ADELE: An Architecture for Steering Traffic and Computations via Deep Learning in Challenged Edge Networks

ADELE: An Architecture for Steering Traffic and Computations via Deep Learning in Challenged Edge Networks

Alessandro Gaballo*‡ Matteo Flocco* Flavio Esposito* Guido Marchetto‡

*Saint Louis University, USA ‡Polytechnic of Turin, Italy

***Abstract*—Edge computing allows computationally intensive tasks to be offloaded to nearby (more) powerful servers, passing through an edge network. The goal of such offloading is to reduce data-intensive application response time or energy consumption, crucial constraints in mobile and IoT devices. In challenged networked scenarios, such as those deployed by first responders after a natural or man-made disaster, it is particularly difficult to achieve high levels of throughput due to scarce network conditions.**

In this paper, we present an architecture for traffic management that may use deep learning to support forwarding during task offloading in these challenging scenarios. In particular, our goal is to study if and when it is worth using deep learning to route traffic generated by microservices and offloading requests in these situations. Our design is different than classical approaches that use learning since we do not train for centralized routing decisions, but we let each router learn how to adapt to a lossy path without coordination, by merely using signals from standard performance-unaware protocols such as OSPF. Our results, obtained with a prototype and with simulations are encouraging, and uncover a few surprising results.

I. Introduction

Data-intensive computing requires seamless processing power which is often unavailable at the network-edge, but rather hosted in the cloud platforms. The large amount of mobile and IoT devices that has become available in the past few years produces and will produce a massive amount of data, introducing several data and network orchestration challenges and opportunities. The majority of these devices do not have or cannot handle the computational requirements to process the data they capture. For this reason, solutions that require outsourcing the responsibility to perform (some or all) computations to the edge cloud grew in popularity in recent years [1]–[5]. The process of transferring or delegating computational tasks is called offloading [2] or onloading [6]. Offloading or onloading operations are crucial for mobile devices because they lead to lower response time, lower processing time and smaller device energy consumption. In critical scenarios, such as natural or man-made disasters [7], where the physical network infrastructure is scarce or likely to be temporarily unavailable, not only is computation offloading helpful, but it becomes a necessity. This is because as latency requirements become more strict, network alternatives are scarce and data needs fast delivery. In this or similar scenarios, responsive path management solutions to direct offloading requests, $e.g.$, mobile-generated traffic steering, may become an essential application requirement.

How are we different? Traffic engineering solutions used in production today ($e.g.$, OSPF, ECMP), are performance-unaware, that is, they react only when losses or delay impact the cost assigned to a path; we argue that those are hence unsuitable for unstable or unreliable networks; moreover, in the presence of dynamic traffic and network conditions, these solutions are known to lead to sub-optimal performance [3], [5], [8]–[11]. To fill the performance-unaware gap of many (edge) network decision problems, the community has revived the decade old [12] idea of Data-driven networking [4], [13]–[15]. Despite the wide use of machine learning to solve networking problems [16], $e.g.$, traffic classification [17], latency prediction [18] and video streaming bitrate optimization [19], most of these approaches follow into two categories: either a model is trained in a centralized fashion, as a Software-Defined Network controller application [16], [20], or distributed machine learning is used to train learning models faster [21].

While several traffic engineering solutions have been devised using deep learning, see for example [16], we could not find an architecture that supports deep learning at every switch, and that provide performance-aware forwarding decisions learning from performance unaware protocols. While it may be challenging to apply our approach to very large networks, despite the recent advances in high-performance switches, we believe that our architecture can be ideal for the task offloading problem during critical networked scenarios, such as those of a data collection for situation awareness in disaster scenarios. In our architecture design, we identify the necessary and sufficient mechanisms for task and traffic offloading management within edge computing, $i.e.$, we extract the management mechanisms required to solve the task offloading problem and we compare several traffic engineering policies.

We prototype a simple yet effective protocol for task offloading and tested its programmability over MiniNeXT [22], a network emulation environment based on containers. Finally, we evaluate the performance tradeoff within several policies using different network conditions and we find a few expected and a few surprising results. By opening the deep learning "can" we found a few "worms". One surprising result can be summarized as: deep learning based traffic offloading policies, when each router runs a separate supervised learning model, may not always help improving network performance, so the training overhead time may not be justified. Another

We would like to thank the organizing committee of IEEE ICCCS 2019 for the invited paper.

message from our study lies in the poorly explored use of our performance-agnostic traffic engineering policies to generate performance-aware policies. We release the code of our prototype [23] to allow the community to exploit it and explore other (deep learning based) traffic offloading policies.

Paper outline. The rest of this paper is organized as follows. Section II illustrates a summary of the related work. Section III and Section IV describe in detail the architecture and the implementation of the LSTM based traffic offloading policy. Section V shows and discusses results obtained by the prototype in several scenarios. Finally, Section VI concludes the paper and also presents a set of open questions.

II. Related Work

The body of existing literature on topics relevant to this paper is large. In this section, our focus is only on the subset of solutions that we believe are the most appropriate to define our contributions. For a recent taxonomy on cyber-foraging or task offloading solution, we recommend these surveys [1], [2], while for machine learning solutions applied to networking, we recommend the survey of Boutaba et al [16].

Machine Learning for Offloading at the Edge. In recent years, machine learning has been used to solve various challenges, but it has not been widely adopted in edge computing problems until recently. Among the most relevant work we found Malmos [24], a mobile offloading scheduler that uses machine learning techniques to decide whether mobile computations should be offloaded to external resources or executed locally. Another example of machine learning applied to computation offloading in mobile edge networks is the work by Crutcher et al. [25]. Here, statistical regressions are used to predict the energy consumption during the offloading process as well as the time for the access point to receive the payload. As in [24], [25] we also adopt machine learning, although we use deep learning, not statistical or reinforcement learning, but our focus is on the architecture design and implementation of a policy based architecture, for the routing between the processes involved in the offloading mechanisms, as opposed to focusing merely on a single policy.

Deep Learning for Traffic Engineering. The closest work to ours is Kato et. al [26] (that we implement as one of our architecture's policies, and whose results we compare against are outperformed). They use a simple deep neural network (DNN) technique for network traffic control. In particular, their data-driven decision is based on the number of inbound packets that a router or a switch sees at a given time; data points used to train the DNN are obtained from the Open Shortest Path First (OSPF) algorithm, a standard internal routing protocol in today's networks. 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 performance in terms of signaling overhead, throughput, and average per hop delay with respect to the classic OSPF algorithm. We also use OSPF to train our own novel traffic offloading policy, but we were able to improve their accuracy and network performance results by using LSTM instead of a DNN.

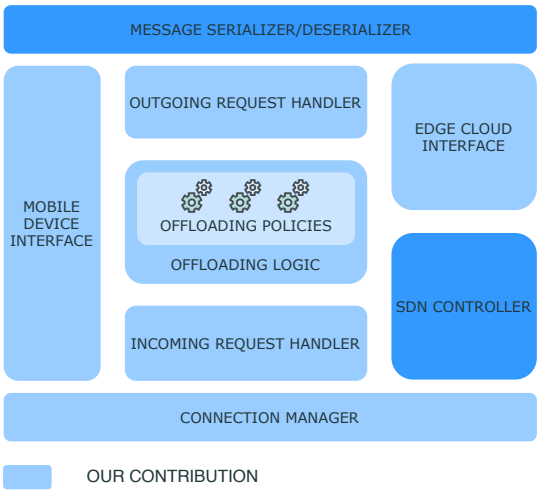


Fig. 1: Architecture overview. Our offloading logic policy uses LSTM to predict the next hop during the inter-process communication of any edge computing offloading process.

Our novel LSTM based policy is also inspired by COYOTE [8], that aims at generating optimal traffic splitting ratios in order to minimize link over-utilization. Given the limited or absent knowledge of traffic demands, this method strategically advertises fake links and nodes to adjust the splitting ratios resulting from traditional OSPF-ECMP. Their results show that the splitting ratios generated by COYOTE were closer to the optimum than those of ECMP. As in [8], we also use performance unaware OSPF and ECMP, but to train an LSTM algorithm and use such information to our advantage when the network experience losses for performance-aware routing during the offloading process. Also, we do not require the installation of another routing algorithm, which was our goal.

III. Architecture and Offloading Protocol

In this section we present the details of our proposed architecture (Figure 1), and the workflow of our offloading protocol (Figure 2).

A. Offloading Architecture

We consider a scenario in which mobile devices wish to offload tasks to the edge cloud in a network that supports SDN. The main components (invariances of the offloading problem) that we envision are: (i) a mobile device interface: interface for communications between the mobile devices and the offloading system (ii) an edge cloud interface: interface for communications between the edge cloud and the offloading system (iii) the offloading logic: each offloading policy can be programmed here to serve the edge client. Note that many offloading solutions [2] merely focus on proposing a different policy for this block, (iv) An SDN controller: this is the component responsible for enforcing the (task or virtual path) offloading policies in the hosting networked infrastructure.

The mobile device interface provides a set of primitives allowing the mobile devices and the offloading system to

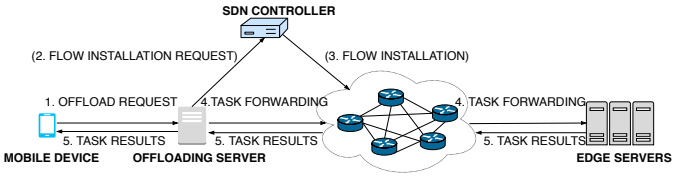


Fig. 2: Offloading workflow: mobile devices and an server orchestrate the offloading requests via an SDN controller: the best path to the best offloading server needs to be found even in presence of challenged environments such as scarce network connectivity in a natural or man-made disaster.

communicate efficiently, and formalizes the offloading request requirements. The edge cloud interface has a similar role, but in this case the primitives are meant for the communication between the offloading system and the edge cloud. 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, our architecture includes an SDN controller. In our implementation we used Ryu [27].

B. Offloading Protocol

This section describes the communication protocol of the architecture components that are: (i) the mobile device, (ii) the SDN controller, (iii) the offloading server and (iv) the edge server.

Figure 2 shows the message sequence needed to complete an offloading request, according to our protocol; we identified five logic steps required to complete an offloading request:

(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 flow rules required by the offloading policy, when needed: the required flows may be already installed. (4) The offloading server forwards the offloaded task to the edge server. (5) The edge server sends the result of the computation back to the offloading server, which in turn forwards it to the device.

Our focus is on optimizing messages (4) and (5) making the routing across the edge network between the offloading server and the edge server performance-aware. However, the task offloading ecosystem is complex, and many other mechanisms are involved. These factors include application requirements formalization, task retrieval, and edge server/-cloud discovery [1], [28]. In this work, we assume that all 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 if it can wait). We assume that the offloading server is aware of the available edge servers. Our implementation over MiniNeXT [22] presented in the following sections uses a Link Layer Discovery Protocol (LLDP). An interesting research direction, however, is the design of a fast and reliable offloading server discovery protocol. With regard to the task retrieval process, our system offers two different possibilities: one in which the task is hosted on the server with the possibility to retrieve it with a unique identifier; in the second one, the task is sent to the server and wrapped in a container. We tested our (open-source) implementation with a java package JAR and a python package EGG.

We end this section describing the implementation specifics of each message of the protocol.

a) OffloadRequest: This message is sent from the mobile device to the offloading server and includes the requirements (in terms of CPU, memory and latency), the type of the task (to support serverless computing), and, optionally, the task itself. Being an application protocol, we wanted to be agnostic to the underlying network architecture and hence we also introduced a latency priority level; this would allow us to support real-time applications.

b) Response: is a message used for several purposes: to confirm the reception of a message, signal an error or return the result of the computation. This protocol has been prototyped with Google Protocol Buffers [29] that guarantee language independence, hardware independence, and expandability with the only limitation of not being self-delimiting, requiring therefore a signaling message.

IV. Offloading Path Prediction via Deep Learning

Low-latency is a crucial aspect of task offloading systems, especially when it comes to computationally expensive tasks. Several studies exist on characterization of the slow path in OpenFlow [30]; *it was surprising to us, however, that a neglected aspect in the edge computing literature [2] is the latency minimization of the inter-process communication among offloading servers and devices, while passing through an edge network.* To reduce such latency, we optimize the end-to-end path between the communicating parties by trying to predict not congested paths. In the rest of this section we explore the use of deep learning techniques to achieve this goal.

The architecture described in Section III-A includes the *offloading logic* block, responsible for determining the criterion on which the offloading is based. Despite leaving this logic as a policy, we explore the performance of a deep-learning based offloading policy.

In particular, we exploit the congestion-agnostic limitation of traditional routing algorithms when applied in this context. These algorithms do not consider how rapid network load changes may affect the data-intensive, latency-sensitive needs of edge computing applications. Relying on higher level TCP-based solutions for congestion and flow control, that by design are (mostly) end-to-end, is insufficient. The path computed by standard routing protocols is computed by taking into account parameters such as the nominal interface speed.

The intuition behind our proposed solution 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). By collaborative we mean requiring (a priori or on demand) the participation of multiple network elements in the routing decision process. The information used by our protoype is the number of incoming packets on any given edge switch or router (node). The idea is that the packet distribution on the nodes reflects the network conditions. For example, 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. We also have to consider that the distribution of packets on the routers is influenced by the routing algorithm: nodes that appear in multiple paths will probably have a higher count than less traversed nodes because they forward 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.

Of course checking all possible outcomes is not scalable; it is known, however, that deep learning models use pruning search space strategies. We compare performance of multiple deep learning models by emulating a small network with ten routers, and using input given by the widely deployed routing algorithm Open Shortest Path First (OSPF) for training the deep learning component. We vary the network configurations and record the traffic patterns. A posteriori, 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 each source to each destination. With our approach, we are correlating traffic patterns and routing decisions; this correlation allows our system to dynamically adapt to the network conditions, a behavior that would not occur with a traditional routing algorithm.

The following section describes that steps we followed to converge to the final deep learning model.

A. Data Generation Process

The majority of datasets available to the community refer to traffic captured in datacenters, non-edge networks, or do not contain details about the underlying topology or the logged routing strategies. For these reasons, we created our own novel dataset by means of a network emulation strategy that considers all the elements we require to train our deep learning system. This includes *(i)* the network topology, *(ii)* the routing information, and *(iii)* the packet count on each node. The final dataset consists of a collection of samples containing, at any given time, the packet count together with the routing decision that was made. During the data generation process, the network is torn down and rebuilt with new link speeds, so that the OSPF configurations are different. We generated a dataset of $17,696$ samples; we then used 85% of these samples as a training set, and the remaining 15% was used as a test set.

B. Deep Learning Model

The deep learning model chosen for this work is a Long-Short Term Memory (LSTM) Recurrent Neural Network, a class of neural networks capable of using 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.

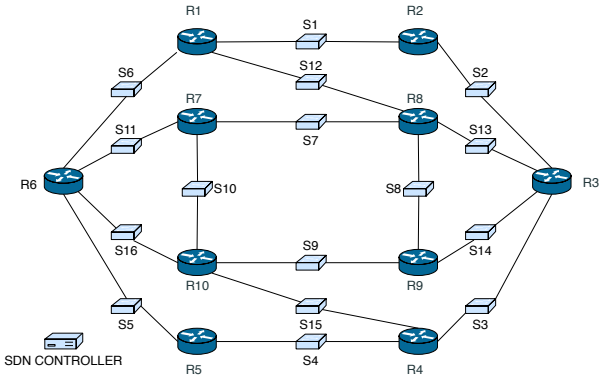


Fig. 3: Model topology: R1-R6 are outer routers while R7-R10 are inner routers. Each router runs a next-hop predictor based on LSTM.

Given the computational complexity of standard routing algorithms, training a single model to route all traffic becomes quickly infeasible as the network between mobile user and offloading server grows in size. To this aim, we trained a separate deep learning model for every source-destination pair, resulting in multiple simpler i.e., smaller, models. Assuming that each router only has a single outgoing interface, training an edge network with N routers will result in $N(N-1)$ deep learning models. Each model can be trained independently, making the training phase easy to parallelize.

1) Modeling Input and Output of the RNN: Supervised learning involves a sample space X and a label space Y, with the neural network responsible for 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 an approach similar to the one described in [26]. Given a set of outer routers O, and the set of all the routers in the edge network R, for each source-destination pair $(s, d) \in R \times O$, the deep learning system learns the next hop for destination d.

The easiest way to model the input is an N-dimensional array, with N being the number of routers in the network. Such an array is indexed by the router identifier, 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* [1] 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.

Choosing the Right 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

[1]In machine learning, one-hot is a group of bits among which the legal combinations of values are only those with a single high bit and all the others low.

	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 I: LSTM parameter exploration and deep learning model tuning: we found that: (1) despite an architecture with 128 neurons giving the highest accuracy, the marginal gain increasing the neurons diminishes; (2) an LSTM architecture with 6 layers gives lowest loss.

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 neural network too small will be unable to solve the problem while neural networks too big will probably overfit on the training set (they will find incorrect solutions). Regardless of the problem solved and of the machine learning structure, a known empirical result in the literature of deep learning states that for the majority of the problems, adding additional hidden layers has a diminishing marginal gain i.e., it often does not improve the performance of the deep learning system.

To define the parameters of our neural network, we follow these three steps: (1) we pick a source-destination pair on the edge computing network that requires a complex model, (2) we cross-validate each combination of layers and neurons of the neural network, and (3) we choose the combination whose accuracy – loss ratio is the highest.

It is important to clarify what we mean with *"source-destination pair that requires a complex model"*: cross-validating different architectures on all the possible pairs would be excessively time-consuming. Instead, we perform the validation on a single source destination pair. From now on, we refer to such pair to be a "target". 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 (virtual) routes or 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 target routers that are multiple hops apart, e.g., $(R1, R4)$.

In the cross-validation phase, we have tested 24 different configurations by trying all the combinations of the following parameters: $hidden\_layers = \{2, 4, 6, 8\}$ and $neurons = \{4, 8, 16, 32, 64, 128\}$.

Note that each hidden layer is a recurrent layer within an LSTM cell. We test each configuration 10 times on different partitions of the dataset, producing the results in Table I. The table shows two metrics: accuracy (percentage of samples correctly classified) and cross-entropy loss (distance between predicted and true label distribution); as expected, adding more layers does not significantly improve the performance. Noticeable improvements instead arose when we increased the number of neurons in each LSTM layer (Table I): 4-layers 128-neurons achieves the best performance in terms of accuracy whereas 6-layers 128-neurons has the lowest loss; it is worth noticing that given a fixed number of neurons, the accuracy typically differs by less than 1%.

From this first analysis, it is clear that to achieve the best accuracy, each layer of the neural network needs to have 128 processing units. Deciding what is the optimal number of layers is the last challenge we need to solve. 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 LSTM architecture. Figure $4a$ 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 at times negligible. Considering the limited (computational power and) training time, and the number of models we wish to train, we decided to use a network with merely 2 layers and 128 neurons, as a good performance - time trade-off.

As a result of this analysis, the final architecture is composed of: one input layer (10 neurons), two hidden layers (128 neurons ea., hyperbolic tangent activation [2]), one output layer (10 neurons, sigmoid activation).

After choosing the correct LSTM architecture, we also apply proper input normalization and regularization techniques to improve the training performance in terms of both accuracy and loss.

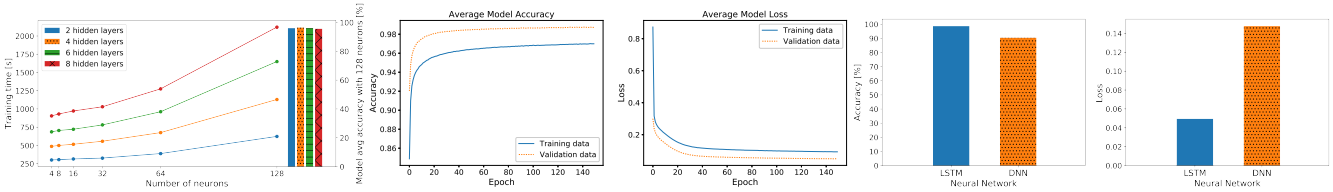
V. Evaluation Results

In this section we evaluate our architecture prototype. All our code is available at [23]. Our evaluation focus is the core of our novel LSTM based algorithm to predict least congested offloading paths. First, we detail the technologies used in our evaluation testbed, then we discuss 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 [26], 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.

A. Evaluation Testbed

Our prototype has been implemented using the following technologies: we employ Ryu [27] as an SDN controller and Google Protocol Buffers [29] as serialization/deserialization abstract syntax notation. To emulate the edge network we used MiniNExT [22], a Mininet [3] extension layer that supports

[2]The activation function defines the output of a node given an input [31].
[3]mininet.org



(*a*) Tradeoff with 128 neurons (*b*) Average accuracy (left) and loss (right) trend (*c*) LSTM and DNN performance comparison

Fig. 4: LSTM policy design and performance comparison with DNN. (a) Fewer hidden layers bring enough accuracy while significantly decreasing the training time. (b-c) LSTM outperforms DNN in terms of Loss and accuracy.

routing engines and process identifier namespaces. Finally, we used Quagga [32] as a routing software suite and Keras [33] as a machine learning library.

B. 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 LSTM RNN as a learning algorithm and build a model for each source-destination pair in our topology (Figure 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 router, the number of addresses associated with it is equal to the number of its interfaces. In our considered scenario, this resulted in 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.

Figure 4*b* 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 epoch 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 training data; even if generally unusual, the reasons of this behavior can be found in the dropout regularization technique.

Connectivity rate	Validation accuracy
30%	99.1%
35%	98.5%
40%	84.6%
45%	88.8%
50%	86.7%

TABLE II: Impact of the network density on the average validation accuracy of the deep learning model (randomly connected physical networks).

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. The system achieves an average accuracy of 98.71%, with a loss of only 0.0496, a promising result. With an accuracy of almost 99%, our LSTM-RNNs policy performs better than traditional DNNs [26], (which our prototype supports) that 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 experiment reproducibility is an open issue in machine learning [34]. The comparison of these two approaches is shown in Figure 4*c*.

Our architecture can be easily extended to support any type of topology using a configuration file with number of nodes and links. However, to prevent accuracy performance drop, a re-tuning of the LSTM parameters and the whole training is of course required. Our tests on random network topologies with increasingly high connectivity values demonstrates that, even though the model accuracy is not idea, small changes in the choice of model parameters lead to practically good performance (Table II).

C. Overwriting OSPF Routing Decisions

To evaluate its performance when overwriting OSPF rules, we observe the behavior of the path prediction system in a functioning network. In particular, we use the same topology (Figure 3) and traffic simulator adopted in the dataset generation phase; to ease the analysis process, all links are set

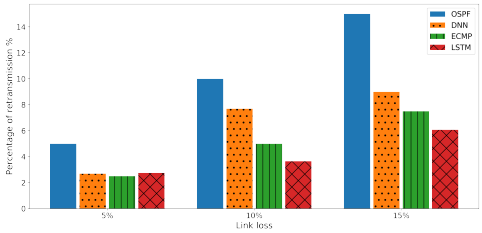


Fig. 5: Routing policies retransmission comparison. Our proposed LSTM policy has the highest throughput by minimizing retransmissions in challenged scenarios.

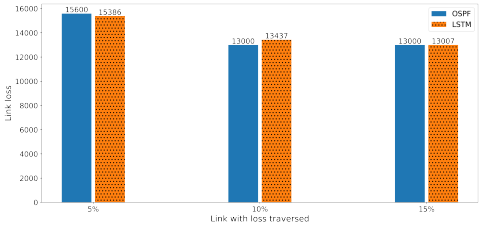


Fig. 6: Comparison of the number of (severely) lossy links traversed by OSPF and LSTM.

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 emulated edge network shows a dynamic behavior, and our prototype predicted several paths for the same destination under different traffic conditions. In particular, we run four traffic simulations, each of them for fifteen minutes, varying the loss rate on the link chosen by OSPF to connect source and destination; at the same time, the path prediction component computes a new path every five seconds. Considering Figure 3, 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. Being performance unaware, OSPF always chooses the same path, even when the link has (some) losses. To adapt the threshold, a human needs to manually reconfigure each router. Our system reacts dynamically by proposing alternative paths.

By studying the system behavior in the 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 III). 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 paths 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, equal 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 section VI. 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 IV compares the resulting retransmission rate of our system, OSPF, and Equal Cost Multi Path (ECMP) routing algorithms. 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 IV for our system. Overall, the LSTM policy that we propose has a lower retransmission rate than the other policies, therefore reaching a higher throughput. In Figure 5 we compare these three policies (LSTM, ECMP and OSPF),

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 III: Path predictions different and equal to OSPF.

	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 IV: Routing strategies retransmission rate comparison.

showing the overhead needed to transmit the same amount of data. When there is no link loss, the three policies behave very similarly; however, as soon as a loss rate is introduced, the performance gap of our proposed LSTM policy increases with the loss rate.

D. Evaluation in Challenged Scenarios

We compared several routing policies in critical scenarios, where network connectivity is scarce. We decide to simulate a network in which statistically, half of the links are affected by a loss rate; we use the same loss rates of the previous experiments (5%, 10%, 15%), running each experiment ten times, and generating traffic between five different targets. The purpose of this experiment is to understand if our approach is used with our LSTM policy, has higher resiliency than OSPF when up to half of the edge network are unavailable. To compare the performance of the two routing policies, we counted the number of times the lossy links were selected (Figure 6). The chart compares the total number of defective links traversed in all runs for each link loss rate. In this case, the LSTM policy does not introduce any significant advantage under critical circumstances; overall, the performance of the two policies are similar, with OSPF performing even better when the link loss rate is set to 10% and 15%. The reasons for the poor performance of the LSTM policy are due to our training approach; our LSTM policy predicts alternative paths based on the network conditions, proposing alternative paths. Given that half of the links in the network are affected by loss, the majority of the proposed alternative paths pass through these links, resulting in poor performance. In Section VI we give a few hints on how to overcome such limitations of these and other deep learning policies.

VI. Discussion and Conclusion

In this work, we presented a policy-based architecture for task and path offloading. Our main goal has been to provide a testing platform for task offloading and routing policies, in support of offloading tasks traversing challenged edge networks. Our virtual network testbed prototype based on MiniNExT found interesting results and was released to allow the community to compare novel or existing routing policies in different edge computing scenarios [23].

In our prototype evaluation, we focused on a specific traffic offloading policy tradeoff. In particular, we compared deep learning based routing policies with ECMP and OSPF. Our policy tradeoff analysis exposed advantages and challenges of using deep learning as alternative to traditional routing algorithms, when deployed on a single node and not as centralized (SDN) controller application.

Despite the limited size of our dataset, our initial policy tradeoff analysis results have shown how a cooperative routing policy may lead to better performance than traditional routing methods at the edge, especially with unstable network conditions such as those that arise within an IoT network trying to operate at the network edge during a natural or man-made disaster.

Acknowledgment

This work has been supported by the National Science Foundation awards CNS-1647084 and CNS-1836906.

References

[1] M. Sharifi, S. Kafaie, and O. Kashefi, "A survey and taxonomy of cyber foraging of mobile devices," *IEEE Communications Surveys Tutorials*, vol. 14, no. 4, pp. 1232–1243, Fourth 2012.

[2] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, "Edge cloud offloading algorithms: Issues, methods and perspectives," Oct 2018.

[3] G. Castellano, F. Esposito, and F. Risso, "A distributed orchestration algorithm for edge computing resources with guarantees," in *IEEE International Conference on Computer Communications*, ser. INFOCOM, 2019.

[4] A. Crutcher, C. Koch, K. Coleman, J. Patman, F. Esposito, and P. Calyam, "Hyperprofile-based computation offloading for mobile edge networks," in *The 14th International Conf, on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2017)*, Orlando, USA, Oct. 2017.

[5] A. Ventrella, F. Esposito, and A. Grieco, "Load profiling and migration for effective cyber foraging in disaster scenarios with formica," in *IEEE 4th Conf. on Network Softwarization (NetSoft 2018)*, June 2018.

[6] F. Esposito, A. Cvetkovski, T. Dargahi, and J. Pan, "Complete edge function onloading for effective backend-driven cyber foraging," in *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2017)*, Rome, Italy, Oct. 2017.

[7] J. Franz et al, "Reunifying families after a disaster via serverless computing and raspberry pis (demo)," in *IEEE LANMAN*, Washington, DC, June 2018.

[8] M. Chiesa, G. Rétvári, and M. Schapira, "Lying your way to better traffic engineering," ser. CoNEXT, 2016.

[9] D. Chemodanov, P. Calyam, and F. Esposito, "A near optimal reliable composition approach for geo-distributed latency-sensitive service chains," in *IEEE International Conference on Computer Communications*, ser. INFOCOM, 2019.

[10] F. Esposito et al, "A behavior-driven approach to intent specification for software-defined infrastructure management," in *In Proc. of IEEE NFV-SDN*, 2018.

[11] B. Eriksson, R. Durairajan, and P. Barford, "Riskroute: A framework for mitigating network outage threats," in *In Proc. of CoNEXT*. ACM, 2013, pp. 405–416.

[12] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the internet," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 3–10.

[13] J. Jiang, V. Sekar, I. Stoica, and H. Zhang, "Unleashing the potential of data-driven networking," in *International Conference on Communication Systems and Networks*. Springer, 2017, pp. 110–126.

[14] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

[15] D. Chemodanov, F. Esposito, A. Sukhov, P. Calyam, H. Trinh, and Z. Oraibi, "Agra: Ai-augmented geographic routing approach for iot-based incident-supporting applications," *Future Generation Computer Systems*, 2017.

[16] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, Jun 2018. [Online]. Available: https://doi.org/10.1186/s13174-018-0087-2

[17] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[18] V. Bui, W. Zhu, A. Pescape, and A. Botta, "Long horizon end-to-end delay forecasts: A multi-step-ahead hybrid approach," in *2007 12th IEEE Symposium on Computers and Communications*, July 2007.

[19] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 197–210.

[20] A. Scalingi et al, "Scalable provisioning of virtual network functions via supervised learning," in *2019 IEEE Conference on Network Softwarization (NetSoft) (NetSoft 2019)*, Paris, France, Jun. 2019.

[21] R. Mayer and H. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques and tools," *CoRR*, vol. abs/1903.11314, 2019. [Online]. Available: http://arxiv.org/abs/1903.11314

[22] "MinineXt," http://mininext.uscnsl.net/.

[23] A. Gaballo and F. Esposito. ADELE code github.com/alegaballo/adele.

[24] H. Eom, R. Figueiredo, H. Cai, Y. Zhang, and G. Huang, "Malmos: Machine learning-based mobile offloading scheduler with online training," in *IEEE MobileCloud 2015*, March 2015, pp. 51–60.

[25] A. Crutcher, C. Koch, K. Coleman, J. Patman, F. Esposito, and P. Calyam, "Hyperprofile-based computation offloading for mobile edge networks," *arXiv preprint arXiv:1707.09422*, 2017.

[26] N. Kato, Z. M. Fadlullah, B. Mao, F. Tang, O. Akashi, T. Inoue, and K. Mizutani, "The deep learning vision for heterogeneous network traffic control: Proposal, challenges, and future perspective," *IEEE Wireless Communications*, vol. 24, no. 3, pp. 146–153, 2017.

[27] "Ryu," https://osrg.github.io/ryu/.

[28] F. Esposito, F. Paganelli, and R. Fantacci, "A decomposition-based architecture for distributed cyber-foraging of multiple edge functions," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, June 2018, pp. 247–251.

[29] "Google protocol buffers," developers.google.com/protocol-buffers/.

[30] R. Sanger, B. Cowie, M. Luckie, and R. Nelson, "Characterising the limits of the openflow slow-path," in *IEEE NFV-SDN*, Nov 2018.

[31] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, 2016.

[32] "Quagga," http://www.nongnu.org/quagga/.

[33] "Keras the python deep learning library," https://keras.io/.

[34] B. K. Olorisade, P. Brereton, and P. Andras, "Reproducibility in machine learning-based studies: An example of text mining," 2017.