UNIVERSITY OF PISA
SCUOLA SUPERIORE SANT’ANNA

Master Degree in Computer Science and Networking

Machine Learning for Automatic Configuration of Structured Parallel Applications

Candidate
Daniele Virgilio

Supervisors
Prof. Marco Danelutto
Prof. Alessio Micheli

Academic Year 2013/2014
# Contents

1 Introduction 1

2 Algorithmical skeleton 5
   2.1 Some skeletons examples ........................................ 7
   2.2 Skeleton Tree .................................................. 9
      2.2.1 Rewriting rules ............................................. 10
   2.3 Cost model and Performance measures .......................... 11

3 Neural Network for Trees 15
   3.1 Structured Domain .............................................. 15
      3.1.1 Recursive Transduction on Tree Domain .................... 16
   3.2 Tree Echo State Network ....................................... 20
      3.2.1 TreeESN architecture ...................................... 21
      3.2.2 Properties .................................................. 23
      3.2.3 Hyper-parameters .......................................... 24

4 Machine Learning for HPC Optimization Problems 27
   4.1 Auto-Tuning ..................................................... 28
   4.2 Predictive Models using Machine Learning ....................... 30
      4.2.1 Example of some possible applications ..................... 33
   4.3 Our vision ..................................................... 36

5 TreeESN Library 38
   5.1 System overview ............................................... 39
   5.2 Implementation details .......................................... 40
      5.2.1 Dataset files ............................................... 40
      5.2.2 Matrices mathematical operations .......................... 42
“Non so esattamente ciò che sto cercando, 
qualcosa che non è stato ancora suonato. Non so che cosa è. 
So che lo sentirò nel momento in cui me ne impossesserò, 
ma anche allora continuerò a cercare.”

Reported by Frank Kofsky on cover notes of 
“The John Coltrane Quartet Plays”
Chapter 1

Introduction

In 1989, a new structured method to exploit the parallelism in the computer architectures has been proposed in the high performance computing community. The new parallel programming model, called *Algoritmical Skeleton* [10], was designed to overcome the problems related to the programmability, portability and performance. Skeletal programming proposes to the user several parallel patterns (called skeletons) which occur regularly and that can be used by the programmer in a natural way. The skeletons hide the complexity of the underling parallel and distributed architectures. The decoupling layer of the skeleton framework allows the user to express the parallel application in terms of skeletons without caring of the details of the target parallel architecture. The orchestration and the synchronization of the parallel entities are dealt with transparently by the skeleton framework. The decomposition of the implementation in layers helps to reduce the errors typically arising when the low-level parallel programming models are used. Furthermore, the skeleton framework is able to face the efficient targeting of parallel hardware by providing an ad-hoc, optimized implementation of the available parallel patterns for each target architecture.

More complex parallel patterns can be built by combining the basic ones. The logical structure built by composing skeletons is a *skeleton tree*. The skeleton trees are the more abstract representation of parallel applications in a skeleton framework.

Knowing in advance the set of parallel patterns implemented in the framework, it is possible to provide a cost model estimating the completion time of the parallel application. Typically, the predictive model is an *analytical model*. The analytical model is defined formally on a system abstraction and it is compositional with respect to the parallel patterns used in the skeleton tree. The abstract analytical model needs to be
instantiated on the target architecture specifying some architecture-dependent characterizing parameters. The availability of a predictive model is very important in order to be able to adequately dimensioning the parallel computation to satisfy the required Quality of Service (QoS) properties.

However, with the constant improvement of the modern computers architectural features, both in terms of complexity and diversity, building a performance model becomes a more and more challenging task. Producing traditional analytic models may become a difficult and time consuming process, since it can be difficult to model/understand the relations among the causes and the effects. A long and error-prone process is needed to acquire the detailed characterizations of the target system and application. There could exist some cases in which the model built is no longer valid when next generation systems are taken into account. Moreover, analytic models require some important simplifying assumptions on the modelled domain that, sometime, may eventually concur to produce a poor final results.

A new paradigm to build predictive models can be explored by using *machine learning predictive models*. Using the machine learning techniques it is possible to analyse the application and its behaviour on the architecture in an “agnostic” way. The learning models are able to learn the behaviour exhibited by the parallel application on the target platform as a black-box. A training set is used to understand the target function behaviour. The training set is composed by a representative set of parallel applications (skeleton trees) along with their completion time. The machine learning model analysing the training set is then able to built a predictive model suitably approximating the target function.

In the high performance computing community, the machine learning approach has been already used in different contexts. However, the point of view with which it is applied is slightly different from the perspective exposed in this thesis. In literature, the machine learning models are typically used to predict time completion of an application where the structure of the parallel computation is unknown. For example in [33], the machine learning techniques are used to understand the behaviour exhibited, in terms of completion time, by some mathematical functions defined in an external library varying their input parameters. The completion time of the operation is tightly related to the architecture on which the function has been run. Other approaches that use machine learning are more close to our vision, analysing structured parallel programs using specific dataparallel computations. For instance, in [17] the completion time of
2/3D Jacobian stencils are taken into account. This experiment makes further progress allowing to generalize the type of computation treated, by exploiting the structure of the parallel computation.

The point of view taken when considering the predictive learning models in this thesis is quite similar to the last one mentioned. However, instead of using data-parallel computations, the structure of the application is defined by a skeleton tree annotated with relevant information. Similarly to the way in which the analytical model processes the skeleton tree, the predictive learning model should take into account the structure of the skeleton tree in order to predict the completion time of the parallel application. Since the input data is not a fixed list of features, the machine learning models must deal with the input structured domains. In particular, a learning model able to process tree domains is needed. Among the different possibilities in machine learning models able to deal with tree structured data, a model in the reservoir computing [25] area has been selected. The reservoir computing models are very suitable for a first analysis. Indeed, they are powerful and, at the same time, they allow a faster training with respect to the standard neural networks dealing with structured domains. In light of these considerations, the tree echo state network (TreeESN) model [16] has been selected for the experiment on the predictive learning model.

The thesis investigates how a predictive learning model, for the completion time of skeleton applications, can be built using the TreeESN. The predictive model obtained can be exploited as internal knowledge in an advanced skeleton framework. The framework, building on the knowledge results, is able to compile the parallel application using the best configuration (or an equivalent one) for the skeleton tree. If the skeleton framework supports autonomic computing, the built predictive model can also be used at run-time to reconfigure dynamically the application.

This thesis is organized as follows:

- The chapter “Algorithmical skeleton” (Chap. 2) describes in detail the algorithmical skeletons and it illustrates the analytic cost model for the evaluation of the structured parallel programs.
- The chapter “Neural Network for Trees” (Chap. 3) presents an overview of the machine learning for structured domain and it focuses on the TreeESN model used in the thesis.
- The chapter “Machine Learning for HPC Optimization Problems” (Chap. 4) exhibits a comparison between the models commonly used in the high performance
computing literature: *analytical* and *empirical optimization model*. In this chapter, it is analysed how the predictive learning models can take advantage from the previous methods and it describes a methodology for building predictive learning models. Finally, it presents how the predictive learning models can be used in a modern skeleton framework.

- In chapter “*TreeESN Library*” (Chap. 5), the library developed in this thesis is introduced which is able to deal with the tree structured domains using the *TreeESN* machine learning technique. The library is implemented in *C++* and the *BLAS/LAPACK* linear algebra libraries. On top of the library, two parallel implementations to deal with the validation problem have been built.

- In chapter “*Machine Learning Task*” (Chap. 6), the methodology described in the Chap. 6 is used to build a predictive learning model. Experiments are discussed that regard performance prediction in a *FastFlow skeleton programming environment*. The developed *Tesn library* is used to deal with the learning task.

- In chapter “*Parallel Implementation of the Model Selection*” (Chap. 7), two different parallel implementations of the model selection process have been implemented. Both the solutions are based on a streaming parallelism and they target the multicores architecture through the *FastFlow* framework.

- Finally, the chapter “*Results*” (Chap. 8) presents the results achieved concerning the two parallel implementations of the model selection process and the results obtained using the *TreeESNs* to build a prediction model.
Chapter 2

Algorithmical skeleton

The concept of algorithmical skeleton [9] has been introduced the first time by Murray Cole in the 1988. He synthetically introduces it as follows:

The new system presents the user with a selection of independent algorithmic skeleton, each of which describes the structure of a particular style of algorithm, in the way in which “higher order function” represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

The crux idea basing on building of this new programming model relies in:

- Typically, in general programming language, a given problem is decomposed to use well known solutions such as for instance divide et impera, dynamic programming, simulated annealing. At the same time, a parallel application programmer should use some well known parallel pattern to deal with the problem in an efficient way.

- The higher order functions, typical of functional languages, are the core of skeleton expressiveness. The skeletons are able to take as input parameters user defined code (the business logic), and if the skeleton related constraints are respected the legacy code can be executed in a parallel way preserving the user code semantics.

The logical separation between the business logic and the used skeleton implementation brings a lot of advantages. The application programmers have to specify the task(s) need to be solved using the proper skeleton, being unaware of the skeleton internal implementation details. The skeleton implementation details are dealt efficiently by the
system programmers, which provide state of the art skeleton implementation, applying all renown optimizations for the different supported computer architectures.

Later, in [10], Cole emphasized his theory pointing out to the four main features that a skeleton model should satisfies:

- **“Propagate the concept with minimal conceptual disruption”** – The simplicity should be one of the main key points. The skeleton model should be easy to learn and use, without altering the user view in terms of the programming language used to exploit the parallelism. It should rather provide a bridge between the user perspective and the de-facto standards, refining or constraining it only where strictly necessary.

- **“Show the pay-back”** – It should be shown that spending a moderated amount of time in learning the skeleton framework there are substantial benefits. They can be provided in terms of reached performance, simplicity of use and development time spent. It must be demonstrated the easiness in migration of the application from an architecture to another without or with a minimal effort, but, in any case, with sustained performance.

- **“Integrate ad-hoc parallelism”** – It is obvious that many parallel computations are not expressible in a skeleton framework. At the same way, it is unrealistic that the system programmers are able to provide all the possible parallel patterns. Thus, the framework must be able to be integrable (composable) in such a way controlled extensions are supported.

- **“Accommodate diversity”** – The skeleton should be customizable basing on the programmer necessity, in such a way slightly different behaviours (with respect to the initially foresight one) are offered. In any case, there should be a balance between the simplicity desires and a concrete need for simplicity.

A skeleton system can be seen as the implementation of a structured parallel programming system. For each skeleton it is possible to have many different implementations, since every skeleton exhibits a well-defined functional and parallel semantics [4]. Each of them may differ in the accepted parametric values, different platform-dependant optimization and parallel exploitation. Obviously, all these implementations should be hidden to the programmer to preserve the logical separation of concerns. At the same
time, the system should select appropriately the best implementation among the available ones in the specific context where the programmer want to use it.

Finally, the separation between user and system programmer enforces a further dichotomy between functional and non-functional constraints. The functional constraints define the requirements coming from the user business code and used parallel skeleton, while the non-functional constraints deal with how the computation is carried on preserving the user specified semantics. All the non functional aspects can be handled in an automatic way using autonomic computing [22] techniques. The most typical non-functional aspects handled in the skeleton context are:

- **Performance constraints** – Constraints related to the application performance metrics, such as throughput and latency, may exist. In order to fit this performance metrics a modification of the skeleton may be necessary both in the parallel degree and in the skeleton structure (Sect. 2.2.1).

- **Fault tolerance** – The application should survive and be resilient to network/nodes failures in such a way even if a failure is detected the semantics of the computation is preserved and the desired result is obtained in the end.

- **Power Consumption** – The power management assumes even more a key rule in the parallel computing. So, in the case the system detects that the gain in performance is not adequate with the current application configuration, it can adopt some changes that reduce the power consumption.

- **Security** – Security problems can be treated in such a way the system is preserved from intrusions and the right level of protection to the data and to the programs crossing the network is provided.

### 2.1 Some skeletons examples

A typical set of skeletons includes *pipeline, farm, map, scan* and *reduction* skeletons. In general, it can be roughly divided in stream and data oriented parallel forms. The *stream oriented computation* is often referred with the term *task parallelism*. In this kind of computations, it is assumed to have a source of data to be analysed as input, known as *stream*. A stream is a flow of data, all having the same type, that arrive in the system one after the other at unpredictable instant in the time. All the tasks belonging to the stream are independent one of the other and therefore they can be computed in
parallel without no dependencies. In the *data-parallel oriented computations* the focus is put on the parallelism that the data itself can explicit. Usually, in this kind of computations the data structure is divided in independent partitions (*map computations*) computed individually and then recomposed to obtain the final result. The computation on the single partition can have also some dependency with other positions. This class of computations is referred to as *stencil computations* and are classified depending on how the data dependency is mapped and how the computation consequently evolves. In the following only the stream oriented skeleton (i.e., farm and pipeline) will be discussed for sake of brevity.

**Pipeline**

The simplest stream parallel pattern is the pipeline. The *pipeline* recalls the “assembly line” work behaviour. It is composed by several stages arranged in a linear fashion. Every stage works on the incoming data, usually, coming from the previous stage and passes the computed data to the following stage. The pipeline parallel pattern decouples a composite function assigning one elementary function (or a composition of them) to each stage. Fig. 2.1 shows how the composite function $\mathcal{H} = \mathcal{F} \odot \mathcal{G} \odot \mathcal{L} \odot \mathcal{K}$ can be used to make a pipeline with 4 stages.

![Figure 2.1: The Pipeline skeleton.](image)

**Farm**

Another common stream parallel pattern is the farm. The *farm* paradigm is based on the functional replication of a computation on multiple workers in such a way they can compute multiple elements at the same time. Usually, the farm pattern is implemented using three module types:

- Emitter – It receives the input and handles the scheduling of the available tasks among the workers basing on a predefined scheduling policy. Common scheduling policies include the *round-robin* and the *on-demand* one. The latter is very useful in the case of high service time variability to obtain a better load balancing between workers.
• Collector – It performs the non-deterministic collection of the computed task from
the workers and it delivers computed tasks to farm output.

• Worker – The workers execute the replicated function on each individual task re-
ceived from the emitter and they send the computed result to the collector.

Multiple implementations of a farm exist depending on which type of node is used.
The “standard” version, illustrated in Fig. 2.1, uses all the node types we mentioned. But
other implementations exist. One of the most common and used farm implementations
is the master worker one, where the emitter and collector functionality are joined in a
single master entity. Another remarkable implementation, possible only in the shared
memory architecture, has no emitter and collector are used and the workers must fulfil
the emitter and collector functionality operating on in memory shared data structures.

![Figure 2.2: The “standard” farm skeleton implementation.](image)

### 2.2 Skeleton Tree

The skeletons are also defined as high order functions, supporting their hierarchical
composition. Since different skeletons exhibit different composability degree, a tree
A skeleton tree can be further composed hierarchically using the two-tier model [24, 11]. The two-tier skeleton model is mainly based on stream parallelism where the computation on a single task may be further decomposed in a data-parallel fashion. In this way, as shown in Fig. 2.2, the skeleton tree can be stratified in three layers: the upper layer is composed by stream parallel skeletons, the middle layer by data-parallel skeletons and finally the lower level is composed by sequential nodes.

2.2.1 Rewriting rules

As stated in [4], all the skeletons have a well defined semantics. Basing on the skeleton semantics, some classes of equivalence between skeletons can be identified. The rewriting rules [3] define skeleton transformations that are semantically equivalent. Thus, a skeleton belonging to a skeleton tree can be replaced by an equivalent one without affecting the final result of the computation. Obviously, only the semantic equivalence
is granted and so the equivalent skeletons may exhibit a different performance. Some of the most common skeleton equivalences are:

- **Skeleton Introduction/Elimination** – The equivalences in this group define the way in which the skeletons can be introduced or removed. Considering the farm and the pipeline skeletons, the equivalences for their insertion/removal are:

  \[ \Delta \equiv \text{Farm}(\Delta) \]

  \[ \text{Comp}(\Delta_1, \Delta_2) \equiv \text{Pipe}(\Delta_1, \Delta_2) \]

  where \(\Delta\) represents a general skeleton and \(\text{Comp}\) is the function that executes sequentially its skeleton arguments. The rewriting rules for the skeleton introduction are the functions that have as input the left side of the equivalence and as output the right side. The rewriting rules for the skeleton elimination is obtained in the opposite way.

- **Skeleton Associativity** – The equivalences in this group define the associativity of a skeleton with respect to another one. For instance, the pipeline skeleton is linear with respect to itself:

  \[ \text{pipe}(\Delta_1, \text{pipe}(\Delta_2, \Delta_3)) \equiv \text{pipe}(\text{pipe}(\Delta_1, \Delta_2), \Delta_3) \]

2.3 Cost model and Performance measures

The analytic cost model is a powerful tool that allows to estimate in a formal way the performance that the skeleton system will exhibit at run-time. A complete cost model analysis valid for generic structured parallel program can be found in [36].

In the case the computation is based on stream parallelism, the well known queueing theory model [23] can be used. This theory is based on firm background of the probabilistic and stochastic process theory. Using this model, under the proper conditions, it is possible to size (e.g. to derive proper parallelism degree) opportunely a system in such a way to obtain a predicted performance metrics values. An exhaustive dissertation of this topic is beyond the chapter goals and only an hint will be provided in order to understand the basic principles of this cost model analysis.

In describing a parallel module different metrics are very important:
- **Parallel degree** \[ n \] – describes how many entities participate to the specific parallel pattern used. In the following the \( n \) parameter will be used as apex to every performance metrics in order to underline that the specific metrics is referred to a parallel system having \( n \) as parallel degree\(^1\).

- **Latency** \[ L^{(n)} \] – It indicates the mean time required for a task to be completely processed.

- **Service time** \[ T_s^{(n)} \] – It indicates the mean time required by the system to accept two consecutive tasks. If the analysed system is sequential, the service time and latency coincide, otherwise, in general, \( T_s^{(n)} \leq L^{(n)} \) holds.

Some other important metrics are useful to characterize a parallel system:

- The **Ideal service time** \[ T_{s-id}^{(n)} \] represents the best service time obtainable from a parallel computation composed by \( n \) parallel module. It can be computed by

\[
T_{s-id}^{(n)} = \frac{T_s^{(1)}}{n} \tag{2.1}
\]

- The **scalability** \[ s^{(n)} \] represents how the system has been “speed up” with respect to the same parallel module with parallel degree equal to 1. It is defined as

\[
s^{(n)} = \frac{T_s^{(1)}}{T^{(n)}} \tag{2.2}
\]

- The **speed-up** \[ speedup^{(n)} \] measures related to how much the computation is faster with respect to the best implementation of the sequential module. It can be computed as

\[
speedup^{(n)} = \frac{T_{seq}}{T^{(n)}} \tag{2.3}
\]

- The **efficiency** \[ \epsilon^{(n)} \] represents how efficiently a parallel system uses the available resources. It is defined as:

\[
\epsilon^{(n)} = \frac{T_{s-id}}{T^{(n)}} \tag{2.4}
\]

\(^1\)If the parallel degree is equal to one the metric value refers to the a parallel system 1-dimensioned and it does not refer to the correspondent sequential system.
The guideline to maximize the performance in a parallel streaming system is reducing the service time. Indeed, if the input stream is sufficiently long, the completion time of a streaming program can be estimated by:

\[ T_c = mT_s \quad if \quad m >> n \]  

(2.5)

Other significant quantities in the evaluation of a system are the utilization factor \((\rho)\) and the inter-departure time \((T_p)\). The system utilization factor represents the congestion degree of the system under analysis and for a given module it can be estimated as \(T_s/T_a\), where \(T_a\) is the mean module inter-arrival time. If this quantity assumes values greater than 1, it means that the computation is a bottleneck since the system is unable to serve all the incoming tasks in time, and, at steady state, the system waiting queue tends to be saturated. The inter-departure time, the mean time between the deliver of two fully elaborated tasks, depends on the \(\rho\) value: if the system is a bottleneck than it can produce, in average, a served task every \(T_s\) period, otherwise every inter-arrival time.

A way to estimate the communication cost \((T_{comm})\) must be provided\(^2\). The communication cost comprises both the time spent in the send and the receive procedure. If the communication is implemented in a zero-copy way, the total communication cost is at the expense of the send mechanism \((T_{comm} = T_{send})\). Generally, the send communication cost is modelled using a linear function:

\[ T_{send} = T_{setup} + MT_{transm} \]  

(2.6)

where \(T_{setup}\) is the time spent in the initialization of the communication, \(T_{transm}\) is the cost for transmitting a data element (commonly a word) and \(M\) is the amount of this elementary data sent.

**Parallel patterns analysis**

All the recalled measures are useful to formally study some well known structured parallel patterns. The users, using on the analytical model, can figure out the performance measures achieved by the specific parallel pattern implementation before actually implements the parallel application. In the following only the cost model relatives to the pipeline and the farm parallel pattern will be analysed.

\(^2\)A communication cost model must be provided independently if a local or global environment is used.
For a pipeline, supposing that it is composed by a pure functional composition without sharing of data between stages, the analytic model can be applied straightforwardly. The pipeline service time is

\[ T_{s-pipe} = \max \left( T_a, \max_{stage_i} T_{s-stage_i} \right) \]  

(2.7)

As it can be seen from the above equation the service time depends on the most computational costly stage. So, this parallel pattern can suffer of load unbalancing problem that can degrade the performance. However, if the stages are all balanced, the ideal performance is achieved. To express all potential parallelism this pattern requires a transition time: all the stages must have a task to compute at the same time. So, the completion time is given by a term proportional to the pipeline service time plus a filling transient time.

\[ T_c = \sum_{stage_i} T_{s-stage_i} + (m - 1) \times T_{s-pipe} \]  

(2.8)

Also for a farm that does not share a common state between the replicated functions, the analytic model can be applied straightforwardly. Supposing that the farm is implemented in the “standard way”, with an emitter and a collector node, it can be seen as a pipeline where the first stage distributes the tasks to the workers, the second one computes the function in a parallel way and the last one collects the computed results.

\[ T_{s-farm} = \max \left( T_{emitter}, \frac{T_{cal}}{N_w}, T_{collector} \right) \]  

(2.9)

where \( N_w \) is the number of the workers inside the farm. If the emitter and the collector are not bottleneck, the service time is equal the the ideal one.

\[ T_{s-id} = \frac{T_{cal}}{N_w} \]  

(2.10)

Unlike the standard implementation, if the farm is implemented as a master-worker it is modelled through a question-answer client-serving.
Chapter 3
Neural Network for Trees

3.1 Structured Domain

Typically, neural networks deal with flat input domain. In the flat data representation, all the samples are represented by a fixed number of attributes (called features). But, there are cases in which a different data representation is more convenient since there are some relationships between individual data components that need to be explicated. Different classes of data type can be used, such as: list, free tree, directed ordered acyclic graph (DOAG), directed positional acyclic graph (DPAG) and graph. The structured data domains allow to specify different information types that can be used to make more effective the learning procedure [27]. The typical information regards the information associated to the vertexes (called label) and to the edges. But, it is also possible to express local/global information and target function values at different levels in the data topology.

The first attempt in trying to deal with structured data is based on the transduction [13] functions. They provide a way to map the input data structure to the desired output value. Often, these functions have to face with the variable-size structure. In these data structures, there is not a fixed structure and all the topological information may change from input to input. The variability of the input data structure makes difficult to build an efficacious transduction functions for a specific task. Two main approaches are identified to approach the problem:

- In the static transductions, an a priori strategy to map the data structure is defined. This method compromises the information carried out by the relationship between vertexes, since the data structure has implicitly same variability in its size and in the existing connections between vertexes. In principle, the encoding of a complex
object in an object of lower complexity can preserve all the informations contained at the cost of increasing the representation size, as it happens for the nested parenthesis tree representation.

- The adaptive transductions, instead, are able to catch the relationship between vertexes and extract the relevant information needed. An automatic learning rule exists to capture the relevant information used as internal knowledge representation. The outcome of the transaction learning rule depends on the data provided and on the specific task need to be solved.

3.1.1 Recursive Transduction on Tree Domain

The main idea for the machine learning in structured domain is to build a class of hypothesis, where the solution must be found, that maps the structured data domain $\mathcal{G}$ in an output space $\mathcal{O}$. In the following, the exposition will be restricted to k-ary (rooted) tree transduction. However, the treatise can be extended, also, to the rooted DPAG transductions.

Some terminologies over tree components is necessary to efficaciously explain what a transduction is. A k-ary tree is represented by $t$ and the set of its nodes by $N(t)$. The tree root is denoted by $\text{root}(t)$, while a generic node is indicated by $n$. The $i^{th}$ child of the node is identified by $\text{ch}_i(n)$. The letter $k$ refers to the maximum arity of a tree node. The sub-tree originated by the node $n$ is referenced as $t(n)$. Finally, the label of the node $n$ is indicated by $u(n)$.

The Structured Data Recursive Processing System is a couple $< \mathcal{G}, \mathcal{T} >$ where $\mathcal{G}$ is the domain of the labelled k-ary tree and $\mathcal{T}$ is a transduction function having type

$$\mathcal{T} : \mathcal{G} \rightarrow \mathcal{O}$$  \hspace{1cm} (3.1)

Depending on how the output domain is characterized, the function $\mathcal{T}$ can generate two kinds of transductions: tree to tree and tree to element transduction. In the tree to tree transduction the function $\mathcal{T}$ is isomorphic to its input: the tree topology is preserved and an output is provided for each node. In the tree to element transduction, an unstructured vector is provided as output. Fig. 3.1 shows respectively the tree to tree and tree to element transduction.
For the tree to tree transduction, the transduction function is a composition of the encoding and the output function:

$$\mathcal{T} = \tau_{\text{out}} \circ \tau_{\text{enc}}$$  \hspace{1cm} (3.2)

The goal of the encoding function is to map the sub-tree originated by a node to a point in a multidimensional encoding space in such a way all the information about the tree labels and the tree structure is retained. The output function maps the tree internal state in the desired output value.

Instead, for the tree to element transduction a way to reduce the output domain (from structured to unstructured) is needed. Since the output domain is no more isomorphic to the input, a function mapping the tree internal representation in a single state is needed. This function is called *state mapping* \((\chi)\). Thus, the transduction function is rewritten as

$$\mathcal{T} = \tau_{\text{out}} \circ \chi \circ \tau_{\text{enc}}$$  \hspace{1cm} (3.3)

The encoding function \(\tau_{\text{enc}}\) is defined as:

$$\tau_{\text{enc}}(n) = \hat{\tau}(t(n), x_{\text{nil}})$$  \hspace{1cm} (3.4)

$$\hat{\tau}(t, x_{\text{nil}}) = \begin{cases} \tau(u(n), \hat{\tau}(t(ch_1(n)), x_{\text{nil}}), \ldots, \hat{\tau}(t(ch_k(n)), x_{\text{nil}})) & \text{if } t \neq n(ch_1(n), \ldots, ch_k(n)) \\ x_{\text{nil}} & \text{otherwise} \end{cases}$$  \hspace{1cm} (3.5)
The tree state, represented as $x(t)$, is the output of the encoding function. It is computed via a recursive function: the $\hat{\tau}$ function encodes a single tree node, while the $\tau$ merges several child states with the current node label. The encoding function produces a state for each node in the tree. In the tree to tree transduction, all these states are assembled to make the same input topology, while in the tree to element transduction they are reduced to a single one. The $\tau$ function can be illustrated as shown in Fig. 3.2, where the $q_i^{-1}$ operator [13] represents the $i^{th}$ child internal state of the analysed tree node. The Encoded Network represents an unrolled version of the $\hat{\tau}$ recursive function. Applying the function in a recursive way to a specific input tree, a multi layer neural network is built basing on the tree topology. An example of mapping between an input tree and the correspondent neural network is shown in Fig. 3.3.
The computation of tree state evolves propagating from the leaves to the root node. The recursive encoding function associates the same state for the same sub-tree, even if it is placed in different trees. But, soon as parent node of the substructure is analysed, the computed state of the two trees immediately diverges as can be seen from Fig. 3.4.

\[ \text{State Space } \mathbb{R}^{N_r} \]

Figure 3.4: Illustration of what happens when two trees sharing the same prefix are encoded by the recursive transduction function.

The recursive transduction method makes two important assumptions: the stationary and the causality assumption. These reduce the hypothesis space by imposing important language biases. The stationary assumption states that the node-wise encoding function \((\tau)\) and the recursive encoding function \((\hat{\tau})\) are same independently from the position in the tree where it is applied. The universal approximation property of the recursive neural network [32, 18] assumes the stationary concept. In principle, it does not reduce the expressive power of the model but, at the same time, it strongly reduces the complexity of the model (i.e. it reduces a lot the hypothesis space where the solution is found). The causality assumption imposes that the state of a node depends only on the already analysed nodes. In particular, it depends only on the current node and on the nodes of the relative subtree. This assumption is a necessary and sufficient condition to admit a recursive state transduction. In some applications this assumption is not satisfactory, so contextual models [28] are created to take into account also the context in which the node is placed.
3.2 Tree Echo State Network

In the recurrent/recursive neural network models a class of models based on a separation between the network recurrent dynamic part (called Reservoir) and a simple non recurrent output tool (called Readout) have been defined. The name of this modelling paradigm, derived from the name of the dynamic recurrent part, is Reservoir Computing [25, 37]. The reservoir entity takes care of analysing and keeping track of the graph information/structure using causality and stationary assumption. The readout only relies on the information provided by the dynamic part. The key idea in reservoir computing is that after the initialization of the reservoir component according to some criteria and then it is left untrained. In this way, it is possible to rely on a semi-adaptive transduction: in which there is a fixed encoding function and a trained output function. Interestingly, even if the reservoir neural network is not trained, under specific conditions, the reservoir computing models are able to discriminate the data structure information presented as input (e.g. it can discriminate the input history when sequence of data elements are analysed). There are various kinds of models using this methodology. One of them is the Echo State Network (ESN) [20, 21, 19] but others models exist such as the Liquid State Machine [26], Back Propagation Decorrelation [34] and Evolino [31].

The Echo State Network name is inspired from the way in which the reservoir component behaves. Indeed, it produces an high dimensional dynamical “echo” response that is combined with the input to reconstruct the desired output. The echo signal is composed by non orthogonal basis component that are produced using the reservoir as a random non-linear excitable medium. The ESN is a neural network able to handle both sequence to sequence and sequence to element transduction adapting to the task needed. In its base form, it deals with time series element, but it has been extended also for dealing with k-ary trees using the Tree Echo State Network [16, 14].

In the following, only the Tree Echo State Network (TreeESN) will be described. However, the concepts described in the following also apply for the standard ESN. Since the TreeESN is a generalization of the standard Echo State Network, all the properties will be valid setting the arity degree of the analysed k-ary tree to one.
3.2.1 TreeESN architecture

The Tree Echo State Network is composed by three distinct parts: the input neurons, the reservoir and the readout. The input layer presents the tree node labels to the reservoir component for the state encoding. It is composed by $N_u$ input units, one for each feature available in the node label. The reservoir component consists of $N_r$ neural units randomly connected to each other in a recurrent fashion, in such a way that also connections loops among neurons are allowed. Usually, the reservoir component is highly dimensional and sparsely connected neural network. The reservoir is connected to the readout component for the generation of the correct output value. Usually, $N_y$ different linear non recurrent units provide the predicted values for each target associated to the input samples. A sketch of the network is depicted in Fig. 3.5.

\[
\hat{W}u(n) = y(t)x(t)
\]

Figure 3.5: The general architecture of an Tree Echo State Network.

**Reservoir**

For a TreeESN, the reservoir component computes the state of each vertex as follow:

\[
x(n) = f \left( W_{in}u(n) + \sum_{i=1}^{k} \hat{W}x(ch_i(n)) \right)
\]

where $x(n)$ represents the state associated to the node $n$, $W_{in} \in \mathbb{R}^{N_r \times N_u}$ is the input to reservoir connection matrix, $\hat{W}_k$ is the $\mathbb{R}^{N_r \times N_r}$ reservoir connection matrix. The reservoir can be customised in order to handle the non positional trees simply replacing the $\hat{W}$ with $k$ different reservoir connection matrices (one for each child positions) in the above equation. In correspondence to an absent node the null state value ($x_{nil} = 0$)
is used. And finally, $f$ is the activation function of each reservoir units. Usually, the hyperbolic tangent ($tanh$) activation function is used since it provides desirable properties (see Sect. 3.2.2).

State Mapping

The encoding mechanism process changes accordingly to the tree data structure analysed. The process produces, for each node, a state composed by $N_r$ components coming from each reservoir neuron. Since the tree to element transduction is taken into account, a method to map the $|N(t)|$ states in a single one is necessary. There are different choices in the selection of the state mapping function. The most commonly used ones are the Root state mapping and the Mean state mapping technique [16]. The root state mapping function

$$\chi(x(t)) = x(root(t)) \quad (3.7)$$

only utilizes the state of the root node as representative element of the whole data structure.

The mean state mapping function

$$\chi(x(t)) = \frac{1}{|N(t)|} \sum_{n \in N(t)} x(n) \quad (3.8)$$

aggregates all the state information coming from all the tree nodes and averages them, instead.

Readout

In the standard TreeESN, the readout component is a simple linear regression tool. The output function is defined differently according to the utilised transduction. If the tree-to-tree transduction is used, the readout component is applied to every node of the input tree:

$$y(n) = W_{out}x(n) \quad (3.9)$$

where $W_{out} \in \mathbb{R}^{N_y \times N_R}$. Otherwise, in the case the tree-to-element is adopted the readout is applied only to the state obtained from the state mapping operation:

$$y(t) = W_{out}\chi(x(t)) \quad (3.10)$$
The readout is trained according to the computed states and the target values. Usually, the readout components are trained using the *least mean square metric*. So the training problem can be restated as:

\[
\min \| X W_{\text{out}} - Y_{\text{target}} \|^2_2
\]

(3.11)

where \( X \in \mathbb{R}^{P \times N_r} \) is the state matrix related to \( P \) training samples and \( Y_{\text{target}} \in \mathbb{R}^{P \times N_Y} \) is the target matrix.

Two specific techniques can be used to solve the least mean square problem: via *Moore-Penrose pseudoinverse* [30] and via *ridge regression* [38]. In the former case, the pseudo inverse of \( X \) denoted as \( X^+ \) is computed to obtain the \( W_{\text{out}} \) weight values.

\[
W_{\text{out}} = X^+ Y_{\text{target}}
\]

(3.12)

In the latter, the \( W_{\text{out}} \) weight values are computed using the following linear algebra equation:

\[
W_{\text{out}} = (X^T X + \lambda_r I_{N_r})^{-1} X^T Y_{\text{target}}
\]

(3.13)

where \( I_{N_r} \) is \( N_r \times N_r \) identity matrix while \( \lambda_r \) is a regularization factor which determines the magnitude of the readout weights. The inverse always exists for any \( \lambda_r \) values greater than 0.

### 3.2.2 Properties

The state space of the tree domain in the echo state network assumes a Markovian nature. A recurrent/recursive neural network that organizes its state space in such a way is able to discriminate among different input history in a suffix-based Markovian flavour, even prior to learning. The contractivity condition of the reservoir network guarantees a Markovian state space organization. In the following a concise but formal enunciation of the properties and theorems will be exposed.

The *Echo State Property* states that the final state in which a node is mapped to only depends on the input data structure itself. The dependence on the initial state of the network is progressively lost as the data structure length goes to infinity. Two main conditions have been enlighten on the echo state property when the hyperbolic activation function is used for the reservoir neurons:
• The matrix spectral radius ($\rho$) less than 1 is a necessary condition for the echo state property. If this condition does not hold, the dynamic of reservoir is locally asymptotically unstable (at 0 state) and echo state cannot be guaranteed.

• A sufficient condition for the echo state property is that the largest singular value ($\sigma$) is less than 1. This condition can be expressed with $\|\hat{W}\|_2 < 1$. It ensures global stability and thus the presence of the echo state.

The Contractivity property is a crux in the echo state dynamics analysis. Indeed, if the node-wise encoding function ($\hat{\tau}$) is contractive then the echo state property holds. The function $\hat{\tau}$ is a contraction with respect to the state space if there exists a non-negative parameter $C < 1$ such that $\forall u \in \mathbb{R}^N$ and $\forall [x_1, ..., x_k], [x'_1, ..., x'_k] \in (\mathbb{R}^N)^k$ it holds that:

$$\|\tau(u, [x_1, ..., x_k]) - \tau(u, [x'_1, ..., x'_k])\| \leq C \max_{i=1}^k \|x_i - x'_i\|$$

(3.14)

Whenever the state transition function of a TreeESN is contractive and the network state space is bounded, the nature of the reservoir dynamics is characterized by Markovianity [35, 15]. For every height $h > 0$, any two input trees $t, t'$ sharing the same suffix of height $h$, i.e. $S_h(t) = S_h(t')$, and any states $x, x' \in N_R$ such that $\|x - x'\| \leq \text{diam}$, the distance between the states computed by the reservoir (using a contractive state function) for the root of the two input trees $t$ and $t'$ is upper bounded by a term proportional to $C^h$

$$\|\hat{\tau}(t, x) - \hat{\tau}(t', x')\| \leq C^h \text{diam}$$

(3.15)

The proof of the theorem can be found in [16]. The equation 3.15 states that the states computed for the root of different trees are clustered together in the space state based on a suffix-based fashion. Fig. 3.6 shows the Markovian nature of the state space organization in the TreeESN. The bounded state space assumption can be ensured using a bounded reservoir activation function. If the selected bounded function is $\text{tanh}$, then the contraction coefficient (identified in the following as $\sigma$) can be obtained using the euclidean distance as $\sigma = k\|\hat{W}\|_2$.

3.2.3 Hyper-parameters

In general, each neural network model has some of parameters to characterize the model and how the learning procedure evolves. They are called hyper-parameters. Each neural
network characterization offers different outcomes in terms of prediction performance. And, since the best combination of assigned values depends on the specific analysed task (i.e. the dataset samples, the samples structure and their target values), the right values for them must be appropriately tuned. The validation process is the procedure responsible for the hyper-parameter selection. Usually, a set of values is assigned for each hyper-parameter $H_{p_i}$ and all the configuration belonging to the hyper-parameter Cartesian product is analysed.

$$\text{VAL}_{H_{PS}} = \prod_{\forall H_{p_i} \in H_{PS}} H_{p_i}$$

For each neural network parametrization, with a specific value of the hyper-parameters, the correspondent model is trained and its prediction performance is examined. Eventually, the best performing model is found and the correspondent hyper-parameters values is used for the final learning procedure.

The TreeESN offers a lot of this hyper parameters that characterize the reservoir, input, output components. In the thesis, the used hyper-parameters are:

- **Reservoir Dimension** ($N_r$) determines the number of recurrent neurons inside the reservoir. It directly influences the expressive power of the learning hypothesis
space and the complexity of the model.

- **Connectivity degree of reservoir** ($W_{\text{sparsity}}$) expresses the connection degree between the reservoir neurons. The number of connections determines the complexity and the richness of the reservoir behaviour in terms of response signal. Usually, a small connectivity degree is considered, since sparse randomly connected network exhibits a behaviour based on the small-world property.

- **Scaling of the reservoir** ($W_{\text{scaling}}$) represents the numeric range within the reservoir connection weight are generated (if a connection between neurons exists). It determines the degree of non linearity exhibited by the reservoir neurons.

- **Contractivity factor** ($\sigma$) governs how the state dynamic evolves in the time. It is related to the echo state and the markovianity properties. A large contractivity value implies a long-term memory persistence of the system. This parameter represents, the facto, a rescaling factor of the reservoir weight matrix.

- **Input connectivity degree** ($W_{i\text{sparsity}}$) determines if a label field of the currently analysed tree node is taken into account for a specific neuron in the reservoir.

- **Input scaling** ($W_{i\text{scaling}}$) identifies the numeric range within the input weight are generated, if the specific label field is analysed for a neuron. As the $W_{\text{scaling}}$ parameter, it determines the degree of non linearity exhibited by the reservoir neurons.

- **Regularization factor** ($\lambda$) helps in tuning the generalization of the hypothesis built in the learning phase. It relaxes the hypothesis in order to reduce the over-fitting phenomena on the training data. It appropriately sets the regularization parameter in the ridge regression formula (see equation 3.13).
Chapter 4

Machine Learning for HPC Optimization Problems

With the constant improvement of the modern computers architectural features, both in terms of complexity and diversity, building a performance model becomes a more and more challenging task. Producing traditional analytic models may become a difficult and time consuming process, since it can be difficult to model/understand the relations among the causes and the effects (causal nexus). A long and error-prone process is needed to acquire the detailed characterizations of the target system and application. There could exist some cases in which the model built is no longer valid when next generation systems are taken into account. Moreover, analytic models require some important simplifying assumptions on the modelled domain that, sometime, may eventually concur to produce a poor final results.

In this context, the automatically built predictive models may be investigated. The approach shown in this chapter can be placed between an analytic model and an empirical optimization, as explained in Sect. 4.2. Briefly, in the empirical optimization method the whole or a big part of the input space is explored looking for elements having some features (also known as targets). The described approach consist in sampling a very small part of the input domain (also known as training set), targeting every sampled element and trying to infer via a machine learning technique a model that can represent the phenomena described by the sampled elements. The procedure to build the predictive model is agnostic with respect to the actual features of the system tested. Indeed, the learning system has no a priori information about the applicative domain except the ones exposed in the training phase. It can adapt to different system simply re-executing the targeting and learning phases. In the same way, since the model is built basing on
the performance values directly exhibited by the system, the achieved predictive system is tightly related to the benchmarked system and targeting a different system would require to re-run the entire procedure.

4.1 Auto-Tuning

A field in High Performance Computing where these topics has been examined are Automated Tuning (also known as auto-tuning). This is a well known technique that tries to find the best implementation for a specific program. Usually, a programmer starts with an high level operation he want to solve. There is a large design space of possible solutions, but he prunes them to a single program code realization. The compiler analyses it, and from the only available information in the programs tries to derive a safe and more efficient transformation. Automated tuning vision is that a plethora of different possible implementation is examined. The hope is that, in generating all these variants, some high-level knowledge is retained when the set is examined collectively. The compiler produces a set of implementations, that are examined by the auto-tuner and the best one is selected. The Alberto Sangiovanni Vincentelli’s diagrams reported in Fig. 4.1 shows this concept.

There are three major aspects involved with the auto-tuning:

- The *optimization space* represents the multidimensional space where the optimum solution is looked for. The simpler auto-tuner takes in consideration loop unrolling, restructuring loops, eliminating branches and explicit vectorization but more advanced ones may take into account also different data type, data layout or data structure. In the future, they will be able, even, to handle different type of algorithms for the same high level problem.

- The *code generation* must reflect the optimization space features. Usually they are generating using some scripting languages to facilitate the task.

- The *optimization space exploration* is done assigning the feasible values to the specific optimization features. For example, the loop-unrolling optimization may be parametrized by the number of loop iterations actually unrolled.

There are two major approaches used for auto-tuning in the literature:
• Analytic model-driven optimization

The model-driven models predict the desire objective basing its knowledge on a simplified abstraction on the system to be used. They are designed by hand developing a mathematical model supporting them. Model-driven models are a very powerful tools, in case they exhibit an accurate prediction level. However, they require a deep understandings of the system properties, and sometimes, with the growing of the decision variables, an accurate model can become very complex to be designed. Indeed, it can be difficult to understand/model relations between the input domain variables, where one of them may cause a non linear effect on the results or even on the other decision variables.

In the past, the model-driven methods were successfully used in the compiler community for the serial code utilizing simplifying but general assumptions. However, for parallel code they require more accurate high-dimensional models. Furthermore, some important information available at runtime can be completely missing at compile time. These limitations render the model-driven optimization approach
less attractive in many high-performance library.

- **Empirical optimization**

In the empirical optimization, all the decision variables space is explored looking for the element(s) having the desired characteristics. For each point of the input space, the target value is directly obtained via experiment or even via simulation. The probed value, coming from the system, is the one in which we are interested in and it is not a surrogate function value. Using this technique, it is possible to collect for each point in the explored space more targets of interest, providing a way to deal with multi-objective optimization problems.

However, this optimization process may require a lot of time, since frequently individual experiments/simulations are not a trivial tasks and the input space is generally huge and multidimensional. These characteristics make the full exploration infeasible, and some techniques are adopted to deal the problem:

- The domain of interest is reduced to a very specific context in such a way the space in which look for the solution is limited.
- The optimization space is explored with some technique coming from the artificial intelligence\(^1\) (usually the hill-climbing technique is used).
- The input domain is explored guided by some ad-hoc heuristics that offer little generalization guarantees. Usually, this heuristics are guided by the user knowledge on the application domain.

This technique has been successfully applied to many fields such as: linear algebra (both for sparse and dense ones), signal processing, sorting and general stencil optimizations.

### 4.2 Predictive Models using Machine Learning

There exists a third way that can be placed in between the empirical and the model-driven optimization. Machine learning techniques may be used to build a predictive model that is an approximation of the target function. This solution exhibits the advantage of both the previous solutions: it provides an predictive model that can be used

\(^1\)Well known algorithms coming from the artificial intelligence are available for the search in \(\mathcal{NP} - \mathcal{H}\) problem exploration. Some of the most used in the artificial intelligence field are hill-climbing, simulated-annealing, evolutionary and swarming techniques.
to achieve rapidly the target function value, in contrast, with the usual experiment/simulation that are generally quite slow. This make possible to make an empirical search in the optimization space using the obtained the target approximation function. However, usually, the predictive function is provided to the user in a not human understandable form. In this case, indeed, it is very complicated (almost impossible) to achieve some form of knowledge from the obtained model.

In the following, it will be explained how this “methodology” can be used with High Performance Computing applications. The first step need to be done is the selection of the type of machine learning tool need to be used. There exist different types of machine learning tools available depending on the task to be solved. Their dissertation is beyond the goal of this thesis, so, here the assumption of supervised regression task is done.

The second step needed is the selection of candidate instances subset of the optimization space on which is possible to execute training phase. Depending on how the information is structured, this phase influences also the previous step. In particular, having to deal with structured domain (e.g., sequences, trees, and graphs) leads to different type of machine learning techniques (see Sect. 3.1). In any case, in the following, the *Structured Domain to Element Transduction* is considered. A very important duty consists in spreading, as much as possible, the sampled instances trying to equally distribute them in the optimization space. Larger and multidimensional spaces usually require more sampled instances to be analysed to extract useful information from the model. Also the features set (referred as *label* in machine learning field) should be carefully selected to describe accurately the examined problem. The labels available for each instance can be rawly divided in:

- *Task details* represent the relevant information in terms of the analysed application.
- “*Implementation* details” represents the information need to be tuned.
- *Platform details* represent the information relative to the used runtime support. They can comprise Hardware, Operating System and Framework characteristics.

Obviously, the above subsets are not disjoint and some of them can be empty.

The next phase is relative to the collection of the target values. All the instances arranged in the training set must be marked with the value achieved via simulation or experiments (referred as target in machine learning field). As already stated this phase
can be very time consuming, but, since it is executed on a small part of the optimized space, it can be considered an acceptable load. In this phase, when the experiments are performed, a particular attention must be given to the level of noise exhibited by the platform under testing, that can negatively influence the learning effectiveness. For this reason, all the external entities that do not contribute to the experiment result should be stopped or retain to the minimum load. A further way to reduce the noise level is using the mean function over multiple runs.

Eventually, the model machine learning tool can be trained. The training is an automatic process that involves the training set to achieve the best model fitting the provided instances but at the same time tries to generalize them (avoiding to fall in overfitting). A recurrent problem in the learning, that can characterize many application field such as HPC, is the problem related to the loss function used. The loss function is used to minimize the error introduce by the learned hypothesis. The most used one Least Mean Square and it is defined as:

\[
\sum_{i \in TR} (y_i - h(x_i))^2
\] (4.1)

where the \(y_i\) represents the \(x_i\)’s target value while the \(h(x_i)\) is the model predicted value. As it can be seen from the equation, the minimized value depends on the difference of the above quantities. In high performance computing, this minimization method does not fit very well with the prediction objective where, usually, the error is measured in proportion with respect to the actual target value. An error quantity is treated at the same way if the target value is big or small, but a fixed error quantity is more significant on smaller target value rather then bigger one. One of the possible adoptable solutions is use the rebalancing technique\(^2\) to compensate the error produced by the least mean square function. The rebalancing puts a varying amount of emphasis on different region space where an higher error is expected. This means to replicate the samples in such way the error distribution is corrected. Therefore, each instance is replicated by a factor inversely proportional to the its target value. This technique has the nice property that the number of samples required for the training does not increase. This allows to use this technique avoiding to collect other data by simulation or experiment.

Once the model hypothesis has been built, a fast target approximation function can be used to explore the optimization space. The techniques that can be used are the

\(^2\)In the article [33] this technique is referred misleadingly as stratification. But, usually, in the machine learning field, the term assumes another meaning.
same used for empirical optimization, but this time the function to achieve the output value is faster. In this way, it is possible to explore the same optimization space faster or a wider one.

4.2.1 Example of some possible applications

The exposed methodology can be exploited in many fields of the high performance computing where an optimization process is necessary. This technique has already demonstrated its efficacy in very different contexts dealing with:

- Completion time prediction of SMG2000 and a LINPACK library

In [33] experiment, a single matrix operation has been taken into account. The input space has been defined in terms of the matrix dimension (on which it is necessary to compute the mathematical operation), some tunable operation options and the number of active entities involved in the computation. These entities are structured in a grid topology. Obviously, the input space has some constraints that, if not matched, make the solution infeasible: both in terms of the total number of processor utilized and the memory occupation. The article goal is build a predictive function for the matrix operation completion time. The regression task is accomplished using a standard feed-forward multilayer perceptron neural network. This model makes possible to explore the optimization space composed by the number of processes and the operation options.

- Performance and Power consumption Optimization on a Jacobi stencil

[17] considers a typical fixed stencil computation coming from the finite differential computations: 7-points and 27-points Jacobi stencil on a 3-dimensional matrix. In their experiments, developed in the auto-tuning context, the authors want to compute the best parameters for executing the stencil taking into account both performance and power consumption issues. The optimization space is relative to the number of threads used, and to the usage (and parameters) of some optimization techniques such as domain decomposition, software pre-fetching, data

\[3\] In the original article this parameter are rawly miscalled “processors”.

33
padding and inner loop optimization\textsuperscript{4}. The target domain comprises the power consumption parameter and a wide variety of performance estimators depending on the specific architecture utilized. In general, the last parameters are measured by cycles per thread, cache (for each cache hierarchic level) and TLB misses. The machine learning technique used to approach the problem is the kernel canonical-correlation analysis (KCCA).

\begin{itemize}
\item \textbf{GPGPU Kernel tuning Filterbank Correlation}

The \textit{Single Instruction Multiple Threads (SIMT)} model, provided by the GP-GPU architecture, supplies a potential powerful computational model but, at the same time, it imposes several constraints on the shared resource usage. Performance degradation may arise dealing with the available architecture parameters inappropriately. Tuning all this conflicting and intercorrelated parameters can be an heavy and tedious task. In [6], a filterbank correlation algorithm implemented using the Cuda kernel has been considered. This implementation provides some application specific parameters and a lot architecture dependant tunable parameters. The latter embrace different architecture features, such as the choice of type of memory to be used (shared, texture or global Memory), the number of threads in a block and the maximum number of registers to be used. The article goal is to find the best performing set of parameters for the filterbank correlation kernel using the machine learning \textit{boosted regression trees} technique.

\item \textbf{Network Chip-Multiprocessor design}

In [39], the network CMP design support by machine learning techniques is analysed such that performance and power consumption problems are dealt with using this techniques. The architecture of the processor to be developed can be simulated using the accurate micro-architecture simulator CASPER\textsuperscript{5} based on SPARC\textsuperscript{V}9 instruction set. CASPER is built with an accurate power estimation model of each microarchitecture block. In the experiments, the architecture design is constrained

\textsuperscript{4}The inner loop optimization is obtained via unrolling, jamming and reordering the stencil source code instructions.

\textsuperscript{5}The acronym CASPER means \textit{Chip multithreading Architecture Simulator for Performance, Energy and aRea analysis}.

34
to be a quad-core with a complete core interconnection and to a L2 caches by a
crossbar interconnection structure. Each core is in-order with 6 pipelined stages
and it utilizes the hardware multi-threading technique called Fine Grain Multi-
threading. The other architecture parameters has been left unbound in such a way
they define the optimization space. The simulation has been run on the described
architecture using the EnepBanch\textsuperscript{6} benchmark. The article goal is to find the
solution in the defined optimization space that fits the performance power con-
sumption needs. The Kernel Canonical Correlation Analysis (KCCA) machine
learning technique is used.

- **Thread Mapping Technique for Transactional Memory Application**

  The article [8] automatically infers the kind of appropriate thread mapping for a
specific Software Transitional Memory (STM) application. The thread mapping
identifies a specific thread mapping policy and, so it can be represented as discrete
values. The analysed thread mapping policies are: Linux, Compact, Round-Robin
and Scattered\textsuperscript{7}. The input space tries to model the existing relations hips between
the application, the framework and the platform. The input space has been divided
in 3 groups of attributes:

  - Category A models the relations between the application and the STM frame-
    work specifying the transition time and the transition abortion ratio.
  - Category B defines the way in which the STM framework handles the con-
    flicting transactions: how they can be detected and how the resolution can
    be carried on.
  - Category C represents the relation between the application and the platform
    highlighting the Last Level Cache (LLC) miss ratio.

  The target value identifies a specific thread policy. The used machine learning
technique is the ID3 decision tree algorithm, based on categorical discrete input
variables.

\textsuperscript{6}The EnepBanch uses a subset of the packets defined in CommBench telecommunication benchmark.
\textsuperscript{7}In the Compact thread mapping policy, adjacent threads are mapped to adjacent cores sharing
the layer 2 cache. In the Round-Robin thread mapping policy, adjacent threads are mapped to cores sharing
only the layer 3 cache. While in the Scattered thread mapping policy, adjacent threads does
not share any level of the cache hierarchy.
4.3 Our vision

The papers cited above concerning the evaluation of parallel programs mainly refer mainly to an unstructured way to deal with the parallelism. Only [17] considers the structured parallel programming addressing the problem for a specific case of data-parallel computation: the Jacobian stencils. The different Jacobian computation runs are identified by a bounded number of parameters representing the available options and some relevant features for the evaluation of the target. The program runs can be directly mapped in a flat machine learning data representation. Furthermore, the way in which the parallelism is explicated do not emerge from the dataset samples. Our trial is to insert the way in which the parallelism is exploited directly in the dataset samples using the skeleton methodology. Differently from the Jacobian programs representation, the skeleton trees are difficultly mapped in a flat representation. So, machine learning techniques dealing with structured domain (see Sect. 3.1) can be used. In particular, the tree structured representation perfectly fits the skeleton tree data without any loss of information. In this way all the information related to the parallel skeletons used as well as their parameters can be represented.

A thesis goal is to evaluate how machine learning for tree structured data techniques fit the needs of skeleton framework. In a new generation skeleton framework some source of knowledge related on how the skeleton trees behaves is necessary. Different orthogonal aspects, that the system should take into account. They range from performance to the energy consumption issues. This knowledge source can be implemented in different ways: as an analytic model driven by an abstraction model, by some form of probing (i.e. via specific benchmarks) and through a machine learning predictive model.

In next generation skeleton frameworks, the programmer provides the sequential code of the program to the framework. Then the framework analyses it and, after some phases, it is able to execute the program in parallel using the skeleton methodology. Fig. 4.2 depicts all the phases involved in the process.

At first, the code provided by the user is analysed by a refactoring tool that highlights the code regions where parallelization can be applied using skeleton patterns. The produced output is a skeleton tree program equivalent to the sequential provided by the user. The obtained tree is passed to the annotation phase that enriches the skeleton tree with annotations [1, 2]. They describe information ranging from the skeleton parameters to the abstract information of the architecture where the application will be deployed.
The *mapping phase*, instead, is aware of the features of the target platform. Therefore, it is able to build an executable that explicitly constrains each application component to the target architecture resources. The mapping strategy uses some internal knowledge. Once the program runs in parallel on the desired architecture, it can be handled dynamically by an *autonomic manager*. If some handled conditions are violated, a *reshaping/reannotation* procedure can be implemented. Obviously, since the autonomic manager is able to use the same (or slightly different) internal knowledge, it can evaluate the different solutions and evaluate which one better satisfies the non-functional constraints of the application. Once the best application configuration is found, the autonomic manager is able to restructure the application acting on the proper actuators.

Both for the mapping phase and for the reshaping/reannotation procedures, a set of skeleton trees representing a semantically equivalent application are generated. The trees generation process is driven applying the available rewriting rules, that can be applied totally or partially. If the issue need to be handled is a multi-objective problem, restraining the evaluated skeleton tree set may lead to a biased solution (i.e. only a sub-optimal solution can be achieved). Then, all the generated skeleton trees are evaluated using the internal knowledge to find the best configuration for the application.

The thesis investigates on how an internal knowledge based on machine learning predictive model can be built using the TreeESN model.
Chapter 5

TreeESN Library

In this chapter, it will be illustrated the machine learning library developed during the thesis. The framework is based on the TreeESN network, previously described in Sect. 3.2.

The code has been developed basing on some pre-existing Matlab scripts made by Claudio Gallicchio. Matlab\(^1\) provides an excellent environment to work with numerical computations such as linear algebra and statistical computations. It allows to prototype a new model in very short time with sustained performance, but it suffers from some drawbacks. The first one is that the Matlab framework is not free both in terms of provided licences and price. However, some free multi-platform alternatives exist. The most famous one is GNU Octave\(^2\) that allows a bidirectional syntactic compatibility with Matlab. The second drawback is the way in which mathematical operations are carried on. If a parallel implementation is provided for them, there is no possibility to understand how the parallelism is exploited and how the parallel execution characteristics can be tuned since they are framework dependant. For these reasons, it has been decided to port the existing code to a freely available, platform independent library.

The library has been reimplemented form scratch using the C++ programming language and the standard library on a POSIX environment. Its implementation is able to exploit efficiently the modern multicores architectures using the well known linear algebra library BLAS/LAPACK and the FastFlow framework for parallelism exploitation. The developed code (available in Appendix C) is going to be released under public domain\(^3\). The project will be published with the name of Tesn library.

\(^1\)Matlab website: http://uk.mathworks.com/products/matlab/

\(^2\)Octave website: https://www.gnu.org/software/octave/

\(^3\)The publishing licence will be a free to use one, but the specific library licence has not yet been decided.
5.1 System overview

The Tesn library has been developed with the idea to render, as much as possible, simple its usage. Thus, a typical object oriented abstraction has been provided to the application programmer. Every TreeESN component is seen as an instance of a class. A further abstraction based on procedural programming style has been provided. It allows to use the neural network even more simply focusing on the typical executed tasks and on the data produced by each learning phase. The second abstraction will be presented later in the Sect. 7.2.2.

In both developed abstractions, the dataset has been represented as an object. Its goal is to store all the information used for the training and the prediction phases when tree input domain is considered. All the samples contained in the dataset can be handled and acquired individually making possible to partition them for validation purposes (see Sect. 6.5). There are two ways in which it is possible to create a dataset object: programmatically or via parsing a dataset file. The library is able to deal with a specific dataset format for the tree dataset representation identified by the gph file extension. A description of this dataset format is presented in Sect. 5.2.1.

The first abstraction provides a view to the Tesn library directly interacting with the TreeESN base components. The reservoir network and the readout has been modelled as classes and every of them expose the methods allowing to deal with the regression task. Both tree-to-element and tree-to-tree\(^4\) transduction are supported by the library.

The Reservoir class models the input units and the reservoir neural networks. The proper methods to initialize them are provided in the Reservoir class. A random initialization is performed for the input units, while for the reservoir network also the normalization based on the contractive parameter ($\sigma$) is taken into account (see Sect. 3.2.2). Once the whole encoding network has been initialized, the reservoir class can compute the state of each sample. The state computation is performed by the programmer sample by sample. Different phases may be necessary to produce the final state depending on the type of transduction considered. In the first phase, every node of a tree sample is mapped to an encoded one. If the tree-to-element transduction is considered, a further computation step is needed to perform the state mapping. The state mapping is

\(^4\)Currently, the tree-to-tree transduction is partially supported since a proper dataset format and object, able to store target values for each tree node, has not been developed.
performed by using an external static class called `StateMapping`. Both the root state mapping and the mean state mapping function are implemented (see Sect. 3.2.1).

The `Readout` abstract class identifies the readout component. It provides methods to perform the training and the prediction. In the Tesn library, there may exist multiple implementations of the Readout class. They differ implementing various machine learning techniques for the solution of the regression problem. They can even expose different bias, both in terms of language and search bias. The Readout implementations are able to solve the regression task independently from the reservoir. Indeed, they can be used to solve the standard regression tasks for flat input domains. However, in the Tesn library, they are trained using the sample states computed by the reservoir and the correspondent target values. Once the readout has been trained, it is able to make prediction for new samples (or their states) presented as input. In the library, two different implementations of the Readout class have been realized. Both of them deal with linear regression problem, solving the least mean square problem (Eq. 3.11). The first readout network (`linearRegressionLMS` class) solves directly the problem making use of the pseudo-inverse matrix (Eq. 3.12), while the second one (`linearRegressionTichonov` class) uses the ridge regression technique (Eq. 3.13). The way in which the two operations are carried on by the library is explained in detail in Sect. 5.2.2.

### 5.2 Implementation details

In the following, some implementation details of the Tesn library are taken into account and discussed.

#### 5.2.1 Dataset files

A dataset file contains all the necessary information to perform the learning and the prediction tasks for tree input domains. The dataset handled by the library is stored in a file having a specific format identified by the file extension `gph`. The contained information is acquired by the `TreeDatasetParser` class using a proper predictive parser based on the following grammar.

```
DATASET ::= HEADER TREE SAMPLES
```
The dataset is composed by an header and a body part. The header contains the common information characterizing all the tree samples in the dataset. It contains the number of available samples, the maximum tree arity allowed and the number of labels contained in each tree node. The body part represents the collection of all the tree samples in the dataset. Each tree in the dataset is characterized by a string name identifying univocally the sample in the dataset, its target for the considered task, the dimension of the tree expressed in terms of the nodes number and a nodes description. Every node is represented by a unique numeric identifier, the child identifiers, the parent identifier and the labels characterising the node. All nodes are sorted in the inverted topological order, in such a way it is possible to build easily and efficiently the data structure starting from the tree frontier. This node organization speedup the state computation phase. Indeed, since it requires to analyse the child nodes before their parents, the node information can be achieved sequentially exploiting the memory hierarchy levels to improve the performance.

The number of contained grammar productions is not detected directly by the grammar, but it is rather achieved by analysing the numeric attribute contained in the dataset description. For instance, the TreeNum attribute specifies the number of tree samples contained in the dataset. Basing on the previously described grammar and the information contained in the dataset, a top-down left-to-right parser able to acquire predictively all the information available in the dataset file has been developed.

The library is able to handle an unique dataset format, however others dataset formats can be considered implementing new parsers able to produce a dataset object. For instance, a new parser capable to handle tree-to-tree transduction dataset can be
implemented.

5.2.2 Matrices mathematical operations

All the information required by the Tesn library has been represented using matrices. For both the input units and the reservoir networks, the connections between input-reservoir and between reservoir neurons have been represented through adjacency matrices. At the same way, the dataset samples and the states produced by a dataset partition are stored in matrices too. Thus, in the developed library, the matrix data structures fulfill a key role and a way to treat them adequately must be found. All the matrices are represented as dense matrices of double values. But, sometimes, it is necessary to represent them in the equivalent sparse representation as happens for the reservoir connection matrix. Indeed, since the reservoir network benefits from the small-world property, it is possible to use sparsely connected reservoir to training the TreeESN model without afflicting the accuracy of the obtained predictive model. For this reason a proper procedure to compute the tree state has been provided both with dense and sparse matrix. The dense matrices are represented using the Dense class, while the sparse matrices using the Sparse class. Both of them provide methods allowing to load/store the matrix content from a file.

The operations performed on matrices are the most critical part in the Tesn library. Different kinds of operations must be implemented to perform the training of a TreeESN model and their implementations strongly influence the performance and accuracy of the training procedure. So, it is pretty natural the choice of relying on an external library to perform these operations in the manner. The mathematical library should solve the problems related to:

- **Numerical Stability** – Since the numerical linear algebra operations taken into account may involve multiple computation steps, the library should limit the numeric error produced with the state-of-art algorithms. The algorithmical error typically derives from rounding and truncation errors propagating during the computation.

- **Performance** – The implementation of the linear algebra operations must efficiently exploit the nowadays architecture to achieve better performance. For example, it must take advantage from the vector ISA (Instructions Set Architecture) available for the target architecture.
In principle, also the parallel exploitation of the target architecture may be taken into account for the selection of mathematical library to be used. However, in the Tesn library this aspect is not fundamental since in Chap. 7 it is assumed that the parallelism is exploited at a coarser grain in the validation procedure.

In the Tesn library, the mathematical operations are carried on using the Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) library. Both libraries define an interface used, by many vendors, as reference guide for different implementations. These libraries provides a de-facto standard for scientific computing when linear algebra operation must be faced. The BLAS library provides some basic linear algebra operations such as vector scaling, vector dot products, linear combinations and matrix multiplication. This library is used as building block for the implementation of very famous programming environments such as MATLAB, GNU Octave, Mathematica and R and in the LINPACK, LAPACK libraries. The LAPACK library makes use of the BLAS library to perform efficiently some complex linear algebra operations. The library provides some operations like the LU, Cholesky, QR matrix factorizations; inversion, linear system solution; and operations dealing with eigenvalues and singular values. In the Tesn library, two different implementations haves been tested: the OpenBLAS\(^5\) and the Intel MKL\(^6\) library. As stated previously, the BLAS/LAPACK library has been used in a context where a single operation is carried on by a single thread. For the OpenBLAS this can be ensured compiling the library with the appropriate flags. Instead, in the Intel MKL library it is sufficient to link appropriately the right object module.

In the Tesn library, the implemented code is designed to be easily optimizable by the compiler. For example, in the state computations the code has been written to take advantage from the vectorization optimization available in the target machine, both for the sparse and dense matrix representations. For the other and more complex mathematical operations, some calls to the BLAS/LAPACK library have been used to deal efficiently with the tasks. The implemented phases relying upon this library are:

- **Linear Regression Training** – For both the learning methods explained in Sect. 3.2.1 the mathematical library has been used. In particular:
  - The classical Least Mean Square problem (Form. 3.11) has been solved calling the \texttt{LAPACKE\_dgels} function.

\(^5\)OpenBLAS library – http://www.openblas.net/

43
The Tikhonov regularization (Form. 3.13) has been implemented using the basic multiplication and transposition matrix operations (in the BLAS library) and an inversion procedure. The latter one has been computed using the LAPACK library by decomposing the matrix with the LU factorization ($LAPACKE_dgetrf$) and then applying the inversion ($LAPACKE_dgetri$).

- Reservoir Initialization – The reservoir initialization is based on the normalization of the contractivity factor ($\sigma$). To apply the normalization is necessary to compute the norm-2 of the reservoir connectivity matrix ($W$) and then apply the normalization using the equation

$$\hat{W} = W \frac{\sigma}{k||W||_2}$$

where $\hat{W}$ is the normalized reservoir matrix and $k$ is the tree arity. The norm-2 has been computed by the $LAPACKE_dgesvd$ function using a singular value matrix decomposition.
Chapter 6

Machine Learning Task

In this chapter, the construction of a task dealing with the completion time for skeleton applications will be analysed. The chapter discussion follows entirely the methodology exposed in Sect. 4.2, starting from the creation of the datasets until the training of the TreeESN models.

6.1 Analysed task

Firstly, the task analysed to train a predictive model must be selected. In the following, a task explored very well in literature has been chosen. Selecting a widely investigated task makes possible to compare the prediction results by machine learning models with the “state of art” analytic model available. Furthermore, it is possible to check how the machine learning predictive models behave in some shadow zones that are not adequately treated by the currently available analytic model. For this experimental phase, the analysed task is the one concerning the performance prediction issues. However, others prediction tasks will be examined in future (see Chap. 9).

The analytic cost model presented in Sect. 2.3 is based on an abstract process architecture. But the parallel applications need to be run on a target architecture that has a limited number of resources. Thus, the abstract modules are not always mapped in the concrete architecture in a completely independent way one from the others, since they may share some common architectural resources. It must be remarked that the analytic model does not take into account any architectural resources sharing. The typical analytic model assumption states that the target architecture must satisfy the parallel degree required by the skeleton application. In the following, this case is referred as lack of parallelism while the opposite case, it is referred as excess of parallelism.
The experiments (reported in Sect. 8.2) investigate how a predictive learning predictive model can discriminate the completion time of a skeleton applications, even in the case excess of parallelism is used. The skeleton applications are run on a multithreaded multicore architecture. This architecture provides an additional sharing of resources between entities running on the same core (e.g. sharing the execution units).

6.2 Data collection

In this phase, all the tree instances selected to constitute the dataset and the associated target value must be gathered. In this section, the identification of the skeleton application samples, the generation of the correspondent FastFlow programs and the acquisition of their completion time are treated.

6.2.1 Dataset generation

The dataset is composed by different characterizations of the skeleton tree applications. Different parallel skeleton compositions must be available in order to correctly sample all the optimization space. Furthermore, the dataset must be a representative subset of the instance on which the prediction is requested. Each skeleton tree is annotated with the parameters useful for the performance prediction task. Each skeleton contains information regarding:

- the sub-skeletons for the farm and pipeline skeletons (allowing the definition of the tree data structure),
- the number of workers for the farm skeletons, and
- the completion time for the sequential skeleton (expressed in milliseconds).

The adopted strategy for the dataset samples generation is based on the skeleton rewriting rules. The splitting and the introducing parallelism rewriting rules (Sect. 2.2.1) are applied to a sequential skeleton. Applying multiple times the rewriting rules, it is possible to build a decision tree containing all the available skeleton alternatives. This data structure is composed by “Skeleton” and “Abstract Skeleton” nodes. A Skeleton data structure represents the classical compositional capability of the skeleton: all the specified sub-skeletons must be taken into account (AND logic). But, instead of nesting
directly a skeleton, a list of possible sub-skeletons alternatives is provided via an Abstract Skeleton node. All the skeletons in that list, since they are produced using the rewriting rules, are semantically equivalent. Thus, each one can be used alternatively (XOR logic) as nested skeleton. The resulting built tree is composed alternatively by AND and XOR layers. Visiting adequately this tree starting from the root is possible to enumerate a wide number of skeleton rewritings. The rewriting rules are applied to several sequential skeletons to differentiate as much as possible the obtained rewritten skeleton trees. An example of a possible resulting decision tree is shown in Fig. 6.1.

![Decision Tree Diagram](image)

Figure 6.1: The decision tree obtained applying multiple times the rewriting rules

Each produced sample is then dumped to a Skeleton Tree Program file (.fttp) for future computations. This file representation is the native way to identify a sample. Every row in the file describes a skeleton among the following types: Sequential (SEQ), Farm (FARM) and Pipeline (PIPE). Every skeleton is uniquely identified by a reference name that it is possible to utilize for compositional purposes. The skeletons are described in terms of:

- the computation time and the type of memory accesses for the sequential skeleton,
- the nested sub-skeleton and the number of workers for the farm skeleton,
• the nested subsketons for the pipeline skeleton.

Furthermore, there are a header and a footer line identifying respectively the input and output stream features expressed in terms of the length of the stream, the dealt tasks size and the timing characterizing the interarrival time (for the stream source) and the consumption task time (for the stream sink).

In the produced datasets:

• The number of produced samples in the dataset is 1340.

• The skeleton trees available have at most a nesting height equal to three.

• The sequential skeletons, used to generate all the rewritings, are characterized by a completion time randomly chosen from a uniform distribution in the time interval $[50, 250]$ milliseconds.

• For any parallel application corresponding to a dataset instance, the input/output streams are characterized as follow:
  – the input stream has a length of 800 tasks,
  – the task size is 8 bytes,
  – the input tasks are generated almost instantaneously (i.e. the interarrival time is equal to 0),
  – no time consuming operation is done at the end of the stream.

• A sample balancing has been preserved in such a way that the samples presenting excess of parallelism are almost the same number as the ones who do not (45.9%).

6.2.2 Program generator

The program generator takes as input a *Skeleton Tree Program* and produces a fully working C++ parallel application using the FastFlow Skeleton Framework. The code, generated by the abstract description of the parallel application, is stored in a *FastFlow Program* file (identified by the file extension *.ffp*). The code generator has been realized
All the generated parallel applications have a pipeline skeleton as root node of their skeleton tree. Its purpose is to handle the tasks stream and performs the computation in the declared way. The first pipeline stage \((\text{IN})\) handles the input stream by generating a task every specified amount of time (i.e. interarrival time to the skeleton application). The last one \((\text{OUT})\), instead, is able to collect a result after a specified time period is actively spent in such a way that it is possible to model the time spent for delivering or storing the computed task. The central stage is where the tree skeleton to be analysed resides. The pipeline and the farm skeletons are implemented in the standard way as described in Sect. 2.1. The emitter (E) and the collector (C) modules in the farm respectively dispatches and collects the task without computing any other task.

The sequential skeleton behaviour has been defined in such a way it is able to spend a specified amount of time in a \textit{synthetic computation}. The sequential function makes double precision computations on a local array opportunely sized \((N\) elements) composed by double precision numbers. For each array position, \(M\) sinusoidal functions computations are performed. Each function takes as input the value of the previously performed trigonometric function. Opportunely dimensioning the size of the array size \((N)\) and the number of the computed sinusoidal functions \((M)\), it is possible to define a computation both in terms of the computational and memory access behaviours. The synthetic computation is repeated different times in order to wait for the desired amount of time in an active computation. Fig. 6.2 shows an example of a parallel application architecture built by generating the FastFlow code.

### 6.2.3 Labelling

The samples have been labelled by executing the generated FastFlow program on a Intel PHI 5100 based on the Knights Corner architecture. This architecture is composed by 60 cores having 4 multithreading contexts each. For labelling purposes, the architecture has been restrained, using the \textit{taskset} Linux command, to an architecture having 16 cores with 2 multithreading contexts per core. The taskset command can set a processess CPU affinity bounding a process to a given set of “logical CPUs” on the system.

All the generated FastFlow programs have been executes several times achieving all the run completion times. The outliers have been discarded and the mean completion time of the remaining runs values is taken. This procedure gives more stability to the
result making it independent from the statistical fluctuation. In the dataset, the outliers have been eliminated simply keeping out the minimum and the maximum achieved completion time values. The target value of each sample has been produced executing 6 times the correspondent FastFlow program.

Fig. 6.3 shows the distribution of the target values (expressed in milliseconds) in the generated dataset.

Figure 6.3: Distribution of the generated samples target sorted in ascending way.
6.3 Representing the data

The information of the samples in the dataset must be properly mapped in a machine learning data format. Since each skeleton application is represented as a tree, it is natural to use a tree representation also for the learning task. This choice guarantees that all the information of the data structure (comprising its structure) are exploitable for the learning task. If an adaptive or semi-adaptive transduction mechanism is used, they can be exploited without inferring any kind of information in a static tree encoding. All the information related to the tree structure are kept unaltered in the sample representation. In this way the learning procedure can deduce the relevant information with respect to the faced task directly from the structure and the labels of the dataset samples.

Once the choice of the tree representation is done, it is fundamental to decide the relevant information for the task and where this information should be put on. Indeed, in a tree structured domain, the information can be inserted as a vertex label and/or as an implicit information available in the skeleton of data structure itself (i.e. connection information between vertexes). The assumption done in the following is that the skeleton of the tree represents only the algorithmical skeleton compositionality feature. This assumption leads to represent the nested skeleton in a farm as a single child node and the information about the parallel degree is placed as a label information in the farm vertex. The alternative representation, in which every farm worker is represented as a child node, conducts to a huge sample representation where the information concerning the parallel degree are moved from a label to the sample data structure. Such a kind of data representation makes the learning task more complex and time consuming.

Once an information is decided to be represented as a label field, the right place where the information should be put must be identified. Indeed, the same information can be represented in different way taking into account the visibility level the information should have in the tree sample. In the previously done assumption, the parallel degree of a farm skeleton has been placed in a farm node. However, the same information can be restructured differently in the tree simply rephrasing it. The parallel degree information can be expressed differently substituting it with the replication information. The replication information specifies how much times a particular skeleton structure must be replicated. If a skeleton is contained in a farm as a nested skeleton, its structure must be replicate the number of time the farm parallel degree specifies. In this way, an equivalent concept can be represented propagating the information from the farm node to the radicated subtree.
All nodes in the tree are represented in an uniform manner: pipe, farm and sequential skeletons own the same labels format. However, understanding the kind of a single node is important to know how to compute the performance prediction task. Thus, the machine learning models must be able to understand the skeleton type represented in a node. Firstly, the identification of a skeleton type is provided by its tree structure. Indeed, it is possible to determine the skeleton represented in a node looking at the number of its sons: the sequential skeleton does not have any sons, the farm skeleton has a single nested skeleton, while the pipeline owns at least two nested skeletons. A further way to discriminate the represented skeleton type has been provided by the well known 1-of-K technique. Three boolean labels has been designed for this goal. These labels can not assume multiple times the true value, and at least one of them must be set to true. The true value has been represented in the dataset with the 1 integer value.

A further representation choice has been accomplished to make possible future dataset analysis. Every tree node representation posses also an identifier to its parent node. This choice gives more liberty in the selection of the machine learning model allowing the use of contextual analysis techniques.

In the following, it is described the utilized datasets representations:

- **Original Dataset** – It represents the “base” dataset representation. All the others dataset, described below, are obtained modifying or adding some labels from the original dataset. The tree represented in the dataset are binary trees with a maximum height equal to 4. The average node nodes number per tree is 8.406. The target associated with the tree represents the completion time associated to the correspondent skeleton tree expressed in milliseconds. The node representation is composed by the node identifier, the 2 sons identifiers, the father identifier, the 1-of-3 skeleton representation, the service time and parallel degree. If a node has not a child or a parent, the identifier to then is represented in the node representation using the special value −1. It is important to notice that the information provided by the nodes identifiers do not belong to the label. Indeed, they define the structure of the tree. The 1-of-3 labels identify the skeleton represented by a node: 0 0 1 is used to identify a sequential skeleton, 0 1 0 for the farm skeleton, while the 1 0 0 is used for pipeline skeleton. The service time field is valid only for the sequential skeleton, for the other skeleton the value has been set to 0. The parallel degree is set to 0 for the pipeline skeleton, 1 for the sequential ones, while for the farm skeletons assume a value greater or equal then 1.
• **Replicated Dataset** – The original dataset has been modified in order to restructure the parallel degree information. The parallel degree information has been replaced with the replication degree of each skeleton. In this dataset, the information related to the exploited parallelism percolate the tree and reach the leaf nodes.

• **Cost Model Dataset** – The original dataset has been modified in such a way to include information achieved by the analytic cost model. Every tree node is annotated with an additional field containing the predicted service time. This dataset has been built to verify how much the inclusion of additional information coming from the model can improve the prediction of a learning model for the faced task. The predictive learning model has the possibility to approximate better the target trends exploiting additional analytic information. Furthermore, its usage makes possible to generalize this provided information in the regions of the optimization space where the analytic model does not fit adequately the targets.

• **Synergetic Dataset** – This dataset merges both the enhancements carried by the Replicated and the Cost Model datasets. Thus, the parallel degree information has been replaced with the replication degree and a further field has been added to include the information coming form the analytic model.

In Fig. 6.4, it has been reported graphically the four different tree sample representations. The tree sample named “0007”, in the produced dataset, is taken into account. Its representations can be seen in the Appendix A. The tree structure, reported in the figure as nodes and arrows, has been extract from the node identifier information. Since the tree structure information is not modified in the different representations, the tree structure is the same in all the representations. The information that change are those in the label. The 1-of-k information, representing the skeleton type, has been shown as a label inside the node using the letters F, P, S to identify respectively the farm, the pipeline and the sequential skeleton. This information does not change in the different representations. Instead, the other label informations, the ones that actually change, are displayed in a rectangle near the node. Comparing the “Original” representation with the “Replicated” one, it can be seen that the farm parallel degree information propagates to the child nodes assuming a different meaning (i.e. the number of times that the skeleton is replicated in the parallel application). Comparing the “Original” representation with the “Cost Model” one, it can be seen that the basic label information do not change, but the analytic model information has been added in the label. As it can be seen easily from the figure, the “Synergetic” representation merges the label
modifications adopted by the “Replicated” and the “Cost Model” representations.

\[\text{Original} \quad \text{Replicated} \]

\[\begin{array}{c}
\text{CostModel} \\
\text{Synergetic}
\end{array}\]

Figure 6.4: The different tree sample representations are shown graphically for the sample named “0007” in the produced dataset.

The datasets have been built basing on the process highlight in Fig. 6.5. A Skeleton Tree Program is analysed to produce a local representation of the sample (called Represented Instance) for the dataset representation. Once the information about the skeleton structure are acquired from the skeleton application description, several pre-processing phases are sequentially executed to produce the desired output format for the data. The order in which this phases are computed is relevant since their composition is not commutative. The pre-processing phases comprises:

- Id Assignment assigns an integer unique identifier to each node. The identifiers are assigned starting from the number 0 and visiting the tree in inverse topological
order. Furthermore, this phase sets appropriately, in each node, the identifier of the parent node.

- **Decompose Pipeline** reduces the arity of the tree to a specified number, passed as parameter. The procedure rearranges the nodes belonging to the tree. A splitting function is applied in correspondence of the pipe skeleton and it reduces its arity degree. This pre-processing phase takes care of producing a tree as balanced as possible.

- **Reverse Parallel Degree**, basing on the parallel degree label contained in each node, computes how many time each skeleton is replicated in the whole parallel application. This pre-processing phase is optional.

- **Service Time Computation** reckons the service time for each tree node using the analytic cost model. This pre-processing phase is optional.

\[
V' = \frac{V - V_{\text{min}}}{V_{\text{max}} - V_{\text{min}}} \tag{6.1}
\]

Figure 6.5: The schema depicts how a dataset can be built starting from a set of Skeleton Tree Program. In the figure, it is shown the targeting and the representation procedures respectively in the upper and lower side.

Once all the desired preprocessing phases are computed, the obtained information can be stored in a file using the inverse topological order. Several represented instances, belonging to the same dataset, are analysed together to perform a post-processing phase. This phases analyses the instances of the dataset and it performs some modifications basing on the whole group. For our purposes, this phase is optional and when it is taken into account only a normalization procedure is executed. For all the previously described datasets have been generated also a version in which the labels are normalised in the interval \([0, 1]\) using the equation
The $V_{\text{min}}$ and $V_{\text{max}}$ represents respectively the minimum and the maximum value for a label in a specific dataset (among all the analysed represented instances). The $V$ variable represents the currently analysed label value. This normalization phase does not include the 1-of-K labels.

The dataset represented instances, obtained from the post-processing procedure, can be merged together with the respective target values to produce the final representation of the sample in the dataset. Then, all of them are grouped together in a single file representing the whole dataset. The dataset has been stored in the .gph format described in the Sect. 5.2.1. An excerpt from all the datasets and the respective meaning of the node label fields are provided in the Appendix A.

### 6.4 Selecting the machine learning model

Since the sample data are represented in the dataset as trees (Sect. 6.3), the selection of the appropriate neural network model is straightforward. A neural network model able to deal with tree structured data is necessary. This choice promotes the use of the information present in the structure of the tree for the prediction goal. For example, a structured domain machine learning model is able to discern the type of skeleton in a node directly from the structure of the sample trees.

Among the different possibilities in machine learning models able to deal with tree structured domains, the choice fell on the network reservoir computing methods. Thanks to their computational properties, as claimed in [25], the reservoir methods are very suitable for a first analysis. Indeed, they are powerful model but, at the same time, the constraint bias allows a faster training with respect to the standard recursive neural networks. The TreeESN model (Sect. 3.2) has been selected for this experimentations. The accomplished choice implies some design features on the language bias: the stationary and the causality assumption. These assumptions can limit the expressiveness power of the predictive learning model. However, they fit perfectly the needs for a first analysis and they can help to discover which assumption can be relaxed to improve the prediction accuracy of the faced task.
6.5 Training the learning model

The method used for training a predictive learning model, during the experiments, makes use of a model selection and a training phases. As already explained in Sect. 3.2.3, a machine learning model can be characterized by some hyper-parameters that leads the evolution of the system during learning. Different hyper-parameters bring to different results in the accuracy of the achieved predictive model. The way in which the best hyper-parameters is selected is called model selection, sometimes it is referred as validation process. Several neural networks characterized by different hyper-parameters are evaluated on a validation set to select the best preforming hyper-parameters tuple. Then, the highlight configuration can be used for the final learning procedure. For understanding the goodness of the obtained learning model, it is necessary to perform a prediction procedure to evaluate the performance of the selected model. This prediction phase estimates the accuracy of the realised model on a partition of the dataset that has not be used for the training. In the experimental results, both the validation and test tasks are taken into account.

The holdout method is used to face the model selection and the performance prediction tasks. Using the holdout technique, the dataset has been divided in 3 partitions called respectively training, validation and testing set. The model selection phase is performed on the training and validation set. All the neural network hyper-parametrizations are trained on the training set and their performance are evaluated on the validation set. Once the best performing configuration is found, the model is retrained on the union of the training and validation set using the identified hyper-parameters tuple to initialize and train the neural network. The performance of the model is evaluated by measuring the error of the samples in the test set. A typical partitioning of the dataset for the holdout procedure is shown in Fig. 6.6.

![Figure 6.6: The typical partitioning of the dataset for the holdout procedure in which the dataset has been divided in 3 partitions of equal size.](image)
The partitioning of the dataset has been done using a \emph{stratification} processes. The samples in the dataset are properly selected in order to provide an equal distribution of their target values in all the partitions. Thus, all the samples has been ordered ascendantly using their target values and they have been partitioned using the following partitioning ratio:

- $\frac{1}{3}$ of the samples have been used for the test set,
- $\frac{2}{9}$ for the validation set and
- the remaining $\frac{4}{9}$ for the training set.

For both the validation and the learning process an additional hyper-parameter has been added. Since the connection weights associated to the input and reservoir networks are randomly chosen, several neural network instances can be generated from the same hyper-parametrization tuple. This allows to verify the goodness of a hyper-parametrization distinguishing from the specific network instance generated. Indeed, generating different representatives for the same hyper-parameter tuple permits to furnish the accuracy of a TreeESN network both in terms of its mean and standard deviation. The number of the trials done has been identified by an additional hyper-parameter called \textit{NTrials}. 
Chapter 7

Parallel Implementation of the Model Selection

The model selection and the test procedure, exposed in Sect. 6.5, are very computationally intensive. Even if the time required for evaluating a single TreeESN hyper-parametrization is short, with respect to the traditional recursive neural networks, the time necessary to build a predictive model for a specific faced task, validate and test it is very long. In Sect. 8.1.1 a deep analysis on the time required for completing an instance of a validation procedure has been presented. For this reason, the full validation/test procedures has been parallelized. Two different implementations has been realized. Both the versions are based on streaming parallelization. The choice to parallelize the learning procedure in a dataparallel way has been discarded for different reasons, such as the required developing time and the problem related to the numerical stability issues arising from the parallelization (as explained in Sect. 5.2.2). In any case, the streaming parallel implementation of model selection process would not lead to significantly different results in terms of completion time.

The two different parallel solutions addressing the validation/test problem has been developed using FastFlow [12]. FastFlow is a C++ skeleton framework that grants both programmability and efficiency. The parallel patterns are built on top of an efficient communication mechanisms that provides lock-free and memory fence free synchronization mechanisms. The library targets cache coherent shared memory multicore architectures. The programmability of the frameworks comes from the different parallel patterns implemented (provided as primitive objects to the application programmers), mainly exploiting streaming parallelism\(^1\). Furthermore, the possibility to use the effi-

\(^1\)However, the FastFlow framework provide some dataparallel patterns.
cient lower level mechanisms supports the possibility to build from scratch the missing parallel pattern or to adapt the already existing ones.

7.1 Farm parallelization

The first parallelization developed is a classic, naive solution. It is based on the general model selection process and, so, it can be applied to every kind of neural network model. In the model selection procedure, all the hyper-parameters combinations are tested to look for the best performing parameter configurations. All these trials are independent and so an embarrassing parallelization schema can be implemented.

The embarrassing parallel computation has been implemented using a farm skeleton realized using a master-worker parallelization schema (shown in Fig. 7.1). The tasks of the input stream are produced by the master. It generates all the possible combinations for the hyper-parameters, and dispatches them to the workers. The workers are able to train independently the neural network configurations and verify the performance achieved by the analysed input hyper-parametrization. In detail, the fundamental steps executed for the TreeESN neural network are:

1. the initialization of the reservoir and the input neurons,
2. the state computation for both the training and the validation set,
3. the learning phase and,
4. the performance estimation of the built predictive model.

All this operations are executed sequentially on a single parallel entity. Since the master-worker has been implemented using FastFlow framework, the considered parallel entities are threads. Once the elaboration of a task has been completed, the workers store the results in a common array data structure indexed by the hyper-parameter configuration. Every parametrization is mapped to a single array position in a bijective way. So, since an hyper-parameter is dispatched only to a single worker, no resource contentions happens among them while accessing the shared data structure.

The scheduling policy used by the master is the on-demand one. Thus, a backwards message is sent to the master to signal the completion of a task. Receiving this task the master is aware that the worker has completed the elaboration of a previously scheduled task and a new one can be submitted to be computed. The on-demand scheduling policy
Figure 7.1: The figures show master-worker parallelization schema implemented using FastFlow. The tasks dispatched by the master are the neural network hyper-parametrizations.

has been set in order to have a load balancing between workers.

The hyper-parameters values in a task are generated keeping in consideration the amount of calculation they will generate. Sorting them from the most to the least computationally intensive, it is possible to reduce the unbalancing between workers. Indeed, if the tasks are sorted in the ascending computationally order then the last tasks take a long time to be computed and some workers may result underutilized if the stream is exhausted. Ordering the tasks in the reverse way, a similar situation happens but the idle time where the workers are underutilized decreases, allowing to achieve better performance both in terms of completion time and scalability. Luckily, it is very simple to obtain a rough estimation of the most computational intensive task looking at the number of the reservoir neural units (identified by $N_r$) in the hyper-parameters tuple.
7.2 Macro-DataFlow parallelization

The second parallel implementation is based on the macro-dataflow high level parallel pattern. In this parallelization an analysis of the data flow in the learning process is performed trying to identify all the steps in the computation that can proceed in parallel using a dataflow graph.

7.2.1 The macro-dataflow parallel pattern

The macro-dataflow [5, 7] is an high level parallel pattern. It exploits the data dependencies to schedule the required operations in parallel. The dependencies between data are described using a dataflow graph (a Direct Acyclic Graph) by arrows connecting computational phases. A dataflow graph can be instantiated multiple times, in case the computation must be repeated (e.g. with different input values). The macro-dataflow instructions represent instances of computational steps in the dataflow graph. In order to make possible to distinguish the same dataflow step between different instances, the macro-dataflow instructions are provided with an unique identifier. A macro-dataflow instruction specifies the function required to be executed on a given input. Direct edges highlight the data dependencies arising from the dataflow graph. They point out the input/output of the macro-dataflow instructions defining how the computed data must be exchanged between macro-dataflow instructions. The input of a macro-dataflow instruction may be also obtained specifying directly its value (i.e. specifying a constant value or referencing a value coming from outside the dataflow graph). In every cases, the input of a macro-dataflow instruction must specified when the macro-dataflow instruction is created.

All the macro-dataflow instructions, including the ones ready to be executed, are stored in a macro-dataflow repository. A distributed interpreter is able to understand and execute ready ones (fireble) in a parallel fashion. A macro-dataflow instruction is fireable if and only if all its input data dependencies are satisfied. The macro-dataflow distributed interpreter is composed by multiple interpreters working in parallel. The number of interpreters contained defines the maximum exploitable parallel degree. A macro-dataflow interpreter proceeds in the computation following the next steps:

- Every interpreter instance is able to logically access to the macro-dataflow repository. A fireable macro-dataflow instruction is extracted from the repository to be computed.
• Before the interpreter is able to compute the macro-dataflow instruction, the interpreter must obtain all the input data and the function to be computed. Sequentially, the function can be applied to the input to produce the output result.

• Once the output data is computed, some further data dependencies may be satisfied. Thus, all the input dependencies, related to the computed output data, of macro-dataflow instructions are marked as fulfilled.

7.2.2 The new interface

Since the macro-dataflow parallelization is based on the functions that can be run in parallel, a decomposition of the training process used in the model selection must be identified. In that way, it is possible to build a new interface for the Tesn library. It comprises the all functions typically used in the training phase of a TreeESN model. These functions are more intuitive rather then the ones provided object oriented interface (Sect. 5.1). The procedural interface is focussed on the operation accomplished by the TreeESN rather then its components. It allows to concentrate on the data passing between functions and on the hyper-parameters characterizing that particular computational step. However, this interface constraints the library usage to a limited number of defined functions. Obviously, since the procedural interface is based on the object oriented one, the two interfaces can be interleaved by the programmer to benefit from both.

The functions belonging to the interface are able to deal with a linear regression task faced via ridge regression using a tree-to-element transduction. The identified building block functions in the validation process are:

• Input Neurons Initialization – Initializes the input part of the encoding network producing a dense matrix representing the connection matrix between inputs and reservoir neurons. The hyper-parameters influencing this phase are:
  – Reservoir Dimension \( (N_r) \)
  – Input Connectivity Degree \( (W_{ispmctiy}) \)
  – Input Scaling \( (W_{iscaling}) \)

• Reservoir Initialization – Initializes the reservoir component producing a sparse matrix representing the connection matrix among neurons in the reservoir. The hyper-parameters influencing this phase are:
  – Reservoir Dimension \( (N_r) \)
- Reservoir Connectivity Degree ($W_{\text{sparsity}}$)
- Reservoir Scaling ($W_{\text{scaling}}$)
- Contractivity Factor ($\sigma$)

- **States Computation** – Produces the state of a particular dataset fold using a previously initialized encoding network. The operation result is provided in a dense matrix.

- **Training** – Trains a linear regression unit with the ridge regression technique using a particular dataset fold and the correspondent state. The operation produces a $\text{linearRegressionTichonov}$ object. The hyper-parameter influencing this phase is:
  - Regularization Factor ($\lambda$)

- **Performance Prediction** – Uses a $\text{linearRegressionTichonov}$ object to perform the evaluation of the built predictive model. The evaluation has been done on two different folds providing their states. The two folds typically represent the training and the test fold.

### 7.2.3 Implementation details

The function identified in the previous section (Sect. 7.2.2) are used to build a dataflow graph exploiting the dependencies on the data. Fig. 7.2 shows the existing dependencies between the different TreeESN training steps. The initialization of the encoding network can be executed in parallel since they are independent. The state computation can start only once the encoding network is built. Two different state computations are performed to achieve the state of the samples contained in the training and testing set. Once the state related to the training set is computed, the Tichonov regression can be used to produce a regularized predictive model. After a linear regression hypothesis as been built for the dataset task, the model accuracy on the training and test set state can be performed. Some cleanup procedures are necessary to keep as much as possible the memory consumption low.

Several instances of the dataflow graph are instantiated basing on the hyper-parameters optimization space. In Fig. 7.2, the hyper-parameters values are highlighted by bold arrows. The generation of the macro-dataflow instructions allows to reutilize some computation steps for some similar hyper-parametrization. For instance, the output of the reservoir network initialization can be reused for computing multiple time the states (i.e.
Figure 7.2: The dataflow graph used for the TreeESN learning procedure.

with different input networks). In the same way, the training set state can be reused multiple times for building different linear hypothesis based on many regularization factors.

The macro-dataflow parallel pattern implemented in FastFlow has been used for the implementation of the parallel validation process. It is composed by two service modules plus a number of workers defined from the user. Its component architecture can be seen in Fig. 7.3. The task generator module performs the generation of all the macro-dataflow instructions needed to perform the computation. The generated task are delivered to a macro-dataflow engine (also known as parallel interpreter). It is implemented with a master-worker parallel pattern. The task dispatcher (representing the master) receives the macro-dataflow instructions and, when all the dependencies are solved, it dispatches the macro-dataflow instruction to the workers. The workers execute the macro-dataflow instruction. Once a dispatched instruction is completed, a loop-back message is sent to the task dispatcher in order to signal the completion of the instruction and the availability of a data that can make other instructions fireable.
Figure 7.3: The figure shows the macro-dataflow parallelization scheme implemented using FastFlow. The tasks scheduled by the Task Dispatcher are the fireable instance of the operations shown in Fig. 7.2.

In the FastFlow macro-dataflow implementation, when a macro-dataflow instruction is generated the following parameters must be specified:

- The function code that must be executed
- The input data dependencies it must satisfy in order to be fireable
- The output data dependencies macro-dataflow instruction unlock at the moment the function complete its execution
- The actual parameter used to call the specified function

In the FastFlow framework, the input/output dependencies are usually expressed specifying the pointer to the involved data. However, since the data involved in the model selection process can be allocated and deallocated, the same pointer to the data can be referenced in different time instants. Furthermore, some macro-dataflow instructions are not truly based on data dependencies. Indeed, the cleanup procedures are based on a task dependencies. The dependencies based on tasks are shown in Fig. 7.2 with dashed arrows. For this reasons, all the dependencies in the macro-dataflow has been converted in task based dependencies. For each dataflow step, shown in Fig. 7.2, has been assigned
a numeric identifier that specifies the instance of the treated task. Each macro-dataflow instruction, instantiating a particular computational step in the dataflow, concatenates its identifier with the identifier of its operation type using bits operations. In this way, all the dependencies space has been partitioned between the type of the dataflow instructions.

As stated before, the actual parameters of the function to be called must be specified once the macro-dataflow instruction is generated. This implies that all the input parameters of the macro-dataflow instructions must be known early. Some of the function parameters, such as the hyper-parameter tuples, are known in advance. But others, like the reference to the data generated in previous computational steps, are not available. For this reason, some additional data structures able to store pointers to the not yet created object are used. Using double pointers, it is possible to bypass the problem providing in advance an input to the macro-dataflow instruction.

The default scheduling mechanism in the FastFlow task dispatcher is based on the maximum number of unlockable tasks (i.e. the maximum number of the macro-dataflow instructions that become fireable after the completion of a macro-dataflow instruction). This policy prefers to dispatch a task that can unlock multiple tasks promoting the maximum exploitable parallelism. It is a very useful policy when the dependencies are referred to an already allocated data structure, as happens, for instance, in the Cholesky and LU factorization [7]. Instead, in the analysed context, where the data are progressively allocated and deallocated from the memory, it does not fit very well. Indeed, using the standard scheduling method, the amount of used memory tends to grow in time without control. For this reason, a new scheduling policy based on the order of the macro-dataflow instruction submission has been used. This new scheduling mechanism has been integrated in the current FastFlow version. In this way, if the FastFlow application programmer generates the macro-dataflow instructions in such a way the data locality is preserved, even the macro-dataflow engine satisfies this requirement. A deep-first generation of the macro-dataflow instructions is implemented basing on the hyper-parameters values. The pseudo code below shows the way in which the macro dataflow instructions are generated. The code focusses on the dataflow operations and the hyper-parameters values.

```
for each nr in Nr
    for each <sigma, w_sparse, w_scal> in <Sigma, WSparsity WScaling>
```
emit ( InitW, rho, w_sparse, w_scal );
for each <wi_sparse, wi_scal> in <WiSparsity WiScaling>
    emit ( InitWi, wi_sparse, wi_scal );
    emit ( ComputeTSSState );
    emit ( ComputeTRState );
    for each lambda in Lamda
        emit ( Train, lambda );
        emit ( Estimation );
    emit ( Cleanup_States_and_Wi );
emit ( Cleanup_W );

Finally, the macro-dataflow tasks are ordered in such a way they grant a satisfactory load balancing between workers when the input stream is exhausted. As it happens for the master-worker parallel implementation, it is sufficient to generate the tasks taking into account their computational cost and ordering them in an ascending order. Since, it can be assumed that the operations workload is greater when the reservoir network is bigger, it is sufficient to generate the macro-dataflow instructions basing on the reservoir networks dimension ($N_r$ hyper-parameter).

### 7.3 General Applicability

In the previous sections, the parallelization methods are exposed with regards to the validation procedure, anyway they can be applied also to the test procedure. Considering the aleatory nature of the input and reservoir networks, the same hyper-parametrization tuple can generate different results in terms of performance prediction. Generating different representatives for the same hyper-parametrization permits to provide the accuracy of a specific TreeESN hyper-parametrization in terms of its mean and variance value.

Furthermore, the parallelization methods described in in the previous sections can be applied also to other machine learning models. Since the validation process is based on the evaluation of all the combinations of the hyper-parameters, the farm parallelization can be applied independently from the machine learning model used. The macro-dataflow parallelization, instead, has a more limited applicability. Firstly, the used dataflow schema can be directly applied also to the Echo State Network model, since TreeESN model can be considered a its generalization. However, it can be used for all the reservoir based neural networks making simple changes in the data dependencies.
description (dataflow) of the learning procedure. In general, the parts that need to be modified regard the reservoir initialization and the cleanup procedures. If it is possible, the reservoir initialization should be decomposed, to express the potential parallelism and the data reuse, restructuring the data dependencies in the dataflow graph. Obliviously, also the cleanup procedures should be modify accordingly to reduce how much is possible the global memory consumption.
Chapter 8

Results

This chapter describes the experimental results obtained in the thesis. Firstly, the performance results obtained by the Tesn library (Chap. 5) and the parallel implementations of the model selection process (Chap. 7) are tested. Later, the results obtained by the validation/testing process are discussed for the predictive learning model obtained from the design cycle described in Chap. 6.

8.1 Performance results

In this section, the experimental results of the Tesn library and the two parallel implementations concerning the validation process are presented. The experiments have been run on Pisanosa. Pianosa is a dual processor Intel Xeon E5-2650 (2x8 cores, two way hyper-threading) based on an Intel Sandy Bridge architecture. In particular, the completion time of the different steps for training and testing a TreeESN model have been measured. Moreover, the two parallel versions have been evaluated in terms of their exhibited performance (i.e. completion time and scalability). The tests have been executed on the “Original” dataset without the labels normalization (discussed in Sec. 6.3). The dataset has been partitioned in a training and a validation set. The partition ratios, associated to each set, are respectively of 2/3 and 1/3. The TreeESN hyper-parameters (Sect. 3.2.3) values used for the parallel implementation of the validation process are shown in Tab. 8.1.

8.1.1 Computational phases cost

A testing on the steps involved in the learning procedure has been performed in order to understand the overall performance of the Tesn library. The different experiments are
Table 8.1: The TreeESN hyper-parameters values used as input parameter for the parallel implementation tests.

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial</td>
<td>{0, 1, 2, 3, 4}</td>
</tr>
<tr>
<td>Nr</td>
<td>{750, 500, 250, 100}</td>
</tr>
<tr>
<td>Sigma</td>
<td>{0.1, 0.4, 0.7, 1, 3, 6, 9}</td>
</tr>
<tr>
<td>w_scaling</td>
<td>{0.01, 0.1, 1}</td>
</tr>
<tr>
<td>w_connectivity</td>
<td>{0.05, 0.1, 0.2}</td>
</tr>
<tr>
<td>win_scaling</td>
<td>{0.01, 0.1, 1}</td>
</tr>
<tr>
<td>win_connectivity</td>
<td>{1}</td>
</tr>
<tr>
<td>Lambda</td>
<td>{0.05, 0.1, 0.15, 0.2, 0.25}</td>
</tr>
</tbody>
</table>

computed varying the dimension of the reservoir network and measuring the time spent in the different phases of the TreeESN model training and testing7.2.2. In Tab. 8.2, test results are reported. The state computation and the testing refer to the time elapsed analysing the whole dataset samples, while the learning algorithm is performed on a partition of the dataset. The partition used for the learning procedures are 2/3 of the whole dataset.

Table 8.2: The time spent in the different computational phases (in milliseconds).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Reservoir Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Initialize Wi</td>
<td>0</td>
</tr>
<tr>
<td>Initialize W</td>
<td>1</td>
</tr>
<tr>
<td>State computation</td>
<td>7</td>
</tr>
<tr>
<td>Tichonov learning</td>
<td>2</td>
</tr>
<tr>
<td>LMS learning</td>
<td>10</td>
</tr>
<tr>
<td>Testing</td>
<td>0</td>
</tr>
<tr>
<td>Total (using Tichonov)</td>
<td>11</td>
</tr>
</tbody>
</table>

As it can be seen from Tab. 8.2, the most intensive computational phases are the state computation, the initialization of the reservoir network and the learning procedures. In the following, only the Tichonov regularization is used. It grants both performance and regularization characteristics with respect to the algorithm using the pseudo-inverse (LMS learning). The “Total” row shows the time elapsed for training a TreeESN model.
using a fixed set of hyper-parameters and testing the trained model. The TreeESN learning procedure has been performed using a sparse matrix ($\hat{W}$) to compute the samples states. The reservoir connectivity has been fixed to 0.2. The conversion of the reservoir connectivity matrix, from dense to sparse, has been performed in the $\hat{W}$ initialization phase.

### 8.1.2 Master-worker implementation

The master-worker implementation of the model selection process has been tested varying the parallel degree associated to each execution. The results obtained are shown in Fig. 8.1 and in Fig. 8.2, that display, respectively, the completion time of the whole validation process and the scalability.

![Figure 8.1: The completion time of the master-worker parallelization performed on Pianosa.](image)

As it can be seen from Fig. 8.1, the time required to perform the validation process is huge: the master-worker takes about 7 hours when only one worker is available. However, the parallelization, in this case, does not make reuse of the already computed data. The analysed solution exhibits a good scalability that is limited only by the number of physical cores (Fig. 8.2). Indeed, as soon as the number of worker in the farm implementation grows beyond the number of the physical core available (i.e. 16 cores), the scalability slope changes drastically and the application does not achieve any further scalability (maximum scalability = 14). This behaviour can be explained taking
into account that the Tesn library works using double precision data. The Intel Xeon E5-2650 processors are unable to sustain the huge amount of double precision operations required, since the execution unit becomes a bottleneck in the execution of the mathematical instructions. Moreover, the scalability results are further influenced by heavy vectorization optimizations exploited by the sequential code in the Tesn library that increases the amount of double operations per second.

### 8.1.3 Macro-dataflow implementation

The macro-dataflow implementation of the validation process has been tested varying the parallel degree associated to each execution as we did for the farm implementation. The results obtained are shown in Fig. 8.3 and in Fig. 8.4, that show, respectively, the completion time of the whole process and the scalability.

As it can be seen from Fig. 8.3, the completion time of the macro-dataflow parallel version outperforms the farm one, completing the validation process in about 1/5 of the time. In the macro-dataflow version, many computational steps reuse already computed results rather than computing them again and again. Reusing the data, the validation procedure becomes faster taking about 82 minutes when the macro-dataflow has only one worker. This solution exhibits a similar scalability behaviour with respect to the analysed master-worker parallel version. As it happens in the former case, the scalability
Figure 8.3: The completion time of the macro-dataflow parallelization performed on Pianosa.

Figure 8.4: The scalability of the macro-dataflow parallelization performed on Pianosa.
is limited by the number of physical cores for the same reasons explained in Sect. 8.1.2.

Since the performance of the macro-dataflow parallelization has shown to be very efficient both in terms of completion time and scalability, a further test on a different architecture has been performed, namely on an Intel Xeon PHI 5100 coprocessor with 60 cores 4-way multithreading. The obtained results, see Fig. 8.5 and Fig. 8.6, show a behaviour similar to the one observed onto the Intel Xeon Sandy Bridge architecture, scaling up to 64 multithreading contexts.

Figure 8.5: The completion time of the macro-dataflow parallelization performed on PHI.
Figure 8.6: The scalability of the macro-dataflow parallelization performed on PHI.

8.2 Results of predictive learning models

In the following, the results obtained for the predictive learning model are presented in terms of prediction accuracy. The results are presented using two different metrics. The first one is the mean absolute error (MAE) and it is computed as:

\[
MAE = \frac{\sum_{i \in DS} |y_i - h(x_i)|}{|DS|}
\]  

where \(y_i\) and \(h(x_i)\) are, respectively, the target and the predicted value for the sample \(i\). \(DS\) represents the dataset partition taken into account and \(|DS|\) is its cardinality. As already explained in Sect. 4.2, this metric does not fit very well our needs. Indeed, the metrics does not allow to discern the error quantity with respect to the target (i.e. an error difference is treated at the same way if the target value is big or small). A second metric, named mean absolute normalized error (MANE), has been introduced to overcame the issue. It has been defined as:

\[
MANE = \frac{\sum_{i \in DS} \frac{|y_i - h(x_i)|}{y_i}}{|DS|}
\]  

The MAE and MANE metrics are useful to understand how the error is distribute with respect to the target values. The former measures the absolute difference between the sample targets and the predicted values, while the former measures the relative error.
In the following, the results obtained by the TreeESN model are analysed taking into account separately the datasets that make use of analytic model and the ones do not. The validation process has been performed with the hyper-parameters reported in Tab. 8.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial</td>
<td>{0, 1, 2, 3, 4}</td>
</tr>
<tr>
<td>Nr</td>
<td>{100, 250, 500, 750, 1000}</td>
</tr>
<tr>
<td>Sigma</td>
<td>{0.1, 0.4, 0.7, 1, 3, 6, 9}</td>
</tr>
<tr>
<td>W_scaling</td>
<td>{0.01, 0.1, 1}</td>
</tr>
<tr>
<td>W_connectivity</td>
<td>{0.05, 0.1, 0.2}</td>
</tr>
<tr>
<td>Win_scaling</td>
<td>{0.01, 0.1, 1}</td>
</tr>
<tr>
<td>Win_connectivity</td>
<td>{1}</td>
</tr>
<tr>
<td>Lambda</td>
<td>{0.05, 0.1, 0.15, 0.2, 0.25}</td>
</tr>
</tbody>
</table>

Table 8.3: The TreeESN hyper-parameters values used during the validation process.

The test process has been performed computing 10 different TreeESN instances characterized by hyper-parametrizations identified in the model selection process. For each of them, the MAE and MANE metrics are computed. In the following, the mean and the standard deviation of the achieved metrics values are provided to examine easily the performance results.

### 8.2.1 Datasets without analytic model information

In this subsection, the TreeESN model is tested on the datasets composed by the information available in a skeleton tree. The datasets used in this subsections are the “Original dataset”, the “Replicated dataset” and their normalized versions. For sake of simplicity, in the following these datasets will be referenced using the baseline datasets term.

The results obtained by the TreeESN validation/test process are analysed taking into account the root and the mean state mapping (Sect. 3.2.1). But, since preliminary tests has been highlighted that the mean state mapping performs worst than the root state mapping, two additional hyper-parameter values for the \(Nr\) hyper-parameter \((2000, 3500)\) has been added in order to produce an accuracy aligned to the root state mapping. Using a larger reservoir network entails a bigger hypothesis space in which the solution can be found. If the TreeESN network does not overfit the training data
with larger reservoir, it helps the model to obtain a better accuracy. However, the training and the prediction phases become more computationally intensive. Since the mean state mapping performs worse than the root one, in this subsection only the root state mapping is discussed. In Appendix B, the results obtained using the mean state mapping are reported.

In Tab. 8.4, the errors exhibited during the validation/test phase are reported to understand the behaviour of the TreeESN model in the different phases: training, model selection and test respectively. The analysis has been done on the best performing model trained using the baseline datasets. The error obtained in validation and test, respectively VAL and TS, is approximately two times bigger than the ones achieved in the trainings, respectively TR and TRVAL. However, while the training errors remain almost constant for the validation and the test phase, the test error is better than the validation one. This means that a similar performance in training achieves a better performing model in prediction when the training dataset partition is bigger (i.e. the union of the training and test partitions is used).

<table>
<thead>
<tr>
<th>Metric</th>
<th>TR</th>
<th>VAL</th>
<th>TRVAL</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>2528.802</td>
<td>4573.899</td>
<td>2538.841</td>
<td>4261.183</td>
</tr>
<tr>
<td>MANE</td>
<td>0.137118</td>
<td>0.229945</td>
<td>0.138344</td>
<td>0.191294</td>
</tr>
</tbody>
</table>

Table 8.4: The test error obtained by the TreeESN model trained using the “Replication Normalized” dataset in the validation and test phases when root state mapping is used.

Tab. 8.5 shows the hyper-parameters values selected by the validation process for each baseline dataset. The reservoir network, identified by the model selection, is characterized by the maximum number of neurons (1000), a small connectivity degree (5-10%) and an high contractivity values (6-9). In accordance with the small-world property, the small connectivity degree value grants a sufficient richness of the reservoir signal and, at the same time, it allows an efficient state computation if the sparse matrices are used. The high values of the contractivity coefficient indicate that the reservoir exhibit a longer memory in the state computation. This condition may not ensure a global stability of the reservoir and it is a signal that the analysed task does not have a purely Markovian nature.

Tab. 8.6 reports the errors obtained in the testing phase. The best model, highlighted in this phase, is the one trained on the “Replication Normalized” dataset. Its
test error measured with the MANE metric is 19.1%. The errors obtained during the validation/test phases are displayed in Tab. 8.4.

The dataset normalization does not always enhance the model accuracy. An example of this phenomenon can be observed comparing the results of the “Original datasets”. Furthermore, using the root state mapping it is not clear if the “Replication” representation helps the TreeESN model in achieving a more accurate model. However, the normalization joint with the “Replication” representation make possible for the TreeESN model to achieve a better accuracy.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nr</th>
<th>Sigma</th>
<th>WScal</th>
<th>WSpar</th>
<th>WiScal</th>
<th>WiSpar</th>
<th>Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1000</td>
<td>3.0</td>
<td>1.00</td>
<td>0.05</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Original Norm.</td>
<td>1000</td>
<td>6.0</td>
<td>1.00</td>
<td>0.10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Replication</td>
<td>1000</td>
<td>3.0</td>
<td>1.00</td>
<td>0.10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.10</td>
</tr>
<tr>
<td>Replication Norm.</td>
<td>1000</td>
<td>6.0</td>
<td>1.00</td>
<td>0.05</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 8.5: The TreeESN hyper-parameters values selected by the model selection for the baseline datasets when the root state mapping is used.

Table 8.6: The test errors, reported using the MAE and MANE metrics, exhibited by the baseline datasets when the root state mapping is used.

8.2.2 Dataset including cost model information

In this subsection, the TreeESN is tested on the datasets that make use of analytic model information. The datasets used in this subsections are the “Cost Model dataset”, the “Synergetic dataset” and their normalized forms. For sake of simplicity, in the following these datasets will be referenced using the CM datasets term.
The results obtained by the TreeESN validation/test process are analysed taking into account the root and the mean state mapping. As in the previous subsection, the mean state mapping exhibits worst accuracy using the same range of hyper-parameters values. Even the change to the \( N_r \) hyper-parameter values, done in the previous subsection, has not made substantial improvement in the alignment of the test errors. Thus, in this analysis, the root state mapping outperforms the mean one. For this reason, in this subsection only the root state mapping will be analysed. However, in Appendix B, the results obtained using the mean state mapping are reported.

In Tab. 8.7, the errors exhibited during the validation/test phase are reported to understand the behaviour of the model in the different phases: training, model selection and test respectively. The analysis has been done on the best performing model obtained using the CM datasets. The training error in the test (TRVAL) worsens with respect to the one obtained in the validation phase (TR). While, the obtained test error (TS) is smaller than validation error (VAL). This phenomenon points out that by analysing more samples, the model trained in the test phase is able to perform a better generalization of the target function.

<table>
<thead>
<tr>
<th>Metric</th>
<th>TR</th>
<th>VAL</th>
<th>TRVAL</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>337.470</td>
<td>426.884</td>
<td>357.563</td>
<td>353.909</td>
</tr>
<tr>
<td>MANE</td>
<td>0.027487</td>
<td>0.035498</td>
<td>0.031040</td>
<td>0.032447</td>
</tr>
</tbody>
</table>

Table 8.7: The test error obtained by the TreeESN model trained using the “Synergetic Normalised” dataset in the validation and test phases when root state mapping is used.

Tab. 8.8 shows the hyper-parameters values selected by the validation process for each dataset. The number of recurrent neural units has been set to the maximum value available for the \( N_r \) hyper-parameter (1000). The obtained values of the contractivity coefficient are less than 1. When the contractivity coefficient value (\( \sigma \)) is less than 1, the reservoir dynamics is more stable and, since the markovianity property holds, the reservoir memory becomes shorter. In the non normalized datasets, their contractivity coefficient values (0.1) indicate that in the most of the information is taken by the root while too little information is taken by the rest of the tree. Instead, the normalized datasets exhibit optimal \( \sigma \) values providing stability and making possible to exploit the state information coming from the whole tree structure. Since the analytic model information are computed from the tree frontier to the root, they can be included in the tree data structure in a Markovian fashion. Thus, the added information are suitable
for the TreeESN model that has a Markovian space organization.

Table 8.8: The TreeESN hyper-parameters values selected by the model selection for the CM datasets when the root state mapping is used.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nr</th>
<th>Sigma</th>
<th>WScal</th>
<th>WSpal</th>
<th>WiScal</th>
<th>WiSpal</th>
<th>Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Model</td>
<td>1000</td>
<td>0.1</td>
<td>0.01</td>
<td>0.05</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Cost Model Norm.</td>
<td>1000</td>
<td>0.4</td>
<td>0.10</td>
<td>0.10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.10</td>
</tr>
<tr>
<td>Synergetic</td>
<td>1000</td>
<td>0.1</td>
<td>0.10</td>
<td>0.10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Synergetic Norm.</td>
<td>1000</td>
<td>0.7</td>
<td>0.01</td>
<td>0.20</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Tab. 8.9 reports the errors obtained in the testing phase. The test errors exhibited by the models trained with the CM datasets are significantly reduced with respect to one trained with the dataset not including the analytic model information. The best model selected is the one trained on the “Synergetic Normalised” dataset. Its test error measured with the MANE metric is 3.2%, while the one measured with the MAE metric is 353.9 ms. The errors obtained during the validation/test phases are shown in Tab. 8.7.

The dataset normalization brings benefits both in terms of the average and the standard deviation of the error metrics. Adding the information coming from the analytic model becomes clear that the replication degree of the skeleton present in the “Synergetic dataset” helps the TreeESN in the regression task.

Table 8.9: The test errors, reported using the MAE and MANE metrics, exhibited by the CM datasets when the root state mapping is used.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MAE avg</th>
<th>MAE std</th>
<th>MANE avg</th>
<th>MANE std</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Model</td>
<td>708.919</td>
<td>120.395</td>
<td>0.048090</td>
<td>0.006281</td>
</tr>
<tr>
<td>Cost Model Norm.</td>
<td>362.598</td>
<td>1.324</td>
<td>0.033243</td>
<td>0.000124</td>
</tr>
<tr>
<td>Synergetic</td>
<td>562.192</td>
<td>71.052</td>
<td>0.037781</td>
<td>0.001959</td>
</tr>
<tr>
<td>Synergetic Norm.</td>
<td>353.909</td>
<td>2.377</td>
<td>0.032447</td>
<td>0.000236</td>
</tr>
</tbody>
</table>

8.2.3 Bipartite analysis

The best performing TreeESN models obtained by the model selection phase (i.e. for each dataset the correspondent TreeESN model is provided) has been analysed in order
to understand how the error is distributed in the dataset. For the bipartite analysis, the dataset samples have been partitioned in two sets: a set containing the samples that use the excess of parallelism and another set containing the samples that use lack of parallelism. Tab. 8.10 reports the errors for the two sets obtained by the TreeESN learning TreeESN models that use the root state mapping. The errors between the two groups result to be almost balanced. Indeed, the learning model tries to find a general law able to describe both phenomena simultaneously. The best performing model in each dataset group (i.e. the TreeESN models trained respectively on the “Replication Normalized” and “Synergetic Normalised” dataset) exhibits a better accuracy for the cases where excess of parallelism is used. Instead, as it can be seen in the data reported in Appendix B, the best mean state mapping models manifest an opposite result: a better accuracy is achieved for the cases where excess of parallelism is not used.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MAE Excess of Parallelism</th>
<th>MANE Excess of Parallelism</th>
<th>MAE Lack of Parallelism</th>
<th>MANE Lack of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>4310.560</td>
<td>0.243736</td>
<td>4691.284</td>
<td>0.234404</td>
</tr>
<tr>
<td>Original Norm.</td>
<td>5878.041</td>
<td>0.278088</td>
<td>5346.958</td>
<td>0.322037</td>
</tr>
<tr>
<td>Replication</td>
<td>5727.742</td>
<td>0.281724</td>
<td>5257.885</td>
<td>0.345942</td>
</tr>
<tr>
<td>Replication Norm.</td>
<td>4200.160</td>
<td>0.186524</td>
<td>4303.483</td>
<td>0.194600</td>
</tr>
<tr>
<td>Cost Model</td>
<td>852.198</td>
<td>0.052202</td>
<td>583.662</td>
<td>0.044495</td>
</tr>
<tr>
<td>Cost Model Norm.</td>
<td>320.228</td>
<td>0.029780</td>
<td>391.429</td>
<td>0.035599</td>
</tr>
<tr>
<td>Synergetic</td>
<td>693.873</td>
<td>0.042058</td>
<td>448.617</td>
<td>0.034093</td>
</tr>
<tr>
<td>Synergetic Norm.</td>
<td>314.454</td>
<td>0.029251</td>
<td>381.006</td>
<td>0.034643</td>
</tr>
</tbody>
</table>

Table 8.10: Bipartite test errors analysis of TreeESN model obtained by the model selection phase using the root state mapping. The analysis is performed using the MAE and MANE metrics on the baseline and CM datasets.

8.2.4 Analytic model comparison

The analytic model presented in Sect. 2.3 has been tested in order to understand the goodness of the results obtained by the TreeESN model. The tests has been performed on the parallel application samples used to create the datasets. Tab 8.11 reports the errors exhibited by the analytic model with respect the MAE and MANE error metrics. The mean error manifested by the analytical model is 4.4%. This result evidences that the TreeESN model trained using the “Synergetic Normalised” dataset with the root
state mapping obtains, on average, better performance then the analytical model, while the one built using the “Replication Normalized” is far to obtain good result.

Furthermore, the error reported by the analytical model is very low when the skeleton applications make use of lack of parallelism (0.8%), while it is an order of magnitude bigger for the cases in which excess of parallelism is used (8.8%). These results highlight that the TreeESN model trained using “Synergetic Normalised” dataset actually generalizes the analytic model by distributing the errors between the samples making use of excess of parallelism and the others. Thus, the results obtained by the best TreeESN model show that it is able to perform a generalization between the two samples groups. This TreeESN model obtains a better result with respect to the analytical model on average and in particular when the excess of parallelism is used. However, the analytical model cannot be replaced when the lack of parallelism is used.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Average</th>
<th>Excess of Parallelism</th>
<th>Lack of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>393.176</td>
<td>697.964</td>
<td>144.142</td>
</tr>
<tr>
<td>MANE</td>
<td>0.0443</td>
<td>0.0884</td>
<td>0.0082</td>
</tr>
</tbody>
</table>

Table 8.11: The mean errors exhibits by the analytical model for the generated dataset using the MAE and MANE metrics.

8.3 Evaluation of the studied methodology

In this section, the methodology examined during the thesis is going to be further evaluated. The accuracy results achieved by the learning TreeESN model has been already discussed in the previous section (Sect. 8.2). Thus, the analysis in this chapter is focused on the applicability of the examined task (Sect. 6.1) to a specific computation architecture.

As already explained in Chap. 4, the predictive model built using a machine learning technique is platform dependant. Indeed, the learning model has to be trained using the samples labelled with the target information relative to a particular architecture. In the thesis, the architecture selected to be analysed is a constrained Intel Phi 5100
coprocessor\textsuperscript{1}. For the analysed learning task, if a new architecture/platform has to be faced, the process of the dataset construction and the learning process must be repeated. In details, the phases that need to be processed are:

1. Tree samples generation – The skeleton trees representing the parallel application must be regenerated in order to face with the specific analysed architecture, taking into account the available parallel degree in the architecture. This operation can be fully automatized using the rewriting rules.

2. Tree samples labelling – A relabelling of all the samples must be performed. Since the target of the analysed task regards the completion time, the targeting phase is computationally burdensome. Furthermore, the labelling process has been performed associating to each sample the mean time of multiple runs (see Sect. 6.2.3). In the thesis, this process has lasted about 65 hours. But, since the tests are carried out with coarse grain (i.e. in order of milliseconds), their duration can be limited by reducing by one order of magnitude the computational grain. This change makes the targeting procedure approachable.

3. Dataset building – The tree samples with the correspondent targets has to be encoded in a dataset using the best samples representation (i.e. “Synergetic Normalised”) obtained by the accuracy results shown in section 8.2.

4. TreeESN training – The TreeESN model has to be retrained on the created dataset. Since both the validation and test phases must be considered, the TreeESN model trained is computationally intensive. However, it should be remarked that the TreeESN model result to be very efficient with respect to other machine learning models dealing with tree domains. In the thesis, using the Holdout technique, the validation phase lasts about 7 hours\textsuperscript{2} with the hyper-parameters values highlighted in the thesis (see Tab. 8.3). The developed parallel implementation based on macro-dataflow parallel pattern (see Sect. 7.2) helps in reducing the time spent in the validation/testing phases. It makes possible to validate and test the model in 29 minutes\textsuperscript{2} using a parallel degree of 20.

\textsuperscript{1}The Intel Phi coprocessor has been constrained to appear as if it had 16 cores (2 contexts per core) only.
\textsuperscript{2} The time are referred to the dual processor Intel Xeon E5-2650.
Chapter 9

Conclusions

In this thesis, it has been analysed how a predictive model based on a machine learning methodology for structured data can be used in structured parallel programming. The predictive models are used as internal knowledge in a framework able to refactor a parallel application. Different equivalent configurations of the parallel application are tested in order to select the best performing one. The structured parallel programming model has been provided by the skeleton programming paradigm. Thus, the skeletons represent the only mechanism available to express parallelism in an application. Since the skeletons are composable and nestable, a skeletons tree can be used to represent a parallel application structure. The skeleton trees are used as input domain for our machine learning model. The Tree Echo State Network has been used to deal with the tree structured input domains.

The thesis contributions can be divided in two main parts:

1. Development of a TreeESN library – The TreeEsn library has been developed in C++ using the BLAS/LAPACK mathematical library to deal with the linear algebra operations for the TreeESN training. The TreeESN model selection process, which is very computational intensive, has been parallelized. Two different parallel solutions, based on a streaming parallelism, have been provided those actually speedup the computation. They are implemented using the farm and a macro-dataflow parallelization patterns respectively. The solutions target the multicores architecture through the FastFlow framework. The macro-dataflow implementation outperforms the farm based one in terms of completion time. However, both the parallel implementations achieve good performances in terms of speedup. The identified parallelization schema can be reused, with some limitations, for other machine learning models.
2. Experiments on a specific machine learning task – A typical problem in high performance computing has been examined using a predictive model based on the TreeESN. The selected task faces with the prediction of the completion time when skeleton tree is used to model a parallel application. It takes also into account the case in which the analytical models usually fail in predicting a correct result (i.e. when excess of parallelism is used excess of parallelism is used). A full machine learning “design cycle” has been implemented starting from scratch. The set of structured parallel programs, that has to been evaluated, has been generated, labelled with target information and stored in a dataset. Different datasets representation has been produced in order to understand which one fits better the performance prediction task. A subset of the tested datasets includes the information coming from an analytical predictive model in order to understand how and how much the TreeESN can benefit from them. All the datasets has been subject of a validation/testing procedure (using the holdout technique) to achieve the best characterization of the TreeESN and to evaluate the accuracy of the obtained predictive model. The results obtained from the standard datasets are not full satisfactory reaching an accuracy of 19%. Instead, the dataset including the cost model information achieved a very good accuracy (3.2%). They obtain in average, on the analysed dataset, an accuracy degree even better then the analytic cost model. Furthermore, the predictive model built on this dataset are capable to spread the error, almost equally, between the excess/lack of parallelism dataset partitions. It is able to generalize the relation existing between tree samples and the targets even for the case where the analytic model cannot be used.

9.1 Future works

This thesis opens an interesting research area both for the high performance computing and machine learning community. There are many research opportunities offered by:

1. Extending the developed parallelizations to others architectures
   
   • A mix of the two parallel implementations may be implemented targeting a cluster of multicores workstations. Taking into account that the master-worker parallel implementation minimizes the information exchanged, a mixed approach can be used to send a subset of hyper-parameters to the workers. The worker can compute the task taking advantage on the macro-dataflow multicores implementation reusing, when it is possible, the data.
• GP-GPUs architectures may be taken into account to create a new parallelization method or to delegate some computational steps in the macro-data flow parallel implementation.

2. Refining the made predictive learning model and identifying new learning tasks

• Further investigations should be done on the proposed learning task. In particular, different dataset representations should be taken into account. One of them could be a dataset where the arity of the nodes in the tree representation is augmented in order to reduce the influence of the anti-markovianity factor.

• The developed learning tasks can be a useful benchmark for some innovative solution in the reservoir computing. Solutions facing the problem of mean absolute normalized error can be identified. At the moment, two different solutions have been identified:
  – Replicate the state of the training trees with the balancing technique, and
  – Develop a new readout component optimizing directly the interested error metric.

• Investigate how some additional information can help the predictive models. In particular, which kinds of information can be included for a learning task where the analytic model is not or partially known. Some preliminary tests can be conducted on a tasks for which an analytic model already exist including very naive information. For example, for the performance prediction learning task, the ideal completion time (even for the pipeline) information could be considered.

• New learning tasks can be explored on the skeleton trees. Currently, a very interesting field that can be investigated, using the learning machine techniques explained in the thesis, is the energy consumption problem.

3. Applying the predictive model to an advanced skeleton framework

• The developed predictive learning models can be integrated as internal knowledge in an advanced skeleton framework in order to verify how the predictive model behaves in a real scenario.
Appendix A

Generated DataSet

- Name: Nested skeleton completion time
- Data Type: Binary trees
- Nature of the Data: Achieved via profiling
- Task Type: Regression
- Target Transduction Type: Structured domain to element
- Brief Description: Each dataset represents a sampling of the optimization space when rewriting rules for skeleton application are considered. All the samples contained in each dataset refer to the same set of skeleton trees with a different representation. All the skeleton applications are labelled with their completion time perceived on a restrained Phi 5100 architecture. The executed skeleton application are implemented using the FastFlow framework. A full description of how the dataset has been made is available in Sect. 6.2, while the choices done for the dataset representation are available in Sect. 6.3.
- Dataset Dimension: The dataset is composed by 1340 samples. The trees available in the dataset are characterized by having a maximum height equal to 4 and an average nodes number of 8.406.

In the following, the meaning of the node representation in each dataset will be provided. Furthermore, an extract coming from all the generated datasets (the first 10 samples) will be provided to be able to compare between each others the dataset.
Original Dataset

\[
\text{NodeRepresentation} := \langle \text{NodeId} \rangle \langle \text{Son1Id} \rangle \langle \text{Son2Id} \rangle \langle \text{ParentId} \rangle \text{ LABELS} \\
\text{LABELS} := \langle \text{1of3Representation} \rangle \langle \text{ServiceTime} \rangle \langle \text{ParallelDegree} \rangle
\]

TreeNum 1340
MaxArity 2
LabelDim 5

Name 0000
Target 11003.6
TreeDim 11
0 -1 -1 1 0 0 1 15 1
1 0 -1 8 0 1 0 0 3
2 -1 -1 3 0 0 1 3 1
3 2 -1 8 0 1 0 0 14
4 -1 -1 5 0 0 1 12 1
5 4 -1 9 0 1 0 0 15
6 -1 -1 7 0 0 1 103 1
7 6 -1 9 0 1 0 0 8
8 1 3 10 1 0 0 0 0
9 5 7 10 1 0 0 0 0
10 8 9 -1 1 0 0 0 0

Name 0001
Target 10727
TreeDim 8
0 -1 -1 1 0 0 1 4 1
1 0 -1 6 0 1 0 0 7
2 -1 -1 3 0 0 1 24 1
3 2 -1 6 0 1 0 0 9
4 -1 -1 5 0 0 1 93 1
5 4 -1 7 0 1 0 0 7
6 1 3 7 1 0 0 0 0
7 6 5 -1 1 0 0 0 0

Name 0002
Target 2723.72
TreeDim 11
0 -1 -1 1 0 0 1 14 1
1 0 -1 8 0 1 0 0 12
2 -1 -1 3 0 0 1 14 1
<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>9</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name 0003  
Target 28092.9  
TreeDim 11  

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>9</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name 0004  
Target 39321.5  
TreeDim 5  

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name 0005  
Target 11708.1  
TreeDim 7  

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

90
Name 0006
Target 10995.5
TreeDim 10
0 -1 1 0 0 1 3 1
1 0 -1 7 0 1 0 0 13
2 -1 1 3 0 0 1 63 1
3 2 -1 7 0 1 0 0 15
4 -1 1 5 0 0 1 122 1
5 4 -1 8 0 1 0 0 9
6 -1 -1 8 0 0 1 4 1
7 1 3 9 1 0 0 0 0
8 5 6 9 1 0 0 0 0
9 7 8 -1 1 0 0 0 0

Name 0007
Target 4665.94
TreeDim 8
0 -1 1 4 0 0 1 69 1
1 -1 -1 4 0 0 1 17 1
2 -1 -1 5 0 0 1 28 1
3 -1 -1 5 0 0 1 58 1
4 0 1 6 1 0 0 0 0
5 2 3 6 1 0 0 0 0
6 4 5 7 1 0 0 0 0
7 6 -1 -1 0 1 0 0 14

Name 0008
Target 104138
TreeDim 10
0 -1 1 7 0 0 1 130 1
1 -1 -1 2 0 0 1 37 1
2 1 -1 7 0 1 0 0 9
3 -1 1 4 0 0 1 25 1
4 3 -1 8 0 1 0 0 10
5 -1 -1 6 0 0 1 41 1
6 5 -1 8 0 1 0 0 10
7 0 2 9 1 0 0 0 0
8 4 6 9 1 0 0 0 0
9 7 8 -1 1 0 0 0 0

Name 0009
Target 2378.93
TreeNum 1340  
MaxArity 2  
LabelDim 5  

Name 0000  
Target 11003.6  
TreeDim 11  
0 -1 -1 1 0 0 1 1 0 0 6 7568 0.067568 0.062500  
1 0 -1 8 0 1 0 0 12  
2 -1 -1 3 0 0 1 1 4 1  
3 2 -1 8 0 1 0 0 14  
4 -1 -1 5 0 0 1 1 4 1  
5 4 -1 9 0 1 0 0 11  
6 -1 -1 7 0 0 1 1 4 1  
7 6 -1 9 0 1 0 0 6  
8 1 3 10 1 0 0 0 0  
9 5 7 10 1 0 0 0 0  
10 8 9 -1 1 0 0 0 0  

Name 0001  
Target 10727
<table>
<thead>
<tr>
<th>Name</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>2723.72</td>
</tr>
<tr>
<td>0003</td>
<td>28092.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TreeDim</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1</td>
<td>1  0 0 1 0.018018 0.062500</td>
</tr>
<tr>
<td>1 0 -1 6</td>
<td>0 1 0 0.000000 0.437500</td>
</tr>
<tr>
<td>2 -1 -1 3</td>
<td>0 0 1 0.108108 0.062500</td>
</tr>
<tr>
<td>3 2 -1 6</td>
<td>0 1 0 0.000000 0.562500</td>
</tr>
<tr>
<td>4 -1 -1 5</td>
<td>0 0 1 0.418919 0.062500</td>
</tr>
<tr>
<td>5 4 -1 7</td>
<td>0 1 0 0.000000 0.437500</td>
</tr>
<tr>
<td>6 1 3 7</td>
<td>1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td>7 6 5 -1 1</td>
<td>0 0 0.000000 0.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TreeDim</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1</td>
<td>1  0 0 1 0.063063 0.062500</td>
</tr>
<tr>
<td>1 0 -1 8</td>
<td>0 1 0 0.000000 0.750000</td>
</tr>
<tr>
<td>2 -1 -1 3</td>
<td>0 0 1 0.063063 0.062500</td>
</tr>
<tr>
<td>3 2 -1 8</td>
<td>0 1 0 0.000000 0.500000</td>
</tr>
<tr>
<td>4 -1 -1 5</td>
<td>0 0 1 0.063063 0.062500</td>
</tr>
<tr>
<td>5 4 -1 9</td>
<td>0 1 0 0.000000 0.875000</td>
</tr>
<tr>
<td>6 -1 -1 7</td>
<td>0 0 1 0.063063 0.062500</td>
</tr>
<tr>
<td>7 6 -1 9</td>
<td>0 1 0 0.000000 0.312500</td>
</tr>
<tr>
<td>8 1 3 10</td>
<td>1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td>9 5 7 10</td>
<td>1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td>10 8 9 -1 1</td>
<td>0 0 0.000000 0.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TreeDim</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1</td>
<td>1  0 0 1 0.130631 0.062500</td>
</tr>
<tr>
<td>1 0 -1 8</td>
<td>0 1 0 0.000000 0.125000</td>
</tr>
<tr>
<td>2 -1 -1 3</td>
<td>0 0 1 0.045045 0.062500</td>
</tr>
<tr>
<td>3 2 -1 8</td>
<td>0 1 0 0.000000 0.250000</td>
</tr>
<tr>
<td>4 -1 -1 5</td>
<td>0 0 1 0.040541 0.062500</td>
</tr>
<tr>
<td>5 4 -1 9</td>
<td>0 1 0 0.000000 0.750000</td>
</tr>
<tr>
<td>6 -1 -1 7</td>
<td>0 0 1 0.472973 0.062500</td>
</tr>
<tr>
<td>7 6 -1 9</td>
<td>0 1 0 0.000000 0.187500</td>
</tr>
<tr>
<td>8 1 3 10</td>
<td>1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td>9 5 7 10</td>
<td>1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td>10 8 9 -1 1</td>
<td>0 0 0.000000 0.000000</td>
</tr>
<tr>
<td>Name</td>
<td>Target</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>0004</td>
<td>39321.5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>0 1</td>
</tr>
<tr>
<td>4</td>
<td>3 2</td>
</tr>
<tr>
<td>Name</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>1 3</td>
</tr>
<tr>
<td>6</td>
<td>5 4</td>
</tr>
<tr>
<td>Name</td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>1 3</td>
</tr>
<tr>
<td>8</td>
<td>5 6</td>
</tr>
<tr>
<td>9</td>
<td>7 8</td>
</tr>
<tr>
<td>Name</td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>
3 -1 -1 5 0 0 1 0.261261 0.062500
4 0 1 6 1 0 0 0.000000 0.000000
5 2 3 6 1 0 0 0.000000 0.000000
6 4 5 7 1 0 0 0.000000 0.000000
7 6 -1 -1 0 1 0 0.000000 0.875000

Name 0008
Target 104138
TreeDim 10
  0 -1 -1 7 0 0 1 0.585586 0.062500
  1 -1 -1 2 0 0 1 0.166667 0.062500
  2 1 -1 7 0 1 0 0.000000 0.562500
  3 -1 -1 4 0 0 1 0.112613 0.062500
  4 3 -1 8 0 1 0 0.000000 0.625000
  5 -1 -1 6 0 0 1 0.184685 0.062500
  6 5 -1 8 0 1 0 0.000000 0.625000
  7 0 2 9 1 0 0 0.000000 0.000000
  8 4 6 9 1 0 0 0.000000 0.000000
  9 7 8 -1 1 0 0 0.000000 0.000000

Name 0009
Target 2378.93
TreeDim 11
  0 -1 -1 1 0 0 1 0.063063 0.062500
  1 0 -1 8 0 1 0 0.000000 0.750000
  2 -1 -1 3 0 0 1 0.063063 0.062500
  3 2 -1 8 0 1 0 0.000000 0.875000
  4 -1 -1 5 0 0 1 0.063063 0.062500
  5 4 -1 9 0 1 0 0.000000 0.687500
  6 -1 -1 7 0 0 1 0.063063 0.062500
  7 6 -1 9 0 1 0 0.000000 0.375000
  8 1 3 10 1 0 0 0.000000 0.000000
  9 5 7 10 1 0 0 0.000000 0.000000
 10 8 9 -1 1 0 0 0.000000 0.000000
Replicated Dataset

NodeRepresentation := <NodeId> <Son1Id> <Son2Id> <ParentId> LABELS
LABELS := <1of3Representation> <ServiceTime> <ReplicationDegree>

TreeNum 1340
MaxArity 2
LabelDim 5

Name 0000
Target 11003.6
TreeDim 11
0 -1 -1 1 0 0 1 15.000 3
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 3.000 14
3 2 -1 4 0 1 0 0 1
4 1 3 10 1 0 0 0 1
5 -1 -1 6 0 0 1 12.000 15
6 5 -1 9 0 1 0 0 1
7 -1 -1 8 0 0 1 103.000 8
8 7 -1 9 0 1 0 0 1
9 6 8 10 1 0 0 0 1
10 4 9 -1 1 0 0 0 1

Name 0001
Target 10727
TreeDim 8
0 -1 -1 1 0 0 1 4.000 7
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 24.000 9
3 2 -1 4 0 1 0 0 1
4 1 3 7 1 0 0 0 1
5 -1 -1 6 0 0 1 93.000 7
6 5 -1 7 0 1 0 0 1
7 4 6 -1 1 0 0 0 1

Name 0002
Target 2723.72
TreeDim 11
0 -1 -1 1 0 0 1 14.000 12
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 14.000 8
Name 0003
Target 28092.9
TreeDim 11
0 -1 -1 1 0 0 1 29.000 2
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 10.000 4
3 2 -1 4 0 1 0 0 1
4 1 3 10 1 0 0 0 1
5 -1 -1 6 0 0 1 9.000 12
6 5 -1 9 0 1 0 0 1
7 -1 -1 8 0 0 1 105.000 3
8 7 -1 9 0 1 0 0 1
9 6 8 10 1 0 0 0 1
10 4 9 -1 1 0 0 0 1

Name 0004
Target 39321.5
TreeDim 5
0 -1 -1 2 0 0 1 49.000 1
1 -1 -1 2 0 0 1 49.000 1
2 0 1 4 1 0 0 0 1
3 -1 -1 4 0 0 1 49.000 1
4 2 3 -1 1 0 0 0 1

Name 0005
Target 11708.1
TreeDim 7
0 -1 -1 1 0 0 1 146.000 10
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 6.000 3
3 2 -1 4 0 1 0 0 1
4 1 3 6 1 0 0 0 1
5 -1 -1 6 0 0 1 1.000 1
6 4 5 -1 1 0 0 0 1
Name 0006
Target 10995.5
TreeDim 10
0 -1 -1 1 0 0 1 3.000 13
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 63.000 15
3 2 -1 4 0 1 0 0 1
4 1 3 9 1 0 0 0 1
5 -1 -1 6 0 0 1 122.000 9
6 5 -1 8 0 1 0 0 1
7 -1 -1 8 0 0 1 4.000 1
8 6 7 9 1 0 0 0 1
9 4 8 -1 1 0 0 0 1

Name 0007
Target 4665.94
TreeDim 8
0 -1 -1 2 0 0 1 69.000 14
1 -1 -1 2 0 0 1 17.000 14
2 0 1 6 1 0 0 0 14
3 -1 -1 5 0 0 1 28.000 14
4 -1 -1 5 0 0 1 58.000 14
5 3 4 6 1 0 0 0 14
6 2 5 7 1 0 0 0 14
7 6 -1 -1 0 1 0 0 1

Name 0008
Target 104138
TreeDim 10
0 -1 -1 3 0 0 1 130.000 1
1 -1 -1 2 0 0 1 37.000 9
2 1 -1 3 0 1 0 0 1
3 0 2 9 1 0 0 0 1
4 -1 -1 5 0 0 1 25.000 10
5 4 -1 8 0 1 0 0 1
6 -1 -1 7 0 0 1 41.000 10
7 6 -1 8 0 1 0 0 1
8 5 7 9 1 0 0 0 1
9 3 8 -1 1 0 0 0 1

Name 0009
Target 2378.93
Replicated Dataset Normalized

NodeRepresentation := <NodeId> <Son1Id> <Son2Id> <ParentId> LABELS
LABELS := <1of3Representation> <ServiceTime (Normalized)>
           <ReplicationDegree (Normalized)>

TreeDim 11
0 -1 -1 1 0 0 1 14.000 12
1 0 -1 4 0 1 0 0 1
2 -1 -1 3 0 0 1 14.000 14
3 2 -1 4 0 1 0 0 1
4 1 3 10 1 0 0 0 1
5 -1 -1 6 0 0 1 14.000 11
6 5 -1 9 0 1 0 0 1
7 -1 -1 8 0 0 1 14.000 6
8 7 -1 9 0 1 0 0 1
9 6 8 10 1 0 0 0 1
10 4 9 -1 1 0 0 0 1

TreeNum 1340
MaxArity 2
LabelDim 5

Name 0000
Target 11003.6
TreeDim 11
0 -1 -1 1 0 0 1 0.067568 0.133333
1 0 -1 4 0 1 0 0.000000 0.000000
2 -1 -1 3 0 0 1 0.013514 0.866667
3 2 -1 4 0 1 0 0.000000 0.000000
4 1 3 10 1 0 0 0.000000 0.000000
5 -1 -1 6 0 0 1 0.054054 0.933333
6 5 -1 9 0 1 0 0.000000 0.000000
7 -1 -1 8 0 0 1 0.463964 0.466667
8 7 -1 9 0 1 0 0.000000 0.000000
9 6 8 10 1 0 0 0.000000 0.000000
10 4 9 -1 1 0 0 0.000000 0.000000

Name 0001
Target 10727
TreeDim 8
0 -1 -1 1 0 0 1 0.018018 0.400000
1 0 -1 4 0 1 0 0.000000 0.000000
2 -1 -1 3 0 0 1 0.108108 0.533333
3 2 -1 4 0 1 0 0.000000 0.000000
4 1 3 7 1 0 0 0.000000 0.000000
5 -1 -1 6 0 0 1 0.418919 0.400000
6 5 -1 7 0 1 0 0.000000 0.000000
7 4 6 -1 1 0 0 0.000000 0.000000

Name 0002
Target 2723.72
TreeDim 11
0 -1 -1 1 0 0 1 0.063063 0.733333
1 0 -1 4 0 1 0 0.000000 0.000000
2 -1 -1 3 0 0 1 0.063063 0.466667
3 2 -1 4 0 1 0 0.000000 0.000000
4 1 3 10 1 0 0 0.000000 0.000000
5 -1 -1 6 0 0 1 0.063063 0.866667
6 5 -1 9 0 1 0 0.000000 0.000000
7 -1 -1 8 0 0 1 0.063063 0.266667
8 7 -1 9 0 1 0 0.000000 0.000000
9 6 8 10 1 0 0 0.000000 0.000000
10 4 9 -1 1 0 0 0.000000 0.000000

Name 0003
Target 28092.9
TreeDim 11
0 -1 -1 1 0 0 1 0.130631 0.066667
1 0 -1 4 0 1 0 0.000000 0.000000
2 -1 -1 3 0 0 1 0.045045 0.200000
3 2 -1 4 0 1 0 0.000000 0.000000
4 1 3 10 1 0 0 0.000000 0.000000
5 -1 -1 6 0 0 1 0.040541 0.733333
6 5 -1 9 0 1 0 0.000000 0.000000
7 -1 -1 8 0 0 1 0.472973 0.133333
8 7 -1 9 0 1 0 0.000000 0.000000
9 6 8 10 1 0 0 0.000000 0.000000
10 4 9 -1 1 0 0 0.000000 0.000000
Name 0004
Target 39321.5
TreeDim 5
0 -1 -1 2 0 0 1 0.220721 0.000000
1 -1 -1 2 0 0 1 0.220721 0.000000
2 0 1 4 1 0 0 0.000000 0.000000
3 -1 -1 4 0 0 1 0.220721 0.000000
4 2 3 -1 1 0 0 0.000000 0.000000

Name 0005
Target 11708.1
TreeDim 7
0 -1 -1 1 0 0 1 0.657658 0.600000
1 0 -1 4 0 1 0 0.000000 0.000000
2 -1 -1 3 0 0 1 0.027027 0.133333
3 2 -1 4 0 1 0 0.000000 0.000000
4 1 3 6 1 0 0 0.000000 0.000000
5 -1 -1 6 0 0 1 0.004505 0.000000
6 4 5 -1 1 0 0 0.000000 0.000000

Name 0006
Target 10995.5
TreeDim 10
0 -1 -1 1 0 0 1 0.013514 0.800000
1 0 -1 4 0 1 0 0.000000 0.000000
2 -1 -1 3 0 0 1 0.283784 0.933333
3 2 -1 4 0 1 0 0.000000 0.000000
4 1 3 9 1 0 0 0.000000 0.000000
5 -1 -1 6 0 0 1 0.549550 0.533333
6 5 -1 8 0 1 0 0.000000 0.000000
7 -1 -1 8 0 0 1 0.018018 0.000000
8 6 7 9 1 0 0 0.000000 0.000000
9 4 8 -1 1 0 0 0.000000 0.000000

Name 0007
Target 4665.94
TreeDim 8
0 -1 -1 2 0 0 1 0.310811 0.866667
1 -1 -1 2 0 0 1 0.076577 0.866667
2 0 1 6 1 0 0 0.000000 0.866667
<table>
<thead>
<tr>
<th>Name 0008</th>
<th>Target 104138</th>
<th>TreeDim 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 -1 -1 3 0 0 1 0.585586 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 -1 -1 2 0 0 1 0.166667 0.533333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 1 -1 3 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 0 2 9 1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 -1 -1 5 0 0 1 0.112613 0.600000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 4 -1 8 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 -1 -1 7 0 0 1 0.184685 0.600000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 6 -1 8 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 5 7 9 1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 3 8 -1 1 0 0 0.000000 0.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name 0009</th>
<th>Target 2378.93</th>
<th>TreeDim 11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 -1 -1 1 0 0 1 0.063063 0.733333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 -1 4 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 -1 -1 3 0 0 1 0.063063 0.866667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 2 -1 4 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 1 3 10 1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 -1 -1 6 0 0 1 0.063063 0.666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 5 -1 9 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 -1 -1 8 0 0 1 0.063063 0.333333</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 7 -1 9 0 1 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 6 8 10 1 0 0 0.000000 0.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 4 9 -1 1 0 0 0.000000 0.000000</td>
</tr>
</tbody>
</table>
Cost Model Dataset

Node Representation := <NodeId> <Son1Id> <Son2Id> <ParentId> LABELS
LABELS := <lof3Representation> <ServiceTime> <ParallelDegree> <EstimatedServiceTime>

TreeNum 1340
MaxArity 2
LabelDim 6

Name 0000
Target 11003.6
TreeDim 11
0 -1 -1 1 0 0 1 15.000 1 15.000000
1 0 -1 4 0 1 0 0 3 5.000000
2 -1 -1 3 0 0 1 3.000000 1 3.000000
3 2 -1 4 0 1 0 0 14 0.214286
4 1 3 10 1 0 0 0 5.000000
5 -1 -1 6 0 0 1 12.0000 1 12.000000
6 5 -1 9 0 1 0 0 15 0.800000
7 -1 -1 8 0 0 1 103.0000 1 103.000000
8 7 -1 9 0 1 0 0 8 12.875000
9 6 8 10 1 0 0 0 12.875000
10 4 9 -1 1 0 0 0 0 12.875000

Name 0001
Target 10727
TreeDim 8
0 -1 -1 1 0 0 1 4.000 1 4.000000
1 0 -1 4 0 1 0 0 7 0.571429
2 -1 -1 3 0 0 1 24.0000 1 24.000000
3 2 -1 4 0 1 0 0 9 2.666667
4 1 3 7 1 0 0 0 2.666667
5 -1 -1 6 0 0 1 93.0000 1 93.000000
6 5 -1 7 0 1 0 0 7 13.285714
7 4 6 -1 1 0 0 0 0 13.285714

Name 0002
Target 2723.72
TreeDim 11
0 -1 -1 1 0 0 1 14.000 1 14.000000
1 0 -1 4 0 1 0 0 12 1.166667
<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.000</td>
<td>1</td>
<td>14.000000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1.750000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.750000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>1.000000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2.800000</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.800000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>9</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.800000</td>
<td></td>
</tr>
</tbody>
</table>

Name 0003
Target 28092.9
TreeDim 11

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>29.000</td>
<td>1</td>
<td>29.000000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>14.500000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10.000</td>
<td>1</td>
<td>10.000000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2.500000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14.500000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9.000</td>
<td>1</td>
<td>9.000000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0.750000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>105.000</td>
<td>1</td>
<td>105.000000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>35.000000</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>35.000000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>9</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>35.000000</td>
<td></td>
</tr>
</tbody>
</table>

Name 0004
Target 39321.5
TreeDim 5

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000</td>
<td>1</td>
<td>49.000000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000</td>
<td>1</td>
<td>49.000000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>49.000000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000</td>
<td>1</td>
<td>49.000000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>49.000000</td>
<td></td>
</tr>
</tbody>
</table>

Name 0005
Target 11708.1
TreeDim 7

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>146.000</td>
<td>1</td>
<td>146.000000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>14.600000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6.000</td>
<td>1</td>
<td>6.000000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2.000000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14.600000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.000</td>
<td>1</td>
<td>1.000000</td>
<td></td>
</tr>
</tbody>
</table>
6 4 5 −1 1 0 0 0 0 14.600000
Name 0006
Target 10995.5
TreeDim 10
0 −1 −1 1 0 0 1 3.000 1 3.000000
1 0 −1 4 0 1 0 0 13 0.230769
2 −1 −1 3 0 0 1 63.000 1 63.000000
3 2 −1 4 0 1 0 0 15 4.200000
4 1 3 9 1 0 0 0 4.200000
5 −1 −1 6 0 0 1 122.000 1 122.000000
6 5 −1 8 0 1 0 0 9 13.555556
7 −1 −1 8 0 0 1 4.000 1 4.000000
8 6 7 9 1 0 0 0 13.555556
9 4 8 −1 1 0 0 0 0 13.555556

Name 0007
Target 4665.94
TreeDim 8
0 −1 −1 2 0 0 1 69.000 1 69.000000
1 −1 −1 2 0 0 1 17.000 1 17.000000
2 0 1 6 1 0 0 0 69.000000
3 −1 −1 5 0 0 1 28.000 1 28.000000
4 −1 −1 5 0 0 1 58.000 1 58.000000
5 3 4 6 1 0 0 0 58.000000
6 2 5 7 1 0 0 0 69.000000
7 6 −1 −1 0 1 0 0 14 4.928571

Name 0008
Target 104138
TreeDim 10
0 −1 −1 3 0 0 1 130.000 1 130.000000
1 −1 −1 2 0 0 1 37.000 1 37.000000
2 1 −1 3 0 1 0 0 9 4.111111
3 0 2 9 1 0 0 0 130.000000
4 −1 −1 5 0 0 1 25.000 1 25.000000
5 4 −1 8 0 1 0 0 10 2.500000
6 −1 −1 7 0 0 1 41.000 1 41.000000
7 6 −1 8 0 1 0 0 10 4.100000
8 5 7 9 1 0 0 0 4.100000
9 3 8 −1 1 0 0 0 0 130.000000

Name 0009
Target 2378.93
TreeDim 11
0 −1 −1 1  0  0 1 14.000 1 14.000000
1 0 −1 4  0  1 0 12 1.166667
2 −1 −1 3  0  0 1 14.000 1 14.000000
3 2 −1 4  0  1  0 14 1.000000
4 1 3 10 1  0  0 0 0 1.166667
5 −1 −1 6  0  0 1 14.000 1 14.000000
6 5 −1 9  0  1  0 11 1.272727
7 −1 −1 8  0  0 1 14.000 1 14.000000
8 7 −1 9  0  1  0  6 2.333333
9 6 8 10 1  0  0 0 0 2.333333
10 4 9 −1 1  0  0 0 0 2.333333

Cost Model Dataset Normalized

```
NodeRepresentation := <NodeId> <Son1Id> <Son2Id> <ParentId> LABELS
LABELS := <1of3Representation> <ServiceTime(Normalized)>
           <ParallelDegree(Normalized)>
           <EstimatedServiceTime(Normalized)>
```

TreeNum 1340
MaxArity 2
LabelDim 6

Name 0000
Target 11003.6
TreeDim 11
0 −1 −1 1  0  0 1 0.067568 0.062500 0.067568
1 0 −1 4  0  1 0 0.000000 0.187500 0.022523
2 −1 −1 3  0  0 1 0.013514 0.062500 0.013514
3 2 −1 4  0  1 0 0.000000 0.875000 0.000000
4 1 3 10 1  0  0 0.000000 0.000000 0.022523
5 −1 −1 6  0  0 1 0.054054 0.062500 0.054054
6 5 −1 9  0  1 0 0.000000 0.937500 0.000000
7 −1 −1 8  0  0 1 0.463964 0.062500 0.463964
8 7 −1 9  0  1 0 0.000000 0.500000 0.054054
9 6 8 10 1  0  0 0.000000 0.000000 0.054054
10 4 9 −1 1  0  0 0.000000 0.000000 0.054054

106
<table>
<thead>
<tr>
<th>Name 0001</th>
<th>Target 10727</th>
<th>TreeDim 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 0 0 1 0.018018 0.062500 0.018018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0.000000 0.437500 0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 0.108108 0.062500 0.108108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0.000000 0.562500 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 3 7 1 0 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 0.418919 0.062500 0.418919</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 5 -1 7 0 1 0 0.000000 0.437500 0.058559</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 4 6 -1 1 0 0 0.000000 0.000000 0.058559</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name 0002</th>
<th>Target 2723.72</th>
<th>TreeDim 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 0 0 1 0.063063 0.062500 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0.000000 0.750000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 0.063063 0.062500 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0.000000 0.500000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 3 10 1 0 0 0.000000 0.000000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 0.063063 0.062500 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 5 -1 9 0 1 0 0.000000 0.875000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 -1 -1 8 0 0 1 0.063063 0.062500 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 7 -1 9 0 1 0 0.000000 0.312500 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 6 8 10 1 0 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 4 9 -1 1 0 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name 0003</th>
<th>Target 28092.9</th>
<th>TreeDim 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 0 0 1 0.130631 0.062500 0.130631</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0.000000 0.125000 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 0.045045 0.062500 0.045045</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0.000000 0.250000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 3 10 1 0 0 0.000000 0.000000 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 0.040541 0.062500 0.040541</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 5 -1 9 0 1 0 0.000000 0.750000 0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 -1 -1 8 0 0 1 0.472973 0.062500 0.472973</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 7 -1 9 0 1 0 0.000000 0.187500 0.157658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 6 8 10 1 0 0 0.000000 0.000000 0.157658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 4 9 -1 1 0 0 0.000000 0.000000 0.157658</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Name 0004  
Target 39321.5  
TreeDim 5  
0 -1 -1 2 0 0 1 0.220721 0.062500 0.220721  
1 -1 -1 2 0 0 1 0.220721 0.062500 0.220721  
2 0 1 4 1 0 0 0.000000 0.000000 0.220721  
3 -1 -1 4 0 0 1 0.220721 0.062500 0.220721  
4 2 3 -1 1 0 0 0.000000 0.000000 0.220721

Name 0005  
Target 11708.1  
TreeDim 7  
0 -1 -1 1 0 0 1 0.657658 0.062500 0.657658  
1 0 -1 4 0 1 0 0.000000 0.625000 0.063063  
2 -1 -1 3 0 0 1 0.027027 0.062500 0.027027  
3 2 -1 4 0 1 0 0.000000 0.187500 0.009009  
4 1 3 6 1 0 0 0.000000 0.000000 0.063063  
5 -1 -1 6 0 0 1 0.004505 0.062500 0.004505  
6 4 5 -1 1 0 0 0.000000 0.000000 0.063063

Name 0006  
Target 10995.5  
TreeDim 10  
0 -1 -1 1 0 0 1 0.013514 0.062500 0.013514  
1 0 -1 4 0 1 0 0.000000 0.812500 0.000000  
2 -1 -1 3 0 0 1 0.283784 0.062500 0.283784  
3 2 -1 4 0 1 0 0.000000 0.937500 0.018018  
4 1 3 9 1 0 0 0.000000 0.000000 0.018018  
5 -1 -1 6 0 0 1 0.549550 0.062500 0.549550  
6 5 -1 8 0 1 0 0.000000 0.562500 0.058559  
7 -1 -1 8 0 0 1 0.018018 0.062500 0.018018  
8 6 7 9 1 0 0 0.000000 0.000000 0.058559  
9 4 8 -1 1 0 0 0.000000 0.000000 0.058559

Name 0007  
Target 4665.94  
TreeDim 8  
0 -1 -1 2 0 0 1 0.310811 0.062500 0.310811
<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Name 0008
Target 104138
TreeDim 10

0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0

Name 0009
Target 2378.93
TreeDim 11

0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0

109
Synergetic Dataset

NodeRepresentation := <NodeId> <Son1Id> <Son2Id> <ParentId> LABELS
LABELS := <lof3Representation> <ServiceTime> <ReplicationDegree>
  <EstimatedServiceTime>

<table>
<thead>
<tr>
<th>TreeNum</th>
<th>MaxArity</th>
<th>LabelDim</th>
<th>Name</th>
<th>Target</th>
<th>TreeDim</th>
</tr>
</thead>
<tbody>
<tr>
<td>1340</td>
<td>2</td>
<td>6</td>
<td>0000</td>
<td>11003.6</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 -1 -1</td>
<td>1 0 0 1 15.000 3 15.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 0 -1</td>
<td>4 0 1 0 0 1 5.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 -1 -1</td>
<td>3 0 0 1 3.000 14 3.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 2 -1</td>
<td>4 0 1 0 0 1 0.214286</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 1 3</td>
<td>10 1 0 0 1 5.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 -1 -1</td>
<td>6 0 0 1 12.000 15 12.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 5 -1</td>
<td>9 0 1 0 0 1 0.800000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 -1 -1</td>
<td>8 0 0 1 103.000 8 103.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 7 -1</td>
<td>9 0 1 0 0 1 12.875000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9 6 8</td>
<td>10 1 0 0 1 12.875000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 4 9</td>
<td>-1 1 0 0 0 1 12.875000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0001</td>
<td>10727</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 -1 -1</td>
<td>1 0 0 1 4.000 7 4.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 0 -1</td>
<td>4 0 1 0 0 1 0.571429</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 -1 -1</td>
<td>3 0 0 1 24.000 9 24.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 2 -1</td>
<td>4 0 1 0 0 1 2.666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 1 3</td>
<td>7 1 0 0 0 1 2.666667</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 -1 -1</td>
<td>6 0 0 1 93.000 7 93.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6 5 -1</td>
<td>7 0 1 0 0 1 13.285714</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 4 6</td>
<td>-1 1 0 0 0 1 13.285714</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0002</td>
<td>27.23</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 -1 -1</td>
<td>1 0 0 1 14.000 12 14.000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 0 -1</td>
<td>4 0 1 0 0 1 1.166667</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>9</td>
<td>-1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Name 0003  
Target 28092.9  
TreeDim 11

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>29.000</td>
<td>2</td>
<td>29.000000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.500000</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10.000</td>
<td>4</td>
<td>10.000000</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2.500000</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.500000</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9.000</td>
<td>12</td>
<td>9.000000</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.750000</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>105.000</td>
<td>3</td>
<td>105.000000</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>35.000000</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>35.000000</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>9</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>35.000000</td>
</tr>
</tbody>
</table>

Name 0004  
Target 39321.5  
TreeDim 5

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000</td>
<td>1</td>
<td>49.000000</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000000</td>
<td>1</td>
<td>49.000000</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000000</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000000</td>
<td>1</td>
<td>49.000000</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>49.000000</td>
</tr>
</tbody>
</table>

Name 0005  
Target 11708.1  
TreeDim 7

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>146.000</td>
<td>10</td>
<td>146.000000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.600000</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6.000</td>
<td>3</td>
<td>6.000000</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2.000000</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>14.600000</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.000</td>
<td>1</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

Name 0003
Target 28092.9
TreeDim 11

Name 0004
Target 39321.5
TreeDim 5

Name 0005
Target 11708.1
TreeDim 7

111
Name 0006
Target 10995.5
TreeDim 10
0 -1 -1 1 0 0 1 3.000 13 3.000000  
1 0 -1 4 0 1 0 0 1 0.230769  
2 -1 -1 3 0 0 1 63.000 15 63.000000  
3 2 -1 4 0 1 0 0 1 4.200000  
4 1 3 9 1 0 0 0 1 4.200000  
5 -1 -1 6 0 0 1 122.000 9 122.000000  
6 5 -1 8 0 1 0 0 1 13.555556  
7 -1 -1 8 0 0 1 4.000 1 4.000000  
8 6 7 9 1 0 0 0 1 13.555556  
9 4 8 -1 1 0 0 0 1 13.555556

Name 0007
Target 4665.94
TreeDim 8
0 -1 -1 2 0 0 1 69.000 14 69.000000  
1 -1 -1 2 0 0 1 17.000 14 17.000000  
2 0 1 6 1 0 0 0 14 69.000000  
3 -1 -1 5 0 0 1 28.000 14 28.000000  
4 -1 -1 5 0 0 1 58.000 14 58.000000  
5 3 4 6 1 0 0 0 14 58.000000  
6 2 5 7 1 0 0 0 14 69.000000  
7 6 -1 -1 0 1 0 0 1 4.928571

Name 0008
Target 104138
TreeDim 10
0 -1 -1 3 0 0 1 130.000 1 130.000000  
1 -1 -1 2 0 0 1 37.000 9 37.000000  
2 1 -1 3 0 1 0 0 1 4.111111  
3 0 2 9 1 0 0 0 1 130.000000  
4 -1 -1 5 0 0 1 25.000 10 25.000000  
5 4 -1 8 0 1 0 0 1 2.500000  
6 -1 -1 7 0 0 1 41.000 10 41.000000  
7 6 -1 8 0 1 0 0 1 4.100000  
8 5 7 9 1 0 0 0 1 4.100000  
9 3 8 -1 1 0 0 0 1 130.000000

Name 0009
<table>
<thead>
<tr>
<th>Target 2378.93</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeDim 11</td>
</tr>
<tr>
<td>0 -1 -1 1 0 0 1 14.000 12 14.000000</td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0 1 1.166667</td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 14.000 14 14.000000</td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0 1 1.000000</td>
</tr>
<tr>
<td>4 1 3 10 1 0 0 0 1 1.166667</td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 14.000 11 14.000000</td>
</tr>
<tr>
<td>6 5 -1 9 0 1 0 0 1 1.272727</td>
</tr>
<tr>
<td>7 -1 -1 8 0 0 1 14.000 6 14.000000</td>
</tr>
<tr>
<td>8 7 -1 9 0 1 0 0 1 2.333333</td>
</tr>
<tr>
<td>9 6 8 10 1 0 0 0 1 2.333333</td>
</tr>
<tr>
<td>10 4 9 -1 1 0 0 0 1 2.333333</td>
</tr>
</tbody>
</table>

Synergetic Dataset Normalized

NodeRepresentation := <NodeId> <Son1Id> <Son2Id> <ParentId> LABELS
LABELS := <1of3Representation> <ServiceTime (Normalized)>
         <ReplicationDegree (Normalized)>
         <EstimatedServiceTime (Normalized)>

TreeNum 1340
MaxArity 2
LabelDim 6

Name 0000
Target 11003.6
TreeDim 11
0 -1 -1 1 0 0 1 0.067568 0.133333 0.067568
1 0 -1 4 0 1 0 0.000000 0.000000 0.022523
2 -1 -1 3 0 0 1 0.013514 0.866667 0.013514
3 2 -1 4 0 1 0 0.000000 0.000000 0.000000
4 1 3 10 1 0 0 0.000000 0.000000 0.022523
5 -1 -1 6 0 0 1 0.054054 0.933333 0.054054
6 5 -1 9 0 1 0 0.000000 0.000000 0.000000
7 -1 -1 8 0 0 1 0.463964 0.466667 0.463964
8 7 -1 9 0 1 0 0.000000 0.000000 0.054054
9 6 8 10 1 0 0 0.000000 0.000000 0.054054
10 4 9 -1 1 0 0 0.000000 0.000000 0.054054
<table>
<thead>
<tr>
<th>Name 0001</th>
<th>Target 10727</th>
<th>TreeDim 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 0 0 1 0.018018 0.400000 0.018018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0.000000 0.000000 0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 0.108108 0.533333 0.108108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 3 7 1 0 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 0.418919 0.400000 0.418919</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 5 -1 7 0 1 0 0.000000 0.000000 0.058559</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 4 6 -1 1 0 0 0.000000 0.000000 0.058559</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name 0002</th>
<th>Target 2723.72</th>
<th>TreeDim 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 0 0 1 0.063063 0.733333 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0.000000 0.000000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 0.063063 0.466667 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0.000000 0.000000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 3 10 1 0 0 0.000000 0.000000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 0.063063 0.866667 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 5 -1 9 0 1 0 0.000000 0.000000 0.004505</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 -1 -1 8 0 0 1 0.063063 0.266667 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 7 -1 9 0 1 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 6 8 10 1 0 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 4 9 -1 1 0 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name 0003</th>
<th>Target 28092.9</th>
<th>TreeDim 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 0 0 1 0.130631 0.666667 0.130631</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 -1 4 0 1 0 0.000000 0.000000 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 -1 -1 3 0 0 1 0.045045 0.200000 0.045045</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 -1 4 0 1 0 0.000000 0.000000 0.009009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 3 10 1 0 0 0.000000 0.000000 0.063063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 -1 -1 6 0 0 1 0.040541 0.733333 0.040541</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 5 -1 9 0 1 0 0.000000 0.000000 0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 -1 -1 8 0 0 1 0.472973 0.133333 0.472973</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 7 -1 9 0 1 0 0.000000 0.000000 0.157658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 6 8 10 1 0 0 0.000000 0.000000 0.157658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 4 9 -1 1 0 0 0.000000 0.000000 0.157658</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

114
Name 0004  
Target 39321.5  
TreeDim 5  
0 -1 -1 2 0 0 1 0.220721 0.000000 0.220721  
1 -1 -1 2 0 0 1 0.220721 0.000000 0.220721  
2 0 1 4 1 0 0 0.000000 0.000000 0.220721  
3 -1 -1 4 0 0 1 0.220721 0.000000 0.220721  
4 2 3 -1 1 0 0 0.000000 0.000000 0.220721  

Name 0005  
Target 11708.1  
TreeDim 7  
0 -1 -1 1 0 0 1 0.657658 0.600000 0.657658  
1 0 -1 4 0 1 0 0.000000 0.000000 0.063063  
2 -1 -1 3 0 0 1 0.027027 0.133333 0.027027  
3 2 -1 4 0 1 0 0.000000 0.000000 0.009009  
4 1 3 6 1 0 0 0.000000 0.000000 0.063063  
5 -1 -1 6 0 0 1 0.004505 0.000000 0.063063  
6 4 5 -1 1 0 0 0.000000 0.000000 0.063063  

Name 0006  
Target 10995.5  
TreeDim 10  
0 -1 -1 1 0 0 1 0.013514 0.800000 0.013514  
1 0 -1 4 0 1 0 0.000000 0.000000 0.000000  
2 -1 -1 3 0 0 1 0.283784 0.933333 0.283784  
3 2 -1 4 0 1 0 0.000000 0.000000 0.018018  
4 1 3 9 1 0 0 0.000000 0.000000 0.018018  
5 -1 -1 6 0 0 1 0.549550 0.533333 0.549550  
6 5 -1 8 0 1 0 0.000000 0.000000 0.058559  
7 -1 -1 8 0 0 1 0.018018 0.000000 0.018018  
8 6 7 9 1 0 0 0.000000 0.000000 0.058559  
9 4 8 -1 1 0 0 0.000000 0.000000 0.058559  

Name 0007  
Target 4665.94  
TreeDim 8  
0 -1 -1 2 0 0 1 0.310811 0.866667 0.310811  

115
Name 0008
Target 104138
TreeDim 10
0 -1 -1 3 0 0 1 0.585586 0.000000 0.585586
1 -1 -1 2 0 0 1 0.166667 0.533333 0.166667
2 1 -1 3 0 1 0 0.000000 0.000000 0.018018
3 0 2 9 1 0 0 0.000000 0.000000 0.585586
4 -1 -1 5 0 0 1 0.112613 0.600000 0.112613
5 4 -1 8 0 1 0 0.000000 0.000000 0.009009
6 -1 -1 7 0 0 1 0.184685 0.600000 0.184685
7 6 -1 8 0 1 0 0.000000 0.000000 0.018018
8 5 7 9 1 0 0 0.000000 0.000000 0.009009
9 3 8 -1 1 0 0 0.000000 0.000000 0.585586

Name 0009
Target 2378.93
TreeDim 11
0 -1 -1 1 0 0 1 0.063063 0.733333 0.063063
1 0 -1 4 0 1 0 0.000000 0.000000 0.004505
2 -1 -1 3 0 0 1 0.063063 0.866667 0.063063
3 2 -1 4 0 1 0 0.000000 0.000000 0.004505
4 1 3 10 1 0 0 0.000000 0.000000 0.004505
5 -1 -1 6 0 0 1 0.063063 0.666667 0.063063
6 5 -1 9 0 1 0 0.000000 0.000000 0.004505
7 -1 -1 8 0 0 1 0.063063 0.333333 0.063063
8 7 -1 9 0 1 0 0.000000 0.000000 0.009009
9 6 8 10 1 0 0 0.000000 0.000000 0.009009
10 4 9 -1 1 0 0 0.000000 0.000000 0.009009
Appendix B

Mean state mapping training

Datasets without analytic model information

<table>
<thead>
<tr>
<th>Metric</th>
<th>TR</th>
<th>VAL</th>
<th>TRVAL</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>2109.063</td>
<td>4605.383</td>
<td>2141.115</td>
<td>3947.252</td>
</tr>
<tr>
<td>MANE</td>
<td>0.108581</td>
<td>0.215500</td>
<td>0.109119</td>
<td>0.179524</td>
</tr>
</tbody>
</table>

Table B.1: The test error obtained by the TreeESN model trained using the “Replication Normalized” dataset in the validation and test phases when mean state mapping is used.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nr</th>
<th>Sigma</th>
<th>WScal</th>
<th>Wspar</th>
<th>WiScal</th>
<th>WiSpar</th>
<th>Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>3500</td>
<td>6.0</td>
<td>1.00</td>
<td>0.20</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Original Norm.</td>
<td>3500</td>
<td>9.0</td>
<td>1.00</td>
<td>0.05</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Replication</td>
<td>3500</td>
<td>6.0</td>
<td>1.00</td>
<td>0.05</td>
<td>1.0</td>
<td>1.0</td>
<td>0.10</td>
</tr>
<tr>
<td>Replication Norm.</td>
<td>3500</td>
<td>9.0</td>
<td>0.10</td>
<td>0.20</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table B.2: The TreeESN hyper-parameters values selected by the model selection for the baseline datasets when the mean state mapping is used.
### Datasets including cost model information

<table>
<thead>
<tr>
<th>Metric</th>
<th>TR</th>
<th>VAL</th>
<th>TRVAL</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>549.713</td>
<td>794.168</td>
<td>536.753</td>
<td>692.246</td>
</tr>
<tr>
<td>MANE</td>
<td>0.039357</td>
<td>0.051213</td>
<td>0.040759</td>
<td>0.048389</td>
</tr>
</tbody>
</table>

Table B.4: The test error obtained by the TreeESN model trained using the “Synergetic Normalised” dataset in the validation and test phases when mean state mapping is used.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nr</th>
<th>Sigma</th>
<th>WScal</th>
<th>WSpars</th>
<th>WiScal</th>
<th>WiSpars</th>
<th>Lambda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Model</td>
<td>3500</td>
<td>6.0</td>
<td>1.00</td>
<td>0.10</td>
<td>0.1</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Cost Model Norm.</td>
<td>3500</td>
<td>3.0</td>
<td>0.10</td>
<td>0.05</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Synergetic</td>
<td>3500</td>
<td>3.0</td>
<td>0.10</td>
<td>0.10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Synergetic Norm.</td>
<td>3500</td>
<td>3.0</td>
<td>0.10</td>
<td>0.10</td>
<td>1.0</td>
<td>1.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table B.5: The TreeESN hyper-parameters values selected by the model selection for the CM datasets when the mean state mapping is used.
Table B.6: The test errors, reported using the MAE and MANE metrics, exhibited by the CM datasets when the mean state mapping is used.

**Bipartite analysis**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MAE</th>
<th>MANE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>std</td>
</tr>
<tr>
<td>Cost Model</td>
<td>2811.075</td>
<td>482.988</td>
</tr>
<tr>
<td>Cost Model Norm.</td>
<td>702.509</td>
<td>16.715</td>
</tr>
<tr>
<td>Synergetic</td>
<td>2539.471</td>
<td>197.503</td>
</tr>
<tr>
<td>Synergetic Norm.</td>
<td>692.246</td>
<td>8.510</td>
</tr>
</tbody>
</table>

Table B.7: Bipartite test errors analysis of TreeESN model obtained by the model selection phase using the mean state mapping. The analysis is performed using the MAE and MANE metrics on the baseline and CM datasets.
Appendix C

Tesn library source code
#ifndef _RESERVOIR
#define _RESERVOIR
#include <stdio.h>
#include "tree_dataset.h"
#include "stateMapping.h"
#include "random.h"
#include "sparse.hpp"

/** Activation function type */
#define Activation_Function double

/** Class representing the Reservoir component of the TreeESN */
class Reservoir{
private:
  int Ni; // Number of labels contained in a node
  int Nr; // Number of neural unit contained in the reservoir
  int No; // Number of produced output
  int degree; // Maximum number of children for each node (out degree)
public:
  double (*Activation_Function)(double); // Activation function type

  // ENCODING NETWORK HYPER PARAMETERS
  double scalingWin; // Scaling factor for the input neural network
  double WinConnectivity; // Input to reservoir connectivity ratio
  double* Win; // Connectivity matrix \[Nr x Ni\] from input to reservoir units
  double connectivity; // Reservoir connectivity ratio
  double scalingW; // Scaling factor for the reservoir neural network
  double sigma; // Contractivity coefficient of the reservoir
  double* W; // Connectivity matrix \[Nr x Nr\] among reservoir units
};

/*  Author:  Daniele Virgilio    Credits: Computational Intelligence & Machine Learning Group − University of Pisa        Parallel Programming Models Group − University of Pisa */

21 apr 15 14:25 Pagina 1/4
reservoir.h
The reservoir destructor is called **\*/

*/ 
flag deallocating if the state data structure must be deallocated in case 
*
*
*/

Random matrix used for the connectivity matrix generation **\*/

*/ 
set the state number the state data structure is able to score **\*/

double *computeTreeState(TreeDataset* tds, int i);

double *computeTreeState(TreeDataset* tds, int i, Sparse<double> *sp);

/** Set the flag to destroy the state when the destructor is called **\*/

void stateDestroy();

/** Set the flag to preserve the connection matrix when 
the destructor is called. **\*/

void matrixUndestroy();

// State mapping points to an array
//

double* computeStateMapping(double* state,
int length,
double** stateMappingBuffer,
int offset);

todo
#ifndef

bool stateDestroyB;
/** Flag defining if the input/reservoir connectivity matrix must be dealloced in case the reservoir destructor is called */
bool matrixDestroyB;
#endif

21 apr 15 14:25 Pagina 4/4

reservoir.h
```cpp
/** Logistic function: hyperbolic tangent (logistic) function */

double logistic(double value) {
  return tanh(value);
}

Reservoir::Reservoir() {
  stateMapping[0] = StateMapping::superSourceStateMapping;
  stateMapping[1] = StateMapping::meanStateMapping;
  f = logistic;
  Nr = 0;
  W = NULL;
  Win = NULL;
  states = NULL;
  stateLen = 0;
  stateDestroyB = false;
  matrixDestroyB = true;
}

Reservoir::Reservoir(int Nr, int Ni, int degree) {
  stateMapping[0] = StateMapping::superSourceStateMapping;
  stateMapping[1] = StateMapping::meanStateMapping;
  f = logistic;
  this->Nr = Nr;
  this->Ni = Ni;
  this->degree = degree;
  W = NULL;
  Win = NULL;
}

return rank(value); // double logistic(double value) {
  double tangent = tan(value);
  return log(1 + tangent) / tanh(value); // Logistic function: hyperbolic tangent (logistic) function */
```

void Reservoir::initialize_Win(double scalingWin, double WinConnectivity)
{
    if (scalingWin == 0)
        return -1;
    if (WinConnectivity <= 0 || WinConnectivity > 1)
        return -2;
    if (Nr <= 0)
        return -3;
    this->scalingWin = scalingWin;
    this->WinConnectivity = WinConnectivity;
    double random_value;
    int ir, ic;
    Win = (double *) malloc((Nr * (Ni + 1)) * sizeof(double));
    memset(Win, 0, Nr * (Ni + 1) * sizeof(double));
    for (ir = 0; ir < Nr; ir++)
    {
        // also the input bias is set randomly
        for (ic = 0; ic <= Ni; ic++)
        {
            random_value = rv.random(0, 1);
            if (random_value <= WinConnectivity)
                Win[ir * (Ni + 1) + ic] = rv.random(-scalingWin, scalingWin);
        }
    }
    return 0;
}

int Reservoir::initialize_W(double scalingW, double connectivity, double sigma)
{
    double random_value;
    if (scalingW == 0)
        return -1;
    if (connectivity <= 0 || connectivity > 1)
        return -2;
    return 0;
}
if (Nr <= 0) return -3;
  this->scalingW = scalingW;
  this->connectivity = connectivity;
W = (double*) malloc(Nr*Nr * sizeof(double));
  memset(W, 0, Nr*Nr * sizeof(double));
  for (int ir=0; ir<Nr; ir++)
    for (int ic=0; ic<Nr; ic++)
      
        random_value = rv.random(0, 1);
        if (random_value <= connectivity)W[ir*Nr+ic] = rv.random(-scalingW, scalingW);
  // Scaling Matrix to have respect the contractivity coefficient
  // basing on the value of the L2 norm
  double max_sv, scalingFactor;
  double* copy = (double*) malloc(Nr*Nr * sizeof(double));
  memcpy((void*) copy, (void*) W, Nr*Nr * sizeof(double));
  MatrixOperations::maxSingularValue(copy, Nr, Nr, &max_sv);  free(copy);    scalingFactor = sigma/(this->degree * max_sv);
  for (int ir=0;ir<Nr;ir++)
    for (int ic=0; ic<Nr; ic++)
      W[ir*Nr+ic] *= scalingFactor;
  return 0;
}

void Reservoir::print_Wi( FILE * fp)
{  int non_zero=0;
  if (Win == NULL) return 0;
  for (int ir = 0; ir<Nr; ir++)
    for (int ic = 0; ic<
      if (Win[ir*(Ni+1)+ic]) non_zero++;
    fprintf(fp, "%lf ",Win[ir*(Ni+1)+ic];
}  fprintf(fp, "\n\n");  fprintf(fp, "Non zero elem = %d\n ", non_zero);
}void  Reservoir::print_W( FILE * fp)
{  int non_zero=0;
if (W == NULL) return;

for (int ir = 0; ir < Nr; ir++)
{
    for (int ic = 0; ic < Nr; ic++)
    {
        if (W[ir*Nr + ic] != 0)
        {
            fprintf(fp, "%lf \n", W[ir*Nr + ic]);
        }
    }
    fprintf(fp, "\n");
}

fprintf(fp, "Non zero elem = %d \n");

if (W == NULL) return NULL;

for (int i = 0; i < Nr; i++)
{
    TreeSample* ts = tds->getSample(i);
    if (ts == NULL)
    return NULL;

    int treeLen = ts->treeDim;
    int totalTreeLen = treeLen + 1;
    // +1 for the nil state
    if (totalTreeLen > stateLen)
    {
        stateLen = (stateLen * 2 > totalTreeLen) ? stateLen * 2 : totalTreeLen;
        if (states) free(states);
        states = (double*) malloc((stateLen * Nr) * sizeof(double));
    }

    // Initialization of the nil node state
    for (int nr = 0; nr < Nr; nr++)
    states[nr] = 0.0;

    // Compute the state for each tree node
    for (int nodeNumb = 0; nodeNumb < treeLen; nodeNumb++)
    {
        double tmp;
        double* input_record = ts->treeRecords + (nodeNumb * Ni);

        // Compute the node states in each reservoir neuron
        for (int nr = 0; nr < Nr; nr++)
        {
            // Input: Bias + Linear combination
            double* Win_row = W + nr * (Ni + 1);
            double* W_row = W + nr * Nr;
            tmp = Win_row[Ni];
            // Win Bias
            for (int labelIdx = 0; labelIdx < Ni; labelIdx++)
            tmp += input_record[labelIdx] * Win_row[labelIdx];

            // Sons node states: Linear combination
            for (int son = 0; son < degree; son++)
            }
int idx_son = nodeNumb*(degree+1) + (son + 1);

if (ts->treeAdjencies[idx_son] != -1) {
    double * son_state = states + (ts->treeAdjencies[idx_son]+1)*Nr;
    for (int inner_nr=0; inner_nr<Nr; inner_nr++)
        tmp+= W_row[inner_nr] * son_state[inner_nr];}

states[(nodeNumb+1)*Nr+nr] = f(tmp);    }

return states;
}

double * Reservoir::computeTreeState(TreeDataset* tds, int i, Sparse<double> *spW) {
    TreeSample* ts = tds->getSample(i);
    if (ts==NULL) return NULL;

    int treeLen = ts->treeDim;
    int totalTreeLen = treeLen + 1;  
    // +1 for the nil state
    if (stateLen<totalTreeLen) {
        stateLen = (totalTreeLen>2*stateLen)? totalTreeLen: 2*stateLen;
        if (states) free(states);
        states = (double*) malloc((stateLen*Nr) * sizeof (double));
    }

    // Initialization of the nil node state node
    for (int nr=0; nr<Nr; nr++)
        states[nr] = 0.0;

    // compute the state for each tree node
    for (int nodeNumb=0; nodeNumb<treeLen; nodeNumb++) {
        double tmp;
        double * input_record = ts->treeRecords + (nodeNumb*Ni);

        // compute the node state in each reservoir neuron
        for (int nr=0; nr<Nr; nr++)
            // Input: Bias + Linear combination
            double * Win_row = Win + nr*(Ni+1);
            tmp = Win_row[Ni];
            // Win Bias
            for (int labelIdx=0; labelIdx<Ni; labelIdx++)
                tmp +=  input_record[labelIdx] *  Win_row[labelIdx];
        // compute the state of the nil node in each reservoir neuron
        double tmp2 = 0.0;
        for (int nr=0; nr<Nr; nr++)
            tmp2 += states[nr] * Win_row[nr];

        states[nodeNumb] = f(tmp2);
    }

    return states;
}
stored in the state computation and used as additive factor

\[ \text{states}[(\text{nodeNumb} + 1) \times \text{Nr} + \text{nr}] = \text{tmp}; \]

\[ \text{Descendants states: Linear combination} \]

\[ \text{for} \quad (\text{int} \quad \text{son} = 0; \text{son} < \text{degree}; \text{son}++) \]

\[ \text{//NOTE: Take initial value for the non existing sons (TODO)} \]

\[ \text{int} \quad \text{idx}_\text{son} = \text{nodeNumb} \times (\text{degree} + 1) + (\text{son} + 1); \]

\[ \text{// skip father node} \]

\[ \text{if} \quad (\text{ts} \rightarrow \text{treeAdjencies}[\text{idx}_\text{son}] \neq -1) \]

\[ \text{double} \quad \ast \text{sonState} = \text{states} + (\text{ts} \rightarrow \text{treeAdjencies}[\text{idx}_\text{son}] + 1) \times \text{Nr}; \]

\[ \text{// addictive matrix multiplication} \]

\[ \text{spW} \rightarrow \text{additiveMultiplicationPerDense}(\text{sonState}, 1, \text{states} + (\text{nodeNumb} + 1) \times \text{Nr}); \]

\[ \text{// Computing the activation function} \]

\[ \text{for} \quad (\text{int} \quad \text{nr} = 0; \text{nr} < \text{Nr}; \text{nr}++) \]

\[ \text{states}[(\text{nodeNumb} + 1) \times \text{Nr} + \text{nr}] = \text{f}(\text{states}[(\text{nodeNumb} + 1) \times \text{Nr} + \text{nr}]); \]

\[ \text{return} \quad \text{states}; \]

\[ \text{void} \quad \text{Reservoir::stateDestroy()} \]

\[ \text{void} \quad \text{Reservoir::matrixUndestroy()} \]

\[ \text{double} \ast \text{Reservoir::computeStateMapping}(\text{double} \ast \text{state}, \text{int} \text{length}, \text{double} ** \text{stateMappingBuffer}, \text{int} \text{offset}) \]

\[ \text{// mapped state to be allocated} \]

\[ \text{additive matrix multiplication} \]

\[ \text{double, state, int length, double, stateMappingBuffer, int offset} \]

\[ \text{return} \quad \text{NULL}; \]
#ifndef __TESN_STATEMAPPING
#define __TESN_STATEMAPPING

/** State Mapping function type. It describes respectively:
   − where the state information are achievable,
   − the state mapping function that must be stored in memory between adjacent states,
   − the memory area where the state mapping must be stored
   − the offset in the resulting feature storage */

typedef void (*StateMappingFunction)(double*, int, int, double*, int);

/** Class containing state mapping functions */
class StateMapping{
public:

/** Method implementing the mean state mapping: performs the mean average of the available states.
   * @param states the address where the state information are achievable
   * @param statesNumber the number of available states
   * @param stateDimension the size of the state information achievable in memory
   * @param stateMapping the address where the state mapping must be stored
   * @param offset the offset in the resulting feature storage */
static void meanStateMapping(double* states, int statesNumber, int stateDimension, double* stateMapping, int offset=1);

/** Method implementing the super source state mapping: returns the state of the super source node.
   * @param states the address where the state information are achievable
   * @param statesNumber the number of available states
   * @param stateDimension the size of the state information achievable in memory
   * @param stateMapping the address where the state mapping must be stored
   * @param offset the offset in the resulting feature storage */
static void superSourceStateMapping(double* states, int statesNumber, int stateDimension, double* stateMapping, int offset=1);

};
#endif
```cpp
#include <string.h>
#include <stdio.h>
#include "stateMapping.h"

void StateMapping::meanStateMapping(
double* states, int statesNumber, int stateDimension,
double* stateMapping, int offset)
{
    int stateIdx;
    int stateIndex;
    int stateComp;
    if (offset != 1)
    {
        stateIndex = 0;
        for (stateComp = 0; stateComp < stateDimension; stateComp++)
        {
            stateMapping[stateIndex] = 0;
            stateIndex += offset;
        }
    } else
    {
        memset(stateMapping, 0, stateDimension * sizeof(double));
    }

    // Exclude the state 0th
    for (stateIdx = 1; stateIdx < statesNumber; stateIdx++)
    {
        stateIndex = 0;
        for (stateComp = 0; stateComp < stateDimension; stateComp++)
        {
            stateMapping[stateIndex] += states[stateIdx*stateDimension + stateComp];
            stateIndex += offset;
        }
    }

    stateIndex = 0;
    for (stateComp = 0; stateComp < stateDimension; stateComp++)
    {
        stateMapping[stateIndex] /= (statesNumber - 1);
        stateIndex += offset;
    }
    return;
}

void StateMapping::superSourceStateMapping(
double* states, int statesNumber, int stateDimension,
double* stateMapping, int offset)
{
    if (offset != 1)
    {
        stateIndex = 0;
        for (stateComp = 0; stateComp < stateDimension; stateComp++)
        {
            stateMapping[stateIndex] = states[stateIdx*stateDimension + stateComp];
            stateIndex += offset;
        }
    } else
    {
        memset(stateMapping, 0, stateDimension * sizeof(double));
    }

    // Exclude the state 0th
    for (stateIdx = 1; stateIdx < statesNumber; stateIdx++)
    {
        stateIndex = 0;
        for (stateComp = 0; stateComp < stateDimension; stateComp++)
        {
            stateMapping[stateIndex] += states[stateIdx*stateDimension + stateComp];
            stateIndex += offset;
        }
    }

    stateIndex = 0;
    for (stateComp = 0; stateComp < stateDimension; stateComp++)
    {
        stateMapping[stateIndex] /= (statesNumber - 1);
        stateIndex += offset;
    }
    return;
}
```

int stateComp;
int stateIndex = 0;
for (stateComp=0; stateComp < stateDimension; stateComp++)
{      stateMapping[stateIndex] = states[(statesNumber−1)*stateDimension + stateComp];
      stateIndex += offset;
    }
else
    {      memcpy(stateMapping, states + (statesNumber−1) * stateDimension,      stateDimension * sizeof (double));
    }
return ;
class Readout { public:
  /** Abstract class representing the readout stage */

  virtual int train(double *X, double *Y, int sampleNumber, int featuresNumber, int targetsNumber) = 0;

  virtual void prediction(double *value, double *results) = 0;

  /** Destructor method */
  virtual ~Readout() {};

  int featureNumber;
  int targetNumber;
};

class linearRegressionLMS : public Readout {
  /** Number of target of readout have to predict */
  int targetNumber;
  /** Number of feature available for each sample */
  int featureNumber;
};

class linearRegressionLMS : public Readout {
  /** Number of target of readout have to predict */
  int targetNumber;
  /** Number of feature available for each sample */
  int featureNumber;
public:
/** Default constructor method */
linearRegressionLMS();
 /** The method needed to train the linear regression tool via least mean square technique using the matrix of the samples (organized in the row major fashion) and the target values. If the training procedure is successful, the trained model is returned, otherwise the MatrixOperations::LMS result is returned. */
int train(
    double* X,
    double* Y,
    int sampleNumber,
    int featureNumber,
    int targetNumber);
 /** Method used to predict new results after the selected model has been trained. */
void prediction(
    double* value,
    double* results);
 /** Pointer to the memory area where the state of the trained model is stored. */
double* weights;
};
class linearRegressionTichonov:public Readout{
public:
/** Default constructor method */
linearRegressionTichonov();
 /** Destructor method */
~linearRegressionTichonov();
 /** Method used to set the proper regularization factor to be used for the next learning procedure. */

};
void setLambda(double value);
double getLambda();
/** The method needed to train the linear regression tool via Tikhonov Regularization technique (\( (A'A + \lambda I)^{-1} \cdot A'Y \)) using the matrix of the samples (organized in the row major fashion), the training set of the samples are required to be scored, the training set of the samples is retrieved from the training set of the samples, the equation of the training set, and the matrix needed to train the linear regression tool via Tikhonov Regularization technique is retrieved. 
+ \( (A'A + \lambda I)^{-1} \cdot A'Y \)
+ $$\text{setLambda(}\text{double value})$$
+ $$\text{getLambda()}$$
+ $$\text{train(double* X, double* Y, int sampleNumber, int featureNumber, int targetNumber)}$$
+ $$\text{prediction(double* value, double* results)}$$
+ $$\text{weights}$$
+ $$\text{lambda}$$
+ $$\text{square}$$
*/
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "readout.h"
#include "matrixOperations.h"

linearRegressionLMS::linearRegressionLMS() { weights = NULL; }

int linearRegressionLMS::train(double* X, double* Y, int sampleNumber, int featureNumber, int targetNumber)
{
    int result;
    int workingDimension;
    if(weights == NULL)
    {
        workingDimension = (featureNumber > sampleNumber) ? featureNumber : sampleNumber;
        weights = (double*) malloc(workingDimension * targetNumber * sizeof(double));
    }
    else if(this->featureNumber != featureNumber || this->targetNumber != targetNumber)
    {
        free(weights);
        workingDimension = (featureNumber > sampleNumber) ? featureNumber : sampleNumber;
        weights = (double*) malloc(workingDimension * targetNumber * sizeof(double));
    }
    this->featureNumber = featureNumber;
    this->targetNumber = targetNumber;
    memcpy(weights, Y, sampleNumber * targetNumber * sizeof(double));
    result = MatrixOperations::LMS(X, weights, sampleNumber, featureNumber, targetNumber);
    if(result != 0) free(weights);
    return result;
}

void linearRegressionLMS::prediction(double* value, double* results)
{
int target;
for (target=0; target<targetNumber; target++)
{    results[target] = 0.0;
    for (int i=0; i<featureNumber; i++)
    {        results[target] += value[i]*weights[target*featureNumber+i];    }
}

linearRegressionTichonov::linearRegressionTichonov(){
    weights = NULL;
    square = NULL;
    lambda = 0.0;
}

linearRegressionTichonov::~linearRegressionTichonov(){
    if (square) { free(square); }
    if (weights) free(weights);
}

void linearRegressionTichonov::setLambda(double value)
{    lambda = value;
}

double linearRegressionTichonov::getLambda()
{    return lambda;
}

// Used Tichonov Regularization: (A'*A + lamda*I)^-1 * A' Yint
linearRegressionTichonov::train(double* X, double* Y, int sampleNumber, int featureNumber, int targetNumber)
{
    int i;
    int result=0;
    double* partial;
    this->featureNumber = featureNumber;
    this->targetNumber = targetNumber;
    /// TODO: Retrain function feature reusing the square matrix
    if (square == NULL)
    {        square = (double*) malloc(featureNumber*featureNumber *
sizeof(double));
    }
    // in the implementation A is already transposed (wrt the paper)
    MatrixOperations::At_A(X, square, sampleNumber, featureNumber);
    // adding lambda
    double* tmp=square;
    for (i=0; i<featureNumber; i++)
    {        *tmp += lambda;
        // next diagonal element
        tmp += (featureNumber+1);
    }
```c
result = MatrixOperations::inverse(square, featureNumber);
if (result < 0) {printf("inverse fail
"); return ...
```

```c
partial = (double*)malloc(featureNumber*sampleNumber *
```

```c
MatrixOperations::A_Bt(square, X, partial, featureNumber, featureNumber, sampleNumber);
```

```c
if (weights == NULL) weights = (double*) malloc(featureNumber*targetNumber *
```

```c
MatrixOperations::Matrix_Matrix(partial, Y, weights, 1.0, 0.0, featureNumber, sampleNumber, targetNumber); free(partial);
```

```c
return result;
```

```c
void linearRegressionTichonov::prediction(double* value, double* results)
```

```c
int target;
```

```c
for (target=0; target<targetNumber; target++) {    results[target] = 0.0;
```

```c
for (int i=0;i<featureNumber;i++) results[target] += value[i]*weights[target*featureNumber+i];
```

```c
return result;
```

```c
free(partial);
```

```c
MatrixOperations::Matrix_Matrix(value, results, 1.0, 0.0, featureNumber, sampleNumber, featureNumber);
```

```c
result = MatrixOperations::inverse(A-Bt, featureNumber);
```

```c
if (result < 0) {printf("inverse fail
"); return ...
```

```c
linearRegressionTichonov::prediction(value, results)
```

```c
for (target=0; target<targetNumber; target++) {    results[target] = 0.0;
```

```c
for (int i=0;i<featureNumber;i++) results[target] += value[i]*weights[target*featureNumber+i];
```
```cpp
#include <stdlib.h>
#include <stdio.h>
#include "encoding.hpp"

class Dense {
public:
    Dense(double *matrix, int N, int M) {
        this->matrix = matrix;
        this->N = N;
        this->M = M;
        this->destroy = false;
    }

    Dense(char *fileName, bool plain=true) {
        double *temp;
        FILE *fp = fopen(fileName, "r");
        if (fp == NULL) return;
        printf("Load:%s\n", (plain) ? "true" : "false");
        fscanf(fp, "#%d#%d#
" , &this->N, &this->M);
        matrix = (double *) malloc(N*M*sizeof(double));
        if (matrix == NULL) return;
        temp = matrix;
        for (int i=0; i<N; i++)
            for (int j=0; j<M; j++)
                if (plain) fscanf(fp, "%lf", temp);
                else Encoding<double>::read(fp, temp);
            fscanf(fp, "\n");
        fclose(fp);
    }

private:
    double *matrix;
    int N, M;
    bool destroy;
};
```

21 apr 15 14:23 Pagina 1/2

dense.hpp

/*  Author:  Daniele Virgilio    Credits: Computational Intelligence & Machine Learning Group − University of Pisa        Parallel Programming Models Group − University of Pisa */
```c++
int store(char * fileName, bool plain = true)
{
    FILE * fp = fopen(fileName, "w");
    if (fp == NULL)
        return -1;
    printf("Store:\n%"/(plain)?"true":"false)
    double *temp = matrix;
    fprintf(fp, "#%d#%d#
", N, M);
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            if (plain) fprintf(fp, "%.10lf"/(temp);
            else Encoding<double>::write(fp, temp);
            temp++;
        }
        fprintf(fp, "\n"/(N - M));
    }
    fclose(fp);
    return 0;
}
```
#include <stdlib.h>
#include <stdio.h>
#include "encoding.hpp"

template<typename T>
class Sparse {
public:
    Sparse(T* matrix, int N, int M, int nonZero=-1) {
        int nonZeroIndex;
        int i, j;
        this->destroy = false;
        if (nonZero==−1) {
            nonZero = 0;
            for (i=0; i<N; i++)
                for (j=0; j<M; j++)
                    if (matrix[i*M+j]!=0) nonZero++;
        }
        if (nonZero!=0) {
            this->N = N;
            this->M = M;
            indexRow = (int*) malloc(nonZero*sizeof(int));
            indexColumn = (int*) malloc(nonZero*sizeof(int));
            values = (T*) malloc(nonZero*sizeof(T));
            nonZeroIndex=0;
            for (i=0; i<N; i++)
                for (j=0; j<M; j++)
                    if (matrix[i*M+j]!=0) {
                        indexRow[nonZeroIndex] = i;
                        indexColumn[nonZeroIndex] = j;
                        values[nonZeroIndex] = matrix[i*M+j];
                        nonZeroIndex++;
                    }
    }

    ~Sparse() {
        this->destroy = true;
        if (nonZero) {
            delete[] indexRow;
            delete[] indexColumn;
            delete[] values;
        }
    }

    T& operator()(int i, int j) {
        return values[indexRow[i]*M+indexColumn[i]];
    }

private:
    int nonZero;
    int N, M;
    T* indexRow;
    T* indexColumn;
    T* values;
};
nonZeroElem = nonZero;    }
else
    nonZeroElem = -1;  }

Sparse(
    char * fileName,
    bool plain=true)
{
    nonZeroElem=0;
    N=0; M=0;
    this->destroy = false;
    FILE * fp = fopen(fileName, "r");
    if (fp == NULL)
      return -1;
    fscanf(fp, "#%d#%d#%d#
", &nonZeroElem, &N, &M);
    indexRow = (int*) malloc(nonZeroElem*sizeof(int));
    indexColumn = (int*) malloc(nonZeroElem*sizeof(int));
    values = (T*) malloc(nonZeroElem*sizeof(T));
    for (int i=0; i<nonZeroElem; i++)
      if (plain)
        fscanf(fp, "<%d#%d#%.10lf>
", indexRow+i, indexColumn+i, values+i);
      else
        {
          Encoding<int>::read(fp, indexRow+i);
          Encoding<int>::read(fp, indexColumn+i);
        }
    fclose(fp);
}

int store(
    char * fileName,
    bool  plain=true)
{
    int i;
    FILE * fp = fopen(fileName, "w");
    if (fp == NULL)
      return -1;
    fprintf(fp, "#%d#%d#%d#
", nonZeroElem, N, M);
    for (i=0; i<nonZeroElem; i++)
      if (plain)
        fprintf(fp, "<%d#%d#%.10lf>
", indexRow[i], indexColumn[i], values[i]);
      else
        {
          Encoding<int>::write(fp, indexRow+i);
          Encoding<int>::write(fp, indexColumn+i);
        }
    fclose(fp);
}
void additiveMultiplicationPerDense(T* B, int K, T* result)
{
    int i, j, k;
    int indexSparse = 0;
    for (indexSparse = 0; indexSparse < nonZeroElem; indexSparse++)
    {
        // for each column of B
        i = indexRow[indexSparse];
        j = indexColumn[indexSparse];
        // transposed representation
        for (k = 0; k < K; k++)
            result[k * N + i] += values[indexSparse] * B[k * M + j];
    }
}

void print()
{
    for (int i = 0; i < nonZeroElem; i++)
        printf("<%d,%d,%lf>\n", indexRow[i], indexColumn[i], values[i]);
}

int getN() { return N; }

int getM() { return M; }

void setDestroy(bool destroy = true)
{
    this->destroy = destroy;
    free(indexRow);
    free(indexColumn);
    free(values);
}

~Sparse()
{
    if (destroy)
    {
        free(indexRow);
        free(indexColumn);
        free(values);
    }
}

private:

    int N;
    int M;
    // internal representation
    int* indexRow;

// sparse.hpp
```c
#include

int* indexColumn;
T* values;
int nonZeroElem;
bool destroy;

#endif
```
class MatrixCBLAS { 
public:

/** A generic matrix to matrix multiplication (Cblas wrapper). It
  * implements the C = alpha * (A B) + beta * C   
  * @param A the left side matrix [NxK] need to be multiplied. 
  * @param B the right side matrix [KxM] need to be multiplied.
  * @param C the matrix [NxM] whose the transpose need
  * to be multiplied.  
  * @param alpha the scalar that multiplies the A matrix.  
  * @param beta the scalar that multiplies the C matrix.
  * @param N number of columns of the A matrix.
  * @param K number of rows of the A matrix  
  * @param M number of columns of the B matrix   
  */
static 
void 
Matrix_Matrix(
double *A,
double *B,
double *C,
double alpha,
double beta,
int N,
int K,
int M);

/** Implements the multiplication from a matrix A and
  * the transponse of the matrix B (ie. C = A B').   
  * @param A the left side matrix [NxK] need to be multiplied. 
  * @param B the right side matrix [KxM] whose the transpose need
  * to be multiplied.  
  * @param C the matrix [NxA'] whose the transpose need  
  * @param N number of columns of the A matrix.
  * @param K number of rows of the A matrix  
  * @param M number of columns of the B matrix   
  */
static 
void 
A_Bt(
double *A,
double *B,
double *C,
int N,
int M,
int K);

/** Implements the multiplication from a transposed matrix A and
  * the orginal A matrix (ie. C = A' A).   
  * @param A the matrix [NxM] need to be processed.   
  * @param C the matrix [NxA'] whose the transpose need
  * to be multiplied.  
  * @param N number of rows of the matrix   
  * @param M number of columns of the matrix   
  */
static 
void 
At_A(
double *A,
double *C,
int N,
int M);

};
class MatrixLAPACKE {
public:
    /** Inverts a square matrix A using the LU factorization method. 
     * @param A the square matrix [NxN] need to be inverted. The matrix A 
     * is modified after the method call, −1 if the LU decomposition fails, 
     * −2 if the inversion process fails. 
     * @return 0 if the LU decomposition succeed, −1 if the inversion process 
     * failed. 
     */
    static int inverse(double* A, int N);

    /** Method to solve a Least Mean Square problem (@f$ \min_x \|Ax - B\|_2 @f$) 
     * @param A system matrix [NxM]. The matrix A is modified after the 
     * method call, −1 if the LMS process fails, −2 if the LMS process 
     * was not able to find a solution. 
     * @param B input matrix [Mx1]. The matrix B is modified after the 
     * method call, −1 if the LMS process fails, −2 if the LMS process 
     * was not able to find a solution. 
     * @param K number of columns of the B matrix. 
     * @return 0 if the LMS process successfuly complete, −1 otherwise. 
     */
    static int LMS(double* A, double* B, int N, int M, int K);

    /** Method to returning the Maximum Singolar Value of the matrix 
     * achived by solving its Singular Value Decomposition 
     * @param A system matrix [NxM]. The matrix A is modified after the 
     * method call, −1 if the Singular Value Decomposition fails, 
     * −2 if the Singular Value Decomposition was not able to find a solution. 
     * @param result pointer where the result will be stored. 
     * @return 0 if the Singular Value Decomposition successfuly complete, 
     * −1 otherwise. 
     */
    static int maxSingularValue(double* A, int N, int M, double* result);
};
```c
#include "matrixOperations.h"

#ifdef __MKL
#include "mkl_cblas.h"
#else
#include <common.h>
#include <cblas.h>
#endif

void MatrixCBLAS::Matrix_Matrix(double* A, double* B, double* C, double alpha, double beta, int N, int M)
{
    int ldA = K;
    int ldB = M;
    int ldC = M;
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, M, K,
                alpha, A, ldA, B, ldB, beta, C, ldC);
}

void MatrixCBLAS::A_Bt(double* A, double* B, double* C, int N, int M, int K)
{
    int ldA = M;
    int ldB = M;
    int ldC = K;
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, N, K, M,
                1.0, A, ldA, B, ldB, 0.0, C, ldC);
}

void MatrixCBLAS::At_A(double* A, double* C, int N, int M)
{
    int ldA = M;
    int ldC = M;
    cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, M, M, N,
                1.0, A, ldA, A, ldA, 0.0, C, ldC);
}
```
#include <matrixOperations.h>
#include <stdlib.h>
#include <stdio.h>

#ifdef __MKL
#include "mkl_lapacke.h"
#else
#include "lapacke.h"
#endif

int MatrixLAPACKE::inverse(double *A, int N)
{
    lapack_int PIV[N];
    if (LAPACKE_dgetrf(LAPACK_ROW_MAJOR, N, N, A, N, PIV) != 0)
        return -1;
    if (LAPACKE_dgetri(LAPACK_ROW_MAJOR, N, A, N, PIV) != 0)
        return -2;
    return 0;
}

int MatrixLAPACKE::LMS(double *A, double *B, int N, int M, int K)
{
    if (LAPACKE_dgels(LAPACK_ROW_MAJOR, '
        return -1;
    return 0;
}

int MatrixLAPACKE::maxSingularValue(double *A, int N, int M, double *result)
{
    int m=N, n=M, lda=M, ldu=N, ldvt=M;
    int superb_dim = ((N<M)? N:M)−1;
    int info;
    double *s = (double*) malloc( M * sizeof(double));
    double *u = (double*) malloc( ldu*N * sizeof(double));
    double *vt = (double*) malloc( ldvt*M * sizeof(double));
    info = LAPACKE_dgesvd(LAPACK_ROW_MAJOR, 'A', 'A', m, n, A, lda,
        ... });
    info = LAPACKE_dgesvd(LAPACK_ROW_MAJOR, 'A', 'A', m, n, A, lda,
        ... });
    return 0;
}
if (info > 0) return -1;

// Specification states the singular values are stored in descending order
*result = s[0];
free(s);
free(u);
free(vt);
return 0;
#ifndef __TENSEN_DATASET
#define __TENSEN_DATASET

/** Class describing a tree sample */
class TreeSample{public:

/** Name associated to the tree */
char* name;

/** Tree target */
double target;

/** Number of nodes contained in the tree */
int treeDim;

/** Number of features contained in each node */
int labelDim;

/** The number of connection for each node in the tree */
int connectivityDegree;

/** Linear array containing all the features nodes */
double* treeRecords;

/** Linear array containing the adjacency list of the nodes. For each node, the first adjacent node in the list represents the father. */
int* treeAdjencies;

/** Default constructor method */
TreeSample();

/** Constructor explicitly define the tree parameters @param name the tree name @param target the target associated to the whole tree @param treeDim the number of nodes contained in the tree @param labelDim the number of features contained in each node @param connectivityDegree the number of connection for each node in the tree */
TreeSample(char* name, double target, int treeDim, int labelDim, int connectivityDegree);

/** Method to set the main tree parameters @param name the tree name @param target the target associated to the whole tree @param treeDim the number of nodes contained in the tree @param labelDim the number of features contained in each node @param connectivityDegree the number of connection for each node in the tree */
void setTreeSample(char* name, double target, int treeDim, int labelDim, int connectivityDegree);

};
#endif

/*  Author: Daniele Virgilio    Credits: Computational Intelligence & Machine Learning Group − University of Pisa        Parallel Programming Models Group − University of Pisa */
```cpp
/** Print a linear representation of the whole tree */
void print();

/** Class representing the whole dataset composed by tree samples */
class TreeDataset{
public:
  /** The number of instance in the dataset */
  int sampleNumber;
  /** The maximum arity of each node in the dataset */
  int maxArity;
  /** The number of features for each tree node */
  int labelDim;
  /** The array containing the tree instances */
  TreeSample * treeSamples;

  /** Constructor TreeDataset method that parses a dataset file 
   * @param fileName the file to be parsed 
   */
  TreeDataset(char * filename);

  /** Constructor method to set explicitly the dataset information. 
   * @param sampleNumber the number of instance in the dataset 
   * @param maxArity the maximum arity of each node in the dataset 
   * @param labelDim the number of features for each tree node 
   * @param treeSamples the array containing the tree instances 
   */
  TreeDataset(int sampleNumber, int maxArity, int labelDim, TreeSample* treeSamples);

  /** Return the index\textsuperscript{th} sample in the dataset. 
   * @param index the number of the tree sample need to return 
   * @return the index\textsuperscript{th} sample in the dataset 
   */
  TreeSample * getSample(int index);
};
#endif
```

#include <iostream>
#include "tree_dataset.h"
#include "parser.h"

TreeSample::TreeSample()
{
    this->name = NULL;
    this->treeDim = 0;
    this->treeRecords = NULL;
    this->treeAdjencies = NULL;
}

TreeSample::TreeSample(char *name, double target, int treeDim, int label_dim, int connectivity_degree)
{
    this->name = name;
    this->target = target;
    this->treeDim = treeDim;
    this->labelDim = label_dim;
    this->connectivityDegree = connectivity_degree;
    this->treeRecords = new double[treeDim * label_dim];
    this->treeAdjencies = new int[treeDim * connectivity_degree];
}

void TreeSample::setTreeSample(char *name, double target, int treeDim, int label_dim, int connectivity_degree)
{
    this->name = name;
    this->target = target;
    this->treeDim = treeDim;
    this->labelDim = label_dim;
    this->connectivityDegree = connectivity_degree;
    this->treeRecords = new double[treeDim * label_dim];
    this->treeAdjencies = new int[treeDim * connectivity_degree];
}

void TreeSample::print()
{
    int i, j;
    std::cout << "Tree name = " << name << std::endl;
    std::cout << "TreeDim = " << treeDim << std::endl;
    std::cout << "Target = " << target << std::endl;
    std::cout << "LabelDim = " << labelDim << std::endl;
    std::cout << "Connectivity Degree = " << connectivityDegree << std::endl;
    std::cout << "TreeRecords = " << treeRecords << std::endl;
    std::cout << "TreeAdjencies = " << treeAdjencies << std::endl;
}

void TreeSample::set.TreeSample()
{
    this->name = name;
    this->target = target;
    this->treeDim = treeDim;
    this->labelDim = labelDim;
    this->connectivityDegree = connectivity_degree;
    this->treeRecords = new double[treeDim * label_dim];
    this->treeAdjencies = new int[treeDim * connectivity_degree];
}

void TreeSample::print()
{
    int i, j;
    std::cout << "Tree name = " << name << std::endl;
    std::cout << "TreeDim = " << treeDim << std::endl;
    std::cout << "Target = " << target << std::endl;
    std::cout << "LabelDim = " << labelDim << std::endl;
    std::cout << "Connectivity Degree = " << connectivityDegree << std::endl;
    std::cout << "TreeRecords = " << treeRecords << std::endl;
    std::cout << "TreeAdjencies = " << treeAdjencies << std::endl;
}

/*
 * Author: Daniele Virgilio
 * Credits: Computational Intelligence & Machine Learning Group - University of Pisa
 * Parallel Programming Models Group - University of Pisa
 */
```cpp
TreeDataset::TreeDataset(char* filename) {
  TreeDatasetParser tdsp;  
  tdsp.parse_dataset(filename, this);
}

TreeDataset::TreeDataset(int sampleNumber, int maxArity, int labelDim, TreeSample* treeSamples) {
  this->sampleNumber = sampleNumber;
  this->maxArity = maxArity;
  this->labelDim = labelDim;
  this->treeSamples = treeSamples;
}

TreeSample* TreeDataset::getSample(int index) {
  if (index < 0 || index > sampleNumber)
    return NULL;
  return treeSamples + index;
}
```
#ifndef __TESN_PARSER
#define __TESN_PARSER

#include <stdio.h>
#include "tree_dataset.h"

/** Class used to parse the dataset file. The dataset parsed is called * gph and its format is: * * HEADER: *    TreeNum <sampleNumber> *    MaxArity <arityNumber> *    LabelDim <featuresNumber> *    <TREE_SAMPLE>* * *

class TreeDatasetParser{public:
/** Default constructor */
TreeDatasetParser();

/** Method used to parse the dataset file in a TreeDataset structure * @param fileName name of the gph file containing the dataset * @param tds_in (optional) the already created ... * @return the TreeDataset representing the parsed dataset, * NULL if the parsing process fail */
TreeDataset * parse_dataset(char * fileName, TreeDataset* tds_in=NULL);

private:
/** The file descriptor related to the dataset file */
FILE * dataset_file;
/** Current size of the buffer used for buffering */
size_t max_buff;
/** Buffering area */
char * buffer;

/** Internal method used to parse a sample tree. A tree has the * following format: *<TREE_SAMPLE>: *  Name <stringName> */
tree_dataset * parse_tree(char * fileName, tree_dataset * tds_in=0);

/* NULL if the parsing process fail */
/* Execute the TreeDataset representing the parsed dataset, */
/* to be filled (it has been already created) */
/* the already created TreeDataset structure */
/* the file containing the gph */
/* the dataset file used to parse the dataset file in a TreeDataset structure */
TreeDatasetParser();
/* Default constructor */
}

class TreeDatasetParser{
/*
 * TREE_SAMPLE: *<tree_name> "name"<features> <arity> *"label":<tree_structure> ...
 */

#include <stdlib.h>
#include <stdio.h>

#define TESN_PARSER
#endif

#ifdef TESN_PARSER
#include <stdio.h>
#endif
#include <iostream>

int parse_tree(TreeSample* ts, int label_dim, int connectivity_degree);
```cpp
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tree_dataset.h"
#include "parser.h"
using namespace std;

TreeDatasetParser::TreeDatasetParser(){
  max_buff = 256;
  buffer = new char[max_buff];
}

TreeDataset* TreeDatasetParser::parse_dataset(char* fileName, TreeDataset* tds_in)
{
  int result = 0;
  char* p = NULL;
  TreeDataset *tds = NULL;
  int tree_num = -1;
  int max_arity = -1;
  int label_dim = -1;
  TreeSample * treeSamples;

  dataset_file = fopen(fileName, "r");
  if (dataset_file != NULL)
  {
    //get dataset parameters
    while ( getline(&buffer, &max_buff, dataset_file) )
    {
      p = strtok (buffer, " ");
      if(strcmp(buffer, "TreeNum")==0){
        p = strtok(NULL," ");
        tree_num = atoi(p);
      }
      else if (strcmp(buffer, "MaxArity")==0){
        p = strtok(NULL," ");
        max_arity = atoi(p);
      }
      else if (strcmp(buffer, "LabelDim")==0){
        p = strtok(NULL," ");
        label_dim = atoi(p);
      } else {
        break;
      }
    }
    //close dataset file
    fclose(dataset_file);
  }

  //create tree dataset
  tds = new TreeDataset;
  tds->treeSamples = new TreeSample[treesNum];
  tds->max_arity = max_arity;
  tds->label_dim = label_dim;
  tds->tree_num = tree_num;

  return tds;
}
```

if (tree_num<1 || max_arity<1 || label_dim<1)
{
  cerr<< "Error in parsing the tree features
  tree_num = " << tree_num 
  " max_arity = " << max_arity 
  " label_dim = " << label_dim 
  << "n"
  "fclose(dataset_file);
  return NULL;
}

// allocate data structures

TreeSample* treeSamples = new TreeSample[tree_num];
result = 0;
for (int i=0; i<tree_num && result == 0; i++)
  result = parse_tree(treeSamples+i, label_dim, max_arity+1);

if (tds_in==NULL)
  tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
else
  tds_in->sampleNumber = tree_num;
  tds_in->maxArity = max_arity;
  tds_in->labelDim = label_dim;
  tds_in->treeSamples = treeSamples;
  tds = tds_in;

fclose(dataset_file);

return tds;

int TreeDatasetParser::parse_tree(TreeSample* ts,
  int label_dim,
  int connectivity_degree)
{
  int i,j;
  char* p = NULL;
  char* name = NULL;
  int treeDim = -1;
  double target = 0.0;

  // parse the 3 tree parameters
  while ( getline(&buffer, &max_buff, dataset_file) )
  {
    if (strcmp(buffer,"n")!=0)
      label_dim = atoi(p);
    else  break;
  }
  if (tree_num<1 || max_arity<1 || label_dim<1)
  {
    cerr<< "Error in parsing the tree features
    tree_num = " << tree_num 
    " max_arity = " << max_arity 
    " label_dim = " << label_dim 
    << "n"
    "fclose(dataset_file);
    return NULL;
  }

  tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = 0;
  for (int j=0; j<tree_num && result == 0; j++)
    result = parse_tree(treeSamples[j], label_dim, max_arity+1);
  return tds;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;

  if (tds_in==NULL)
    tds = new TreeDataset(tree_num, max_arity, label_dim, treeSamples);
  result = parse_tree(treeSamples, label_dim, max_arity+1);
  return tds;

  cerr << "Unable to open the dataset file 
";
  return NULL;
```c
if (strcmp(buffer, "Name") == 0) {
    p = strtok(NULL, " ");
    if (name == NULL) {
        i = strlen(p);
        // replace '\n' with '\0'
        if (p[i - 1] != '\n') i++;
        name = new char[i];
        strncpy(name, p, i);
        name[i - 1] = '\0';
    }
} else if (strcmp(buffer, "Target") == 0) {
    p = strtok(NULL, " ");
    target = atof(p);
} else if (strcmp(buffer, "TreeDim") == 0) {
    p = strtok(NULL, " ");
    treeDim = atoi(p);
} else break;

if (treeDim <= 0) {
    cerr << "Error in parsing the tree details " << treeDim << "\n"
    fclose(dataset_file);
    return -1;
}

// build the record data structure
ts->setTreeSample(name, target, treeDim, label_dim, connectivity_degree);
```

for (j=0; j<label_dim; j++)
    {      p = strtok( NULL ,"  ");
      if (p== NULL ) break ;
            ...
            if (getline(&buffer, &max_buff, dataset_file) <= 0) break ;
  }  
if (i!=treeDim) return  −1;
  return  0;
}
```cpp
#include "dense.hpp"
#include "sparse.hpp"

/** Class used to store the **/
class TreeEsnState {
  public:
    static int saveEchoState(char* fileName, double* denseMatrix, int N, int M, bool plain = true) {
      Dense dm(denseMatrix, N, M);
      return dm.store(fileName, plain);
    }

    static int saveEchoState(char* fileName, Sparse<double>* sp, bool plain = true) {
      return sp->store(fileName);
    }

    static int saveLinarReadoutState(char* fileName, double* readout, int N, bool plain = true) {
      Dense dm(readout, 1, N);
      return dm.store(fileName, plain);
    }

    static void loadEchoState(char* fileName, Dense** denseMatrix, bool plain = true) {
      *denseMatrix = new Dense(fileName, plain);
    }

    static void loadEchoState(char* fileName, Sparse<double>** sparseMatrix, bool plain = true) {
      *sparseMatrix = new Sparse<double>(fileName);
    }

    static void loadLinarReadoutState(char* fileName, Dense** readout, bool plain = true) {
      *readout = new Dense(fileName, plain);
    }
};
```
#ifndef __TESN_ENCODING
#define __TESN_ENCODING
// TODO: testing

template <typename T>
class Encoding{
public:
static void read(FILE* fp, T* data)
{
    fread(data, sizeof(char), sizeof(T), fp);
}

static void write(FILE* fp, T* data)
{
    fwrite((char*)data, sizeof(char), sizeof(T), fp);
}
};
#endif
```cpp
#include <sys/time.h>

/** Class used to measure the elapsed time in milliseconds */
class Timer {
public:
  /** Default constructor method */
  Timer();
  /** Set the left limit of the interval need to be measured */
  void tic();
  /** Set the right limit of the interval need to be measured */
  /** Compute the time interval of the selected time interval. */
  long getDifference();
private:
  /** The left limit of the time interval */
  struct timeval t_tic;
  /** The right limit of the time interval */
  struct timeval t_toc;
  /** Define if the tic procedure has been called */
  bool tic_initialized;
  /** Define if the toc procedure has been called */
  bool toc_initialized;
};
```

#include <stdlib.h>
#include <time.h>

Timer::Timer() {  
tic_initialized = false;

toc_initialized = false;
}

void Timer::tic() {  
tic_initialized = true;

called_timeevaled t_tmp;

tic = gettimeofday(&t_tic, NULL);
}

long Timer::toc() {  
if (!tic_initialized) return -1;

toc_initialized = true;

called_timeevaled t_tmp;
	toc = gettimeofday(&t_toc, NULL);

return getDifference();
}

long Timer::getDifference() {  
struct timeval t_tmp;

if (tic_initialized && toc_initialized) {    
timersub(&t_toc, &t_tic, &t_tmp);
return t_tmp.tv_sec*1000 + t_tmp.tv_usec/1000;
}  
else  return 0;
}
#ifndef _RANDOM
#define _RANDOM

/** Class used for generating random numbers */
class Random{
public:

/** Default constructor methods */
Random();

/** Generate a random integer number in the [0, RAND_MAX] interval. */
int random(void);

/** Generate a random double value in the selected interval
@parma lowerbound the generation interval lowerbound
@parma upperbound the generation interval upperbound
@return a random double number in the [lowerbound, upperbound] interval. 
*/
double random(double lowerbound, double upperbound);
};
#endif
```cpp
#include <time.h>
#include <stdlib.h>
#include "random.h"

Random::Random() { srand(time(NULL)); }

int Random::random() { return rand(); }

double Random::random(double lowerbound, double upperbound) {
  double difference = upperbound - lowerbound;
  return (((double)random()) / RAND_MAX) * difference + lowerbound;
}
```

```c
#include <stdio.h>
#include <string.h>
#include "../reservoir.h"
#include "../readout.h"

int main ()
{
    TreeDataset tds("./test/minimal/dataset.ds");
    printf("<<Dataset>>\n");
    for (int i=0;i<tds.sampleNumber;i++)
        tds.getSample(i)->print();

    Reservoir rsv;
    int result;
    // input dimension
    rsv.Ni = tds.labelDim;
    // number of child per node
    rsv.degree = tds.maxArity;
    // reservoir dimension
    rsv.Nr = 15;
    // set scaling and connectivity
    result=rsv.initialize_Win(0.5,1.0); printf("<<Initialization of Win %d>>\n", result);
    if (result==0)
    {
        printf("W_i:\n");
        rsv.print_Wi(stdout);
    }
    // set scalingW, connectivity, sigma
    result = rsv.initialize_W(3.3, 0.2,0.5); printf("<<Initialization of W %d>>\n", result);
    if (result==0)
    {
        printf("W:\n");
        rsv.print_W(stdout);
    }

    double * states[tds.sampleNumber];
    double finalMappedState[tds.sampleNumber * (rsv.Nr+1)];
    double tmpMappedState[rsv.Nr+1];
    double *stateMappingPointers[rsv.stateMappingNumber];
```
  
  for (int idx=0; idx<tds.sampleNumber; idx++){
    printf("Full state of %s
", tds.getSample(idx)->name);
    states[idx] = rsv.computeTreeState(&tds, idx);
    for (int node=0; node<=tds.getSample(idx)->treeDim; node++)
      for (int nr=0; nr<rsv.Nr; nr++)
        printf("%lf
", states[idx][node*rsv.Nr+nr]);
    printf("Reduced state of %s
", tds.getSample(idx)->name);
    stateMappingPointers[0] = finalMappedState + idx*(rsv.Nr+1);
    rsv.computeStateMapping(states[idx], tds.getSample(idx)->treeDim+1, stateMappingPointers, 1);
    printf("SuperSourceStateMapping
");
    for (int nr=0; nr<=rsv.Nr; nr++)
      printf("%lf
", stateMappingPointers[0][nr]);
    printf("MeanStateMapping
");
    for (int nr=0; nr<=rsv.Nr; nr++)
      printf("%lf
", stateMappingPointers[1][nr]);
  }
  printf("<<Final State Matrix>>
  
  for (int idx=0; idx<tds.sampleNumber; idx++){
    for (int nr=0; nr<=rsv.Nr; nr++)
      printf("%lf
", finalMappedState[idx*(rsv.Nr+1) + nr]);
  }
  printf("<<Targets values>>
  
  double targets[tds.sampleNumber];
  for (int idx=0; idx<tds.sampleNumber; idx++)
    targets[idx] = tds.getSample(idx)->target;
  for (int idx=0; idx<tds.sampleNumber; idx++)
    printf("%lf
", targets[idx]);
  
  linearRegressionTichonov lrt;  lrt.setLambda(3.0);  lrt.train(finalMappedState, targets, tds.sampleNumber, rsv.Nr+1, 1);"
Random rv;

double newState[rsv.Nr+1];
double predicted;

for (int nr=0; nr<=rsv.Nr; nr++) newState[nr] = rv.random(-1,1);

printf("<<Prediction of ");

for (int nr=0; nr<=rsv.Nr; nr++) printf("%lf ", newState[nr]);

printf(">>
");
lrt.prediction(newState, &predicted);

printf("%lf
", predicted);
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
#include "../reservoir.h"
#include "../readout.h"
#include "../tesnState.cpp"

void fillVector(char* path, std::vector<int>* vect)
{
    int tmp, result;
    FILE* fp = fopen(path, "r");
    result = fscanf(fp, "%d
" , &tmp);
    while(result==1)
    {
        vect->push_back(tmp);
        result = fscanf(fp, "%d
" , &tmp);
    }
}

int main(void)
{
    TreeDataset* tds;  Sparse<double> *spW;
    Dense *Wi;  Dense *W;
    int Nr;
    std::vector<int>* TR = new std::vector<int>();
    tds = new TreeDataset("./test/oracoloData-dic/dataset.ds");
    fillVector("./test/oracoloData-dic/foldsSkeleton-TR.txt", TR);
    TreeEsnState::loadEchoState("./test/oracoloData-dic/matrixW.txt", &W,
true);
    printf("Matrice W [%d,%d]
", W->getN(), W->getM());
    spW = new Sparse<double>(W->getMatrix(), W->getN(), W->getM());
    TreeEsnState::loadEchoState("./test/oracoloData-dic/matrixWin.txt", &Wi,
true);
    Reservoir rsv(spW->getN(), tds->labelDim, tds->maxArity);
    Nr = spW->getN();
    printf("rsv(%d,%d,%d);
",spW->getN(), tds->labelDim, tds->maxArity);
    rsv.Win=Wi->getMatrix();    rsv.W=W->getMatrix();
linearRegressionTichonov *lrt = new linearRegressionTichonov();
int sampleNumber = tds->sampleNumber;
int offset = 1;
double * states[sampleNumber];
double * finalMappedState = (double*) malloc(sampleNumber*(Nr+1) * sizeof(double));
double * stateMappingPointer;
std::vector<int>::iterator it;
int idx;
for (idx=0, it = TR->begin(); it != TR->end(); idx++, it++) {
    states[idx] = rsv.computeTreeState(tds, *it, spW);
    stateMappingPointer = finalMappedState + idx*(Nr+1);
    rsv.computeTreeState(tds, *it, TR->getSample(*it)->treeDim+1, rsv.Nr, stateMappingPointer, offset);
    stateMappingPointer[rsv.Nr] = 1.0;
}
int targetDimension = 1;
double lambda = 0.1;
// load target values
double targetsTR[TR->size() * sizeof(double)];
for (it=TR->begin(), idx=0; it!=TR->end(); it++, idx++) targetsTR[idx] = tds->getSample(*it)->target;
lrt = new linearRegressionTichonov();
lrt->setLambda(lambda);
lrt->train(finalMappedState, targetsTR, TR->size(), Nr+1, targetDimension);
TreeEsnState::saveEchoState("./oracoloData-dic/TEST_Tichonov", lrt->weights, 1, rsv.Nr+1, true);
}
#include <iostream>
#include <vector>
#include "farm.hpp"
#include ../time.h
#include ../reservoir.h
#include ../readout.h
#include ../tesnState.cpp
#include validationInfo.cpp
#include commonValidation.hpp

class ESN_Configuration {
public:
    ESN_Configuration(std::vector<ValidationInfo*> *validationParameters)  {
        for(size_t b=0; b<validationParameters->size(); b++)
            if(validationParameters->at(b)->name.compare("Trial") == 0)
                trial = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("NR") == 0)
                nr = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("Sigma") == 0)
                sigma = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("scalingW") == 0)
                scalingW = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("W_connectivity") == 0)
                connectivityW = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("scalingWin") == 0)
                scalingWin = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("Win_connectivity") == 0)
                connectivityWin = *validationParameters->at(b)->get();
            else if(validationParameters->at(b)->name.compare("Lambda") == 0)
                lambda = *validationParameters->at(b)->get();
            
        learningResultIndex = RESULT_ADDRESS(trialList->getIdx(), trialSize, nrList->getIdx(), nrSize, sigmaList->getIdx(), sigmaSize,
                                                    connectWList->getIdx(), connectWSize, scalingWList->getIdx(), scalingWSize,
                                                    connectWinList->getIdx(), connectWinSize, scalingWinList->getIdx(), scalingWinSize,
                                                    lambdaList->getIdx(), lambdaSize);    
    }

    ~ESN_Configuration() {
        for(ValidationInfo *v : *validationParameters)
            delete v;
    }

};
void print()
{
 std::cout << "Trial= " << trial << "   " 
<< " NR= " << nr << "   " 
<< " Sigma= " << sigma << "   " 
<< " scalingW= " << scalingW << "   " 
<< " W_connectivity= " << connectivityW << "   " 
<< " scalingWin= " << scalingWin << "   " 
<< " Win_connectivity= " << connectivityWin << "   " 
<< " Lambda= " << lambda << "  

};

double trial;
double nr;
double sigma;
double scalingW;
double connectivityW;
double scalingWin;
double connectivityWin;
double lambda;

class TaskDispatcher:
public ff::ff_node
{
public:
 TaskDispatcher(ff::ff_loadbalancer* lb) 
{
 validationParameters = new std::vector<ValidationInfo*>();
 initialized = false;
 terminate = false;
 this->lb = lb;
 hyperParametersGenerations(validationParameters);
 setValidationParameterAndResultDatastruct(validationParameters);
 ... connectWList->print();
 scalingWinList->print(); 
connectWinList->print();
 lambdaList->print();
}

int svc_init() {
return 0; }

void* svc(void* task) 
{ 
 ESN_Configuration *esnc = NULL ;
"}
}
unsigned int worker;

// The initial task arrives
if (task == NULL)
{

    // State initialization
    active_workers = 0;
    while (active_workers < lb->getnworkers() && (esnc = nextTask()) != NULL)
    {
        lb->ff_send_out_to((void*) esnc, active_workers++);
        for (worker = active_workers; worker < lb->getnworkers(); worker++)
            lb->ff_send_out_to(EOS, worker);
    }

    // a computed task arrives from a worker: generate another task
    else
    {
        if (!terminate)
        {
            esnc = nextTask();
            if (esnc == NULL)
            {
                terminate = true;
                for (worker = 0; worker < active_workers; worker++)
                    lb->ff_send_out_to(EOS, worker);
            }
            else
                lb->ff_send_out_to((void*) esnc, lb->get_channel_id());
        }
    }

    return GO_ON;
}

void svc_end() { }

private:

    std::vector<ValidationInfo*> *validationParameters;
    bool initialized, terminate;
    ff::ff_loadbalancer* lb;

    unsigned int active_workers;

    int nextTask() {
        int counter;
        double * next;
        if (!initialized)
        {
            // at least one element exists
            for (size_t ga = 0; ga < validationParameters->size(); ga++)
                validationParameters->at(ga)->next();
            initialized = true;
        }

        // a computed task arrives from a worker: generate another task
        if (worker < validationParameters->size() - 1)
            worker++;
        else
            worker = 0;

        // State initialization
        while (active_workers < validationParameters->size() - 1)
            if ((next = validationParameters->at(worker)) != NULL)
                // The initial task arrives
                ++active_workers;
            else
                active_workers = 0;

        if (next == NULL)
            return NOTINIT;
    }
else
    {       counter = validationParameters−>size()−1;      next=NULL;
    while (next== NULL  && counter>=0) ...  Nr = (int) esnc−>nr;
    Reservoir rsv(Nr, Ni, degree);
21 apr 15 14:27 Pagina 4/8parallelFullProcedureOnDemand.cpp
double scalingWin = esnc->scalingWin;
double Win_connectivity = esnc->connectivityWin;

double scalingW = esnc->scalingW;
double W_connectivity = esnc->connectivityW;
double sigma = esnc->sigma;

Sparse<double> *spW = new Sparse<double>(rsv.W, Nr, Nr);
free(rsv.W); rsv.W = NULL;

double *stateTR = NULL, *stateTS = NULL;
stateComputation(&rsv, spW, TR, &stateTR);

int targetDimension = 1;
// targets values are read in the constructor
linearRegressionTichonov* lrt = new linearRegressionTichonov();
lrt->setLambda(lambda);
lrt->train(stateTR, targetTR, TR->size(), Nr+1, targetDimension);
singleExtim(stateTR, targetTR, TR->size(), Nr+1, lrt, "TR",
 wholeValidationResults + esnc->learningResultIndex);
free(stateTR);
stateComputation(&rsv, spW, TS, &stateTS);
rsv.stateDestroy(); spW->setDestroy();
delete spW;

singleExtim(stateTS, targetTS, TS->size(), Nr+1, lrt, "TS",
 wholeValidationResults + esnc->learningResultIndex);
delete lrt;
return task;
}
double * targetTR;
  double * targetTS;
  bool  rootStateMapping;
  void  stateComputation( Reservoir* rsv, Sparse<double> *spW, std::vector<int> *partition,
  double** finalState){
      int Nr = rsv->Nr;
      double* states;
      double* stateMappingPointer1;
      double* finalMappedState1 = (double*) malloc(partition->size()*(Nr+1) * sizeof(double));
      int idx;
      int offset = 1;
      for(it=partition->begin(), idx=0; it!=partition->end(); it++, idx++) {
          states = rsv->computeTreeState(tds, *it, spW); stateMappingPointer1 = finalMappedState1 + idx*(Nr+1);
          if(!rootStateMapping)
              StateMapping::meanStateMapping(states, tds->getSample(*it)−>treeDim+1, rsv->Nr,
              stateMappingPointer1, offset);
          else
              StateMapping::superSourceStateMapping(states,tds->getSample(*it)−>treeDim+1, rsv->Nr,
              stateMappingPointer1, offset);
          stateMappingPointer1[Nr] = 1.0;
      }
      *finalState = finalMappedState1;  }

double singleEstimation(double* state, double* target, int sampleNumb, int featureDim, linearRegressionTichonov* readout,
    const char* datasetName, LearningResult** result)  {
    double predictedValue;
    double stat1_diff, stat1_aggr = 0;
    double stat2_diff, stat2_aggr = 0;
    for(int idx=0; idx<sampleNumb; idx++) {
        readout->prediction(state+idx*featureDim, &predictedValue);
        stat1_diff = target[idx] − predictedValue;
        if(stat1_diff<0) stat1_diff*=−1;
        stat1_aggr += stat1_diff;
        stat2_diff = stat1_diff/target[idx];
        if(stat2_diff<0) stat2_diff*=−1;
    }
    return stat1_aggr/sampleNumb;  
}
stat2_aggr += stat2_diff;

if (result != NULL) // result is required
{
    if (*result == NULL)
    {
        (*result) = new LearningResult();
        (*result)->lambda = readout->getLambda();
    }
    if (datasetName)
    {
        if (strcmp(datasetName,"TS") == 0) // Training set
        {
            (*result)->resultTs = stat1_aggr / sampleNumb;
            (*result)->ratioTs = stat2_aggr / sampleNumb;
        }
        else // Test set
        {
            (*result)->resultTr = stat1_aggr / sampleNumb;
            (*result)->ratioTr = stat2_aggr / sampleNumb;
        }
    }
    return stat1_aggr/sampleNumb;
}

return stat1_aggr/sampleNumb;

}
```cpp
int parDegree = atoi(argv[4]);
if (parDegree<=0) parDegree = 1;
TreeDataset* tds = new TreeDataset(argv[1]);
std::vector<int>* TR = new std::vector<int>();
std::vector<int>* TS = new std::vector<int>();
fillVector(argv[2], TR);
fillVector(argv[3], TS);
std::vector<ff::ff_node*>* workers;  
ff::ff_farm<>* ff_f = new ff::ff_farm<>();
TaskDispatcher* td = new TaskDispatcher(ff_f->getlb());
ff::ff_node* worker;
for (int i=0; i<parDegree; i++)
{
    worker = (ff::ff_node*) new StateComputationStage(tds, TR, TS);
    workers.push_back(worker);
}
ff_f->add_emitter(td);
ff_f->wrap_around();
ff_f->set_scheduling_on_demand();
ff_f->add_workers(workers);
Timer time;  
time.tic();
ff_f->run_and_wait_end();
std::cout << "Completion Time " << time.toc() << "\n";
```

```cpp
#include <vector>
#include <string>
#include <iostream>

class ValidationInfo {
public:
    std::string name;
    std::vector<double> *possibleValues;

    ValidationInfo(std::string name, std::vector<double> *possibleValues) {
        this->name = name;
        this->possibleValues = possibleValues;
        this->currentIndex = -1;
    }

    double* next(bool resetVal = false) {
        if (++currentIndex < possibleValues->size())
            return &(possibleValues->at(currentIndex));
        reset(resetVal);
        return NULL;
    }

    double* get() {
        if (currentIndex >= possibleValues->size())
            return NULL;
        return &(possibleValues->at(currentIndex));
    }

    double* get(int idx) {
        if (idx < 0 || (unsigned int)idx >= possibleValues->size())
            return NULL;
        return &(possibleValues->at(idx));
    }

    int getIdx() {
        return currentIndex;
    }

    int size() {
        return possibleValues->size();
    }

    void set(int idx, double value) {
        possibleValues->at(idx) = value;
    }

    void reset(bool resetVal) {
        if (!resetVal)
            currentIndex = -1;
    }
};
```

template<>

void print()
{
    size_t g;
    std::cout << name << "\[
    \[
    \]
    for (g=0; g<possibleValues->size()-1; g++)
        std::cout << possibleValues->at(g) << ",";
    if (g<possibleValues->size())
        std::cout << possibleValues->at(g);
    std::cout << "\]
    \]
};
#define LINEAR(X,Y,Z) ( ((X) * (Y)) + (Z) )

#define WIN_ADDR(sparWin, scalWin, NscalWin, inner, Ninner) ( LINEAR(sparWin, NscalWin * Ninner, \ 
LINEAR(scalWin, Ninner, inner)) )

#define W_ADDR(rho, sparW, NsparW, scalW, NscalW, inner, Ninner) ( LINEAR(rho, NsparW * NscalW * Ninner, \ 
LINEAR(sparW, NscalW * Ninner , \ 
LINEAR(scalW, Ninner, inner))) )

#define READOUT_ADDR(lambda, inner, Ninner) ( LINEAR(lambda, Ninner, inner) )

#define RESULT_ADDRESS(trial, Ntrial, Nr, NNr, rho, Nrho, sparW, NsparW, scalW, NscalW, sparWin, NsparWin, scalWin, NscalWin, lambda, Nlambda) ( LINEAR( Nr, ( (Nrho * NsparW * NscalW)*(NsparWin * NscalWin)*(Ntrial * Nlambda) ), \ 
W_ADDR(rho, sparW, NsparW, scalW, NscalW, \ 
WIN_ADDR(sparWin, scalWin, NscalWin, \ 
READOUT_ADDR(lambda, \ 
trial, Ntrial ), \ 
(Ntrial * Nlambda) ), \ 
(NsparWin, NscalWin)*(Ntrial * Nlambda) ) )

class LearningResult {
public:
  double lambda;
  double resultTr, resultTr_over, resultTr_under;
  double resultTs, resultTs_over, resultTs_under;
  double ratioTr, ratioTr_over, ratioTr_under;
  double ratioTs, ratioTs_over, ratioTs_under;
};

LearningResult** wholeValidationResults = NULL;
HyperParametersGenerations(std::vector<ValidationInfo*> *vals)
{
    std::vector<double>* possibleValues;
    //#### Trial ####//
    possibleValues = new std::vector<double>();
    for (int trial = 0; trial < 5; trial++) possibleValues->push_back(trial);
    vals->push_back(new ValidationInfo("Trial", possibleValues));
    //#### Nr ####//
    possibleValues = new std::vector<double>();
    possibleValues->push_back(750); possibleValues->push_back(500); possibleValues->push_back(250); possibleValues->push_back(100);
    vals->push_back(new ValidationInfo("NR", possibleValues));
    //#### Sigma ####//
    possibleValues = new std::vector<double>();
    possibleValues->push_back(0.1); possibleValues->push_back(0.4); possibleValues->push_back(0.7); possibleValues->push_back(1); possibleValues->push_back(3); possibleValues->push_back(6); possibleValues->push_back(9);
    vals->push_back(new ValidationInfo("Sigma", possibleValues));
    //#### scalingW ####//
    possibleValues = new std::vector<double>();
    possibleValues->push_back(0.01); possibleValues->push_back(0.1); possibleValues->push_back(1);
    vals->push_back(new ValidationInfo("scalingW", possibleValues));
    //#### W_connectivity ####//
    possibleValues = new std::vector<double>();
    possibleValues->push_back(0.05); possibleValues->push_back(0.1);
}
possibleValues.push_back(0.2); vals.push_back(new ValidationInfo("W_connectivity", possibleValues));

//#### scalingWin ####//
possibleValues = new std::vector<double>();
possibleValues.push_back(0.01); possibleValues.push_back(0.1); possibleValues.push_back(1.0); vals.push_back(new ValidationInfo("scalingWin", possibleValues));

//#### Win_connectivity ####//
possibleValues = new std::vector<double>();
//possibleValues.push_back(0.5);
possibleValues.push_back(1.0); vals.push_back(new ValidationInfo("Win_connectivity", possibleValues));

//#### Lambda ####//
possibleValues = new std::vector<double>();
possibleValues.push_back(0.05); possibleValues.push_back(0.1); possibleValues.push_back(0.15); possibleValues.push_back(0.20); possibleValues.push_back(0.25); vals.push_back(new ValidationInfo("Lambda", possibleValues));

ValidationInfo* getParameters(std::vector<ValidationInfo*> *vals, const char* name)
{
    std::vector<ValidationInfo*>::iterator it;
    for (it=vals->begin(); it!=vals->end(); it++)
        if ((*it)->name.compare(name) == 0)
            break;
    if (it!=vals->end())
        return (*it);
    return NULL;
}

void setValidationParameterAndResultDatastruct(std::vector<ValidationInfo*> *validationParameters)
{
    // Generate hyper-parameters
    //validationParameters = new std::vector<ValidationInfo*>();
    //hyperParametersGenerations(validationParameters);
    // Extract validation parameters
    trialList = getParameters(validationParameters, "Trial");
    nrList = getParameters(validationParameters, "NR");
    sigmaList = getParameters(validationParameters, "Sigma");
    scalingWList = getParameters(validationParameters, "scalingW");
    connectWList = getParameters(validationParameters, "W_connectivity");
}
scalingWinList = getParameters(validationParameters, "scalingWin");
connectWinList = getParameters(validationParameters, "connectWin");
lambdaList = getParameters(validationParameters, "lambda");
trialSize = trialList->size(); nrSize = nrList->size(); sigmaSize = sigmaList->size();
scalingWSize = scalingWinList->size(); connectWSize = connectWinList->size(); lambdaSize = lambdaList->size();

int size = nrSize * sigmaSize * scalingWSize * connectWSize * scalingWinSize * connectWinSize * lambdaSize * trialSize;
wholeValidationResults = (LearningResult**) malloc(sizeof(LearningResult*) * size);
#include <stdio.h>

#define THESIS 0

#include "ff/mdf.hpp"

#include "validationInfo.cpp"

#include "commonValidation.hpp"

#include "MDF_operations.cpp"

#include "../time.h"

#include "blockedVector.hpp"

#include "statistics.cpp"

// The TAG is set in the lowest id part
#define TAG_W_MATRIX 0x2
#define TAG_STATES 0x4
#define TAG_LEARN 0x6
#define TAG_OTHER 0x0
#define TAG_Wi (TAG_W_MATRIX | 0x0)
#define TAG_W (TAG_W_MATRIX | 0x1)
#define TAG_TR (TAG_STATES | 0x0)
#define TAG_TS (TAG_STATES | 0x1)
#define TAG_LRT (TAG_LEARN | 0x0)
#define TAG_RESULT (TAG_LEARN | 0x1)

#define MAKE_PARAM_INFO(PI, TAG, DIR) (PI).tag=(uintptr_t)(TAG); PI.dir=(DIR);

void fillVector(char* path, std::vector<int>* vect)
{
    int tmp;
    FILE* fp = fopen(path, "r");
    while (fscanf(fp, "%d\n", &tmp) == 1)
        vect->push_back(tmp);
}

struct StartupStructure
{
    TreeDataset *tds;
    std::vector<int> *TR, *TS;
    int Ni, degree;

    // be careful for the other identifier (ie. all 0s)
    #define TAG_LEARN_REDUCE ( TAG_OTHER | 0x1 )

    #define MAKE_PARAM_INFO(PI, TAG, DIR) (PI).tag=(uintptr_t)(TAG); PI.dir=(DIR);
};

```cpp
void taskGenerator(
    struct StartupStructure *start)
{
    // init vector
    vectW = new BlockedVector<Sparse<double>*, 50>();
    vectWi = new BlockedVector<Dense*, 50>();
    vectTr = new BlockedVector<Dense*, 50>();
    vectTs = new BlockedVector<Dense*, 50>();
    vectLrt = new BlockedVector<linearRegressionTichonov*, 50>;
    trialList->print();  nrList->print();   sigmaList->print();
    scalingWList->print(); connectWList->print();  scalingWinList->print();
    connectWinList->print();  lambdaList->print();

    int nr;
    double sigma, scalW, connW, scalWi, connWi, lambda;
    // Data Identifier
    uintptr_t wiIdx = 0, wIdx = 0, currWiIdx, currWIdx;
    uintptr_t stateTrIdx = 0, stateTsIdx = 0, currTrIdx, currTsIdx;
    uintptr_t trainIdx = 0, resultIdx = 0, currTrainIdx, currResIdx;
    #if THESIS == 0
    uintptr_t reduceIdx = 0, currReduceIdx;
    #endif
    ff::ff_mdf *mdf = *(start->mdf);
    std::vector<ff::param_info> param;
    std::vector<ff::param_info> reduce;
    #if THESIS == 0
    std::vector<ff::param_info> wBarrier;
    #endif
    ff::param_info pi1, pi2, pi3, pi4;
    LearningResult** wholeValidationResults = start->wholeValidationResults;
    while (trialList->next() != NULL)
```
while (nrList->next() != NULL) {
    nr = (int)(*nrList->get());
    while (connectWList->next())
        while (sigmaList->next())
            while (scalingWList->next())
                // MDF_Operations::init_W(nr, start->degree, sigma, scalW, connW, &W_Matrix);
                currWIdx = (uintptr_t)((TAG_W)|(wIdx<<3));
                MAKE_PARAM_INFO(pi1, currWIdx, ff::OUTPUT);
                MAKE_PARAM_INFO(pi2, 0, ff::INPUT);
                param.push_back(pi2);
                param.push_back(pi1);
                vectW->push_back(NULL);
                mdf->AddTask(param, MDF_Operations::init_W, nr, start->degree, sigma, scalW, connW, &vectW->back());
                param.clear();
    while (connectWinList->next())
        while (scalingWinList->next())
            // MDF_Operations::init_Wi(nr, start->Ni, scalWi, connWi, &Wi_Matrix);
            currWiIdx = (uintptr_t)((TAG_Wi)|(wiIdx<<3));
            MAKE_PARAM_INFO(pi1, currWiIdx, ff::OUTPUT);
            MAKE_PARAM_INFO(pi2, 0, ff::INPUT);
            param.push_back(pi2);
            param.push_back(pi1);
            vectWi->push_back(NULL);
            mdf->AddTask(param, MDF_Operations::init_Wi, nr, start->Ni, scalWi, connWi, &vectWi->back());
            param.clear();
    // MDF_Operations::stateComputation(start->tds, start->TR, W_Matrix, Wi_Matrix, &trState);
    currTrIdx = (uintptr_t)((TAG_TR)|(stateTrIdx<<3));
    MAKE_PARAM_INFO(pi1, currWiIdx, ff::INPUT);
    MAKE_PARAM_INFO(pi2, currTrIdx, ff::OUTPUT);
    param.push_back(pi1);
    param.push_back(pi2);
    param.push_back(pi3);
    vectTr->push_back(NULL);
    mdf->AddTask(param, MDF_Operations::stateComputation, start->tds, start->TR, &vectW->back(), &vectWi->back(), &vectTr->back(), true);
    // rootStateMapping
}
MAKE_PARAM_INFO(pi2, currWIdx, ff::INPUT);      MAKE_PARAM_INFO(pi3, currTsIdx, ff::OUTPUT);      param.push_back(pi1); param.push_back(pi2); param.push_back(pi3);       vectTs−>push_back(NULL);       mdf−>AddTask(param, MDF_Operations::stateComputation, start−>tds, start−>TS, &vectW−>back(), &vectWi−>back(), &vectTs−>back(), true);

  rootStateMapping

param.clear();      wiIdx++;      BlockedVector<LearningResult*, 50> *vectRes = new BlockedVector<LearningResult*, 50>();

while (lambdaList−>next())
{
lambda = *lambdaList−>get();      //MDF_Operations::train(trState, start−>targetTR, lambda, &lrt);
      currTrainIdx = (uintptr_t)((TAG_LRT)|(trainIdx<<3));MAKE_PARAM_INFO(pi1, currTrIdx, ff::INPUT);MAKE_PARAM_INFO(pi2, currTrainIdx, ff::OUTPUT);param.push_back(pi1); param.push_back(pi2); vectLrt−>push_back(NULL);
      mdf−>AddTask(param, MDF_Operations::train, &vectTr−>back(), start−>targetTR, lambda, &vectLrt−>back());param.clear();
      //MDF_Operations::estimation(trState, tsState, start−>targetTR, start−>targetTS, readoutMatrix);
      currResIdx =  (uintptr_t)((TAG_RESULT)|(resultIdx<<3));;MAKE_PARAM_INFO(pi1, currTrainIdx, ... currResIdx, ff::OUTPUT);param.push_back(pi1); param.push_back(pi3); param.push_back(pi4); vectRes−>push_back(NULL);
      LearningResult** tmpResult = wholeValidationResults +   RESULT_ADDRESS(trialList−>getIdx(), trialSize, nrList−>getIdx(), nrSize, sigmaList−>getIdx(), sigmaSize, connectWList−>getIdx(), connectWSize, scalingWList−>getIdx(), scalingWSize, connectWinList−>getIdx(), connectWinSize, scalingWinList−>getIdx(), scalingWinSize, lambdaList−>getIdx(), lambdaSize);
      *tmpResult = NULL;
      mdf−>AddTask(param, MDF_Operations::estimation, &vectTr−>back(), start−>targetTR, start−>TR_discriminationD, &vectTs−>back(), start−>targetTS, start−>TS_discriminationD, &vectLrt−>back(),  tmpResult);
      param.clear();
      // reducing procedure input
      MAKE_PARAM_INFO(pi1, currResIdx, ff::INPUT);reduce.push_back(pi1);trainIdx++;

}}
resultIdx++;
// delete lrt;
// readoutMatrix->setDestroy(); delete readoutMatrix;

// trState->setDestroy(); delete trState;
// tsState->setDestroy(); delete tsState;
#if THESIS == 0
currReduceIdx = (uintptr_t)((TAG_LEARN_REDUCE)|(reduceIdx<<3));
MAKE_PARAM_INFO(pi3, currReduceIdx, ff::OUTPUT);
reduce.push_back(pi3);
#endif
mdf->AddTask(reduce, MDF_Operations::reduceLearning, nr, sigma, scalW, connW, scalWi, connWi, vectRes, &vectWi->back(), &vectTr->back(), &vectTs->back());
reduce.clear();
#if THESIS == 0
MAKE_PARAM_INFO(pi1, currReduceIdx, ff::INPUT);
wBarrier.push_back(pi1);
reduceIdx++;
#endif
stateTsIdx++; stateTrIdx++;    }
#if THESIS == 0
mdf->AddTask(wBarrier, MDF_Operations::destroy_W, &vectW->back()); wBarrier.clear();
#endif
wIdx++;  }
}  }
//trial parenthesis
//it discriminates over parallelism samples
//it requires TR be a super-set of TR_over

discriminate_over(
char* baseFileName, std::vector<int> *sampleIdxs, bool* discriminationArray)
{
std::vector<int>::iterator it, it2;
int idx;
std::string* over_par = new std::string(baseFileName);
over_par->append(".over");
std::vector<int> *DS_over = new std::vector<int>();

21 apr 15 14:26 Pagina 5/10
mdfFullProcedure.cpp
mdfFullProcedure.cpp
for (it = sampleIdxs->begin(), it2 = DS_over->begin(), idx = 0; it != sampleIdxs->end() && it2 != DS_over->end(); it++, idx++) {
    if (*it == *it2) {
        discriminationArray[idx] = true;
        it2++;
    } else {
        discriminationArray[idx] = false;
    }
}

if (it2 == DS_over->end()) {
    while (it != sampleIdxs->end()) {
        discriminationArray[idx++] = false;
        it++;
    }
}

delete over_par;
delete DS_over;

} // void getStatistics(LearningResult** wholeValidationResults)
stat_ratioTr.resetStatistics();
stat_ratioTs.resetStatistics();
stat_ratioTr_over.resetStatistics();
stat_ratioTs_over.resetStatistics();
stat_ratioTr_under.resetStatistics();
stat_ratioTs_under.resetStatistics();

for (int trial=0; trial<trialSize; trial++) {
  printf("<-> %d %lf %lf %lf %lf %lf %lf %d 
[ %lf %lf %lf ] [ %lf %lf %lf ]
", *(nrList->get(nr)), *(sigmaList->get(sigma)), *(scalingWList->get(scalingW)), *(connectWList->get(connectW)), *(scalingWinList->get(scalingWin)), *(connectWinList->get(connectWin)), *(lambdaList->get(lambda)), *(trialList->get(trial)), wholeValidationResults[i]->resultTr, wholeValidationResults[i]->resultTr_over, ...
      wholeValidationResults[i]->ratioTs_over, wholeValidationResults[i]->ratioTs_under);
  stat_resultTr.addElement( wholeValidationResults[i]->resultTr );
  stat_resultTs.addElement( ...
      ...
  stat_ratioTs_under.addElement( wholeValidationResults[i]->ratioTs_under );
  i++;}
printf("<=> %d %lf %lf %lf %lf %lf %lf 
[ %lf %lf %lf ] [ %lf %lf %lf ]
", *(nrList->get(nr)), *(sigmaList->get(sigma)), *(scalingWList->get(scalingW)), *(connectWList->get(connectW)), *(scalingWinList->get(scalingWin)), *(connectWinList->get(connectWin)), *(lambdaList->get(lambda));

// Mean Values
printf("[ %lf %lf %lf ]
", stat_resultTr.getMeanValue(), stat_resultTr_over.getMeanValue(), stat_resultTr_under.getMeanValue());
printf("< %lf %lf %lf ]
", stat_ratioTr.getMeanValue(), ...
      ...
  stat_ratioTs.getMeanValue());
}
// Variance values
printf("[ %lf %lf %lf > ", stat_resultTr.getStandardDeviation(),
stat_resultTr_over.getStandardDeviation(), stat_resultTr_under.getStandardDeviation());
printf("< %lf %lf %lf ]
", stat_ratioTr.getStandardDeviation(),
stat_ratioTr_over.getStandardDeviation(), stat_ratioTr_under.getStandardDeviation());
}

int main (int argc, char * argv[])
{
    if (argc!=5)
    {
        fprintf(stderr, "Error in number of parameters
Correct input parameter %s <dataset> <TR> <TS> <parDegree>
", *argv);
        return -1;
    }
    int parDegree = atoi(argv[4]);
    // Generate hyper-parameters
    std::vector<ValidationInfo*> *validationParameters =
new std::vector<ValidationInfo*>();
    hyperParametersGenerations(validationParameters);
    // Extract validation parameters
    trialList = getParameters(validationParameters, "Trial");
nrList = getParameters(validationParameters, "NR");
sigmaList = getParameters(validationParameters, "Sigma");
scalingWList = getParameters(validationParameters, "scalingW");
connectWList = getParameters(validationParameters, "W_connectivity");
scalingWinList = getParameters(validationParameters, "scalingWin");
connectWinList = getParameters(validationParameters, "Win_connectivity");
lambdaList = getParameters(validationParameters, "Lambda");
int size = nrSize * sigmaSize * scalingWSize * connectWSize * scalingWinSize * connectWinSize * lambdaSize * trialSize;
LearningResult** wholeValidationResults = (LearningResult**) malloc(sizeof(LearningResult*) * size);

// Load Dataset Informations (Done before the computation starts)
int idx=0;
std::vector<int>::iterator it;
TreeDataset tds(argv[1]);
std::vector<int>* TR = new std::vector<int>();
std::vector<int>* TS = new std::vector<int>();
fillVector(argv[2], TR);
fillVector(argv[3], TS);
#if THESIS==1
bool TR_discrimination[TR->size()];
bool TS_discrimination[TS->size()];
discriminate_over(argv[2], TR, TR_discrimination);
discriminate_over(argv[3], TS, TS_discrimination);
#endif
int Ni = tds.labelDim, degree = tds.maxArity;
double targetsTR[TR->size()];
for (it=TR->begin(), idx=0; it!=TR->end(); it++, idx++)
    targetsTR[idx] = tds.getSample(*it)->target;
double targetsTS[TS->size()];
for (it=TS->begin(), idx=0; it!=TS->end(); it++, idx++)
    targetsTS[idx] = tds.getSample(*it)->target;
Dense *targetTR = new Dense(targetsTR, TR->size(), 1);
Dense *targetTS = new Dense(targetsTS, TS->size(), 1);
// MDF Engine starts
int queueLen = 300;
ff::ff_mdf *mdf;
struct StartupStructure start = { &tds, TR, TS, Ni, degree, targetTR, targetTS, NULL, NULL, validationParameters, wholeValidationResults, &mdf }; 

#endif 

mdf = new ff::ff_mdf( &taskGenerator, &start, queueLen, parDegree ); 

Timer time; time.tic(); mdf->run_and_wait_end(); printf("Completion Time %lu\n", time.toc()); 

#if THESIS == 1 
getStatistics( wholeValidationResults ); 
#endif 

}
```cpp
#include <vector>
#include <pthread.h>
#include "../reservoir.h"
#include "../readout.h"
#include "../tesnState.cpp"
#include "blockedVector.hpp"

static pthread_spinlock_t spinLock;

class MDF_Operations{
public:
    static void init_W(int Nr, int degree, double sigma, double scalW, double connW, Sparse<double>** W)
    {
        Reservoir rsv(Nr, 1, degree);
        rsv.initialize_W(scalW, connW, sigma);
        *W = new Sparse<double>(rsv.W, Nr, Nr);
        (*W)->setDestroy();
        return;
    }

    static void init_Wi(int Nr, int Ni, double scalWin, double connWin, Dense** Wi)
    {
        Reservoir rsv(Nr, Ni, 1);
        rsv.initialize_Win(scalWin, connWin);
        rsv.matrixUndestroy();
        *Wi = new Dense(rsv.Win, Nr, Ni+1);
        (*Wi)->setDestroy();
        return;
    }

    static void destroy_W(Sparse<double>** W)
    {
        if (W && *W)
            delete *W;
    }

    static void stateComputation(TreeDataset* tds, std::vector<int> *partition, Sparse<double> **spW, Dense **Wi, Dense** finalState, bool rootStateMapping=false)
    {
    }
};
```
int Nr = (*spW)->getN();
int Ni = (*Wi)->getM() - 1;

Reservoir rsv(Nr, Ni, tds->maxArity);
rsv.Win = (*Wi)->getMatrix();
rsv.matrixUndestroy();

double* states;
double* stateMappingPointer1;

double* finalMappedState1 = (double*) malloc(partition->size()*(Nr+1) * sizeof(double));
int idx;
int offset = 1;

std::vector<int>::iterator it;
for (it=partition->begin(), idx=0; it!=partition->end(); it++, idx++)
{
    states = rsv.computeTreeState(tds, *it, *spW);
    stateMappingPointer1 = finalMappedState1 + idx*(Nr+1);
    if (!rootStateMapping)
        StateMapping::meanStateMapping(states, tds->getSample(*it)->treeDim+1, rsv.Nr, stateMappingPointer1, offset);
    else
        StateMapping::superSourceStateMapping(states, tds->getSample(*it)->treeDim+1, rsv.Nr, stateMappingPointer1, offset);
    // Free is no need because the buffer is reused over the iterations
    stateMappingPointer1[rsv.Nr] = 1.0;
    lrt Bias
}

*finalState = new Dense(finalMappedState1, partition->size(), Nr+1);
(*finalState)->setDestroy();
rsv.stateDestroy();
}

static void train(Dense** finalState, Dense *target, double lambda, linearRegressionTichonov** lrt)
{
    double*finalMappedState = (*finalState)->getMatrix();
    int sampleNumber = (*finalState)->getN();
    int featureNumber = (*finalState)->getM();
    double*targetTR = target->getMatrix();
    int targetNumber = target->getM();
    *lrt = new linearRegressionTichonov();
    (*lrt)->setLambda(lambda);
    (*lrt)->train(finalMappedState, targetTR, sampleNumber, featureNumber, targetNumber);
}
static inline double singleExtim(Dense **state, Dense *target, bool * discrimination, linearRegressionTichonov** readout, const char* datasetName, LearningResult** result)
{
    double predictedValue;
    double stat1_diff, stat1_aggr = 0, stat1_aggr_over = 0;
    double stat2_diff, stat2_aggr = 0, stat2_aggr_over = 0;
    int sampleOverNumb = 0;
    double*stateM = (*state)->getMatrix();
    int sampleNumb = (*state)->getN();
    int featureDim = (*state)->getM();
    double*targetM = target->getMatrix();
    for (int idx=0; idx<sampleNumb; idx++)
    {      (*readout)->prediction(stateM+idx*featureDim, &predictedValue);
        stat1_diff = targetM[idx] − predictedValue;
        if (stat1_diff<0) stat1_diff*=−1;
        stat1_aggr += stat1_diff;
        stat2_diff = stat1_diff/targetM[idx];
        if (stat2_diff<0) stat2_diff*=−1;
        stat2_aggr += stat2_diff;
        if (discrimination!=NULL && discrimination[idx])
        {
            stat1_aggr_over += stat1_diff;
            stat2_aggr_over += stat2_diff;
            sampleOverNumb++;
        }
    }
    if (result!=NULL)
    { // result is required
        if (*result==NULL)
        { (*result) = new LearningResult();
            (*result)->lambda = (*readout)->getLambda();
        }
        if (datasetName)
        { if (strcmp(datasetName,"TS")==0) // Test set
            { // GLOBAL STATISTICS
                (*result)->resultTs = stat1_aggr / sampleNumb;
                // GLOBAL STATISTICS
                (*result)->resultTs = stat1_aggr / sampleNumb;
            }
        }
    }
}
```c
    (*result)−>resultTs = stat2_aggr / sampleNumb;
    //printf("<TS> %lf %lf\n", (*result)−>resultTs, (*result)−>ratioTs);
    if(discrimination!=NULL) {
        // OVER STATISTICS
        if(sampleOverNumb!=0) {
            (*result)−>resultTs_over = stat1_aggr_over / sampleOverNumb;
            (*result)−>ratioTs_over = stat2_aggr_over / sampleOverNumb;
        } else {
            (*result)−>resultTs_over = 0;
            (*result)−>ratioTs_over = 0;
        }
        // UNDER STATISTICS
        if(sampleNumb − sampleOverNumb != 0) {
            (*result)−>resultTs_under = (stat1_aggr − stat1_aggr_over) / (sampleNumb − sampleOverNumb);
            (*result)−>ratioTs_under = (stat2_aggr − stat2_aggr_over) / (sampleNumb − sampleOverNumb);
        } else {
            (*result)−>resultTs_under = 0;
            (*result)−>ratioTs_under = 0;
        }
    } else {
        // Training set
        // GLOBAL STATISTICS
        (*result)−>resultTr = stat1_aggr / sampleNumb;
        (*result)−>ratioTr = stat2_aggr / sampleNumb;
        //printf("<TR> %lf %lf\n", (*result)−>resultTr, (*result)−>ratioTr);
        if(discrimination!=NULL) {
            // OVER STATISTICS
            if(sampleOverNumb!=0) {
                (*result)−>resultTr_over = stat1_aggr_over / sampleOverNumb;
                (*result)−>ratioTr_over = stat2_aggr_over / sampleOverNumb;
            } else {
                (*result)−>resultTr_over = 0;
                (*result)−>ratioTr_over = 0;
            }
            // UNDER STATISTICS
            if(sampleNumb − sampleOverNumb != 0) {
                (*result)−>resultTr_under = (stat1_aggr − stat1_aggr_over) / (sampleNumb − sampleOverNumb);
                (*result)−>ratioTr_under = (stat2_aggr − stat2_aggr_over) / (sampleNumb − sampleOverNumb);
            } else {
                (*result)−>resultTr_under = 0;
                (*result)−>ratioTr_under = 0;
            }
        }
    }
```
if (sampleNumb - sampleOverNumb != 0)
  {  (*result)−>resultTr_under = (stat1_aggr - stat1_aggr_over) / (sampleNumb - sampleOverNumb);
    (*result)−>ratioTr_under = (stat2_aggr - stat2_aggr_over) / (sampleNumb - sampleOverNumb);
  }
else  //No error in the absence of samples
  {  (*result)−>resultTr_under = 0;
    (*result)−>ratioTr_under = 0;
  }  

return stat1_aggr/sampleNumb;

static inline void estimation(Dense **state1, Dense *target1,
  bool *discrimination1, Dense **state2, Dense *target2,
  bool *discrimination2,  linearRegressionTichonov** readout, LearningResult** result)  {    (*result) = new LearningResult();    (*result)−>lambda = (*readout)−>getLambda();    singleExtim(state1, target1, discrimination1, readout, "TR", result);
    singleExtim(state2, target2, discrimination2, readout, "TS", result);
#if  THESIS == 0
    if (Wi && *Wi) delete  *Wi; 
#endif  }};

static void initSpinLock(){ pthread_spin_init(&spinLock, 0); }

static void reduceLearning(
  int nr,
  double sigma,
  double scalW,
  double connW,
  double scalWi,
  double connWi, BlockedVector<LearningResult*, 50> *vectResult, Dense** Wi, Dense** Tr, Dense** Ts)  {
    // deletion of the Wi matrix and the computed states for TR/TS
    if (Tr && *Tr) delete  *Tr;
    if (Ts && *Ts) delete  *Ts;
#if  THESIS == 0
    if (Wi && *Wi) delete  *Wi; 
#endif  }
#ifndef __TESN_BLOCKED_VECTOR
#define __TESN_BLOCKED_VECTOR

#include <stdio.h>
#include <vector>

template<typename T, int N>
class BlockedVector {
public:
    BlockedVector() noexcept {
        dataStruct = new std::vector<T*>(N);
        lastIndex = -1;
        vectorIndex = -1;
        extend();
    }

    void push_back(T value) {
        if (lastIndex == N - 1) {
            extend();
            lastIndex = -1;
        }
        dataStruct->at(vectorIndex)[++lastIndex] = value;
    }

    T& back() { return dataStruct->at(vectorIndex)[lastIndex]; }

    T& at(int index) {
        return dataStruct->at(index / N)[index % N];
    }

    int size() { return lastIndex + 1; }

private:
    std::vector<T*>* dataStruct;
    int lastIndex, vectorIndex;

    void extend() {
        dataStruct->push_back(new T[N]);
        vectorIndex++;
    }
};
#endif
Bibliography


