UNIVERSITY OF PIRAEUS

DEPARTMENT OF DIGITAL SYSTEMS

POSTGRADUATE PROGRAM "DIGITAL SYSTEMS AND SERVICES"

Traffic Load Prediction in SDN/OpenFlow Networks

Master Thesis

Candidate

Theoni Petropoulou

Advisor

Prof. Panagiotis Demestichas

Piraeus, November 2016

Acknowledgments

I would like to express my sincere gratitude to my advisor, Professor Panagiotis Demestichas, for all the support and guidance he provided during the completion of this thesis. I would also like to thank Adjunct Lecturer Dr. Kostas Tsagaris for his help, encouragement and patience, as well as Dr. Vassilios Fotinos, Michail Michaloliakos and George Poulios for all their help and cooperation throughout this work. Finally, I would like to thank my dear Marios for all his support and faith in me.

This work is dedicated to Konstantinos.

Abstract

This thesis is an experimental attempt at applying machine learning techniques to predict the load of a network within an SDN (Software Defined Networking) environment.

Utilizing an Autonomic Network Management Framework, which was implemented by the University of Piraeus, as well as actual network infrastructure with OpenFlow switches, an autonomous software component was developed using the Java programming language, which makes predictions of future traffic between five nodes on a network by implementing the Backpropagation algorithm. The software component is deployed and tested in the above Framework as an SDN application.

The main objective of the conducted experiments was both to assess the efficiency of a well-known machine learning algorithm regarding prediction of future network traffic in real conditions based on specific network traffic patterns, as well as to investigate the possibility of simultaneous execution and cooperation of more than one SDN applications in order to facilitate the autonomic decision-making of traffic routing based on future predictions without any human intervention.

Περίληψη

Η παρούσα διπλωματική εργασία αποτελεί μία πειραματική απόπειρα εφαρμογής τεχνικών μηχανικής μάθησης για την πρόβλεψη του φόρτου ενός δικτύου στο πλαίσιο ενός περιβάλλοντος SDN (Software-Defined Networking).

Αξιοποιώντας ένα Autonomic Network Management Framework, το οποίο υλοποιήθηκε από το Πανεπιστήμιο Πειραιώς, καθώς και πραγματικές δικτυακές υποδομές με OpenFlow switches, αναπτύχθηκε με την χρήση της γλώσσας προγραμματισμού Java ένα αυτόνομο στοιχείο λογισμικού, το οποίο πραγματοποιεί προβλέψεις της μελλοντικής κίνησης μεταξύ πέντε κόμβων ενός δικτύου υλοποιώντας τον αλγόριθμο Backpropagation. Το στοιχείο λογισμικού αυτό προσαρμόστηκε στο ανωτέρω Framework και εκτελέστηκε ως εφαρμογή SDN.

Ο στόχος των σχετικών πειραμάτων, που εκτελέστηκαν, ήταν διπλός, και αφορούσε αφενός την αξιολόγηση της αποτελεσματικότητας ενός τυπικού αλγορίθμου μηχανικής μάθησης ως προς την έγκυρη πρόβλεψη του μελλοντικού δικτυακού φόρτου σε πραγματικές συνθήκες βάσει συγκεκριμένων μοντέλων δικτυακής κίνησης και αφετέρου την διερεύνηση της δυνατότητας της ταυτόχρονης εκτέλεσης και συνεργασίας περισσότερων από μίας εφαρμογών SDN με σκοπό την αυτόνομη λήψη αποφάσεων από το δίκτυο σχετικά με την δρομολόγηση της κίνησης βάσει μελλοντικών προβλέψεων χωρίς ανθρώπινη παρέμβαση.

Table of Contents

Table of Figures

Table of Tables

Introduction

The main goal of this master thesis was the development of a software application that predicts future network traffic based on monitoring and analyzing current traffic in real time. The application was implemented as an autonomous component in order to be executed in a Software-Defined Networking environment.

Artificial Neural Networks and machine learning techniques were utilized for the implementation of network load prediction, whereas an Autonomic Network Management framework developed by the University of Piraeus was used for the integration and the execution of the application on top of an SDN Controller, using the Controller's NorthBound Interface to communicate with the network.

For our experiments, access to GÉANT physical testbed facility was provided by the University of Piraeus TNS Lab under the AUTOFLOW Project. The application was thus tested within a real small-scale network environment consisting of five Points of Presence, which produced and consumed real network traffic that was generated based on specific traffic patterns.

Several scenarios were tested to evaluate the prediction algorithm itself as well as the efficiency of the application running simultaneously and cooperating with other applications in an SDN context, in order to facilitate the network in making rapid autonomous decisions regarding traffic engineering and flow control.

The outline of this work is as follows:

- **Chapter 1**: A brief presentation of basic SDN concepts as well as the experimentation framework used for the application execution.

- **Chapter 2**: A brief description of the implemented software, the algorithm and the traffic models used for the experiments.

- **Chapter 3**: Experimentation and results of using load prediction mechanisms in an OpenFlow SDN network.

- **Chapter 4**: Experimentation and results of using load prediction mechanisms to influence traffic engineering in an OpenFlow SDN network.

- **Chapter 5**: General conclusions.

- **Appendix**: Application source code developed by the author.

Chapter 1 Background and Experimentation Framework

1.1 Software Define Networking and OpenFlow

The recent trends in technology industry, such as the massive virtualization of servers, the evolution of enterprise data centers and cloud infrastructures, the diversity of network access devices and applications and the need for parallel big data processing, established a need to reexamine network design and operation. The traditional network infrastructure paradigm consisting of hardcode-configured physical switches in a hierarchical design could no longer accommodate for the new high-bandwidth dynamic traffic demands and communication models. Software Defined Networking is a concept which addresses these needs of a highly flexible, adaptive, dynamically programmable and configured, as well as cost-effective network infrastructure.

The Software-Defined Networking [1] is based on the idea of decoupling the Control Plane and the Forwarding (Data) Plane on a network switch and moving the forwarding decisions to a centrally located controller. A traditional network switch uses its local logic, the Control Plane, to forward packets through the Data Plane to different destinations based on its local configuration. In a Software-Defined Network the switches use their data plane to forward actual packets, but the packet switching mechanisms and decisions are controlled by an external central element, which is implemented as a software component. The software-based SDN Controller is vendor-independent and it offers network administrators the ability to develop and run their own algorithms to control data flows and packet forwarding in a network. The abstraction between low-level packet switching and network applications and services allows for a highly adjustable and dynamically optimized network intelligence, which makes the network adapt rapidly to changes and special circumstances by addressing its particular needs.

Other basic architectural components of a Software-Defined Network beside the SDN Controller are the SDN Applications and the SDN Datapaths. SDN Applications are software components that implement a certain logic of various network requirements and actions. They communicate with the SDN Controller via NorthBound Interfaces (NBI), which typically provide open and vendor-independent abstract network views. The SDN Controller transmits the desired network actions required by the SDN Applications to the SDN Datapaths, which are logical packet switching mechanisms located in physical network devices, consisting of forwarding engines and traffic processing functions.

A general view of the SDN architecture is presented in Figure 1-1.

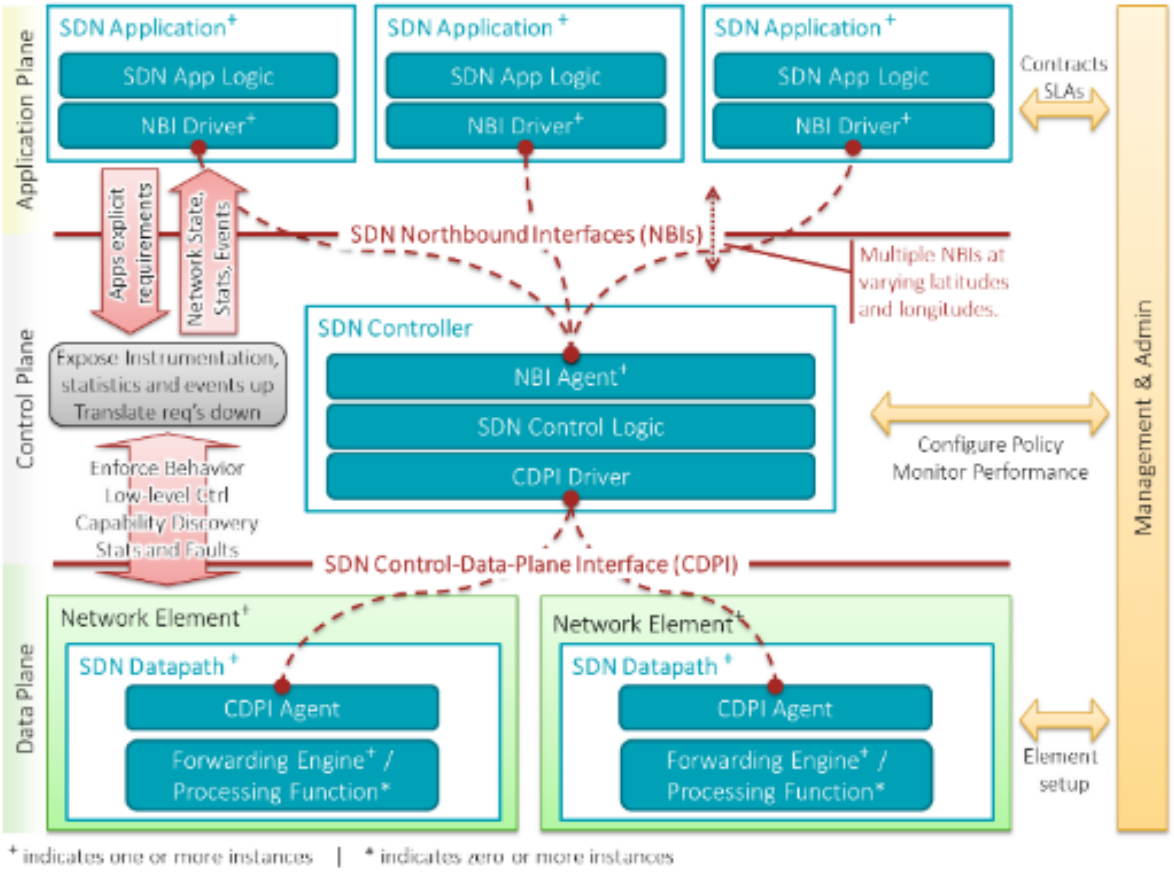

Figure 1-1: Overview of the SDN Architecture (image from Open Networking Foundation)

In a conventional switch device the communication between the Control Plane and the Forwarding Plane is implemented with a vendor-specific internal bus. In a Software-Defined Network the Controller communicates with the devices over the network using mainly the OpenFlow [2] protocol.

The OpenFlow specification defines an open standard for communication between a Controller and an OpenFlow-enabled switch through a standardized interface. An OpenFlow switch consists of:

1. A Flow Table with *actions* associated with each flow, which indicate to the switch the way to handle the specific flow.
2. The Secure Channel interface to allow communication with the Controller.

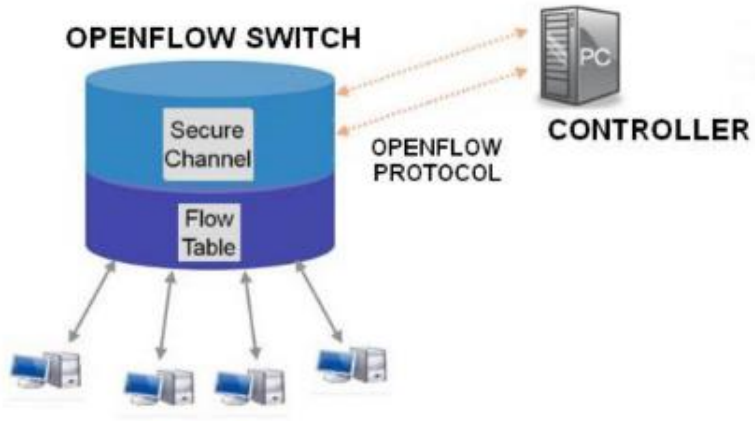

Figure 1-2: OpenFlow Switch (available at http://yuba.stanford.edu/cs244wiki/index.php/)

Through the Secure Channel the Controller can program the switch to add, remove or process flows through specific commands based on the actions supported by each device.

1.2 GÉANT Testbed

The GÉANT OpenFlow Testbed [3] is a facility designed to support Software-Defined Networking experiments and prototyping. The facility is deployed on top of the GÉANT backbone production environment and is located in five GÉANT Points of Presence in the cities below:

- London
- Frankfurt
- Vienna
- Zagreb
- Amsterdam

Computing resources are offered as Virtual Machines (VMs) upon dedicated physical servers using Xen [4] hypervisor-based virtualization. Network resources are offered utilizing software-based OpenFlow switches based on Open vSwitch (OvS) [5] and network links for the interconnection between the switches.

Two general-purpose servers are installed in each of the five GÉANT PoPs. Each of the servers that are located in PoPs is either the host of a software-based OpenFlow switch (Open vSwitch [5]) on top of a native Linux Debian distribution or the host of a Xen hypervisor for the instantiation of multiple VMs that can be allocated to user slices. Server specifications are presented in Table 1-1.

General GÉANT Server Specifications	
Number of CPUs	≥ 2
Number of cores per CPU	≥ 4
CPU Cache Size	≥ 6 MBytes
CPU Frequency	≥ 2.60 GHz
Memory Size	≥ 16 GBytes
RAID Controller	RAID 1/5 with SAS HDDs &≥ 4 HDDs support
HDDs	≥2 x SAS 146 GB
Network Interfaces	12 Gigabit Ethernet

Table 1-1: GÉANT server specifications

The Data Plane topology of the GÉANT OpenFlow facility is configured as a full mesh graph, so that every OpenFlow switch has direct connectivity with all of the other OpenFlow switches through direct connection with the routers of each facility and further on through Layer 2 MPLS VPN circuits.

Network resources are offered by utilizing OpenFlow switches based on Open vSwitch and network interconnection links of **1Gbps**.

End users can connect to the GÉANT OpenFlow Facility using the Virtual Machines hosted on servers on the five PoPs as traffic producers and consumers, as well as installing their own equipment within the facility.

The network virtualization technology used by the GÉANT OpenFlow Facility, referred to as “network slicing technique”, dynamically allocates one or a range of VLAN IDs to each user slice to distinguish traffic. The network is partitioned per physical or logical Open vSwitch interface. VLANs can be involved in experimentation within the slice when a set of VLAN IDs is allocated to a slice. Thus any of the experimenters can use its own range of VLAN IDs on top of the OpenFlow topology.

The switches are logically separated for each research group with the use of VLAN technology.

Moreover, the hosts of the software-based OpenFlow Switches are hosted in the pre-installed servers in each PoP. These hosts can be reachable through the Internet via their public IP addresses via SSH.

For our experiments in the GÉANT OpenFlow facility we used the 192.168.1.0/24 private subnet with VLAN ID 256.

The IP addresses of the hosts that are connected to the relevant OpenFlow switch are shown in Table 1.2.

Host IP Addresses		
PoP	**Public IP Address**	**Private IP Address**
London	62.40.110.137	192.168.1.3
Frankfurt	62.40.110.71	192.168.1.2
Vienna	62.40.110.110.3	192.168.1.4
Zagreb	62.40.110.101	192.168.1.5
Amsterdam	62.40.110.35	192.168.1.1

Table 1-2: GÉANT host IP addresses

The facility's management and control plane elements and software are hosted in the Frankfurt PoP. This includes the GÉANT OpenFlow Control Framework and the FlowVisor software [6]. FlowVisor is a network virtualization application, which can be considered as a proxy protocol sitting logically between the multiple controllers and the OpenFlow switches in order to allow multiple controllers to share the same network infrastructure without interfering with each other.

A general overview of the GÉANT OpenFlow facility is presented in Figure 1-3.

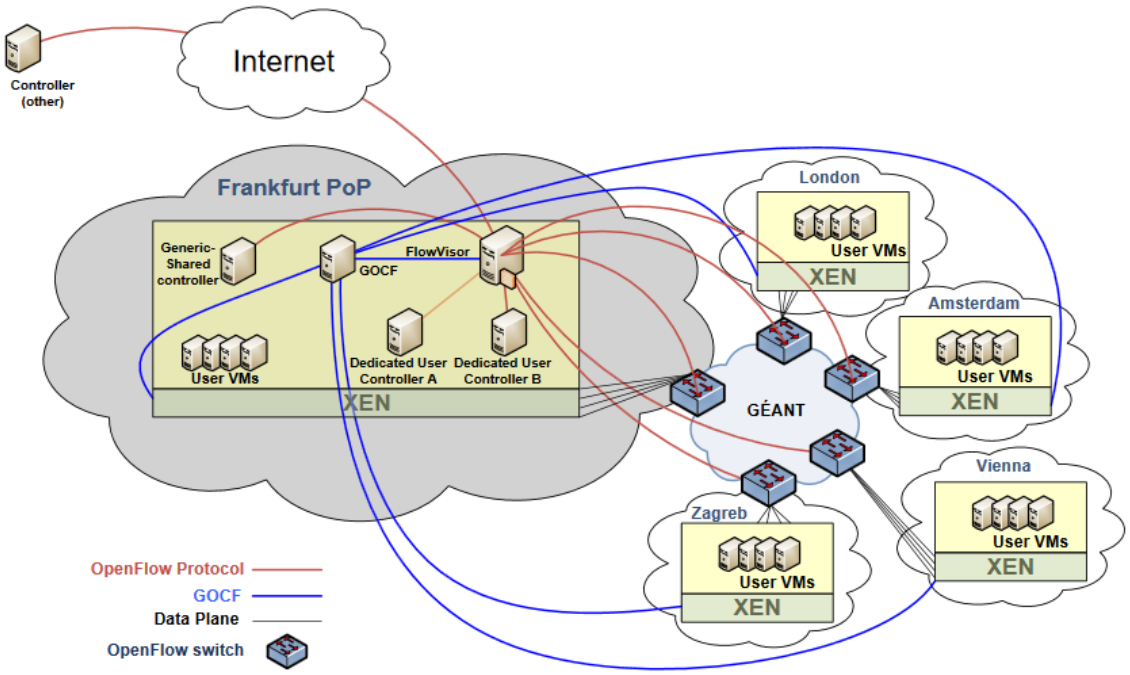

Figure 1-3: GÉANT OpenFlow Facility (image from GÉANT OF Testbed Users Manual)

1.3 ANM Framework

Load prediction mechanisms in a Software Defined Network were implemented as an autonomous software component which was integrated in an Autonomic Network Management (ANM) Framework, designed and implemented by the ***University of Piraeus TNS Lab*** for the purposes of **AUTOFLOW Project**. The main objective of AUTOFLOW is to exploit both Autonomic Network Management (ANM) and Software-Defined Networking (SDN) in order to provide a realistic solution to the complexity of network management. The main component of the project is the ANM Core, which includes a set of blocks that provide fundamental functionalities (Governance, Coordination and Knowledge) for the efficient operation of autonomic mechanisms. The latter are deployed as SDN applications on top of an SDN controller and use the controller's NorthBound API to interact with the network.

The main role of the ANM core is to provide a seamless integration and expandability ("plug 'n' play" and "unplug 'n' play"), as well as to ensure a trustworthy interworking of autonomic mechanisms within an operator's management ecosystem, based on three architectural blocks. The *Governance* block aims at giving a human operator a mechanism for controlling the network from a high-level business point of view, without the need of having a deep technical knowledge of the network. Information regarding autonomic mechanisms as well as monitoring parameters and alerts are sent to the operator, while the operator's policies are translated into specific parameters relevant to the operation of the mechanisms. All this information is sent to the mechanisms through this block. The *Knowledge* block manipulates information and knowledge to be exploited from the other blocks and mechanisms. The role of the *Coordination* block is to ensure the proper sequence in triggering of mechanisms and

the conditions and constraints under which they will be invoked taking into account operator and service requirements.

The ANM core is the framework that allows for the efficient operation and coexistence of multiple autonomic mechanisms (Autonomic Control Loops - ACLs), which run as SDN applications. In the scenarios described in this thesis we focus on a Load Prediction (LP) mechanism that addresses the problem of network congestion prediction, whereas in a specific experiment we also exploit an autonomic Traffic Engineering (TE) mechanism (Core-TE) developed by the University of Piraeus TNS Lab, that addresses the problem of policy-based traffic engineering in SDN/OpenFlow networks.

The ANM Core uses the Floodlight [7] Controller, which offers an easy-to-use northbound API, remotely accessible through a REST API, which was easily integrated with the rest of the framework, as well as various monitoring statistics regarding network status.

An overview of the framework used for our experiments is presented in Figure 1-4.

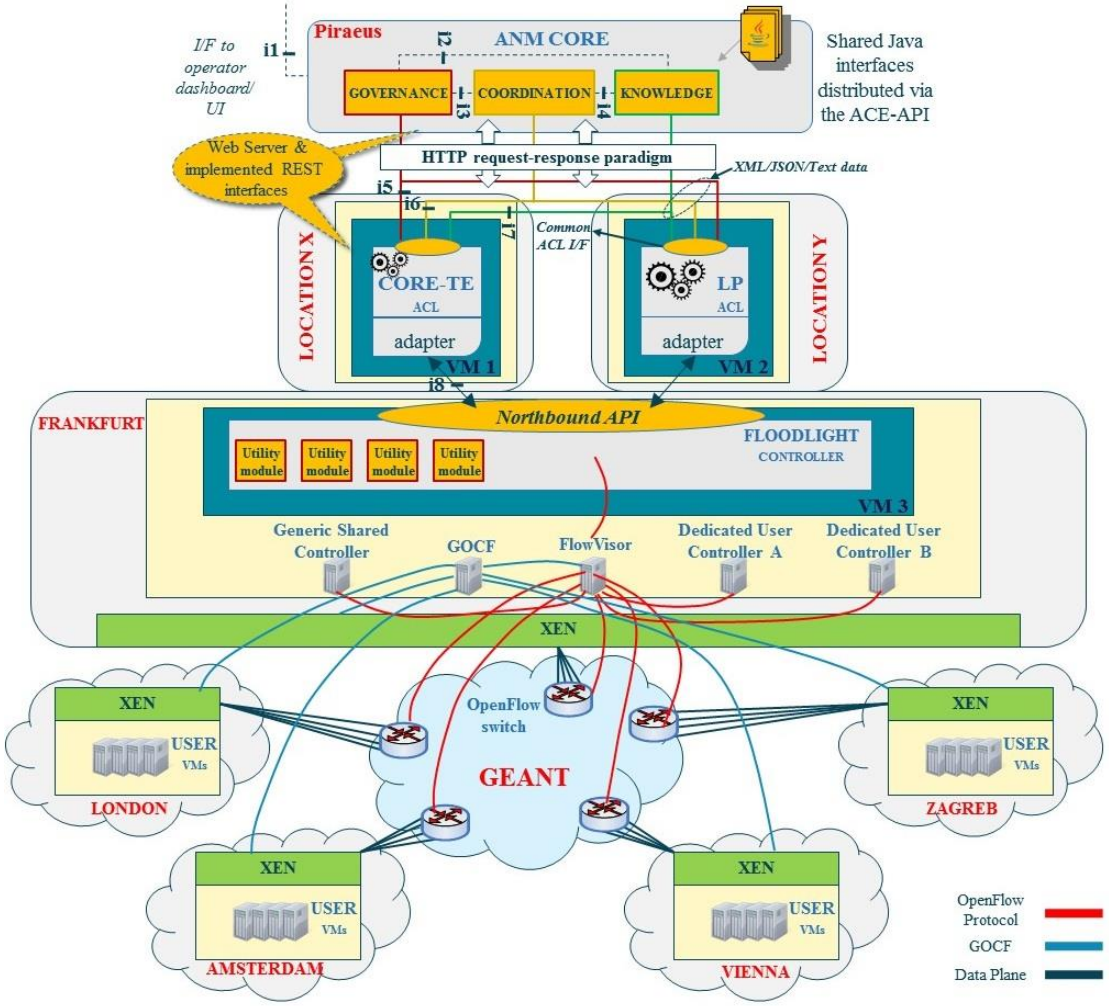

Figure 1-4: Experimentation framework overview (image provided by the University of Piraeus TNS Lab)

Chapter 2 Implementation

2.1 Software

In order to carry out the experiments described in the following sections, certain software components have been developed.

Load prediction mechanism has been implemented as an Autonomic Control Loop (ACL) software component (LP ACL), which can be deployed within the ANM Framework, the core component developed by the **University of Piraeus TNS Lab** under the **AUTOFLOW Project**.

Our load prediction software component integrates and cooperates with the following **AUTOFLOW project components**:

- **ANM CORE**: The Autonomic Network Management framework that provides fundamental functionalities to facilitate network operations and an ecosystem to host SDN applications.

- **Floodlight**: The Java software SDN Controller used based on the OpenFlow protocol.

- **FloodlightRESTClient**: A library to ease Floodlight NB integration through Java/HTTP.

- **MetricMonitor**: A visualization tool that taps to Floodlight's NB and plots various port statistics using FloodlightRESTClient

- **Core-TE ACL**: An Autonomic Control Loop which implements an algorithm for policy-based traffic engineering.

In order to implement load prediction mechanisms in the abovementioned SDN ecosystem, the following two software projects have been developed:

1) **TimeSeriesPrediction**. This project implements the actual machine learning algorithm for load prediction over time, as described in the following section.

2) **LPACL**. This project is the actual autonomic mechanism which is deployed within the ANM Core Framework. It integrates with the Floodlight Controller Northbound API in order to read actual traffic demand values between the five GEANT PoP in real time and utilizes TimeSeriesPrediction project to perform the actual load forecasting based on those values. The software design is based on the autonomic system paradigm, where MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) feedback loops are used to ensure self-adaptation. Upon the ACL deployment, four methods run consecutively and repeatedly throughout the life cycle of the ACL, Monitor, Analyze, Plan and Execute, referred to as MAPE Loops, in order to perform traffic monitoring and future network load calculation.

All software has been implemented using Java 8 SE programming language with Eclipse SDK [8]. The source code of the implemented components can be found in the Appendix.

2.2 Prediction Algorithm

Extended research in time-series forecasting has been carried out in the recent years, examining mainly the application of Artificial Neural Networks (ANNs) as prediction tools for modelling non-linear systems. Supervised learning algorithms and regression-based techniques such as Backpropagation, Support Vector Machine and hybrid models have been extensively used for years for the prediction of future electric load demand, monthly river flow, network load etc. Most cases focus on pre-trained ANNs with historical data, while there are some approaches which use unsupervised techniques such as Kohonen maps and clustering algorithms in order to perform short or long-term forecasting.

In order to carry out the experiments described in the following sections, we decided to use machine learning techniques to investigate the possibility of continuous forecasting of the future network load based on the monitoring of current network load established by common traffic patterns.

In particular, we used a standard two-layer Backpropagation algorithm with a bipolar sigmoid activation function to train each neural network. Each neural network consists of five neurons at the input layer and one neuron at the output layer. The terms of Learning Rate and Momentum have been implemented as parameters for the user to experiment with the speed and the stability of the training algorithm.

The *iterative forecasting*, according to which the predicted values are used as inputs for the next forecasts, is used as a method for multi-step-ahead prediction.

Specific details about the algorithm execution are presented along with each experiment in the following Chapters.

2.3 Traffic Models

As we indicated earlier, in order to perform the experiments described in the following sections, we utilized the GEANT facility. End-to-end flows between the five GEANT PoP (OpenFlow Switches) were established by generating real UDP traffic using Iperf [9] scripts in each PoP.

For each experiment we generated network traffic based on two traffic models presented in [10], which we describe briefly below.

- **Traffic Model 1 (TM1)**: random traffic demand follows a uniform distribution with various Average Demand Intensity (ADI). For the specific experiments we utilized TM1 with low, medium and high average load demand intensity of **40 Mbps**, **280 Mbps** and **400 Mbps** respectively.

- **Traffic Model 2 (TM2)**: traffic demand between source and destination nodes is variable in time according to a sinusoidal function.

Chapter 3 Load Prediction in SDN/OpenFlow Networks

3.1 Introduction

In this experiment we attempt to investigate whether the ANM and SDN/OpenFlow technologies can be used for short prediction of network load, with the ultimate purpose of using the acquired knowledge of future traffic during the process of decision-making on routing and traffic engineering directly by the network in real time without any human intervention.

By providing valuable information in advance, short-term network load forecasting aims at improving the efficiency of the network and facilitating the real-time management and control of the network functions.

Load forecasting is implemented in real time for each *Source-Destination Pair* using neural networks trained with the Backpropagation algorithm. For each traffic model used, the predicted values are compared to the actual load values and the corresponding diagrams for indicative number of source-destination pairs are presented.

The main objective of the experiment is to evaluate the behavior of the real-time load prediction algorithm based on specific traffic models.

3.2 Actors, setup, topology

All experiments described in this Chapter are carried out by generating real traffic on the GÉANT OpenFlow physical testbed. The GÉANT OpenFlow facility consists of five Points of Presence (PoPs) located in Amsterdam, Frankfurt, London, Vienna and Zagreb respectively. An OpenFlow software switch and a Virtual Machine on a physical server are located in each PoP. The five OpenFlow switches are interconnected in a full mesh with 1Gbps links. The network controller is located in a Virtual Machine in Frankfurt. Hosts are accessible through the Internet via SSH on their public IP addresses.

Real traffic is generated between the five hosts, which correspond to a total of twenty (20) **Source-Destination pairs (SD Pairs)**, by running Iperf scripts on each Virtual Machine. For this particular experiment certain traffic scripts were selected to reflect the two different traffic models as described in 2.3.

In all the experiments described below, monitoring and load prediction are performed *per SD Pair*.

For this particular experiment a two-layer neural network with five neurons in the input layer and one neuron in the output layer has been implemented for each source-destination pair. Each neural network is trained using the standard Backpropagation algorithm as described in Section 2.2.

The algorithm has two discrete phases, the training and the prediction phase. During the training phase a scenario-specific set of real load values per SD Pair is presented to each neural network as the training set. In each epoch of the algorithm, after samples of the training set are presented to the neural network, each neuron's weights are adjusted and the summary mean square error for all neurons is calculated.

During the prediction phase, the already trained neural networks, after being presented with input vectors of 5 sequential network load values per SD Pair (equal to the number of neurons in the input layer), they calculate the vectors of the estimated future load values. The prediction horizon i.e. the size of the prediction vectors varies according to the specific configuration of each experiment, as described in the following sections.

In order to demonstrate the behavior of the Backpropagation algorithm regarding its forecasting capabilities, we tested two different experiment scenarios.

3.3 Scenario 1 – Online Forecasting

3.3.1 Storyline

The goal of the first experimentation scenario is to evaluate the ability of the ACL to recognize short-term network patterns in real time without previous knowledge, in order to use them for future prediction.

Initially, traffic is generated according to each traffic model between the five PoPs and, at the same time, LP ACL is deployed in the ANM framework.

The two phases of the prediction algorithm (training and prediction phase) are executed consecutively in each MAPE loop repeatedly throughout the lifecycle of the ACL. During the training phase a set of 50 real network load values (measured in Kbps) for each SD Pair are recorded each second by monitoring the network traffic for 50 seconds and used to train each neural network. The size of the training sets (50 values corresponding to 50 seconds of monitoring) was chosen after experimenting as a trade-off in order to minimize the effects of the time-consuming training phase on the runtime efficiency of the prediction phase.

During the prediction phase, an input vector of the next 5 real load values per SD Pair is presented to the trained neural networks, after monitoring the actual network traffic for 5 seconds. Each network produces a prediction vector of 55 estimated future load values per SD Pair. The 55-length prediction vector corresponds to the duration of the next monitoring period (50 seconds of monitoring to form the 50-value training set and 5 seconds of monitoring to form the 5-value input vector to be used in the next prediction cycle).

Actual and predicted load values are recorded in two CSV files for reasons of comparison and result evaluation, as well as possible utilization by other ACLs.

3.3.2 Execution

In this experiment the prediction algorithm was evaluated in four cases with real traffic based on the traffic models described in Section 2.3. To further evaluate the behavior of the Backpropagation algorithm in the context of each traffic model, each experiment case was also executed with different parameters of Learning Rate and Momentum.

Execution steps are as follows:

- The ANM Core component runs and the AUTOFLOW ANM Application starts.
- The LP ACL runs and appears in the right up pane of the ANM Application.
- Real network traffic is generated by running the Iperf scripts of the specific traffic model on each GÉANT PoP.
- The LP ACL is deployed in the ANM framework.
- After 5 minutes of deployment time, two CSV files are updated with the actual and the predicted load values per SD Pair.

Upon completion of each experiment case, the time-series graphs comparing the actual to the predicted traffic for each source-destination pair are drafted. Indicative number of graphs is presented in the following evaluation section.

3.3.3 Results

Case 1:

- Traffic model: 1
- Average Intensity: 40

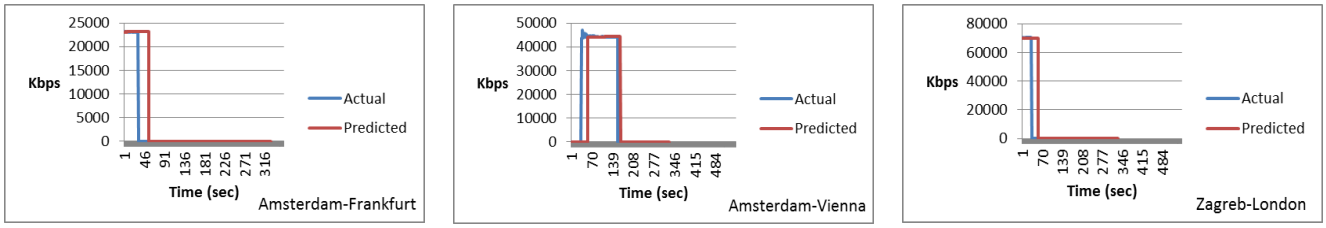

Figure 3-1: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 40 (Learning rate: 0.05, Momentum: 0.9)

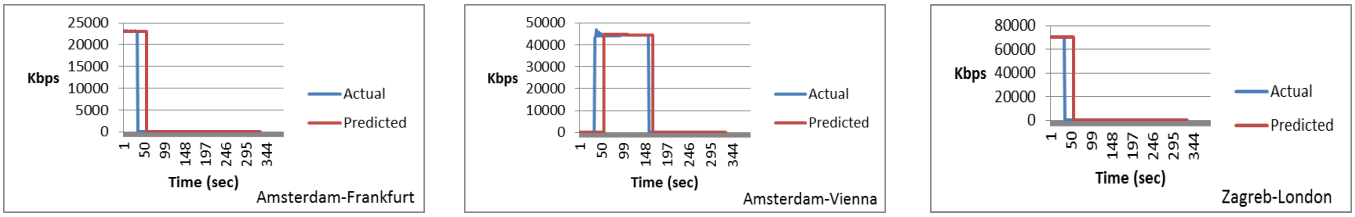

Figure 3-2: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 40 (Learning rate: 0.01, Momentum: 0.6)

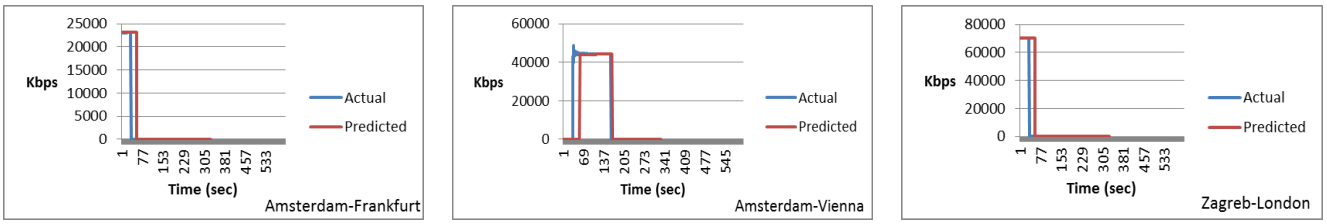

Figure 3-3: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 40 (Learning rate: 0.01, Momentum: 0.0)

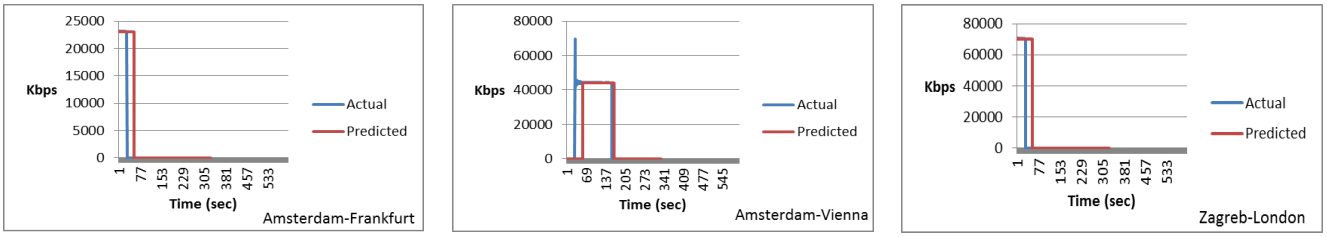

Figure 3-4: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 40 (Learning rate: 0.1, Momentum: 0.0)

Case 2:

- Traffic model: 1
- Average Intensity: 280

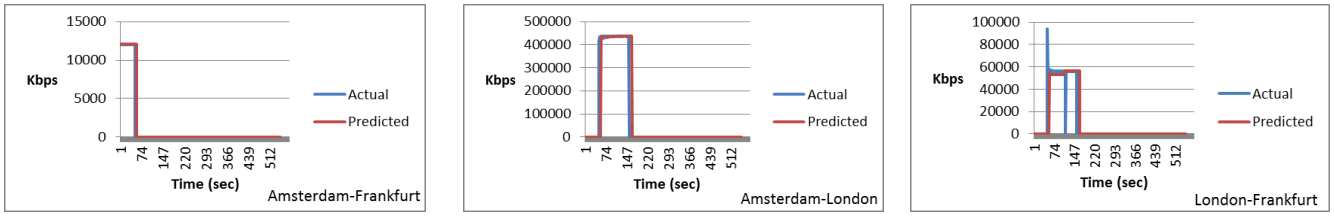

Figure 3-5: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 280 (Learning rate: 0.05, Momentum: 0.9)

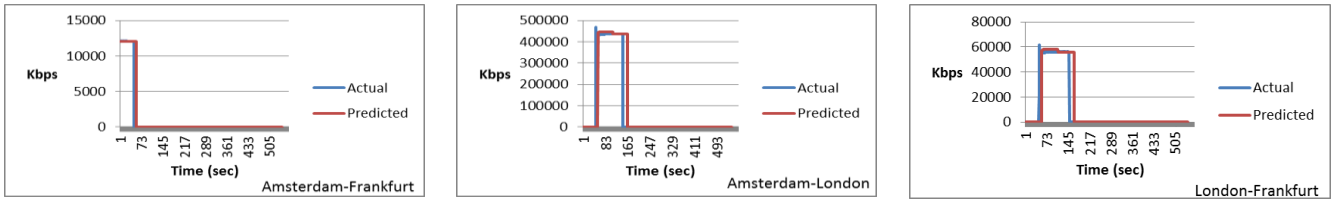

Figure 3-6: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 280 (Learning rate: 0.01, Momentum: 0.6)

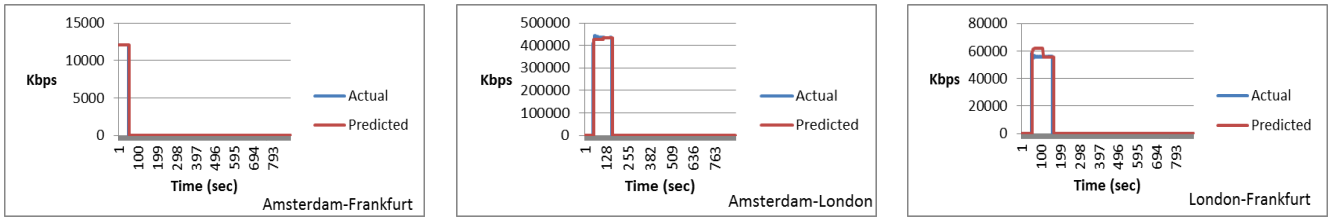

Figure 3-7: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 280 (Learning rate: 0.01, Momentum: 0.0)

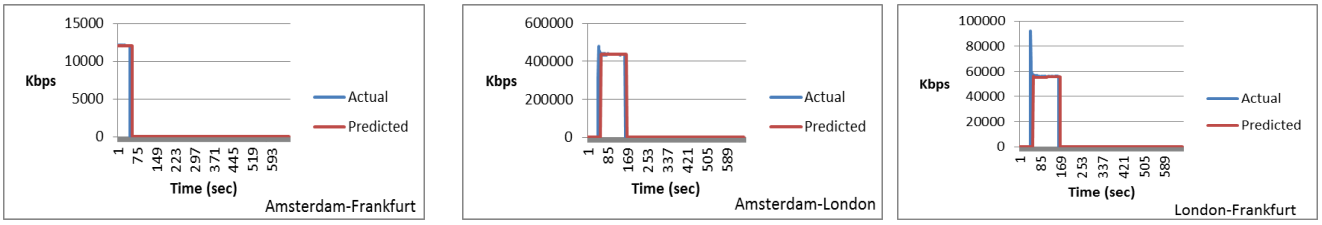

Figure 3-8: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 280 (Learning rate: 0.1, Momentum: 0.0)

Case 3:

- Traffic model: 1
- Average Intensity: 400

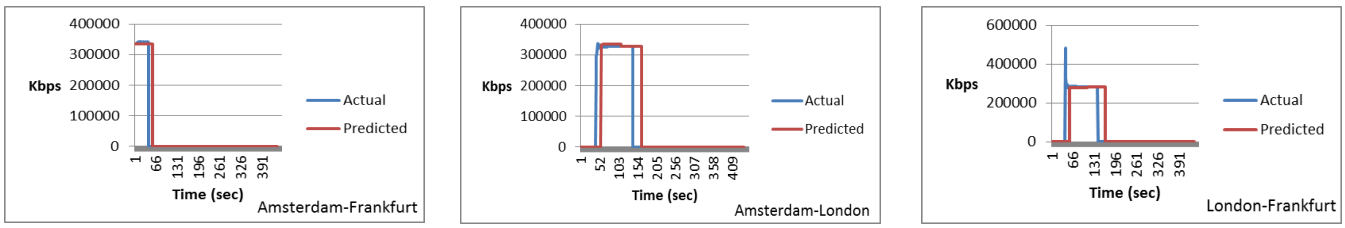

Figure 3-9: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 400 (Learning rate: 0.05, Momentum: 0.9)

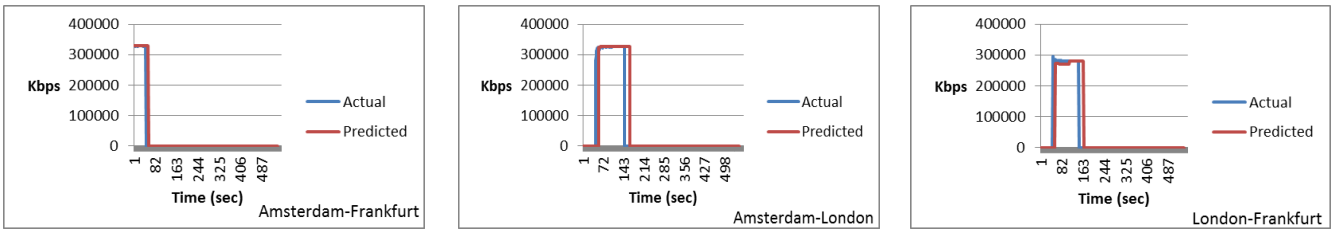

Figure 3-10: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 400 (Learning rate: 0.01, Momentum: 0.6)

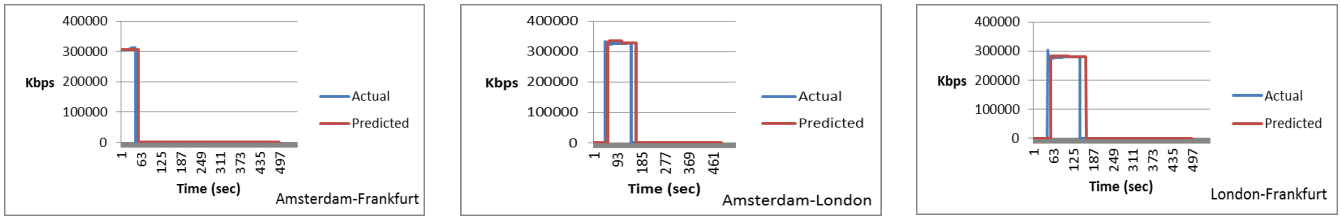

Figure 3-11: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 400 (Learning rate: 0.01, Momentum: 0.0)

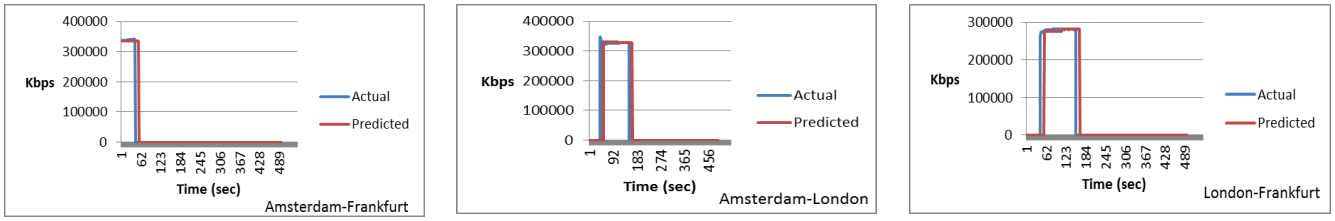

Figure 3-12: Actual and predicted traffic for 3 SD Pairs with real-time training with TM1-ADI 400 (Learning rate: 0.1, Momentum: 0.0)

Case 4:

- Traffic model: 2

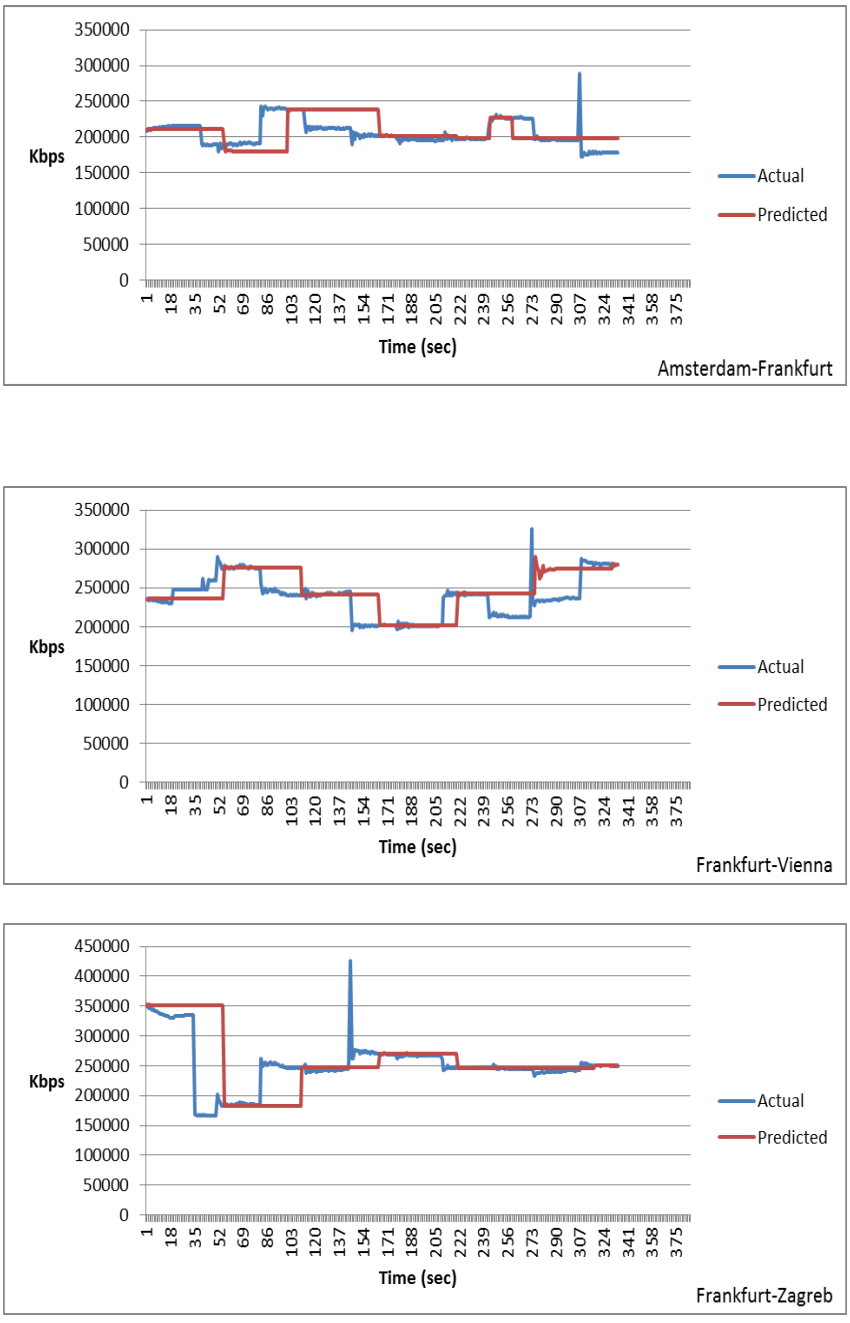

Figure 3-13: Actual and predicted traffic for 3 SD Pairs with real-time training with TM2 (Learning rate: 0.05, Momentum: 0.9)

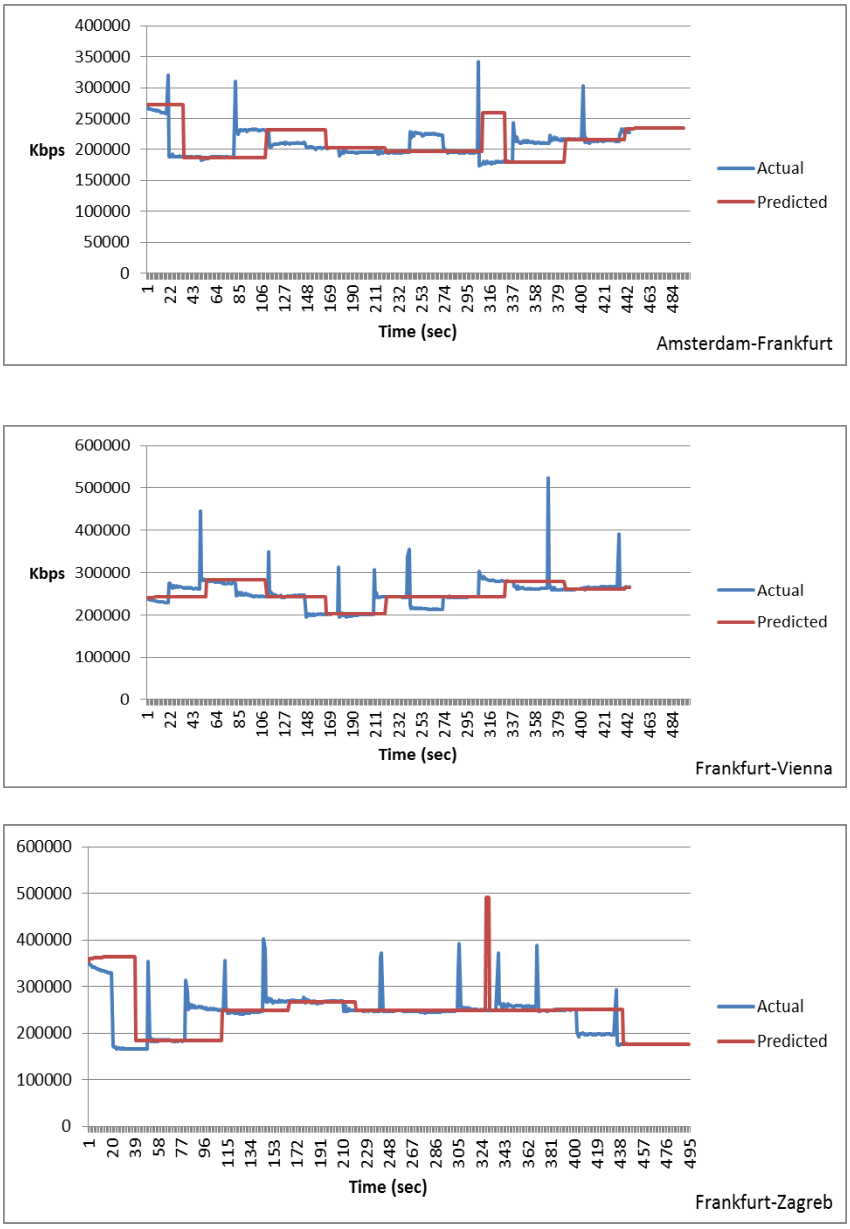

Figure 3-14: Actual and predicted traffic for 3 SD Pairs with real-time training with TM2 (Learning rate: 0.01, Momentum: 0.6)

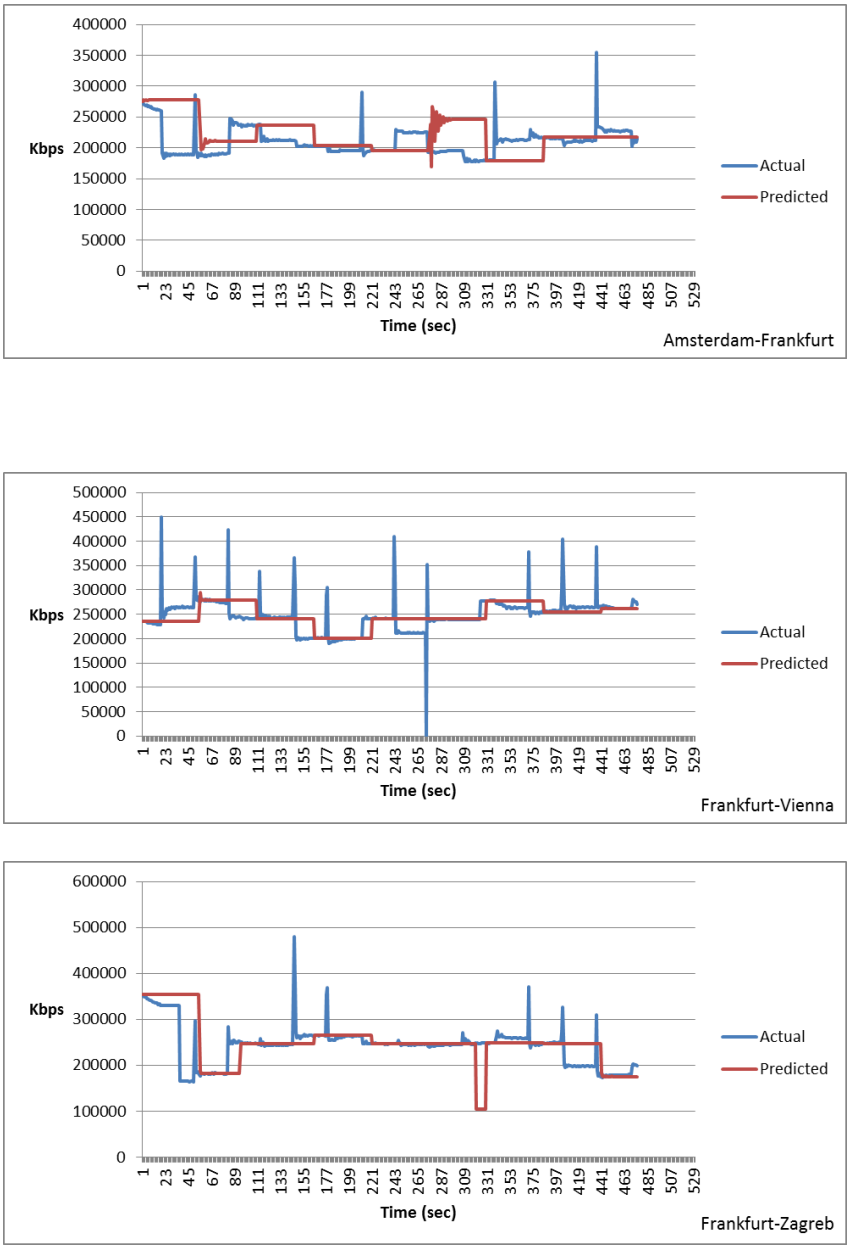

Figure 3-15: Actual and predicted traffic for 3 SD Pairs with real-time training with TM2 (Learning rate: 0.01, Momentum: 0.0)

The experimental results indicate that the load intensity of the network traffic has little to no effect on the behavior of the prediction algorithm. Moreover, tuning Backpropagation parameters such as the learning rate and the momentum has no significant effect on prediction accuracy in the specific context.

The objective of the implemented algorithm is to predict the general trend of network traffic in a long period of time (at least as long as the LP ACL is running) by utilizing small periods of traffic as training sets for the neural networks. Nevertheless, in *Case 4*, in which Traffic Model 2 was tested, apart from the less accurate results compared to the results of Traffic Model 1, we observe that there is a time shift of the predicted time series. In other words, prediction is

delayed as the result of the variation of the real traffic produced by the specific model, which forms a pattern that spans multiple training phases. Since Backpropagation is a supervised learning algorithm, a change in load which occurs at the end of the training phase will be realized by the algorithm after the completion of the current training and prediction phases, i.e. during the next prediction cycle. In worst case scenarios short-term load variations will not be detected by the algorithm at all.

In most of our experiments the length of the time shift was half the duration of the training phase (25 seconds). Increasing the length of the training set would prolong the training phase without any significant effect on real-time prediction of network load that varies over time. On the other hand, it would prove useful in case of network monitoring and prediction based on known network traffic patterns observed in predefined time periods (days or weeks). This case is examined in Section 3.4 (Scenario 2) below.

The deviation of the numeric values of the actual and the predicted load is limited. Given that the network load is measured in Kbps, this deviation is actually insignificant.

Short-term bursts or instantaneous load variations cannot be estimated by the algorithm.

3.4 Scenario 2 – Forecasting by pre-trained networks

3.4.1 Storyline

The goal of the second experimentation scenario is to investigate ways to improve the algorithm's accuracy and performance, as well as to optimize the ACL's efficiency, based on the assumption that network load is not subject to random high variations, but it follows some general trends throughout certain time periods (i.e. certain weekdays or certain dates of a year).

Therefore, a different approach is attempted in the second scenario, where real-time prediction is performed by pre-trained neural networks.

For scaling reasons, we assume that the four traffic models described in Section 2.3 reflect four different traffic patterns that occur within four different days of the week. Prior to forecasting, the LP ACL is deployed four times to train neural networks with the four above-mentioned traffic scenarios (TM1 with three different average load intensities and TM2) after five minutes (300 seconds) of traffic monitoring with training sets of 300 load values per SD Pair. No changes have been made to the algorithm other than increasing the training set size to 300.

For each traffic model the training phase took place within a different day of the week. The state of each neural network after final weight adjustment has been serialized and saved in a binary file named after the specific day (one file for each traffic model, representing the state of the neural network for each SD Pair at the corresponding day).

The LP ACL was deployed four times within the same days of the next week. In order to evaluate the forecasting algorithm and validate the prediction results against real load values,

the corresponding traffic for each model is being generated before each deployment of the LP ACL.

Upon detection of the current day, the neural network state is de-serialized and loaded from the file that corresponds to the specific day. This technique allows for the ACL to perform immediate prediction based on already known network traffic patterns, completely omitting the training phase. The already trained neural networks continuously calculate prediction vectors of 5 future load values, after being provided with input sets of 5 actual traffic values (5 seconds of monitoring the network), throughout the deployment duration of the ACL.

As in the first scenario, actual and predicted load of the network is recorded in CSV files for comparison, evaluation and future exploitation.

3.4.2 Execution

Training

- The ANM Core component runs and the AUTOFLOW ANM Application starts.
- The LP ACL runs and appears in the right up pane of the ANM Application.
- Real network traffic is generated by running the Iperf script of the specific traffic model on each GÉANT PoP.
- The LP ACL is deployed in the ANM framework.
- After 5 minutes of training, a file named after the specific day is generated, containing the serialized state of the neural network for each SD Pair.

Prediction

- Traffic is generated according to the traffic model used during the same day.
- The LP ACL is deployed in the ANM framework.
- After 5 minutes of execution, the actual and predicted load values for each SD Pair are recorded in two CSV files.

Upon completion of each experiment case, the time-series graphs comparing the actual to the predicted traffic for each source-destination pair are drafted. Graphs for indicative number of SD Pairs are presented in the following section.

3.4.3 Results

In each experiment case the Backpropagation algorithm was executed with a *Learning Rate* of **0.01** and a *Momentum* of **0.0**. The specific experiment has also been performed with

different learning rate and momentum parameter values. We observed that parameter tuning had no effect on the performance of the algorithm in the specific context. Hence, we do not consider necessary to present additional time-series graphs.

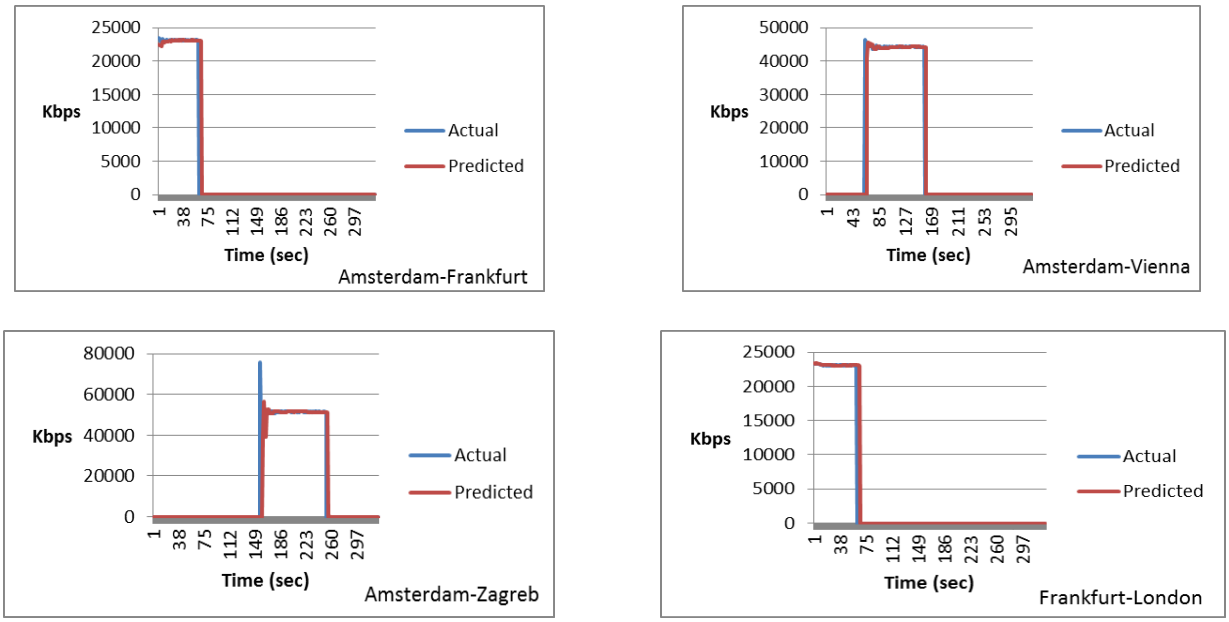

Figure 3-16: Actual and predicted traffic for 4 SD Pairs using pre-trained neural networks (TM1-ADI 40)

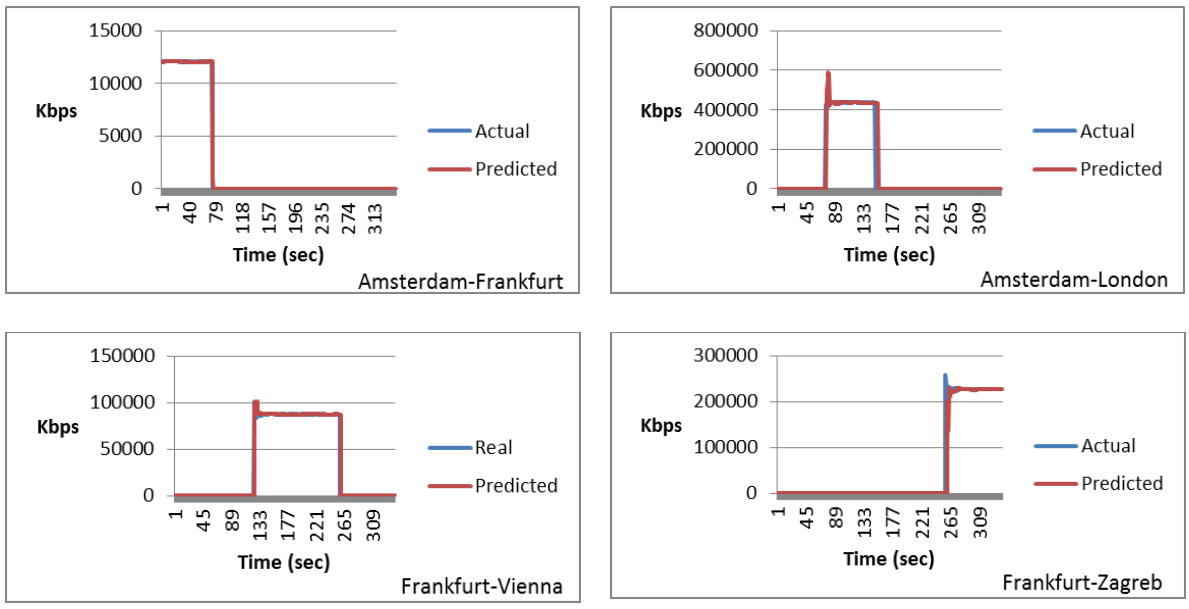

Figure 3-17: Actual and predicted traffic for 4 SD Pairs using pre-trained neural networks (TM1-ADI 280)

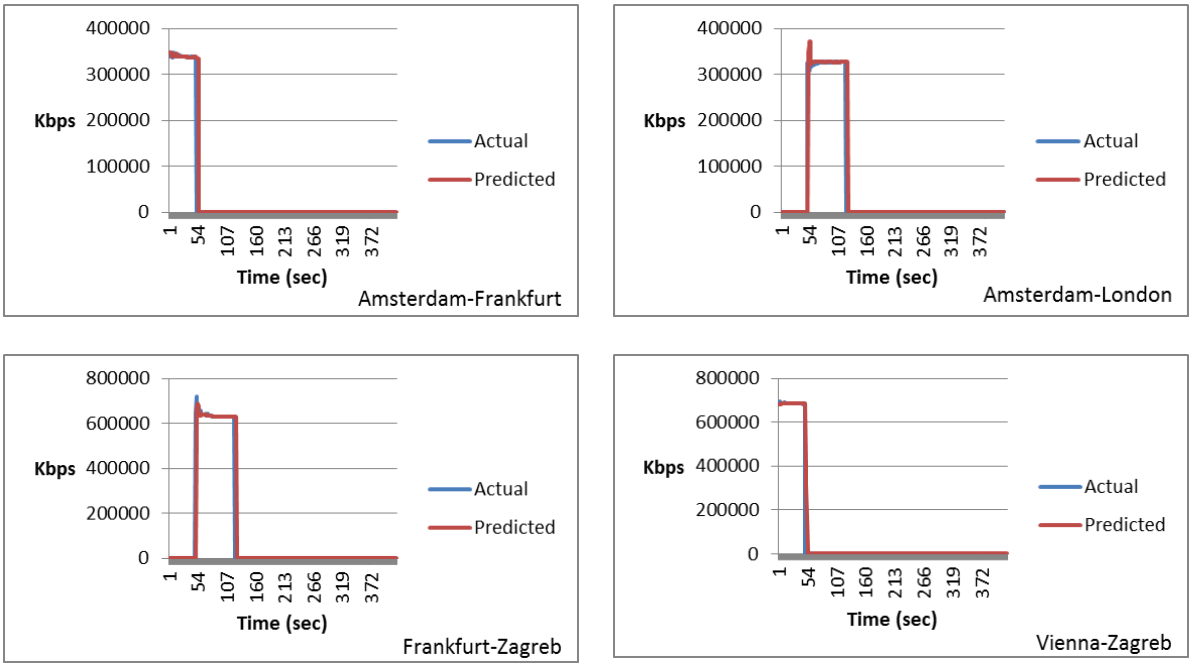

Figure 3-18: Actual and predicted traffic for 4 SD Pairs using pre-trained neural networks (TM1-ADI 400)

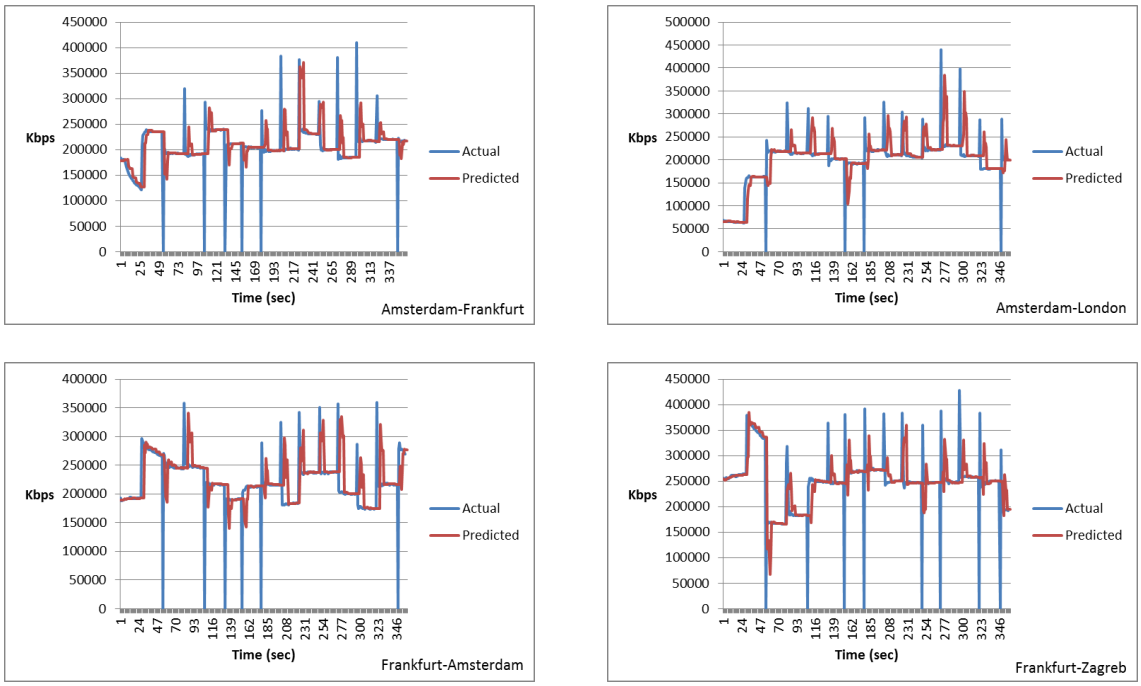

Figure 3-19: Actual and predicted traffic for 4 SD Pairs using pre-trained neural networks (TM2)

The results of the second scenario indicate that there is a high time and value accuracy of the predicted series against the actual ones. Compared to the results of the first scenario of the experiment, the time shift of the prediction is now minimized to 2-3 seconds. This is absolutely normal, since the entire network traffic pattern was used as the training set for Backpropagation algorithm in this case.

Moreover, because the training phase was omitted and the duration of the prediction phase in each MAPE loop was minimized to 5 seconds, we observed that the execution of the algorithm was faster and more efficient. In the first scenario the prediction phase in each MAPE loop is suspended for 50 seconds during each training phase, hence there is the need for a relatively large prediction vector of 55 values, which increases the probability of prediction error. In this case prediction phases producing small 5-value vectors follow short monitoring periods of 5 seconds, resulting in faster traffic variation detection and subsequently prediction error minimization.

Despite the high accuracy of the prediction results, the main drawback of the method presented in this second scenario is that it requires offline training for various network traffic patterns prior to the actual forecasting.

After running both scenarios, we also made some observations which apply in both cases. While Traffic Model 1 scripts are running, at any moment there is a maximum of eight (8) SD Pairs with concurrent traffic. This fact has no impact on the ACL execution. When using Traffic Model 2 though, especially with scripts running on all five GÉANT PoPs, there are more than 30 active flows and all 20 SD Pairs have simultaneous traffic. In this case, having also to access files on disk, the execution of the ACL appeared to be slower, especially during the training phase in the first scenario. In some cases we observed that the 1-second interval between reading traffic demand of each SD Pair during the training phase actually corresponded to a real interval of 3 to 5 seconds. Although not affecting the quality of the prediction, since predicted values are based on simple vectors of sequential values, this fact is certainly indicative of a scalability issue.

3.5 Conclusions

The goal of the first experiment was to evaluate the performance and the quality of the implemented Backpropagation algorithm in predicting future network load in real time, during the deployment of the LP ACL in the ANM framework. The evaluation method was based on the comparison of the predicted values against the actual values of network load, which were being recorded throughout the experiment execution.

The results of both scenarios clearly show that the algorithm is quite sufficient in predicting network load in cases of traffic with little volatility or generally known traffic patterns and it can be used by SDN components in real time load forecasting, although it is limited by the network size. The algorithm is not able to handle random sudden surges in traffic demand (1-2 seconds).

Further investigation can be carried out on implementing different approximation and classification machine learning algorithms (i.e. Support Vector Machine and Self-Organizing

Maps with regression) and comparing prediction results to Backpropagation, as well as utilizing load prediction results from other SDN components in decision making on routing and traffic engineering.

Chapter 4 Autonomic Traffic Engineering with Load Prediction in SDN/OpenFlow Networks

4.1 Introduction

In this experiment we explore possible ways of load prediction exploitation in the ANM/SDN context, hence the ability of the network to continuously adjust its behavior regarding routing and traffic engineering in an autonomic way, based on short-term forecasting of network traffic. The motivation behind this particular experiment is to show whether a prediction algorithm, as described in the previous experiment, can be successfully applied in a SDN framework and contribute to the optimization of resource management and the efficiency of the network operation.

In order to realize the scenario proposed in this experiment, we utilized two different Autonomic Control Loops (ACLs), the LP (Load Prediction) ACL and the Core-TE (Traffic Engineering) ACL. The LP ACL, which has been described in the previous section, performs online load prediction and outputs vectors of estimated future load values for each source-destination pair (SD Pair) of the network. The Core-TE ACL controls the network functions through the enforcement of high-level policies like Energy Efficiency or Load Balancing. In this scenario, after both ACLs have been deployed in the ANM framework, they run concurrently and they exchange information. The short-term load prediction per SD Pair produced by the LP ACL is received by the Core-TE ACL. The Core-TE ACL enforces network configuration to proactively adapt to the future state by adjusting the paths used to accommodate traffic, either by aggregation or diffusion, based on estimated future load, thus resulting in Energy Efficiency or Load Balancing respectively.

The evaluation methodology relies on the comparison of the routing decisions made by the Core-TE ACL when receiving the actual traffic against the paths it chooses when it reads the estimated load of the next moment.

The main objective of this experiment is to highlight any possible practical value of load prediction in autonomous network management, as well as to examine the co-operation of different SDN applications in the ANM framework.

4.2 Actors, setup, topology

The general setup of the experiment resembles that of the previous experiment, with the addition of the Core-TE ACL. The ANM Core is used for the deployment of LP and Core-TE ACLs.

The GÉANT OpenFlow physical testbed with Virtual machines in the five Points of Presence (PoPs) located in Amsterdam, Frankfurt, London, Vienna and Zagreb, connected with OpenFlow software switches in a full mesh with 1Gbps links, was used for traffic generation. The general topology setup is shown in Table 4-1.

PoP	Host Private IP address	Switch name
Amsterdam	192.168.1.1	S1
Frankfurt	192.168.1.2	S2
London	192.168.1.3	S3
Vienna	192.168.1.4	S4
Zagreb	192.168.1.5	S5

Table 4-1: Topology set-up

In this particular experiment we observe what happens to the routing paths of the traffic of the active SD Pairs when traffic is detected on another link. For this purpose traffic was generated between three PoPs as follows.

Two scripts run on the Virtual Machines of *Amsterdam* and *Vienna* respectively. Initially, the host at Amsterdam (192.168.1.1) starts sending *30 Mbps* of traffic to Vienna (192.168.1.4) and after 150 seconds it starts sending *10 Mbps* of traffic to London (192.168.1.3). After 150 seconds the host in Vienna starts sending *20 Mbps* of traffic to the host in London.

4.3 Storyline

For the following experiment we assume that load prediction is performed by pre-trained neural networks, as described in the second scenario of the previous experiment (Section 3.4). For this purpose we have trained neural networks for the traffic model described in Section 4.2 and verified the prediction results prior to the execution of the experiment, following the steps described in Section 3.4.2.

In order to examine how load prediction influences traffic engineering, initially we deployed only the Core-TE ACL, after traffic has been generated between the three PoPs (Amsterdam to Vienna, Amsterdam to London and Vienna to London). Energy Efficiency was selected as a high level policy throughout the experiment. During execution the paths chosen by the Core-TE ACL for each SD Pair in each MAPE loop were stored, for future reference. To validate the integrity of the results, the above scenario (Core-TE ACL) was executed five (5) times.

Then, we generated the same traffic again between the three PoPs. This time, both the LP and Core-TE ACLs were deployed in the ANM framework. Upon deployment, the LP ACL started calculating the five future load values for each SD Pair and updated a CSV file in each MAPE loop. Instead of receiving real-time traffic demand of each SD Pair, the Core-TE ACL read the estimated load of each SD Pair from the CSV file in each MAPE loop. The ultimate goal of this

experiment is to examine whether the Core-TE ACL will choose to reroute traffic of one or more SD Pairs as soon as it detects estimated additional traffic (by reading the predicted values).

The paths chosen by the Core-TE ACL for each SD Pair in each MAPE loop were also saved. The above scenario (Core-TE ACL with LP ACL) was also executed five (5) times.

Upon completion of execution, the paths chosen by the Core-TE ACL and the number of activated links in both scenarios (Core-TE ACL running with and without LP ACL) were compared. The results are presented in Section 4.5.

4.4 Execution

For validation purposes, each of the following test cases was executed five times. The execution steps are presented below:

Case 1 – Core-TE ACL

- The ANM Core component runs and the AUTOFLOW ANM Application starts.
- The Core-TE ACL runs and appears in the right up pane of the ANM Application.
- Real network traffic is generated by running Iperf scripts on two GÉANT PoPs (Amsterdam and Vienna).
- The Core-TE ACL is deployed in the ANM framework with Energy Efficiency as the high-level policy.
- After 10 minutes of execution, the Core-TE ACL was un-deployed and the file with the recorded paths for each SD Pair was updated.

Case #2 – Core-TE ACL with LP ACL

- The ANM Core component runs and the AUTOFLOW ANM Application starts.
- The Core-TE and the LP ACLs run and appear in the right up pane of the ANM Application.
- Real network traffic is generated by running Iperf scripts on two GÉANT PoPs (Amsterdam and Vienna).
- The LP ACL is deployed in the ANM framework.
- The Core-TE ACL is deployed in the ANM framework with Energy Efficiency as the high-level policy.

- After 10 minutes of concurrent execution, the two ACLs were un-deployed and the recorded paths for each SD Pair were updated.

4.5 Results

In the following tables we observe the active SD Pairs, the traffic route (paths) and the number of activated inter-switch links in two consecutive MAPE loops, as they were recorded during execution. Time values T0 and T1 refer to the loops before and after detecting additional traffic on the third SD Pair respectively.

The results after five executions of the Core-TE ACL running without the LP ACL are presented below.

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	5
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S5-S1-S2-S3	

Table 4-2: Core-TE ACL without LP ACL – Execution #1

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	5
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S5-S1-S2-S3	

Table 4-3: Core-TE ACL without LP ACL – Execution #2

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	5
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S5-S1-S2-S3	

Table 4-4: Core-TE ACL without LP ACL – Execution #3

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	5
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S5-S1-S2-S3	

Table 4-5: Core-TE ACL without LP ACL – Execution #4

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	5
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S5-S1-S2-S3	

Table 4-6: Core-TE ACL without LP ACL – Execution #5

We should clarify the aforementioned routing decisions (Table 4-2-Table 4-6) taken by the Core-TE ACL. Although the selected policy is Energy Efficiency, Core-TE routes traffic not from the direct link between an SD Pair but from paths with more links. Two reasons can be identified for this behavior. At first, Core-TE attempts to re-use already activated links and avoid the activation of unused ones. Therefore, it is possible to select a longer path (in number of hops) for the accommodation of traffic between an SD Pair and not the shortest one. In addition, Core-TE is designed to act proactively. It promotes the activation of specific links in the network, which belong to a ring that connects all switches. In this manner, subsequent requests will be accommodated by the already activated resources, resulting in the increase of the energy savings in the long term. Taking into account these two aspects of the Core-TE mechanism, the aforementioned routing decisions can be justified. More information about the Core-TE mechanism and the heuristic algorithm that is based on can be found in [10].

From the above tables it is clear that the Core-TE ACL shows a steady behavior during the five executions with the specific traffic model. As soon as actual traffic is detected on the third SD Pair (Vienna-London), the Core-TE chooses to route this traffic via the default path for the Energy Efficiency high-level policy, i.e. the ring between all switches (S4-S5-S1-S2-S3), according to the implementation of the traffic engineering algorithm. In each case the number of activated links increases from 3 to 5. Eventually, after some loops the Core-TE decides to reroute the traffic of the third SD Pair via a shorter path (S4-S3) to allow for better use of resources.

The tables below show the results after five executions of Core-TE and LP ACLs running simultaneously.

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S3	

Table 4-7: Core-TE ACL with LP ACL – Execution #1

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	4
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S2-S3	

Table 4-8: Core-TE ACL with LP ACL – Execution #2

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S3	

Table 4-9: Core-TE ACL with LP ACL – Execution #3

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S3	

Table 4-10: Core-TE ACL with LP ACL – Execution #4

Time	SD Pairs	Paths	Active links
T0	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	3
	S1->S3 (Amsterdam-London)	S1-S2-S3	
T1	S1->S4 (Amsterdam-Vienna)	S1-S2-S3-S4	5
	S1->S3 (Amsterdam-London)	S1-S2-S3	
	S4->S3 (Vienna-London)	S4-S5-S1-S2-S3	

Table 4-11: Core-TE ACL with LP ACL – Execution #5

The results above show that in one case the Core-TE used the "ring" path to route the traffic of the third SD Pair, whereas in the other four cases it opted for shorter paths (S4-S3 and S4-S2-S3). In these cases the number of activated links after the detection of new traffic remains the same or increases from 3 to 4, allowing the deactivation of more links (for energy efficiency).

To further examine and confirm the behavior of the two ACLs running concurrently, we executed the experiment another three times. The results were identical to the ones presented in Table 4-7.

Comparing the paths used to route the traffic and the activated links in the above two cases, we observe that there is an average gain of 30% regarding the number of activated links when traffic engineering utilizes load prediction to route the traffic. When Core-TE ACL runs without LP ACL, it decides to reroute the traffic in every loop based on the high-level policy and the state of the network at the specific millisecond during which various network parameters such as active SD Pairs, flows and traffic demand are captured. When traffic engineering algorithm (Core-TE) is in place, the frequent rerouting of traffic, which may occur on some flows even without changing the high-level policy, it is possible to affect the way that the ACL reads the value of traffic demand of a given SD Pair, if it is captured right after the traffic has been rerouted –although the actual traffic demand value of the SD Pair is not affected. On the other hand, load prediction is based on neural networks which have been pre-trained without traffic engineering being present in the network, thus the routing paths are the same throughout the execution and the traffic demand values recorded from each SD Pair are straightforward. When Core-TE ACL runs with load prediction, it is forced to read the predicted load for each SD Pair instead of reading actual traffic demand values. Being presented with more consistent sequential load values from the prediction CSV file, and considering the fact that each load value represents the traffic demand of the future moment, the Core-TE ACL has the ability to rapidly adapt to a future state by making a more efficient routing decision.

4.6 Conclusions

In this section we investigated the co-existence of traffic engineering and load prediction SDN applications, as well as the practical use of real-time network traffic forecasting in an SDN environment. To facilitate the evaluation method and simplify the presentation of the results, we performed a small-scale experiment examining the Energy Efficiency policy with a traffic model designed specifically for this purpose.

The execution process of the experiment clearly proved an absolutely smooth co-operation of two different software components in runtime inside the ANM framework. The results of this particular small-scale experiment demonstrated also that traffic engineering can be positively affected by real-time load forecasting, since the “software-based nature” of the network and its control functions allows for almost instantaneous adaptation to a future different state.

The average 30% gain in utilization of resources, although it seems quite promising, it should be considered as a positive indication in the context of the specific experiment only. Further experimentation is encouraged in order to discover the actual benefits of load prediction on traffic engineering in a network with more nodes and heavier traffic, as well as to examine possible scalability issues.

Chapter 5 Conclusions

The main objective of this thesis was to evaluate a typical forecasting algorithm used for real-time network load prediction within the context of an SDN network.

The conducted experiments demonstrated that the synergy of ANM and SDN may provide a realistic solution towards a more intelligent and adaptive network with minimum human intervention. The well-known and simple techniques used for the software development of the SDN components indicate that network researchers and administrators can develop any algorithm to manage and control various network circumstances. Moreover, the management framework proved to be an operator-friendly environment, where any SDN application can be rapidly deployed on demand. In comparison with conventional network architectures, the experimentation with SDN Controller and applications clearly demonstrated the advantages of a centralized software management that provides a universal view of the network and enhances the concept of the "Network as a Service" without the need to configure individual devices.

As for the specific small-scale experiments, the forecasting algorithm proved to be quite reliable for predicting network traffic following generally known patterns and to have a positive influence on dynamic traffic engineering even in real-time execution in small networks. Nevertheless, after monitoring the concurrent execution of both the LP and Core-TE ACLs, it was obvious that in a realistic implementation of real-time traffic engineering with load prediction there would be certain limitations imposed by the network diameter, the number of flows, the short changes of traffic demand, which may cause a large number of useless variations in traffic forwarding, as well as the processing resources. Thus, at this point we consider it as a valuable solution for small network segments which experience deterministic traffic patterns, as a dynamic network management mechanism.

References

[1] Open Network Foundation (2012), “Software-defined networking: The new norm for networks,” White Paper. Available at https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] Open Networking Foundation, OpenFlow. Available at https://www.opennetworking.org/sdn-resources/openflow

[3] Technical Annex B: GÉANT OpenFlow Facility. Available at http://geant3.archive.geant.net

[4] “Xen”. Available at http://www.xenserver.org/

[5] “Open vSwitch”. Available at http://openvswitch.org/

[6] Sherwood, Rob, et al. (2009), “Flowvisor: A network virtualization layer,” OpenFlow Switch Consortium, Tech. Rep.

[7] “Floodlight”. Available at http://www.projectFloodlight.org/Floodlight/

[8] Eclipse Software Development Kit (SDK), http://www.eclipse.org

[9] “Iperf”. Available at https://iperf.fr/

[10] Foteinos, Vassilis, et al. (2014), “Operator-friendly Traffic Engineering in IP/MPLS Core Networks,” IEEE Transactions on Network and Service Management, Vol. 11, Issue 3, 333-349.

Bibliography

Alberti, A.M. (2012), "Software-Defined Networking: Perspectives, Requirements, and Challenges, 7th API Think-Tank on Software-Defined Networking.

Baumann, T., Germond, A. J. (1993), “Application of the Kohonen network to short-term load forecasting,” Neural Networks to Power Systems, 1993. ANNPS '93, Proceedings of the Second International Forum on Applications of IEEE, 407-412.

Cheng, H. et al. (2006), “Multistep-ahead time series prediction”, PAKDD 10, 765-774.

Diamantaras, K., Kung, S.Y. (1996), Principal Component Neural Networks: Theory and Applications, John Wiley, NY.

Ganek, A.G., Corbi, T.A. (2003), “The dawning of the autonomic computing era”, IBM Systems Journal 42 (1), 5-18.

Hamzaçebi, C., Akay, D., Kutay, F. (2009), “Comparison of direct and iterative artificial neural network forecast approaches in multi-periodic time series forecasting”, Expert Systems with Applications, Vol. 36, Issue 2, Part 2, 3839-3844.

Heineman, G., Councill, W. (2007), Component-based software engineering, Addison-Wesley.

Kaastra, I., Boyd, M. (1996), “Designing a neural network for forecasting financial and economic time series”, Neurocomputing 10, 215-236.

Kephart, J. O., Chess, D.M. (2003), “The vision of autonomic computing”, Computer 36.1, 41-50.

Papalexopoulos, A. D., Shangyou Hao, Peng, T.-M. (1993), “Short-term system load forecasting using an artificial neural network,” Neural Networks to Power Systems, ANNPS '93, Proceedings of the Second International Forum on Applications of IEEE, 239-244.

Park, Dong C., et al. (1991), “Electric load forecasting using an artificial neural network, IEEE Trans. Power Systems, Vol. 6, Issue 2, 442-449.

Sfetsos, A., Siriopoulos, C. (2004), “Time series forecasting with a hybrid clustering scheme and pattern recognition,” IEEE Trans. Systems Man & Cybernetics: Systems, Vol. 34, Issue 3, 399-405.

Weigend, A.S., Gershenfeld, N.A. (1993), Time Series Prediction: forecasting the future and understanding the past, Addison-Wesley.

Yang, B., Sun. Y. (2008), “An improved neural network prediction model for load demand in day-ahead electricity market,” Intelligent Control and Automation, WCICA 2008, 4425-4429.

"How the emergence of OpenFlow and Software-Defined Networking (SDN) will change the networking landscape" (2012), Brocade, available at http://docs.media.bitpipe.com/io_10x/io_102956/item_484489/TT%2012-129%20Brocade%20openFlow%20and%20SDN.pdf

Open Network Foundation (2013), "SDN Architecture Overview", available at https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf

GÉANT OpenFlow Facility – GOFF, User Manual, v0.3, October 2013.

Glossary

ACL	Autonomic Control Loop
ADI	Average Demand Intensity
ANM	Autonomic Network Management
ANN	Artificial Neural Network
API	Application Programming Interface
CSV	Comma-Separated Values
LP	Load Prediction
MPLS	Multi-Protocol Label Switching
NB	NorthBound
NBI	NorthBound Interface
OvS	Open vSwitch
PoP	Point of Presence
REST	Representational State Transfer
SD Pair	Source-Destination Pair
SDN	Software-Defined Networking
TE	Traffic Engineering
TM	Traffic Model
VM	Virtual Machine
VPN	Virtual Private Network

APPENDIX

Application source code

Note: Only the source code developed exclusively by the author is included in this Appendix.

TimeSeriesPrediction Project

TSPrediction.java

```
package timeseriesprediction;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;


public class TSPrediction implements java.io.Serializable {

    ArrayList<ArrayList<Double>> allData  = new ArrayList<ArrayList<Double>>();
    ArrayList<double[]> inputData = new ArrayList<double[]>();
    ArrayList<double[]> totalPrediction = new ArrayList<double[]>();
    ArrayList<double[]> testPrediction = new ArrayList<double[]>();
    public static double[][] prediction;

    int threadNum;

    ArrayList<String> testThreadOrder;

    ActivationNetwork actNetworks[];

    private int iterations = 1000;
    private double learningRate = 0.01;
    private int sigmoid = 2;
    private int windowSize = 5;
    private int predictionSize = 1;
    private double factor;
    private double setMin;
    private double momentum = 0.0;


    public ArrayList<String> testOrder() {
        return testThreadOrder;
    }

    public ArrayList<double[]> getPrediction() {
        return totalPrediction;
    }

    public TSPrediction(ArrayList<ArrayList<Double>> inputData) {
        allData = inputData;
        for(ArrayList<Double> list: allData) {
            Object x = list.toArray();
            this.inputData.add((double[]) x);
        }
    }

    public TSPrediction() {

    }

    public TSPrediction(int threadNum) {
        this.threadNum = threadNum;
    }

    public double getLearningRate() {
        return learningRate;
    }

    public void setLearningRate(double value) {
        learningRate = value;
    }
```

```
public double getMomentum() {
    return momentum;
}

public void setMomentum(double value) {
    momentum = value;
}

public int getSigmoid() {
    return sigmoid;
}

public void setSigmoid(int value) {
    sigmoid = value;
}

public int getIterations() {
    return iterations;
}

public void setIterations(int value) {
    iterations = value;
}


public ArrayList<double[]> checkPrediction() {
    return this.testPrediction;
}


public void predict(ArrayList<Double[]> input, int predictionSize) {
    int index;
    testThreadOrder = new ArrayList<String>();
    totalPrediction = new ArrayList<double[]>();
    for(index=0; index<input.size(); index++) {
        // forecast horizon
        int PREDICTION_SIZE = predictionSize;
        ArrayList<Double> predictionVector = new ArrayList<Double>();
        ArrayList<Double> normalizedFeeder = new ArrayList<Double>();
        ArrayList<Double> feeder = new ArrayList<Double>();
        double[] predictedValue = new double[1];
        Double sdpairInputData[] = input.get(index);

        // normalization [-0.85, 0.85]
        double maxVal = Collections.max(Arrays.asList(sdpairInputData));
        double minVal = Collections.min(Arrays.asList(sdpairInputData));

        if((maxVal - minVal) == 0) {
            factor = 0;
        }
        else {
            factor = 1.7 / (maxVal - minVal);
        }
        if(factor==0) {
            factor = 1;
        }
        setMin = minVal;


        for (int r = 0; r < sdpairInputData.length; r++) {
            normalizedFeeder.add((sdpairInputData[r] - setMin) * factor - 0.85);
            feeder.add(sdpairInputData[r]);
        }

        for (int l = 0; l < PREDICTION_SIZE; l++) {
            double[] inputVector = new double[normalizedFeeder.size()];
```

```
            for (int s = 0; s < inputVector.length; s++) {
                inputVector[s] = normalizedFeeder.get(s);
            }

            predictedValue = actNetworks[index].calculate(inputVector);

            predictedValue[0] = (predictedValue[0] + 0.85) / factor + setMin;

            if(predictedValue[0] < 0) {
                predictedValue[0] = 0.0;
            }
            predictionVector.add(predictedValue[0]);
            normalizedFeeder.remove(0);
            normalizedFeeder.add((predictedValue[0] - setMin) * factor - 0.85);
            feeder.remove(0);
            feeder.add(predictedValue[0]);

            maxVal = Collections.max(feeder);
            minVal = Collections.min(feeder);

            if((maxVal - minVal) == 0) {
                factor = 0;
            }
            else {
                factor = 1.7 / (maxVal - minVal);
            }
            if(factor==0) {
                factor = 1;
            }
                setMin = minVal;
        }

        double[] predictionArray = new double[PREDICTION_SIZE];
        for (int s = 0; s < predictionArray.length; s++) {
            predictionArray[s] = predictionVector.get(s);
        }
        totalPrediction.add(predictionArray);
    }
}


public void train(ArrayList<Double[]> inputSet) {
    int index ;
    testThreadOrder = new ArrayList<String>();
    actNetworks = new ActivationNetwork[inputSet.size()];
    for(index=0; index<threadNum; index++) {

        Double sdpairInputData[] = inputSet.get(index);
        double maxVal = Collections.max(Arrays.asList(sdpairInputData));
        double minVal = Collections.min(Arrays.asList(sdpairInputData));
        // fix normalization
        if((maxVal - minVal) == 0) {
            factor = 1;
        }
        else {
            factor = 1.7 / (maxVal - minVal);
        }
        setMin = minVal;

        int samples = sdpairInputData.length - predictionSize - windowSize;

        double[][] input = new double[samples][];
        double[][] output = new double[samples][];

        for (int i = 0; i < samples; i++)
        {
            input[i] = new double[windowSize];
            output[i] = new double[1];
```

```
    // set input
    for (int j = 0; j < windowSize; j++)
    {
        input[i][j] = (sdpairInputData[i + j] - setMin) * factor - 0.85;
    }

    if(factor==0) {
        factor = 1;
    }
    // set output
    output[i][0] = (sdpairInputData[i + windowSize] - setMin) * factor - 0.85;
}

int neuronCount[] = new int[2];
neuronCount[0] = windowSize*2;
neuronCount[1] = 1;

actNetworks[index] = new ActivationNetwork(new BipolarSigmoidFunction(sigmoid),
     windowSize, neuronCount);
actNetworks[index].setNetworkName("Network"+String.valueOf(index));

BackPropagation backpropagation = new BackPropagation(actNetworks[index]);

backpropagation.setLearningRate(learningRate);
backpropagation.setMomentum(momentum);

int solutionSize = sdpairInputData.length - windowSize;
double[] solution = new double[solutionSize];
double[] networkInput = new double[windowSize];

int iteration = 0;
double learningError = 0.0;
double predictionError = 0.0;
double error = 0.0;

while(true)
{
    error = backpropagation.runEpoch(input, output) / samples;

    learningError = 0.0;
    predictionError = 0.0;

    // iterate through data
    for (int i = 0, n = sdpairInputData.length - windowSize; i < n; i++)
    {
        // feed network with normalized input data
        for (int j = 0; j < windowSize; j++)
        {
            networkInput[j] = (sdpairInputData[i + j] - setMin) * factor - 0.85;
        }

        // evaluate
        solution[i] = (actNetworks[index].calculate(networkInput)[0] + 0.85) / factor +
          setMin;
        if (solution[i] < 0.0)
        {
            solution[i] = 0.0;
        }


        if (i >= n - predictionSize)
        {
            predictionError += Math.abs(solution[i] - sdpairInputData[windowSize + i]);
        }
        else
        {
            learningError += Math.abs(solution[i] - sdpairInputData[windowSize + i]);
```

```
                    }
                }

                iteration++;

                if((error <= 0.009) || (iteration > 2000)) {
                    break;
                }

            }

        }

    }

}
```

Neuron.java

```
package timeseriesprediction;

public abstract class Neuron implements java.io.Serializable {

    protected int       inputsCount = 0;
    protected double[]  weights = null;
    protected double    output = 0;

    protected static double rand = Math.random();


    public int getInputsCount() {
        return inputsCount;
    }


    public double getOutput() {
         return output;
    }



    public double getWeight(int index) {
            return weights[index];
    }


    public void setWeight(int index, double value) {
        weights[index] = value;
    }


    public Neuron(int inputs) {
        inputsCount = Math.max(1,  inputs);
        weights = new double[inputsCount];
        randomize( );
    }


    public void randomize() {
        for ( int i = 0; i < inputsCount; i++ ) {
            weights[i] = Math.random();
        }
    }



    public abstract double calculate(double[] input);


}
```

Layer.java

```
package timeseriesprediction;

public abstract class Layer implements java.io.Serializable {

    protected int       inputsCount = 0;
    protected int       neuronsCount = 0;
    protected Neuron[]  neurons;
    protected double[]  output;



    public int getInputsCount() {
        return inputsCount;
    }


    public int getNeuronsCount() {
        return neuronsCount;
    }


    public double[] Output() {
        return output;
    }


    public Neuron getNeuron(int index) {
        return neurons[index];
    }


    protected Layer(int neuronsCount, int inputsCount) {
        this.inputsCount = Math.max(1, inputsCount);
        this.neuronsCount = Math.max(1, neuronsCount);
        neurons = new Neuron[this.neuronsCount];
        output = new double[this.neuronsCount];
    }


    public double[] calculate(double[] input) {
        for (int i = 0; i < neuronsCount; i++) {
            output[i] = neurons[i].calculate(input);
        }
        return output;
    }


    public void randomize() {
        for (Neuron neuron : neurons)
            neuron.randomize();
    }

}
```

Network.java

```
package timeseriesprediction;

public abstract class Network implements java.io.Serializable {

    protected int   inputsCount;
    protected int   layersCount;
    protected Layer[]   layers;
    protected double[]  output;

    public int InputsCount() {
        return inputsCount;
    }

    public int LayersCount() {
        return layersCount;
    }

    public double[] Output() {
        return output;
    }

    public Layer getLayer(int index) {
        return layers[index];
    }


    protected Network(int inputsCount, int layersCount) {
        this.inputsCount = Math.max(1, inputsCount);
        this.layersCount = Math.max(1, layersCount);
        layers = new Layer[this.layersCount];
    }


    public double[] calculate(double[] input) {
        output = input;
        for (Layer layer : layers) {
            output = layer.calculate(output);
        }
        return output;
    }


    public void randomize() {
        for (Layer layer : layers) {
            layer.randomize();
        }
    }

}
```

BipolarSigmoidFunction.java

```
package timeseriesprediction;

public class BipolarSigmoidFunction implements java.io.Serializable {

    private double alpha = 2;



    public double getAlpha() {
        return alpha;
    }


    public void setAlpha(double value) {
        alpha = value;
    }


    public BipolarSigmoidFunction() {

    }


    public BipolarSigmoidFunction(double alpha)
    {
        this.alpha = alpha;
    }


    public double actFunction(double x)
    {
        return ((2 / (1 + Math.exp(-alpha * x))) - 1);
    }


    public double Derivative(double x)
    {
        double y = actFunction(x);

        return (alpha * (1 - y * y) / 2);
    }


    public double Derivative2(double y)
    {
        return (alpha * (1 - y * y) / 2);
    }
}
```

ActivationNeuron.java

```
package timeseriesprediction;

public class ActivationNeuron extends Neuron implements java.io.Serializable {

    protected double threshold = 0.0f;

    protected BipolarSigmoidFunction function = null;


    public double getThreshold() {
        return threshold;
    }

    public void setThreshold(double value) {
        threshold = value;
    }


    public BipolarSigmoidFunction getActivationFunction()
    {
        return function;
    }


    public ActivationNeuron(int inputs, BipolarSigmoidFunction function) {
        super(inputs);
        this.function = function;
    }


    @Override
    public  void randomize() {
        super.randomize();
        threshold = Math.random();
    }


    @Override
    public double calculate(double[] input) {
        // input size not equal to neuronsCount at the input layer
        if (input.length != inputsCount)
            throw new IllegalArgumentException();

        double sum = 0.0;

        // calculate weighted  inputs
        for (int i = 0; i < inputsCount; i++)
        {
            sum += weights[i] * input[i];
        }
        sum += threshold;

        return (output = function.actFunction(sum));
    }

}
```

ActivationLayer.java

```
package timeseriesprediction;

public class ActivationLayer extends Layer implements java.io.Serializable {


    public ActivationNeuron getActivationNeuron(int index) {
        return (ActivationNeuron) neurons[index];
    }



    public ActivationLayer(int neuronsCount, int inputsCount, BipolarSigmoidFunction actfunction) {
        super(neuronsCount, inputsCount);
        for (int i = 0; i < neuronsCount; i++) {
            neurons[i] = new ActivationNeuron(inputsCount, actfunction);
        }
    }
}
```

ActivationNetwork.java

```
package timeseriesprediction;

public class ActivationNetwork extends Network implements java.io.Serializable {

    protected String networkName;



    public String getNetworkName() {
        return networkName;
    }


    public void setNetworkName(String value) {
        networkName = value;
    }

    public  ActivationLayer getActivationLayer(int index) {
        return ((ActivationLayer) layers[index]);
    }


    public ActivationNetwork(BipolarSigmoidFunction actfunction, int inputsCount, int[] neuronsCount)
{
        super(inputsCount, neuronsCount.length);
        // create each layer
        for (int i = 0; i < layersCount; i++)
        {
        layers[i] = new ActivationLayer(
        // neurons count in the layer
        neuronsCount[i],
        // inputs count of the layer
        (i == 0) ? inputsCount : neuronsCount[i - 1],
        // activation function of the layer
        actfunction);
        }
    }
}
```

BackPropagation.java

```
package timeseriesprediction;

public class BackPropagation implements java.io.Serializable {

    private ActivationNetwork network;
    private double learningRate = 0.01;
    private double momentum = 0.0;
    private double[][] neuronErrors = null;
    private double[][][] weightsUpdates = null;
    private double[][] thresholdsUpdates = null;


    public double getLearningRate() {
        return learningRate;
    }

    public void setLearningRate(double value) {
        learningRate = Math.max(0.0,  Math.min(1.0,  value));
    }


    public double getMomentum() {
        return momentum;
    }

    public void setMomentum(double value) {
        momentum = Math.max(0.0,  Math.min(1.0,  value));
    }


    public BackPropagation(ActivationNetwork network) {

        this.network = network;
        neuronErrors = new double[network.LayersCount()][];
        weightsUpdates = new double[network.LayersCount()][][];
        thresholdsUpdates = new double[network.LayersCount()][];

        for (int i = 0, n = network.LayersCount(); i < n; i++)
        {
            Layer layer = network.getLayer(i);
            neuronErrors[i] = new double[layer.getNeuronsCount()];
            weightsUpdates[i] = new double[layer.getNeuronsCount()][];
            thresholdsUpdates[i] = new double[layer.getNeuronsCount()];

            // for each neuron
            for (int j = 0; j < layer.getNeuronsCount(); j++)
            {
                weightsUpdates[i][j] = new double[layer.getInputsCount()];
            }
        }

    }


    public double run(double[] input, double[] output) {
        // compute the network's output
        network.calculate(input);

        // calculate network error
        double error = calculateError(output);

        // calculate weights updates
        calculateUpdates(input);

        // update the network
```

```
    updateNetwork();

    return error;
}


public double runEpoch(double[][] input, double[][] output) {
    double error = 0.0;

    // learning for samples
    for (int i = 0, n = input.length; i < n; i++) {
        error += run(input[i], output[i]);
    }
    return error;
}


private double calculateError(double[] expectedOutput) {
    // current and the next layers
    ActivationLayer layer, layerNext;
    // current and the next errors arrays
    double[] errors, errorsNext;
    // error values
    double error = 0;
    double e;
    double sum;
    // neuron output value
    double output;
    // layers count
    int layersCount = network.LayersCount();

    // get activation function of neurons
    BipolarSigmoidFunction  function =
        network.getActivationLayer(0).getActivationNeuron(0).getActivationFunction();

    // calculate errors
    layer = network.getActivationLayer(layersCount-1);
    errors  = neuronErrors[layersCount - 1];

    for (int i = 0, n = layer.getNeuronsCount(); i < n; i++) {
        output = layer.getActivationNeuron(i).getOutput();
        e = expectedOutput[i] - output;
        errors[i] = e * function.Derivative2(output);
        error += (e * e);
    }

    for (int j = layersCount - 2; j >= 0; j--) {
        layer = network.getActivationLayer(j);
        layerNext = network.getActivationLayer(j+1);
        errors = neuronErrors[j];
        errorsNext = neuronErrors[j + 1];

        // all neurons of the layer
        for (int i = 0, n = layer.getNeuronsCount(); i < n; i++) {
            sum = 0.0;
            // all neurons of the next layer
            for (int k = 0, m = layerNext.getNeuronsCount(); k < m; k++) {

                sum += errorsNext[k] * layerNext.getActivationNeuron(k).getWeight(i);
            }
            errors[i] = sum * function.Derivative2(layer.getActivationNeuron(i).getOutput());
        }
    }
    return error / 2.0;
}


private void calculateUpdates(double[] input) {
```

```
    // current neuron
    ActivationNeuron neuron;
    // current and previous layers
    ActivationLayer  layer, layerPrev;
    // layer's weights updates
    double[][] layerWeightsUpdates;
    // layer's thresholds updates
    double[] layerThresholdUpdates;
    // layer's error
    double[]    errors;
    // neuron's weights updates
    double[] neuronWeightUpdates;
    // error value
    double  error;

    layer = network.getActivationLayer(0);
    errors = neuronErrors[0];
    layerWeightsUpdates = weightsUpdates[0];
    layerThresholdUpdates = thresholdsUpdates[0];

    // update weights of each neuron
    for (int i = 0, n = layer.getNeuronsCount(); i < n; i++) {
        neuron = layer.getActivationNeuron(i);
        error   = errors[i];
        neuronWeightUpdates = layerWeightsUpdates[i];

        for (int j = 0, m = neuron.getInputsCount(); j < m; j++) {
            neuronWeightUpdates[j] = learningRate * (momentum * neuronWeightUpdates[j] + (1.0 -
               momentum) * error * input[j]);
        }
        layerThresholdUpdates[i] = learningRate * (momentum * layerThresholdUpdates[i] + (1.0 -
               momentum) * error);
    }


    for (int k = 1, l = network.LayersCount(); k < l; k++) {
        layerPrev = network.getActivationLayer(k-1);
        layer = network.getActivationLayer(k);
        errors = neuronErrors[k];
        layerWeightsUpdates = weightsUpdates[k];
        layerThresholdUpdates = thresholdsUpdates[k];

        for (int i = 0, n = layer.getNeuronsCount(); i < n; i++) {
            neuron = layer.getActivationNeuron(i);
            error   = errors[i];
            neuronWeightUpdates = layerWeightsUpdates[i];

            for (int j = 0, m = neuron.getInputsCount(); j < m; j++) {
                neuronWeightUpdates[j] = learningRate * (momentum * neuronWeightUpdates[j] + (1.0
                   - momentum) * error * layerPrev.getActivationNeuron(j).getOutput());
            }

            layerThresholdUpdates[i] = learningRate * (momentum * layerThresholdUpdates[i] + (1.0
                   - momentum) * error);
        }
    }
}


// update each neuron in each layer (weights and thresholds)
private void updateNetwork() {
    ActivationNeuron    neuron;
    ActivationLayer     layer;
    double[][]  layerWeightsUpdates;
    double[]    layerThresholdUpdates;
    double[]    neuronWeightUpdates;

    // for each layer
```

```
        for (int i = 0, n = network.LayersCount(); i < n; i++) {
            layer = network.getActivationLayer(i);
            layerWeightsUpdates = weightsUpdates[i];
            layerThresholdUpdates = thresholdsUpdates[i];

            // for each neuron
            for (int j = 0, m = layer.getNeuronsCount(); j < m; j++) {
                neuron = layer.getActivationNeuron(j);
                neuronWeightUpdates = layerWeightsUpdates[j];

                // for each weight
                for (int k = 0, s = neuron.getInputsCount(); k < s; k++) {
                    neuron.setWeight(k, neuron.getWeight(k) + neuronWeightUpdates[k] );
                }

                neuron.setThreshold(neuron.getThreshold() + layerThresholdUpdates[j]);
            }
        }
    }

}
```

TSPrediction.java

```
package timeseriesprediction;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;



public class TSPrediction implements java.io.Serializable {

    ArrayList<ArrayList<Double>> allData  = new ArrayList<ArrayList<Double>>();
    ArrayList<double[]> inputData = new ArrayList<double[]>();
    ArrayList<double[]> totalPrediction = new ArrayList<double[]>();
    ArrayList<double[]> testPrediction = new ArrayList<double[]>();
    public static double[][] prediction;

    int threadNum;

    ArrayList<String> testThreadOrder;

    ActivationNetwork actNetworks[];

    private int iterations = 1000;
    private double learningRate = 0.01;
    private int sigmoid = 2;
    private int windowSize = 5;
    private int predictionSize = 1;
    private double factor;
    private double setMin;
    private double momentum = 0.0;


    public ArrayList<String> testOrder() {
        return testThreadOrder;
    }

    public ArrayList<double[]> getPrediction() {
        return totalPrediction;
    }

    public TSPrediction(ArrayList<ArrayList<Double>> inputData) {
        allData = inputData;
        for(ArrayList<Double> list: allData) {
            Object x = list.toArray();
            this.inputData.add((double[]) x);
        }
    }

    public TSPrediction() {

    }

    public TSPrediction(int threadNum) {
        this.threadNum = threadNum;
    }

    public double getLearningRate() {
        return learningRate;
    }

    public void setLearningRate(double value) {
        learningRate = value;
    }

    public double getMomentum() {
```

```
    return momentum;
}

public void setMomentum(double value) {
    momentum = value;
}

public int getSigmoid() {
    return sigmoid;
}

public void setSigmoid(int value) {
    sigmoid = value;
}

public int getIterations() {
    return iterations;
}

public void setIterations(int value) {
    iterations = value;
}


public ArrayList<double[]> checkPrediction() {
    return this.testPrediction;
}


public void predict(ArrayList<Double[]> input, int predictionSize) {
    int index;
    testThreadOrder = new ArrayList<String>();
    totalPrediction = new ArrayList<double[]>();
    for(index=0; index<input.size(); index++) {
        // forecast horizon
        int PREDICTION_SIZE = predictionSize;
        ArrayList<Double> predictionVector = new ArrayList<Double>();
        ArrayList<Double> normalizedFeeder = new ArrayList<Double>();
        ArrayList<Double> feeder = new ArrayList<Double>();
        double[] predictedValue = new double[1];
        Double sdpairInputData[] = input.get(index);

        // normalization [-0.85, 0.85]
        double maxVal = Collections.max(Arrays.asList(sdpairInputData));
        double minVal = Collections.min(Arrays.asList(sdpairInputData));

        if((maxVal - minVal) == 0) {
            factor = 0;
        }
        else {
            factor = 1.7 / (maxVal - minVal);
        }
        if(factor==0) {
            factor = 1;
        }
        setMin = minVal;


        for (int r = 0; r < sdpairInputData.length; r++) {
            normalizedFeeder.add((sdpairInputData[r] - setMin) * factor - 0.85);
            feeder.add(sdpairInputData[r]);
        }

        for (int l = 0; l < PREDICTION_SIZE; l++) {
            double[] inputVector = new double[normalizedFeeder.size()];

            for (int s = 0; s < inputVector.length; s++) {
                inputVector[s] = normalizedFeeder.get(s);
```

```
            }

            predictedValue = actNetworks[index].calculate(inputVector);

            predictedValue[0] = (predictedValue[0] + 0.85) / factor + setMin;

            if(predictedValue[0] < 0) {
                predictedValue[0] = 0.0;
            }
            predictionVector.add(predictedValue[0]);
            normalizedFeeder.remove(0);
            normalizedFeeder.add((predictedValue[0] - setMin) * factor - 0.85);
            feeder.remove(0);
            feeder.add(predictedValue[0]);

            maxVal = Collections.max(feeder);
            minVal = Collections.min(feeder);

            if((maxVal - minVal) == 0) {
                factor = 0;
            }
            else {
                factor = 1.7 / (maxVal - minVal);
            }
            if(factor==0) {
                factor = 1;
            }
                setMin = minVal;
        }

        double[] predictionArray = new double[PREDICTION_SIZE];
        for (int s = 0; s < predictionArray.length; s++) {
            predictionArray[s] = predictionVector.get(s);
        }
        totalPrediction.add(predictionArray);
    }
}


public void train(ArrayList<Double[]> inputSet) {
    int index ;
    testThreadOrder = new ArrayList<String>();
    actNetworks = new ActivationNetwork[inputSet.size()];
    for(index=0; index<threadNum; index++) {

        Double sdpairInputData[] = inputSet.get(index);
        double maxVal = Collections.max(Arrays.asList(sdpairInputData));
        double minVal = Collections.min(Arrays.asList(sdpairInputData));
        // fix normalization
        if((maxVal - minVal) == 0) {
            factor = 1;
        }
        else {
            factor = 1.7 / (maxVal - minVal);
        }
        setMin = minVal;

        int samples = sdpairInputData.length - predictionSize - windowSize;

        double[][] input = new double[samples][];
        double[][] output = new double[samples][];

        for (int i = 0; i < samples; i++)
        {
            input[i] = new double[windowSize];
            output[i] = new double[1];

            // set input
```

```
    for (int j = 0; j < windowSize; j++)
    {
        input[i][j] = (sdpairInputData[i + j] - setMin) * factor - 0.85;
    }

    if(factor==0) {
        factor = 1;
    }
    // set output
    output[i][0] = (sdpairInputData[i + windowSize] - setMin) * factor - 0.85;
}

int neuronCount[] = new int[2];
neuronCount[0] = windowSize*2;
neuronCount[1] = 1;

actNetworks[index] = new ActivationNetwork(new BipolarSigmoidFunction(sigmoid),
    windowSize, neuronCount);
actNetworks[index].setNetworkName("Network"+String.valueOf(index));

BackPropagation backpropagation = new BackPropagation(actNetworks[index]);

backpropagation.setLearningRate(learningRate);
backpropagation.setMomentum(momentum);

int solutionSize = sdpairInputData.length - windowSize;
double[] solution = new double[solutionSize];
double[] networkInput = new double[windowSize];

int iteration = 0;
double learningError = 0.0;
double predictionError = 0.0;
double error = 0.0;

while(true)
{
    error = backpropagation.runEpoch(input, output) / samples;

    learningError = 0.0;
    predictionError = 0.0;

    // iterate through data
    for (int i = 0, n = sdpairInputData.length - windowSize; i < n; i++)
    {
        // feed network with normalized input data
        for (int j = 0; j < windowSize; j++)
        {
            networkInput[j] = (sdpairInputData[i + j] - setMin) * factor - 0.85;
        }

        // evaluate
        solution[i] = (actNetworks[index].calculate(networkInput)[0] + 0.85) / factor +
               setMin;
        if (solution[i] < 0.0)
        {
            solution[i] = 0.0;
        }


        if (i >= n - predictionSize)
        {
            predictionError += Math.abs(solution[i] - sdpairInputData[windowSize + i]);
        }
        else
        {
            learningError += Math.abs(solution[i] - sdpairInputData[windowSize + i]);
        }
    }
```

```
                iteration++;

                if((error <= 0.009) || (iteration > 2000)) {
                    break;
                }

            }

        }

    }

}
```

LPACL Project

LP.java

```
package autoflow.acl.lp;

import gr.tns.RestUtil;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URI;
import java.net.URISyntaxException;
import java.sql.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.List;
import java.util.Locale;
import umf.common.SysUtil;
import umf.common.gov.IGovernance;
import umf.common.nem.NEMSkin;
import autoflow.acl.lp.monitor.Monitor;
import autoflow.acl.lp.monitor.DataUpdator;
import autoflow.acl.lp.network.Network;
import cern.colt.matrix.DoubleMatrix1D;
import cern.colt.matrix.impl.DenseDoubleMatrix1D;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.io.PrintWriter;
import javax.swing.JOptionPane;
import timeseriesprediction.*;


public class LP extends AbstractLP {

    public static Network netObj = new Network();
    public static Monitor mon = new Monitor();
    public static int cyc=0;

    TSPrediction tsp ;

    public static String inputStr;
    ArrayList<ArrayList<Double>> allData  = new ArrayList<ArrayList<Double>>();
    public static ArrayList<Double[]> trainingSet;
    public static ArrayList<Double[]> inputSet;


    public static String staticFilename;
    public static String filePath;
    public static Boolean trained = true;
    public static Boolean finishedTraining = false;

    public static Boolean preTrainedNetworks = true; //change it to "false" for runtime training
    public static Boolean start = true;
```

```
public static void main(String[] args) throws IOException {
    LP acl;
    try {
        acl = NEMSkin.createBind(LP.class,
                initManifest(LP.class),
                SysUtil.nextAvailableTCPPort(8099));
        IGovernance gov = RestUtil.getHTTPProxy("http://localhost:7777/",
                IGovernance.class);
        gov.onNEMLoaded(acl.getBaseURI());
    } catch (ClassCastException e) {
        e.printStackTrace();
    } catch (URISyntaxException e) {
        e.printStackTrace();
    }

}

/**
 * @param manifestUri
 * @throws ClassCastException
 * @throws FileNotFoundException
 * @throws MalformedURLException
 */
public LP(URI manifestUri) throws ClassCastException,
        FileNotFoundException, MalformedURLException {
    super(manifestUri);
}

@Override
protected void monitor() {
    // check if a file with serialized neural networks exist for the current day
    if(start) {
        Calendar cal = Calendar.getInstance();
        String today = new SimpleDateFormat("EEEE", Locale.ENGLISH).format(cal.getTime());
        //JOptionPane.showMessageDialog(null, today);
        filePath = "C:\\eclipse\\workspace\\LPACL\\data\\" + today + ".ser";
        File f = new File(filePath);
        if(f.exists()) {
            JOptionPane.showMessageDialog(null, "Neural network loaded, starting prediction...");
        }
        else {
            int selectedOption = JOptionPane.showConfirmDialog(null,
                    "A neural network for " + today + " was not found. Click YES to train a new
                       network or No " + "to proceed with real-time prediction", "Choose",
                    JOptionPane.YES_NO_OPTION);
            if (selectedOption == JOptionPane.YES_OPTION) {
                preTrainedNetworks = true;
                trained = false;
            }
            else {
                preTrainedNetworks = false;
            }
        }
        start = false;
    }

    // train neural network based on actual load
    if(preTrainedNetworks && !trained && !finishedTraining) {

        tsp = new TSPrediction(20);

        try {
            PrintWriter writer = new
               PrintWriter("C:\\eclipse\\workspace\\LPACL\\data\\training.csv");
```

```
            writer.print("");
            writer.close();

            for(int i=0; i<300; i++) {
            // Monitor
                mon.operation(netObj, super.nbi, super.flowSpace);
                //Thread.sleep(1000);
                // update data about SD pairs (newData) based on monitoring
                DataUpdator.operation(netObj, true);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        System.out.println();

    }
    // perform runtime prediction by training neural network in each loop
    else if (!preTrainedNetworks){
        tsp = new TSPrediction(20);
        try {
            PrintWriter writer = new
                PrintWriter("C:\\eclipse\\workspace\\LPACL\\data\\training.csv");
            writer.print("");
            writer.close();

            for(int i=0; i<50; i++) { //50 load values, training set
            // Monitor
                mon.operation(netObj, super.nbi, super.flowSpace);
                Thread.sleep(1000);
            // update data about SD pairs (newData) based on monitoring
                DataUpdator.operation(netObj, true);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        System.out.println();
    }

}

@Override
protected void analyze() {
    // read training data from file
    if((!trained && !finishedTraining) || !preTrainedNetworks) {

        String inputFile = "C:\\eclipse\\workspace\\LPACL\\data\\training.csv";
        BufferedReader fileReader = null;


        final String DELIMITER = ";";
        try
        {
            String line = "";
            StringBuilder sb = new StringBuilder();

            for(int i=0; i<20; i++) {
                allData.add(new ArrayList<Double>());
            }
            //create the file reader
            fileReader = new BufferedReader(new FileReader(inputFile));

            //read the file line by line
            while ((line = fileReader.readLine()) != null)
            {

                //get all tokens available in line
```

```
                String[] tokens = line.split(DELIMITER);

                for(int j=0; j<tokens.length; j++) {
                    Double readVal = Double.parseDouble(tokens[j]);
                    if ((readVal.isNaN()) || (readVal.isInfinite())) {
                        readVal = 0.0;
                    }
                    allData.get(j).add(readVal);
                }
            }

        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally
        {
            try {
                fileReader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        System.out.println();


        // train neural networks
        trainingSet = new ArrayList<Double[]>();

        try {
            for(ArrayList<Double> list: allData) {
                Double x[] = new Double[list.size()];
                for(int k=0; k<list.size(); k++) {
                    x[k] = list.get(k);
                }

                trainingSet.add(x);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }

        tsp.setLearningRate(0.01);
        tsp.setMomentum(0.0);
        tsp.setSigmoid(2);
        tsp.train(trainingSet);
        System.out.println();


    }
}

@Override
protected void plan() {
    // serialize trained neural networks
    if((!trained && !finishedTraining) && preTrainedNetworks) {
        try {

            FileOutputStream fout = new FileOutputStream(filePath);
            ObjectOutputStream oos = new ObjectOutputStream(fout);
            //JOptionPane.showMessageDialog(null, "Starting serialization...");
            oos.writeObject(tsp);
            oos.close();
            fout.close();
            //JOptionPane.showMessageDialog(null, "Finished serialization.");
```

```
        }
        catch(IOException e) {
            e.printStackTrace();
        }
        finishedTraining = true;
        //System.out.println();
    }
}

@Override
protected void execute() {
    // runtime monitoring and prediction
    if(trained || !preTrainedNetworks) {
        try {
            String inputFile = "C:\\eclipse\\workspace\\LPACL\\data\\input.csv";
            BufferedReader fileReader = null;
            final String DELIMITER = ";";

            PrintWriter writer = new
                 PrintWriter("C:\\eclipse\\workspace\\LPACL\\data\\input.csv");
            writer.print("");
            writer.close();

            int it=0;
            while(it<5) {
                mon.operation(netObj, super.nbi, super.flowSpace);
                //Thread.sleep(1000);
                //update data about SD pairs (newData) based on monitoring
                if(DataUpdator.operation(netObj, false)) {
                    it++;
                }
            }
            allData = new ArrayList<ArrayList<Double>>();
            String line = "";

            for(int i=0; i<20; i++) {
                allData.add(new ArrayList<Double>());
            }

            fileReader = new BufferedReader(new FileReader(inputFile));

            //read the file line by line
            while ((line = fileReader.readLine()) != null) {

                //get all available tokens in line
                String[] tokens = line.split(DELIMITER);


                for(int j=0; j<tokens.length; j++) {
                    Double readVal = Double.parseDouble(tokens[j]);
                    if ((readVal.isNaN()) || (readVal.isInfinite())) {
                        readVal = 0.0;
                    }
                    allData.get(j).add(readVal);
                }
            }
            fileReader.close();

            // prepare input set in the correct format
            inputSet = new ArrayList<Double[]>();
            for(ArrayList<Double> list: allData) {
                Double x[] = new Double[list.size()];
                for(int k=0; k<list.size(); k++) {
                    x[k] = list.get(k);
                }
                inputSet.add(x);
            }
        }
```

```
            catch(Exception e) {
                e.printStackTrace();
            }

            try {
                ArrayList<double[]> prediction = new ArrayList<double[]>();
                // load serialized neural networks for prediction
                if(preTrainedNetworks) {
                    TSPrediction tsprediction = new TSPrediction();
                    try {

                        FileInputStream fin = new FileInputStream(filePath);
                        ObjectInputStream ois = new ObjectInputStream(fin);
                        tsprediction = (TSPrediction) ois.readObject();
                        ois.close();
                    }
                    catch(Exception e) {
                        e.printStackTrace();
                    }

                    tsprediction.predict(inputSet,  5);
                    prediction = tsprediction.getPrediction();
                }
                else {
                    // prediction with runtime training
                    tsp.predict(inputSet, 55);
                    prediction = tsp.getPrediction();
                }

                // write predicted values to csv file
                FileWriter predictFR = new
                    FileWriter("C:\\eclipse\\workspace\\LPACL\\data\\prediction.csv", true);
                BufferedWriter out = new BufferedWriter(predictFR);
                StringBuilder sb = new StringBuilder();
                String newLineFields[] = new String[prediction.size()];
                for(int i=0; i<prediction.get(0).length; i++) {
                    for(int j=0; j<prediction.size(); j++) {
                        double temp[] = new double[prediction.size()];
                        temp =   prediction.get(j);
                        newLineFields[j] = String.valueOf(temp[i]);
                    }
                    sb.append(String.join(";", newLineFields));
                    sb.append("\n");

                }
                out.write(sb.toString());
                out.close();
            }
            catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

DataUpdator.java

```
package autoflow.acl.lp.monitor;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.text.NumberFormat;
import java.util.Calendar;
import autoflow.acl.lp.network.Network;
import autoflow.acl.lp.network.SDPair;
import autoflow.acl.lp.network.SDPairValue;
import autoflow.acl.lp.network.SDPairValueAssigner;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Locale;
import java.util.Map;
import java.util.Vector;
import java.io.FileOutputStream;
import java.io.File;
import autoflow.acl.lp.network.SDPair;
import autoflow.acl.lp.network.SDPairFlow;

// Updates data for SOM operation, creates the newData file, ... feed SOM
public class DataUpdator {

    public DataUpdator() {

    }


    public static boolean operation(Network netObj, boolean training)
    {
        FileWriter fstream;
        FileWriter fstream3;
        BufferedWriter out3;
        BufferedWriter out;

        FileWriter fstream2;
        BufferedWriter out2;
        Calendar cal = Calendar.getInstance();
        HashMap<Integer, Double> activeSDpairs = new HashMap<Integer, Double>();

        LinkedHashMap<String, Double> SDpairsLoad = new LinkedHashMap<String, Double>();

        boolean allZeros = true;

        // initialize a linked hash map with sdpair load
        // key is the last digits of source-destination IP addresses
        for(int i=1; i<6; i++) {
            for(int j=1; j<6; j++) {

                if(j!=i) {
                    String key = String.valueOf(i) + String.valueOf(j);
                    SDpairsLoad.put(key, 0.0);

                }
            }
        }
```

```
// check active SDpairs from network object
// update load hash map according to IP addresses
HashSet<SDPairValue> sdpairvalues = netObj.getSdpValAss().getSDPairValues();
Iterator<SDPairValue> sdpviter = sdpairvalues.iterator();
SDPairValue sdpv = new SDPairValue();
if(sdpairvalues.size()!=0) {

    while(sdpviter.hasNext()) {
        sdpv = sdpviter.next();
        String src = sdpv.getSrcDPID();
        String dest = sdpv.getDstDPID();
        String sdp = src.substring(src.length()-1) + dest.substring(dest.length()-1);

        Double trafficDemand = 0.0;
        Vector<Integer> uniqueVal = sdpv.getValue();
        for(int p=0; p<netObj.getSdpairs().size(); p++) {
            if (netObj.getSdpairs().elementAt(p).getUniqueVal().equals(uniqueVal)) {
                trafficDemand = netObj.getSdpairs().elementAt(p).getTrafficDemand();
                if(trafficDemand.isNaN() || trafficDemand.isInfinite()) {

                    return false;
                }
                if(trafficDemand > 0.0) {
                    allZeros = false;
                }
                break;
            }
        }

        if(allZeros) {
            return false;
        }

        SDpairsLoad.replace(sdp,  trafficDemand/1024); //kbps

        System.out.println();
    } //end while

} //end if



Iterator<String> loadIter = SDpairsLoad.keySet().iterator();
String next = loadIter.next();


while(loadIter.hasNext()) {
    next = loadIter.next();
}

// write SDpair load to files in a certain order according to source-destination IPs
try {
    if(training) {
        fstream = new FileWriter("C:\\eclipse\\workspace\\LPACL\\data\\training.csv", true);
    }
    else {
        fstream = new FileWriter("C:\\eclipse\\workspace\\LPACL\\data\\input.csv", true);
    }

    fstream3 = new FileWriter("C:\\eclipse\\workspace\\LPACL\\data\\alltraffic.csv", true);
    out = new BufferedWriter(fstream);
    out3 = new BufferedWriter(fstream3);


    Iterator<String> filewriter = SDpairsLoad.keySet().iterator();
    DecimalFormatSymbols otherSymbols = new DecimalFormatSymbols(Locale.US);
    otherSymbols.setDecimalSeparator('.');
```

```
            DecimalFormat df = new DecimalFormat("#.000", otherSymbols);

            while(filewriter.hasNext()) {
                String key = filewriter.next();
                out.write(df.format(SDpairsLoad.get(key)).toString());
                //omit last delimiter - 54 is the last SDpair
                if(!key.equals("54")) {
                    out.write(";");
                }
                out3.write(df.format(SDpairsLoad.get(key)).toString());
                if(!key.equals("54")) {
                    out3.write(";");
                }
            }

            out.write("\n");
            out3.write("\n");

            out.close();

            out3.close();

        }
        catch(Exception e) {
            System.err.println("Error: " + e.getMessage());
        }

        return true;
    }

}
```