of transforming all variables in the input data to a specific range, typically scaling the inputs to have mean 0 and a variance of 1, in order to have the same range of values for each of the inputs, guaranteeing stable convergence of the network weights and biases. Regularization is a technique to prevent overfitting, that is characterized by adding a regularization term to the loss in order to prevent the weights to fit perfectly to the training data, so that over all model is much more generalized. Dropout (45) is another technique to address the overfitting problem; the idea is to randomly drop units and their connections from the neural network during training, to prevent units from co-adapting too much. In our case, we apply a batch normalization layer on the input, dropout on both the recurrent and feed-forward connections, and $L2$

[3] The activation function defines the output of a node given an input (26)

regularization to the loss function. Figure 4.10 shows the impact of normalization on the training performance: for both accuracy and loss, the normalization layer dramatically improves the performance, with a gain of about the 20%.

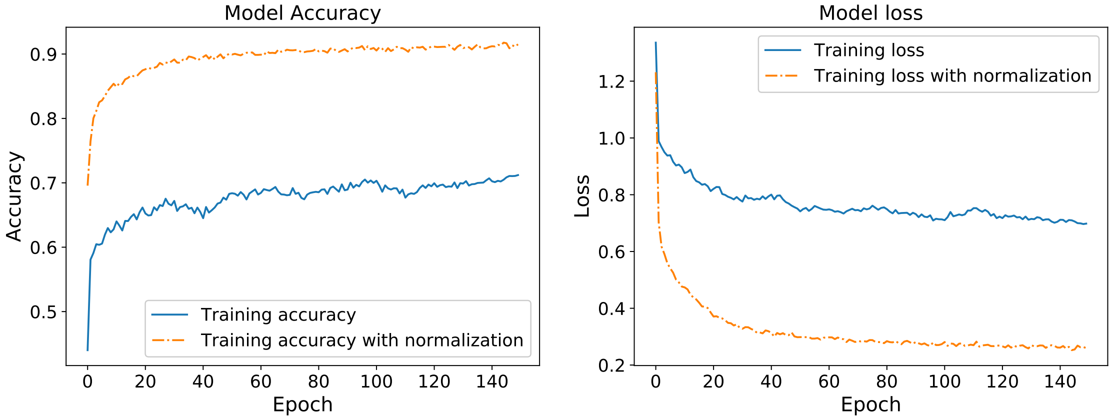

Figure 4.10. Impact of input normalization on training accuracy and loss.

4.3 Implementation

To prototype our system we use the following tools:

- Ryu: SDN controller written in Python (42)
- Google Protocol Buffer: framework for serializing structured data (32)
- Mininet: virtual network creation tool (39)
- Quagga: routing software suite (41)
- Keras: high-level neural networks API written in Python (46)
- Tensorflow: as Keras backend for the deep learning implementation (47)

Chapter 5

Results

In this chapter we evaluate the path prediction system; first we describe how our system can emulate OSPF by analyzing the results of the model training; finally we discuss the performance of the path prediction model as a substitute to more traditional routing algorithms. For a more complete analysis, we also implement the same Deep Neural Network (DNN) described in (44), a traditional neural network with four hidden layers and sixteen neurons in each layer. We use this network to compare the performance between DNNs and LSTM for the same task and understand if our hypothesis about RNNs is correct.

5.1 Learning from OSPF

Our system is built to learn the behavior of OSPF across different configurations, and correlate it with different traffic patterns. We use a LSTM RNN as a learning algorithm and build a model for each source-destination pair in our topology (Figure 4.3): the total number of models is given by all the possible source-destination pairs, with the destination addresses considered only on the outer routers not considering the source; for any given a router, the number of addresses associated to it is equal to the number of its interfaces. In our considered scenario, this resulted into a set of 162 distinct models that are used to determine the hop-by-hop path from a source router to a specific destination address. The path is computed iteratively as follows: starting from the source, the model for the selected destination is used to predict the next hop, then the predicted next hop becomes the new source router; the process is then repeated until the predicted hop is the final destination. Given their significant size, it is unfeasible to analyze all models individually; we are interested in exploring the overall performance. To do so, we analyze the average accuracy and loss over all different models, as discussed in the previous chapter.

Figure 5.1 shows the model training progress over time in terms of accuracy and loss: the plot shows the average of the metrics over all 162 models. The slopes of the graphs give us an idea of what is happening during the training phase: at the beginning (epochs 0-20), both slopes are very steep, indicating that the model is abandoning the initial randomness and converging towards a final stable solution. Afterwards, from epoch 20 to 60, as the gradient diminishes, the slope starts to decrease slowly, indicating that the gradient has probably entered the region of the space in which it will converge to the problem solution. Finally, the curve becomes almost flat, showing that the gradient has reached its minimum. Note how in both graphs, the two curves have the same behavior: this shows that the model is learning "without losing generalities". An increasing accuracy on the training set with a steady or decaying validation accuracy would be a clear sign of overfitting, a situation in which the model becomes too specialized on the training data and it is not able to properly classify new samples. The figure shows better performance for both accuracy and loss on the validation data rather than on the train data; even if generally unusual, the reasons of this behavior can be found in the dropout regularization technique. At training time, because of dropout, only part of the network is used; on the other hand, when testing the development of the model on the validation set, regularization mechanisms (i.e., dropout) are turned off, so the network is used in its completeness. This means that the whole network is used to measure accuracy and loss on the validation set but only a part of it is used for the same metrics at training time: for this reason, the performance on the validation set are slightly better than on the training set.

To understand how well our model can emulate OSPF we need to analyze the performance on the test set; as we did for the training, we evaluate the performance of the system by averaging the results of the single models. The system achieves an average accuracy of **98.71%**, with a loss of only **0.0496**; showing really promising results. With an accuracy of almost 99%, LSTM-RNNs performs better than traditional DNNs (44), which achieves around 90% of accuracy; however this comparison should be taken with caution, considered that the topologies in the two experiments are slightly different and experiments reproducibility is an open issue in machine learning (48). The comparison of these two approaches is shown in Figure 5.2.

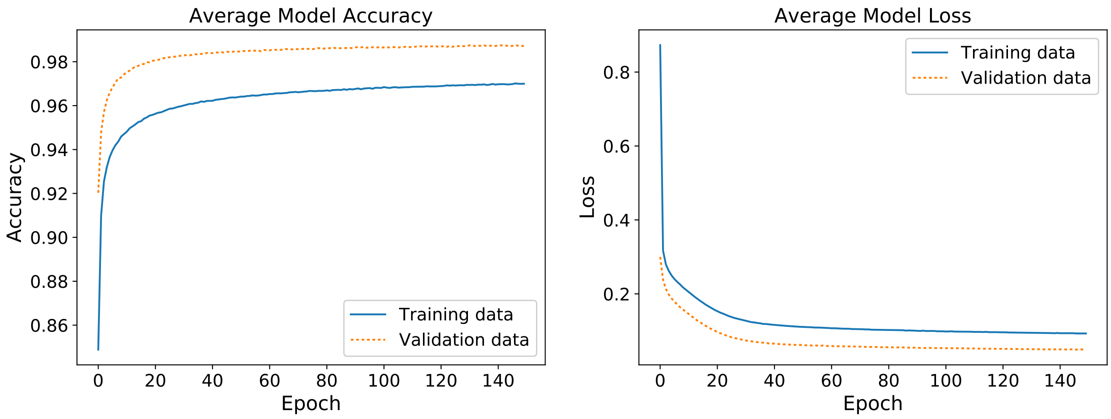

Figure 5.1. Training accuracy and loss progress on training and validation data.

It is important to notice that in this case, the average accuracy could be slightly misleading: the models that represent our model — a source-destination pair connected by a single link — are oversimplified, hence it is very likely for them to have an accuracy close to 100%. Nonetheless, only 32% (obtained counting all directly connected targets including all the interfaces on the destination router) of the models represents this situation, that means that even if they all had 100% accuracy, the remaining models would have an average accuracy of 97.89%.

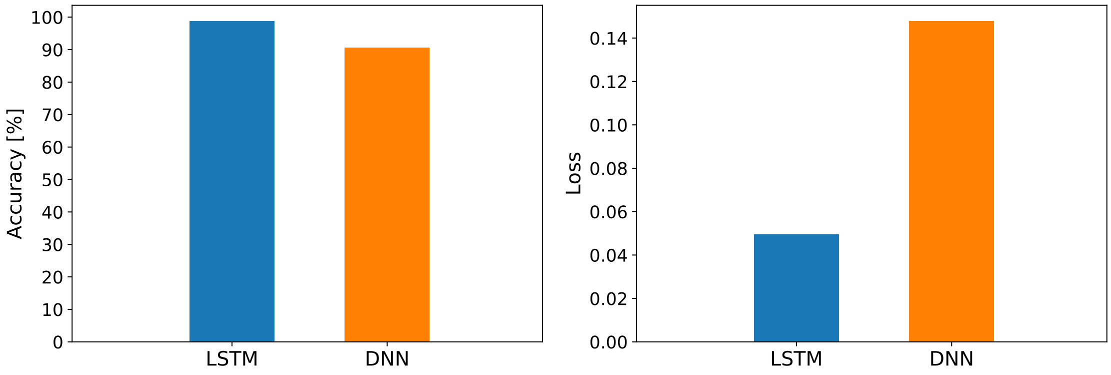

Figure 5.2. LSTM and DNN performance comparison.

5.2 Overwriting OSPF Routing

To evaluate our model as OSPF substitution, we observe the behavior of the path prediction system in a functioning network. In particular, we use the same topology (Figure 4.3) and traffic simulator adopted in the dataset generation phase; to ease the analysis process, all links are set to the same rate. Afterwards, we select a source router and a destination address and examine the difference in behavior between OSPF and our system.

In general, our system shows a dynamic behavior, predicting several paths for the same destination in different traffic conditions. More specifically, we run four traffic simulations, each of them for fifteen minutes, with a varying loss rate on the link chosen by OSPF to connect source and destination; at the same time, the path prediction system computes a new path every five seconds. The selected target is $(R1, R3)$, with the default path being $R1, R2, R3$ and the loss being varied on the link between R1 and R2. Figure 5.3 compares the routing decisions made by the system in comparisons: being performance unaware, OSPF always chooses the same path, even when the link has losses. Our system on the contrary, shows the ability of behaving dynamically by proposing four alternative paths.

By studying the system behavior in presence of losses, it is possible to understand if our model is able to detect and overcome these problems. We test loss rates of 0%, 5%,10%, 15% and count the number of predictions different from OSPF (table 5.1). With the loss set to 0%, 43% of the time the predicted path is different from OSPF; if the loss is increased to 5%, the ratio of path different from OSPF slightly rises to 45%, indicating that the system is able to detect the change. The same happens for a loss of 10%, with a much more noticeable improvement in the system behavior; 63.5% of the proposed paths are in fact, different from the one chosen by OSPF. For the successive loss rate, equals to 15%, the performance goes down a little with only a 59.5% different path ratio; the reasons for this loss in performance are discussed in chapter 6. The ideal behavior would be for the system to detect the link loss and consequently stop predicting paths going through the damaged link. In our analysis this happens only with a limited loss rate.

Table 5.2 compares the resulting retransmission rate of our system, OSPF and Equal Cost Multi Path (ECMP) routing algorithm. The retransmission rate is computed by taking into account how many times traffic would pass through the leaky link, considering two equal-cost paths for ECMP and the ratios in table 5.2

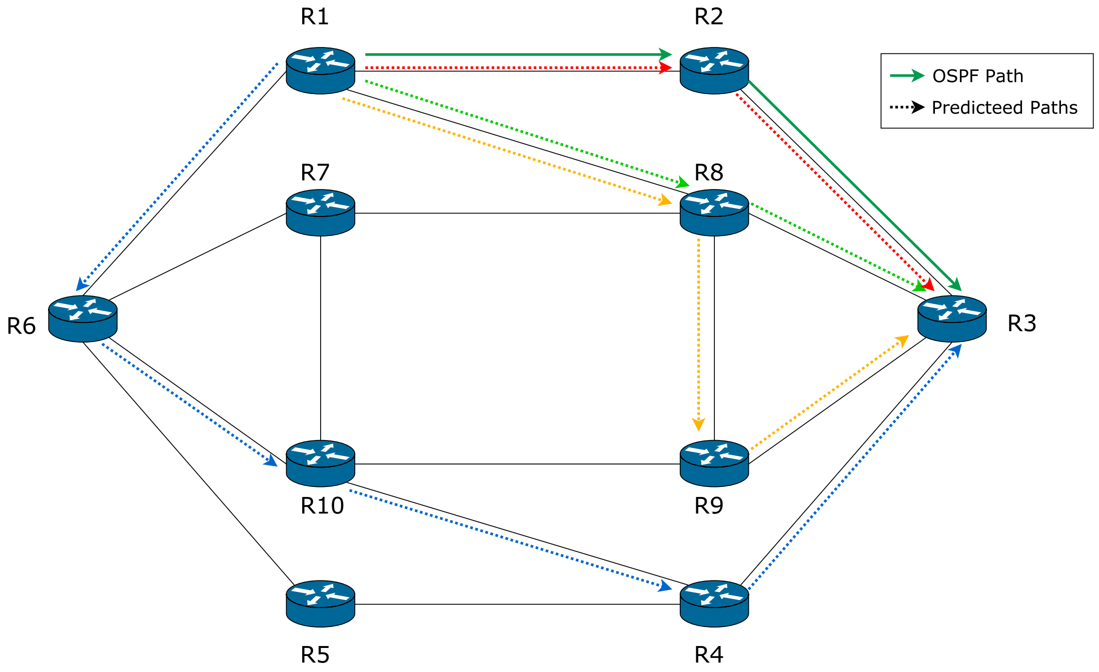

Figure 5.3. Routing policies comparison.

for our system. Overall, the system we propose, has a lower retransmission rate than the other strategies, reaching therefore a higher throughput. Figure 5.4 is a graphical comparison of the three strategies, showing the time difference needed to transmit the same amount of data. If there is no loss, the three approaches behave the same, however, as soon as a loss rate is introduced, the gap between the curves increases. This increases with the loss rate; however, while it is evident with respect to OSPF, the variation between ECMP and our system is less pronounced.

Link loss	Different path rate	Same OSPF path rate
0%	43%	57%
5%	45%	55%
10%	63.5%	36.5%
15%	59.5%	40.5%

Table 5.1. Path predictions different and equal to OSPF.

There is another consideration that needs to be made when talking about OSPF. It would be unfair to consider this protocol completely performance

	Routing Strategy			
Link loss rate	**OSPF**	**ECMP**	**DNN**	**LSTM**
0%	0%	0%	0%	**0%**
5%	5%	**2.50%**	2.70%	2.75%
10%	10%	5%	7.70%	**3.65%**
15%	15%	7.50%	9%	**6.07%**

Table 5.2. Routing strategies retransmission rate comparison.

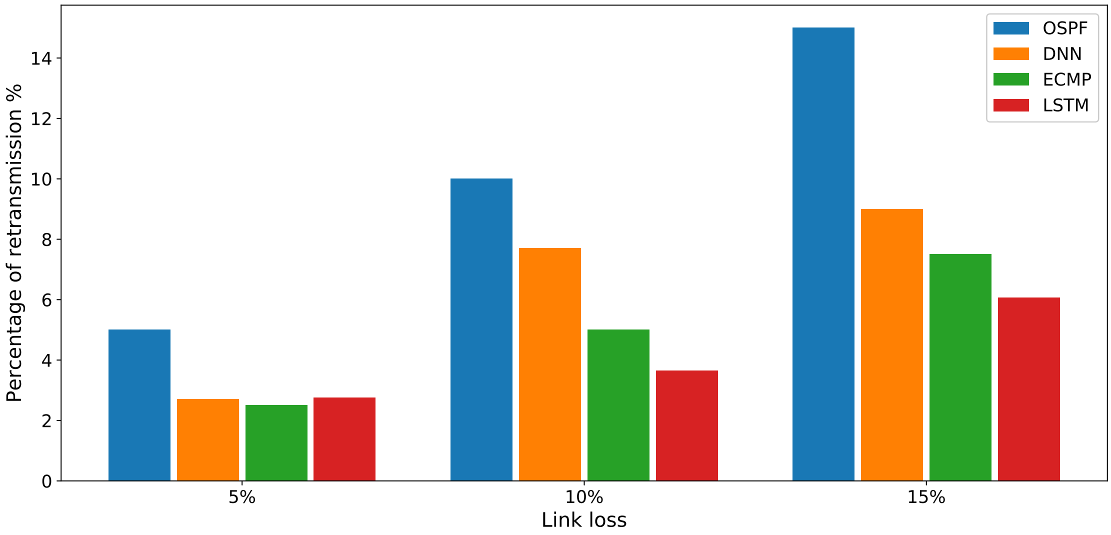

Figure 5.4. Routing policies retransmission comparison.

unaware; during our tests, we have noticed that starting from a certain point, OSPF is able to detect the problem. In our case, with a link loss greater than or equal to 20%, OSPF changes its output, selecting a new shortest path. This behavior is caused by the fact that when the loss on the link is too high, the HELLO (i.e., keep-alive) packets used by the protocol are lost, causing the link discovery part of the algorithm to deviate to other routes. We believe that this is a Mininet's limitation and its way to interpret the loss as a fixed phenomenon happening on the link. We know that in real networks losses don't work this way, therefore we are limited in the testing possibilities; nonetheless these results look promising.

5.3 A critical scenario

To have a more clear idea about our system capabilities, we test it in a critical scenario. We decide to simulate a network in which statistically, half of the links are affected by a loss rate; we test the same loss rates of the previous experiments (5%, 10%, 15%), ten times each, generating traffic between five different targets. The purpose of this experiment is to understand if our system can tolerate better than OSPF a condition where half of the network is not functioning properly. To compare the performance of the two approaches we counted the number of times the links with loss were used; the results are shown in Figure 5.5. The chart compares the total number of defective links traversed in all the runs for each link loss rate. From the results, it appears that our proposed system does not introduce any significant advantage under critical circumstances; the chart shows in fact, that overall, the performance of the two systems are similar, with OSPF performing better when the link loss rate is set to 10% and 15%. The reasons of these poor performance are to be found in how our system works: we trained our model to predict alternative paths based on the network conditions; even in this case, the proposed system is able to suggest alternative paths. The main problem is that, given that half of the link in the networks is affected by a loss, the majority of the proposed alternative paths passes through these links, resulting in poor performance. In the conclusions (chapter 6) we give a few hints on how to overcome such limitations of our system.

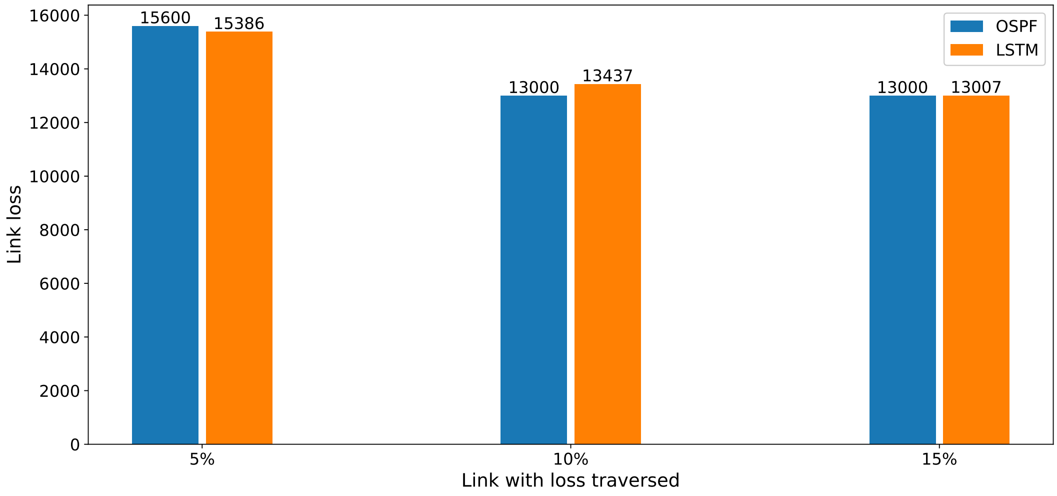

Figure 5.5. Comparison of the number of (severely) lossy links traversed by OSPF and LSTM.

Chapter 6

Conclusions and Research Directions

In this work, we presented an architecture designed in support of mobile devices task and traffic offloading. Our architecture's aim has been to provide guidelines for task offloading protocol implementations; we prototyped our architecture within a local virtual network testbed based on Mininet.

We focus our attention on a specific traffic offloading policy that leveraged deep learning to predict the best path towards a destination, increasing throughput and reducing perceived latency. Our results showed that despite some limitations, machine learning could be a valid alternative to traditional routing algorithms and can be leveraged to improve network performance. Our results also showed that cooperative routing can steer traffic with better performance than traditional methods, suggesting that applying machine learning in this context is an area worthy of further exploration.

Our work has some limitations that are not impossible to overcome: we noticed a prediction performance degradation with the increasing loss rate; we believe this is most likely due to our limited training dataset. We also disclose that the quality of our dataset is limited by constraints in both Mininet and the available computation capability. Another problem that needs to be addressed is the scalability of this approach: training numerous deep learning models requires extensive computational power; being able to reduce the number of models to train would save time and make this system more scalable.

As a future research direction, we suggest analyzing our method with a broader dataset and in different scenarios. Additionally, one could perform a deeper performance analysis by deploying a network that completely replaces OSPF with our system and collects measurements such as throughput and latency. Finally, one could explore more recent machine learning techniques, especially, reinforcement learning, which appear to be promising in solving decision problems.

Appendix A

Protocol messages implementation

messages.proto

```
syntax = "proto3";
message OffloadRequest {
    message Requirements {
        enum Latency {
            URGENT = 0;
            STANDARD = 1;
            LOOSE = 2;
        }

        float cpu = 1;
        int32 memory = 2;
        Latency latency = 3;
    }

    enum Type {
        LAMBDA = 0;
        STANDARD = 1;
    }

    message Task {
        message TaskWrapper {
            enum WrapperType {
                JAR = 0;
                EGG = 1;
            }

            string name = 1;
            WrapperType type = 2;
            bytes task = 3;
        }

        oneof task_location {
            string task_id = 1;
            TaskWrapper wrapper = 2;
        }
    }

    Requirements requirements = 1;
    Type type = 2;
    Task task = 3;

}

message Response{
    enum Result {
        OK = 0;
        INVALID_MSG_SIZE = 1;
        INVALID_REQUEST = 2;
    }

    Result result = 1;
    string msg = 2;
}

message Message{
    enum Type {
        OFFLOAD_REQUEST = 0;
        RESPONSE = 1;
        TASK = 2;
    }

    Type type = 1;
    oneof msg_type {
        OffloadRequest off_req = 2;
        OffloadRequest.Task task = 3;
        Response response = 4;
    }
}
```

Appendix B

Configure mininet to run the Quagga routing suite

In the course of these six months of work, I have been slowed down by numerous issues that came up as my reasearch was proceeding; these issues were due to my inexperience with the software and its complexity. To implement the deep learning model, I needed a functional network with running routing algorithms. At first the solution looked simple, I just had to run the Quagga routing suite on the mininet nodes. Accomplishing this result took me away a lot of time, so I have decided to highlight here the problems I have faced. Here's the list:

- by default, mininet doesn't support loops in a topology: if you want to have a complex closed topology you need to manually enable the spanning tree protcol on every switch

- when you build Quagga from source, the link to the dynamic libraries are not automatically created, to solve the problem run:

  ```
  sudo ldconfig
  ```

- the Quagga service config in mininext relies on init.d scripts which are not installed when you build it from source, the quickest solution is to install quagga from the distribution repositories

- in mininet, the default ovs-controller doesn't support more than sixteen switches; if you need a bigger network you need to install the mininet patched version of the openflow controller (<https://github.com/mininet/mininet/wiki/FAQ#ovs-controller>)

- in miniNext, it is not possible to capture traffic directly on the hosts (in my case quagga routers) because they're in a different namespace. The workaround is to set up a switch on every link (between each pair of routers) and analyze the traffic on its interfaces; for the problem described in the previous point, the number of switch limits the network size

- ryu is not able to talk with the mininet switches that do not have the canonical name (s1, s2, ...)

- the output of OSPF depends on the nominal speed interface declared in the Zebra configuration file, changing the link speed through mininet doesn't affect the algorithm

Bibliography

[1]SHUFF, P. Building a billion user load balancer. In: . Dublin: USENIX Association, 2015.

[2]KARPATHY, A. *Software 2.0.* <https://medium.com/@karpathy/software-2-0-a64152b37c35>.

[3]JIANG, J. et al. Unleashing the potential of data-driven networking. In: SPRINGER. *International Conference on Communication Systems and Networks*. [S.l.], 2017. p. 110–126.

[4]JIANG, J. et al. Eona: Experience-oriented network architecture. In: ACM. *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. [S.l.], 2014. p. 11.

[5]NGUYEN, T. T.; ARMITAGE, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, IEEE, v. 10, n. 4, p. 56–76, 2008.

[6]BUI, V. et al. Long horizon end-to-end delay forecasts: A multi-step-ahead hybrid approach. In: *2007 12th IEEE Symposium on Computers and Communications*. [S.l.: s.n.], 2007. p. 825–832. ISSN 1530-1346.

[7]MAO, H.; NETRAVALI, R.; ALIZADEH, M. Neural adaptive video streaming with pensieve. In: ACM. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. [S.l.], 2017. p. 197–210.

[8]HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural computation*, MIT Press, v. 9, n. 8, p. 1735–1780, 1997.

[9]LEWIS, G. A.; LAGO, P. A catalog of architectural tactics for cyber-foraging. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. New York, NY, USA: ACM, 2015. (QoSA '15), p. 53–62. ISBN 978-1-4503-3470-9. Disponível em: <http://doi.acm.org/10.1145/2737182.2737188>.

[10]WANG, J. et al. Under submission: Edge cloud offloading algorithms: Issues, methods and perspectives. In: . [S.l.: s.n.], 2017.

[11]EOM, H. et al. Malmos: Machine learning-based mobile offloading scheduler with online training. In: *2015 3rd IEEE International Conference*

on Mobile Cloud Computing, Services, and Engineering. [S.l.: s.n.], 2015. p. 51–60.

[12]MAO, H. et al. Resource management with deep reinforcement learning. In: ACM. *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. [S.l.], 2016. p. 50–56.

[13]CRUTCHER, A. et al. Hyperprofile-based computation offloading for mobile edge networks. *arXiv preprint arXiv:1707.09422*, 2017.

[14]KATO, N. et al. The deep learning vision for heterogeneous network traffic control: Proposal, challenges, and future perspective. *IEEE Wireless Communications*, v. 24, n. 3, p. 146–153, 2017. ISSN 1536-1284.

[15]VALADARSKY, A. et al. Learning to route. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2017. (HotNets-XVI), p. 185–191. ISBN 978-1-4503-5569-8. Disponível em: <http://doi.acm.org/10.1145/3152434.3152441>.

[16]KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015.

[17]MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

[18]KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219.

[19]MESTRES, A. et al. Knowledge-defined networking. *ACM SIGCOMM Computer Communication Review*, ACM, v. 47, n. 3, p. 2–10, 2017.

[20]KIM, C. et al. In-band network telemetry via programmable dataplanes. In: *ACM SIGCOMM*. [S.l.: s.n.], 2015.

[21]CLEMM, A.; CHANDRAMOULI, M.; KRISHNAMURTHY, S. Dna: An sdn framework for distributed network analytics. In: IEEE. *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. [S.l.], 2015. p. 9–17.

[22]CLARK, D. D. et al. 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.

[23]DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*, Springer, v. 1, n. 1, p. 269–271, 1959.

[24]SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, v. 3, n. 3, p. 210–229, July 1959. ISSN 0018-8646.

[25]SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning I: Introduction*.

1998.

[26]GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>.

[27]IVAKHNENKO, A.; LAPA, V. *Cybernetic Predicting Devices*. CCM Information Corporation, 1973. (Jprs report). Disponível em: <https://books.google.com/books?id=FhwVNQAACAAJ>.

[28]MASTERING the Game of Go with Deep Neural Networks and Tree Search. *Nature*, v. 529, n. 7587, p. 484–489, jan 2016. ISSN 0028-0836.

[29]GREFF, K. et al. LSTM: A search space odyssey. *CoRR*, v. 1503, 2015.

[30]BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, IEEE, v. 5, n. 2, p. 157–166, 1994.

[31]SHARIFI, M.; KAFAIE, S.; KASHEFI, O. A survey and taxonomy of cyber foraging of mobile devices. *IEEE Communications Surveys Tutorials*, v. 14, n. 4, p. 1232–1243, Fourth 2012. ISSN 1553-877X.

[32]GOOGLE Protocol Buffer. <https://developers.google.com/protocol-buffers/>.

[33]HENDRICKSON, S. et al. Serverless computation with openlambda. *Elastic*, v. 60, p. 80, 2016.

[34]FOX, G. C. et al. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[35]Openflow. <https://www.opennetworking.org/>.

[36]CENTER for Applied Internet Data Analysis. <http://www.caida.org/home/>.

[37]ROY, A. et al. Inside the social network's (datacenter) network. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 123–137. ISBN 978-1-4503-3542-3. Disponível em: <http://doi.acm.org/10.1145/2785956.2787472>.

[38]THE Internet Topology Zoo. <http://www.topology-zoo.org/>.

[39]LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: ACM. *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. [S.l.], 2010. p. 19. <http://mininet.org/>.

[40]MinineXt. <http://mininext.uscnsl.net/>.

[41]QUAGGA. <http://www.nongnu.org/quagga/>.

[42]RYU. <https://osrg.github.io/ryu/>.

[43]IPERF. <https://iperf.fr/>.

[44]KATO, N. et al. The deep learning vision for heterogeneous network traffic control: Proposal, challenges, and future perspective. *IEEE Wireless*

Communications, v. 24, n. 3, p. 146–153, 2017. ISSN 1536-1284.
[45]SRIVASTAVA, N. et al. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, JMLR. org, v. 15, n. 1, p. 1929–1958, 2014.
[46]KERAS. <https://keras.io/>.
[47]TENSORFLOW. <https://www.tensorflow.org/>.
[48]OLORISADE, B. K.; BRERETON, P.; ANDRAS, P. Reproducibility in machine learning-based studies: An example of text mining. 2017.