SAPIENZA
UNIVERSITÀ DI ROMA

Faculty of Information Engineering, Computer Science and Statistics

Department of Computer, Control and Management Engineering

Master of Science Degree Thesis in
Engineering in Computer Science

January 2014

An Architecture for Automatic Scaling of Replicated Services: Proactive Approach

Candidate: Federico Lombardi

Advisor: Silvia Bonomi
Assistant Advisor: Leonardo Aniello

Academic Year
2012-2013

To my Mom . . .

Acknowledgements

I believe that the greatest thing in achieving a goal is to share it with people you love. I really wish to thank my family with all my heart, especially my daddy, my grandparents and Benedetta for always supporting me and helping me to choose the right way. Above all for being there in one of the hardest periods of my life. A special thanks goes to Silvia and Leo, who introduced me to the world of research projects and guided me through out my thesis. Thank you for being always available and for your friendship. Thanks to Alessandro, my fellow student with whom I shared ours days and nights of hard work. At last but not at least, my thanks goes to all my good friends, who make my everyday life easier to enjoy.

"I propose to consider the question, 'Can machines think?' "
A. M. Turing

ABSTRACT

Il recente sviluppo di fenomeni quali social networks, smartphones e internet of things ha incrementato un bisogno di servizi web con elevatissimi livelli di availability e tempi di risposta contenuti con la conseguente diffusione di servizi di Cloud Computing. Il calcolo si è spostato sempre di piú da sistemi centralizzati a sistemi distribuiti e replicati. La replicazione di un servizio consente di mantenere un alto livello di performance anche a fronte di carichi più elevati, offrendo cosí tempi di risposta piú veloci rispetto ad un servizio non replicato quando il carico in ingresso é tale da saturare quest'ultimo. La replicazione offre inoltre una migliore disponibilitá di risorse ed una maggiore tolleranza ai guasti. Se da un lato un sistema replicato offre un miglior servizio, dall'altro lato comporta maggiori costi dovuti al maggior consumo di risorse. Un numero statico e prefissato di serventi attivi puó essere insufficiente a gestire situazioni di traffico elevato, ma un overprovisioning puó comportare costi eccessivi. Riconfigurare un sistema (ossia cambiare il numero di serventi attivi) a runtime é fondamentale per avere contemporaneamente un alto livello di servizio limitando il consumo di risorse. I servizi di riconfigurazione automatica sono chiamati "Automatic Scaling", e lo scopo di questo lavoro é la realizzazione di MYSE, un framework di autoscaling proattivo. MYSE prendendo in input una soglia di Quality of Service (QoS), ha il compito di trovare il minimo numero di repliche capaci a garantire tempi di servizio sotto la soglia QoS desiderata. Il servizio replicato è stato modellato come un sistema di code con s serventi così da poter stimare i tempi di servizio di una richiesta. La realizzazione di un sistema di previsioni mediante l'uso di reti neurali artificiali ci ha permesso di prevedere i livelli attesi di traffico per poter attivare o disattivare un opportuno numero di serventi prima del verificarsi della variazione del traffico. In questo modo sono stati limitati i transitori in cui si sarebbero subite le conseguenze negative dell'over e dell'under provisioning. Test di confronto sull'accuratezza tra il nostro approccio ed uno reattivo (real time senza previsioni) hanno mostrato come la proattività possa realmente portare benefici, poichè l'apprendimento di un pattern e la conseguente previsione del traffico, consente di avere un errore più basso di quello ottenuto con un approccio reattivo.

Contents

Introduction

The recent increase of phenomena like social networks, smartphones and internet of things is fostering web services with high levels of availability and fast response times that in turn require more computational power, hence the tendency to employ Cloud Computing systems. This kind of approach change the centrality of computation to a distributed one which could give some benefit in term of dependability. High levels of reliability and availability are today required for any application providing services to clients as they usually expect to be able to get a fast service access, otherwise negative consequences are likely to arise.

As an example, when clients are human operators interacting with the service through a visual interface (i.e. a user browsing a web site), large delays can make them look for another service provider, which leads to a drop in the number of customers and therefore to money loss. When clients are represented by other applications communicating with the service through synchronous or asynchronous messages (i.e. in business workflows), serving requests at low rates can adversely have an impact on the performance and on the correctness of served applications, which again can cause troubles to the service provider, for example due to the penalties provided for Service Level Agreement (SLA) violations.

As a consequence, in addition to the classical availability property, taking explicitly in to account the need to ensure a certain degree of Quality of Service (QoS) is becoming mandatory for any dependable application, but at the same time it is important to save resources and maintain low the costs.

Scenario: Scalable Replicated Services

The replication of a service allows to maintain a high level of performance even with heavy loads and offers faster response times compared to a service that is not replicated when the input load is such as to saturate the latter, while also offering a better availability of resources and greater fault tolerance. While designing a replicated service to deliver target response time for a fixed workload is easily achievable by properly tuning the number and the specifics of replicas, it becomes really challenging for highly variable loads. Methods based on *over-provisioning* allow to cope with load peaks but entail huge waste of money due to resource waste. Nevertheless *under-provisioning* systematically fails in delivering required performance when input spikes occur. The actual alternative to static provisioning is rendering the service *elastic*, so that it can adapt to fluctuating workloads by changing the number of replicas (*configuration*) on the fly. Such a functionality is called *auto scaling*. Amazon Web Services [4] and Google App Engine [7] are among the most relevant XaaS providers offering the possibility to reconfigure at runtime. Both allow to define policies that trigger a reconfiguration on the basis of the variation of a set of off-the-shelf and custom metrics, like memory usage and CPU utilization.

This kind of solutions has two main issues. One concerns the difficulty to find an accurate relationship between the value of monitored metrics and the configuration required to meet latency requirements. The other regards the timeliness in reacting to load variations: spotting a problematic situation when it is already occurring brings temporary Quality of Service (QoS) violations, while trying to guess a forthcoming overload could mean taking unnecessary countermeasures in advance, which implies the same problems of over-provisioning.

Our Solution: MYSE Framework

We propose a solution aimed at facing these two issues. We designed the *Make Your Service Elastic* (MYSE), a Framework for auto scaling a replicated service. By monitoring input requests patterns and the service times delivered by the replicated service, MYSE learns over time through neural networks how input load and service times vary, and produces estimations to enable early decisions about reconfiguration. A queuing model of the replicated service is used to compute the expected response time given the configuration and the distributions of both input requests and service times. A heuristic is employed that leverages this model to find the minimum number of replicas required to achieve the target performance.

Contributions

Within the context of MYSE Framework introduced earlier, my contribution has been:

- The design of the MYSE Framework architecture and its interaction with the replicated target service modeled as a queueing model with a single queue and s servers. In particular the design, implementation and testing of the following modules:
  1. The Δ-*Forecasters*, three modules able to predict Load, Distribution and Service Time of the system by learning the history of requests
  2. The *Decider*, a module which aims to choose the configuration by employing a heuristic based on optimum number of servers calculated on the basis of the expected latencies provided by the queueing model
- Making a comparison between my proactive MYSE and a reactive MYSE version, introducing the benefits that the forecasters provide to the framework during traffic changes

Work Organization

The thesis is organized as follows:

- Chapter 1 presents an overview of related works on automatic scaling of Cloud systems, a brief introduction on machine learning techniques and related works with forecasting;
- Chapter 2 describes the MYSE Framework architecture with a work description of each single module. The chapter describes also the replicated service modeled by the Queueing Theory in order to estimate the latency of each request;
- Chapter 3 explains the choice of neural networks and contains the design with the relative tests made to obtain the best Forecasters configuration;
- Chapter 4 describes the evaluation of Forecasters impact on replicas number with respect to the optimum and a comparison between the reactive and the proactive version of MYSE;
- Chapter 5 finally explains the conclusions that outlines how the work is going to continue.

Chapter 1

Related Works

1.1 Reconfiguration of Cloud Services: Automatic Scaling

The most common platforms in Cloud computing as Amazon Web Services [4], Google AppEngine [7], Microsoft Azure [8] offer different services for monitoring, managing and provisioning service and resources. These platforms can be integrated by modules which main works are Load Balancing, Provisioning and Auto Scaling. For example, Amazon Web Services (AWS) uses Elastic Load Balancer [2] for automatically provisions incoming workload of applications across the available Amazon Elastic Compute Cloud (EC2) instances [3], Auto Scaling to scale-in or scaling-out the number of active Amazon EC2 instances and CloudWatch [1] for strategic decision basing on the real-time aggregated resources and service performance informations.

Many works currently aimed to negotiating and enforcing QoS and SLAs [62] in grid and cloud computing scenario. Rodero-Merino et al. [56] has proposed an abstraction layer for cloud systems and one implementation of this layer (Caludia) which offers an automatic scaling service based on scalability rules defined by the Service Provider (SP). Our approach is policy-free and without any rules.

Ming Mao et al. [42] has presented a Cloud autoscaling based on deadline and budget constraints, by modeling the problem as an optimization problem and solving it with integer programming. Other works use information about utilization of the resources like InterCloud [21].

Astrova et al. [13] has proposed a mechanism for automatic scaling of complex event processing applications within an IaaS infrastructure. In their work, the provisioning of a new resource is triggered by the detection of a critical workload, such as the CPU utilization exceeding a predefined threshold.

Xiao et al. [73] has reframed the automatic scaling of an application as a Class Constrained Bin Packing (CCBP) problem where each resource is a bin and each class represents an application. They monitor CPU and memory to detect overloads, and need to know in advance the CPU demand of each considered application, which is not an information easy to estimate.

Costache et al. has presented Themis [25], a system based on a spot market for efficiently managing resources and meeting application requirements. Within this scenario, automatic scaling is performed when monitored performance metrics exceed some fixed thresholds, and application specifics have to include a model that correlates application progress rate to allocated CPU resources.

Many other works on automatic scaling trigger reconfigurations upon the detection of overloads and underutilization, basically based on the monitoring of standard metrics like CPU utilization and memory usage. A nice survey on failure prediction techniques can be found in [60].

Beltrán et al. [16] has employed a model for response time prediction based on the DYPAP model, which needs updated CPU usage statistics to verify whether current configuration can meet QoS requirements, and to consequently scale in/out. Upon the arrival of a new request, a decision has to be made on where to dispatch it while still meeting QoS requirements, and the system has to be scaled out in case the DYPAP model predicts that available resources are not enough. Our solution employs a more generic model than DYPAP, and in addition relies on traffic forecasts.

SmartScale [30] is a framework for automatic scaling that leverages input throughput predictions (based on time series analysis) and workload classifications (based on k-means clustering) to take decisions about requesting/releasing resources and dispatching/shedding input requests. It uses an analytical model for estimating resource usage and SLA penalty costs, which are then used to take the proper action. It uses an analytical model for estimating resource usage and SLA penalty costs, which are then used to take the proper action. Our work is more focused on scaling rather than dispatching, and the underlying models used to reconfigure are quite different.

TAS [28] is a system for automating scaling of in-memory transactional data grids. The core of this work is a Performance Predictor module that combines an analytical model (for forecasting performance) and machine learning techniques (for predicting the effects of resource contention on performance) to provide estimates that are then used for scaling according to given SLA requirements. Besides employing different analytical methods, our work is not specific to transactional environments but embraces a wider scope.

1.2 Machine Learning Techniques

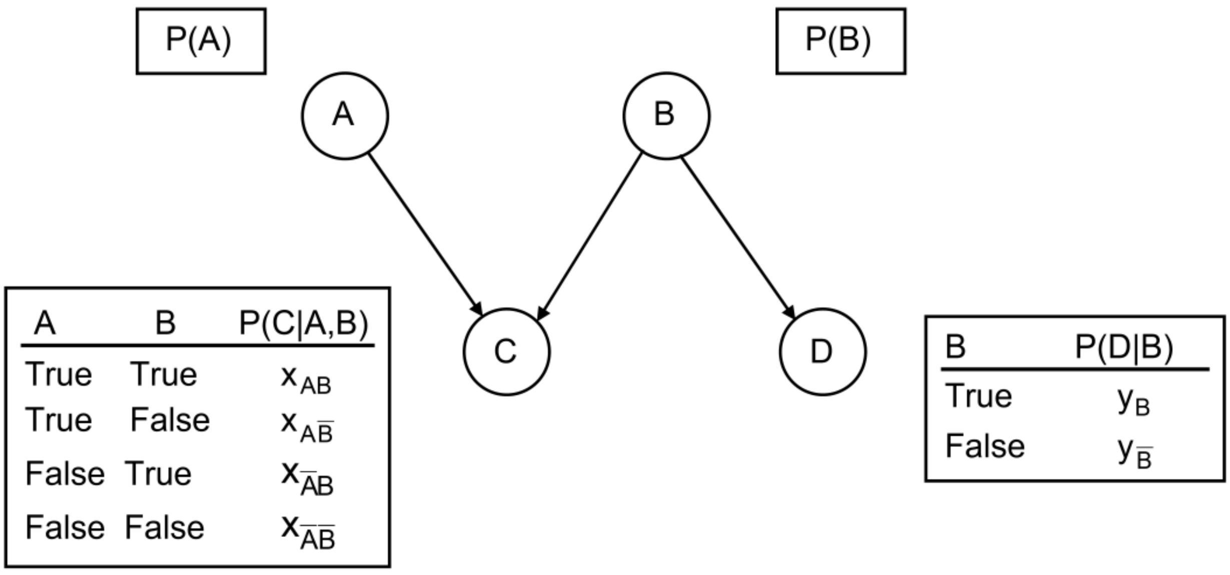
Machine Learning is a branch of Artificial Intelligence that concerns the construction and study of systems that can learn from data. The first step is training the system and then it can be used to classify new data [11, 59].
The core of machine learning deals with *representation* and *generalization*. Representation of data instances and functions evaluated on these instances are part of all machine learning systems. Generalization is the property that the system will perform well on unseen data instances; the conditions under which this can be guaranteed are a key object of study in the sub-field of computational learning theory.
A core objective of a learner is to generalize from its experience. A learning machine is able to perform accurately on new, unseen examples/tasks after having experienced a learning data set. The training examples come from some generally unknown probability distribution (considered representative of the space of occurrences) and the learner has to build a general model about this space that enables it to produce sufficiently-accurate predictions in previously-unseen cases.
There exist many different algorithm types and data structures for making predictions. The most common algorithms is:

- *Supervised Learning*: algorithms aim to generalize a function or to create a map input-output by training on **labelled** examples (i.e. input where the desired output is known) which can be used for making prediction for previously unseen input.
- *Unsupervised Learning*: algorithms trained on **unlabelled** examples (i.e. input where the desired output is unknown) aim to discover structures in the data (e.g. cluster analysis).
- *Semi-Supervised Learning*: combination of both *Supervised* and *Unsupervised* approaches to generate a classifier.
- *Reinforcement Learning*: algorithms based on action-reward which have to maximize a cumulative reward.

Many other kind of algorithms (e.g. *Transduction*, *Learning to Learn*, *Developmental Learning*, etc.) are available for various scopes and there exist also many approaches which use different data structures. Making a combination of Algorithms and Approaches they can reach different goals. All Machine Learning theory is based on statistical computation.

We brief the most important Machine Learning techniques [10, 18].

The **Bayesian Network** is one of the most known models and it is a statistical model employed as a directed acyclic graph that represents a set of random variables with their conditional dependencies. In particular the nodes of graph represented the stochastic variables (or random variables), while the edges between two nodes represent the conditional dependencies of the relative stochastic variables. If two nodes are not connected means that they represent two variables that are conditional independent of each other. A probability function is associated to each nodes and this function takes as input a set of values for the node's parent variables (below we describe the meaning of parent node in *Graph Theory*) and the output is the probability of the variable represented by the node. In *Graph Theory* a *parent* is a node u which has a directed edge from itself to a node v and such node is called *child* [69].



A	B	P(C|A,B)
True	True	x_{AB}
True	False	$x_{A\overline{B}}$
False	True	$x_{\overline{A}B}$
False	False	$x_{\overline{A}\overline{B}}$

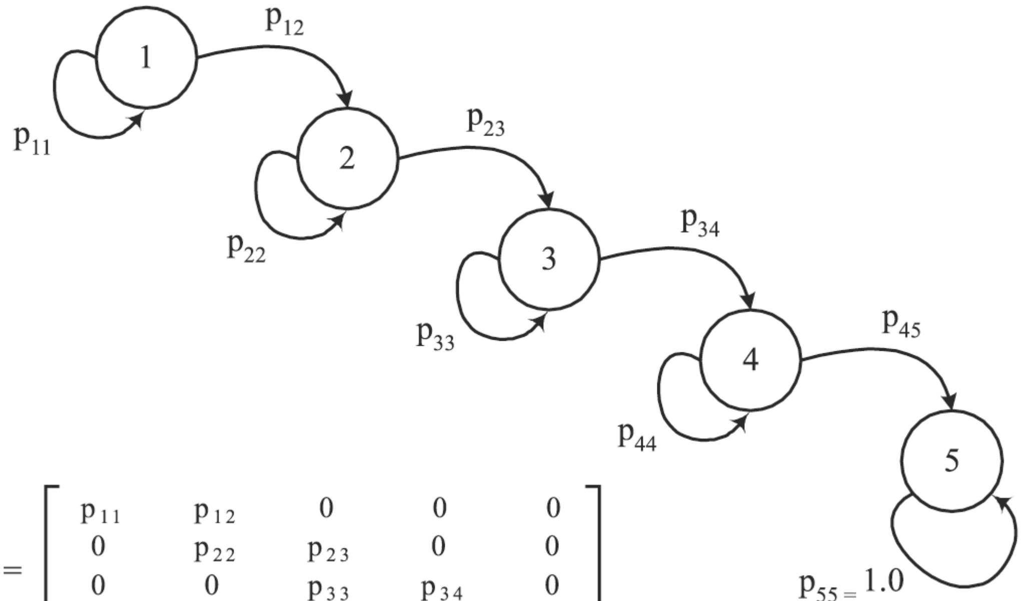
B	P(D|B)
True	y_B
False	$y_{\overline{B}}$

Figure 1.1: Examples of Bayesian Network with 4 nodes and 3 edges

In the figure 1.1 there is an example of Bayes Network with a graph $G = (V, E)$, $V = \{A, B, C, D\}$, $E = \{AC, BC, BD\}$, so the node C depends from A and C, while the node D depends only from B.

The **Markov Network** (also called Markov Random Field) is a similar model but with an undirected graph and it is able to represent some dependencies that a Bayesian Network cannot (e.g. dependencies due to cyclic), but on the other hand it cannot represent some dependencies (as e.g. induced dependencies) that Bayesian Network is able to and for both of these models there exist many efficient algorithms to perform inference and learning.

A Markov Network is a generalization of **Markov Chain** in multiple dimension, in fact while in a Markov Chain the state depends only on the previous one, in a Markov Network each state depends on its neighbors in any of multiple directions.



$$P = \begin{bmatrix} p_{11} & p_{12} & 0 & 0 & 0 \\ 0 & p_{22} & p_{23} & 0 & 0 \\ 0 & 0 & p_{33} & p_{34} & 0 \\ 0 & 0 & 0 & p_{44} & p_{45} \\ 0 & 0 & 0 & 0 & 1.0 \end{bmatrix}$$

Figure 1.2: Examples of Markov Chain

The figure 1.2 show an example of Markov Chain and its probability matrix. Other Markov models are available and these are pretty similar, but there exist some differences due to the visibility of the states. By explaining other models we understand better such differences.

The **Hidden Markov Model** (HMM) is the simplest implementation of a **Dynamic Bayesian Network** which is a Bayesian Network that models a sequence of variables. As mentioned before, the difference between other Markov models is the visibility of the states. While in a Markov Chain the state is directly visible to an observer and so the state transition probabilities are the unique parameters, in a HMM only the output, which is dependent on the state, is visible, but the state is not. This model is called *hidden* is for the state sequence and not for the model parameters which could be known. The sequence of the state can be discovered by the sequence of tokens generated by an HMM, having each state a probability distribution over the possible output tokens.

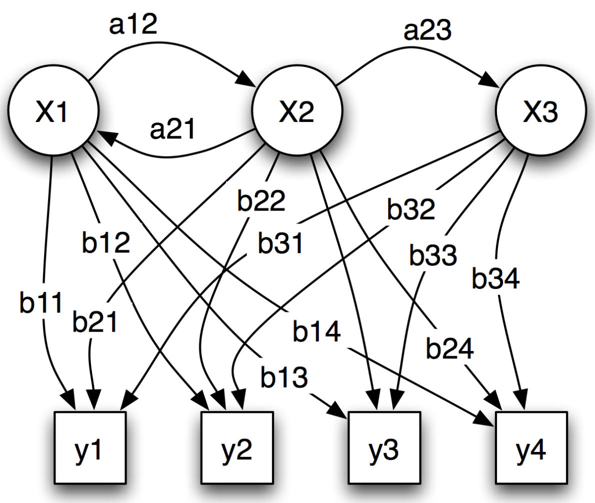


Figure 1.3: Examples of Hidden Markov Model

The figure 1.3 shows an example of HMM where x_i are the states, y_j the possible observations, $a_{s,t}$ the state transition probabilities and $b_{l,k}$ the output probabilities. One of the main applications in which the Hidden Markov models are useful is the Temporal Pattern Recognition. For the same scope are also used other approaches which uses the conditional probability e.g. one of them is the **Bayes Classifier** or many other different models are available as the **Decision Tree** which use a tree structure for taking a decision taking as input a training set with a *Supervised* approach, by using a *Gain Information* depending from the Entropy of a variable in order to choose the most significant sequence of attribute and one of the most important algorithms used for these structures is the ID3.

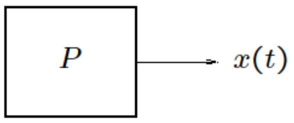
1.3 Time Series Forecasting

A common application of *machine learning* algorithms is *Time Series Forecasting.* We briefed above the powerful of *machine learning* in pattern recognition field and as we can simply figure many real applications have a trend of data which may be predictable.

For a better understanding of Time Series Forecasting we introduce what a Time Series is. It is a sequence of data points in a temporal space which have a natural temporal ordering [22].

$$\{x(t_0), x(t_1), \cdots x(t_{i-1}), x(t_i), x(t_{i+1}), \cdots\}$$

So the value $x(t)$ is the output of some process P we are interested in:



The type of process P could have one of the function below:

- **Predict** future values of $x[t]$
- **Classify** a series into classes
- **Describe** a series by using parameter values of some model
- **Transform** a series into another

The type of process P we interested in this work is the prediction.

Time series are the focus of different overlapping field as Information Theory, Dynamical Systems Theory and Digital Signal Processing and it can represent discrete or continuous phenomena.

Discrete Phenomena sometimes have data may have to be aggregated to get a meaningful values e.g. for a load of a server which receives requests from clients, it could have more sense to have number of request per time unit instead of the single requests [48].

Continuous phenomena are instead characterized by a continuous signal $x(t)$, where t is a real value and so, for having a series $\{x[t]\}$ we must sample the signal at discrete points with a sampling period Δt, so the series is:

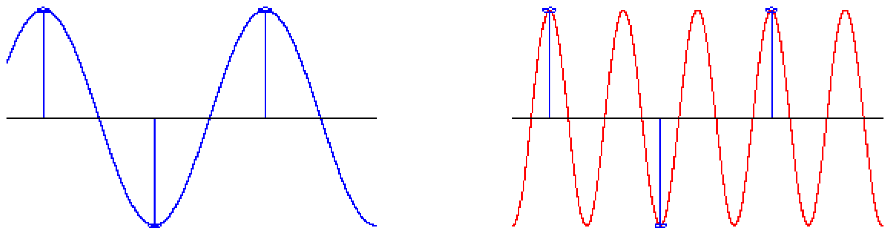
$$\{x[t]\} = \{x(0), x(\Delta t), x(2\Delta t), x(3\Delta t), \cdots\}$$

but the sampling period Δt has to be chosen according to the *Nyquist Sampling Theorem* in order to ensure that $x(t)$ can be recovered from $x[t]$.

Nyquist Sampling Theorem
Given a signal $x(t)$ where f_{max} is the highest frequency component, then the rate sample has to be at least twice as high:

$$\frac{1}{\Delta t} = f_{sampling} > 2f_{max}$$

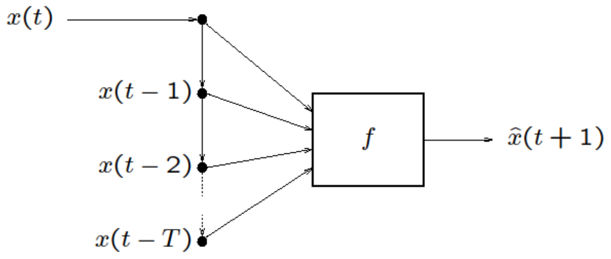
The reason is for avoiding aliasing of frequency in the range $[f_{sampling}/2, f_{max}]$.



Time series analysis analyzes time series data in order to extract characteristics of the data. As mentioned above, in this work we are interested to a Time Series Forecasting which is the use of a model to predict future values based on previously observed values by using of the natural one-way ordering of time so that values for a given period will be expressed as deriving in some way from past values, rather than from future values.

From an existing time series we have $\{x(t), x(t-1), \cdots\}$ where want to estimate some future values $\widehat{x}[t+s] = f(x[t], x[t-1], \cdots)$, where s is called *horizon of prediction*, and if $s = 1$ we mean just we want to find out the next value.

The problem we have is that the time is in an infinite-dimensional spatial vector while we have to work with a shift register with a finite dimension d. At each instant t we have to truncate the history to the previous d sample. That transformation is called *embedding* and the number of previous samples d is the *embedding dimension*.



Many authors researched a way to find out the best embedding dimension, but a general method does not exist. There exist however, some policies which can be used.

The process P which generate the values of a time series can be represented by a stochastic or deterministic model and linear on nonlinear model. The theory of Digital Signal Processing (DSP) give us the instrument to study the output of a linear time series, both deterministic and stochastic. These instrument are the *filters*, which by taking as input a sequence $u[t]$ are able to produce a sequence of output $x[t]$. There exist two main filter i.e. Finite Impulse Response Filters (FIR) and Infinite Impulse Response Filters (IIR) [14].

Finite Impulse Response Filters (FIR)

The name of these filters is because when they received as input $u[t]$ the impulse function, the output $x[t]$ is only as long as $q+1$, which are a finite number of coefficient that characterized the FIR. These filters implement the convolution of the input signal with a given coefficient vector β_i.

$$x[t] = \sum_{i=0}^{q} \beta_i u[t-i]$$

Infinite Impulse Response Filters (IIR)

These filters has a direct contribution of the input $u[t]$ to the output $x[t]$ a the the time t and depends of its own past samples. The name *infinite* derive from the response that only asymptotically decays to zero even if the impulse function and the vector $\{\alpha_i\}$ being finite in duration.

$$x[t] = \sum_{i=1}^{p} \alpha_i x[t-i] + u[t]$$

DSP Process Models

DSP theory offers three classes of linear process models for estimating the output of a time series:

- AutoRegressive models (AR[p]):
  This model uses a IIR filter with the relative parameter p for estimate the future values.
- Moving Average models (MA[q]):
  This model uses a FIR filter with the relative parameter q for estimate the future values.
- AutoRegressive Moving Average models (ARMA[p,q]):
  This model is a linear combination of FIR and IIR filters and the resulted is time series is
  $$x[t] = \sum_{i=1}^{p} \alpha_i x[t-i] + \sum_{i=1}^{q} \beta_i \epsilon[t-i] + \epsilon[t]$$
- AutoRegressive Integrated Moving Average models (ARMA[p,d,q]):
  This is a generalization of an ARMA model applied when the data show evidence of non-stationarity. In this case a initial differencing step (corresponding to "integrated" part of the model) can be applied for the removal of such non-stationarity. The parameter d is referred to integrated part of model, while p and q are referred to AR and MA models respectively.

In order to make forecasting, one of the most common applications of these models is the *Box-Jenkins* method [12] which applies ARMA or ARIMA to find out the best fit of a time series to past values of such time series.

Box-Jenkins method uses an iterative four phase approach:

1. **Preliminary Analysis**: checking out the stationarity of the series with the choice of the *d* parameter, identifying possible anomalous values and searching for the most appropriate changes to make the series stationary.

2. **Identification and Selection of Model**: identifying the parameter *p* and *q* through the analysis of *autocorrelation function* and *partial autocorrelation function* which are the cross-correlation (a function similar to the convolution to measure the similarity between two wave forms) of the signal with itself.

3. **Parameter Estimation**: using algorithms as *maximum likelihood estimation* or *non-linear least-squares estimation* to find out the coefficient ehich best fit the selected ARIMA model.

4. **Model Checking**: checking on the residuals of the estimated model by testing if such model comply with the specifications of a stationary univariate process. If not, we must return to step one and build a better model.

Switching to a nonlinear model to use with Artificial Neural Networks
The linear processes have advantages that they can be understood and analyzed in great detail, but they are totally inappropriate if the underlying mechanism is nonlinear and so the Box-Jenkins method could not be useful with respect to non-linear methodologies [63]. In real world application is very common to find nonlinear system. Nonlinear models are more powerful than the linear models, but need more training data and suffer to different problems as *overfitting* and *local minima* (we describe these problems with the relative solutions later).

So for switching from linear to nonlinear model, we replace the $\sum$ with a nonlinear activation function like $\tanh(\sum)$. We change so the use of ARMA[p,q] model with a nonlinear one, NARMA[p,q] or with a Jordan Net i.e. a NARMA variant memory-less which remembers only the output of the last step.

The figure 1.4 represents a weekly web server load of CAD in the department of Computer, Control and Management Engineering "Antonio Ruberti" from "Sapienza" University of Rome and is possible to notice a sort of periodicity which follows a pattern.

One of the most common applications in Machine Learning theory is the *Pattern Recognition*, which aims to provide a reasonable answer for all possible input configuration, taking

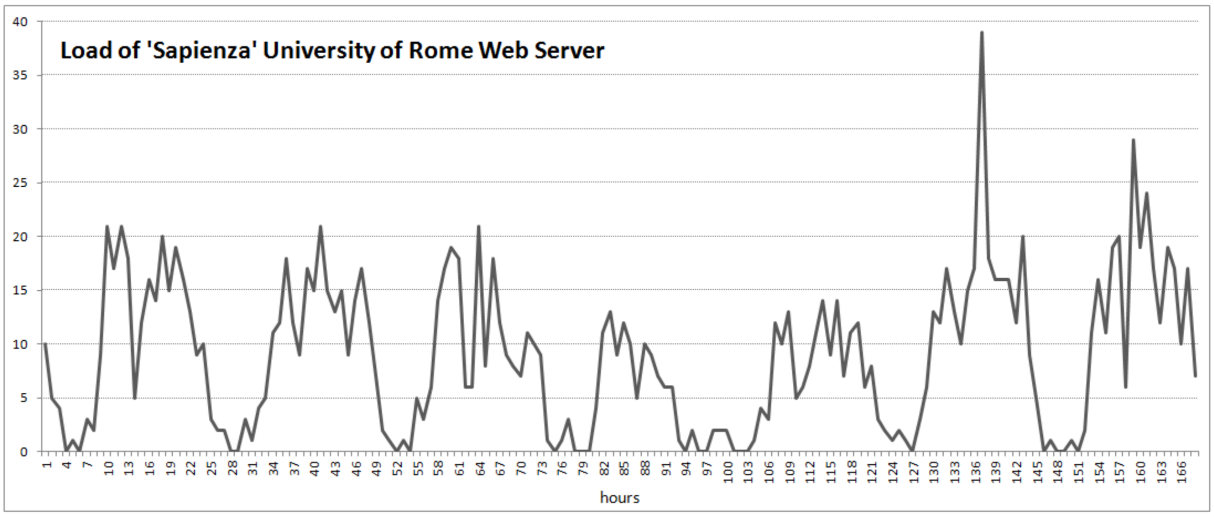


Figure 1.4: Examples of web server load

into account their statistical change, by performing *most likely* matching of the inputs. *Pattern Recognition* is pretty different to *Pattern Matching*, which looks for exact matches in the input with pre-existing patterns and it is generally not considered a type of *Machine Learning* algorithm.

Being our aim to predict a time series of load of a generic target service basing on a list of possible unseen inputs, we consider that the most useful learning type for our goal is the *Supervised Learning* and to create a Classifier with a supervised algorithms one of the best *nonparamteric* way is using the *Artificial Neural Networks* as many author suggest in related works [49, 17, 55]. The main application of the ANNs is the forecasting

Many works are made by using ANN for making predictions with ANNs ([41, 40, 50]) fully connected Feed-Forward [64], others with ANNs non-fully connected [23], others with Recurrent ANN instead of Feed-Forward [24, 76, 33]. An ANN has to be set correctly, but unfortunately there are not exist a general method, but several policies are available for number of input nodes, number of neurons [75]. In a time series approach it is pretty important to set the embedding dimension correctly [43]. Some research work demonstrate as an ANN trained just with a simple *backprop* performs well [39] and many different solutions are proposed to improve its performance and avoid problem arising like overfitting and local minima problem [51]. However many learning algorithms based on Genetics algorithms are available and seem give better performance in some cases [45].

By following these approach for a time series prediction we employed our forecaster.

Chapter 2

Make Your System Elastic!

This chapter describes *Make Your System Elastic* (MYSE), a Framework aimed to auto scaling a generic target replicated service. The first section explains the modeling of the target replicated service as a Queueing Model in order to calculate the expected response times and the second section describes the MYSE Framework Architecture with its modules and the description of each module.

2.1 Replicated Service Model

Given a generic target replicated service with s active servers with $s = 1, \cdots, N$, MYSE makes a parallel integrated work with that replicated service, aimed to provide real time the best configuration of system as possible. The goal is to find out the minimum number of active servers in the configuration which is able to meet a determined QoS while saving resources.

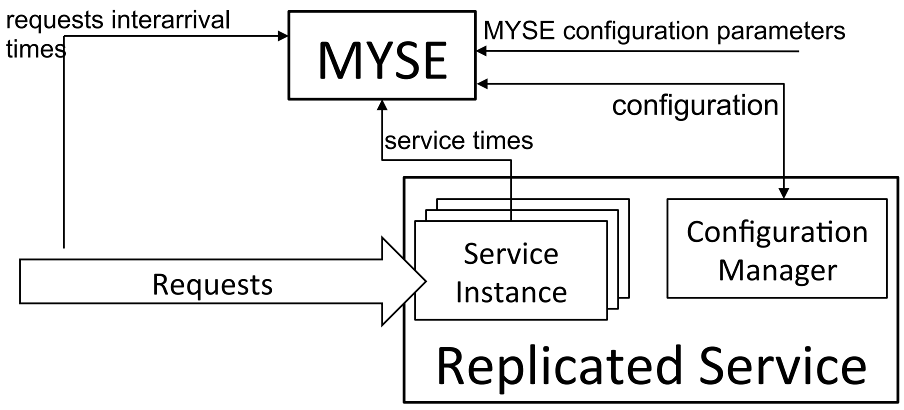


Figure 2.1: Integration of MYSE module with target replicated service.

From the Figure 2.1 it is possible to see the MYSE takes as input:

- communication parameters (Δ and QoS requirements)
- actual configuration of system
- service times
- request interarrival times

The basic idea is to consider the replicated service as a black box and monitor requests patterns over time to identify the relevant characteristics of input traffic so as to properly reconfigure the service through the Configuration Manager. To this aim, we assume that replicated service instances export, as performance metrics, their service times. This allows us to follow the black box approach as in [15, 71] by considering only observable parameters, like the requests patterns and the average request latency, without entering into the details of the specific service implementation and allows the MYSE Framework to work with several different types of applications. Monitoring requests arrival over time enables to predict their probability distribution, while monitoring both requests and replies allows to estimate currently delivered service times.

We modeled the replicated service as a queuing model with s servers and one common queue (Figure 2.2) [31, 61]. Without loss of generality, we considered that replicas are homogeneous (i.e. they have the same computational capabilities and can be used interchangeably) and a single type of service.

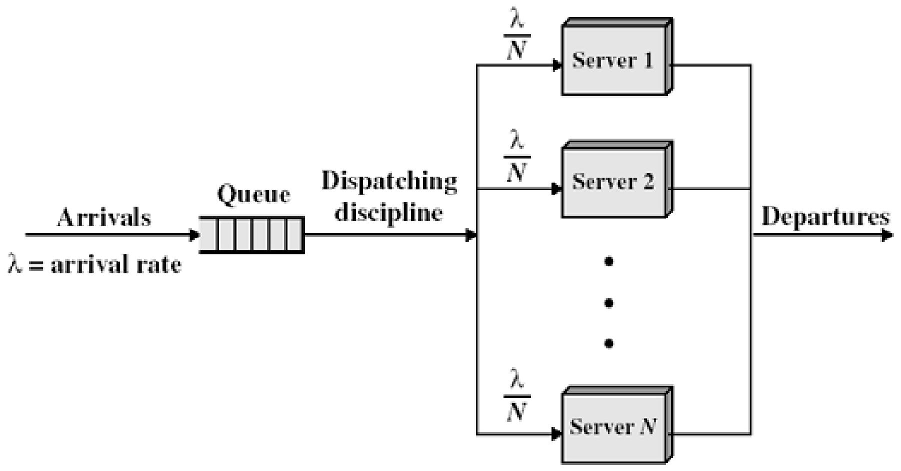


Figure 2.2: Queueing model architecture with single queue

Such modeling permits us to to manage the flow of requests as a queue belonging to Queueing Theory which gives us a mathematical study of waiting lines and we are able to estimate queue lengths and waiting times (Figure 2.3). .

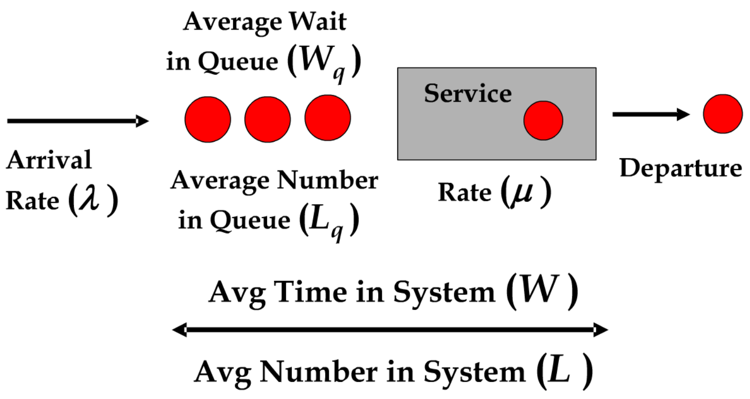

Average Wait in Queue (W_q)
Service
Arrival Rate (λ)
Average Number in Queue (L_q)
Rate (μ)
Departure
Avg Time in System (W)
Avg Number in System (L)

Figure 2.3: Queueing Model Architecture

Queueing Theory uses the Kendall's Notation A/B/s/c/p/Z where 'A' denotes the arrivals schema to the queue (i.e. the probability distribution of inter arrival times), 'B' the service schema (i.e. the probability distribution of service times), 's' the number of servers, 'c' the capacity of the queue, 'p' and 'Z' the population dimension and queueing discipline. We modeled the replicated service with the default FIFO scheduling policy 'Z', by assuming an unlimited buffer for requests and an unlimited population dimension, in order to use just the default value of 'c' and 'p' (i.e. 'infinite') and make the model simpler A/B/s.

Each request is identified with a $req(t) = \{req_{id}, client_j, intertime(t-1)\}$ i.e. a unique number to indicates the request, a number to indicates the user, and a number which is the inter arrival time from the previous request.
The scheduling policy considered is the default one, figuring out the user who need the services have the same priority, otherwise it makes sense to use a queue model based on priority matching at every users a priority level.

There exist 3 main kind of distributions for the arrivals and services: these are *M*, *D*, *G*. The most common and simplest queueing model are M/D/S, M/M/S, G/D/S, but many others there exist. 'M' stands for **M**arkov or **M**emoryless and means arrivals or services occur according to a Poisson process, 'D' stands for **D**eterministic and means jobs arriving at the queue require a fixed amount of service, 'G' stands for **G**eneral and means arrivals occur according to a general process.

STANDARD DEFINITIONS AND NOTATIONS
We introduce the most important notations. Their definitions belonging to the Queueing Theory applied to MYSE Framework:

- **Average Frequency of the Arrival Requests**: the average frequency of the arrival requests in the system is defined as the average arrival number of client requests in time unit. This average frequency is indicated as λ;

- **Average Service Time or Service Speed**: the average service time of client requests is defined as the average number of requests for which the service is provided in time unit. The service speed is indicated as μ;

- **Utilization Factor**: the utilization factor represents the percentage of time the servers are busy, and it's defined as:
$$\rho = \frac{\lambda}{s\mu}$$

CORRECTNESS ASSUMPTION

These models are considered stable if $\rho < 1$ where $\rho = \lambda/\mu n$; this is the existence condition of a steady-state and under this assumption we built our model. Ignoring transitional time the Queueing Theory gives us a way to estimate times and number of requests both in queue and in the global system.

PERFORMANCE MEASURES

The most important performance measures are:

1. N : The average number of requests in the system
2. N^q : The average number of requests in the queue
3. T : The average waiting time of requests in the system
4. T^q : The average waiting time of requests in the queue

Having the expected waiting times and a QoS requirement threshold given as input, we are able to figure out whether the compliance is actually achieved and if the replicated service has yet the optimal configuration. Otherwise the service needs a reconfiguration hence has to switch from s active servers to a new configuration with $s = s \pm x$.

We describes in next section how we pick the correct queueing model.

2.2 MYSE Architecture

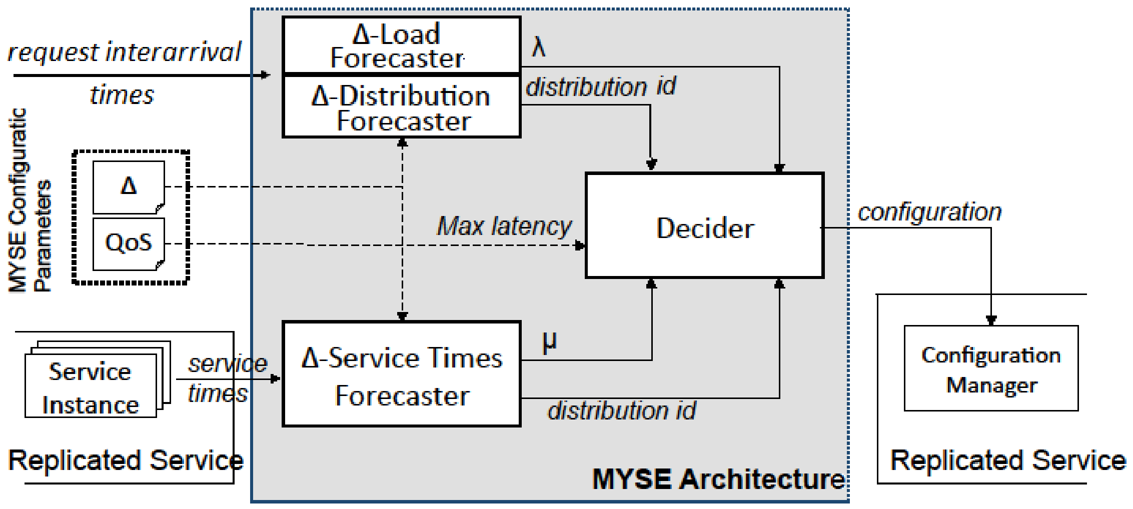


Figure 2.4: MYSE Internal structure

Figure 2.4 details the internal architecture of MYSE and it is possible to see the 4 main modules. The Δ-*Load Forecaster* and Δ-*Distribution Forecaster* work parallel in a single module which takes as input the request interarrival time of each request and the Δ parameter and are in charge of learning and forecasting request rate and request distribution in order to give to the Decider module the load λ (number or requests per time unit) and the distribution probability of such requests. The Δ-*Service Times Forecaster* looks at service time patterns from the replicated service instances to extract the distribution of service times and its mean μ.

The *Decider* determines the suitable configuration to meet QoS on the basis of the inputs supplied by the other submodules.

The forecasters are employed with parallel Artificial Neural Networks (ANNs), so as to conveniently update the configuration early enough to avoid temporary performance worsening or resource under-utilization. The timeline of these predictions is provided externally by the Δ parameter.

Single submodules are described below.

2.2.1 Δ-Load Forecaster Module

This module has the job of making predictions about the load in order to give to the Decider the λ value.

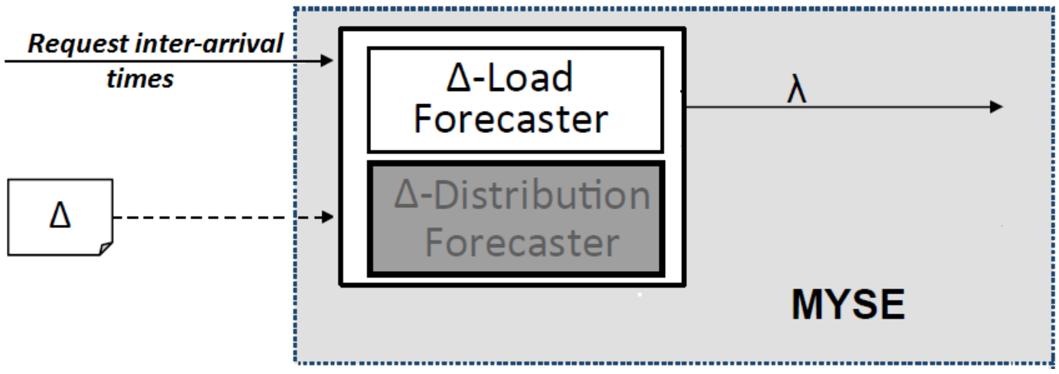


Figure 2.5: MYSE Δ-Load and Δ-Distribution Forecaster

It analyzes request rate over time and employs an ANN to provide predictions on expected request rate λ within Δ time units. It takes as input the Δ parameter and the request time interarrival , but it doesn't use these value properly, it accumulates the only the number of requests in order to have the load per each time unit λ.

An accurate description of the implemented ANN, with their parameters and the tests made to arrive to the best configuration is available in chapter 3 and chapter 4.

2.2.2 Δ-Distribution Forecaster Module

it analyzes request inter-arrival times to produce predictions on request distribution Δ time units ahead.

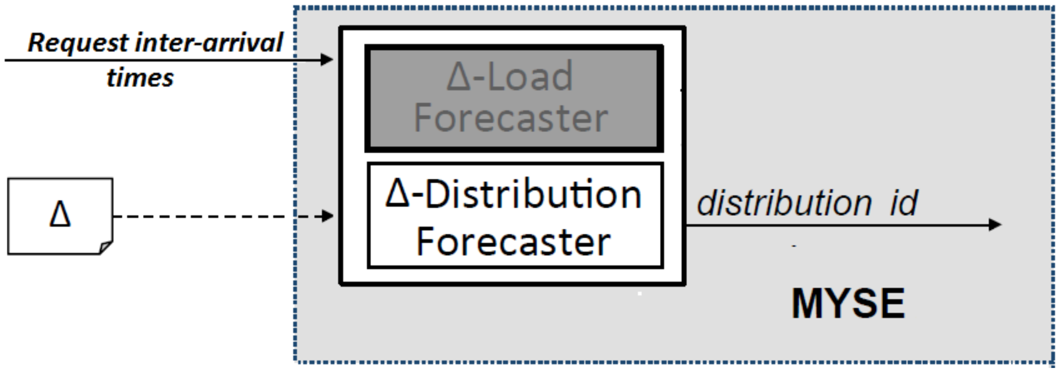


Figure 2.6: MYSE Δ-Load and Δ-Distribution Forecaster

It is composed by two parts, the *Distribution Recognizer* and the *Distribution Predictor*;

The **Distribution Recognizer** ([74]) estimates the "best-fitting" continuous or discrete distribution by analyzing a set of samples given in input, which represent the request inter-arrival times with a sliding fixed-length window (experimental tests made said the length has to be greater than 80 samples to reduce the estimation error below 5% and the estimation latency below 3 ms).

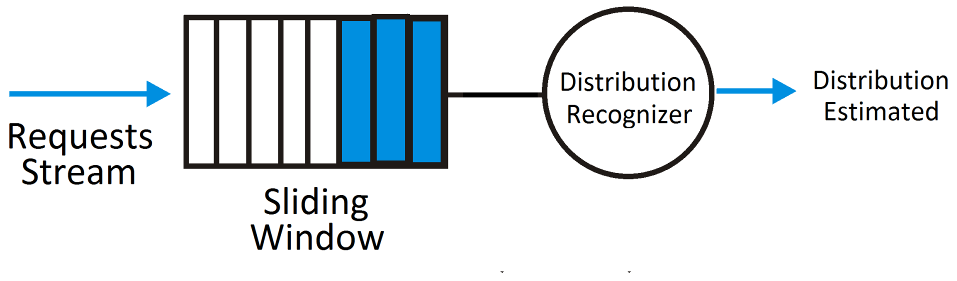


Figure 2.7: Distribution Recognizer

The estimation of distribution parameters (i.e., expected value, standard deviation) is made by using the "maximum-likelihood estimation method" [47]. These parameters are then used to perform "goodness-of-fit" tests, such as the "chi-squared test" for discrete distributions and "kolmogorov-smirnov test" for continuous distributions, for discriminating among distinct distributions [19].

The **Distribution Predictor** is able to predict the future distribution by using a time-series ANN taking as input the output produced by the Distribution Recognizer.

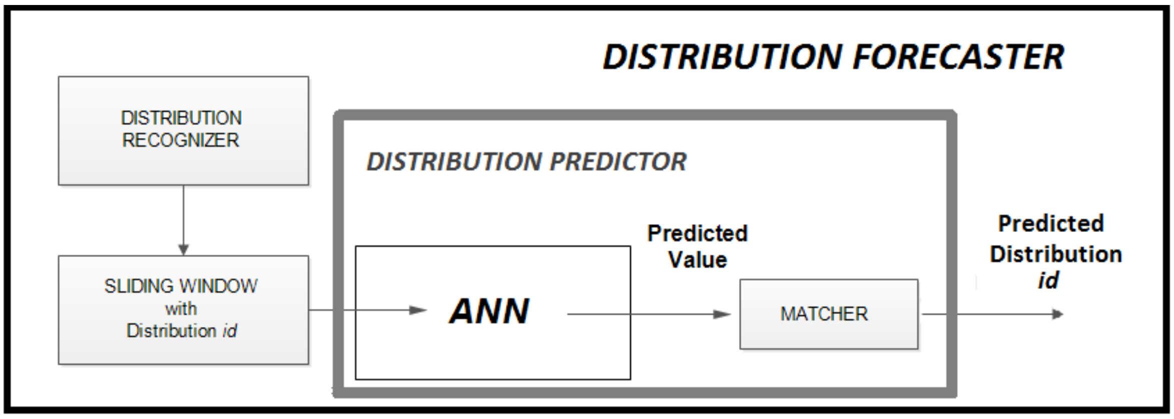


Figure 2.8: Architecture of Distribution Forecaster Module

The Distribution Predictor contains after the ANN a **Matcher** because the output of the ANN is a code whose scope is to represent a distribution. The Matcher give the distribution *id* to the Decider. Its working is explain in chapter 3.

2.2.3 Δ-Service Times Forecaster Module

It takes as input the service time patterns and produces as output the estimation of their distribution and the mean μ of service times. The same techniques employed for the Δ-Load and Distribution Forecasters are used here to recognize the distribution of time series and to predict both their distribution and mean Δ time units ahead. Tracking only input/output traffic to derive the service times allows to be non intrusive with respect to target service. According to the black box approach and in order to be non intrusive, the service requests and responses can be traced by sniffing packets traveling in and out of the system. In our simulation, we considered service time distribution as known, even if the module can be implemented as we did for the Δ-Load and Distribution Forecasters

2.2.4 Decider Module

It computes the minimum configuration for guaranteeing QoS compliance in service provisioning (i.e. the latency in the specific case). It takes as input the latency threshold specified in the QoS, the predictions on request distribution (distribution *id* and λ) and the distribution of service times (distribution *id* and μ).

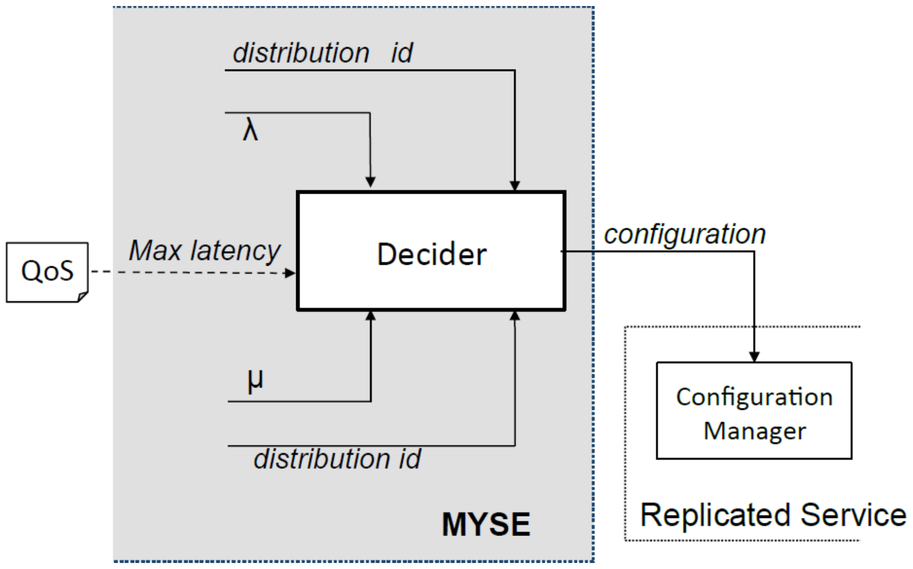


Figure 2.9: Decider Architecture

It first computes the latency T expected for the current configuration with s replicas by applying well-known queuing model technique. Then, it applies a simple decision algorithm to scale out, in case QoS is not violated, or scaled in, in case a configuration with less replicas can still guarantee QoS compliance.

A Simple Decision Heuristic

The reference heuristic used for our tests is the simplest as possible and it is based only on finding the minimum number of servers to ensure the QoS wished. For doing that the Decider module employed several queueing models and chooses the one which is the best fit with the distribution given as input. The expected latencies are so calculated with formulas given by the model and hence the number of active servers in the new configuration are chosen. This heuristic does not consider previously QoS violation or flipping phenomena (consecutive switch-on and switch-off of a server), but permits us to evaluate the performance of MYSE with respect to the optimum number.

A Graph-Based Heuristic

An improved graph-based heuristic was employed to improve the performance with respect to the flipping phenomena which could bring more costs [74]. If QoS compliance can be achieved with current configuration, then the algorithm evaluates if switching off a replica still makes possible to satisfy QoS requirements and in case, it deallocates one server moving from a configuration with s servers to a configuration working with $s-1$. On the contrary, if the expected latency is higher than QoS threshold, then the algorithm decides to activate new replicas moving from a configuration with s servers to a new configuration with $s+1$ servers.

In order to manage isolated request bursts and/or errors introduced in the estimation process that may cause a flipping phenomena among servers configurations (i.e. rapidly alternated switch on and switch off decisions), we introduced a *cost function* that prevents the algorithm to move back in a certain configuration s if such configuration was too recent.

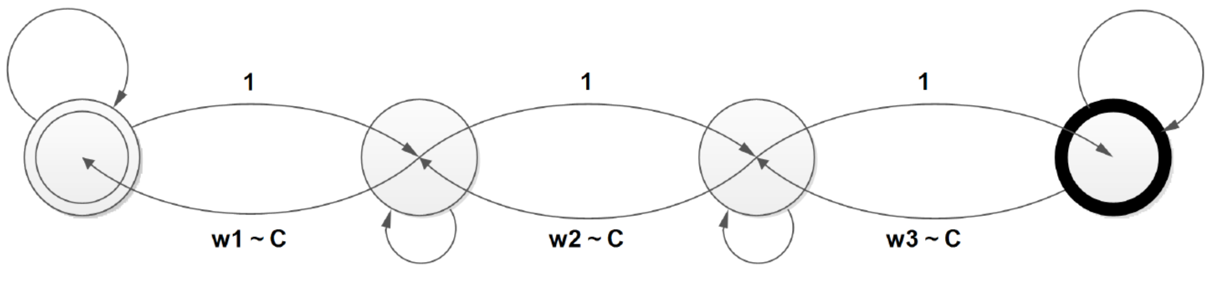


Figure 2.10: Decision Graph with $s = 4$

To this aim, we defined an edge-weighted directed graph $G = (V, E, w(e,t))$ where (i) the set of vertex V represents all the possible configurations (i.e. number of active replicas) i.e. $V = \{1, 2, 3, \ldots s_{max}\}$, (ii) there exists an edge between any pair of adjacent configurations (i.e. there exists an edge $e_{s,s'}$ from any configuration s to the configuration $S' = s + 1$ and $s' = s - 1$ with $s, s' > 0$) and (iii) for any edge $e_{s,s'} \in E$ the edge weight $w(e_{s,s'}, t)$ represents the cost of moving from the configuration s to the configuration s' at the current time t. In the current implementation, we selected a cost function $w(e_{s,s'}, t)$ that is proportional to a certain constant parameter C when the algorithm moves to the configuration s' and then decreases linearly in time until it comes back to 0.

Other heuristics are employed [74], but in this work has been used only the heuristic simply based on optimum number of replicas given by the queueing model.

Chapter 3

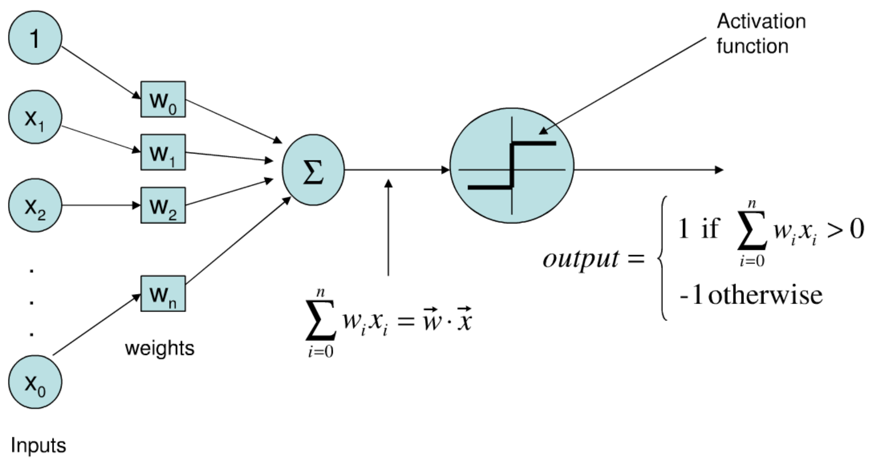
Design and Evaluation of MYSE Forecasters

This chapter explain how we designed the ANNs, following the time series idea and combining together with a date-based approach. We describe the most relevant ANNs made for Load Forecaster and after an explanation of the coding of Distribution Forecaster, the ANN used for the predictions of that code with the functionality of the Matcher. We describe then the dataset used and the working of our Sliding Window with the approach used to set a dimension. It is also available an explanation of the Backpropagation Algorithms employed with a comparison between the main different versions of such algorithm and the solution used for its problem, like local minima and overfitting. Finally we see the evaluation of each single module.

3.1 Artificial Neural Networks

As mentioned above, in this work we chose to use *Artificial Neural Networks* (ANNs) as computational model for machine learning and pattern recognition. ANN is a model inspired by the brain with a network of interconnected *Artificial Neurons* able to compute the output values by feeding information through the network.

3.1.1 Artificial Neuron



Activation function
1
x_1
x_2
x_0
Inputs
w_0
w_1
w_2
w_n
weights
Σ
$\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure 3.1: Neuron architecture

An *Artificial Neuron* is a mathematical function introduced by McCulloch & Pitts in 1943 [36] which has $m + 1$ inputs with signals x_0 trough x_m and weight w_0 trough w_m. Usually the first input x_0 is assigned to value $+1$ which makes it a *bias* input with $w_{k0} = b_k$, so only m actual inputs to the neuron.
The output of a generic neuron k is:

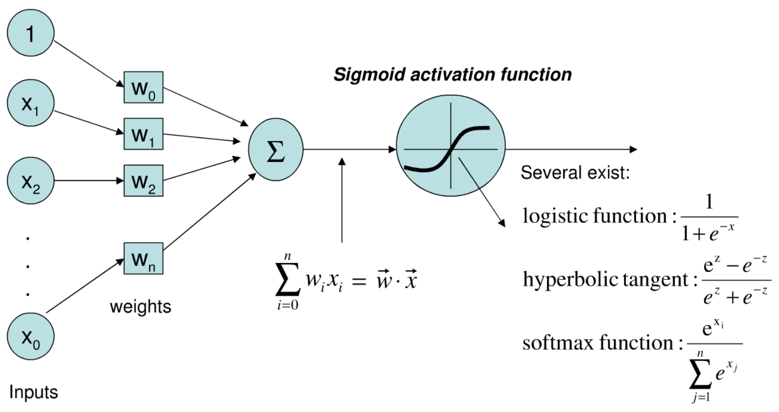
$$y_k = \phi(\sum_{j=0}^{m} w_{kj} x_j)$$

where ϕ is a transfer function which has a task of activation function *"ON-OFF"*. A typical transfer function is the *step function* where a threshold establish when a sum of signals is enough strong to go ahead $(+1)$ otherwise it has to stop (-1):

$$f(x) = \begin{cases} +1 & \text{if } w \cdot x + b > 0 \\ -1 & \text{otherwise} \end{cases}$$

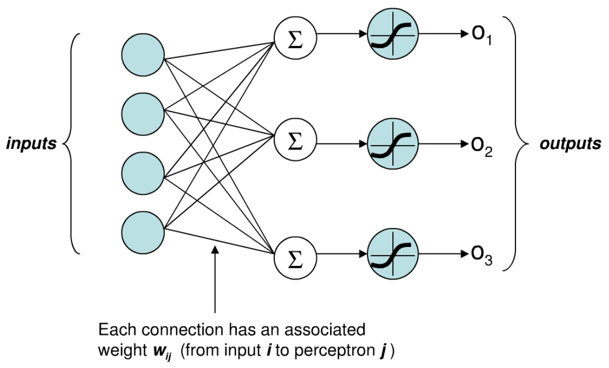
Actually, that function is too much strongly selective, so there exist many different transfer functions belonging to sigmoid functions family, which can be smoother than the step function and to give as output a real number in a range.

3.1.2 The Perceptron



1
x_1
x_2
x_0
Inputs
w_0
w_1
w_2
w_n
weights
Σ
$\sum_{i=0}^{n} w_i x_i = \vec{w} \cdot \vec{x}$
Sigmoid activation function
Several exist:
logistic function : $\frac{1}{1+e^{-x}}$
hyperbolic tangent : $\frac{e^z - e^{-z}}{e^z + e^{-z}}$
softmax function : $\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

Figure 3.2: Perceptron architecture



inputs
Σ
o_1
o_2
o_3
outputs
Each connection has an associated weight w_{ij} (from input i to perceptron j)

Figure 3.3: Single-Layer Perceptron architecture

In Artificial Neuron Networking the *Perceptron*, introduced by Rosenblatt in 1962 [57], is the easiest data structure that improve the *Artificial Neuron* by using a smoother activation function. The *Perceptron* can be trained with a simple supervised algorithm (Delta Rule [11]) that iteratively present from a *training set* the inputs to *Perceptron* which calculate the output and having the actual output value is able to compute the error and update the weights on every *Dendrites* which are the synapses i.e. the final part of links coming from previous layer neurons.

The *Single-Layer Perceptron* (SLP) are a kind of ANN which has an input layer with the input node (that are not neurons), for these not have a computational function, and only one output layer with one or more output neurons.
SLP are able to learn some input and output patterns, but Minsky and Papert demonstrated those have to be *linearly separable* [44].

Two point sets are called *linearly separable* in n-dimensional space if they are able to separate those by a hyperplane. In mathematical terms: Let X_0 and X_1 be two sets of points in an n-dimensional space. Then X_0 and X_1 are linearly separable if there exists $n+1$ real numbers $w_1, w_2, .., w_n, k$, such that every point $x \in X_0$ satisfies $\sum_{i=1}^{n} w_i x_i \geq k$ and every point $x \in X_1$ satisfies $\sum_{i=1}^{n} w_i x_i < k$, where x_i is the $i-th$ component of x.

3.1.3 Multi-Layer Perceptron

Having only one layer, there not exist a way for SLP to learn patterns which is not *linearly separable*. A multi-layered network overcomes this limitation as it can create internal representations and learn different features in each layer.

The most common used ANN are the *Feed-Forward* whose interconnections among *Perceptrons* follow only a direction towards the next layers fully connected, but there exist other connection types as *Recurrent*. A *Multi-Layer Perceptron* besides having an *input layer* and an *output layer* there are one or more *hidden layers* made by an arbitrary number of neurons (or perceptrons).

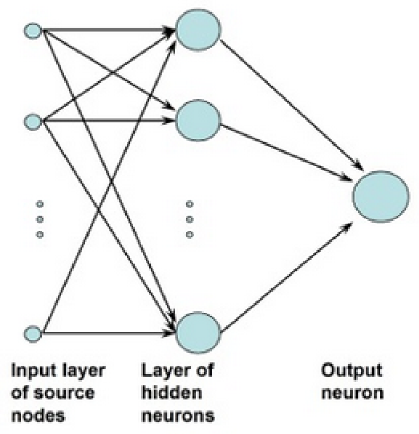


Figure 3.4: Multi-Layer Perceptron architecture with 1 hidden layer

3.1.4 Learning from Data

A *Multi-Layer Perceptron* (MLP) has to be trained to give it the generalization power. Two main different approaches there exist:

1. *BackPropagation Algorithms* (*backprop*): are *supervised learning* algorithms based on calculate and backpropagation of output error and gradient descent.

2. *Genetics Algorithms* (GAs): are algorithms aimed to find out exact or approximate solutions to optimization and search problems.

The choice of the kind of algorithm must be done on the base of the problem we have. Both approach is useful to find out the best network configuration by acting in different way. Many study are made to compare these approach and GAs result more accurate than *backprop* even if GA approach is more sensible in term of speed to network dimension, i.e. increasing network dimension (number of neurons and dendrites give more weights to be calculate) GAs, being an optimization algorithm, has a big number of variables and can be more and more slow. On the other hand, *backprop* is less accurate, but the execution time of error backpropagation is linear with the number of weights, so such approach can be faster than GA.

On our work we employed the learning with the *backprop* for it gives us empirically a good tradeoff between speed and accuracy.

Backpropagation Algorithm
This algorithm was introduced by Rumelhart, Hinton & Williams in 1985 [58] and it is the generalization to a MLP of *Delta Rule* which is a rule for updating the network weights on a SLP based on *gradient descent*. *Gradient Descent* (also called *steepest descent*) is a first-order optimization algorithm useful to find a local minimum of a function by using gradient descent by taking steps proportional to the negative of the function gradient at the current point. Many real applications are made with this approach, like *Least Mean Squares Algorithm* [70] create for reducing the noisy echo generate by the interaction of a mic and a speaker.

The aim of *backprop* is to works by decreasing iteratively the error produced of ANN on the output nodes than the expected result and by backpropagating such error trough the network to update the weights of the *dendrites* of every *perceptrons*.

We represent the error in output node j in the n_{th} data point by $e_j(n) = d_j(n) - y_j(n)$, where d is the target value and y is the value produced by the *perceptron*. We then make corrections to the weights of the nodes based on those corrections which minimize the error in the entire output, given by

$$\mathcal{E}(n) = \frac{1}{n}\sum_j e_j^2(n)$$

Using gradient descent, we find our change in each weight to be

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$$

where y_i is the output of the previous neuron and η is the *learning rate*, which is carefully selected to ensure that the weights converge to a response quickly, without producing oscillations. In programming applications, this parameter typically ranges from 0.2 to 0.8.
The derivative is depending on the induced local field v_j, which itself varies. It pretty easy to prove that for an output node this derivative could be simplified to

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n))$$

where ϕ' is the derivative of the activation function described earlier, which itself does not vary. The same analysis is harder to do for the change in weights to hidden nodes, but it can be shown that the relevant derivative is

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \mathcal{E}(n)}{\partial v_k(n)} w_{kj}(n)$$

This depends on the change in weights of the k_{th} nodes, which represent the output layer. So to change the hidden layer weights, we must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a *backprop* of the activation function.

After calculating the change of weights, the algorithm calculates the update of every *Dendrite* weights:

$$\Delta w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n) + \underbrace{\beta * \Delta w_{ji}(n-1)}_{for\ improved\ \beta\ version}$$

where β is a *momentum* term and its useful is for helping the learning process. We focus this aspect below.

There exist 2 main version of *backprop*:

1. ***Online***: means a weight update after each pattern.
    - order in which learning samples are presented plays a role
    - usually more accurate than *offline*
2. ***Offline (batch learning)***: is a variant in which the weights variations are accumulated during a training epoch, and the network weights are actually updated at the end of a training epoch.
    - order in which learning samples are presented does not play a role
    - more memory consumption needed
    - faster than *online*

Pseudo-Code of Backprop Algorithm

```
Data: Untrained ANN, Training Set, Validation Set
Result: Trained ANN
initialize network weights (often small random values);
while training example exist do
    prediction = neural-net-output(network, ex) // forward pass
    actual = teacher-output(ex)
    compute error (prediction - actual) at the output units
    compute $\Delta w_h$ for all weights from hidden layer to output layer // backward pass
    compute $\Delta w_i$ for all weights from input layer to hidden layer // backward pass continued
    update network weights;
    if all examples classified correctly or stopping criterion satisfied then
        return the network
    else
        go on the loop;
    end
end
```

Algorithm 1: Backprop Algorithm

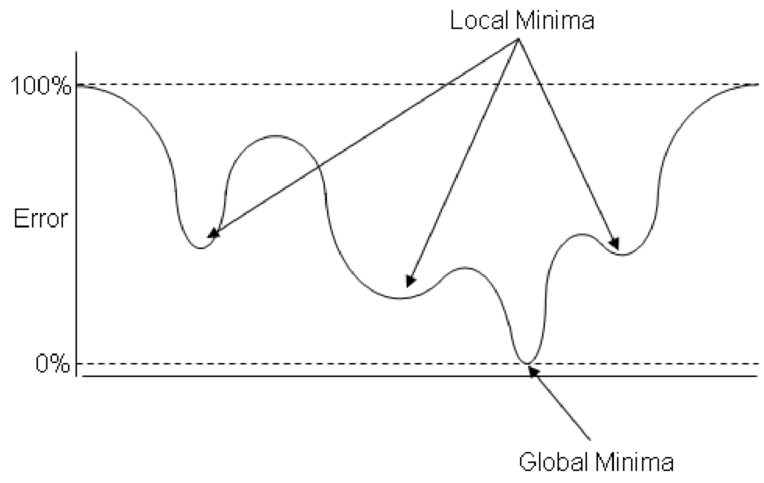


Figure 3.5: Local Minima problem

Limitations of Backprop Algorithm
The worst limitations of *backprop* are:

- Local Minima suffering: i.e. going on with iterations the error rate should decrease but we may have fluctuations (Figure 3.5). The *gradient descent* algorithms suffer from problem 1 of *local minima*, anyway there exist different ways to reduce this problem:
    1. using *Genetic Algorithms*
    2. choosing the *online* version of *backprop* algorithm: due to the noise introduced in the error surface by online learning, it can by itself be sufficient to avoid local minima, but because of the random oscillation introduced, online learning is usually slower that offline learning.
    3. using the *momentum* term which help the algorithm to converge
- Slow convergence to optimum
- Convergence to optimum not guaranteed

To improve the performance of the ANN and to reduce the convergence problems *backprop* can use, even is not required, a normalization of input vectors [20].

More, there exist different improved versions of *backprop* which are aimed to reduce such limitations [67, 66, 34, 68] and one of more used version is the *Resilient Backprop*.

Resilient Backpropagation Algorithm

This algorithm has been introduced by Riedmiller & Braun in 1992 [52] and it is a batch update version of backpropagation. It is considered the best version of *backprop* right now as some authors demonstrated [35, 38].

The algorithm uses the same methods of traditional Backprop for the calculation of errors and for the calculation of partial derivatives of each neurons, but the purpose of this algorithm, also called RPROP, is to eliminate the harmful effects of the magnitudes of the partial derivatives by using only the sign of the derivative to determine the direction of the weight update [53, 54].

The algorithm uses an update value Δ_{ij} for each synaptic weights and the reason of using the derivative sign is due to the fact that when the derivative changes its sign compared to the previous step, means the algorithm has just jumped over a minimum and the update value was to high. In such case RPROP decreases the update value, in order to have at the next step a jump less than the previous and the descent go on towards the minimum. If instead the derivative maintain the same sign the update value is incremented in order to speed up the convergence.

$$\Delta_{ij}(t) = \begin{cases} \eta^{+} * \Delta_{ij}(t-1) & \text{if } \dfrac{\delta E(t-1)}{\delta w_{ij}} * \dfrac{\delta E(t-1)}{\delta w_{ij}} > 0 \\ \eta^{-} * \Delta_{ij}(t-1) & \text{if } \frac{\delta E(t-1)}{\delta w_{ij}} * \frac{\delta E(t-1)}{\delta w_{ij}} < 0 \\ \Delta_{ij}(t-1) & \text{otherwise} \end{cases}$$

where η^{+} and η^{-} are respectively the positive and negative step value typically set up to 1.2 and 0.5. After the setting of the update value Δ_{ij} the synaptic weights update follows the rule below:

$$\Delta W_{ij}(t) = \begin{cases} \text{-}\Delta_{ij}(t) & \text{if } \frac{\delta E(t)}{\delta w_{ij}} > 0 \\ +\Delta_{ij}(t) & \text{if } \frac{\delta E(t)}{\delta w_{ij}} < 0 \\ 0 & \text{otherwise} \end{cases}$$

Pseudo-Code of RPROP Algorithm

```
Data: Untrained ANN, Training Set, Validation Set
Result: Trained ANN
∀ i,j: $\Delta_{ij}(t) = \Delta_0$
while (not convergence) do
    compute $\frac{\partial E(t)}{\partial w}$
    for (all weight and biases) do
        compute $c = \frac{\partial E(t-1)}{\partial w_{ij}} * \frac{\partial E(t-1)}{\partial w_{ij}}$
        if (c > 0) then
            $\Delta_{ij}(t) = min(\Delta_{ij}(t-1) * \eta^{+}, \Delta_{max})$
            $\Delta W_{ij}(t) = -sign(\frac{\partial E(t)}{\partial w_{ij}}) * \Delta_{ij}(t)$
            $\Delta W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij}(t)$
        end
        if (c < 0) then
            $\Delta_{ij}(t) = max(\Delta_{ij}(t-1) * \eta^{-}, \Delta_{min})$
            $\Delta W_{ij}(t+1) = W_{ij}(t) - \Delta W_{ij}(t-1)$
            $\frac{\partial E(t)}{\partial w_{ij}} = 0$
        else
            $\Delta W_{ij}(t) = -sign(\frac{\partial E(t)}{\partial w_{ij}}) * \Delta_{ij}(t)$
            $\Delta W_{ij}(t+1) = W_{ij}(t) + \Delta W_{ij}(t)$
        end
    end
end
```

Algorithm 2: RPROP Algorithm

where Δ_0, Δ_{min} and Δ_{max} are parameter we can set arbitrary, but they are typically set up to $\Delta_0 = 0.1$, $\Delta_{min} = 1/e^6$ and $\Delta_{max} = 50$, while it doesn't need a learning rate and momentum term which may be difficult to find out. A comparison between the original Backprop and RPROP is explained in the next chapter.

Different variants of RPROP there exist, the version we have just explained is the original Reidmiller implementation. Into the library Encog 3.0 [6], implemented for Java and C, are available four versions of RPROP:

3.2 Neural Network Settings

We made our ANNs able to perform predictions to a timeline which is provided externally by the Δ parameter (*horizon of prediction*). Considering $\Delta = 1$ means forecasting the next temporal value. We employed the Δ-Forecasters in Java and structured the ANN with the follow classes:

- **Neuron**: it has one or more input (Dendrites) coming from other Neurons and it has only one output towards the Neurons of next layer. More it has a Bias Weight which is useful to set the point of Neuron work. This synapse is linked to a fake unit which gives a signal equals to $+1$. Input and output signals both can assume a real value between $(0, +1)$. Output signal is called Activity and it is calculated by applying a sigmoid transfer function.
- **Dendrite**: is the input of the Neuron and it can be interpreted as the synapse between real neurons. The Dendrites is characterized by the Neuron pointed and by the weight which change during the learning phase.
- **Layer**: it represents the set of Neurons in the Input, Hidden, and Output Layers.
- **Learning**: it represents the algorithm used to train the network. We used the Backprop algorithm, starting from a random weights. The learning rate represents the speed of error variation of Neuron at the changing of internal Activity. (RIVEDI)

We used ANNs for the Δ-Load Forecaster, Δ-Distribution Forecaster and Δ-Service Time Forecaster, but with several different configurations. We describe in the next sections the implementation of these ANNs.

To obtain the best network configuration we had to choose several parameters.

3.2.1 Design of ANNs

Network Dimension

A general method to set the number of hidden layers and nodes (network dimension) for obtaining good generalization and low overfitting doesn't exist, so we empirically chose the the number of neuron for each level by following several guidelines which are available [75]. One hidden layer is sufficient to approximate any complex nonlinear functions to any desired accuracy, but may be necessary many hidden neurons to have good performances and it could be difficult to find the right dimension, while is known ([75]) that with two hidden layers we are able to implement with a few number of hidden nodes a network with good performances. We implemented several ANNs and run many tests to find out the best approach. We see these tests in the next section.

The input level is made by a set of node which represent the time series of data (in our case the client requests). We empirically discovered that the date gave us a great knowledge of future traffic rate, so we tried to use such approach together with a time series approach (with the last d value as input)([75]). In the next section we see our implementation of similar Jordan Net model and we see how just simply considering only the last traffic value works yet well.

In the same empirical way, we chose the number of hidden nodes structured in one or two layers, while the output nodes simply represent the predicted traffic values [75] and the number of output nodes represents our horizon of prediction. So, having e.g. 5 output nodes means that we are able to predict the next 5 temporal units.

We see in the next chapter also the different configurations between the ANN of the Distribution Predictor and the ANN of the Load Forecaster. The main difference is the number of input and hidden nodes, which are more in the ANN of Distribution Predictor because the use of a coding has make the network more complex having big and unnatural jumps.

There exist in the state of art many algorithm purposed for finding the network dimension: These algorithms belonging mainly of two kind of algorithm:

- Constructive Algorithms: take a minimal network and build up a new layer node connections during the training.
- Destructive Algorithms: take a maximal network and prunes unnecessary layer nodes and connections during the training.

Inter-Neuron Connections and Activation Function

Another two main aspects to consider during the network design are the connections between neurons and the nonlinear activation function. In the state of art for work with similar goals [64, 29] are commonly used the Feed-Forward network, i.e. a network implemented as a directed acyclic graph in which each neurons in a level i is connected to every neurons in level $i + 1$, even if some time-series based network used a Recurrent interconnection [24]. The sum of all weight is processed by each neurons and the resulted value is obtain by the activation function.

The main activation function used are the tanh and the sigmoid which are respectively:

$$\tanh(t) = \frac{1 - e^{-2t}}{1 + e^{-2t}} \quad \text{and} \quad s(t) = \frac{1}{1 + e^{-t}}$$

Some studies demonstrated better performance in term of prediction accuracy and speed of convergence (iterations needed to reach a low error rate) of *tanh* [37], but in our tests we used both of these function and we did not see considerable difference in the output predicted, so we fixed the sigmoid for the following tests.

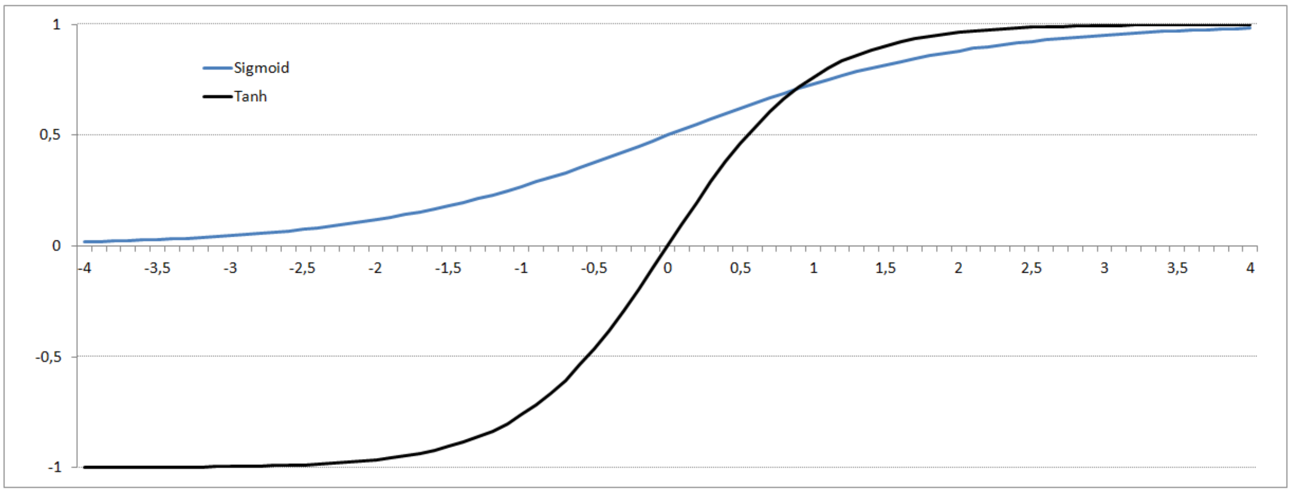


Figure 3.6: Sigmoid and Tanh function

3.2.2 Backprop vs RPROP

The learning is the main phase of ANNs. As mentioned before we used the Backpropagation algorithm with and without its β-variant and the Resilient Backpropagation (RPROP). The main difference in setting is that while the RPROP is parameter free, in the classic Backprop we need to set up the learning rate η and in case also the momentum term β. These parameters are be set up in a range (0.0, 1.0), but unfortunately a general method to choose them does not exist. The η is very important for the network predicted value change capabilities. A low value of η makes the ANN more accurate in predictions, but suffer to a very slow adaptation to sudden changes of value, while an high value of η give to the ANN high capabilities of predict sudden changes, but it may be less accurate. The β instead is very useful to speed up the convergence of the network during the learning process when the weight are moving in a single direction.

So we empirically fixed the *learning rate* to 0.3 in the ANN of Load Forecaster, while we used an higher value in the ANN of Distribution Predictor, in particular a *learning rate* fixed to 0.99, for the reason mentioned above of our using of code which has sudden changes. The *momentum* term is instead fixed to 0.5.

The main difference between Backprop and RPROP is the speed of convergence to find the minimum value of error. Both the algorithms work in iterations and there exist two main policies to stop the iterations. Backprop mainly stop to iterate after a fixed number of iterations, alternatively when reach an error under a threshold. If we set a too low threshold we could fall in a unlimited loop, so we need to setting up a maximum number of iterations to exit from such situations. RPROP used the same exit way, and we see in the next chapter how the RPROP has a few number of iteration than Backprop has.

An important aspect to consider for choosing the number of iteration (if we want to fixed them) is the overfitting problem. It is not true that more iteration we have, more accurate will be the output, because there exist this problem which is an excessive training of the network that causes a great capabilities to predict the training set, but an undue capabilities of generalization. We describe this aspect and the relative tests made in the next chapter, but here we want to say that there exist some technique to overcome such problem.

3.2.3 Data, Dataset and Error Measure

Our dataset is provided from Google Analytics of CAD in the department of Computer, Control and Management Engineering ”Antonio Ruberti” from ”Sapienza” University of RomeSapienza web server and it is structured in hours, so at every line we have a date made of 4 parameters (day, day of the week, month, hour) and the corresponding traffic value. To check the adaptation powerful to different patterns, we improved also a synthetic dataset composed by a normal dataset concatenated with a *sin* function and the real dataset scaled.

The dataset value come into a fixed length sliding window, with the last d value of traffic load. In the next chapter we see how we choose d to obtain the best tradeoff between speed and accuracy.

Input Normalization
The data before to be used with the network could be processed. We chose to normalized the data before entering into the sliding window in a range $[0, +1]$ or $[-1, +1]$, so we tried two normalization technique:

1. **Max-Min Normalization**:

$$v' = \frac{v - \alpha_{min}}{\alpha_{max} - \alpha_{min}} * (\beta_{max} - \beta_{min}) + \beta_{min}$$

where v' is the original value v normalized, α_{min} and α_{max} are the old min and max value, while β_{min} and β_{max} are the new min and max value, so in our case $[0, +1]$ or $[-1, +1]$.

2. **Decimal Scaling**:

$$v' = \frac{v}{10^j}$$

where j is the smallest integer such that $max|v'| < 1$.

Error Measures

Finally, after the learning process of our normalized data with a Backpropagation algorithm used in combination with the Cross-Validation technique (see the next chapter), we predict the next temporal value and we check the errors.
For measuring the error rate committed in our forecasting we used Mean Absolute Error (MAE) and Root Mean Square Error (RMSE):

$$MAE = 1/n * (\sum_{i=0}^{n} |f_i - y_i|)$$

$$RMSE = \sqrt{\frac{\sum_{i=0}^{n}(f_i - y_i)^2}{n}}$$

where f_i is the predicted value calculated with the forecasters and y_i is the real value we had in our dataset.

Dataset Splitting

The tests made with Backpropagation and Cross-Validation has a dataset split in three parts:

1. Training Set - 70% of set
2. Validation Set - 15% of set
3. Test Set - 15% of set

The tests made without the Cross-Validation has only a Training Set (still 70%) and a Test Set (30%). The dimension of global dataset increases starting from 0 to a limit number chosen by us with a sliding window which permit to slide the older value out of a tail and insert the newer. We see accurately the detail in the next chapter.

3.3 Design of Load Forecaster

This module is useful to provide the load value λ to the Decider, taking as input Δ and an arbitrary number of variable. The first step for doing that was to figure out which would have been the most relevant values to give to ANN for making the predictions the more accurate is possible. We explain below the ideas we followed. To obtain the best configuration we employed several ANNs and empirically we chose the configuration we were talking about before. The Figure 3.7 shows the ANN internal architecture.

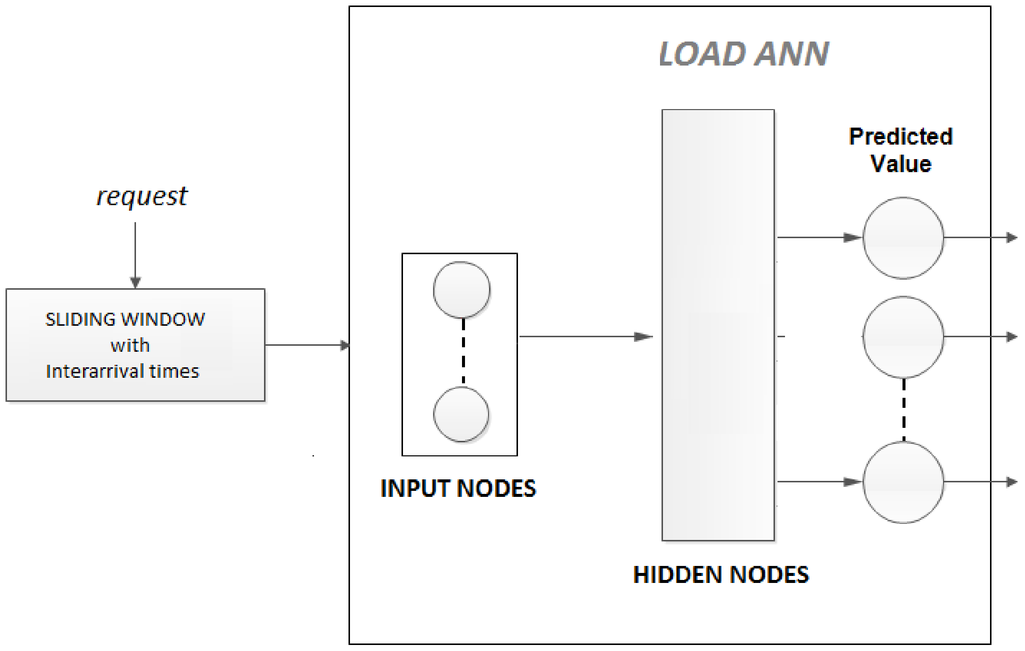


Figure 3.7: Load Forecaster ANN Architecture

The tests we made is with an increasing number of entry in a year of training set, the learning occurred with the *backprop* changing the *learning rate* and *momentum* term to find the optimal final configuration. Our tests mainly has fixed $d = 1$, but we shows a comparison between two predictions with two consecutive temporal instant.

We introduce now the different ANNs designed focusing on their efficiency in term of error rate and capabilities to adaptation to pattern changes.

3.3.1 ANN1: First Idea? Using the date

The first idea was to build an ANN with only 2 input nodes which was

1. hour

2. day of week

for we guessed the load with a big weekly periodicity. So, we started to build an ANN with 2 input nodes an increasing number of hidden neurons in a single layer until we found the best configuration and one output neuron which indicated the next predicted load.

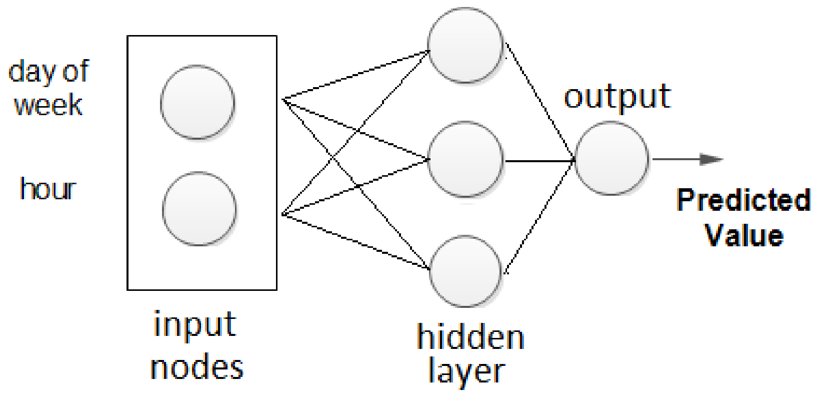


Figure 3.8: ANN1 Architecture

As said in chapter 3, a general method to find the optimal number of hidden nodes doesn't exist. Is known that a higher number of hidden nodes may help to be more accurate, but it is simpler to fall in the overfitting problem. However some guidelines to find the optimum umber of hidden nodes (where optimum means minimum number of nodes which gives a good accuracy) are available and we used an approach in which the number of hidden nodes is proportional to the number of input nodes. In particular we followed a mix of two approach, i.e. "$2n$" [72], "n" [65], where n is the number of input nodes.

The learning rate and momentum term are been chosen in empirical way and we find out that $\eta = 0.3$ gave the lower error rate $MAE = 4.5$ and $RMSE = 5.9$. Momentum term fixed to 0.5, but we noted not so many difference in term of error rate during its variation, but some difference in speed to converge. Below a view of the learning rate impact on error rate.

ACCURACY TEST for ANN1

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the ANN1 to fit real values of load (requests per hour). These test are made on different weeks, after the ANN learned a training set.

- **Experimental Settings:** The ANN tested is the best configuration as possible with 2 input nodes (representing weekly day and hour) so has 3 hidden neurons and 1 output (horizon of prediction $s = 1$), is trained offline with an year dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.3$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: The Figure 3.9 shows one of the week given to the ANN1 as test test. This week has a standard behavior and in such standard situation the ANN1 perform quite well and the pattern is fitted. The 7 fringes are corresponding to 7 days of a week, starting from Wednesday. The ANN with the learning of dataset has automatically understood that on Friday the load becoming to decrease and on Saturday and Sunday we have the lower load weekly value.

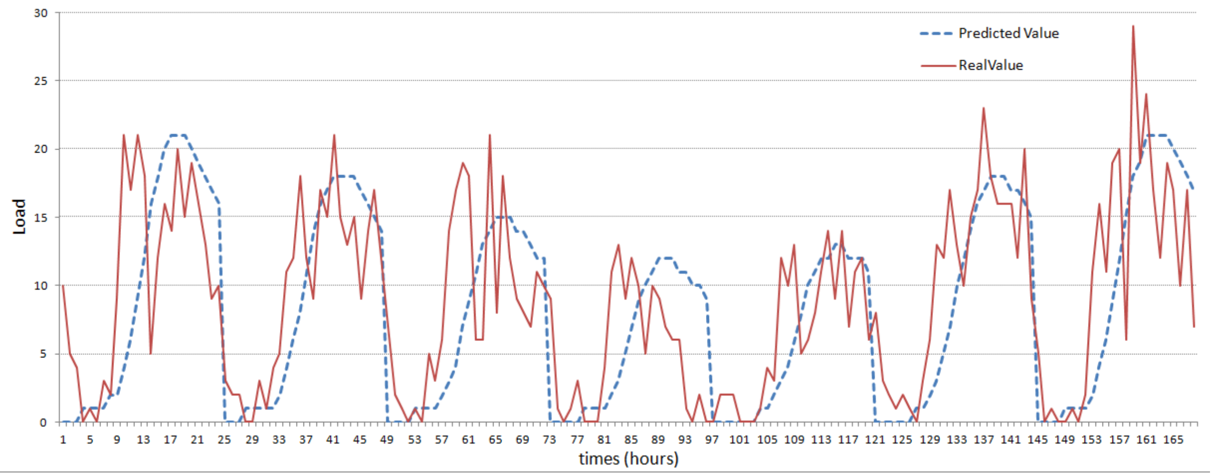


Figure 3.9: Accuracy Test on ANN1: weekly prediction made on the ANN1, with the load (requests per hours) by the increasing of the hours

Unfortunately there are some problems to make the ANN adaptable to wide fluctuations due to an average which the ANN calculates having only 2 input nodes, and more, the 2 input nodes regarding the day of the week and the hour so the forecasting is very similar each hour of every week. We call such issue the *"every week problem"* (Figure 3.10). Hence the dataset used here plays an important rule and we need to dimension pretty well the training set window, e.g. by maintaining only the last week we could improve the accuracy of next week, but we wish to be modular and trying to be the more independent as possible from training set window. More, this test is made with an offline learning which gives to the ANN the worst performances. We will focus this aspect later during the explanation of the impact of an online learning with a sliding window.

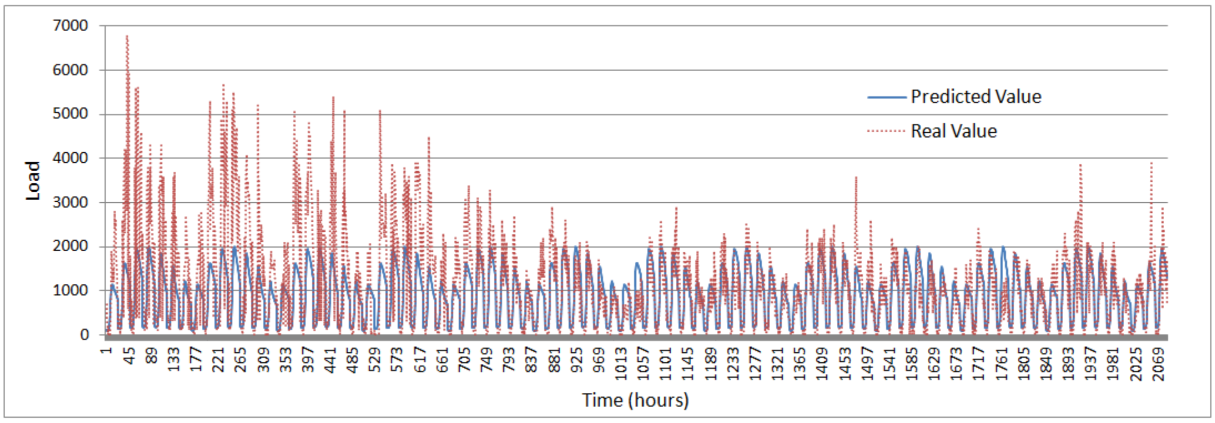


Figure 3.10: Accuracy Test: entire year prediction made on the ANN1, with the load (requests per hours) by the increasing of the hours. Emerging issues like *"every week problem"* due to having only 2 input nodes, an offline learning and a undue training set dimension

Learning Rate	**MAE**
0.25	0.46
0.3	***0.45***
0.4	0.46
0.5	0.47
0.75	0.49
0.9	0.51

Table 3.1: Table of error rate changing learning rate

3.3.2 ANN2: Improving the first ANN capabilities

In order to overcome the weakness due to the only 2 input nodes, the second idea was to build an ANN with other 2 input nodes for having a complete date and to differentiate different month week hours. The input nodes becoming so:

1. hour
2. day of week
3. day number
4. month

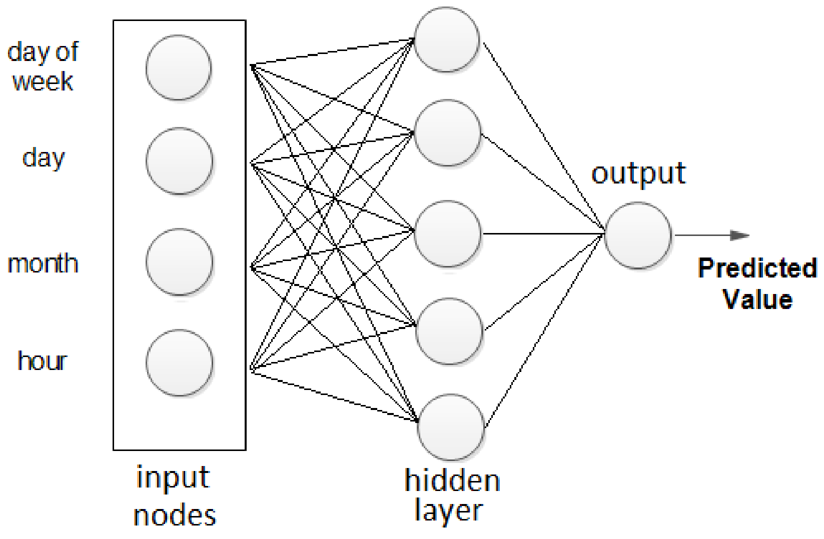


Figure 3.11: ANN2 Architecture

The test we made with this ANN gave us the better learning rate value equals to 0.35. As before, we fixed the momentum term to 0.5. Five neurons in the hidden layer gave us the best response even comparing with ANN with more nodes e.g. from 8 nodes to 11 or 80 nodes.

ACCURACY TEST for ANN2

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the ANN2 to fit real values of load (requests per hour). These test are made on different weeks, after the ANN learned a training set.

- **Experimental Settings:** The ANN tested is the best configuration as possible with 4 input nodes (representing weekly day, hour, day, month) so has 5 hidden nodes and 1 output (horizon of prediction $s = 1$), is trained offline with an year dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.3$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: As is possible to see in the Figure 3.12 the RMSE is $4,4\%$ and a MAE of $3,2\%$, so it is not so less than the ANN1 in a standard, but we overcame the "*every week*" weakness that we have in ANN1 and moreover the ANN2 can be more adaptable to wide fluctuations than the ANN1. That is known in literature for augmenting number of neurons the ANN is more adaptable to fluctuations. In general there is not a technique and the adaptable of ANN depends of *learning rate* as well.

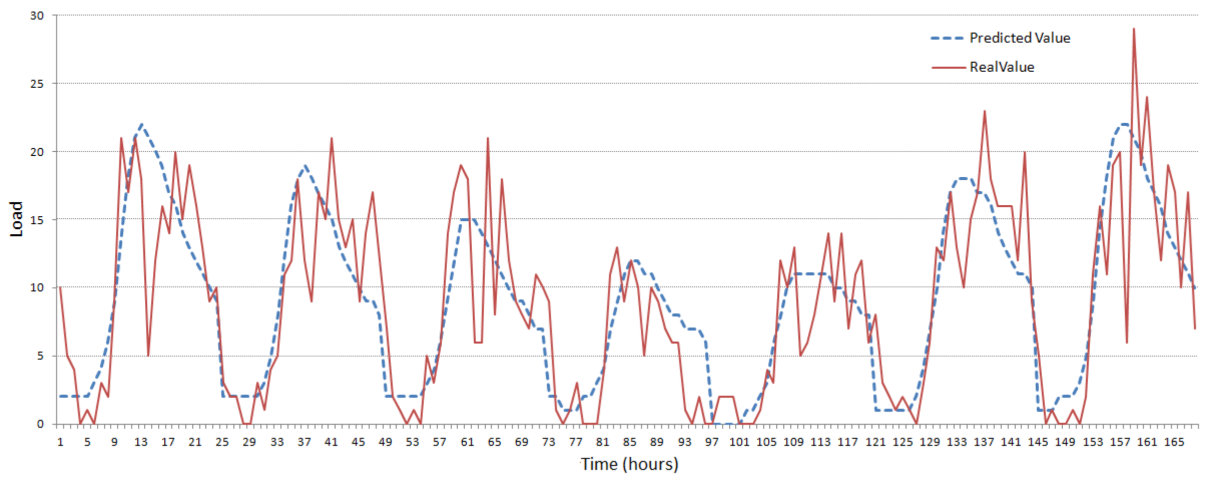


Figure 3.12: Accuracy Test on ANN2: weekly prediction made on the ANN2, with the load (requests per hours) by the increasing of the hours

This ANN give a better accuracy, but again the *every week* problem. Are we able to further improve the performances?

3.3.3 ANN3: Combining date and previous load

To improve the performances our idea was to build a *Time Series ANN* in addition to previous 4-input ANN. So our input nodes are:

1. hour
2. day of week
3. day number
4. month
5. an arbitrary number of previous load value

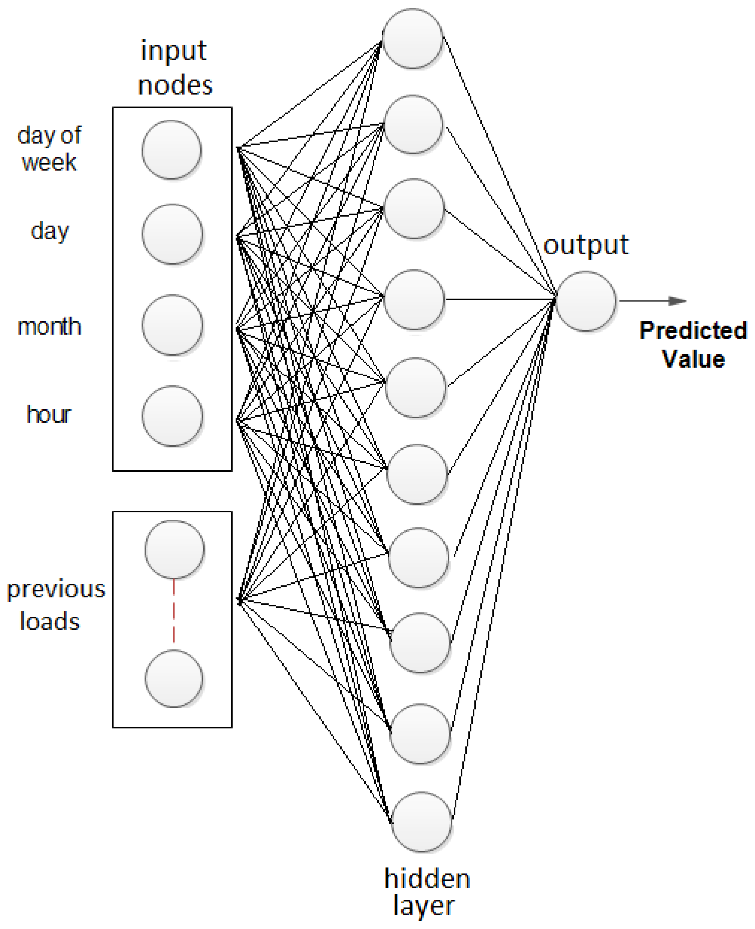


Figure 3.13: ANN3 Architecture

We ran several tests for discovering the best configuration. We find out that 10 hidden neurons, a learning rate equals to 0.4 and a momentum fixed to 0.5 gave the best performance.

To make the ANN more robust as possible, we wanted to minimize the previous load value as input, and having only the previous load value is enough to adapt quickly at changes of pattern as is possible to see in Figure 3.14. Having more previous load value as input may determine have old value not reliable.

ACCURACY TEST for ANN3

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the ANN3 to fit real values of load (requests per hour). These test are made on different weeks, after the ANN learned a training set.
- **Experimental Settings:** The ANN tested is the best configuration as possible with 5 input nodes (representing weekly day, hour, day, month and previous load) so has 10 hidden neurons and 1 output (horizon of prediction $s = 1$), is trained offline with an year dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.4$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: As is possible to see by Figure 3.14 the MAE is $2,9\%$, so it is not so less than the ANN1 and ANN2 in a standard week.

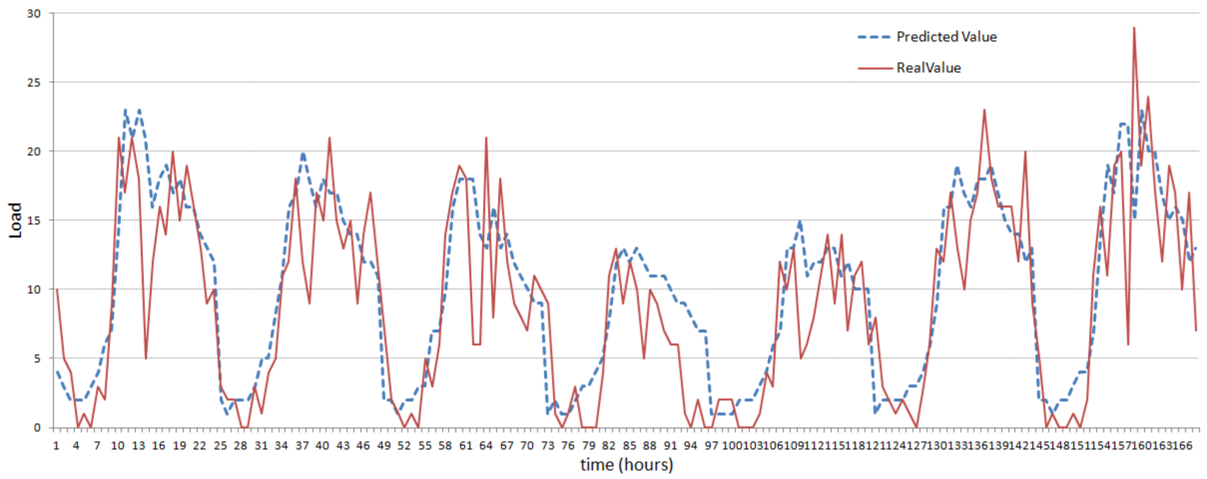


Figure 3.14: Accuracy Test on ANN3: weekly prediction made on the ANN3, with the load (requests per hours) by the increasing of the hours

ROBUSTNESS TEST for ANN3

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the ANN3 to fit real values of load (requests per hour) and test if it suffers the every week problem. These test are made on entire year, after the ANN learned a training set.
- **Experimental Settings:** The ANN tested is the best configuration as possible with 5 input nodes (representing weekly day, hour, day, month and previous load) so has 10 hidden neurons and 1 output (horizon of prediction $s = 1$), is trained offline with an year dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.4$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: This ANN first overcame the *"every week"* weakness that we have in ANN1, as possible to see by the Figure 3.15. We thought also that this kind of implementation could give to us a better accuracy response in changing pattern condition. We see tests about pattern condition in next sections.

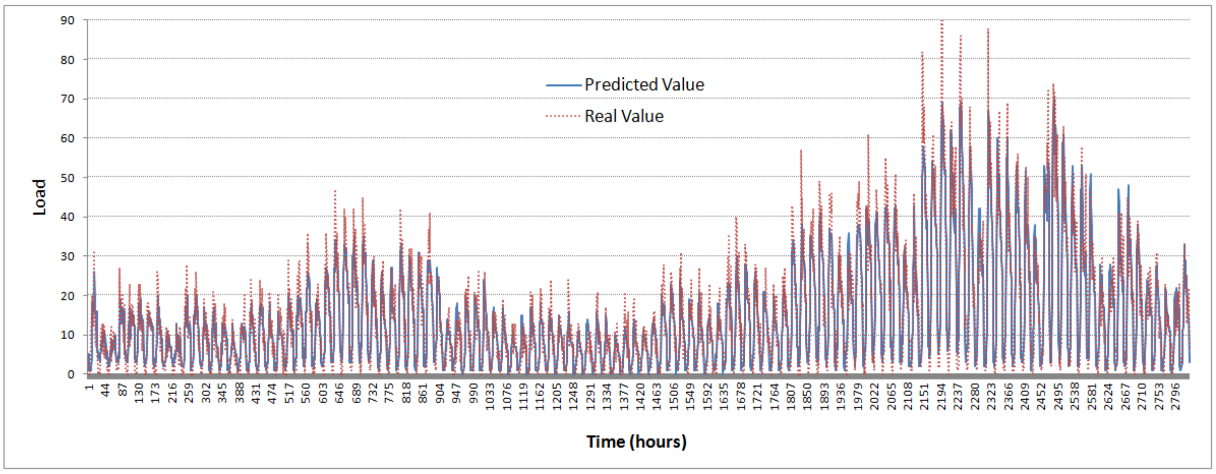


Figure 3.15: Accuracy Test: an entire year prediction made on the ANN2, with the load (requests per hours) by the increasing of the hours. Emerging issues like *"every week problem"* due without having previous loads as input nodes has been overcome

3.3.4 ANN4: is two layers better than one?

The ANN 3 is yet sufficiently to guarantee a good accuracy of predictions. It is known in literature that by dimension well a 2-hidden layers NN sometimes is possible to reach better performances with a smaller number of neurons rather than a 1-hidden layer and having more of 2 hidden layers is considered useless to improve such accuracy even is some kind of ANN could have better performances.

The Universal Approximation Theorem [26] says that a 1-hidden layer MLP with a restricted class of activation function (e.g. sigmoid [27]) is an universal approximator with any desirable accuracy among continuous functions on compact subset of $\Re^n$ just by changing number of neurons.

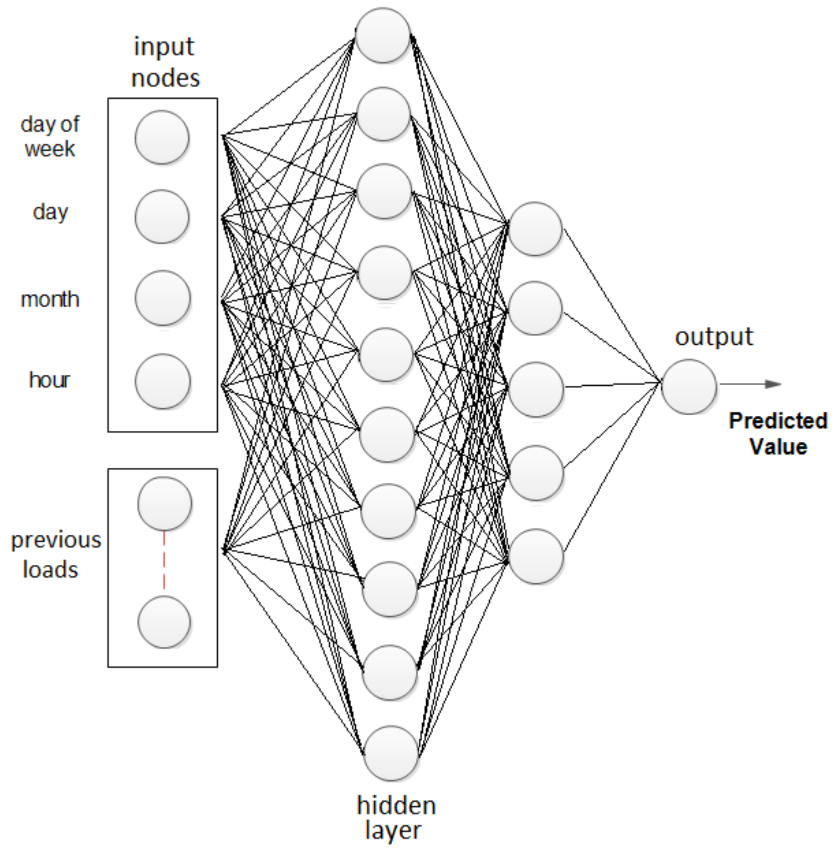


Figure 3.16: ANN4 Architecture

We try to built several ANNs and the best one results to be an ANN with the same input nodes and, regarding the hidden neurons, the policies "$2n$" (Wong, 1991) seems give the best accuracy response.

ACCURACY TEST for ANN4

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the ANN4 to fit real values of load (requests per hour). These test are made on different weeks, after the ANN learned a training set.
- **Experimental Settings:** The ANN tested is the best configuration as possible with 5 input nodes (representing weekly day, hour, day, month and previous load) so has 11 hidden neurons in the first layer and 1 output (horizon of prediction $s = 1$), is trained offline with an year dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.35$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: The 2-layers ANN has showed a great capabilities of adaptation to sudden changes due to the greater number of neurons spreaded in two layers while maintaining almost similar the accuracy of predictions in a standard week with a $MAE = 2.9$. Such capabilities is visible in the figure 3.17 at the hour 15 where the real value is 5, but the best previous value was given by the ANN which gave a value equals to 17, the MLP gave a value equals to 12 which is pretty nearer to 5. Obviously, having the number of neurons a linear impact on learning time, ANN4 is slower to be trained and need a larger number of iteration to converge so additional time if we want to set a larger number of iterations.

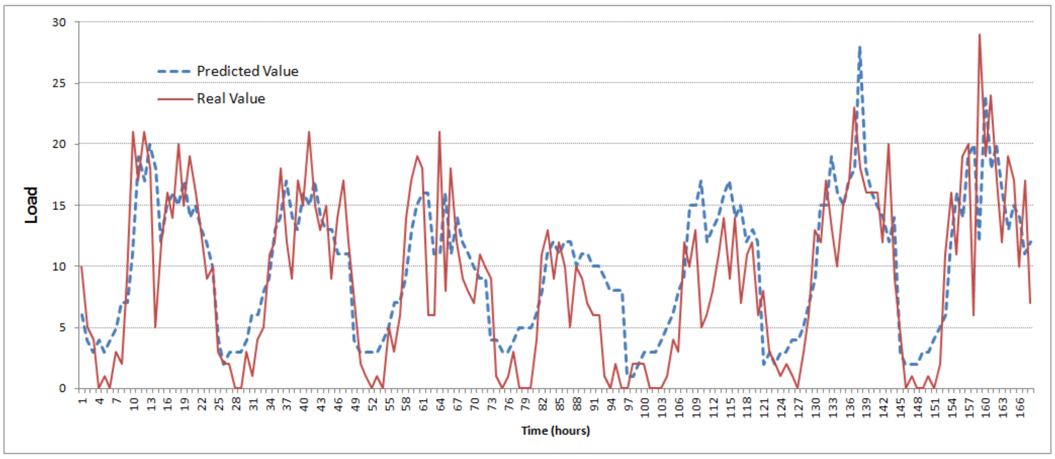


Figure 3.17: Accuracy Test on ANN4: weekly prediction made on the ANN4, with the load (requests per hours) by the increasing of the hours

3.4 Design of Distribution Forecaster

We briefed on chapter 2 the functionality of Δ-Distribution Forecaster. We described its partitioning in Distribution Recognizer and Distribution Predictor which contains the ANN and the Matcher. First of all, the data are a code empirically found for associating an integer number to each distribution. In out examples we codified the distribution as in the table 3.2. The ANN learn the history of distribution (represented with the code) and gives as output a number. This number goes into the Matcher whose work is simply to match that number with the nearest numerical *id* associated to distribution. The distribution *id* chosen is the final output of this module and goes towards The decider. Differently from the Load Fore-

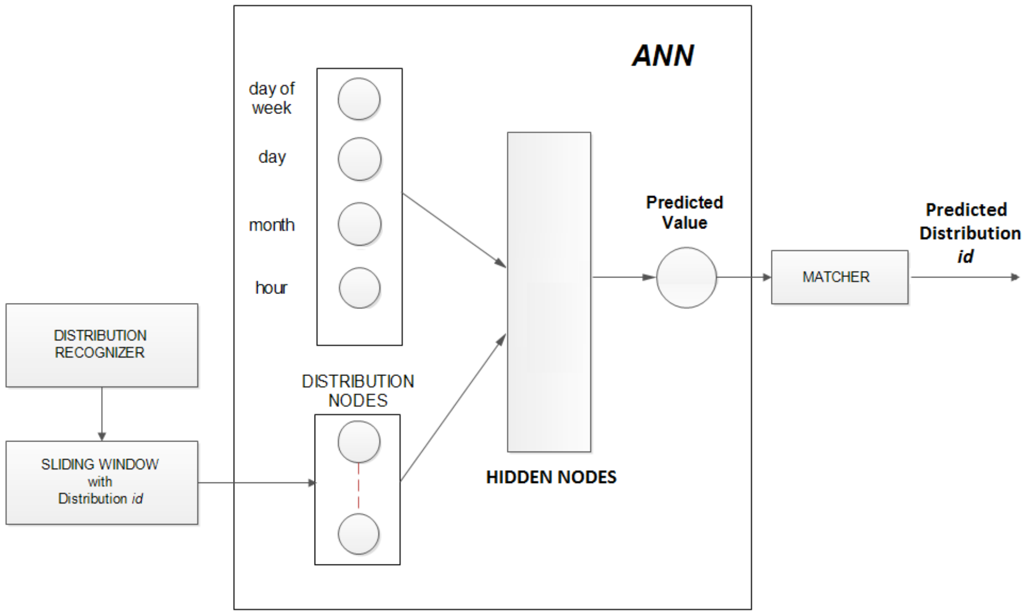


Figure 3.18: Δ-Distribution Forecaster Architecture

caster, this ANN must have a capabilities to jump like an impulse from a value to another one, so the main difference from the other networks employed are basically the number of neurons which composed the hidden layer and the setting of the learning rate. The Backprop algorithm has pretty much different behaviors corresponding to different configurations of number of nodes and learning rate. In particular by augmenting the number of hidden nodes the ANN is more "powerful" i.e. has more capability to make accurate predictions, while a changing of learning rate implies a different attitude of ANN to sudden changes. Remembering that the learning rate must be choice in a range $[0; 1]$, choosing a low value implies a smooth adaptation to sudden changes, on the contrary a high value implies a good adaptation. But this value must choose accurately, because different situations need different values.

ACCURACY TEST for Distribution ANN

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the Distribution Forecaster. The results are to consider without the Matcher influences, so only the predicted values of the ANN with the distribution codified. These test are after the ANN learned a training set.

- **Experimental Settings:** The ANN tested is the best configuration as possible with 5 input nodes (representing weekly day, hour, day, month and previous distribution) so has 150 hidden neurons and 1 output (horizon of prediction $s = 1$), is trained offline with a syntetic weekly dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.9$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: The results visible by Figure 3.19 proof the capabilities of that ANN to jump from a value to another while maintaining a good accuracy.

Distribution	**numerical** *id*
Poisson	0
Uniform	3
Exponential	11

Table 3.2: Distribution Encoding

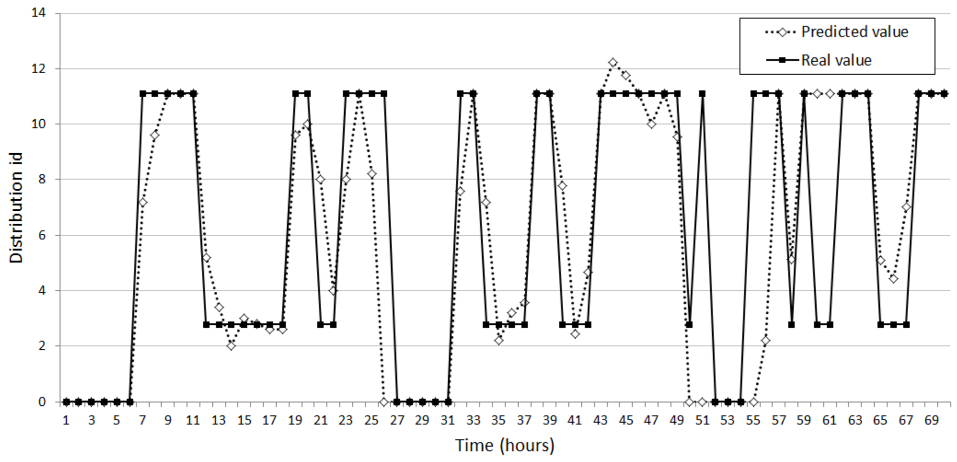


Figure 3.19: Accuracy Test on Distribution ANN: 72-hours prediction with the numerical id by the increasing of the hours

3.5 Setting the Horizon of Prediction

The main interests for this work is the next temporal instant, but we wanted to build ANNs with a larger temporal range to improve the efficiency and the capabilities of prediction for in case future applications.

It is called **Horizon of Prediction** the range in which we are able to make a forecast. Our tests are made with a dataset split in hours, so technically to forecast the next value we have time in order of an hour and it is not a problem taking this time. Having a dataset with a smaller granularity we instead may be interest to quicker and try to predict more following temporal instant. The Backpropagation learning time in some applications for example could push us to predict a larger horizon. Wee see the Backpropagation learning time difference between the ANN in a next section, we focus in this section to the way to choose an horizon of prediction with a value d.

The tests seen until now are all made with the forecasting of the next temporal instant, so $\Delta = 1$. Providing a $\Delta = k > 1$ means augmenting the horizon of prediction from $s = l$ to $s = k$. The ANN will have hence $d = k$ output corresponding to each following temporal time. Obviously our hypothesis was that the farther we try to predict the less accurate we are, but we want to evaluate the error rate by increasing the horizon in order to discover how many output nodes could give us a good tradeoff between low error and large horizon of prediction.

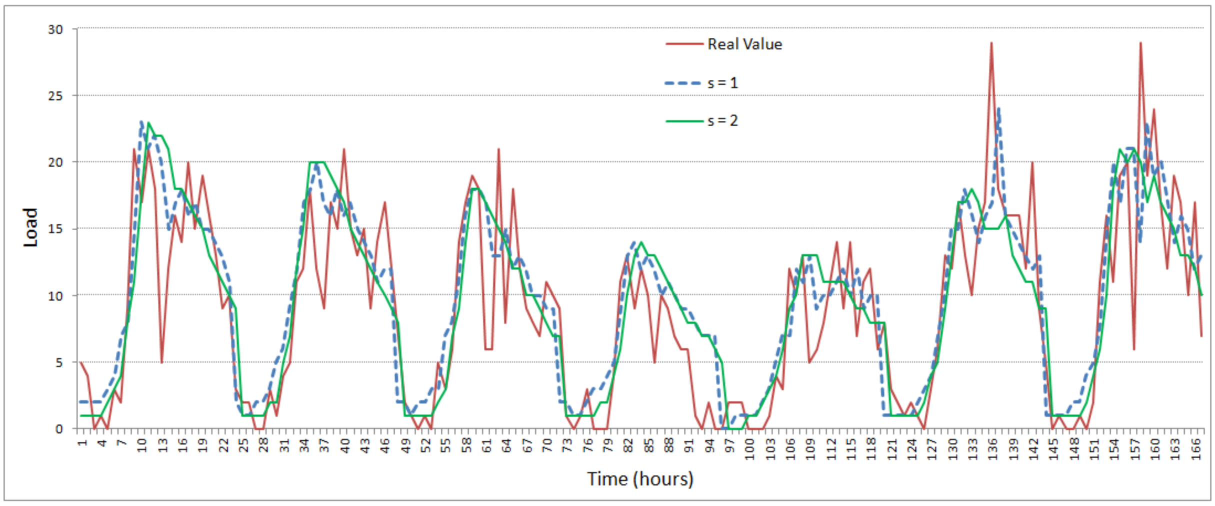


Figure 3.20: Comparison between predictions made with s=1 and s=2

EFFICIENCY TEST by increasing the Horizon of Prediction

- **Test Description:** This test series wants to evaluate the prediction accuracy capabilities of the ANN3 to fit real values of load (requests per hour) at the changing of horizon of prediction in order to find out which is the best horizon of prediction that give us a large horizon while maintaining a low error. These test are made on different weeks, after the ANN learned a training set.

- **Experimental Settings:** The ANN tested is the best configuration as possible of ANN3, i.e. 5 input nodes (representing weekly day, hour, day, month and previous load) so has 10 hidden neurons and 20 output (horizon of prediction $s = 20$), is trained offline with an year dataset by using the Backpropagation algorithm, with a learning rate $\eta = 0.4$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation.

Results: The results shown by Figure 3.21 were found in many other tests and we found out an unexpected good result. The first 3 hours are always predicted with a similar low error. Sometimes could happen that the prediction made with $s = 2$ give better response that $s = 1$. Starting from $s = 4$ the error rate start increasing following the horizon of prediction. So the best result is given by a network with $d = 3$ output nodes.

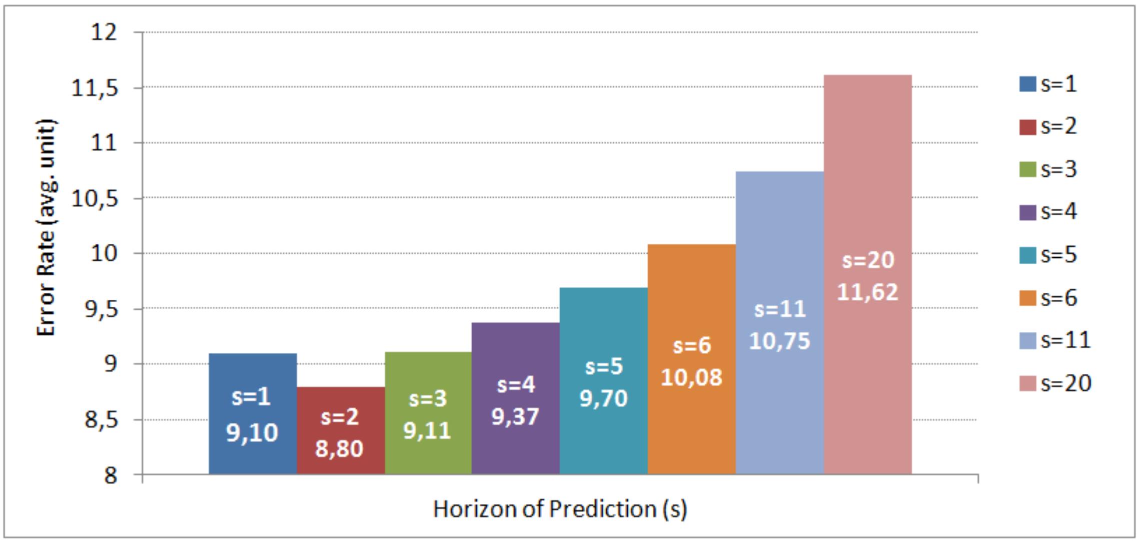


Figure 3.21: Efficiency Test by Increasing the Horizon of Prediction: comparison between the error committed on d output nodes. It is possible to note how the first 3 hours has the lowest error rate

3.6 Setting the Training Dimension

In the previous sections we presented the ANN designed with their accuracy performances measured with an off-line learning, i.e. by calculating the forecasting error rate committed with respect to the real values on a test set, just having a training set with a fixed dimension.

One of the thing to consider is how big has to be the training set window. Obviously the increasing of the training set dimension gives a better accuracy, but the ANN become slower and slower to train, it would need more memory and the training cannot increases its dimension towards infinite, but it has to be limited. We wanted to focus our attention on that, so we studied how changes the error rate by increasing the training set dimension.

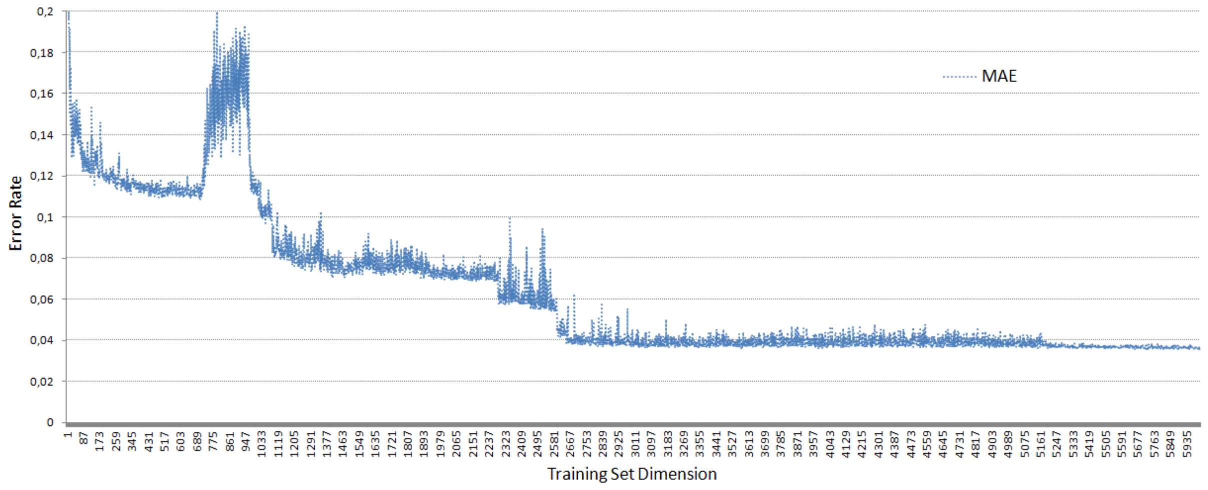


Figure 3.22: Error rate by increasing the hours of an entire training set dimension

EFFICIENCY TEST by increasing the Training Set dimension

- **Test Description:** The aim is to find the optimum size of training set to train offline the ANN, considering that increasing the training set dimension means increasing the learning time of *backprop* algorithm. Hence we want to find out the best tradeoff between error rate and dimension.

- **Experimental Settings:** The ANN tested is the best configuration as possible of ANN2, i.e. 4 input nodes (representing weekly day, hour, day, month), 5 hidden neurons and 1 output (horizon of prediction $s = 1$). The ANN is trained offline starting with an empty dataset and by increasing its dimension for each iteration. The learning is made with the Backpropagation algorithm, with a learning rate $\eta = 0.4$, a momentum $\beta = 0.5$ and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation. The test set is always the same fixed week.

Results: The Figure 3.22 shows the error rate committed on the test set with respect by the increasing of the training set dimension. After only a day (24 hours) we have a dataset $\{x(0) = req_0, x(1) = req_1, \cdots x(23) = req_{23}\}$ and the error rate of predictions decreases from the 100% committed with the empty dataset (first prediction) under to a threshold of 20% (Figure 3.23).

To go under a lower threshold we need more records. We should have a dataset of a year to give to the ANN the entire knowledge of an year pattern, as possible to see in Figure 3.22. The spikes around the iteration 600 occurring to a pattern changes due to a period of unavailability of servers in that period. It is possible to see that 3000 entry seems be a good tradeoff to be under the error threshold of 5% using only 4 months of dataset instead of a entire year dataset.

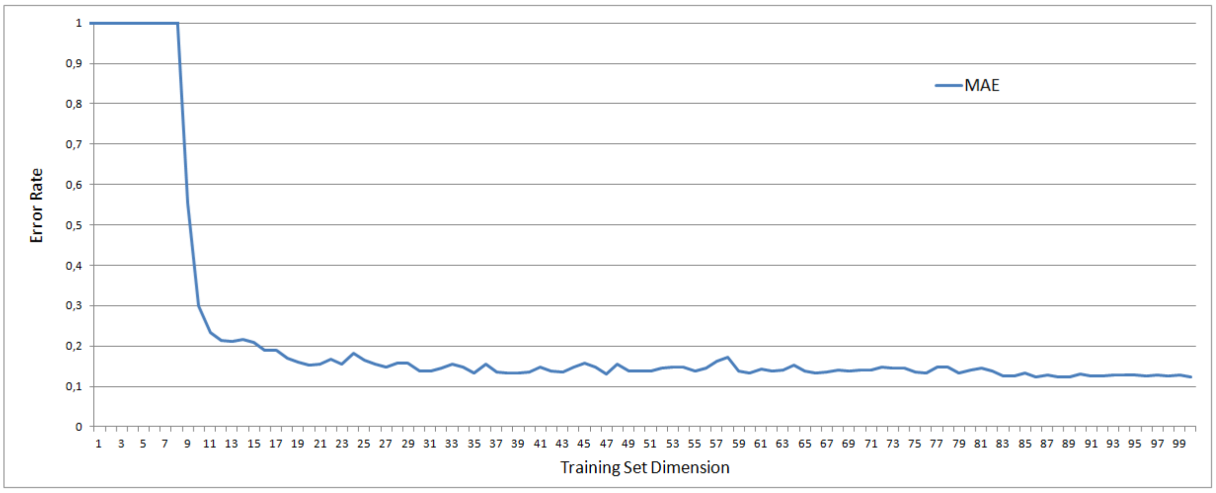


Figure 3.23: Error rate by increasing the firsts training set dimension

These results are the consequence of an off-line learning. Let's see now how we switched to an on-line approach and the impact on the training set dimension.

3.6.1 On-Line Learning: The Sliding Window

The tests seen helped us to design the Forecasters and to improve their performances in an off-line model, but MYSE Framework is built to work on-line with real time requests so such model is not reliable for our reference scenario.

Obviously using an approach date-based it make sense having a year dataset, in order to be ready to predict every temporal instant because we the ANN has a knowledge of each previous year temporal instant and, in general, having a bigger dataset could help to be more robust giving to the ANN more entry to study.

We saw before how to dimension the training set in the off-line scenario and we saw that 3000 entry is a good tradeoff between error rate and training dimension, so now in order to make the MYSE Framework able to work on-line we present the Sliding Window.

The goal is to find the optimum size dimension win of training set and to carry on the last d value by using a Sliding Window (SW). So at every new request, the oldest request (tail) get off the window, all the previously requests $req_0, req_1, req_2 \ldots, req_{d-1}$ slide back of 1 unit in the window and the new request become req_0.

ROBUSTNESS TEST of ANNs by increasing the Sliding Window dimension

- **Test Description:** The accuracy tests made off-line gave a clear result: the ANN3 performs better than the previous version. The following tests are made again on both the ANNs ANN2 & ANN3 in order to figure out their adaptation capabilities (robustness) to pattern changes as well. The aim is to understand the impact that the Sliding Window has on the robustness in order to find out the best dimension which give a good tradeoff between prediction accuracy and speed of convergence.

- **Experimental Settings:** The ANN tested is the best configuration as possible of ANN2 & ANN3 yet describe previously. The ANN is trained online starting with an empty dataset and by increasing its dimension for each iteration until reaching the dimension set win. Then only the last win remain in the training set following the sliding window technique described above. Each tests of this series has a different set win. The learning is made with the Backpropagation algorithm, with learning rate and momentum fixed by following the best values found out in previously test and a fixed iteration number as stop policies set to 100 without neither an early stop technique and a Cross-Validation. The test set is at each iteration just the next value. The training set is a synthetic dataset built with the original dataset concatenated with data pretty different (e.g. sin function, or 5-times scaled up) and the robustness is tested in the concatenation points.

Results: The results obtained confirm the accuracy tests results, i.e. the ANN3 performs better than the other ANNs, but while the accuracy tests were comparable each other, these tests give performances significantly better to ANN3. The main and the best important thing is that the ANN3 results to be independent from the SW dimension, i.e. the ANN has the same response with significantly different dimensions, while the ANN2 is still dependent on the SW dimension. We can see the difference of robustness between the ANN2 and the ANN3 by observing the figure 3.24 and 3.25

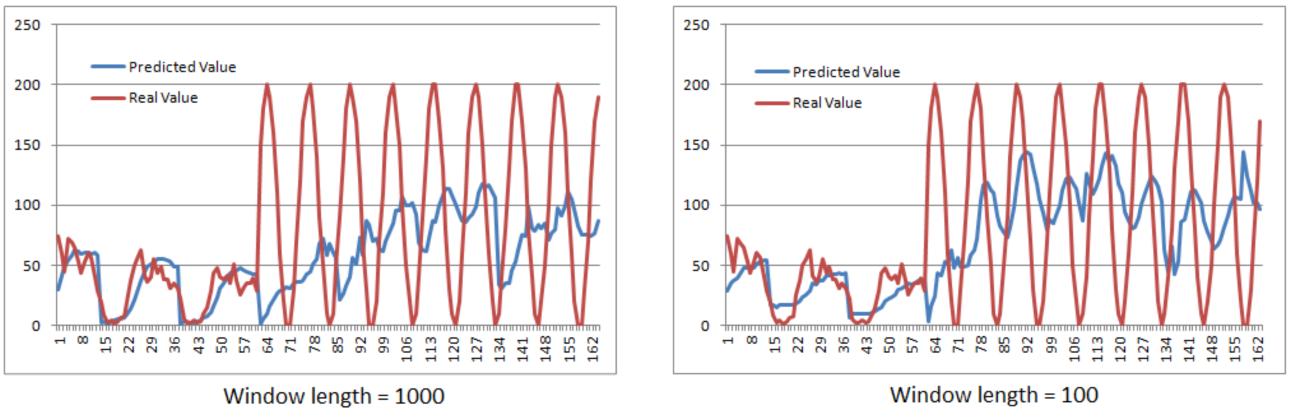


Figure 3.24: Change pattern situation in ANN2 with 2 window length

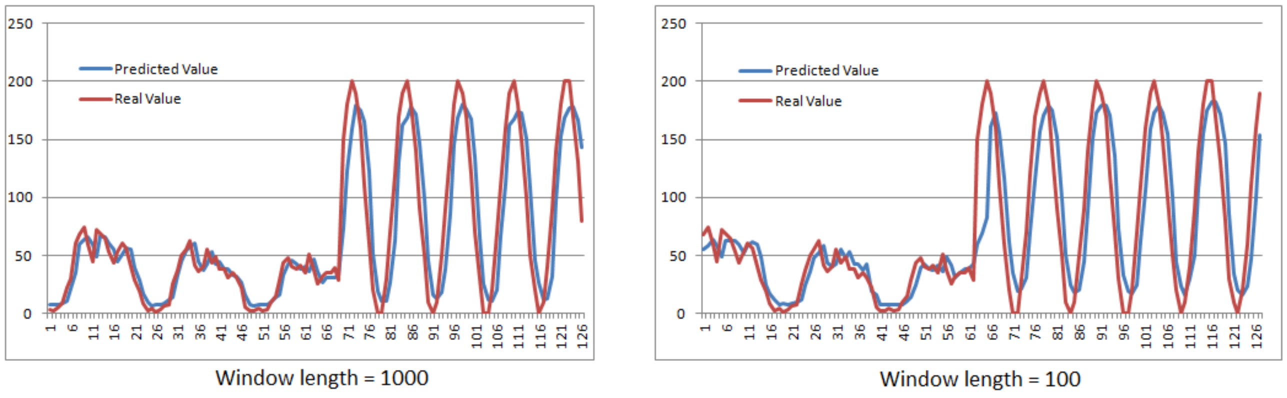


Figure 3.25: Change pattern situation in ANN3 with 2 window length

These are synthetic datasets obtained by the original dataset provided by Google Analytics for our site server department, concatenated with a scaled *sin* function. Other tests with a synthetic dataset are made for demonstrate the robustness of our designed ANN to this sudden changes. By these figure above is pretty simple to note how the introduction of the previous load value has significantly reduced the importance of the dataset length, in fact while the ANN2 suffer to a normal traffic situation and in particular a change of pattern with both of 100 entry and 1000 entry dataset, the ANN3 has great capabilities of prediction either with a 1000 entry dataset and with a 100 entry dataset.

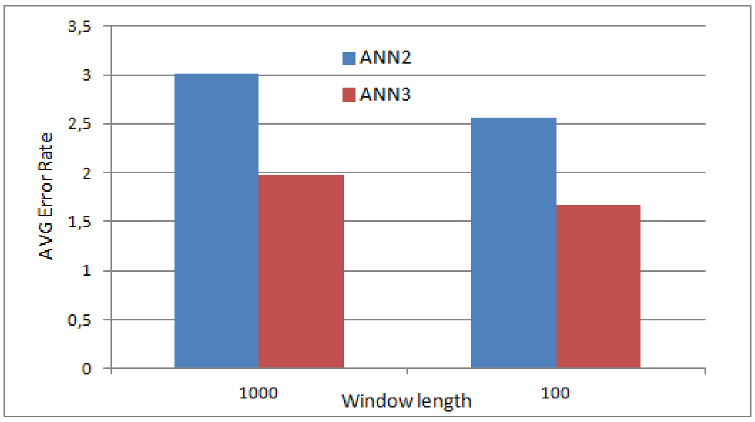


Figure 3.26: Comparison between avg error rate in change pattern situation

It is possible also to see how the average error rate in a change pattern situation is lower in the ANN3 by observing the figure below. While the error rate in the ANN2 is for the window length equals to 1000 and to 100 respectively 3.02 and 2.56, for the ANN3 is 1.98 for the window length equals to 1000 and 1.67 for the window fixed to 100. So even using a 10-times smaller window for the ANN3 we have a better performance than we have with the ANN2 based only on the date.

To stress more our ANNs we tried to change pattern from the original dataset to a dataset composed by a real part of dataset concatenated with the real dataet scaled x5, again with a standard dataset and then with *sin* scaled function. The Figure 3.27 shows the capabilities of ANN3 to quickly adapt to new conditions

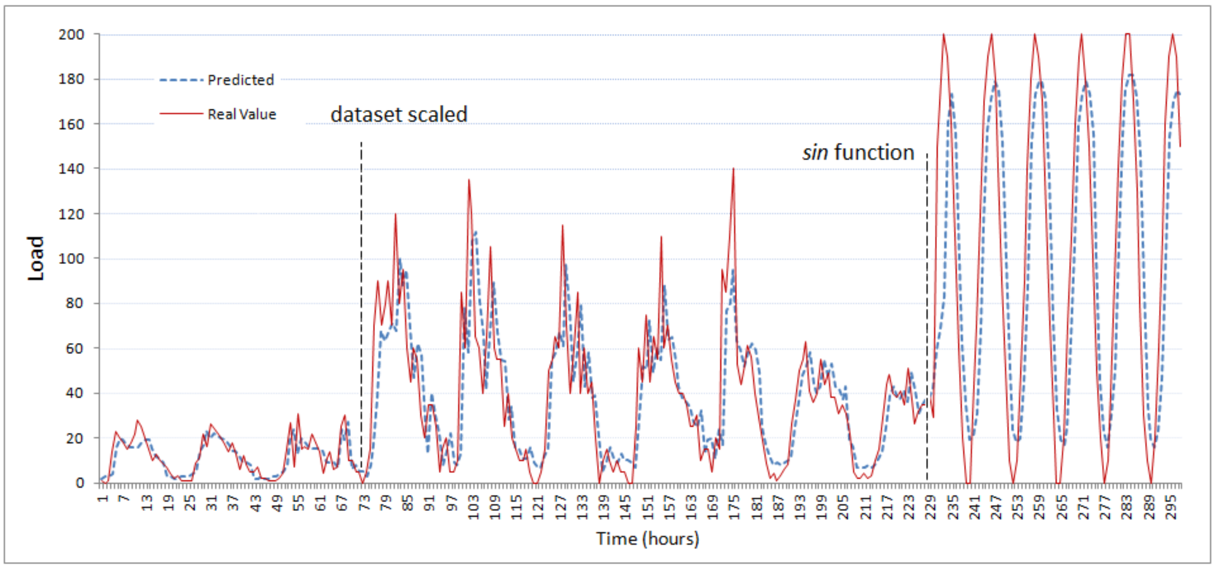


Figure 3.27: Adaptation capabilities of MYSE ANN

3.7 Backpropagation Algorithms

In this section we talk about the *Backprop* used, in particular we focused our attention on the iteration the algorithm needed for making good predictions avoiding waste of time and the techniques used for reducing the *overfitting* problem. For choosing when the algorithm has to stop the iterations there exist 3 main different ways:

1. stop when the error on validation set is under a threshold
2. stop after a fixed number of iterations
3. stop when overfitting occurring (Early Stopping techniques)

The first approach need to choose a threshold which gives us a good accuracy and it depends on application. We saw dimensioning the training set in section 3.6 that a good threshold of 5%. Choosing manually the iteration number to stop the algorithm needs for some strategy. We wanted to find out the best tradeoff between speed of learning and accuracy of prediction by following the results given by the test below.

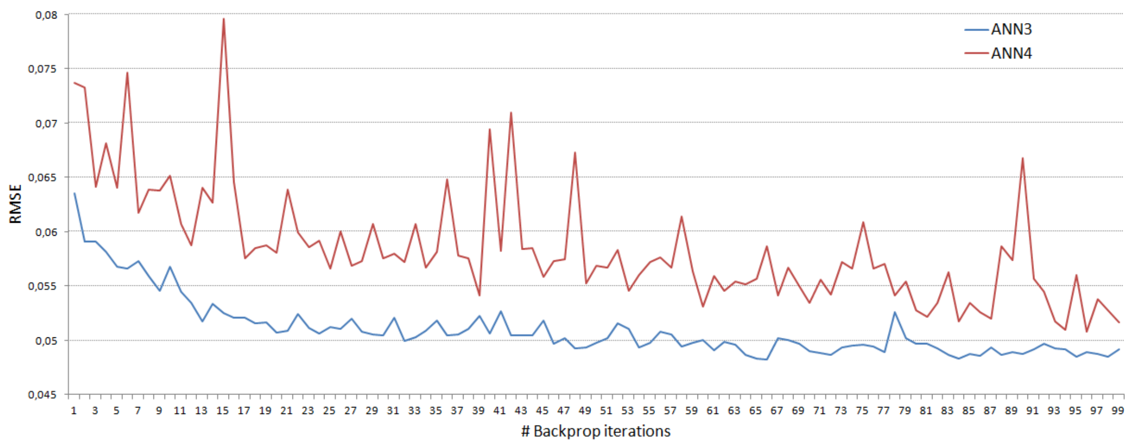


Figure 3.28: Comparison between ANN3 and ANN4 on RMSE by increasing the backprop iterations

EFFICIENCY TEST of Backpropagation

- **Test Description:** This tests aim to find out the best number of fixed iterations for Backpropagation by studying the average error rate with respect to the iterations number.
- **Experimental Settings:** The ANN tested is the best configuration as possible of ANN3 & ANN4 yet describe previously. The ANN is trained offline with a fix dimension

training set (3000 hours) with learning rate and momentum fixed by following the best values found out in previously test without considering possible overfitting.

Results: The results obtained confirm the accuracy and robustness tests results about which is the best ANN designed, i.e. the ANN3. In Figure 3.29 we can see the ANN4 is able to go down under the threshold of 5% of RMSE after 200 iterations, while the ANN3 need for the same scope 30-40 iterations (Figure 3.28) and in general it don't need usually more of 100 iterations. That numbers of iterations give us an idea of how many times we need to train the ANN.

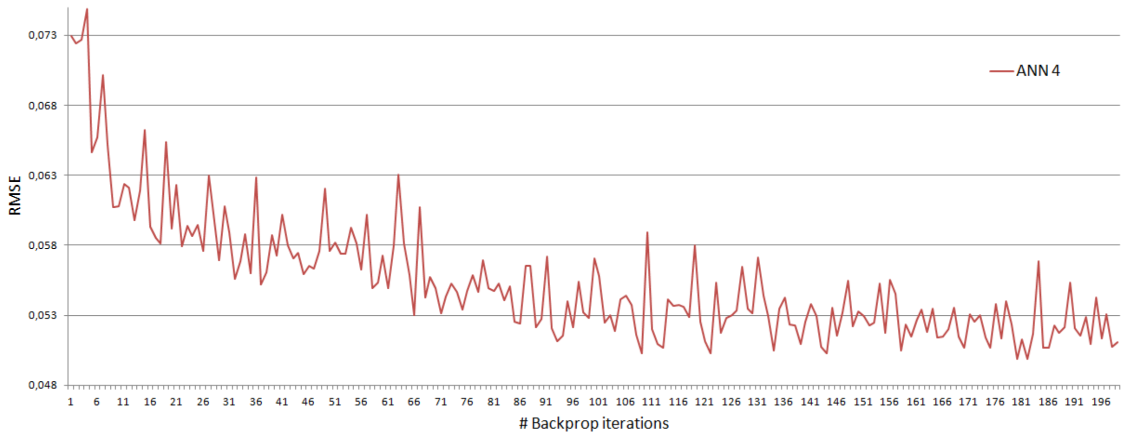


Figure 3.29: Trend of ANN4 RMSE by increasing the backprop iterations

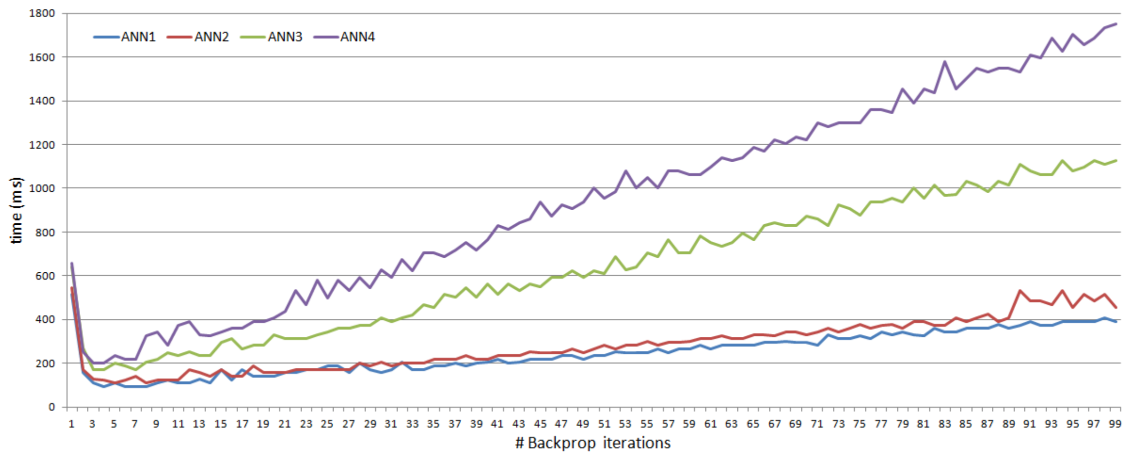


Figure 3.30: Trend of learning time by increasing the backprop iterations of 4 ANNs

The learning time is linearly dependent from the number of neurons, hence having the ANN3 a lower number of neurons seems be the best ANN designed, giving the best tradeoff between accuracy, speed of learning and robustness at pattern changes.

3.7.1 Comparison between different Backpropagation versions

In order to improve the forecasting capabilities we have yet reached with Backpropagation, we tried different versions and we made a comparison. In particular we tried the following three version of such algorithm:

1. Backpropagation
2. Momentum Backpropagation
3. Resilient Backpropagation

Tests made in other works demonstrated the RPROP is the best algorithm for ANN right now and in our case we obtain the same result.

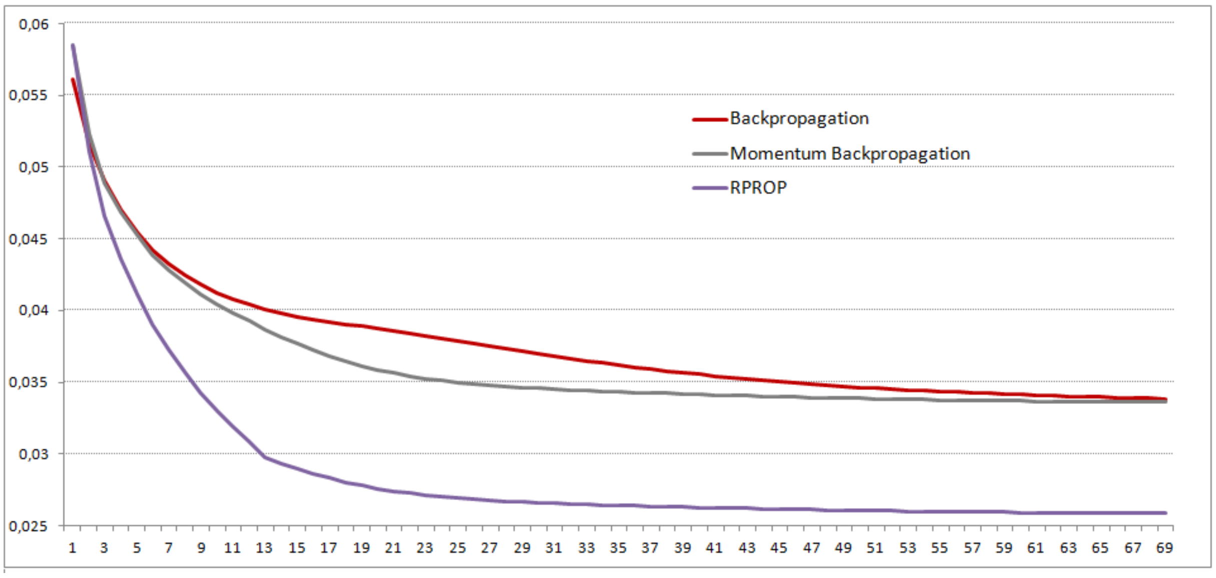


Figure 3.31: Error rate comparison between 3 kinds of backpropagation algorithms

The best performance given by the RPROP makes possible to use our ANN4 - Multilayer Perceptron (MLP) with such algorithm in order to have a best accuracy together with acceptable training time.

3.7.2 Overfitting Management

The **overfitting** occurs when we have an undue training and\or an insufficient number of entry in a training set. The consequence of overfitting is a great adaption to the characteristics which are proper of training set, but the lose of generalization capability. So the model (in our case the ANN) would be powerful to predict the same data we have in the training set, but with worst performance to forecast new value.

To discover when the ANN is going to overfitting, for while the error on training set is decreasing, the error on a validation set become to grow up. The common used techniques are based on **Early Stopping**, i.e. to save at each iteration of training the weight of each neurons so if at iteration t overfitting occurring we reload the safe weight for each neurons and we have the best setting. But the gradient descent algorithms suffer to the local minima, so one of the problem of this approach is to find when stopping the iteration [51].

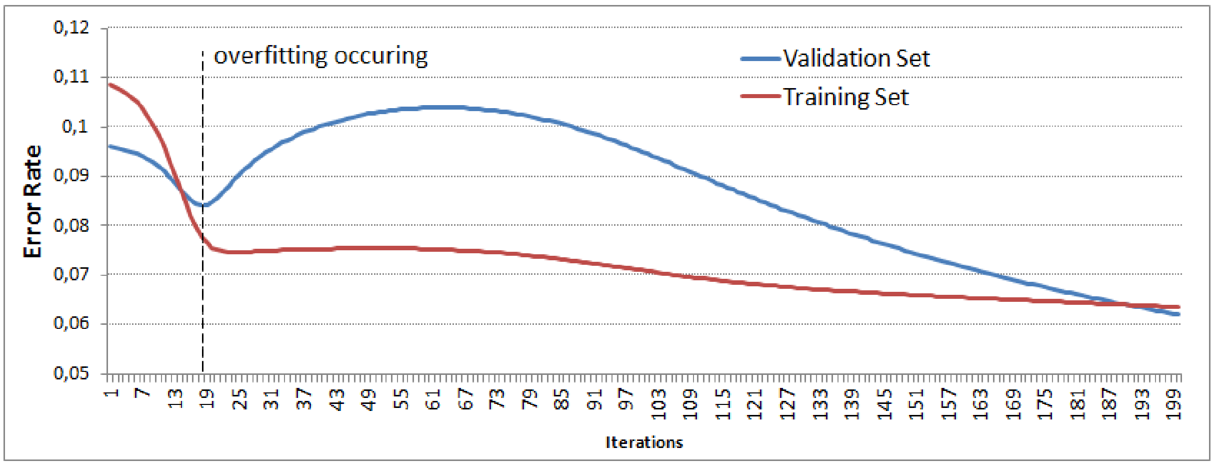


Figure 3.32: Overfitting occurs by increasing the backprop iterations

Figure 3.32 shows an overfitting occurs to ANN after 19 iterations, but it is possible to see also that after 200 iterations the performances are better then these are at iteration 19. This is due to the local minima problem suffered by the algorithm based on gradient descent. In order to avoid or reduce such problem there exist different solutions (e.g. cross-validation, regularization, pruning, Bayesian priors on parameters or model comparison). We chose to implement the cross-validation method.

Cross Validation:

Cross-validation, also called rotation estimation, is validation technique [32] used especially to generalize the predictions capability of a machine learning model from a a given data set.

A round of cross-validation consists in the partitioning of dataset into a training set (to train the model) and a validation set (to validate the model). The aim of that partitioning is to reduce variability so for this scope are performed multiple rounds using different partitions, and the validation results are averaged over the rounds.

There exist different types of Cross Validation. One of the most commons is the ***K*-fold Cross Validation**. The partitioning happen into k subsets and rotary one of them is the validation while the other $k-1$ form the training set, having so k repetitions (dimension of fold) in which each of the k subsamples used exactly once as the validation data. Than the k results can be combined (e.g. averaged) to generate a unique estimation. The great advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once. It is very common to set $k = 10$ [46] but in general k is an unfixed parameter.

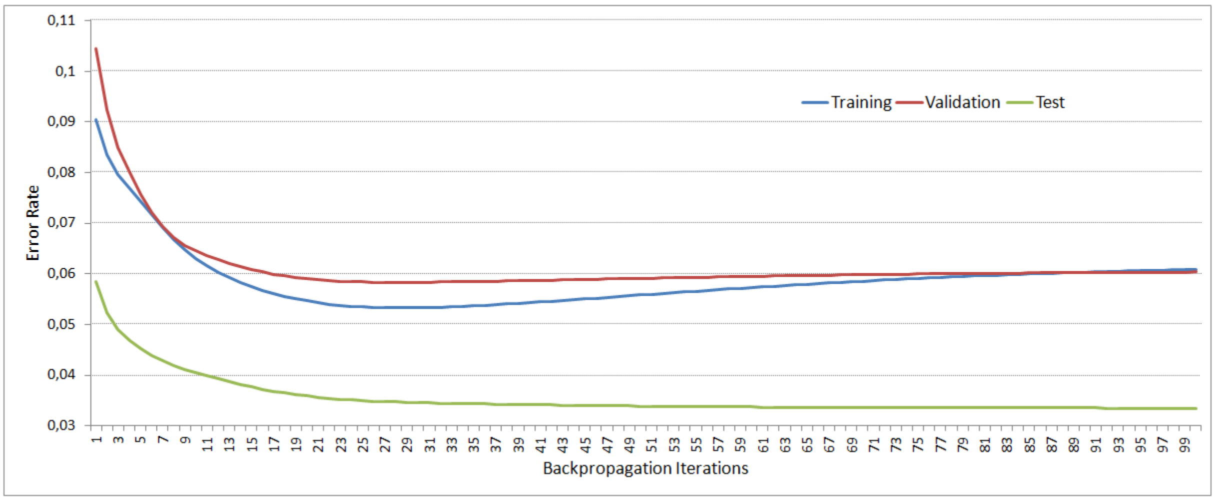


Figure 3.33: Overfitting doesn't occur by increasing the backprop iterations

MYSE Framework is designed to test if the ANN is suffering the overfitting problem or not. Usually the ANN3 (considered the best) doesn't suffer so much that problem how is possible to see by Figure 3.33, however the Forecasters contains a specific software module to figure out when the ANN is going to overfitting and in such case run the Cross-Validation.

Chapter 4

Evaluation of Overall MYSE Framework

In this chapter first of all we present the Decider and the algorithm the reconfigure the system with a heuristic based on finding the optimum. We evaluate the functionality of the overall MYSE Framework with the impact it had on the number of replicas by exploiting also a comparison between itself and a Non-Trained MYSE (NT-MYSE), which is the same Framework, but uses a reactive approach, in order to determine the importance of being proactive.

4.1 The Decider Module

The Decider module by taking as input λ and the distribution *id* for the arrival schema and μ and the distribution *id* for the service schema has to make a decision on the number of replica to give a configuration towards the Configuration Manager of replicated service.

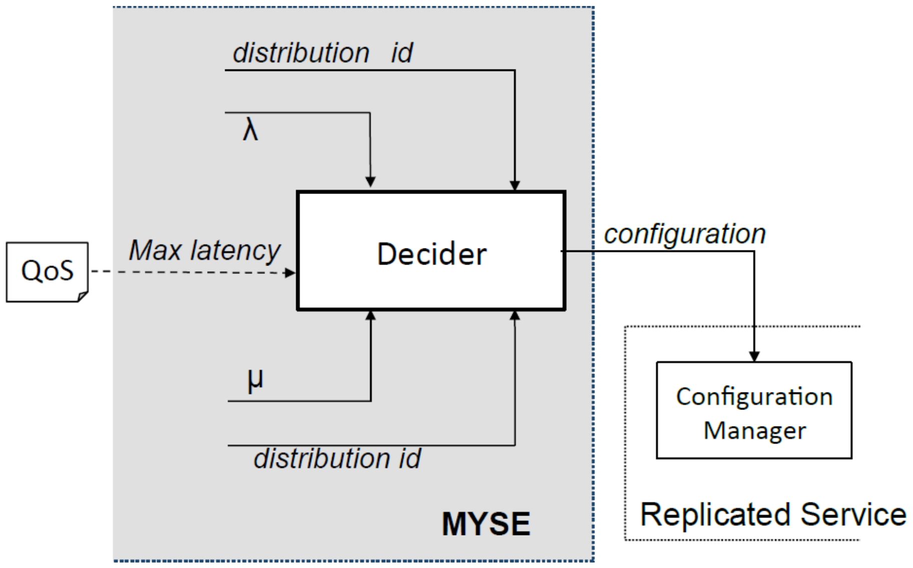


Figure 4.1: Decider Architecture

Several models can be employed into the Decider and each queueing model has, the main one employed to make our tests is the M/M/S which has the following formulas:

$$T = T^q + \frac{1}{\mu}$$

$$T^q = \frac{N^q}{\lambda}$$

$$N^q = \frac{1}{s!}\left(\frac{\lambda}{\mu}\right)^s \frac{\rho}{(1-\rho)^2} p_0$$

$$N = \lambda T = N^q + \frac{\lambda}{\mu}$$

$$p_0 = \frac{1}{\sum_{n=0}^{s-1} \frac{1}{n!}\left(\frac{\lambda}{\mu}\right)^n + \frac{1}{s!}\left(\frac{\lambda}{\mu}\right)^s \frac{1}{1-\rho}}$$

The algorithm employed in this work is simply based on the optimum number of servers to ensure QoS, without considering the QoS violation made in the past or other problem like flipping. Other heuristics are employed [74], but we employed just the simplest one because our goal was making a comparison with the reactive NT-MYSE with respect to the optimum number of replicas.

Algorithm Phases

1. **Picks** the right queueing model according to the input received (in our tests we fixed this value to M/M/S)
2. **Calculates** the expected latencies by using the formula of the queueing model chosen
3. **Verifies** if the QoS is ensured with the current configuration and if it is ensured with the minimum number of servers
4. **Decides** the new configuration

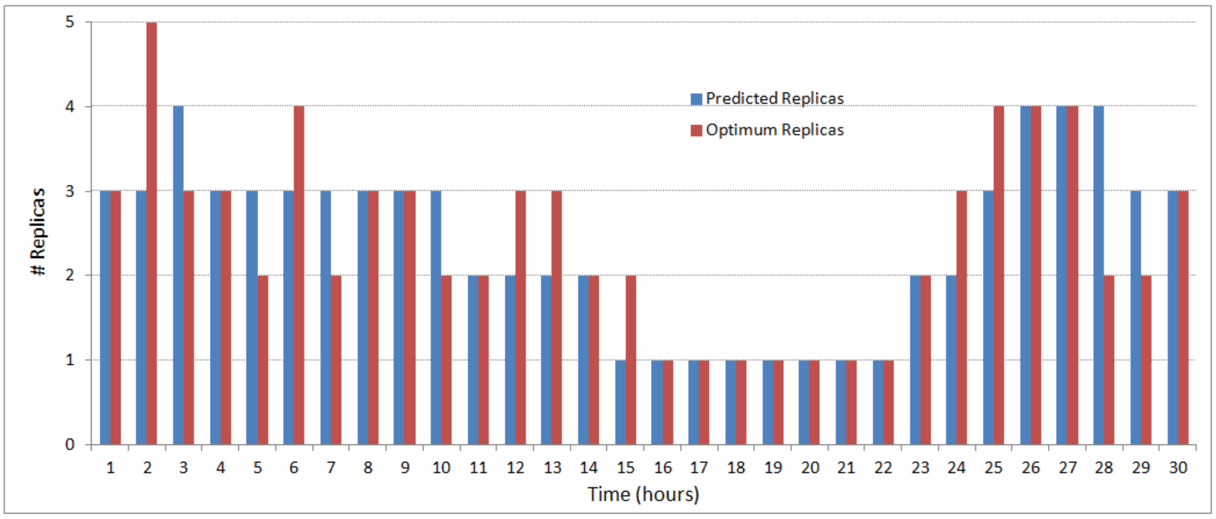


Figure 4.2: This example shows the application of the Decider and the capabilities of MYSE to follow the real number of replicas

4.2 Comparison between Reactive and Proactive MYSE

4.2.1 A Reactive MYSE Version

First implementation idea of MYSE was a reactive version, in which a decision of how many replica to pull up or down was taken real time at every request arrives. The NT-MYSE uses the Distribution Recognizer with a fixed length sliding window which contains the inter arrival time of each request, chose a distribution and its relative parameter (e.g. λ for the load) and took a decision about the number of replicas. The approach performed yet good enough comparing to simple heuristic based on a static number. The problem of choosing a static number was that we could choose a tradeoff between QoS and energy consumption or choose one to unbalance towards one of them, but it is impossible to adapt dynamically unto the optimum value. Having a few number of replicas we preserve energy consumption, but we aren't able to adapt to augmenting value of traffic. On the contrary if we set a big number of replicas we cannot sure an energy efficiently.

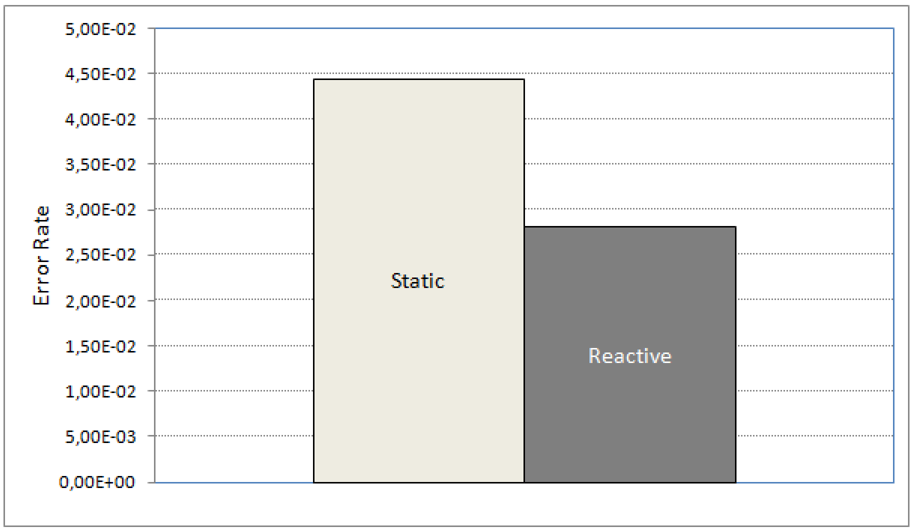


Figure 4.3: Comparison between the average error rate committed λ in a 24-hours test by a static model and by the reactive NT-MYSE version

The reference static model is a tradeoff between the 2 policies with s fixed to the average value of load. The NT-MYSE performs obviously better than a static system, but the problem is that the Distribution Recognizer is based on its window. It means that if we have a short window the choice accuracy of the correct distribution is not enough, but the system is able to adapt quickly to recognize different patterns, on the contrary a greater window could be more accurate, but not quick enough to adaptation at pattern changes.

4.2.2 The Importance of Being *Proactive*

The NT-MYSE seen before has the problem of the choice of one policies between accuracy or speed to convergence or a tradeoff. The idea to overcome such limitation was to make the system proactive, in order to make a decision about the number of replicas $n(t)$ for the instant t in a previous temporal time $t - d_i$, where d_i is one of the possible value of the horizon of prediction, so with $i = 1, 2 \cdots, d$.

A reactive MYSE has the strength that the real λ is exactly the same, but just in flat traffic condition, because during a traffic rate changes the performances decrease significantly. The ANN of proactive MYSE, on the contrary, showed a great capabilities of adaptation to totally different pattern while maintaining under a threshold the error of prediction and we want to use this strength to improve the performances of a reactive MYSE.

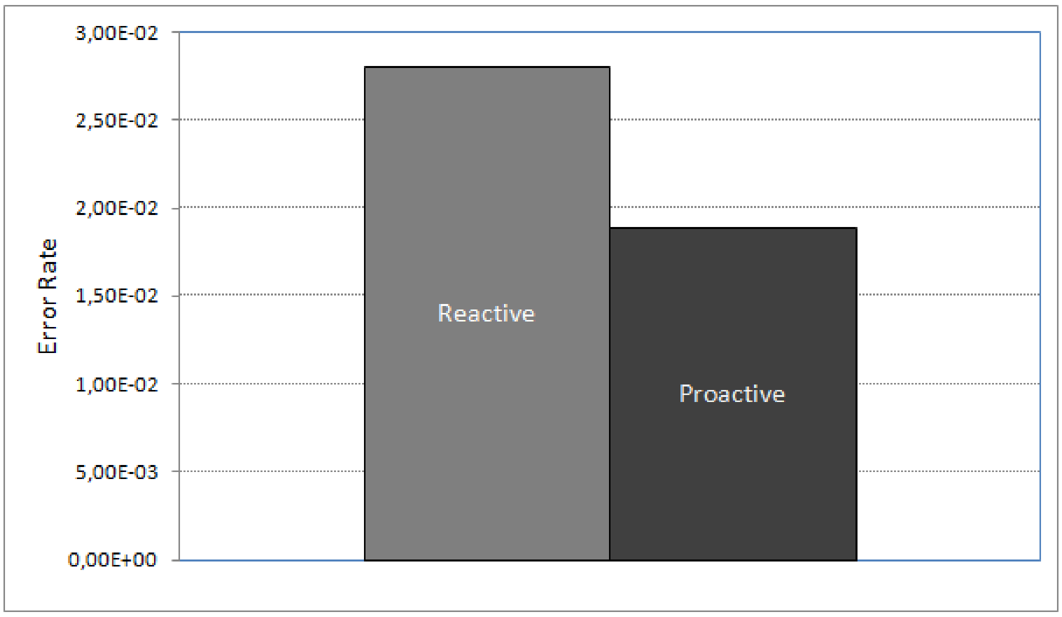


Figure 4.4: Comparison between the average error rate committed λ in a 24-hours test by the reactive NT-MYSE and by the proactive MYSE

ACCURACY TEST: NT-MYSE vs MYSE

- **Test Description:** This 24-hours test evaluate the error committed by NT-MYSE and MYSE by calculating λ and applies the same decision algorithm (shown in the previous section) with that λ to set a number of servers s. The test is aimed to verifies our hypothesis that the error of accuracy due to the introduction of forecasters is less significant that the error due to pattern changes of NT-MYSE for sudden changes of traffic load during a day.

- **Experimental Settings:**

  1. **Load Forecaster:** the ANN used is the best setting ANN3 trained with Back-propagation online with a sliding window of 100 hours, a learning rate and momentum fixed by following the best values found out previously and without considering possible overfitting.
  2. **Queueing Model:** M/M/S
  3. **Correctness Assumption:** the system is always in a stationary state

     $$\rho = \frac{\lambda}{s\mu} < 1$$

  4. **Fixed Parameters:** $QoS = 1.0$ and $\mu = 6.0$

Results: This evaluation is aimed to highlight the relevant added value of employing traffic predictions for correctly issuing resource provisioning. Figure 4.5 shows how the number of requested replicas varies over time on a hour-basis during a whole day. These results have been obtained by simulating three distinct scenarios:

1. the optimal configuration, that is the minimum number of replicas required to meet QoS requirements,

2. the configuration produced without the contributions of Δ-Load Forecaster Prediction Forecaster, referred to as *Non-Trained MYSE* (NT-MYSE) mentioned above and

3. the configuration requested by the complete MYSE module.

In the simulation of NT-MYSE, the Decider is fed with traffic details that are produced in real-time by the Distribution Recognizer on the basis of the current traffic only.

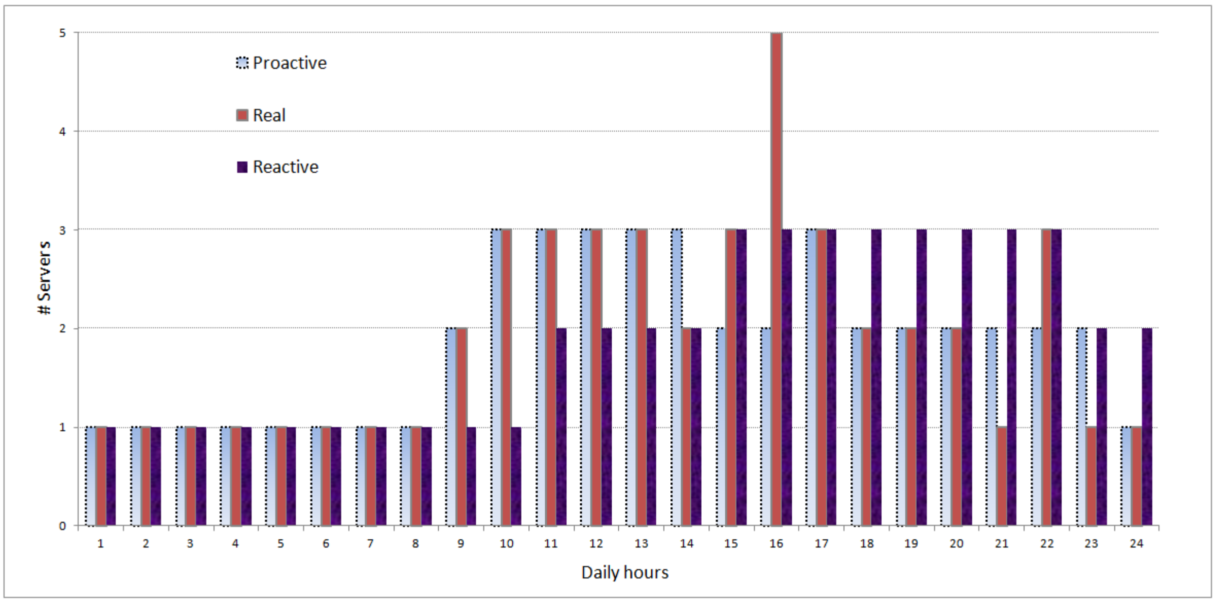


Figure 4.5: Comparison between the replicas set up in a 24-hours test by the reactive NT-MYSE and by the MYSE and the optimum number

Until 8:00 the traffic is stable and both the approaches behave correctly, then traffic begins changing and the use of predictions shows its effectiveness by contributing to generate configurations that are nearer to the optimum compared to NT-MYSE approach. Another important advantage of employing predictions is rendering the system more robust to unexpected peaks. This can be seen by observing the effect of the isolated peak occurring at 16:00 (a peak in the number of the optimal number of replicas corresponds to a peak in traffic load): NT-MYSE is biased by such occurrence and hereafter keeps to over-provision, while MYSE correctly recognizes it as an outlier.

We quantified numerically the error of each approach by averaging over the entire day the number of replicas above (over-provisioning) and below (QoS violation) the optimum. NT-MYSE provided on average 0.29 replicas in excess and 0.33 replicas less than the minimum required, for a total of 0.62. MYSE allows on average to over-provision 0.13 replicas (55% more accurate) and under-provision 0.21 replicas (36% more accurate), that is a total error of 0.34 replicas (45% more accurate).

By these series of tests we verified that a proactive MYSE has better performances than the reactive one. The figure 4.4 shows this result with the error rate committed with the reactive version NT-MYSE and the proactive MYSE.

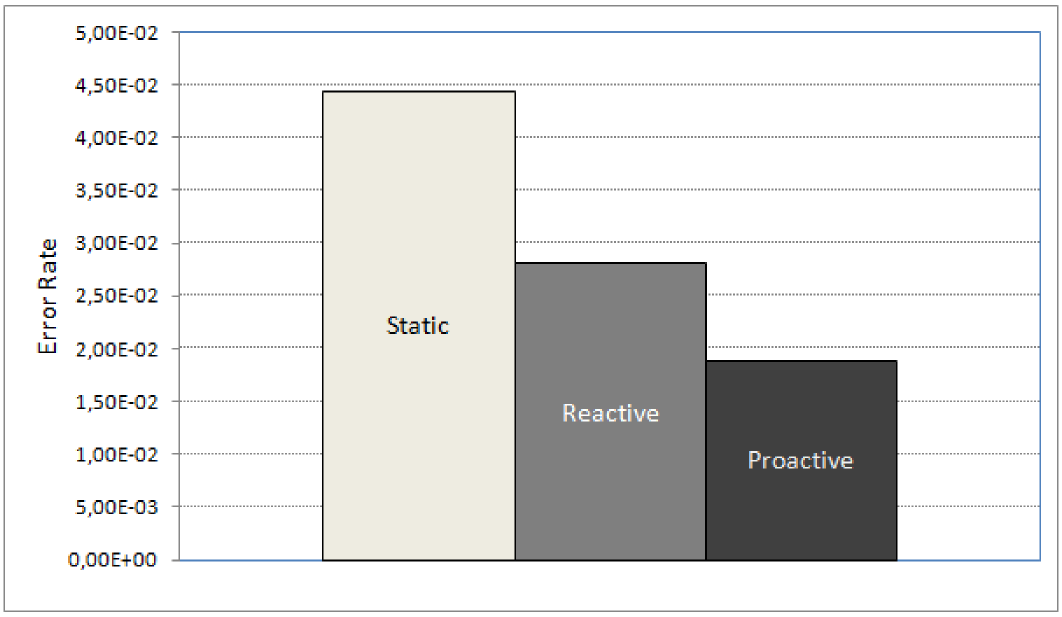


Figure 4.6: Comparison between the average error rate committed on λ in a 24-hours test by a static model, by a reactive NT-MYSE and by the proactive MYSE

This result is due to a good accuracy of predictions which give to the global system an error rate fewer than the error rate committed during a transaction between different pattern. From this result we understand the importance of having a forecasting for have a better accuracy in every temporal instant. Comparing also the first static error rate with that obtained with the proactive MYSE, we can note a really improving of the performances. The Figure 4.6 shows the final comparison where it is possible to see the improving obtained by switching from a static system which make an average provisioning of replicas to a reactive MYSE, finally to a proactive one.

Chapter 5

Conclusions

This chapter concludes summarizing the whole work and giving some hints about possible future directions of research and application of what has been described so far.

Providing smart scaling functionalities for elastic infrastructures so as to adapt to the evolution of the load is proving to be a must-have in the emerging scenarios of big data and cloud-based services. To automatic scaling a replicated service avoiding both performance worsening and over-provisioning is the goal of the architecture we propose in this work. In particular the aim is to meet QoS while resources saving by finding the minimum number of replicas which can ensure it. For example, a company with its own servers can save money with less energy consumption, others using services like Amazon Web Services [4] can save money with cheaper fares.

Modeling a replicated target service as a queue model belonging to the Queueing Theory, allows us to estimate the response time of each request and checking if the QoS that we want to ensure is guaranteed and if we have an excessive number of replicas in the current configuration.

The reactive version of MYSE Framework performed well in a standard traffic condition, but provided worse performances during traffic changes.

The goal of this work has been the building of a proactive MYSE so as to address such weakness, exploiting the fact that the error introduced by the forecasters is less significant than the error committed by the reactive MYSE during a input traffic changes.

Hence the core of this Framework are the Forecasters. We have built several ANNs for Δ-Load Forecaster and Δ-Distribution Forecaster in order to predict next values of load and distribution by providing to these modules Δ which indicates the horizon of prediction we want to predict. Several tests are made for each forecaster to find out the right ANN dimension, the best horizon of prediction and the best learning scenario (training set dimension, iteration of Backpropagation algorithm, real time sliding window). The best ANN configuration found is tested within different tests on the accuracy, robustness and efficiency and the result demonstrated great capabilities of prediction accuracy either in a standard traffic condition and in traffic changes situation.

Finally, tests on the error committed by the reactive MYSE and by the proactive MYSE on the λ value (traffic) and on the number of replicas with respect to the optimum are made for having a comparison between these two approaches. The results confirm our hypothesis that the proactivity would introduce an error less significant that the error committed by the reactive MYSE giving a better accuracy to the overall system.

According to the results obtained by the simulations we carried out, we believe that forecasting requests and service times, and carefully modeling how performance gets affected, are the proper building blocks for achieving our goal.

Future Directions

The main goal after the results obtained by these simulations is to implement the MYSE architecture on Amazon Web Service in order to concretely assess strengths and weaknesses of the model. Selected a target service for a prototype implementation, we will be able to provide both a validation of the MYSE module and the specification of the analytical model for the concrete service in order to compare the results. The MYSE deployment in a real cloud infrastructure will allow to estimate the delays of resource activation/deactivation, which will contribute to the accuracy of the model itself.

We are investigating how to make the MYSE architecture completely non-intrusive. i.e., such that MYSE does not need any status information, such as ”service times”, from the replicated service. This would enable and ease its employment in a wider range of applications.

Our idea is to use the same approaches employed in the Δ-Load and Distribution Forecaster to directly infer the expected latencies for a given request/reply pattern. This would also allow to replace the mechanisms currently used in the Decider module, that are based on queuing models, with others based on ANNs.

Another important direction regards the modeling of further types of systems, rather than only replicated services. Large scale storages (i.e. Cassandra, [5]) and complex event processing frameworks (i.e. Storm, [9]) don't fit well with the simple queuing model we employed so far, and devising techniques to accurately compute how the performance of these kinds of systems are influenced by input traffic seems considerably challenging.

Another interesting direction concerns the inclusion of locality awareness within the algorithms for choosing which resources to add/remove: distinct resources can have hugely different impacts on both performance and cost of a deployment.

Finally, it would be really interesting to obtain an even better prediction results by using more complex machine learning techniques e.g. trying to use Recurrent Network instead Feed-Forward and with learning based on Genetics Algorithms for finding the best ANN configuration.

"Never cut the bridges. . . "
J. P. Lombardi Jr.

Bibliography

[1] Amazon CloudWatch Service. `http://aws.amazon.com/cloudwatch/`.

[2] Amazon Load Balancer Service. `http://aws.amazon.com/elasticloadbalancing/`.

[3] Amazon Elastic Compute Cloud - EC2. `http://aws.amazon.com/ec2/`, 2006.

[4] Amazon Web Services. `http://aws.amazon.com/`, 2006.

[5] Cassandra. `http://cassandra.apache.org/`, 2008.

[6] Encog 3.0. `http://www.heatonresearch.com/encog/`, 2008.

[7] Google App Engine. `https://appengine.google.com/`, 2008.

[8] Microsoft Azure. `http://www.microsoft.com/azure/`, 2008.

[9] Storm. `http://storm-project.net/`, 2011.

[10] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2004.

[11] John Robert Anderson, Ryszard Spencer Michalski, Ryszard Stanisław Michalski, Thomas Michael Mitchell, et al. *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986.

[12] Oliver Duncan Anderson. *Time series analysis and forecasting: the Box-Jenkins approach*. Butterworths London, UK, 1976.

[13] I. Astrova, A. Koschel, and M. Schaaf. Automatic Scaling of Complex-Event Processing Applications in Eucalyptus. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 22–29, 2012.

[14] Andrew D Back and Ah Chung Tsoi. Fir and iir synapses, a new neural network architecture for time series modeling. *Neural Computation*, 3(3):375–385, 1991.

[15] Roberto Baldoni, Giorgia Lodi, Luca Montanari, Guido Mariotta, and Marco Rizzuto. Online black-box failure prediction for mission critical distributed systems. In *SAFECOMP*, pages 185–197, 2012.

[16] M. Beltran and A. Guzman. An Automatic Machine Scaling Solution for Cloud Systems. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, 2012.

[17] Christopher M Bishop. *Neural networks for pattern recognition.* Oxford university press, 1995.

[18] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[19] Sangeeta Biswas, Shamim Ahmad, M. Khademul Islam Molla, Keikichi Hirose, and Mohammed Nasser. Kolmogorov-smirnov test in text-dependent automatic speaker identification. *Engineering Letters*, 16(4):469–472, 2008.

[20] Mat Buckland. *A1 techniques for game programming.* CengageBrain. com, 2002.

[21] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.

[22] Chris Chatfield. *Time-series forecasting.* Chapman and Hall/CRC, 2002.

[23] S-T Chen, David C Yu, and Alireza R Moghaddamjo. Weather sensitive short-term load forecasting using nonfully connected artificial neural network. *Power Systems, IEEE Transactions on*, 7(3):1098–1105, 1992.

[24] Jerome T Connor, R Douglas Martin, and LE Atlas. Recurrent neural networks and robust time series prediction. *Neural Networks, IEEE Transactions on*, 5(2):240–254, 1994.

[25] S. Costache, N. Parlavantzas, C. Morin, and S. Kortas. Themis: Economy-based Automatic Resource Scaling for Cloud Systems. In *High Performance Computing and Communication IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE 14th International Conference on*, pages 367–374, 2012.

[26] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, page 24, 2001.

[27] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[28] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 125–134. ACM, 2012.

[29] Georg Dorffner. Neural networks for time series processing. In *Neural Network World.* Citeseer, 1996.

[30] S. Dutta, S. Gera, Akshat Verma, and B. Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *Cloud Computing (CLOUD), IEEE 5th International Conference on*, pages 221–228, 2012.

[31] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Architecting Dependable Systems. chapter Increasing System Dependability Through Architecture-based Self-repair, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003.

[32] Seymour Geisser. *Predictive interference: an introduction*, volume 55. CRC Press, 1993.

[33] Min Han, Jianhui Xi, Shiguo Xu, and Fu-Liang Yin. Prediction of chaotic time series based on the recurrent predictor neural network. *Signal Processing, IEEE Transactions on*, 52(12):3409–3416, 2004.

[34] DR Hush and JM Salas. Improving the learning rate of back-propagation with the gradient reuse algorithm. In *Neural Networks, 1988., IEEE International Conference on*, pages 441–447. IEEE, 1988.

[35] Christian Igel and Michael Hüsken. Empirical evaluation of the improved rprop learning algorithms. *Neurocomputing*, 50:105–123, 2003.

[36] Dušan Joksimović. Neural networks and geographic information systems. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[37] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[38] Özgür KISI and Erdal Uncuoglu. Comparison of three back-propagation training algorithms for two case studies. *Indian journal of engineering & materials sciences*, 12(5):434–442, 2005.

[39] Gerson Lachtermacher and J. David Fuller. Back Propagation in Time-series Forecasting. *Journal of Forecasting*, 14(4):381–393, 1995.

[40] KY Lee, YT Cha, and JH Park. Short-term load forecasting using an artificial neural network. *Power Systems, IEEE Transactions on*, 7(1):124–132, 1992.

[41] Seyed Saeed Madani. Electric load forecasting using an artificial neural network. *Middle-East Journal of Scientific Research*, 18(3):396–400, 2013.

[42] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48. IEEE, 2010.

[43] A. Maus and J.C. Sprott. Neural Network Method for Determining Embedding Dimension of a Time Series. *Commun. Nonlinear Sci. Numer. Simul.*, 16(8):3294–3302, 2011.

[44] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19:88, 1969.

[45] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[46] Frederick Mosteller. A k-sample slippage test for an extreme population. In *Selected Papers of Frederick Mosteller*, pages 101–109. Springer, 2006.

[47] In Jae Myung. Tutorial on Maximum Likelihood Estimation. *Journal of Mathematical Psychology*, 47(1):90–100, February 2003.

[48] Alan V Oppenheim, Ronald W Schafer, John R Buck, et al. *Discrete-time signal processing*, volume 5. Prentice Hall Upper Saddle River, 1999.

[49] Yohhan Pao. Adaptive pattern recognition and neural networks. 1989.

[50] Dong C Park, MA El-Sharkawi, RJ Marks, LE Atlas, MJ Damborg, et al. Electric load forecasting using an artificial neural network. *Power Systems, IEEE Transactions on*, 6(2):442–449, 1991.

[51] Lutz Prechelt. Early stoppingâbut when? In *Neural Networks: Tricks of the trade*, pages 53–67. Springer, 2012.

[52] Martin Riedmiller and Heinrich Braun. Rprop-a fast adaptive learning algorithm. In *Proc. of ISCIS VII), Universitat.* Citeseer, 1992.

[53] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[54] Martin Riedmiller and I Rprop. Rprop-description and implementation details. 1994.

[55] Brian D Ripley. *Pattern recognition and neural networks.* Cambridge university press, 2007.

[56] Luis Rodero-Merino, Luis M Vaquero, Victor Gil, Fermín Galán, Javier Fontán, Rubén S Montero, and Ignacio M Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8):1226–1240, 2010.

[57] Frank Rosenblatt. A model for experiential storage in neural networks. *Computer and information sciences. Washington, DC: Spartan*, 1964.

[58] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[59] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[60] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3), 2010.

[61] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. The mds queue: Analysing latency performance of codes and redundant requests. *CoRR, http://arxiv. org/abs/1211.5405*, 2012.

[62] Vladimir Stantchev and Christian Schröpfer. Negotiating and enforcing qos and slas in grid and cloud computing. In *Advances in Grid and Pervasive Computing*, pages 25–35. Springer, 2009.

[63] Zaiyong Tang, Chrys de Almeida, and Paul A Fishwick. Time series forecasting using neural networks vs. box-jenkins methodology. *Simulation*, 57(5):303–310, 1991.

[64] Zaiyong Tang and Paul A. Fishwick. Feed-forward neural nets as models for time series forecasting. *ORSA Journal of Computing*, 5:374–385, 1993.

[65] Zaiyong Tang and Paul A Fishwick. Feedforward neural nets as models for time series forecasting. *ORSA Journal on Computing*, 5(4):374–385, 1993.

[66] Arjen Van Ooyen and Bernard Nienhuis. Improving the convergence of the back-propagation algorithm. *Neural Networks*, 5(3):465–471, 1992.

[67] Thomas P Vogl, JK Mangis, AK Rigler, WT Zink, and DL Alkon. Accelerating the convergence of the back-propagation method. *Biological cybernetics*, 59(4-5):257–263, 1988.

[68] Michael K Weir. A method for self-determination of adaptive learning rates in back propagation. *Neural Networks*, 4(3):371–379, 1991.

[69] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Englewood Cliffs, 2001.

[70] Bernard WIDROW, Marcian E HOFF, et al. Adaptive switching circuits. 1960.

[71] Andrew W. Williams, Soila M. Pertet, and Priya Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *IPDPS*, pages 1–8, 2007.

[72] Francis S. Wong. Time series forecasting using backpropagation neural networks. *Neurocomputing*, 2(4):147–159, 1991.

[73] Zhen Xiao, Qi Chen, and Haipeng Luo. Automatic Scaling of Internet Applications for Cloud Computing Services. *IEEE Transactions on Computers*, 99(PrePrints):1, 2012.

[74] Alessandro Zelli. An Architecture for Automatic Scaling of a Replicated Service: Reactive Approach. Master's thesis, Sapienza University of Rome, Italy, 2013.

[75] Guoqiang Zhang, B. Eddy Patuwo, and Michael Y. Hu. Forecasting With Artificial Neural Networks: the State of the Art. *International Journal of Forecasting*, 14(1):35 – 62, 1998.

[76] Jia-Shu Zhang and Xian-Ci Xiao. Predicting chaotic time series using recurrent neural network. *Chinese Physics Letters*, 17(2):88, 2000.

List of Figures