POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

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

Relatore
prof. Flavio Esposito

Correlatore:
prof. Guido Marchetto

Alessandro Gaballo
matricola: 231587

Anno accademico 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 computation 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 chose 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 [**?**].

To forecast path metrics, we use machine learning techniques, which in the last few years have affected the way we make software [**?**]. 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

List of Figures

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, which contributes to the very famous world of big data. The majority of these devices do not have or can not handle the computational requirements to process the data they capture and so leave to the cloud the responsibility to perform the 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. In critical scenarios, such as natural disasters, not only computation offloading becomes necessary, but latency requirements become more strict, making paths management essential to satisfy these requisites. Current systems for traffic offloading are usually performance unaware (e.g. OSPF, ECMP), achieving therefore sub-optimal performance. Many people have proposed the use of machine learning techniques to solve network problems, including traffic classification [**?**] and latency prediction [**?**]; however, no one has ever used LSTM to train network devices to steer traffic in a virtual network. 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. The goal is to outperform the classic routing policies in terms of latency and throughput by learning implicit patterns in network traces.

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

- the design of an edge computing architecture for **offloading** management in edge computing, that describes the macro blocks required for such a system and proposes an offloading protocol
- 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

- performance evaluation extended to different use cases: we evaluate the prediction system first for its ability to learn from an existing model, second as a routing algorithm replacement.

This report is organized as follows:

Chapter 2 contains a brief background about machine learning and networking notions and main techniques used in this project.

Chapter 3 describes the personal contribution of this project, including the details about the architecture and the implementation of the path predictor system.

Chapter 4 illustrates a summary of the related work.

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

Background

Before describing our approach, we would like to give a brief introduction to some of the methods and concepts that are needed to understand our implementation and related 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. Even though these might seem separate topics, keep in mind that in this project, we take advantage of machine learning to address networking problems.

2.1 Software-Defined Networking

Software-Defined Networking (SDN) is paradigm that promises to break networks' vertical integration of control and data plan, 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 [**?**].SDN is a network architecture built on four principles:

1. separate control and data planes: network devices are left responsible only of packet forwarding
2. flow-based forwarding: packets are forwarded by looking to a set of fields instead of only the destination, introducing great flexibility [**?**]
3. external control logic: the control logic is moved to an external entity called SDN controller or Network Operating System, 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.

2.2 Knowledge-Defined Networking

Knowledge-defined networking (KDN) [**?**] 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 centralize 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 [**?**]. Information provided by network telemetry are usually provided to a centralized Network Analytics (NA) platform [**?**], that combined with SDN can bring to light the Knowledge Plane proposed in [**?**]. Knowledge-Defined Networking is the paradigm resulting from the combination of these tools, specifically Software Defined Networking, Network Analytics and Knowledge Plan. 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, NA and ML to provide automated network control. 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.

OSPF is a link-state routing algorithm for IP networks. OSPF falls in the family of iBGP and it is widely adopted in large networks. 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. The OSPF routing policies are dictated by link metrics associated with each routing interface, typically the interface speed.

Machine learning is a field of computer science that studies the ability of making computers learn without explicitly programming them [**?**]. In machine learning there are three main approaches: supervised learning, unsupervised learning and

reinforcement learning. Supervised learning is used to classify labeled data, the label is a sort of supervisor describing the class of an observation. In unsupervised learning on the other hand there is no supervisor, meaning that data are unlabeled and the goal is to find the hidden relation of the data. Reinforcement learning is learning what actions to do 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 [**?**]. Among the several branches of machine learning, neural networks and in particular deep learning, have recently attracted a lot of attentions.

Teaching a computer to solve tasks that are hard to describe formally (e.g. speech recognition) 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, with many layers. For this reason, we call this approach to AI deep learning [**?**]. In other words, 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 [**?**]. The first working deep learning algorithm was a multilayer perceptrons developed by Ivakhnenko et al. [**?**]. Later on new techniques including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) were developed; these techniques dramatically increased AI performance in tasks such as image recognition. More recently deep learning has been used to build AlphaGo [**?**], a program that plays the board game Go and managed to beat, in 2017, the Go world No.1 Ke Jie.

2.2.1 LSTM

Long Short-Term Memory cells are a further development of Recurrent Neural Networks (RNNs). They were developed by Hochreiter and Schmidhuber in 1997 and have been further improved ever since [**?**]. The major flaw of a traditional neural network is that it does not capture the relation between the input it currently looks at and the previous training example. This however, is crucial for e.g. developing a language model. Making predictions about a subsequent word strongly depends on the preceding words. RNNs try to solve this problem; yet it turns out that they are not scalable to model long-term dependencies in practice. This is due to numerical problems commonly referred to as the *vanishing/exploding gradient* when weight updates are backpropagated through the time steps. LSTMs overcome this problem and enable capturing long-term temporal dependencies among the input elements. They 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 2.1 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 feedforward) 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 based on the training of the network. 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 *Backpropagation 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.

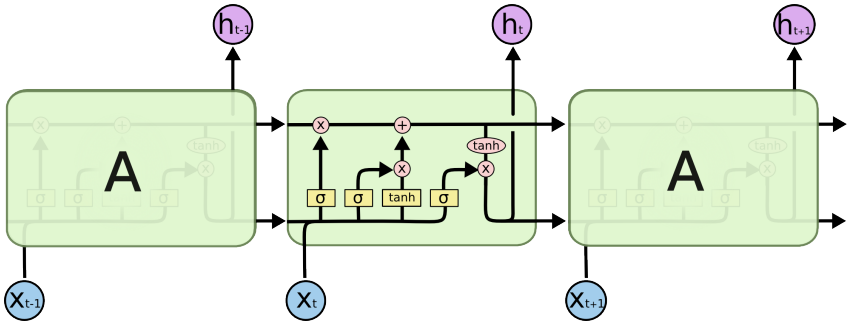

Figure 2.1. LSTM cell overview.[1]

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

Chapter 3

Offloading Architecture

This chapter is organized as follows: section 3.1 illustrates the task offloading architecture and its implementation, section 3.2 describes the path prediction system and its details and finally, section 3.3, summarizes the tools adopted in the development of the project.

3.1 Task Offloading

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. To the best of our knowledge, there has not been any effort in trying to implement a general abstraction that provides the guidelines for the protocol implementations. The proposed system is a first draft of how such abstraction could be implemented, with all the limitations of a six-months work.

3.1.1 Architecture

The system architecture is described in figure 3.1; we consider a scenario in which mobile devices want to offload tasks to the edge cloud in a modern network that supports SDN.

The main components are:

- mobile device interface: interface for communications between the mobile devices and the offloading system
- edge cloud interface: interface for communications between the edge cloud and the offloading system
- offloading logic: offloading policy to be used to serve the edge client

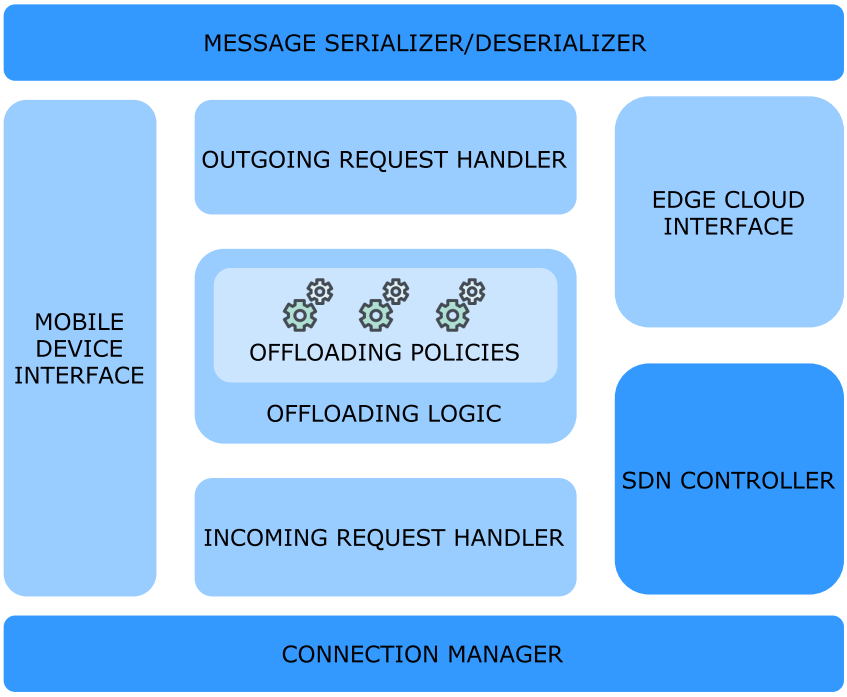

Figure 3.1. Offloading system architecture.

- 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 system, it provides 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.

3.1.2 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 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.

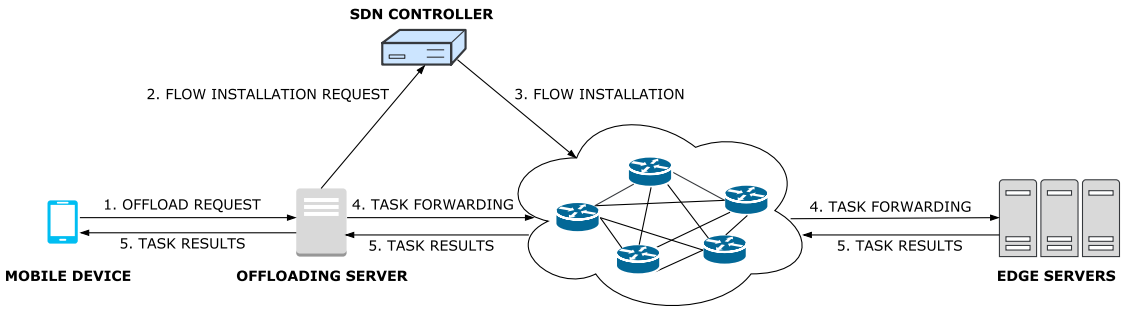

Figure 3.2. Offloading workflow: mobile devices and an offloading server orchestrate the request via an SDN controller

Figure 3.2 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 it is possible or not to accept the request and if so, where to offload it according to the specified or default policy
3. the offloading server asks the SDN controller to install on the switches the flows required by the offloading policy
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 forwards it to the device.

Task offloading is a complex problem, with a high number of factors involved in the final decision. These factors include application requirements formalization, task retrieval, edge server/cloud discovery. Before proceeding with the protocol implementation, we address some of these problems.

Application requirements formalization

As a starting point 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 JAR, EGG).

Edge servers discovery

Since 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.

With these simplifications in mind, we carry out the protocol messages definition using Google Protocol Buffer [**?**], 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.

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 server-less computing [**?**] [**?**] and implies two possible scenarios. In case of server-less 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.

3.2 Path prediction via Deep Learning

The architecture described in the previous section 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 problem of traditional routing algorithms is that they do not take into account how the network load changes over time: 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. The intuition behind our project is that *collaborative traffic steering* should be able to identify and avoid congestion situations. A collaborative policy requires in some way the participation to all the parts in the decision process, however, achieving nodes collaboration in a network is a complicated task. A simplified version of this collaborative process could be the following: instead of all the nodes to be directly involved in the decision process, they could limit their role to notify 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. 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 algorithms: 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 we were able to see all the possible outcome of a routing protocol in a network and extract the consequent traffic patterns, we 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.

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 use.

3.2.1 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 given that the model does not have enough information to learn; on the other hand, even a big dataset which covers only 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.

Recently a lot of effort is being put in making datasets available to the machine learning community, with the aim of promoting research. Regarding networks, there is plenty of dataset available [**?**][**?**][**?**]; however those are mostly network captures of datacenters or small portions of networks, without any detail about the underlying topology nor the routing strategies. To implement our path prediction system, given a network, we need:

- the network topology
- the routing informations
- the packet count on each node.

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 [**?**] is a software commonly used in research, that can create a virtual network running real kernel, switch and application code with support for OpenFlow and Software-Defined Networking[**?**].

Figure 3.3 shows the topology used in our system: it is composed of ten routers, R1-R6 which I am going to refer to as *outer routers*, and R7-R10 which I am going to call *inner routers*. The topology includes also fourteen switches, the reasons of their presence is explained in section 3.2.1. 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, it will learn several different ways to go from a source A to a destination B.

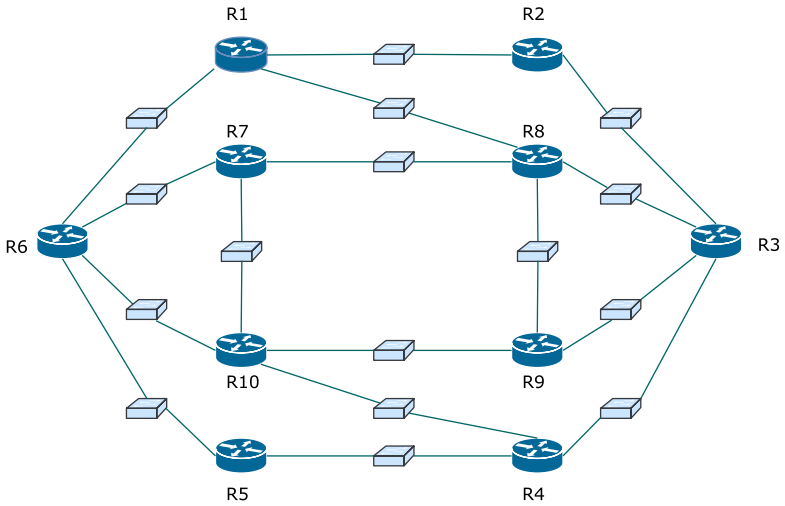

Figure 3.3. Model topology

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 [**?**], a mininet extension layer that support routing engines and PID namespaces. As a routing engine we use Quagga [**?**], 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 3.4 shows an outline of this architecture: each Mininet host runs in a separate namespace and lunch the zebra and the routing daemons; the zebra daemons updates each router routing table with the routing informations exchanged through the virtual network

built with the Linux Kernel.

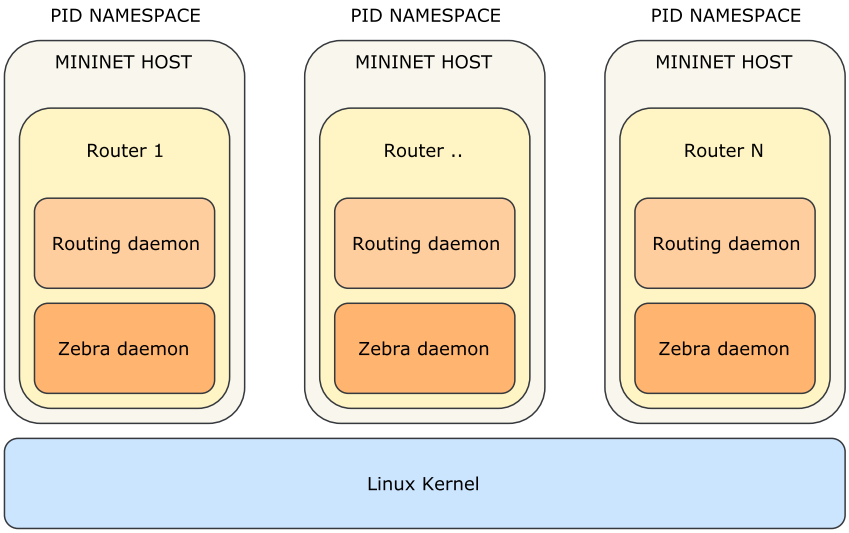

Figure 3.4. Quagga router implementation in MiniNext

Router packet counter

The last elements necessary before putting everything together are the statistics about the number of packets on each router. In section 3.2.1 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.

A SDN controller can retrieve switches information with the OpenFlow protocol; in this project we use Ryu [**?**], a 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 3.5 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.

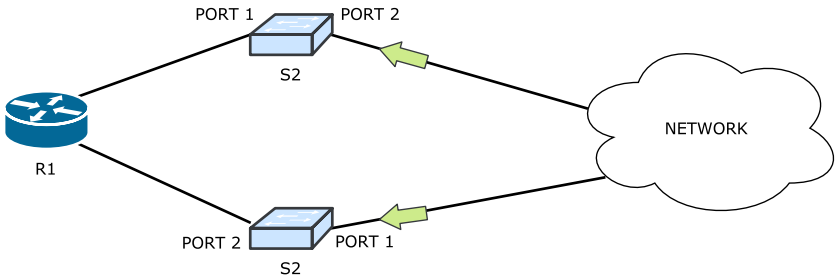

Figure 3.5. Router packet counter explanation

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 3.2.2; 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. 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* [**?**], 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 the path cost according to the interface speed, therefore it is sufficient to change the speed of the interfaces on the routers to make OSPF compute the new routes.

To summarize, we generate the dataset as follows: for *ospf_configurations* times, every *ospf_conf_duration* seconds, the network is teared down and rebuilt with new link speed, producing a new OSPF configuration; for each OSPF configuration, traffic is generated in the network by running *iperf* 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 3.6 while table 3.1 describes the algorithm parameters.

Figure 3.6. 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

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 3.1. Dataset generation parameters

Using the parameters of table 3.1, we obtain a dataset of 17696 samples; the 85% of these samples forms the training set, the remaining 15% is test set.

3.2.2 Deep Learning Model

The deep learning model chosen for this project is a LSTM recurrent neural network (RNN): recurrent neural network is a class of neural network in which connections between nodes form a directed cycle (3.7). 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 want our model to learn the correlation between changes in the packet distribution and routing decisions over time.

Up to this moment we have talked about a model as if a single model would be able to predict every path for every possible source-destination pair; however, given the complexity of routing, this scenario does not seem feasible. The 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 in our case, more than one network interface; in this scenario there will be more model than the previous case, each one describing a particular $(src\_router, dst\_address)$ pair.

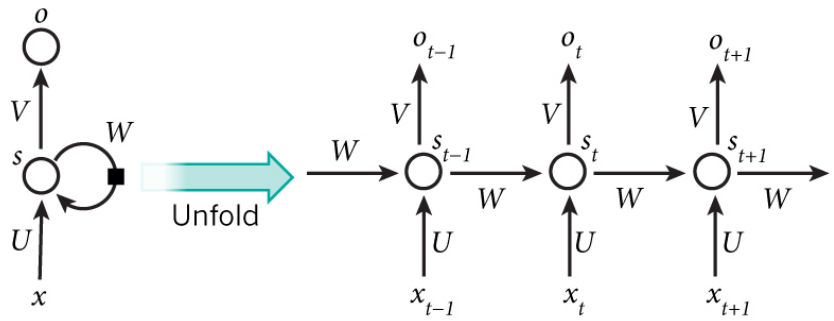

Figure 3.7. Recurrent neural network and its unfolded version.[1]

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 [**?**]: 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 3.2.1: even though these router do not generate traffic, they are still responsible of forwarding packets coming from other sources, making therefore necessary to model this behavior.

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 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.

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

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 steps:

- pick a source-destination pair that requires a complex model
- cross-validate each combination of layer and neurons
- choose the combination with the best combination of accuracy and loss.

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 of these considerations, we choose the target $(R1, R4)$; figure 3.8 illustrates some of the 3-hops 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.

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 3.2. 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

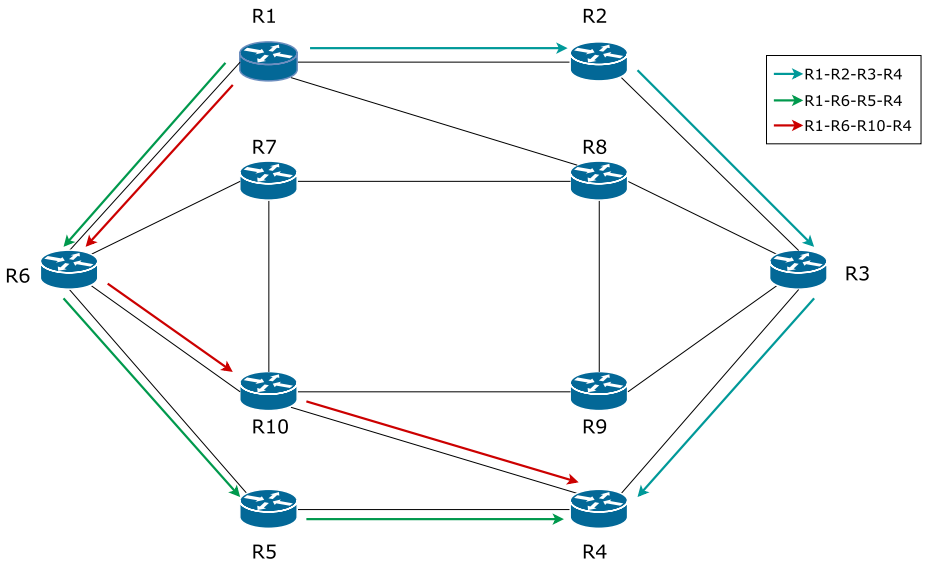

Figure 3.8. Examples of paths connecting the validation target.

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 for 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 3.2. LSTM architectures comparison

From this first analysis it is clear that to achieve the best results, each layer of the network needs to have 128 processing units; however deciding on the number of layers is not that immediate. As previously discussed, the gain in performance with an increase of the number of layers is not noticeable, therefore we decide to take into account further factors to choose the final architecture. Table 3.3 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.

Considering the limited computational power and time, and the number of models we need to train, we decide to use a network with two layers, as a good compromise 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 3.3. LSTM architectures average training time

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

- input layer (10 neurons)
- two hidden layers (128 neurons ea., *tanh* activation)
- output layer (10 neurons, sigmoid activation)

To increase training efficiency and avoid overfitting, we apply a batch normalization layer on the input, dropout on both the recurrent and feedforward connections, and $L2$ regularization to the loss function. Figure 3.9 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%.

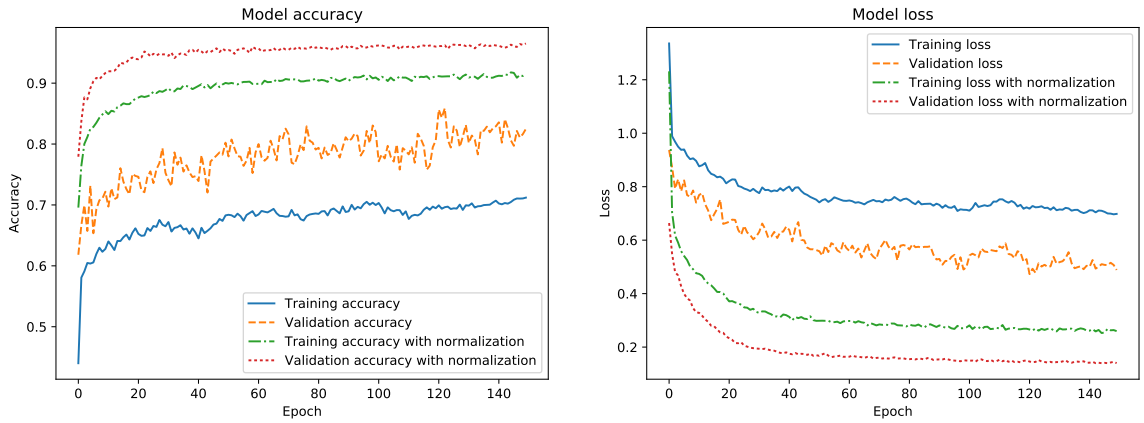

Figure 3.9. Impact of input normalization on training.

3.3 Implementation

To prototype our system we use the following tools:

- Ryu: SDN controller written in Python [**?**]
- Google Protocol Buffer: framework for serializing structured data [**?**]
- Mininet: virtual network creation tool [**?**]
- Quagga: routing software suite [**?**]
- Keras: high-level neural networks API written in Python [**?**]
- Tensorflow: as Keras backend for the deep learning implementation [**?**]
- Python: as a main programming language.

Chapter 4

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 [**?**] 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 [**?**] 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 [**?**] 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.

In [**?**] machine learning is used for computation offloading in mobile edge networks. 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 [**?**] 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 [**?**]. 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.

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 the traditional routing algorithms.

5.1 Learning from OSPF

The system is built to learn OSPF behavior in different configurations and correlate it with the traffic patterns. We use a LSTM RNN as a learning algorithm and build a model for each source-destination pair in our topology (figure 3.3): the total number of models is given by the all the possible (src_router, dst_addr) pairs, with the destination addresses considered only on the outer routers different from the source; given a router, the number of addresses associated to it, is equal to the number of its interfaces. The result is a set of 162 models which are used to determine the hop-by-hop path from a source router to a specific destination address. The whole path is computed 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 and the process is repeated until the predicted hop is the final destination. Given their high number, it is not feasible to analyze all models individually; moreover, we are interested in exploring the overall performance. To do so, we analyze the average accuracy and loss over the 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 is the average of the metrics over the 162 models. The slopes of the graphs give us an idea of what is happening during the training: at the beginning (epochs 0-20), both of the slopes are very steep, indicating that the model is abandoning the initial randomness and moving towards the 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 the problem minimum and cannot improve anymore.It is very important to notice how in both the graphs, the two curves have the same tendency: this is crucial because it shows that the model is learning without losing generality. 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.

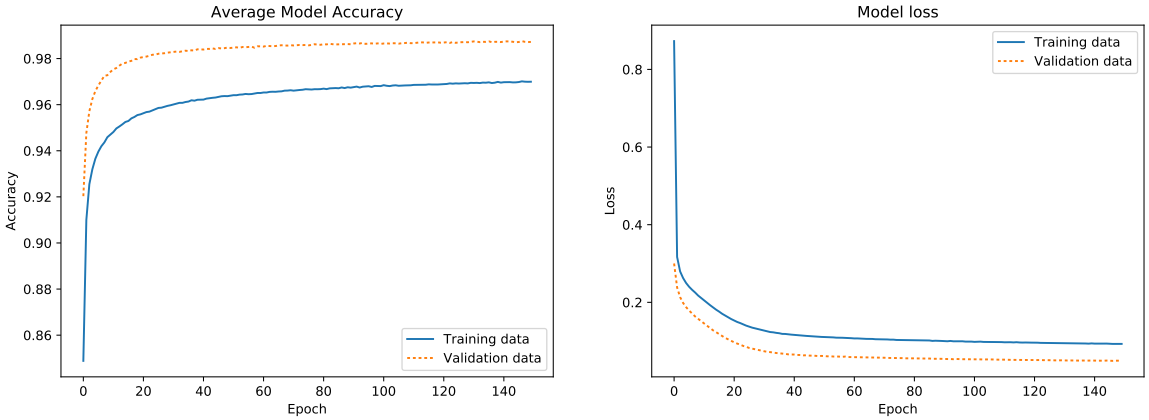

Figure 5.1. Training accuracy and loss progress.

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 seem to perform better than traditional DNNs [**?**]; 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 [**?**]. It is important to notice that in this case, the average accuracy could be slightly misleading: the models the represents a directly connected target (a source-destination pair connected by a single link), are very simple, so it is very likely for them to have an accuracy close or even equal to 100%. Nonetheless, only the 32% (obtained counting all the directly connected target 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%.

5.2 Replacing OSPF

To evaluate our model as OSPF replacement, we observe the behavior of the path prediction system in a functioning network. In particular, we use the same topology (figure 3.3) and traffic simulator adopted in the dataset generation phase; to ease the analysis process, all the links are set to the same speed. 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 simulation, each fifteen minutes long, 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 5.2 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.

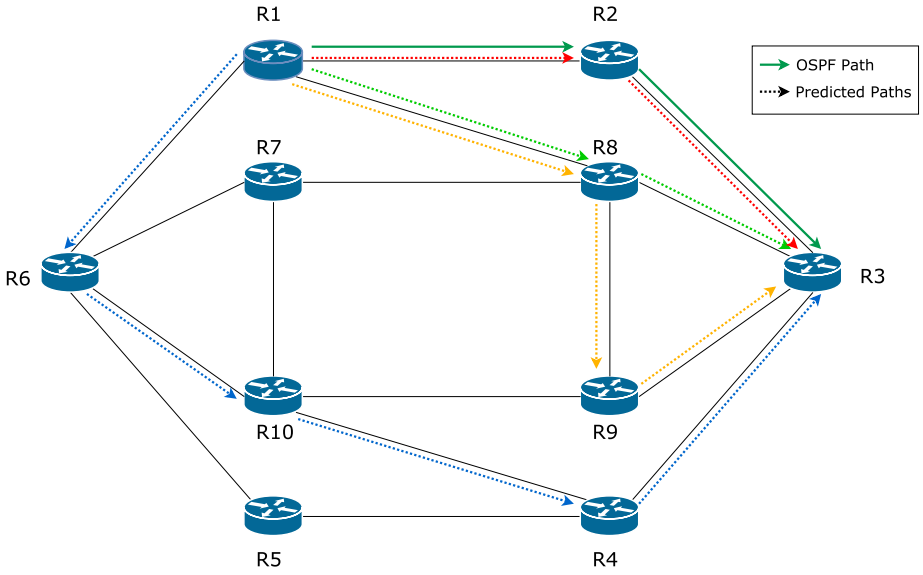

Figure 5.2. Routing policies comparison.

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%, 10%, 15%, 20% and count the number of predictions different from OSPF (table 5.1). With the loss set to zero, 43% of the time the predicted path is different from OSPF; if increased to 10%, the ratio of path different from OSPF rises to

63.5%, indicating that the system is able to detect the change. For the successive loss rates, 15% and 20%, the performance goes down a little with respectively only a 59.5% and 54.5% 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, however, even though not perfect, the results are encouraging.

Table 5.2 compares the resulting retransmission rate of our system, OSPF and equal-cost multipath (ECMP) routing strategies. 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 for our system. Overall, the system we propose, has a lower retransmission rate than the other strategies, reaching therefore a higher throughput. Figure 5.3 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 difference becomes bigger as the loss rate increases, however, while it is evident with respect to OSPF, the variation between ECMP and our system is less evident.

Link loss	**Different path rate**	**Same OSPF path rate**
0%	43.0%	57.0%
10%	63.5%	36.5%
15%	59.5%	40.5%
20%	54.5%	45.5%

Table 5.1. Path predictions different and equal to OSPF

	Routing Strategy		
Link loss rate	**OSPF**	**Equal-cost multi-path**	**Path prediction system (LSTM)**
0%	0%	0%	**0%**
10%	10%	5%	**3.65%**
15%	15%	7.5%	**6.07%**
20%	20%	10%	**9.1%**

Table 5.2. Routing strategies retransmission rate

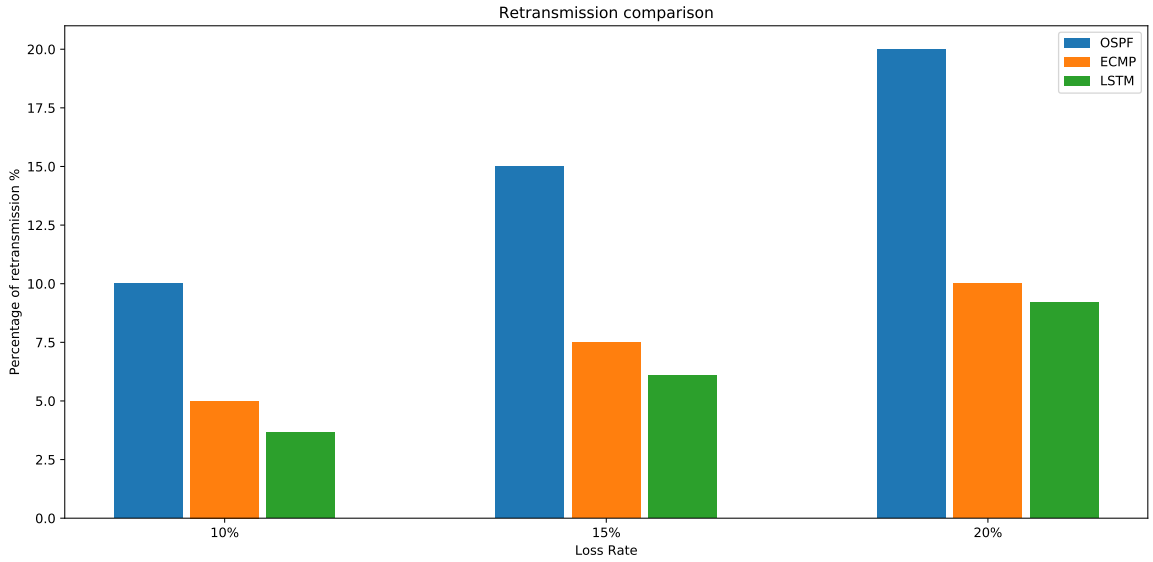

Figure 5.3. Routing policies retransmission comparison.

Chapter 6

Conclusions and future work

In this work, we presented a system meant to support mobile devices task offloading. We described an architecture designed to provide the guidelines for protocol implementations and prototyped it in a small test bed. We also developed an offloading policy to be adopted in our architecture; this policy uses deep learning to predict the best path towards a destination, increasing throughput and reducing perceived latency. The preliminary results showed that even with some limitations, machine learning could be a valid alternative to traditional routing algorithms and can be leveraged to improve network performance. Our results showed that cooperative routing can steer traffic with better performance than traditional methods, suggesting that applying machine learning in this context is a worthy research area.

There are nonetheless, some limitations that need to be addressed: the decreasing performance of the prediction system with the increasing loss rate described in the previous section, are very likely due to the poor quality of the dataset. Even though we put our effort in trying to create a complete dataset, the result is probably far from completeness: after the analysis of the first results, it is reasonable to say that the quality of our dataset is limited by constraints in both Mininet and the available computation capability. Unfortunately the Mininet emulator is not free from bugs; this fact, together with the limitation of a personal computer in terms of computational power have probably decreased the quality of our traffic simulator, resulting in a poor dataset. Having a large and representative dataset is crucial for the successful outcome of the model, therefore more effort should be put by the research community into making available public dataset. Another problem that needs to be solved is the scalability of this approach: training numerous deep learning model requires a lot of computational power; being able to reduce the number of models to train would save a lot of time and make this system more scalable.

As a future work, we intend to expand the dataset and improve its quality and measure how the quality of the model improves. Additionally, we plan to perform a deeper performance analysis by deploying a network which completely

replaces OSPF with our system and collect measurements such as throughput and latency. Finally, we want to explore new machine learning techniques, especially, reinforcement learning, which appears to be able to solve policy-based problem.

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