Applications of ML in Networking and IoT

Saif Eddine Nouma

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université de Carthage

Ecole Polytechnique de Tunisie

وزارة التعليم العالي و البحث العلمي

جامعة قرطاج

المدرسة التونسية للتقنيات

Graduation Project Report
to obtain :

The National Engineering Diploma from the Ecole Polytechnique de Tunisie

Applications of ML in Networking & IoT

Elaborated by:
Saif Eddine NOUMA

within:

Supported 04/09/2020 before the examination board

Academic Year: 2019-2020

Abstract

The emergence of Machine Learning (ML) has increased exponentially in numerous applications, including computer vision, speech recognition and natural language processing. This is due to the intersection of the massive data generated by IoT devices (e.g. sensors, mobile phones and smart devices) and the computing power available in cloud and data centers. Hence, complex systems have been designed to improve the quality of life such as health monitoring, smart homes and intelligent transport systems. However, the cloud computing has a number of issues related to the trade-off between efficiency and latency. Indeed, the former cannot ensure the performance for real-time applications due to the long-range network latency between the cloud and end devices. For that reason, we propose to leverage these issues in this work by proposing alternative solutions. First, we present a real-time network traffic prediction using different DL models and compare their performance to a classic ML model named SVR by using the Hyperparameters tuning process. In fact, traffic prediction is a tool for Internet service providers to serve the customers better QoS, i.e. lessen the latency for sensitive applications. Second, we study the edge computing paradigm that attempts to unleash DL services by bringing the cloud capabilities close to end users by deploying edge servers. Thus, we propose a distributed inference framework for DL models. The latter intends to attain a high performance while minimizing the computation latency for sensitive services and the energy consumption of IoT devices. Extensive evaluations on the standard CNN "ResNet20" for object recognition task using CIFAR10 dataset shows the effectiveness of the proposed framework in controlling latency versus inference accuracy. To further enhance the distributed inference framework, we propose automating the offloading decision and resource allocation of computing tasks for IoT devices using Deep Q-Network (DQN), a well-known DRL algorithm. Several Simulations show that DQN has been successful in managing execution of tasks belongs to IoT devices in order to make the best use of available resources.

Keywords: Network Traffic Prediction, Time Series, Machine Learning, Recurrent Neural Networks, Edge Computing, Distributed Computation, Convolutional Neural Network, Deep Q-Network.

Acknowledgment

I would like to express my deep sense of gratitude to my supervisors, Pr. Khalil Drira and Dr. Hassan Hassan for their availability, valuable guidance, and constant supervision during this graduation internship.

Any attempt at any level can't satisfactorily completed without the support and guidance of Pr. Nawal Guermouche through her constructive criticism and her constant supervision.

I send my warm thanks to my academic supervisor and my teacher Pr. Takwa Abdellatif for her availability, and continuous support. I want to thank her especially for her constructive criticism and revision of this work.

My gratitude goes also to all the members of SARA team at LAAS, for their hospitality and for their valuable support and assistance.

Furthermore, I am highly indebted to all my family members for their endless love and support.

Finally, I would like to convey my most respectful thanks to the members of the Jury who kindly agreed to evaluate my humble work.

Table of Contents

List of Figures

List of Tables

Acronyms

ANN Artificial Neural Networks.

AP Access Point.

AR Augmented Reality.

CC Cloud Computing.

CNN Convolutional Neural Network.

CNRS Centre National de Recherche Scientifique.

DL Deep Learning.

DNN Deep Neural Network.

DQN Deep Q-Network.

DRL Deep Reinforcement Learning.

EC Edge Computing.

EEoI Early Exit of Inference.

FFNN Feed Forward Neural Network.

GRU Gate Recurrent Unit.

ICMP Internet Control Message Protocol.

IoT Intenet of Things.

IP Internet Protocol.

LSTM Long-Short Term Memory.

ML Machine Learning.

PCAP Packet Capture.

QoS	Quality of Service.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
SVM	Support Vector Machine.
SVR	Support Vector Regression.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
ZB	Zettabytes.

Introduction

Thanks to advances in cloud computing capabilities and data center storage, Deep Learning (DL) has been very effective in ML across a wide range of application domains, including computer vision, virtual/augmented reality, speech recognition and natural language processing. For instance, the worldwide translator "Google Translate" uses Long-Short Term Memory (LSTM) network, a DL model dedicated for natural language processing, in order to mine the semantics of the sentences instead of mapping sentence to sentence [16]. However, DL's high performance comes at the cost of a long training of a deep network that encapsulates millions of parameters. Additionally, DL inference is computationally expensive due to the high dimensionality of the input data and the millions of computations that need to be performed over multiple layers. Furthermore, it is becoming very difficult for the cloud to meet the demands of computing tasks and real-time response due to the enormous expansion of IoT devices connected to the Internet. According to Statista, the number of IoT connected devices has increased by 200% from 2015 to 2020 [29]. The study also found that it would reach approximately 75 billion connected devices in 2025 as depicted in fig. 1. Moreover, Cisco Annual Internet report reveals that IoT devices will produce around 850 ZB of data each year starting from 2021 [30].

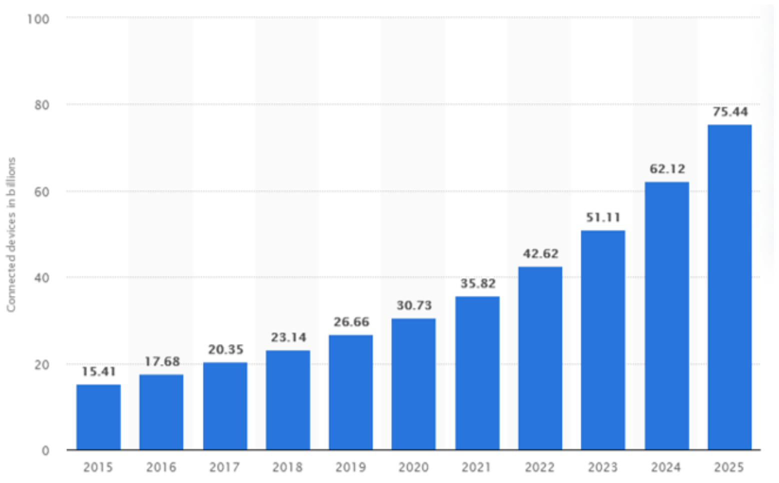


Figure 1: The number of IoT connected devices from 2015 to 2025 [29]

Thus, actual cloud computing capabilities are unable to manage this massive data generated from billions of devices. Indeed, the former cannot guarantee the delay requirements for real-time applications of end devices (e.g. sensors, mobile users, cameras). Besides, offloading the computation from end devices to the cloud causes privacy issues related to sensitive data such as sending personal information or indoor image capture [31].

A possible alternative is to analyze locally the massive flow of data generated by the end devices . Yet the problem of computing resources and energy limitations arises. In fact, IoT devices are known as resource-constrained devices because they were unable to meet the DL needs. Hence, edge computing paradigm [32] appear to be an attractive solution that meets the issues related to latency, scalability and privacy. The latter consists of migrating the DL computations near the end devices by deploying local edge servers so that cloud resources becomes available and with a promising response time. As a result, edge servers unburden the huge computation demands from end devices to the cloud. For instance, a possible use-case is to deploy edge servers to enable interactive experience at home by creating augmented reality service for online shopping [33]. Even Edge Computing (EC) has a number of issues related to the computing power. In fact, edge servers have not been able to cover computing capabilities offered by cloud servers. As a result, edge servers cannot respond to the high resource requirements that DL applications need.

Therefore, EC and CC should work together in order to respond to the IoT needs. In fact, while EC can provide low latency and privacy, CC offers high power resources and scalability. Actually, tremendous reseach that is going around in the thematic of collaboration between EC and CC. It is in this perspective, that we want to implement, during this graduation project, an alternative solutions that responds to the high demand of end devices using the collaboration between cloud computing and edge computing.

This report summarizes the work we have achieved and the results that we have reached since the month of February.

The outline of this report as follows:

- **General Framework of The Project**
- **Mathematical Background**
- **Network Traffic Prediction**
- **Distributed DL Inference in EC scenario**
- **Computation Offloading for multi-user Mobile EC**

Chapter 1

General Framework of the Project

Introduction

In this chapter, we start by representing the host laboratory **LAAS-CNRS** and the general context of this internship.

1.1 Presentation of the host organization: LAAS-CNRS

1.1.1 Presentation of CNRS

The CNRS (Centre National de la Recherche Scientifique) is the french national research organization and the main basic science institution in Europe. It is founded on several reseach units. Each research unit has a specific mission in specific domains. Therefore, CNRS carries out his activity in all fields of knowledge.

1.1.2 Presentation of LAAS-CNRS

The LAAS-CNRS (Laboratory of Analysis and Architecture of Systems) is a CNRS research unit, established in 1968, based in Toulouse.

The research at LAAS-CNRS aims at fundamental understanding of complex systems while considering the use that can result from it. Actually, The laboratory holds 8 scientific departments defines the directions for the coming years and coordinate the activities of 22 research teams. In fact, the names of the scientific departments are as follows:

- Crucial Computing

- Networks and Communications
- Robotics
- Decision and Optimization
- Microwaves and Optics : from Electromagnetism to Systems
- Technology & Instrumentation for the Monitoring of Complex Systems
- Micro Nano Bio Technologies
- Energy Management

During my internship, I have worked in SARA team (Services et Architectures pour Réseaux Avancés) which his main thematic lies in networking and communication. In fact, the team's contributions relate in particular to the development of methods, models as well as to the proposal of architectures, protocols and services.

Chapter 2

Mathematical Background

2.1 Introduction

In this chapter, we provide an overview of the theoretical background of ML models used in this project. The first section focuses on the presentation of ML and explains the different categories of ML algorithms. A second part explains Deep Learning (DL) concepts and its main components. Then, we present DL models used in this project. The third section introduces the basic concepts in RL with a formulation of problems solved by RL models. In the last part, we turn our focus to Deep Reinforcement Learning (DRL), a combination of RL and DL, and we discuss the different class of RL algorithms.

2.2 Machine Learning (ML)

2.2.1 Definition

Machine Learning is a subfield of Artificial Intelligence, which involves the development of self learning algorithms in order to make forecasts on new data. Because of the vast amounts of data generated in our real world, taking advantage of these algorithms is becoming extremely potential. Additionally, ML algorithms allow data patterns to be captured autonomously in order to make data driven decisions.

2.2.2 Types of ML

In the literature, ML algorithms are classified into three categories:

- **Supervised ML:** In this class, we want to learn a predictive function f that maps inputs X to the desired outputs Y. A training process will be applied until the model accomplishes a desired level of performance on a training data.

- **Unsupervised ML:** Unlike supervised ML, this type of algorithms doesn't impose labels for each row of data. Therefore, it learns hidden patterns in the data in order to make clustering and feature reduction. For instance, it's widely used for segmenting customers in different groups for specific intervention.
- **Reinforcement Learning (RL):** RL is a type of learning where a machine is trained to make optimal decisions autonomously by exposing it to an external environment.

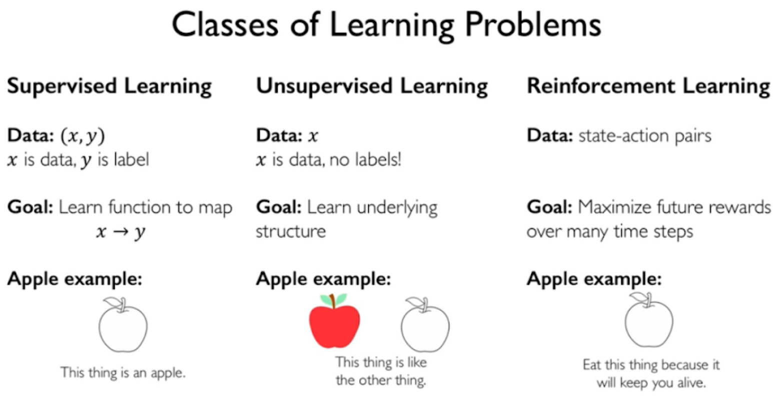

Classes of Learning Problems
Supervised Learning
Unsupervised Learning
Reinforcement Learning
Data: (x, y)
x is data, y is label
Data: x
x is data, no labels!
Data: state-action pairs
Goal: Learn function to map $x \to y$
Goal: Learn underlying structure
Goal: Maximize future rewards over many time steps
Apple example:
This thing is an apple.
Apple example:
This thing is like the other thing.
Apple example:
Eat this thing because it will keep you alive.

Figure 2.1: Types of ML problems [3]

Hence, this project focuses on both supervised ML and RL.

2.3 Deep Learning (DL)

2.3.1 Introduction

Deep Learning is a new subfield of ML research. In fact, it was inspired by the structure and the function of the brain in processing data and creating patterns for use in decision making. Back through time, DL paradigm was explored since 1950 [34]. Then, a lot of research was carried out until 1980, when they were struggling to train their proposed models and to provide the expected storage for their data sets due to the limited hardware capacities. In 1989, the first handwritten digits recognition model was build by Y. Le Cun using back-propagation networks [35]. Nowadays, DL surpasses the human expert knowledge in several fields such as computer vision, natural language processing and speech recognition by dint of the high computing performance and the incredible available storage in cloud data centers. For instance, Google's DeepMind developed a system capable of beating a professional Go player in 2015 using deep reinforcement learning [36].

2.3.2 Artificial Neural Networks (ANN)

2.3.2.1 Definition

Neural Network is a series of algorithms that tried to mimic the function of the brain in order to recognize the relationships in a set of data. It was discovered in the 80s and the early 90s and was utilized later in many applications such as face recognition, object detection and many other applications that are very difficult to solve unless the neural networks are existed.

An ANN is based on a collection of connected units where each neuron can transmit an information to another neuron. A neuron could contain a weight that varies as learning proceeds in order to minimize the loss function which we will refer to it in the next parts.

In the theoretical perspective, it is simply a non-linear function that maps the inputs to the outputs by finding correlations and could be written in a formula:

$$Y = f(X) \tag{2.1}$$

where Y represents the Output, X is the Input and f is a non-linear function.

2.3.2.2 ANN Architecture

In ANN, each node presents a neuron, and each column represent a layer. The first layer is called the input layer. The rightmost or output layer contains the output neurons, the number of neurons represents the number of classes we are classifying. Any intermediate layer is called a hidden layer. In addition, each neuron has a propagation function that transforms the outputs of the connected neurons often with a weighted sum followed by an activation function.

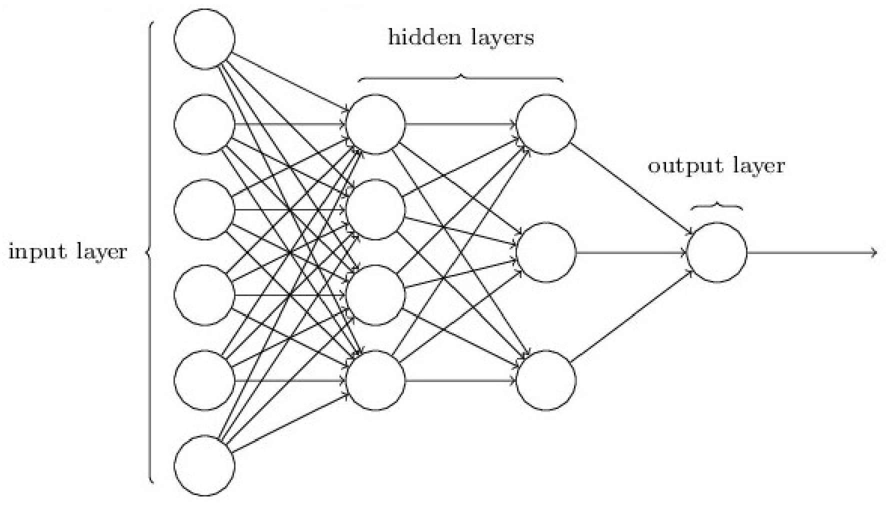


Figure 2.2: ANN Architecture [37]

The example depicted in fig. 2.2 of an ANN is composed of 6 neurons as input layers, 1^{st} hidden layer with 4 layers, 2^{nb} hidden layer with 3 neurons, and the output layer contains one single neuron.

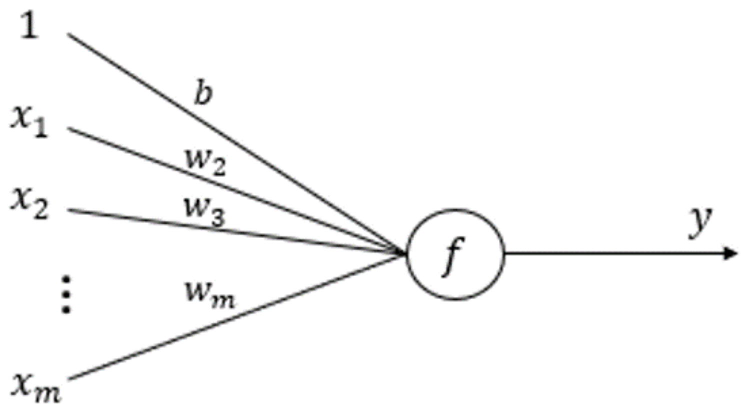


Figure 2.3: Propagation Function for a Single Neuron

Indeed, the relationship between the neurons in each layer is a nonlinear function. As depicted in fig. 2.3, each neuron which is connected with neurons from the previous layer have $m + 1$ parameters where m represents the number of neurons in the previous layer and it could be written in a formula:

$$y = f(b + \sum_{i=1}^{m} w_i.x_i) \tag{2.2}$$

where:

- b: represent the bias which avoid the model from the overfitting problem where the NN adopts its parameters to the training set and achieve low performance on the test set.
- f: represents a nonlinear activation function applied to the weighted sum of inputs. It is one of the key knob that make it possible to model non-linear problems. Actually, we need to use nonlinear model to make sense of a complicated and a high dimensional data.

The most used activation functions are shown in the table. 2.1:

Activation Function	Formula
ReLU	$f(x) = \max(0, x)$
Tanh	$f(x) = \frac{2}{1+\exp(-2x)} - 1$
Sigmoid	$f(x) = \frac{1}{1+\exp(-x)}$

Table 2.1: Examples of Activation Functions

2.3.2.3 Forward Propagation

As the name reveals, the input data will be forwarded through the network. Each hidden layer accepts input data, processes the data by calculating the activation function of each received signals for each neuron, and then continues to extract the information until it reaches the output layer where we have the prediction result.

2.3.2.4 Backward Propagation

Backward propagation is the essence of neural network training. It is the practice of fine-tuning the weights of a neural network based on error rate induced by the loss function. Backward propagation uses the gradient of the error function with respect to the weights and biases of the model to find the correct direction to minimize the error. The optimization algorithm and the learning rate variable are those who control the correction. This mechanism is based on loss function and optimization functions that I did not mention in my report so as not to go deeper into the concept of the neural network. For those who are interested in understanding this concept, I recommend them viewing the courses of DeepLearning.ai offered by Andrew Ng in Coursera platform [38].

2.3.3 Convolutional Neural Network (CNN)

2.3.3.1 Introduction

CNN is a class of ANN and the most used type in Computer Vision and Augmented Reality. It was proposed by **Yann Lecun** in order to diminish the complexity of the calculations in a complex neural network i.e. a fully connected neural network.

2.3.3.2 Definition

The CNN inherits the same architecture of an artificial neural networks i.e. he possessed an input layer, a series of hidden layers, and an output layer. Unless, he has specific hidden layers, where each hidden layer consists of a convolutional layer that convolves using multiplication or dot product. Hence, we present the most used types of layers in CNN as demonstrated in the next section.

2.3.3.3 CNN Layers

- **Convolutional Layer:** The convolutional layer is the core of building block of a CNN that does most of the computational heavy lifting. This latter is a collection of filters that are convoluted with the previous layer that are designed to extract features and information. Indeed, each filter has its own hyperparameters such as: Dimensions, Stride, Padding.

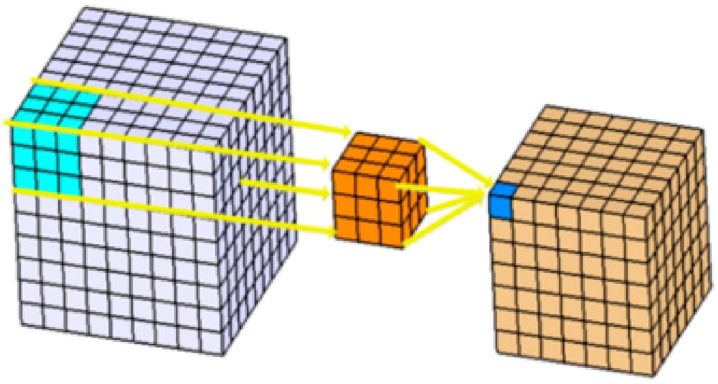


Figure 2.4: Convolutional Layer Example [39]

- **Pooling Layer:** Pooling Layers are used after convolutional layers in a CNN in order to reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single layer. There are many types of Pooling layer such as:
  - **Max-Pooling**: Compute the maximum value in each cluster in order to extract the most important pixel in the patch.
  - **Average-Pooling:** Compute the average value in each cluster in order to extract the mean value in the patch.

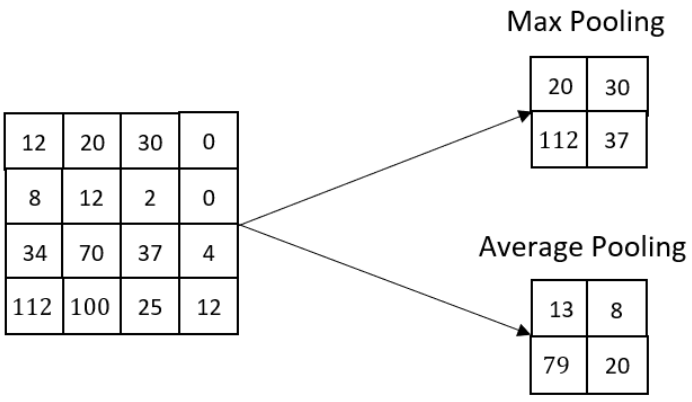


Figure 2.5: Example of Pooling Layers

- **Fully Connected Layer:** This layer is used to make high-level reasoning by connecting all the activation signals from the previous layer. Usually, it is used to perform classification based on features extracted from the convolutional layers.

2.3.4 Recurrent Neural Network (RNN)

2.3.4.1 Definition

RNN is a subclass of ANN that contains feedback loops as opposed to FFNN. As depicted in fig. 2.6, these recurrent connections allow the information about previous time step to persist in the internal state of the network and hence influence the output efficiency. For instance, RNN have recieved huge success in dealing with data such as time series which have a sequential order that needs to be followed in order to be understandable.

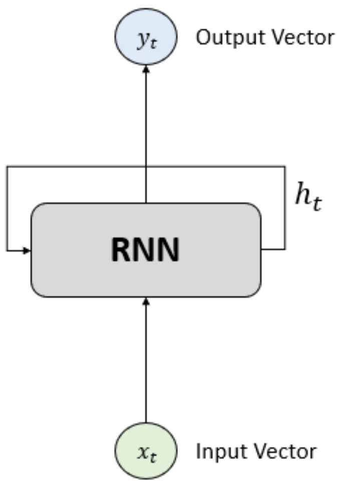


Figure 2.6: Example of a Simple RNN

Formally, the output of a recurrent neuron could be formalized as follow:

$$\hat{y}_t = f_Y(h_t) \tag{2.3}$$

Where:

- h_t: represents the hidden state of the network. In fact, it stores the actual input signal each time we make a forward pass and apply a non linear function as demonstrated below:

$$h_t = f_W(x_t, h_{t-1}) \tag{2.4}$$

- f_W: represents a non linear function that merge the signals of both the input vector and the previous hidden state.
- f_Y: represents a non linear function applied to the actual hidden state in order to produce the predicted vector.

As a result, the RNN model updates both its hidden state and the predicted vector within each forward pass.

For example, one of the most used nonlinear functions for the RNN update is represented as follows:

$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t) \tag{2.5}$$

$$y_t = W_{hy}^T h_t \tag{2.6}$$

Where: W_{xh}, W_{hh} and W_{hy} represents respectively the weight matrices of the input, the hidden state and the output.

Figure 2.7 depicts a frequent form of visualization for RNNs known as unfolding. Indeed, an RNN is unfolded by developing its computational graph over time and removing the loop connections. Thus, RNN unfolding reveal the repetition in the recurrent neuron and show the number of time steps required to complete a task.

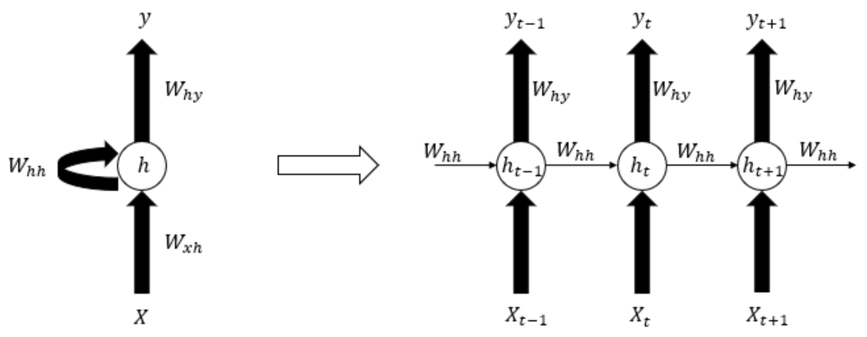


Figure 2.7: Unfolding of RNN neuron

2.4 Reinforcement Learning (RL)

2.4.1 Definition

RL is a subfield of ML that allows machines to learn automatically by interacting with an external environment without the need to acquire a data set for training and evaluation.

More formally, RL is a computational approach to learn from interaction where an agent pick out an action and receives a numerical signal, named reward, that informs the agent of the relevance of the action chosen. Instead of drawing on the experience of experts to learn from, RL allows an agent to learn on its own by focusing on maximizing the cumulative rewards received from the environment. Actually, the concept of learning by interacting with an external environment is an imitation from the natural learning process of human being.

In this section, we present the elements of RL. Then, we turn our focus to explain in advance the deployed algorithm in this project.

2.4.2 RL Components

Reinforcement Learning is based on learning behavior through interactions with the environment in a discrete time steps. As depicted in figure 2.8, the interaction has two main stages, observation and action. The interaction is basically between the agent and the environment. Hence, the environment begins by sending the current state s_t to the agent. Based on the knowledge acquired by the agent, he will take an action a_t in response to that state. As a result, the environment emits a feedback based on the chosen action. The feedback contains a reward signal r_t and a new state s_{t+1}. The former represents the influence of the action on the environment and measures the success or failure of the agent's action while the latter is the new state of the agent. This process is repeated between the agent and the environment until a stop condition has been validated.

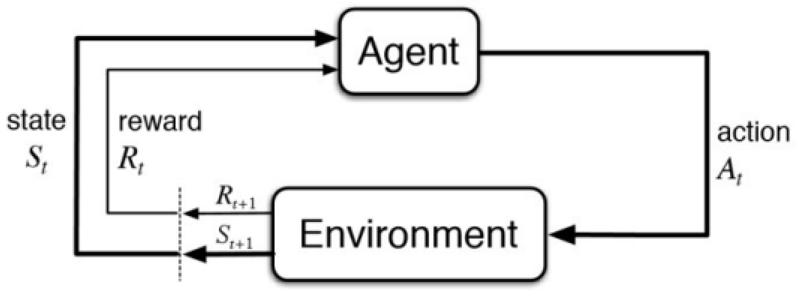


Figure 2.8: Interaction Between Agent and Environment

The agent attempts to maximize the discounted cumulative reward R_t instead of the current reward. Indeed, the cumulative reward is expressed as follows:

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{t+n} r_{t+n} + ... \tag{2.7}$$

Where:

- γ represents the discount factor. It is always comprised between 0 and 1. It was designed to give more importance to earlier rewards. From a mathematical perspective, it ensures the convergence of the time series.

2.4.3 Policy

The agent perceive the environment and decides the optimal action according to a policy function π. Actually, the policy represents the core of RL in the sense that it is sufficient to determine behavior [53]. It informs the agent what action to be selected in a given state. It could be deterministic or stochastic depending on the problematic and the choice of the model.

In the case of a deterministic policy, it is simply a mapping function from perceiving states of the environment to the actions to be taken in those states. Hence, we have the following expression:

$$\pi(s_t) = a_t \qquad s_t \in S, a_t \in A \tag{2.8}$$

Otherwise, the policy becomes a probability distribution in case of a stochastic policy and it could be expressed as follows:

$$\pi(a_t|s_t) = pi \qquad s_t \in S, a_t \in A, pi \in [0,1] \tag{2.9}$$

During the interaction between the RL agent and the environment, the agent tries to converge to the optimal policy π^*. In essence, there are several learning algorithms that exists in the literature that aim to converge the policy using different strategies.

2.4.4 Value Function

Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long term. For instance, a state might always provide a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. Indeed, value functions estimates the discounted cumulative reward R_t for every possible state. It could be formalized as follows:

$$V_\pi(s) = E_\pi(R_t|s_t = s) = E_\pi[\sum_{i=t}^{\infty} \gamma^i r_i | s_t = s] \tag{2.10}$$

2.4.5 Q-Function

A Q-function defines how good an action is for each state. It represents the expected cumulative reward for every action in each state. Hence, it could be expressed as follow:

$$Q_\pi(s,a) = E_\pi(R_t|s_t = s, a_t = a) = E_\pi[\sum_{i=t}^{\infty} \gamma^i r_i | s_t = s, a_t = a] \tag{2.11}$$

The use of Q-function is more appropriate than Value function since it evaluates every possible action that could be token in a given state. In contrast, value function takes into account every possible actions and observes all possibles next states and selects the action based on the highest state value which appears too complicated and impractical. Hence, the majority of RL research studies focuses on estimating Q-Function rather than value function.

Furthermore, there is a relationship between Value function and Q-function which could be formalized as follows:

$$V_\pi(s) = \max_{a \in A} Q_\pi(s,a) \tag{2.12}$$

Ultimately, the agent needs a policy to infer the best action in its state s_t. Indeed, the strategy is to choose the best action that maximizes future rewards. The optimal policy could be expressed as follows:

$$\pi^*(s_t) = \arg\max_{a \in A} Q(s_t, a) \tag{2.13}$$

2.4.6 Exploration vs. Exploitation Trade-Off

Imagine that you are going out to eat with your friends tonight. When you get to the restaurant, you will have to decide what to order. You've been there a couple of times before and you always order the same thing. So you know you are going to be pretty happy if you order it again. Many of the other items though look really tasty. How do you decide when to order the same good meal again, or try something new.

Actually, the same issue arises for RL agent when it comes to selecting actions based on the actual optimal policy or exploring unvisited states by selecting random action. Indeed, exploration is needed because there is always uncertainty about the actual estimated Q-values. However, the agent risks to end up in a bad state and accumulates negative rewards. Hence, this is so called exploration exploitation trade-off.

To solve this dilemma, there are several approaches that attempts to balance exploration and exploitation. For instance, ε-Greedy is the most widely used strategy in the literature due to its simplicity. Indeed, the agent selects an action randomly with a certain probability ε. Otherwise, the latter exploits the current knowledge by choosing the optimal action with the highest Q-value. As a result, ε-Greedy strategy could be formalized as follows:

$$a_t = \begin{cases} \arg\max_{a \in A} Q(s_t, a) & \text{with a probability } 1 - \epsilon \\ \text{pull random action} & \text{with a probability } \quad \epsilon \end{cases} \tag{2.14}$$

Besides, it is ε-Greedy strategy that we have adopted in our project.

2.5 Deep Reinforcement Learning (DRL)

2.5.1 Definition

In RL, a large amount of memory is usually required to store value functions and Q-functions. Actually, the easiest way to represent an RL problem is to store each state-action pair in the form of table. This is known as Tabular Method. In fact, most of RL uses cases have an infinite state space, which makes it impossible to store value functions or Q-functions in tables. Therefore, the trial and error interaction between the agent and the environment is hard to be learned due to the huge computation and the storage requirements for real world problems. This is where Deep Learning (DL) comes into place in order to estimate Q-functions and value functions. As a consequence, the fusion of DL and RL becomes a promising solution in order to leverage complex real world use cases. For instance, the estimator of Q-function could be expressed as follows:

$$\hat{Q}(s, a, \theta) \simeq Q_\pi(s, a) \tag{2.15}$$

where:

- θ : represents a real vector that contains the parameters of the estimator. These weights are updated throughout the learning of the agent in the aim to converge to the actual discounted total reward.
- $\hat{Q}$: represents the estimator of Q-function.

Q-Function could be approximated using Artificial Neural Networks (ANN). Indeed, ANN could be trained using gradient descent to minimize the error between $\hat{q}$ and the discount cumulative reward R.

2.5.2 Deep Q-Network (DQN)

Based on estimating Q-Function using neural networks, Mnih et al. [54] introduce Deep Q-Network algorithm in 2015 which have achieved high performance in learning ATARI games.

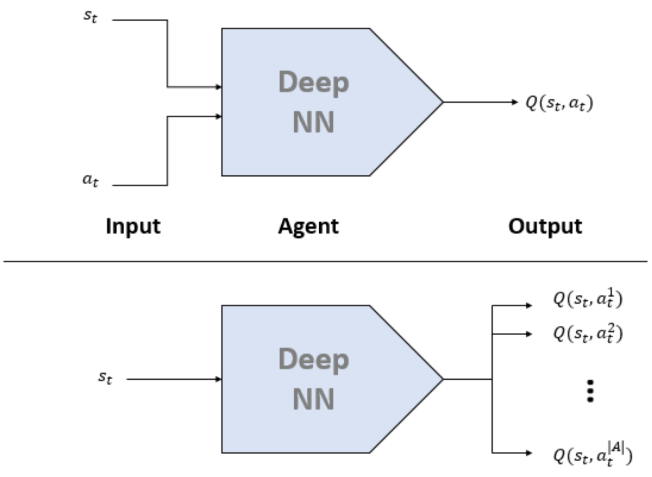


Figure 2.9: Two Possible Configurations of DQN

As depicted in figure 2.9, the neural network in DQN has two configurations: 1) takes as input the state and estimate the Q-values for each possible action 2) takes as input both the state and its corresponding action and the model estimate the Q-value for this state-action pair.

2.6 Conclusion

In this chapter, we had an overview on the main components about ML. We have introduced the DL paradigm and the various ANN models used in this project. Then, we presented RL as a self-learning approach and its main elements. Finally, we discussed the benefits of DL for RL and we presented the well-known algorithm Deep Q-Network (DQN).

Chapter 3

Network Traffic Prediction

Introduction

We start this chapter by a general description of network traffic prediction. Then, we turn our focus on exploratory data analysis and the modelling phase. Finally, we end up with performance evaluation of the fitted models and the Hyper-Parameters tuning.

3.1 General Description

3.1.1 Problem Statement

With the rapid growth of Internet Technology, network traffic is growing exponentially. As a result, the resource management is becoming more difficult and more complex for Internet service providers. Hence, current traditional models cannot forecast network traffic that acts as a nonlinear system. In this chapter, we propose to leverage the emerging deep learning techniques to predict network traffic. Extensive experiments were conducted to evaluate the performance of the proposed models using real-world data and we compare the accuracy of the chosen model with a baseline model named Support Vector Regression (SVR).

3.1.2 Project Goals

By improving the precision of network traffic prediction, this yields to significant gains for network providers. In fact, traffic prediction has become the engine for many interesting applications such as resource allocation, short-term traffic scheduling, Long-Term capacity planning, network design and anomaly detection.

3.1.3 Categories of Network Traffic Prediction

We can distinguish four types of prediction according to [2] based on the forecasting time scale:

- **Real-Time Forecasting:** This type requires the shortest time interval between data that do not exceed a few minutes, which can be used to establish an online real-time forecasting system.
- **Short-Term Prediction:** It often linked with a duration of one day to few hours. Hence, its application is mainly anomaly detection and decision making in critical situations.
- **Medium-Term Prediction:** Its duration is from several days to one month. In fact, it can be used to guide resource planning.
- **Long-Term Prediction:** Long-Term prediction is usually one year. Hence, it can be used for economic investment and designing infrastructure.

Thus, based on the duration of collected Internet traces which is about several days, we have been interested in Real-Time forecasting class. Specifically, we want to forecast the volume of traffic in the range of seconds and minutes, using as input a time series signal comprised by the previous observations of this value sampled at one-minute intervals.

3.1.4 Related Work

In the literature, several approaches to network traffic prediction have been proposed. Actually, A. Lazaris et al. [44] propose that multiple types of LSTM be used to forecast the link throughput using real world data set. In addition, W. Shihao et al. [45] choose to use LTSM model to predict network traffic volume. The former considers the characteristics of the auto-correlations in order to improve the accuracy of the model. Additionally, Azzouni et al. [43] consider the network traffic matrix to estimate the future network traffic from the previous network traffic data using LSTM models. In a study on prediction of TCP throughput, Mizana et al. [46] leverage Support Vector Regression (SVR) to predict the TCP throughput based on basic network characteristics, including available bandwidth, queuing delays, and packet loss.

3.1.5 Motivation and Main Contribution

The contributions of our work are summarized below:

- We present an in-depth analysis of the capabilities of Recurrent Neural Network (RNN) models in network traffic prediction.

- We evaluate our proposed models using real world network traffic.
- We compare the existing schemes to find their effectiveness for real-time applications. A detailed analysis of performance for three proposed models (LSTM, GRU, SVR).
- We propose to perform a Hyperparameter Tuning process to find out the optimal weights that provides accurate models. To the best of our knowledge, we are the first that tries to leverage the difference between RNN variants in network traffic prediction problem using Hyperparameter Tuning process.

3.2 Forecasting Models

3.2.1 Introduction

In this section, we present different forecasting models used to predict the volume of network traffic.

3.2.2 Support Vector Regression (SVR)

3.2.2.1 Definition

In 1997, a new regression model has been introduced by Vapnik called Support Vector Regression which represents the most common variant of SVM.

In fact, SVR attempts to minimize the generalization error bound in order to achieve good generalization performance as opposed to the traditional regression algorithms that aims to minimize the observed training error. Hence, it gives us the mobility of how much error is acceptable and find out the appropriate curve that fit the data.

More specifically, SVR is formulated as an optimization problem as shown in the equation below by defining a convex ϵ-insensitive loss function to be minimized and tune the margin ϵ in order to gain the desired accuracy of the trained model by finding the optimal tube that contains most of training samples.

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & |y_i - w_i x_i| \leq \epsilon, \forall i \in [1, N] \\ & \epsilon \geq 0 \end{aligned} \tag{3.1}$$

The following figure illustrate the optimization problem that SVR is trying to solve:

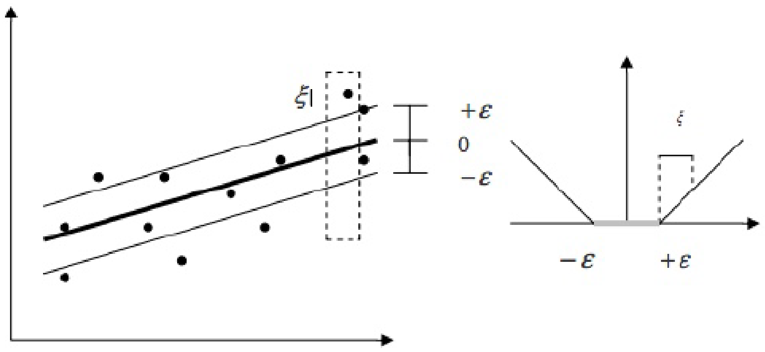


Figure 3.1: One-Dimensional Linear SVR [4]

3.2.2.2 Kernel SVR and Different Loss Functions

For nonlinear functions, where data could be mapped into higher dimensional space, standard SVR couldn't be applied due to its linearity. Thus, a *kernel function* was introduced that tries to make transformation from feature to kernel space. The problem is formulated in equation (3.2) :

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{N} \xi_i + \xi_i{}^*, \\ \text{s.t.} \quad & |y_i - w^T k(x_i)| \leq \epsilon + \xi_i, \forall i \in [1, N] \\ & \xi_i, \xi_i{}^* \geq 0 \end{aligned} \tag{3.2}$$

Threfore, The following figure illustrate the nonlinear regression problem that SVR is trying to solve:

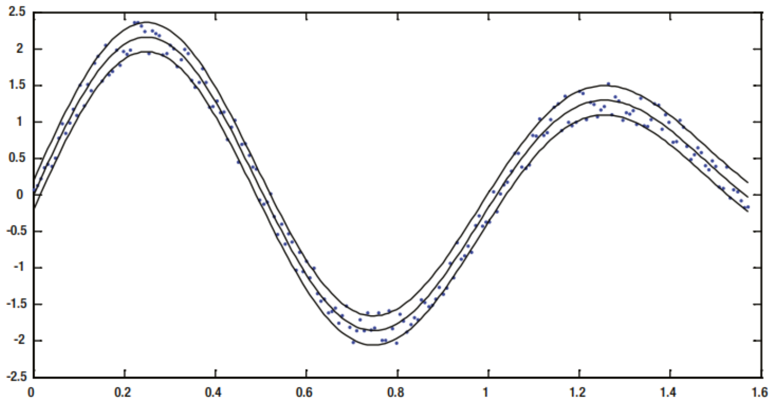


Figure 3.2: Non-Linear Regression

Source: [5]

There are several types of kernel functions such as:

- Polynomial Kernel
- Gaussian Kernel
- Gaussian Radial Basis Function
- Laplace RBF Kernel

We have chosen RBF kernel function because of its success with most of nonlinear regression problems and its wide use in the literature. In fact, the RBF kernel can be defined as follows:

$$k(x^{(i)}, x^{(j)}) = exp(\frac{||x^{(i)} - x^{(j)}||^2}{2\sigma^2}) \tag{3.3}$$

3.2.3 Long-Short Term Memory (LSTM)

LSTM is one of the most well-known variant of RNN layers. It was introduced in 1997 by S. Hochreiter et al. [40]. In fact, it contains computational blocks that control the flow of information within a cell. Computational blocks are called gates that selectively allow information to enter or quit the cell based on the acquired knowledge. As a result, if the information is not important, the input signal will be discarded from the LSTM unit. In contrast, the LSTM cell stores merely relevant information in its internal memory. Thus, LSTM extents the RNNs models with two major advantages: 1) Introduce memory information 2) Handle long sequences more better.

Looking at LSTM cell as black box in fig. 3.3, the input is a vector of combination of the input vector x_t and the previous hidden state h_{t-1}, where the output at time t is denoted as the output vector y_t, the actual hidden state h_t.

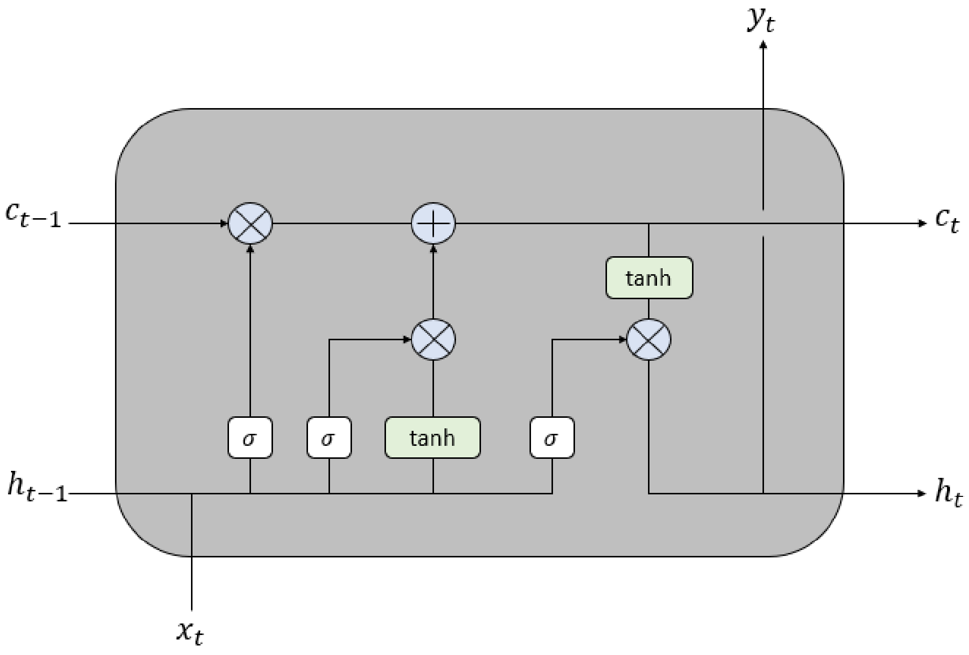


Figure 3.3: LSTM Cell

Looking at the design of LSTM cell, it consists of a memory state C_t and three gates. As mentioned Above, Input gate controls what needs to be updated in the memory state, and the Forget gate controls what needs to be forgotten. The output gate controls the part of the memory state that will be used as output. Each gate has its computational block to estimate the importance of the input information. Indeed, each block consists of a matrix to vector multiplication $\{W_i, W_f, W_o, W_c\}$ followed by a bias term $\{b_i, b_f, b_o, b_c\}$. The computational blocks that forms the LSTM neuron are as follows:

- **Input Gate:** The role of the input gate is to decide which information is to be renewed. The input gate output is calculated using the weight matrix W_i and the bias b_i. The activation function used in this block is Sigmoid.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{3.4}$$

- **Forget Gate:** The role of the forget gate is to decide which information should be discarded.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{3.5}$$

- **State Computation:** The role of this block is to update the new memory state C_t of the neuron. First, it calculate the possible combinations for the memory state:

$$\hat{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \tag{3.6}$$

  Then, the new memory state is calculated using the estimated new memory state, the previous memory state multiplied by the output vector of the forget gate f_t.

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t \tag{3.7}$$

- **Output Gate:** Finally, the output gate computes the output vector for the given input signal. First, the output gate vector is computed using Sigmoid activation function:

$$\hat{o}_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.8}$$

  Then, the output vector is calculated using element wise multiplication between the output of the output gate and the activation signal of the actual memory state:

$$\hat{y}_t = o_t \odot \tanh(Ct) \tag{3.9}$$

3.2.4 Gate Recurrent Unit (GRU)

GRU is a variant of RNN layers. It was proposed by K. Cho in 2014 [41]. It was designed to capture dependencies at different time scales in an adaptive manner. Unlike LSTM, this type of RNN neurons consists of only two gates and one memory state. Hence, it contains three computational blocks. As a result, GRU is becoming more computationally efficient and more promising for resource-constrained devices, as is our case. Fig. 3.4 illustrate in advance the architecture of the GRU neuron.

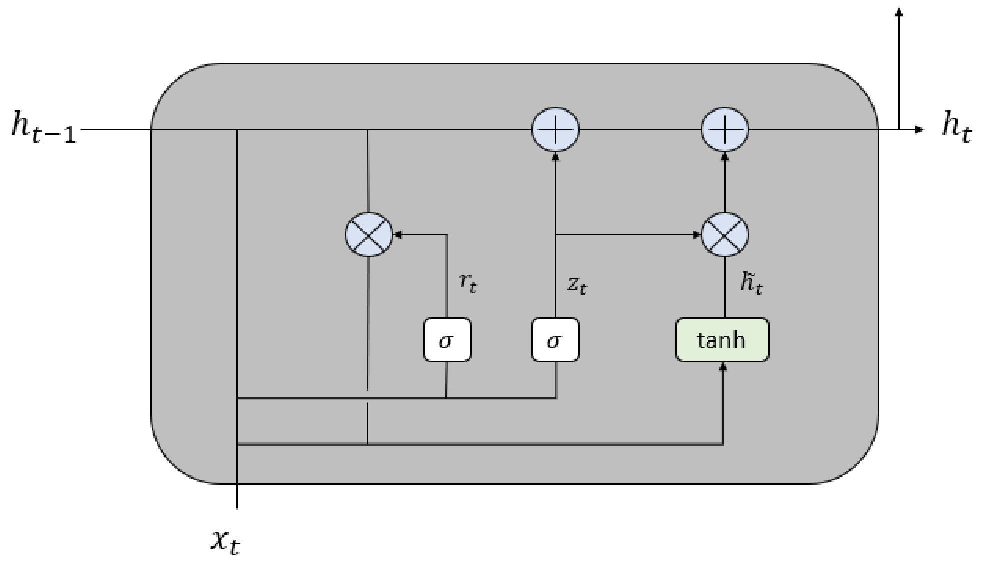


Figure 3.4: GRU Cell

The three computation blocks are as follows:

- **Reset Gate:** The reset gate is used to decide whether to use the previous output or discard the input as the first symbol in a sequence. The reset output vector r_t is calculated as follow:

$$r_t = \sigma(W_r[h_{t-1}, x_t]) \tag{3.10}$$

Where: x_t is the input vector, h_{t-1} is the hidden state output vector, W_r is the weight matrix and σ is the Sigmoid activation function used for this gate.

- **Update Gate:** The update gate determines how much of the output is updated. The output of the update gate is calculated using the weight matrix W_r:

$$z_t = \sigma(W_z[h_{t-1}, x_t]) \tag{3.11}$$

- **Output Gate:** The role of this gate is to calculate the hidden state h_t. Similarly to LSTM output gate, it computes the possible combinations of the hidden state $\hat{h}_t$:

$$\hat{h}_t = \tanh(W_o[r_t \odot h_{t-1}, x_t]) \tag{3.12}$$

Then, the new hidden state is computed based on the possible combinations:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{3.13}$$

According to [42], the number GRU operations and parameters is about 75% compared to LSTM neurons.

3.3 Experimental Methodology

In order to evaluate our forecasting models, we have used real network traces. In fact, we compare the performance of two variants of RNN with the baseline regression model SVR.

Hence, we start by presenting the software environment. Then, we turn our focus on exploratory data analysis and data preprocessing. Finally, we will discuss the modeling phase by including the performance evaluation of the fitted models.

3.3.1 Software Environment

3.3.1.1 Programming Language Python

Python has been the choice for Machine Learning and Artificial Intelligence developers and researchers for a long time. Hence, it contains the most advanced features and libraries that increase the productivity by abstracting complex algorithms and versatile workflows.

3.3.1.2 Libraries and Packages

Python comes with the support of a broad set of open source libraries:

- **Numpy:** Numpy is one of the most basic library in Python for multidimensional data manipulation.
- **Matplotlib:** Matplotlib is a library for producing visualizations in Python.
- **Scikit-learn:** Scikit-learn is a python Machine Learning library.
- **TensorFlow:** TensorFlow is a high-level library available for working with Deep Learning on Python.

3.3.2 Data Analysis

3.3.2.1 Data Collection

In order to implement a predictive model, there is always a mandatory step which is choosing the right data set for both training and evaluation. In fact, we have used a real traffic data taken from the University of Leipzig Internet access link [1]. A pair of DAG 3.2 cards were used to monitor network traffic from campus network to the German research network (G-WiN). A diagram of the configuration is illustrated in figure 3.5.

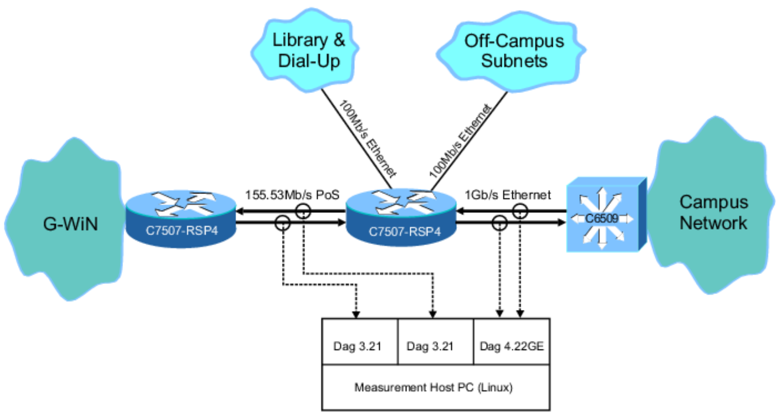


Figure 3.5: Network Configuration

Source: `https://wand.net.nz/wits/leipzig/gmwin.png`

After selecting the data source, we have chosen the traffic that is directed from the German research network to the campus network with a duration of 4 days and 18 hours. Thus, all outer connections pass through this measurement point.

In addition, all non-IP traffic has been removed, and only TCP, UDP and ICMP traffic are kept in the traces. Then, all IP addresses have been anonymised using one-to-one mapping into another address space.

3.3.2.2 Data Preprocessing

At first, we have converted the traces from compressed format into PCAP files using Libtrace tool [6] in order to make them readable . Thus, The collected raw data set consists of network packets in PCAP format. The network traffic captured 3429 million packets over 4 days and 18 hours. In fact, the storage of this traffic was about 1615 GB.

Hence, we processed the data in order to transform it into a time series that represents the length of packets at each minute. This resulted in a time series containing 6839 values as illustrated in the figure 3.6. This part of the process was done using Bash scripts in order to optimize the execution time.

Indeed, this data was sufficient to implement our network traffic prediction because we notice that it's about a periodic curve where the long term regularities are evident.

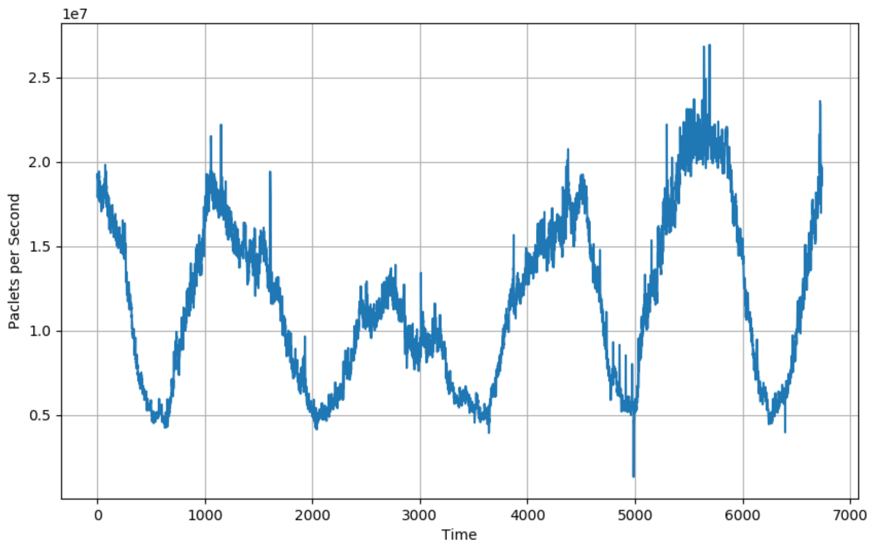


Figure 3.6: Time Series of the Network Traffic Volume for about 5 days

3.3.2.3 Data Normalization

Training of ML algorithms often performs better with standardized input data. In fact, normalizing features is not only essential for comparing features with different scales, it is a common prerequisite for many machine learning algorithms. For instance, in the absence of normalizing features for optimization algorithms, some weights may update faster than others as feature values plays a role in updating weights.

Hence, we apply the Gaussian transformation (i.e. subtracting the mean and dividing by the standard deviation) as mentioned in the equations above.

First, we need to calculate the mean:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3.14}$$

Then, we calculate the standard deviation:

$$\mu = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{3.15}$$

Finally, we calculate the normalized values as follows:

$$x_i = \frac{x_i - \mu}{\sigma} \qquad \forall i \in [1, N] \tag{3.16}$$

3.3.3 Data Transformation

Time Series prediction requires a set of past observations in order to estimate future values of the features of interest. Actually, there are mainly two categories as explained below:

- **One-Step Prediction:**

  The most straightforward strategy is one step prediction which is about taking a vector that include the last observations as input to the model responsible for yielding the prediction. Specifically, for a set of samples observed at times $t_0, t_1, ..., t_n$, the value at time t_{n+1} is estimated.

- **Multi-Step Prediction:**

  Multi-Step Prediction is about predicting upper than one step ahead based on the same historical samples. Specifically, for a set of samples observed at times $t_0, t_1, ..., t_n$, the values at time $t_{n+1}, t_{n+2}, ..., t_{n+m}$ are estimated.

  For formalization, we denote:

  - **Step Back** : The number of historical samples that will be put as input to the predictive model.
  - **Step Ahead**: The number of future samples that will be predicted.

  Indeed, Multi-Step Prediction is more challenging than One-Step Prediction due to the accumulation of errors and the lack of information needed to predict further samples in the future [7].

  In order to overcome these issues, a multiple-input multiple-output (MIMO) strategy was proposed by Bontempi [8] which consists of producing a vector of future predicted values instead of a single value. Therefore, this method uphold the stochastic dependency of the time series.

To wrap up, a sliding window concept is introduced to indicate a fixed number of previous samples to be learned in order to forecast the future time steps. Hence, for a given time slot t, the input vector contains the following values:

$$[X_{t-StepBack}, ..., X_{t-2}, X_{t-1}] \tag{3.17}$$

On the other side, the output vector contain the following values:

$$[X_t, X_{t+1}, ..., X_{t+StepAhead}] \tag{3.18}$$

3.4 Modeling: Training and Validation

In this section, we will demonstrate how each model was fitted and evaluated.

At first, the data set was divided into two parts, as indicated below, using **Test-Set Validation** technique which is the most basic strategy for estimating the reliability and the robustness of the predictive models. Indeed, we split the data into two separate portions:

- **Training Set**: It contains 80% of the original data. Hence, this portion will be used for training the predictive models.
- **Test Set**: It contains 20% of the original data. Hence, the fitted models will be validated on the Test Set.

Then, we turn our focus on the model architecture of each forecasting model.

3.4.1 Recurrent Neural Network (RNN)

We have used two variants of RNN:

- Long Short Term Memory (LSTM)
- Gate Recurrent Unit (GRU)

The proposed architecture of the two RNN variants has the same number of layers as depicted below:

- **Encoder**: The encoder consists of a single layer containing 16 (LSTM / GRU) neurons. Indeed, it is responsible for encoding the time series to the state of the network.

- **Decoder**: The decoder consists of four consecutive layers where each layer contain by default 8 (LSTM / GRU) neurons. This will decode the encoding vector in order to generate the output sequence.

In order to set the optimal weights of the chosen RNN models and minimize the error rate between the actual output and the predicted sequence, a loss function is required. In the literature, the most used loss function for regression problems is Mean Squared Error (MSE). Therefore, it could be defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \tag{3.19}$$

where

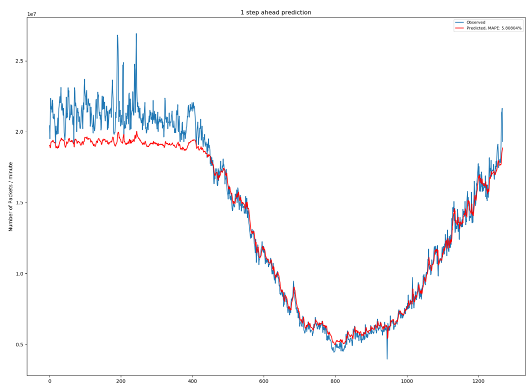
- Y_i represents the actual output.
- $\hat{Y}_i$ represents the predicted output.

We have picked out RMSProp as an optimizer for updating the NN weights in the Gradient Descent algorithm. The selection was actually based on a rigorous benchmark of three optimizers (SGD, RMSProp and AdaGrad).

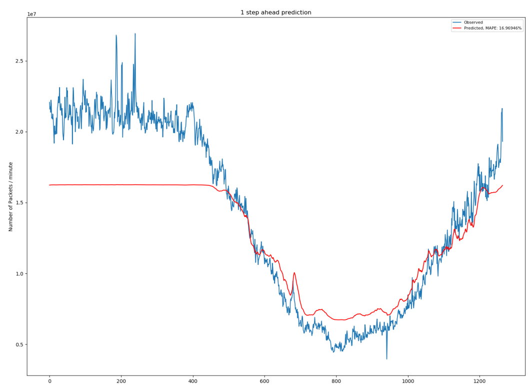
Then, The proposed models were trained on the training set with the following Hyperparameters (Randomly Chosen):

- **Time Granularity:** 1 minute.
- **Step Back**: 60
- **Step Ahead**: 20
- **Encoder**: Composed of one single (LSTM/GRU) layer that contains 16 neurons.
- **Decoder**: Composed of four (LSTM/GRU) layers that contains respectively 8, 8, 16, 16 neurons.
- **Epochs**: 1000
- **Batch Size**: 10
- **Loss Function:** Mean Squared Error (MSE)
- **Optimizer:** RMSProp

At this stage, the model is ready for training and evaluation phase.



(a) Prediction Result of GRU Model



(b) Prediction Result of LSTM model

Figure 3.7: Difference of the Test Result Between two RNN variants

For evalutation purpose, the mean absolute percentage error (MAPE) is used as the evaluation indicator of the proposed models:

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}|\frac{Y_i - \hat{Y_i}}{Y_i}| \times 100\% \tag{3.20}$$

Indeed, we have found accurate results from our primary test. The mean absolute percentage errors of the two RNN models on the test set are shown in the table 3.1 below:

	LSTM	GRU
MAPE (%)	16.96	5.8

Table 3.1: Comparison of Results Between two variants of RNN

Figures 3.7a, 3.7b show the difference between predicted values and the actual times series for both GRU and LSTM models.

Based on this primary run, it appears that GRU model outperform the LSTM model in the test subset. An interesting finding that stimulates a study of the fine tuning process in order to determine the most reliable model for network traffic prediction task.

3.4.2 Support Vector Regression (SVR)

We have chosen SVR model as the baseline for the comparison of performance between DL models and classical ML models. Hence, the same metric function was used (MAPE). As far as the Hyper Parameters is concerned, we have set the default values as illustrated below:

- **Time Granularity:** 1 minute.

- **Step Back**: 60
- **Step Ahead**: 20
- **Kernel Function**: RBF
- **C**: 1.0
- ξ: 0.01

Then, the SVR baseline model was trained with the default parameters. Hence, the predictions are made on the test subset. The MAPE was 7.53% for One-Step prediction as illustrated in the figure.

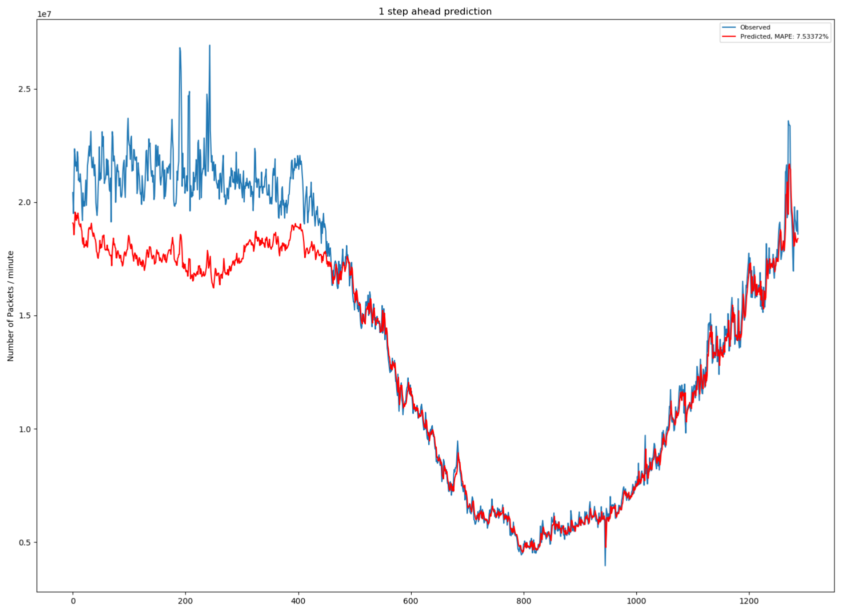


Figure 3.8: Prediction Result of SVR model

Based on the default configuration, we observe that GRU model outperforms the SVR model. Therefore, the primary results represents a blurry picture. Hence, a fine tuning process is required to pick out the most accurate model.

3.5 Fine Tuning

Throughout this section, we will try to refine the hyper parameters of the forecasting models in order to improve the accuracy. Indeed, we conducted our optimization using **Grid Search** approach. This latter performs every conceivable combination of the Hyper Parameters. As a result, it necessitate a massive computation in order to complete the task.

Hence, we define a grid of n dimensions where each dimension represents a Hyper Parameter. In our case, $n =$ (Step Back, Step Ahead, Number of Neurons in the Encoder layers, Number of Neurons in the Decoder layers).

The table 3.2 below demonstrate the range of possible values that the Hyper Parameters could receive.

	Minimum	Maximum	Step
Step Back	10	200	5
Step Ahead	20	200	5
Encoder "Layer 1" (#Neurons)	20	200	5
Decoder "Layer 1" (#Neurons)	20	200	5
Decoder "Layer 2" (#Neurons)	20	200	5
Decoder "Layer 3" (#Neurons)	20	200	5

Table 3.2: Configuration of the Hyper Parameters for Grid Search

As a consequence, we search for all the possible configurations. As we can see, with more dimensions, the more time-consuming the search will be. In our case, we have around 6500 combinations for the Hyper Parameters. This means that we have to train the predictive models 6500 times.

In order to compensate this huge computation, we have run the Fine Tuning process on the LAAS computing platform. This latter contains 228 cores and 1.6 TB of RAM spread over 7 machines. This cluster is primarily designed for distributed computing. Hence, all the tasks are executed using the resource manager SLURM [9].

Thanks to this platform, the Hyper Parameters Tuning process took about 24 hours to complete. Hence, we have found interesting insights as illustrated in the table 3.3.

Id	algorithm	look back	look ahead	neuron1	neuron2	neuron3	MAPE: 1 Step
1	GRU	40	20	50	50	50	3.06
2	GRU	60	20	40	70	70	3.08
3	GRU	30	30	100	80	80	3.12
16	LSTM	20	30	70	70	70	3.18
17	LSTM	30	30	50	90	90	3.18
4281	SVR	10	20				6.06
6491	LSTM	100	100	60	10	10	33.38
6492	LSTM	30	100	10	10	10	34.97

Table 3.3: Results of the Fine Tuning Process

In the first rows, we find that GRU model has achieved the best MAPE metric for around 3%. Then, the first LSTM model is ranked 16 with a MAPE equals to 3.18%. The baseline model SVR has achieved a MAPE equals to 6.06% as the highest result from all the possible combinations and it is ranked 4281 in the Fine Tuning process.

A closer look at the variations of the RNN models demonstrate that GRU has better performance over LSTM as illustrated in the figure 3.9 below:

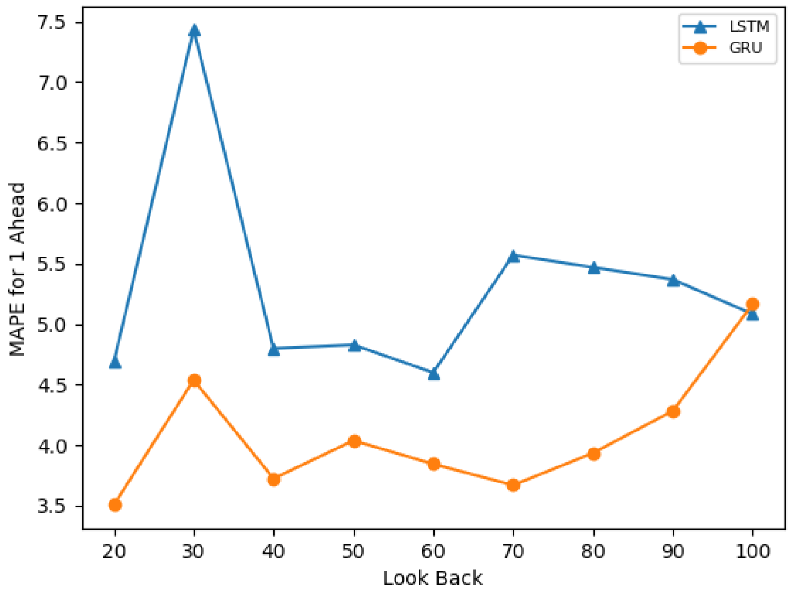


Figure 3.9: Comparison Between two variations of RNN

In conclusion, GRU has better performance than both of LSTM and SVR on the network traffic prediction problem. This result couldn't be observed unless the Fine Tuning process. Hence, it shows the importance of this phase.

3.6 Conclusion

In this work, we reviewed the current state of the art for DL on the network traffic prediction. We have shown that GRU is well suited for network traffic prediction. We have proposed a data preprocessing, RNN feeding technique that achieves high predictions within few milliseconds of computation. The fine tuning process has shown that GRU outperforms classical ML algorithm and the LSTM model with several orders of magintude.

For future work, network traffic prediction leads to many interesting applications such as QoS management, resource allocation, and anomaly detection. These applications serve better quality of Internet service and enhance the actual network infrastructure. Therefore, we believe that network traffic prediction represents a major key to provide minimum latency for real-time applications (e.g. augmented reality, autonomous vehicles).

Chapter 4

Distributed DL Inference in EC scenario

Introduction

We start this chapter by presenting the general scope of the project. Then, we turn our focus on the deployment of DL task at the edge. Finally, we tackle the modelling phase by including the performance evaluation of the chosen models.

4.1 General Description

4.1.1 Introduction

Edge Computing has become an important solution to break the bottleneck of emerging technologies by virtue of its advantages in reducing data transmission, improving service latency and easing cloud computing pressure. The edge computing architecture will become an important complement to the cloud, even replacement the role of the cloud in some scenarios.

Indeed, the architecture of Edge Computing could be organized as follows:

- **End Device:** refers to mobile edge devices (e.g. smartphones, smart vehicles, etc) and IoT devices (e.g. sensors, cameras, accelerators, etc) .
- **Edge Device:** refers to the edge servers, MEC servers, and generally servers deployed at the edge of the network.
- **Cloud:** represents the on-demand availability of computer system resources, especially data storage and computing power.

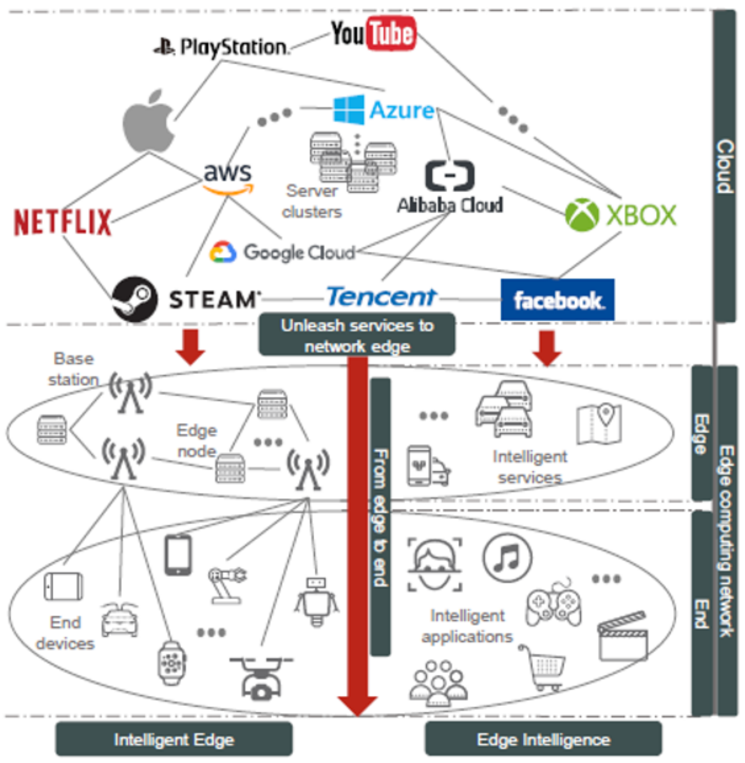


Figure 4.1: Edge Computing Hierarchy [27]

While cloud computing is created for processing computation-intensive tasks, such as DL, it cannot guarantee the delay requirements throughout the entire process from data generation to transmission to execution. On the other side, independent processing on end or edge devices is limited by their computing capability. Therefore, collaborative end-edge-cloud computing for DL is the most suitable solution to avoid latency while satisfying the required precision.

4.1.2 Problem Statement

Deep neural networks are the state of the art methods for many learning tasks thanks to their potential to derive better features at each network layer. However, the increased efficiency of additional layers in a deep neural network comes at the cost of additional latency and energy consumption in feed forward inference. Thus, it becomes more challenging to deploy them in the edge with limited resources.

Therefore, large-scale DL models are generally deployed in the cloud while end devices merely send input data to the cloud and then wait for the DL inference results. Specifically, it cannot guarantee the delay requirement for real-time services such as real-time object recognition with strict demands.

To address these issues, DL applications tend to resort to edge computing. In fact, the use of optimization techniques, distributed DNNs and collaborative inference between IoT devices and the cloud becomes a promising solution.

4.1.3 Related Work

End devices (e.g. mobile phones, sensors) were unable to support the computational costs of DL inference due to their limited resources. To address these issues, a lot of research is underway to optimize the inference performance on IoT devices [26][27] [28] . The first common approach is the optimization of DL models at the edge. For instance, network compression is one of the optimization directions that aims to compress deep networks into tiny models that achieves a slightly reduced level of accuracy and alleviate considerably the amount of computation and the inference time [18]. In addition, industrial companies are trying to produce custom edge hardware that speed up the inference of DL models. Indeed, Coral [20], a company supported by both Google and TensorFlow, provides custom hardware components devoted for resource-constrained devices that allow fast ML inference. A second branch of study focuses on segmenting DL models by pushing portion of the computation from cloud to the edge. Based on the network bandwidth, the energy state of IoT devices, and the delay requirements of running applications, the system will decide how much of the computation will be processed on the cloud server. Actually, it could be classified mainly into two categories 1) Model parallelism 2) Data parallelism. The former tries to distribute the computation among edge nodes by slicing the DL model into partitions. In fact, Neurosurgeon [21] intelligently decides where to split the DNN based on preliminary constraints (e.g. network bandwidth, device energy state, delay requirements). In addition, IONN [23] propose an incremental offloading system which uploads the DL model to the edge server sequentially. It allows the IoT device to offload partial DNN execution even before the uploading finishes. Furthermore, S. Teerapittayanon et al. [24] propose to multiply the output exits by adding secondary branches. The latter could achieve primarily inference result with slightly lower performance so that they could gain in delay response and energy consumption. In [25], there is another line of work focuses on splitting the input data and dividing CNN layers into independent tasks for distribution across edge devices in order to unburden the computation.

4.1.4 Motivation and Main Contribution

The main contributions of this work is represented as follows:

- Propose an EC scenario for Early Exit of Inference (EEoI) Model.
- Extensive Simulations using virtualization technique show that our distributed framework achieves a better trade-off between execution delay and model performance.

4.2 Proposed Work

4.2.1 Model Architecture

In order to reduce the computation time of DL inference, we add early exits branches at different stages of the network. Hence, this latter enable the inference to exit early from these additional branches based on a confidence criteria. For instance, as depicted in figure 4.2, an edge device could give primary inference results at an early stage if the confidence criteria is satisfied. Otherwise, further computation should take place on the cloud or on the edge server.

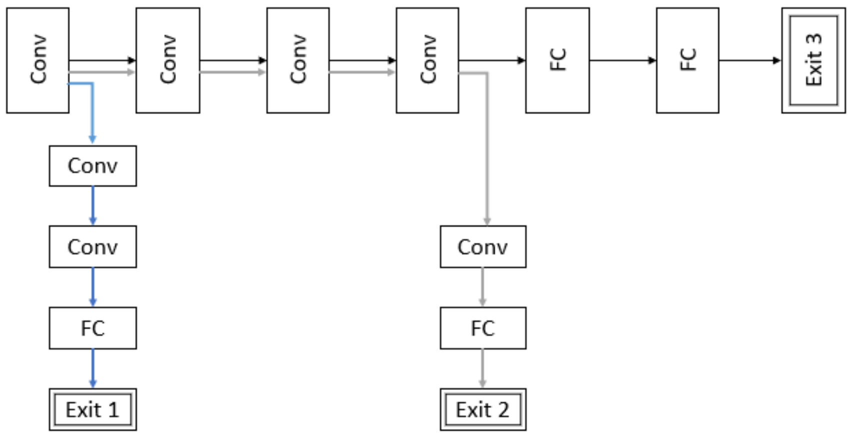


Figure 4.2: Example of Early Exit Mechanism for DNN

Actually, EEoI mechanism has been proposed by Branchynet [24]. In fact, this latter is composed of one single input, and several exit points. The last output could be considered as the principal network output from which intermediate outputs are added.

4.2.2 Training Phase

This sort of models is barely considered as a neural network models where both Forward Propagation and Backward Propagation algorithms are still applicable.

For a classification problem, one of the most used loss function is Softmax cross entropy. In fact, it could be expressed as the following:

$$L(\hat{y}, y, \theta) = -\frac{1}{|C|} \sum_{c \in C} y_c \log \hat{y}_c \tag{4.1}$$

where:

- x: represents the input data.
- y: represents the actual output.
- $\hat{y}$: represents the predicted output.
- θ: represents the models parameters including the multiple exit points.
- C: represents the set of classes for the classification problem

The expression of the predicted output for a given ext point n is written as follows:

$$\hat{y}_{exit_n} = Softmax(z_{exit_n}) = \frac{exp(z_{exit_n})}{\sum_{c \in C} exp(z_c)} \tag{4.2}$$

where: $z_{exit_n} = f_{exit_n}(x;\theta)$

As a result, each exit point has its corresponding loss function. In order to evaluate the entire EEoI model, we form a joint optimization problem as weighted sum of the loss functions of each exit point as demonstrated below:

$$L_{Branchynet}(\hat{y}, y, \theta) = \sum_{n=1}^{N} w_n L(\hat{y}_{exit_n}, y, \theta) \tag{4.3}$$

The training algorithm is composed of:

- **Forward Propagation:** The data will flow from input layer into the multiple branches in order to calculate the loss function written in Eq. 4.3.
- **Backward Propagation:** The back-propagation algorithm computes the partial derivatives of the loss function with respect to all the weights in the EEoI model (including the intermediate exit points) and then go back through the whole network in order to adjust the model parameters using gradient descent.

4.2.3 Fast Inference using EEoI

The proposed model is ready for making predictions the moment we make the necessary steps of modeling and training. Therefore, the classifier starts by making predictions from the earliest exit point until reaching the main output layer. Once the model predict an output that has less confidence than a given threshold, the former will pass to the next exit point. This results in more computation and longer runtime.

The policy which measures the degree of confidence is the entropy. This latter is actually a measurement of the disorder in a given system. As a result, if the value of the entropy for a specific exit point is much higher than a threshold, then this output becomes uncertain and we pass the verdict to the next exit point for further processing.

Formally, the formula of the entropy for a given exit point is defined as follows:

$$entropy(y) = \sum_{c \in C} y_c \log y_c \tag{4.4}$$

where:

- y: represents the output vector.
- C: represents the set of classes for the classification problem.

Hence, the algorithm that runs inference for EEoI models could be formalized as follows:

Algorithm 1: Fast Inference of EEoI (x, T)

```
Result: Predicted Label for the Given Input $x$
initialization;
for $i$ = 1...$N$ do
    $z_n = f_{exit_n}(x)$ ;
    $\hat{y}_n = softmax(z_n)$ ;
    $e_n \leftarrow entropy(\hat{y}_n)$ ;
    if $e_n \leq T$ then
        return $\arg\max \hat{y}_n$ ;
    end
    return $\arg\max \hat{y}_n$ ;
end
```

As a result, the prediction starts from the earliest exit point to the main branch of the EEoI. Indeed, the algorithm is trying to minimize the computation process which is directly tied to the execution time and the energy consumption by identifying the earliest exit point that satisfies the condition of confidence.

4.2.4 Edge Computing Scenario

The proposed architecture of EEoI could be implemented in an Edge Computing scenario as depicted in 4.3. For instance, we assigned the exit points as follows:

- **Exit 1:** The first branch contains the lowest number of layers. Therefore, it has the minimum computation i.e. minimum latency and efficient energy consumption. As a result, this configuration is well suited for Edge Devices. Hence, we assign the 1st **exit point** to the **Edge Device** for primary decisions.

- **Exit 2:** The second branch has the medium amount of computation because it's located in the middle i.e. it has the average number of layers. Hence, we assigned this exit point to the edge server. Indeed, this connection is well adapted because the ES has an average of computation resources.
- **Exit 3:** The main branch which has the maximum number of layers, automatically needs the most computation resources because of the massive processing it produces. Hence, we assign this exit point to the **Cloud**.

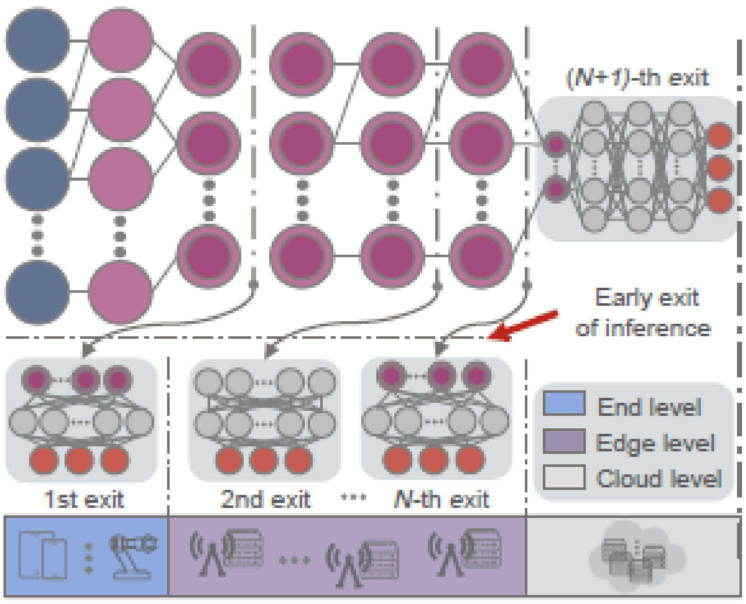


Figure 4.3: EEoI in an EC scenario [27]

4.2.5 Distributed Workflow

During inference, the predictions will be made by the earlier exits until the main branch (last exit point) is reached. As a result, the forecast will be returned by the last exit in the worst case. To deal with the communication between different nodes, we employ a distributed workflow to take charge of the message activity and the synchronization as depicted in figure 4.4.

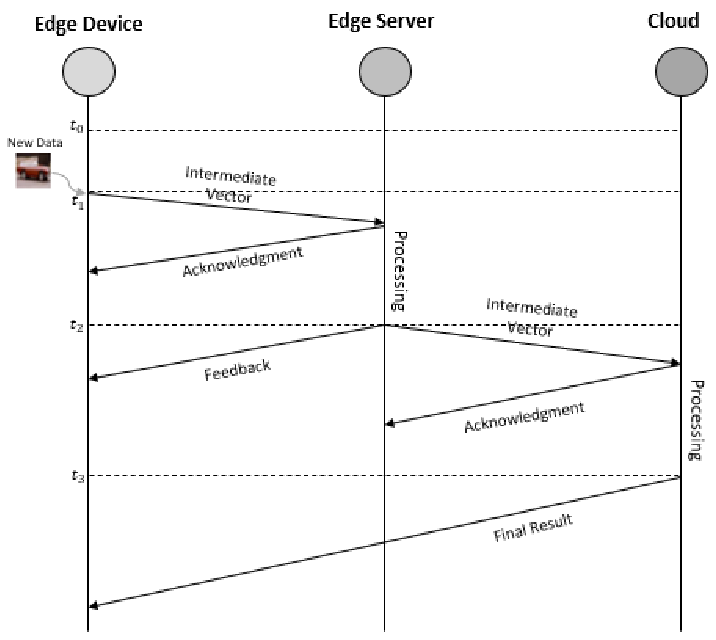


Figure 4.4: Message Activity at the Edge

4.3 Simulations and Results

4.3.1 Modeling: EEoI Architecture

To manifest the performance of the proposed work, we have conducted several simulations using the standard CNN model "ResNet 20" [19] on CIFAR10 dataset [14] for object recognition problem.

As the name is indicated, the ResNet20 model consists of 20 convolutional layers. Indeed, the location of the exit points in the proposed model is not yet automated. Hence, we should make several experiments as shown in table 4.1 in order to find out the best positions for the secondary branches. One interesting finding is that the locations of the exit points influence on the model accuracy.

	Exit 1	Exit 2	Exit 3
ResNet20-A	2	7	20
ResNet20-B	7	13	20

Table 4.1: Different Combinations for the location of ResNet Exits

As far as data is concerned, CIFAR10 dataset consists of 60000 labeled images for 10 classes.

The former is composed of 50000 samples for training and 10000 samples for test. Explicitly, the classes are:

- Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck.

The input of EEoI model is a color image with a shape of: $(32, 32, 3)$.

The output of EEoI model is an embedding vector that contains the probabilities for each class with a size of: $(1, 10)$.

4.3.2 Training and Evaluation of the EEoI model

Training the proposed models is carried out by Python Programming Language and TensorFlow 2.3 library. The hyper-parameters that defines the training phase of the proposed models are presented as follows:

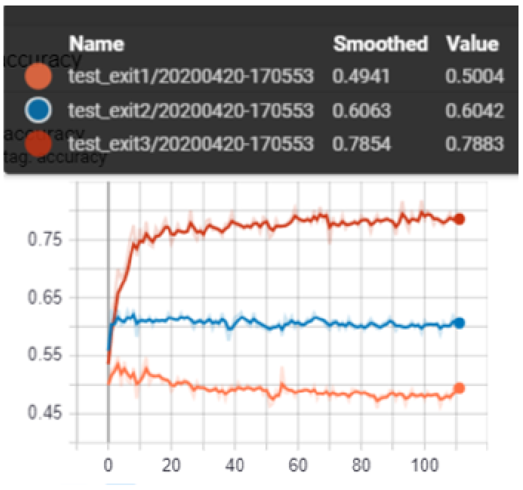
- **Epochs :** 110
- **Batch Size :** 32
- **Loss Function :** Mean Squared Error (MSE)
- **Optimizer:** Adam
- **Loss Function :** Categorical Crossentropy
- **Metric :** Accuracy

The training process was done on Google Colab platform using free access GPU card "Tesla T4". Indeed, this latter performs inference of ResNet50 model 27 times more faster than CPU [11]. As a result, the training of each DL model took about 1 hour and 18 minutes which is considerably fast.

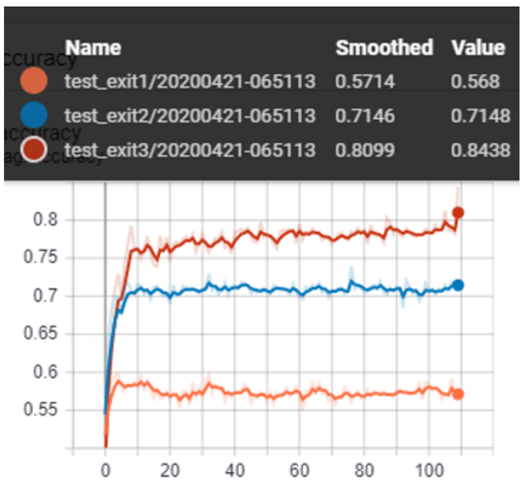
Results have shown that the deeper we make the secondary branches, the better the accuracy of the entire EEoI model. In fact, the table 4.2 illustrate the results of the training phase of the proposed models:

	Exit 1	Exit 2	Exit 3
Accuracy of ResNet20-A (%)	50.04	60.42	78.83
Accuracy of ResNet20-B (%)	56.8	71.48	84.38

Table 4.2: Performance Results of the proposed models



(a) ResNet20-A [12]



(b) ResNet20-B [13]

Figure 4.5: Curve of the Accuracy on the Test Set

Figures 4.5a, 4.5b depicts the evolution of the accuracy on the test set. Indeed, we apparently notice the differentiation between the exit points in terms of accuracy during the training process. Hence, we deduce that the more layers we add , the better performance we achieve. In addition, we notice that the emplacement of the branches in the EEoI model affects the total performance. For instance, ResNet20-B achieved a better accuracy than ResNet20-A at the three exit points. Hence, there is a deeper connection between the location of the exit points and the model performance which must be given more attention by studying its behaviour. In contrast, we should always take into account of the trade-off between model performance and the latency requirements of real-time applications in an Edge Computing environment.

4.3.3 Distributed Inference Implementation

In order to implement the EC scenario of the EEoI model as illustrated in section 4.2.4, we need to implement the three-tier architecture i.e. cloud servers, edge servers and edge devices. Hence, due the COVID-19 Coronavirus Pandemic, we decide to make a virtualization technique instead of a real testbed by providing a set of virtual nodes built on the top of the workstation that make up the three-tier architecture.

In fact, we have implemented the scenario using "Docker" open source platform [15] to containerize the applications on virtual nodes. Thanks to this latter, we could create containers that process application packaging to deliver edge computing applications to the network. Indeed, a container is a running process that embodies additional functionalities to keep it separated from the host and the rest of containers. In addition, Docker provide ways to control the runtime options i.e. Memory, CPU. As a result of that, we have built three containers that represents the EC environment and we have set the runtime options as shown in table 4.3 below:

	Edge Device	Edge Server	Cloud Server
Memory (GB)	1	32	32
CPU (%)	100	100	100

Table 4.3: Runtime Options of the Three-Tier Architecture

As far as the host machine is concerned, it is about a laptop that is equipped with 32 GB of RAM and with Intel Processor 3.4 GHz, and runs on Ubuntu 18.04. So that to limit the resources for the edge device known to be a resource-constrained device (e.g. sensors, cameras, etc.), we have only varied the RAM memory from 512 MB to 3 GB in order to perceive the variation in latency of the applications. In contrast, the docker containers could consume from the host machine as much as they could in terms of CPU cycles. In fact, the variation of CPU cycles for Docker containers induce an abnormal behaviour. This is due to intervention of the operating system which consume likewise from the CPU. As a consequence, the virtualization becomes inaccurate.

The last brick in this implementation is the communication part between the nodes. Indeed, Docker provides a powerful feature that enables the connection between different containers using a specific network. In our case, we have used the bridge network provided by Docker as depicted in figure 4.6. The latter uses a software bridge network, which is a link layer that dispatch traffic between network segments, in order to enable communication between different Docker containers. Thus, we have used ZeroMQ [17] to pass messages and synchronizations between different Docker containers using the bridge network channels. In fact, ZeroMQ is an asynchronous messaging library that focuses on developing distributed and concurrent applications. Furthermore, we have used TCP transport layer for data transmission.

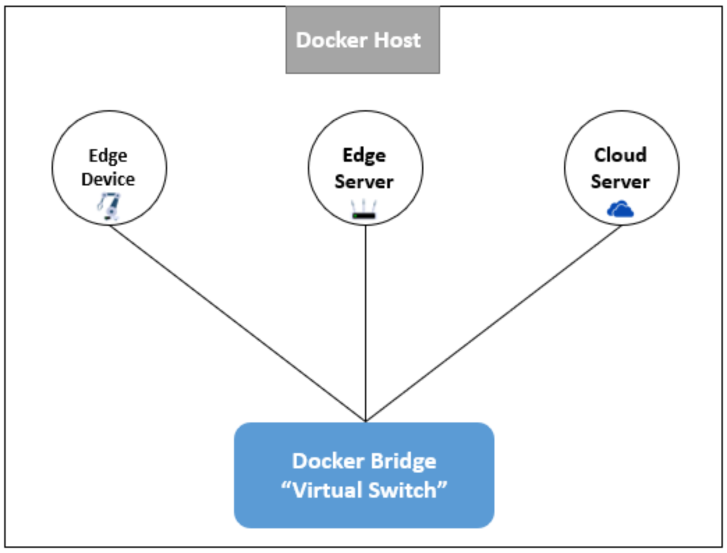


Figure 4.6: EC Virtualization using Docker Platform

4.3.4 Performance Analysis of the distributed Inference

We deploy the distributed inference of ResNet20-B model using Docker to evaluate the performance of EEoI model. Since the entropy threshold T directly affects on the model performance, as well as on the exit point that will perform the inference, we evaluate the performance of ResNet20-B under different threshold values.

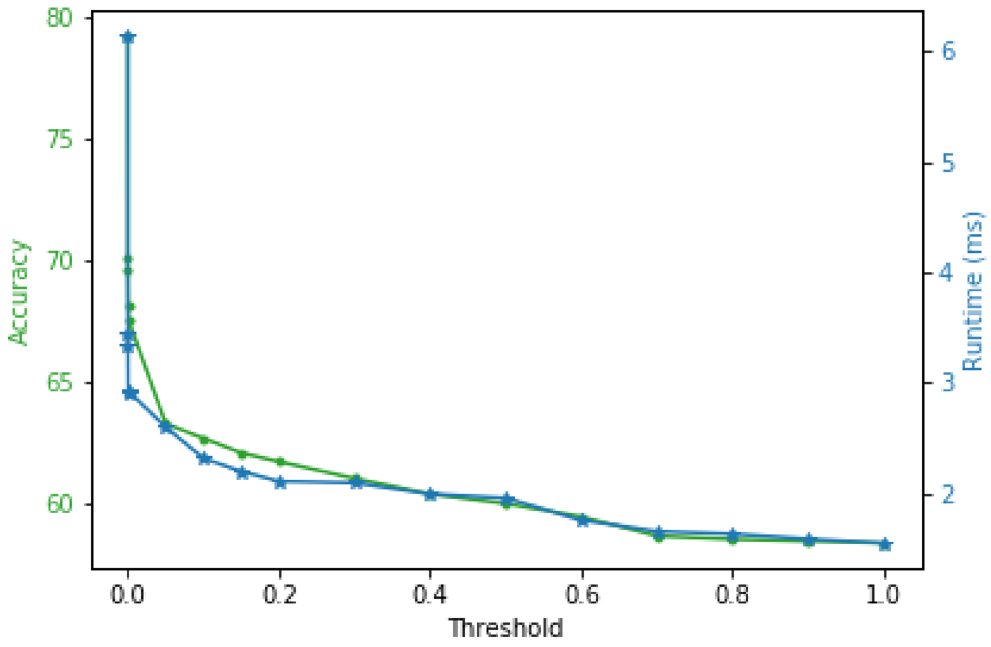


Figure 4.7: Performance Results for Distributed Inference using ResNet20-B model

Fig. 4.7 depicts the influence of the entropy threshold on both the accuracy (i.e. model performance) and the inference runtime. In fact, we can see that both of the accuracy and the runtime become lower as the entropy threshold increases. Meaning that the higher threshold leads to a decreasing in the inference time while the accuracy is deteriorating. Fig. 4.8 shows much further the reduction of the accuracy while the entropy threshold rises. In fact, we could understand the degradation of the model performance by contemplating on Fig. 4.9. The explanation for the decrease in accuracy is that the samples are getting less and less offloaded to both the edge server and the cloud server. Therefore, the prediction becomes made by the first exit point allocated to the edge device, where we have the minimum number of layers i.e. the minimum computation, which leads to a minimum latency and a minimum accuracy by itself.

For instance, if we set the entropy threshold to its maximum value ($T = 1$), we get the minimum values for both the accuracy and the inference time. Hence, we have improved the latency of the real-time application by diminishing the processing at the cost of the model performance. In contrast, if we set the entropy threshold to its minimum ($T = 10^{-15}$), we get the maximum values for both the accuracy and the inference time. As a result, we have improved the model performance by augmenting the processing at the cost of the inference time.

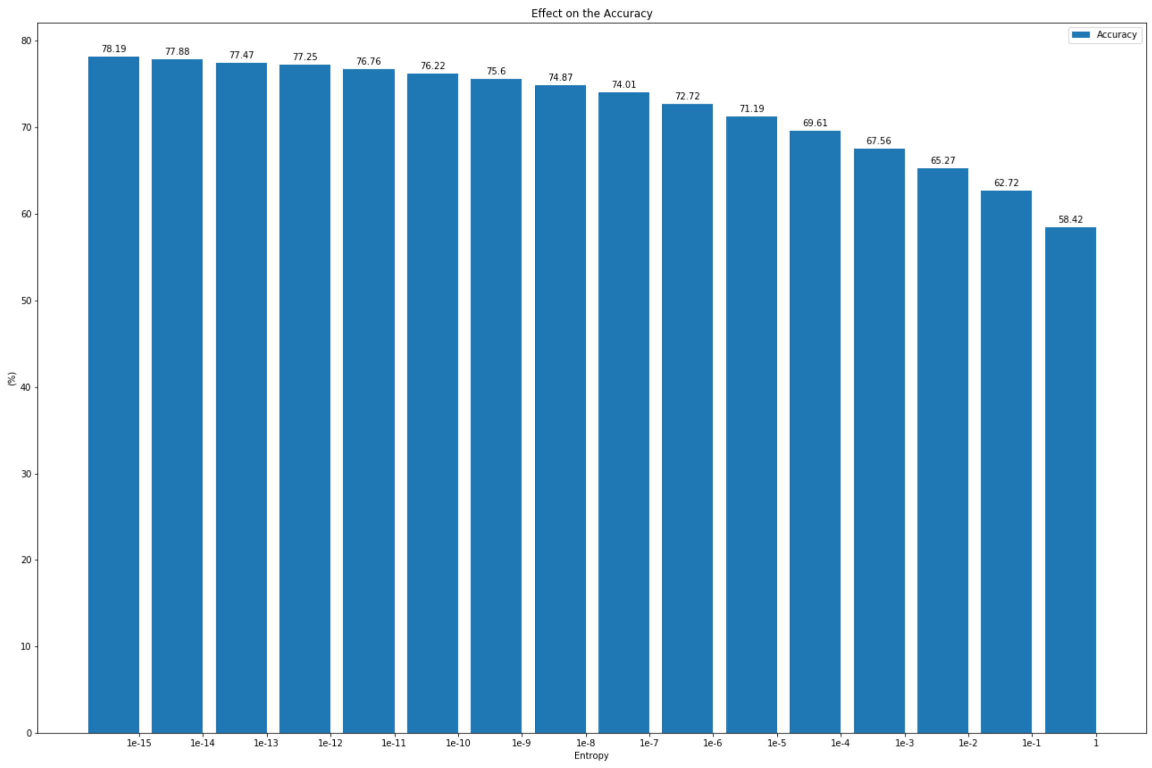


Figure 4.8: Effect of The Threshold on the Accuracy

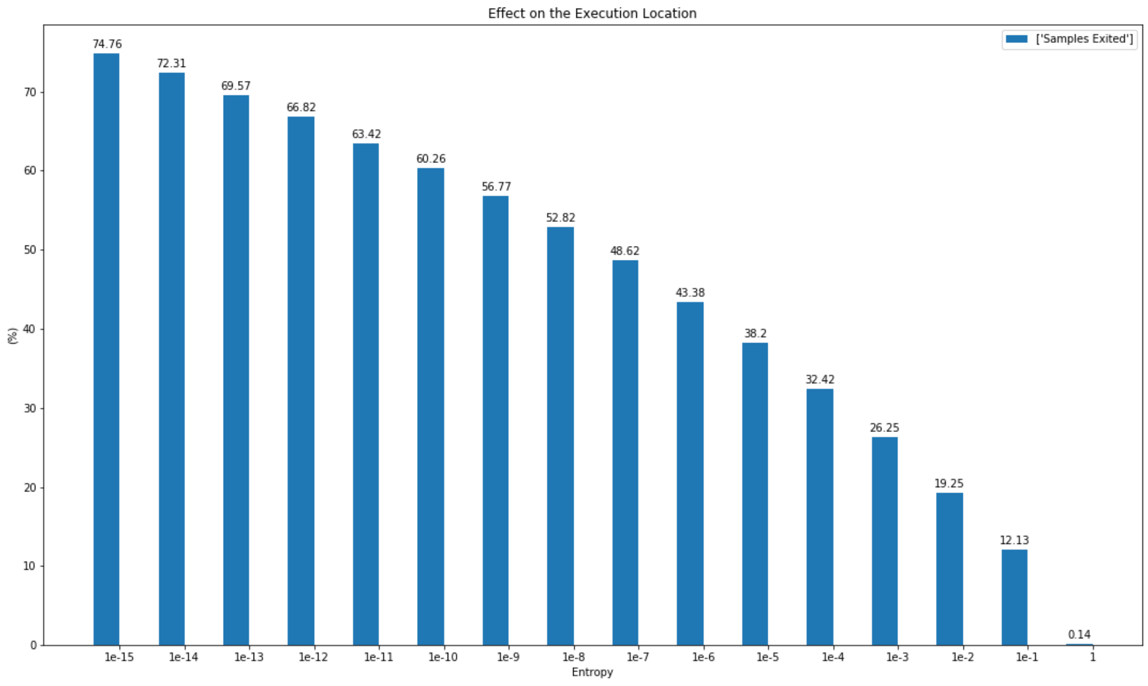


Figure 4.9: Effect of the The Threshold on the Exited Samples

Furthermore, we have varied the RAM memory for edge device from 512MB to 3GB in order to observe the impact of constraining the resources on the inference time. Indeed, Fig. 4.10 illustrates in advance the increase of the execution time while we are limiting the resources for the edge device. Thus, we should be aware of the consequences on the latency requirement for real-time applications when dealing with resource-constrained devices known as IoT devices. In fact, the modeling of the EEoI model, such as the selection of optimized Hyper-Parameters using automated algorithms, becomes the most important element in order to obtain the well-adapted distributed inference model for real-time applications (e.g. AR applications).

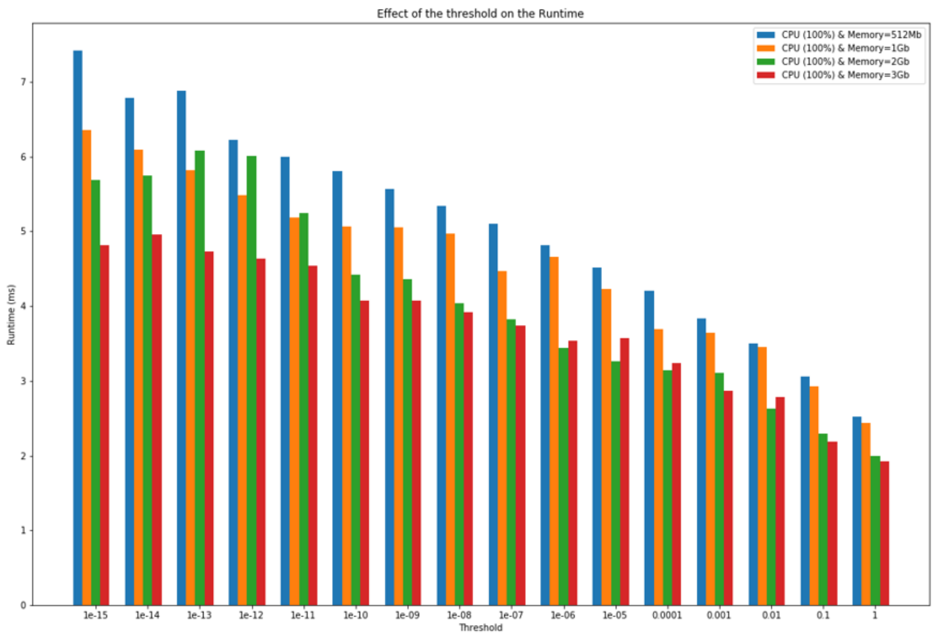


Figure 4.10: Effect of The Resource Limitation on the Model Runtime Under Different Threshold

4.4 Conclusion

For an EC environment, we present a distributed inference of DL models. Preliminary implementation and evaluations shows that the EEoI framework manages to control the trade-off between model performance and execution latency by varying the entropy threshold. Interesting findings demonstrate that the more constrained-resource device we have, the more we should be sensible about the DL model size, as well as the location of secondary exit points.

Future works reveals an implementation of real testbed in order to observe the influence of varying the network bandwidth between edge devices and the cloud on the inference latency.

Chapter 5

Computation Offloading for multi-user Mobile EC

5.1 General Description

5.1.1 Problem Statement

The rapid growth of mobile internet services has led to a wide variety of computation intensive applications (e.g. augmented reality, object recognition). Indeed, there are sensible applications that require a short latency for the execution. On the other hand, end devices such as mobile users or IoT devices have limited energy resources. As a result, they cannot handle this enormous amount of processing locally. Thus, Edge Computing attempts to resolve these issues of delay requirements and energy consumption by bringing the cloud capabilities closer to the end user by deploying edge servers. It is in this line of research, that we aim to implement an efficient computation offloading and resource allocation for mobile edge computing.

5.1.2 Related Work

Sensible decisions should be made whether offload the task to a near edge server or perform the computation locally. Hence, several works have been done in this branch of research. For instance, L. Huang et al. [50] propose a system that optimizes the offloading decisions and the resource allocation in an edge computing environment. In fact, they optimize offloading decisions using Deep Reinforcement Learning (DRL) while adopting convex optimization for the allocation of network resources. Furthermore, X. Feng et al. [51] presents a joint task offloading and resource allocation optimization method in EC environment using DQN. The overall offloading cost is evaluated based on energy consumption of end devices and the delay of tasks execution. In addition, X. Liu et al. [52] propose to leverage a distributed Q-learning algorithm by allowing each end device to control the offloading decision on its own. Based on the channel state and the remaining resources, the former tries to optimize the energy consumption and the execution delay of the remaining tasks.

5.1.3 Motivation and Main Contribution

In the previous works, the implementation of a real testbed or virtual simulation has rarely been considered for offloading decision and resource allocation optimization in EC environment. Recently, P. Gawłowicz et al. [49] is presenting a new framework ns3-gym for RL research in networking field by interconnecting the network simulator ns-3 and the RL toolkit "OpenAI Gym" together. Thus, ns3-gym becomes the benchmark for research related to applications of RL in networking. For that occasion, we propose to leverage offloading task optimization using DRL paradigm by exploiting the new framework ns3-gym.

The major contributions of this work are:

- Proposing a joint optimization problem for offloading decision and resource allocation of edge devices in a wireless EC environment.
- Solving the optimization problem using Deep Q-Network (DQN).
- Simulation of the EC scenario using ns-3 network simulator. In fact, the RL agent interacts in real-time with ns-3 environment using the framework ns3-gym.

5.2 Scenario Description

5.2.1 System Model

As illustrated in fig. 5.1, the scenario consists of an IoT network with N end devices connected to an Access Point (AP) via wireless communications. The AP aggregates the data from end devices and processes it using its equipped edge server. Each end device has M independent tasks to be completed. As a result, offloading their tasks could improve the computation experience in terms of task execution latency and energy consumption.

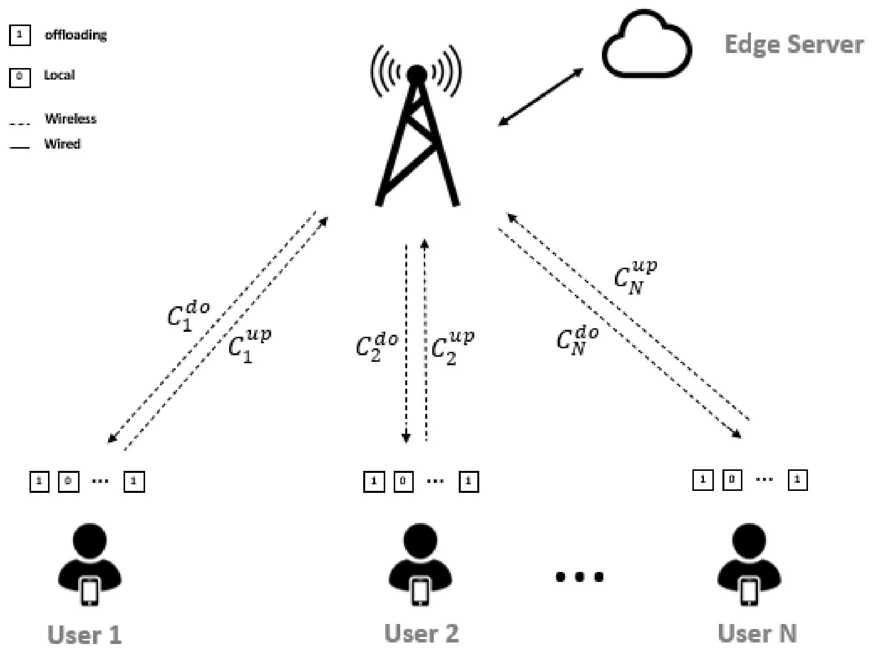


Figure 5.1: System Model

To model the task, we use S_{nm}^{in} to denote the input data size of the m^{th} task for the n^{th} user. Likewise, we denote S_{nm}^{out} as the output data size of the m^{th} task for the n^{th} user. Furthermore, each end device n could decide whether to offload or execute locally a particular task m. As a result, we denote by x_{nm} the binary decision for offloading or local execution. For instance, the offloading occurs when x_{nm} is set to 1. In contrast, the task will be executed locally if x_{nm} has been set to 0.

When end devices offload their tasks to the edge server. Energy consumption includes two parts for both transmission and receiving. Hence, we denote by E_{nm}^{t} and E_{nm}^{r} respectively, the energy consumption for both transmitting and receiving. The total energy consumed for end device n when offloading the task m is denoted by E_{nm}^{off}. On the other side, E_{nm}^{loc} represents the energy consumed by the end device n while performing the task m locally.

As far as the delay is concerned, we denote by C_n^{up} and C_n^{do} respectively, the allocated bandwidth for user n for transmission and receiving the task m. As a result, the uplink transmission delay for offloading could be expressed as follows $T_{nm}^{up} = \frac{S_{nm}^{in}}{C_n^{up}}$. Contrariwise, the downlink transmission delay for local execution could be expressed as follows: $T_{nm}^{do} = \frac{S_{nm}^{out}}{C_n^{do}}$. In addition, the computation latency at the edge server is denoted by T_{nm}^{edge}. As a result, the total delay for offloading the task m for the user n could be formalized as follow: $T_{nm}^{off} = x_{nm}(T_{nm}^{edge} + T_{nm}^{up} + T_{nm}^{do})$. In contrast, we denote by T_{nm}^{loc} the computation delay for executing the task n for the user m.

5.2.2 Optimization Problem

The problem is formulated with the aim of minimizing both the total delay and the energy consumption of end devices in order to complete the execution of the tasks, by jointly optimizing the offloading decisions for each user n and the allocated bandwidth for transmission and reception.

In this work, we propose to design an optimal task offloading scheme to minimize the task execution latency and the energy consumption of end devices. Hence, the optimization problem is formulated as follow:

$$\begin{aligned}
\min_{w} \quad & \sum_{n=1}^{N}[\sum_{m=1}^{M}[E_{nm}^{loc}(1-x_{nm})+E_{nm}^{off}x_{nm}+w_n(x_{nm}T_{nm}^{off}+(1-x_{nm})T_{nm}^{loc})]] \\
\text{s.t.} \quad & \sum_{n=1}^{N} C_n^{up} \leq C^{up,max} \\
& \sum_{n=1}^{N} C_n^{do} \leq C^{do,max} \\
& C_n^{up}, C_n^{do} \geq 0, \forall n \\
& x_{nm} \in \{0,1\}
\end{aligned} \tag{5.1}$$

The parameter w_n represents the weight between energy consumption and the execution delay. In addition, the constraints of the optimization problem describe the limitations of both the binary decision variables and the allocated bandwidths for transmission and reception.

In fact, we notice that the proposed optimization problem consists of a mixed non linear programming problem, which is typically an NP-hard problem.

5.3 RL Approach

5.3.1 Motivation

The aim of this work is to optimize the joint task offloading 5.1. Since the DRL have shown high efficiency in solving complex problems, it would be interesting to solve the optimization problem with DRL especially when DRL algorithms provides a promising solution to handle complicated decision-making process. Specifically, we have inspired by the success of modelling an optimal action-state Q-function with Deep Neural Network (DNN). As a result, we adopt a Deep Q-Network (DQN) algorithm to address this optimization problem.

5.3.2 DQN Specifications

In order to implement DQN to solve our optimization problem 5.1, we need to specify the algorithm components i.e. the action signal, the state, and the reward function. Hence, we describe these components in the following subsections:

5.3.2.1 State

To interpret our optimization problem as a DQN problem, we define the system state at time epoch t as follow:

$$s(t) = \{x_{11}(t), x_{12}(t), ..., x_{nm}(t), ..., x_{NM}(t), C_1^{up}, ..., C_N^{up}(t), C_1^{do}(t), ..., C_N^{do}(t)\} \tag{5.2}$$

In fact, the state encapsulates the offloading decisions and the allocated bandwidth for transmission and reception for all the users. As a result, the system state consists of $(NM + 2N)$ elements.

5.3.2.2 Action

The action allows the agent to interact with the environment by selecting decisions at each time epoch. Thus, we define the action as an index selection that induces a variation between two different states. We use a variable ν to denote the selection of index where $\nu = 1, 2, ..., NM + 4N$. As a result, we denote by $a(t)$ the action at time epoch t, and it could formalized as follow:

$$a(t) = \{a_\nu(t)\} = \{a_1(t), a_2(t), ..., a_{NM+4N}(t)\} \tag{5.3}$$

In fact, we took the corresponding decision based on the element index in the action vector. Hence, we have the following cases regarding each element of the action:

- **Case 1:** if $\nu \in [1, NM]$, then the corresponding variation will be on the offloading decision variable $x_{nm}(t)$. More formally, we have the following treatment:

  ```
  $n = \frac{\nu}{N}$
  $m = \nu \mod M$
  if $a_\nu(t) = 1$ then
      $x_{nm}(t) = 1 - x_{nm}(t)$
  else
      $x_{nm}(t) = x_{nm}(t)$
  end if
  ```

- **Case 2:** if $\nu \in [NM + 1, NM + N]$, then the corresponding variation will be an increase in the bandwidth allocated for transmission $C_n^{up}(t)$. Formally, we have the following process:

```
$n = \nu - NM$
**if** $a_\nu(t) = 1$ **then**
    $C_n^{up}(t) = C_n^{up}(t) + \delta C^{up}$
**else**
    $C_n^{up}(t) = C_n^{up}(t)$
**end if**
```

- **Case 3:** if $\nu \in [NM + N + 1, NM + 2N]$, then the corresponding variation will be a decrease on the allocated bandwidth for transmission $C_n^{up}(t)$. Formally, we have the following process:

```
$n = \nu - NM - N$
**if** $a_\nu(t) = 1$ **then**
    $C_n^{up}(t) = C_n^{up}(t) - \delta C^{up}$
**else**
    $C_n^{up}(t) = C_n^{up}(t)$
**end if**
```

- **Case 4:** if $\nu \in [NM + 2N + 1, NM + 3N]$, then the corresponding variation will be an increase on the allocated bandwidth for reception $C_n^{do}(t)$. Formally, we have the following process:

```
$n = \nu - NM - 2N$
**if** $a_\nu(t) = 1$ **then**
    $C_n^{do}(t) = C_n^{do}(t) + \delta C^{do}$
**else**
    $C_n^{do}(t) = C_n^{do}(t)$
**end if**
```

- **Case 5:** if $\nu \in [NM + 3N + 1, NM + 4N]$, then the corresponding variation will be a decrease on the allocated bandwidth for reception $C_n^{do}(t)$. Formally, we have the following process:

```
$n = \nu - NM - 3N$
**if** $a_\nu(t) = 1$ **then**
    $C_n^{do}(t) = C_n^{do}(t) - \delta C^{do}$
**else**
    $C_n^{do}(t) = C_n^{do}(t)$
**end if**
```

5.3.2.3 Reward Function

Based on the current state and the chosen action, we could determine the reward signal that the environment should send to the agent. Specifically, we define the function $J(t)$ as the value of the objective function for our optimization problem 5.1. Indeed, by extracting the binary variables for the offloading decision and the allocated bandwidth for transmission and reception, we denote by $J_{s(t)}(t) = J_{\{x_{nm}(t)\},\{C_n^{up}(t)\},\{C_n^{do}(t)\}}(t)$ the cost function that we aim to be minimize. The agent is supposed to minimize the cost function $J(t)$ during the interaction by attempting to choose the optimal decisions. Therefore, the agent receive a reward when the cost function is reduced. In contrast, when the agent increases the cost function, the environment attributes a negative reward, as if it is a retribution. More formally, we define the reward signal as follow:

```
if $J_{s(t+1)}(t+1) < J_{s(t)}(t)$ then
    $r_{(s(t),a(t),s(t+1))}(t) = 1$
else if $J_{s(t+1)}(t+1) > J_{s(t)}(t)$ then
    $r_{(s(t),a(t),s(t+1))}(t) = -1$
else if $J_{s(t+1)}(t+1) = J_{s(t)}(t)$ then
    $r_{(s(t),a(t),s(t+1))}(t) = 0$
end if
```

5.3.3 DQN Algorithm

We present in the section the DQN algorithm that will serve to optimize the agent decisions in order to minimize the execution latency of the tasks and the energy consumption of IoT devices.

Algorithm 2: Proposed DQN Algorithm to Solve the Optimization Problem

```
Initialize the evaluation and target Q network parameters with $w$;
for $episode = 1$ to $M$ do
    Initialize $s_1(t)$ ;
    for $t = 1$ to $T$ do
        Following $\epsilon$-Greedy policy, select $a_t$ according to the uniform distribution within
          $[0, 1]$ ;
        if $p < \epsilon$ then
            $a_t = \arg\max_a Q^{pred}(s(t), a, w)$ ;
        else
            Select a random action $a_t$ with probability $\epsilon$ ;
        end
        Calculate $J_{s(t)}(t)$ according to 5.1 ;
        Observe the reward $r(t)$ and the next state $s(t+1)$ ;
        Calculate the target Q-value $Q^{target}$:
          $Q^{target} = r(t) + \gamma \max_{a'} Q^{pred}(s(t+1), a', w')$ ;
        Perform Gradient Descent step to minimize $(Q^{target} - Q^{pred}(s(t), a(t), w))^2$;
        Update the weights for the evaluation network;
    end
end
```

5.4 Simulations and Results

5.4.1 Software Environment

- **ns-3 Network Simulator:** ns-3 is a discrete-event network simulator for networking systems, developed primarily for research and educational use [47]. It is an open source simulator developed in C++ using object oriented programming. In essence, ns-3 attempts to mimic the reality as closely as possible. It actually uses several core concepts and abstractions that map efficiently the way computers and network are designed. For instance, a Node is a key entity connected to the network. It is the a container for applications, protocols and network devices. An application represent a user program that generates packet flows. In addition, a protocol represents a logic of network and transport level protocols e.g. TCP, OpenFlow. A network device is an entity connected to the channel that is the basic communication sub-network abstraction e.g. Wifi, CSMA, LTE, etc. Furthermore, ns-3 allows to model the energy consumption and the mobility of different types of Node entity.
- **OpenAI Gym:** OpenAI Gym [48] is a framework for developing and comparing reinforcement learning algorithms. It provides simple API that unifies interactions between

RL and the environment. Specifically, the main purpose of the Gym framework is to provide a standardized interface that allows access to the state and to execute actions in the environment.

Therefore, we have used ns-3 to simulate the environment while Gym was used to implement the DQN agent. In addition, we resort to a recent work [49] which enables real-time communication between ns-3 and Gym framework using ZMQ sockets [17].

5.4.2 Implementation Details

In order to simulate the EC scenario, we have chosen the following characteristics of ns-3 environment:

- **Communication Between Access Point (AP) and End Devices:** We have chosen the communication to be a wireless communication. Specifically, we have used Wifi communication.
- **Communication Between AP and Edge Server:** We have chosen a wired communication named Point-to-Point Communication in ns-3 simulator.

In our primary simulation, we set the number of end devices $N = 6$ and the number of tasks for each end device $M = 2$. In addition, to simplify the simulation and avoid the ambiguity in studying Offloading tasks in EC environment, we have made some assumptions and simplifications related to the system parameters:

- The cost function J becomes only a function of the execution time of tasks.
- We assume that the allocated bandwidth for both transmission and reception are fixed variables. Hence, we remove the allocated bandwidth from the action vector.
- In order to simulate the execution of tasks for end devices, we decided to send a fixed number of packets to local nodes as if it is a computation.
- We suppose that the local execution is less heavier than offloading to the edge server.

5.4.3 Performance Analysis

In fig. 5.2, we plot different episodes during the training of the agent. The results shows that the agent could not minimize the cost function. In fact, we observe that the cost function oscillates at a local minimum. This primary simulation motivates us to carry out further implementations in order to explain the behaviour of the DQN agent.

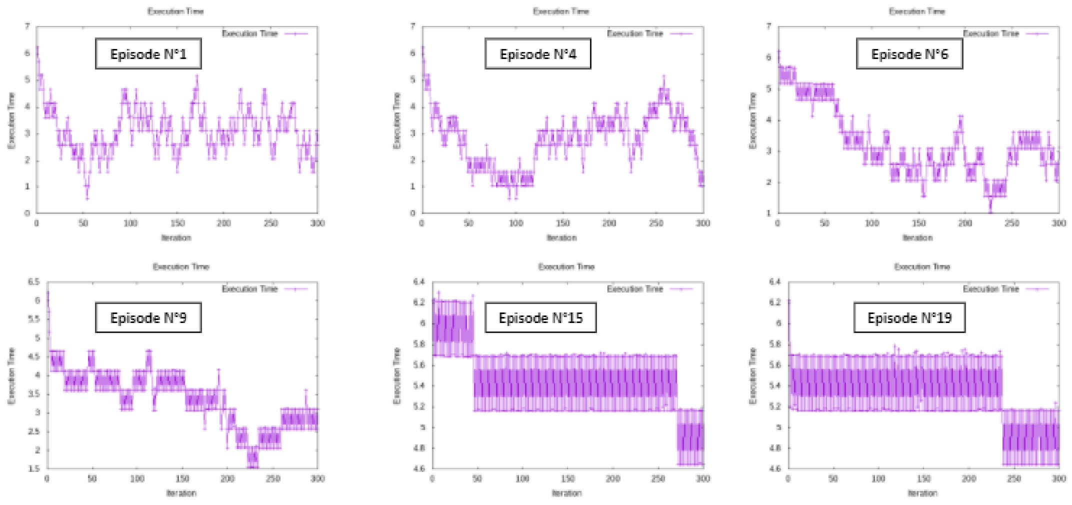


Figure 5.2: Convergence Performance

In the second run, we have tried to change the reward function. In fact, the reward function represents the signal which the agent receives while performing an action in a given sate. As a result, the reward function plays a key role for the agent to understand the environment. Therefore, we have modified the reward function as follows:

```
if $J_{s(t+1)}(t+1) < J_{s(t)}(t)$ then
    $r_{(s(t),a(t),s(t+1))}(t) = 1$
else if $J_{s(t+1)}(t+1) > J_{s(t)}(t)$ then
    $r_{(s(t),a(t),s(t+1))}(t) = -1$
else if $J_{s(t+1)}(t+1) = J_{s(t)}(t)$ and $J_{s(t)}(t) < \epsilon$ then
    $r_{(s(t),a(t),s(t+1))}(t) = +2$
else if $J_{s(t+1)}(t+1) = J_{s(t)}(t)$ and $J_{s(t)}(t) > \epsilon$ then
    $r_{(s(t),a(t),s(t+1))}(t) = -2$
end if
```

In order to prevent the agent from falling into local minimums as if it was the case in 5.2, we assign a negative reward when the cost function stabilizes at a local minimum. In contrast, we send a positive reward to the agent in case we have a stabilization in a global minimum.

Fig. 5.3 illustrate the impact of the reward function modification on the convergence performance of the agent. Indeed, we observe that the DQN agent has achieved a good performance in minimizing the cost function. In episode 9, we notice that the agent is trying to stabilize the cost function at the global minimum until he reaches it in the episode 19. Hence, we deduce an interesting finding from these simulations that the DQN components, i.e. state, action, reward function, have an enormous impact on the agent performance.

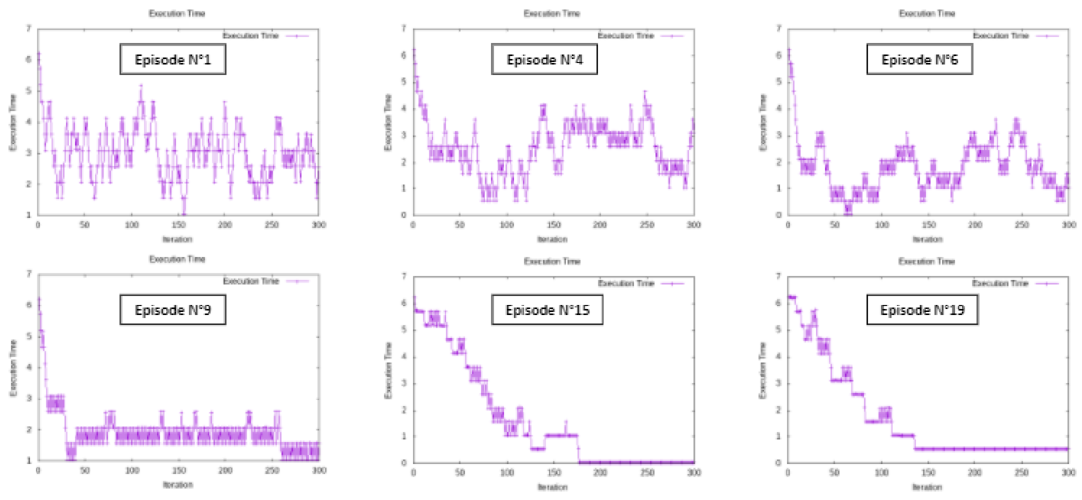


Figure 5.3: Convergence Performance using a second Reward Function

5.5 Conclusion

In this work, we investigate DRL for task offloading and bandwidth allocation using multiple simulations on ns-3 network simulator. We formulate an optimization problem with the objective of minimizing the overall cost, including the execution delay of the running applications and the energy consumption of end devices. Primary results stimulate advanced simulations by using more features, i.e adding the energy model to end devices, implementing real tasks instead of sending packets.

Conclusion

This work presents three relevant applications that serve end devices with better computing power, energy consumption and minimum latency for sensitive real-time applications.

First, network traffic prediction is the engine for many interesting applications in networking point of view such as resource allocation and QoS management. In fact, Recurrent Neural Network (RNN) have shown high capability to forecast network traffic with a promising execution time.

Second, we have leveraged Edge Computing paradigm which works in tandem with Cloud Computing to deliver real-time applications for end devices with minimum latency and efficient energy consumption. In fact, we have proposed a distributed DL inference that controls the trade-off between the accuracy of the prediction and execution latency by deciding where the execution will take place. Actually, the model performance and the latency of application are two key knobs for an efficient DL inference at the edge. This line of research is still ongoing. For instance, a possible enhancement for the proposed framework is to intervene reinforcement learning in order to automate the control of these key knobs (i.e. DL inference accuracy and the multiple constraints of networking, communication and energy consumption).

Finally, we have investigated Deep Reinforcement Learning (DRL) for optimizing offloading decision and resource allocation for multiple mobile end users. The offloading decisions have been taken centrally. Therefore, future works reveals the need to extend this framework by implementing a distributed framework by allowing each end device to take the offloading decision on its own instead of a centralized way. Additionally, more complex scenarios could be designed by adding features such as multiplying the number of edge servers. As a result, the mobile user encounter the decision of which edge server he chooses in order to perform his tasks.

Bibliography

[1] WAND Group , https://wand.net.nz/wits/leipzig/1/leipzig_i.php (Accessed the 3rd of May 2020)

[2] Ding X, Canu S, Denoeux T. Neural Network Based Models for Forecasting Proc. ADT'95. New York, USA: Wiley and Sons, 1995: 243-252.

[3] Alexander Amini, "Deep Reinforcement Learning", MIT 6.S191, 29 January, 2020.

[4] Bartosz Lewandowski, "Time Series Analysis and Prediction", University of Strathclyde, 31 Mars 2017.

[5] Mariette Awad, Rahul Khanna, "Efficient Learning Machines", Apress Open, 31 may 2016.

[6] WAND Network Research Group, https://research.wand.net.nz/software/libtrace.php (Accessed the 20th of May 2020)

[7] Zhongyi Hu Yukun Bao Tao Xiong. "Multi-Step-Ahead Time Series Prediction using Multiple-Output Support Vector Regression." In: (2013).

[8] G. Bontempi. "Long term time series prediction with multi-input multi-output local learning." In: (2008).

[9] Slurm Workload Manager - Documentation, https://slurm.schedmd.com/, (Accessed the 15th of June 2020).

[10] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in 2016 23rd ICPR, 2016.

[11] GPU Nividia Tesla T4 à coeurs Tensor pour l'inférence IA | Nividia Data Center, https://www.nvidia.com/fr-fr/data-center/tesla-t4/, (Accessed the 5th of August 2020).

[12] TensorBoard.dev - Upload and Share ML Experiments for Free, https://tensorboard.dev/experiment/tnpbEaj6RDmevmW2NdBd8w/#scalars, (Accessed the 20th June of August 2020).

[13] TensorBoard.dev - Upload and Share ML Experiments for Free, https://tensorboard.dev/experiment/ZKZ5CpWMTnKKpzkyyEoJZA/#scalars, (Accessed the 20th June of August 2020).

[14] A. Krizhevsky, “Learning multiple layers of features from tiny images,” 2009.

[15] Turnbull, J. (2014). The Docker Book: Containerization is the new virtualization.

[16] Wu, Yonghui et al. (2016). "Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". arXiv:1609.08144 [cs.CL].

[17] ZeroMQ, `http://zeromq.org`, (Accessed the 20th of June 2020).

[18] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. "Model compression". In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 535–541. ACM, 2006.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.

[20] Coral, `https://coral.ai/products/dev-board`, (Accessed the 5th of August 2020).

[21] Y. Kang, J. Hauswald, C. Gao et al., “Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge,” in Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS 2017), 2017, pp. 615–629.

[22] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, “DeepDecision: A mobile deep learning framework for edge video analytics,” in Proc. IEEE INFOCOM, Apr. 2018, pp. 1421–1429.

[23] H.-j. Jeong, H.-j. Lee, C. H. Shin, and S.-M. Moon, “IONN: Incremental Offloading of Neural Network Computations from Mobile Devices to Edge Servers,” in Proc. the ACM Symposium on Cloud Computing (SoCC 2018), 2018, pp. 401–411.

[24] S. Teerapittayanon et al., “BranchyNet: Fast inference via early exiting from deep neural networks,” in Proc. the 23rd International Conference on Pattern Recognition (ICPR 2016), 2016, pp. 2464–2469.

[25] Z. Zhao, K. M. Barijough, and A. Gerstlauer, “DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters,” IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol. 37, no. 11, pp. 2348–2359, Nov. 2018.

[26] Laizhong Cui, Shu Yang, Fei Chen, Zhong Ming, Nan Lu, Jing Qin, "A survey on application of machine learning for Internet of Things", in International Journal of Machine Learning and Cybernetics, 29 May 2018.

[27] Xiaofei Wang,Yiwen Han, et al. “Convergence of Edge Computing and Deep Learning: A Comprehensive Survey”, In 28 Janv 2020, IEEE COMMUNICATIONS SURVEYS & TUTORIALS.

[28] H. Li, K. Ota, and M. Dong, “Learning IoT in edge : Deep learning for the Internet of Things with edge computing,” IEEE Netw., vol. 32, no. 1,pp. 96–101, Jan. 2018.

[29] The Statistics Portal. Internet of things (iot): number of connecteddevices worldwide from 2012 to 2020 (in billions). `http://www.statista.com/statistics/471264/iot-number-of-connected-devicesworldwide/`, (Accessed the 5th of August 2020).

[30] "Cisco Global Cloud Index: Forecast and Methodology." [Online]. Available: `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html`, (Accessed the 5th of August 2020).

[31] Christian Ledig et al., "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network" In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017. 4681–4690.

[32] "Mobile-Edge Computing Introductory Technical White Paper," ETSI. [Online]. Available: `https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf`

[33] Dalia Adib, "Edge computing – what is edge computing?" [Online]. Available: `https://stlpartners.com/edge-computing/what-is-edge-computing/`.

[34] Warren S. McCullochWalter Pitts. "Handwritten digit recognition with a back-propagation network". In: Advances in Neural Information Processing Systems (1989).

[35] Warren S. McCullochWalter Pitts. "Handwritten digit recognition with a back-propagation network". In: Advances in Neural Information Processing Systems (1989).

[36] "Google AI algorithm masters ancient game of Go". Nature News & Comment. Retrieved 2016-01-30.

[37] Micheal Nickel, "Using neural nets to recognize handwritten digits" in Dec 2019. `http://neuralnetworksanddeeplearning.com/chap1.html`, .

[38] Andrew Ng & al., "Spécialisation Deep Learning", deeplearning.ai, in Coursera, `https://www.coursera.org/specializations/deep-learning`, (Accessed the 10th of August 2020).

[39] Shivam Bansal, "3D Convolutions : Understanding + Use Case", in Kaggle, 2018. `https://www.kaggle.com/shivamb/3d-convolutions-understanding-use-case`

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Comput., vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[41] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: encoder-decoder approaches," CoRR, vol. abs/1409.1259, 2014.

[42] Rezk et al. "Recurrent Neural Networks: An Embedded Computing Perspective" in IEEE Access, Mar 2020. [Online] `https://arxiv.org/pdf/1908.07062.pdf`

[43] A. Azzouni and G. Pujolle, “A long short-term memory recurrent neural network framework for network traffic matrix prediction,” arXiv preprint arXiv:1705.05690, 2017.

[44] A. Lazaris et al. "Deep Learning Models For Aggregated Network Traffic Prediction" in 2019 15th International Conference on Network and Service Management (CNSM), in 2019.

[45] W. Shihao et al. "A Network Traffic Prediction Method Based on LSTM" in ZTE, June 2019.

[46] M. Mirza, J. Sommers, P. Barford, and X. Zhu, “A machine learning approach to tcp throughput prediction,” IEEE/ACM Transactions on Networking (ToN), vol. 18, no. 4, pp. 1026–1039, 2010.

[47] “ns-3 source code”, `http:code.nsnam.org`, accessed: 2020-06-10.

[48] OpenAI Gym documentation. `https:gym.openai.com`, Accessed: 2020-05-20.

[49] P. Gawłowicz, A. Zubow "ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research", in ACM International Conference on Modeling, Nov 2019.

[50] L. Huang, S. Bi, and Y. J. Zhang, “Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks,” IEEE Trans. Mobile Comput., p. 1, 2019.

[51] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, “Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing, ”Digital Communications and Networks, vol. 5, no. 1, pp. 10–17, 2019.

[52] X. Liu, Z. Qin, and Y. Gao, “Resource allocation for edge computing in IoT networks via reinforcement learning,” in Proc. ICC 2019 - 2019 IEEE Int. Conf. Communications (ICC), May 2019, pp. 1–6.

[53] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. Second. The MIT Press, 2018.

[54] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, p. 529, 2015.