Neural Computing and Applications (2018) 30:1715–1731
https://doi.org/10.1007/s00521-018-3517-y

REVIEW

Proposal of a declarative and parallelizable artificial neural network using the notification-oriented paradigm

Fernando Schütz[1] · João A. Fabro[2] · Adriano F. Ronszcka[1] · Paulo C. Stadzisz[1] · Jean M. Simão[1]

Received: 14 August 2017 / Accepted: 27 April 2018 / Published online: 4 June 2018

Abstract
Since the 1960s, artificial neural networks (ANNs) have been implemented and applied in various areas of knowledge. Most of these implementations had their development guided by imperative programming (IP), usually resulting in highly coupled programs. Thus, even though intrinsically parallel in theory, ANNs do not easily take an effective distribution on multiple processors when developed under IP. As an alternative, the notification-oriented paradigm (NOP) emerges as a new programming technique. NOP facilitates the development of decoupled and distributed systems, using abstraction of knowledge through logical–causal rules, as well as the generation of an optimized code. Both features are possible by means of a notification-oriented inference process, which avoids structural and temporal redundancies in the logic–causal evaluations. These advantages are relevant to systems that have parts decoupled in order to run in parallel, such as ANN. In this sense, this work presents the development of a multilayer perceptron ANN using backpropagation training algorithm based on the concepts of a NOP implementation. Such implementation allows, transparently from high-level programming, parallel code generation that runs on multicore platforms. Furthermore, the solution based on NOP, when compared against the equivalent on IP, presents a high level of decoupling and explicit use of logic–causal elements, which are, respectively, useful to distribution, understanding and improvement of the application.

Keywords Notification-oriented paradigm · Artificial neural network · Backpropagation algorithm · Decoupled systems

1 Introduction

The process of developing systems that mimic intelligence features is a constant goal in the computer science research, even with the progress already obtained in some areas. Techniques for building models that represent human cognitive capacity and nature are part of the so-called artificial intelligence (AI) and computational intelligence (CI). Actually, advances in this area, in the early twenty-first century, moved beyond the academy and are becoming a reality in the industry [1–3].

An example of technique of AI and CI that presents interest to both academy and industry is the so-called artificial neural network (ANN). Studies involving ANNs started in the 1960s and kept advancing, thereby allowing achieving real applications. ANN has been applied as an important and innovative solution in fields such as time series forecasting, classification, and pattern recognition inclusively in effective contexts, such as industrial systems [4–7].

✉ Fernando Schütz
fernando@utfpr.edu.br

João A. Fabro
fabro@utfpr.edu.br

Adriano F. Ronszcka
ronszcka@alunos.utfpr.edu.br

Paulo C. Stadzisz
stadzisz@utfpr.edu.br

Jean M. Simão
jeansimao@utfpr.edu.br

[1] CPGEI (Graduate Program on Electrical Engineering and Industrial Informatics), UTFPR (Federal University of Technology), Curitiba, PR 80230-901, Brazil

[2] PPGCA (Graduate Program on Applied Computing), UTFPR (Federal University of Technology), Curitiba, PR 80230-901, Brazil

Improvements in digital computers and programming techniques have contributed to the application of such techniques in an effective way [5, 8]. However, algorithms that implement ANNs, running on sequential machines based on the Von Neumann architecture, do not achieve their inherent theoretical potential for parallel execution. Actually, even parallel architectures based on Von Neumann, such as multicore ones, do not allow achieving such parallelism in a substantial uncomplicated way [9–11].

Such problems happen due to coupling factors in actual programming techniques from the current software development paradigms [12–14]. Indeed, there is a lack of models and techniques that use the potential of parallel architecture, in a comprehensive and easy way [15]. This is worsened by the techniques of current programming paradigms that do not even assist in exploring the total capacity of the sequential processors [7].

Linhares et al. [16] state that current programming languages paradigms are neither well prepared to the parallel and distributed systems nor to the optimized processing. In short, this happens because source code developed in the current paradigms tends to generate coupled and redundant code. Under this assumption, Simão created the notification-oriented paradigm (NOP) [17, 18].

NOP is derived from previous works of notification-oriented control and inference [19]. Essentially, the NOP is based upon the concept of small, smart, and decoupled entities that collaborate by means of precise notifications. The inference process of NOP is innovative once it happens by means of active collaboration of notifier entities, which deal with factual and logic–causal knowledge [17].

NOP proposes a solution that, thanks to notification-oriented inference, optimizes logical–causal evaluation by means of artifacts (e.g., code) implicitly without structural and temporal redundancies. The avoidance of these redundancies allows, in a correct implementation, implicit decoupling (or minimal coupling) among the system entities. This facilitates parallelization and distribution of the processing tasks [10, 20].

Another important NOP feature, inherited from the declarative paradigm, is the ability to use knowledge abstraction about logical–causal rules. This characteristic facilitates the development of NOP systems at high-level programming [21].

In this context, this paper presents the construction of a backpropagation (BP) training algorithm for multilayer perceptron (MLP) ANNs using NOP. The aim is to demonstrate the NOP applicability for the implementation of ANNs with decoupled and parallelizable parts. In addition, corroborate that NOP allows making explicit the logical–causal knowledge of the MLP ANN, in order to adjust its behavior, thereby ratifying the possibility to define and program an ANN using a language with declarative properties. Finally, the feasibility of generating parallel multicore code in a transparent way is demonstrated.

The remaining sections are organized as follows: Sect. 2 presents NOP; Sect. 3 summarizes the theory of ANNs mainly focusing on the MLP architecture; Sect. 4 shows the development of ANNs in NOP, and Sect. 5 presents conclusions and perspective of future works.

1.1 Case of studies

In order to demonstrate the applicability of the NOP concepts in ANNs and to compare its features with respect to the ANN implemented under IP, experiments with the IRIS Dataset are developed in each paradigm. For the implementations, the C programming language and the FANN library [22] were used for IP, whereas the NOP Frameworks 2.0 and 3.0 were used for NOP.

In time, the work of Schütz et al. [23] is related to this research. In this context, Schütz et al. [23] presents the results of the first experiment to implement the training of an ANN MLP in NOP, using the NOP Framework 2.0, through the Backpropagation (BP) method. Finally, in such work, the XOR function was used for demonstration and testing purposes.

This paper applies to the IRIS Flower classification problem. There are 150 samples of input data in the dataset, with the measure of the length and width, in cm, of IRIS Flowers sepals and petals. From this set of input data, it is possible to rank the three subtypes: *Setosa*, *Virginica*, and *Versicolor*. The data set consists of 50 samples of each of three species [24–26].

2 Notification-oriented paradigm (NOP)

The notification-oriented paradigm (NOP) presents a new approach to build software, using a hybrid approach derived from both declarative and imperative paradigms. For the present work, it is important to introduce the concepts of NOP and to describe its materializations.

2.1 The NOP principles

The NOP allows in software development to represent logic–causal relationships through rules encapsulated in entities called *Rules*. Each *Rule* correlates *Attributes* and *Methods* of entities called *Facts Base Elements* (*FBE*). A *FBE* is an abstraction of an entity that must be represented in the system. An example of a *FBE* can be a neuron in an ANN system or a student in an academic control system. In turn, one *Rule* has two parts. One of them is called *Condition,* and the other is called *Action*. Such entities work

together to manipulate the causal knowledge of the *Rule*. Both *Condition* and *Action* of each *Rule* refer to a set of *FBEs* of the system [23, 27].

The *Condition* concerns to decisional part of its *Rule*. It correlates to the referenced *Attributes* of *FBEs* by means of entities called *Premises*. The *Action*, in turn, concerns to the execution of its *Rule*. It normally instigates services of these *FBEs* by means of entities called *Methods* [27]. Figure 1 presents an illustrative representation, in a logic–causal rule form of a *Rule* entity in NOP with its referenced *FBEs*.

The main feature of NOP is that all entities collaborate through an inference approach based on notifications in order to approve and activate the relevant *Rules* for execution. In this approach, each *FBE* entity notifies its factual knowledge to the other entities involved through reactive capabilities called notifications. Each *FBE* includes a set of *Attributes* who keep their states and have the ability to notify precisely just the interested entities, normally *Premises* of *Rule Conditions*. Figure 2 shows the class diagram of the NOP entities [27].

According to Simão et al. [18] in every pertinent change in its value, each *Attribute* precisely notifies just the very concerned *Premises*, which evaluate such state using logical operators. If the logical value or state of one or more *Premises* modifies, each *Premise* precisely notifies just the very concerned *Conditions*, in order to their logic states to be re-evaluated. Each *Condition* that turns to be true notifies its respective *Rule* as approved.

Still, when two or more approved *Rules* refer to the same *FBE* and require exclusive access to this *FBE*, a conflict occurs. In order to resolve such conflicts, there are different ways, such as *Rule* priorities. Each approved *Rule*, after resolution of eventual conflict, can notify its *Action* to execute. In turn, the *Action* notified invokes by notification its *Instigations*. Then, each *Instigation* invokes by notification concerned *Methods* to be executed. Such *Methods* may change *Attributes*, thereby restarting the notifications cycle. Figure 3 shows the collaboration mechanism

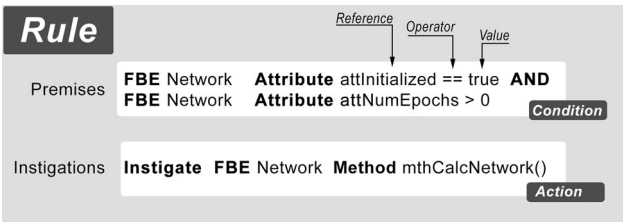

Fig. 1 A *Rule* in NOP: the diagram shows the components of a *Rule* in NOP, by means of an example with two *Premises* that makes up the *Condition*, and an *Instigation* that composes the *Action*. In this example, the *Attributes* used in *Premises*, as well as the *Method* used in *Instigation*, belong to the *FBE Network* [23, 27]

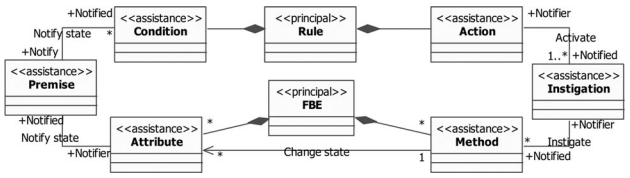

Fig. 2 NOP model: NOP entities and their relationships, represented by a class diagram in UML. A *Rule* contains *Conditions* and *Actions*, whereas a *FBE* contains *Attributes* and *Methods*. Each *Condition*, in turn, contains one or more *Premises*, which are notified by *Attributes*. Each *Action*, in turn, contains one or more *Instigations*, which instigate the execution of *Methods* [27]

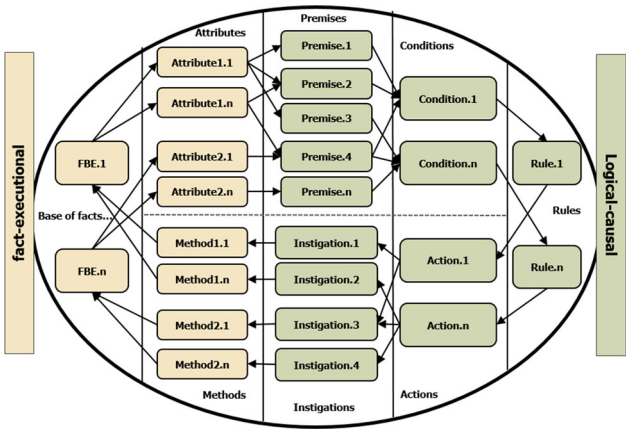

Fig. 3 Diagram representing the collaboration mechanism of notifications between NOP entities. The left side represents the fact-executional entities (FBEs, Attributes and Methods), and the right side represents the logical–causal entities (Rules, Condition, Premises, Actions and Instigations) [27]

implemented by the notifications between NOP entities [27].

Simão et al. [18] states that this notification mechanism of NOP assures that each change of *Attribute* state (i.e., "variable" state) activates only the strictly necessary evaluations of *Premises* and *Conditions* of *Rules* (i.e., "logical and causal expressions"). In addition, NOP shares the *Premise* collaborations (i.e., "results of logic evaluation") between *Conditions* (i.e., causal evaluations), thereby avoiding unnecessary repetitions of code in the *Rules* and avoiding unnecessary processing in the execution of the *Rules*. Thus, temporal and structural redundancies are, respectively, avoided, guaranteeing suitable performance by definition [17, 19].

NOP is also potentially applicable to develop parallel and/or distributed applications because of the "decoupling" (or minimal coupling) of its entities [20]. In inference terms, it does not matter if an entity is notified in the same memory region, in the same computer memory or in the same sub-network. For instance, a notifier entity (e.g., an *Attribute*) can execute in one machine or processor, whereas a notified entity (e.g., a *Premise*) can execute in

another. For the notifier, it is "only" necessary to know the address of the client entity [18].

That said NOP innovates in the way of carrying out the so-called logical or causal inference or calculation, which occurs through specific notifications as previously described. This, in fact, generates properties such as low-causal processing and decoupled entities, which facilitates to diminish processing time and to distribute processing in fine-grained way [10, 16, 27, 28]. Moreover, NOP inherits from declarative paradigm the ability to use abstraction of knowledge through logical–causal rules. Therefore, the NOP systems can be developed using the concept of rule-based or rule-oriented system, which is claimed to be easy and high level [21]. Therefore, NOP would be useful to make some systems that require these properties.

2.2 NOP materializations

By being a relatively new solution, NOP has been the subject of several research and development efforts. These efforts aim to develop and improve the concepts, techniques and tools associated with it, in such a manner to enable its validation and application.

The concepts of NOP were first materialized using the object-oriented paradigm (OOP), through a framework developed with the C++ programming language, elaborated as proof of concept. This framework materializes the collaborating entities defined in NOP. This occurs through classes and objects that are related to data structures. These classes allow each entity to have references to other entities interested in their states. The first application on smart control system of (simulated) manufacturing system demonstrated the NOP feasibility [29].

Subsequently, a stable version called Framework 1.0 was implemented specifically for mono-processor environments, covering the earlier presented concepts of NOP [18]. In this NOP Framework 1.0, it was observed the rule-oriented development and the decoupling of notifier entities. In addition, the optimization of the logical–casual calculation was observed [18, 28]. However, the performance was not as expected due to (in computational terms) expensive data structure used to implement the notification cycle. Those data structures were based on standard template library (STL) of C++.

Therefore, a new version was developed and called NOP Framework 2.0 [30]. In short, the NOP Framework 2.0 brings optimization of the notification cycle in terms of use of specific in-house data structure instead of STL, as well as more easiness in the NOP codification [30]. Even after the NOP Framework 2.0, experiments demonstrated that the results were not still with the expected performance to NOP, taking into account its linear asymptotical analysis [18]. This is mainly due to the use of data structures that, even if better than STL in the NOP Framework 2.0, are quite "expensive" to execute the process of inference based on notification.

Besides, in order to evaluate and enable the intrinsic parallelization and distribution features of NOP in software, an extension to the NOP Framework 2.0 was proposed by Belmonte et al. [10]. Called the NOP Framework 3.0, the solution allows the development of parallel applications in NOP in a transparent way. The work of Belmonte et al. [10] also presents a method of dynamic balancing of the work of software in multicore, denominated LobeNOI (*load balancing engine for NOI—notification-oriented inference—applications*). The NOP Frameworks 2.0 and 3.0 are used in this work and explained in detail in further sections.

Still in software, due to the performance of the frameworks being below expectations, the NOP research group created a specific language and compiler to NOP model, called NOP Language [31]. As a proposal for software development using only NOP artifacts, the developer defines the *FBEs* with *Attributes* and *Methods* and can create the *Rules*, *Premises*, and *Instigations* following the own NOP language grammar [31]. The source code in NOP Language can be compiled to generate applications in specific NOP C code, specific NOP C++, and even to NOP Framework 2.0, by using a proper compiler. The results of performance are quite as expected to NOP, even if the technology is still very prototypal [31, 32].

In addition, inclusively in distribution terms, NOP has been explored as a computational alternative to hardware paradigm beyond software [16, 33, 34]. In this sense, an innovative way of implementing the NOP using digital hardware elements was developed. Thus, NOP in digital hardware (NOP-DH) is a hardware implementation of the computational entities of the NOP notifications chain. The NOP-DH has been implemented in reconfigurable logic devices, namely FPGA [35]. The results obtained demonstrate that NOP-DH allows implicit distribution and makes somehow easier and organized the hardware development [36].

Still, Linhares et al. [16] proposed a new architecture called notification-oriented computer architecture (NOCA). This architecture was designed to execute NOP software from program memory into specialized cores to each sort of NOP elements, such as cores for *Premises* and cores for *Conditions*. The proposal is organized as a fine-grained multiprocessor that executes instructions hierarchically through sets of specialized cores. Its prototype was implemented in FPGA. Experiments conducted on Linhares et al. [16] demonstrated that NOCA is a viable alternative for software execution with parallelism, with compatible or better performance to Von Neumann model implementations [16].

In time, this present work did not use the materializations of NOP in hardware. However, it was necessary describing them due to its proposed use in future works.

That said, the next section presents, in a detailed way, the NOP Framework 2.0, as well as the modifications inferred by the proposed extension for multicore development purposes.

2.3 NOP Framework 2.0

As previously stated, one of the ways to program using the concepts of NOP is through a materialization (over the programming language C++) called NOP Framework 2.0 [30]. Due the NOP Framework 1.0 presented a degradation of performance, in NOP Framework 2.0 there was performed a series of optimizations in order to improve the execution performance of applications developed (in terms of data structure). The package structure of the NOP Framework 2.0 is shown in Fig. 4.

The NOP Framework 2.0 is divided into three main packages: the *Application* package, formed exclusively by the *Application* class, which makes the link between a NOP application and the other classes of the NOP Framework 2.0; the *Scheduler* package, which implements the classes of conflict resolution; and the *Core* package, which contains the collaborating classes that perform the notification process of the framework [30]. Figure 5 shows the *Core* package of the NOP Framework 2.0.

The source code in the NOP Framework 2.0 follows the C++ syntax. In order to create an application, the developer needs to extend the *NOPApplication* class, whose instance coordinates all the process. Also, it is necessary to extend the FBE class for each FBE idealized in the application. In the extended classes, the objects that represent

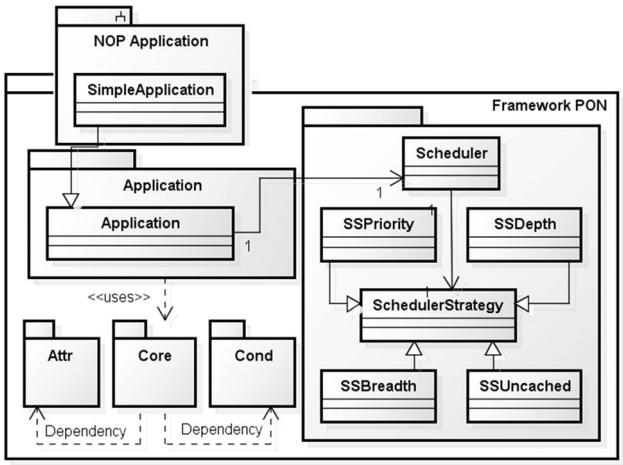

Fig. 4 UML package structure of the NOP Framework 2.0: relationship between the main packages *Application*, *Scheduler* and *Core*. Adapted from [37]

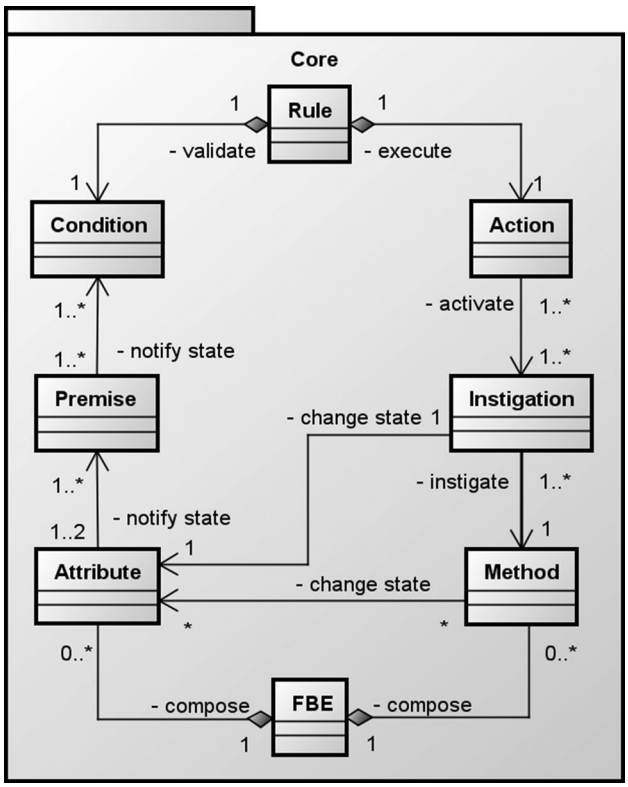

Fig. 5 The Core package of NOP Framework 2.0: NOP elements represented by classes and their relationships

the NOP entities, as *Rules*, *Conditions* and *Premises* are declared. The *Attributes* are declared according to its types: *Boolean, Integer, Double, Char* and *String*.

As shown in Fig. 6, in the lines 09–11 the three Attributes are declared (*Double* and *Boolean*). The lines 13 and 14 contain the declaration of the *Premises*: the lines 15 and

```
04 class NeuronOutput : public FBE {
05 public:
06     NeuronOutput();
07     virtual ~NeuronOutput();
08 public:
09     Double *attInput;
10     Double *attOutput
11     Boolean *attDone;
12 public:
13     Premise *preCalcOutput;
14     Premise *preUpdateWeights;
15     Method *mthCalcOutput;
16     Method *mthUpdateWeights;
17     RuleObject *rleCalcOutput;
18     RuleObject *rleUpdateWeights;
19 };
```

Fig. 6 Example of a FBE implementation: the *NeuronOutput* class extends the *FBE* class of the *Core* package. In the code, the first block contains the declaration of the constructor and the destructor of the class; the second block contains the *Attributes* declaration, being that the third block contain the implementation of other NOP entities related to this FBE

16 the declaration of the *Methods*; and the lines 17–18 the declaration of the *Rules*, all of them declared as pointers.

After the declaration of the NOP entities, it is necessary to implement the entities. As an example, Fig. 7 shows a source code snippet which implements the elements of the *Rule rleCalcOutput*.

The code presented in Fig. 7 defines a *Rule* and its entities. Using the syntax of the NOP Framework 2.0, the snippet code demonstrates:

- in the line 4, the creation of a pointer to a *Method*. At this case, the *Method* is called *calcOutput*, and its pointer is called *mthCalcOutput*;
- in the line 5, the creation of the *Rule rleCalcOutput*, which *Condition* is *STANDARD*, because it has only one *Premise*;
- in line 6, the creation of a *Premise* called *preCalcOutput*. This entity is responsible for verifying when the calculus of an output neuron can be done. The *Premise* returns *true* when the *Attribute attClock* (that controls the synchronization points of the neural network) is equal to 4.

It is observed therefore that the syntax used in the NOP Framework 2.0 facilitates the definition of a *Rule* and its entities by simple and well-organized code. Thus, from a correct definition of *FBEs*, the code becomes readable and maintainable.

2.4 NOP Framework 2.0 extension: NOP Framework 3.0

The demand for faster application processing speed is a constant in the area of software development. In this way, increasing the clock frequency of processors as well as the addition of multiple processing cores on a chip made computer systems become faster [38, 39].

In this context, the extension of the NOP Framework 2.0, entitled NOP Framework 3.0, provides a way to distribute NOP elements in multiple cores, in a transparent way [10]. Thus, one of the characteristics of this extension is the possibility of using source codes of existing applications, written in NOP Framework 2.0, without major changes.

```
04 mthCalcOutput = new MethodPointer<NeuronOutput>(this,
   &NeuronOutput::calcOutput);
05 RULE(rleCalcOutput, SingletonScheduler::getInstance(),
   Condition::STANDARD, mthCalcOutput);
06 PREMISE(preCalcOutput, net->attClock, 4, Premise::EQUAL,
   Premise::STANDARD, false, rleCalcOutput);
```

Fig. 7 Implementation of the *Rule rleCalcOutput* and its entities: an example of *Rule* in NOP Framework 2.0 syntax. The code snippet presents the creation of a pointer for a *Method*, the creation of a *Rule* and the creation of a *Premise*

The main classes that implement the NOP elements in the NOP Framework 2.0 have been retained. Nevertheless, some classes were created to support the multi-threaded mechanism when available. Figure 8 shows the additional NOP Framework 3.0 packages and their relationship to the main NOP Framework 2.0 packages that have been maintained.

The *NOPVectorMulticore* package contains specializations of the *NOPVECTOR* structure of NOP Framework 2.0 to work with multicore applications. Essentially, the *NOPVECTOR* structure acts as an iterator over the NOP entities in an optimized way, reducing the use of the application's data cache.

Otherwise, the classes belonging to the *Multicore* package, presented in Fig. 8, are responsible for:

- *SynchronizedQueue*: such a class implements a data structure in the form of a queue. In this are stored NOP entities to be executed, synchronously, in each of the cores of a processor.
- *CoreControllersManager*: class responsible for creating an instance of the *CoreController* class for each core of the target machine. Also, during application execution, this class records the notifications sent to each relevant NOP entity.

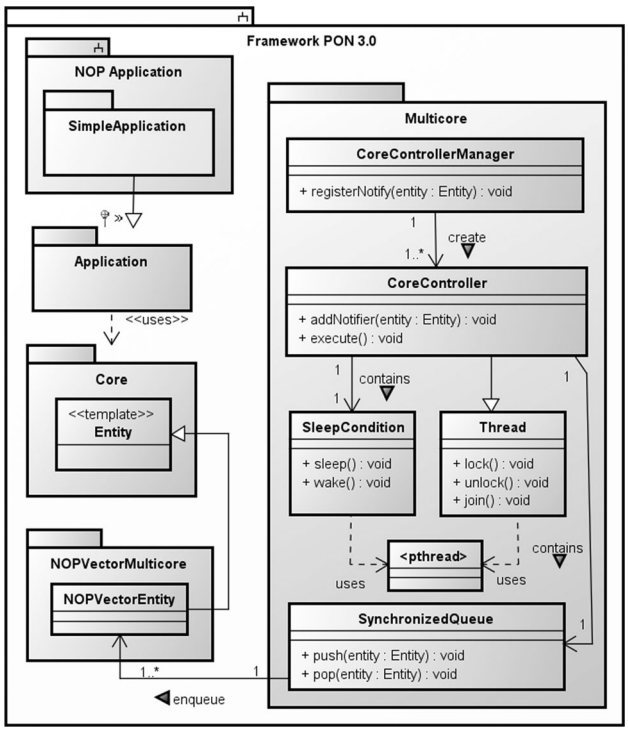

Fig. 8 UML package structure of the NOP Framework 3.0. Two additional packages are presented, *NOPVectorMulticore* and *Multicore*, aiming the execution of NOP entities in a concurrently way, using multiple cores

- *CoreController*: this class is responsible for queuing NOP entities when the *CoreControllerManager* class registers a notification. It is also responsible for the concurrent execution of the *Premises*, *Conditions*, *Instigations* and *Methods* in each processing core.
- *SleepCondition*: such class has methods that assist in controlling the execution queue of NOP entities. These methods use the relevant methods of the *<pthread>* class in a controlled way.
- *Thread*: the instances of this class are used as the execution flows themselves. In this proprietary class, the methods of the mutual exclusion of the *<pthread>* class are used in a controlled way.
- *<pthread>*: this class, belonging to the GNU language C library, has all the primitives for the creation, control and destruction of execution flows, in multicore environments based on the Linux operating system. As already presented, the methods of this class are used in a controlled way in other classes of the *Multicore* package, aiming at the correct distribution of execution, in the cores target processor.

Regarding the thread and core control in NOP Framework 3.0, each FBE instantiated in the application is related to an instance of the *CoreController* class. The *CoreControllerManager* class, in turn, instantiates each of the application's FBEs in a specific core. Thus, the entities in the notification chain are distributed among the cores of the target machine.

In order to respect the correctly execution of the NOP entities in each processing core, the *CoreController* class has a list of entities to be executed. Such list is implemented as a queue. Thus, unlike the NOP Framework 2.0 where each NOP entity was processed as soon as it was notified, when an entity is notified it goes to an execution queue. In this case, once the processor core is available for execution, an entity is removed from the queue and executed. Figure 9 shows the control flow of execution in the NOP Framework 3.0, using an UML activity diagram.

As can be seen, the UML activity diagram in Fig. 9 is divided into two lanes:

- The first lane shows the sequence of activities of the main thread. It is noted, therefore, that there is the creation of the threads and the wait for their completion (join process). When the main thread takes back the control, the threads are destroyed.
- the second lane represents each of the threads created in the application and is divided into two parts:
- *Entity execution process*: initially, an instance of the *CoreController* class checks if there is any entity in the queue. Such process repeats itself until some element is enqueued. If there is any element, the dequeue process of an entity happens, and its actions are executed. After the execution, there is a verification regarding the sending of notifications by this entity. If there is no notification, the flow is terminated. Otherwise, a notification is sent.
- *Queue management process*: as previously presented, NOP reactive entities send timely notifications during execution of the inference process. In this context, once the thread control manager receives a notification, the reporting entity is enqueued. At the end, a wake signal is sent to the *CoreController* instance that handles the entity execution process.

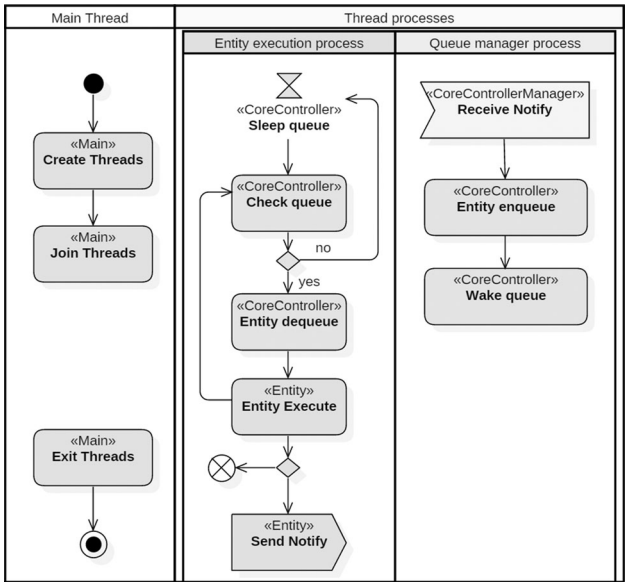

Fig. 9 UML activity diagram: control flow of the NOP Entities. The first lane shows the main thread. The second lane is shows an example of other threads create in the application divided into two parts: the first is the process of carrying out the actual entities, and the second part is the queue control process

Figure 9 presents the activities that are performed aiming the correct processing of each NOP entity of the application, sequentially in each thread. However, the NOP Framework 3.0 allocates the entities in the cores available on the target processor. In this context, the activities presented in Fig. 9 are executed concurrently. Figure 10 presents, in a UML object diagram, an example of concurrent execution between NOP entities.

The UML object diagram, shown in Fig. 10, demonstrates an example of concurrent computation between two hidden layer neurons in an RNA. The notification process starts when the value of the *Attribute net::attClock* is modified. Such modification notifies the *Premises hidneu0::preCalc* and *hidneu1::preCalc*. From this point on, the entire process of notification and execution of the NOP entities occurs concurrently. At the end, when the values of the *Attributes hidneu0::attDone* and *hidneu1::attDone* are modified, *Premise net::preCalcDone* is notified.

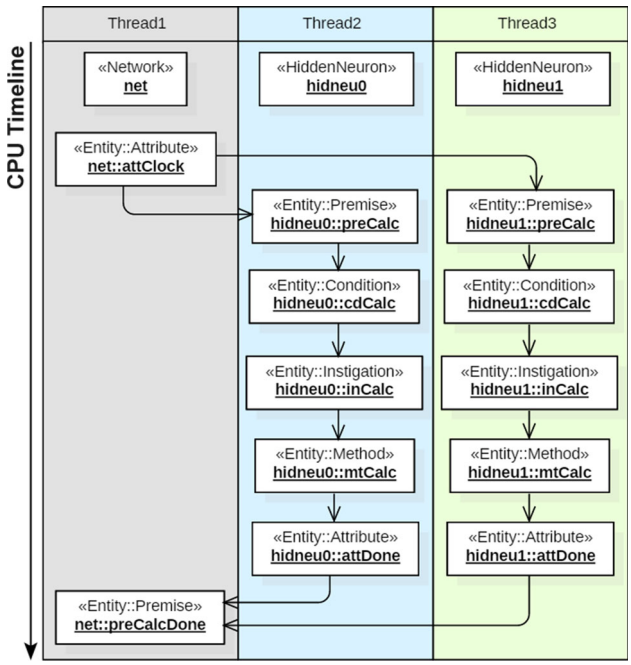

Fig. 10 UML object diagram: concurrently execution flow of NOP Entities. Each lane shows a different thread, allocated in different cores. As an example, entities belonging to three instances of *FBEs* are arranged: the *net* instance of the *FBE Network*, and the *hidneu0* and *hidneu1* instances of *FBE HiddenNeuron*

Also, in addition to the concurrent execution shown in Fig. 10, it is important to note that NOP entities that are in one thread access NOP entities that are in another thread. This mechanism is considered critical for multicore processing. In this context, mutual exclusion on access to shared NOP entities is guaranteed by the methods of the classes *Thread* and *SleepCondition*, belonging to the *Multicore* package.

As previously mentioned, the NOP Framework 3.0 uses an application program interface (API) that implements the execution model *Posix Threads* (*pthreads*) [40]. The *pthreads API* provides a set of libraries, in which parallelism is operationalized through threads. In this context, the library provides thread control and synchronization capabilities [10]. Thereby, the *Multicore* package of the NOP Framework 3.0 uses the concepts of mutual exclusion (*mutex*) to control the access and execution of shared entities, between the multiple flows (*threads*) created.

Another important goal of the NOP Framework 3.0 is that the process of allocating NOP entities in multiple cores is transparent to the developer. The only change in the code used in NOP entities, in relation to the NOP Framework 2.0 source code, is the specification of the execution core of each entity. Figure 11 shows the implementation, using the NOP 3.0 Framework, of the *rleCalcOutput Rule*.

Comparing the code shown in Fig. 11 with the code shown in Fig. 7, the only difference is the indication of the

```
04 mthCalcOutput = new MethodPointer<NeuronOutput>(this,
   &NeuronOutput::calcOutput);
05 RULE(rleCalcOutput, SingletonScheduler::getInstance(),
   Condition::STANDARD, core, mthCalcOutput);
06 PREMISE(preCalcOutput, net->attClock, 4, Premise::EQUAL,
   Premise::STANDARD, false, core, rleCalcOutput);
```

Fig. 11 Implementation of the *Rule rleCalcOutput* and its entities in NOP Framework 3.0. The specification of the core execution in each NOP entity is highlighted (bold)

core in which the NOP entity is to be executed. In general, such a parameter is indicated in the *FBE* instantiation and copied to each related entity.

That said, it is observed that the NOP Framework 3.0 enables concurrent programming on multicore systems, transparently. Experiments conducted on multicore computers have shown that there is a real distribution of NOP entities among the cores of the target processor. Such experiments are described in Chapter 4 of this paper.

3 Artificial neural networks (ANN)

Artificial neural networks (ANN) can be defined as a set of processing units called neurons, which are interconnected by artificial synapses (synaptic weight matrices) somehow imitating the nervous system of human beings [2]. Neurons are grouped in layers, each layer qualifying the role of the neuron in the network. The main features of ANN, highlighted by Haykin [2], are adaptation from experience, learning ability, generalization ability, data organization, fault tolerance, distributed storage, and ease of prototyping.

An ANN can be arranged in various ways, through the connection of its neurons. Among the possible architectures, there is the feedforward architecture, where the connections are made in only one direction, from the input to the output. In another architecture, the recurrent one (with feedback), it allows to connect the output of the neurons of subsequent layers in the input of the previous layer neurons. Therefore, the used way defines the architecture to which it belongs [2].

Beyond the topological architecture, another important aspect is the type of ANN [2]. The MLP ANN is the focus on this work. This ANN architecture was chosen because it is widely used, even in the most modern and recent researches in machine learning area, becoming useful for the research of NOP concepts applied in ANNs [24].

In this sense, this work describes the use of the MLP ANN in a classification experiment. The source codes implementing the MLP ANN for the solution of the IRIS Dataset problem, both in IP and NOP, can be found in subsequent sections.

3.1 Training process of ANN

The training process of ANN involves the application of a learning algorithm for adjusting the weights and thresholds of its neurons. It is expected that after such training process, the system would be capable of generalizing solutions for the proposed problem, producing outputs close to those expected. The ANN, therefore, has the capacity of learning by input patterns that represent the behavior of the system where it is applied [24]. As an example, Fig. 12 shows an UML activity diagram for a MLP ANN training process.

There are several specific training algorithms, such as the training algorithm for RBF (radial basis function) and Kohonen (self-organization maps—SOM) networks, which uses Euclidean distances between neurons for its adjustment [2]. In this work, the BP algorithm is used and is described in the subsequent section. This choice is justified by its widespread utilization by the scientific community, even in the most modern architectures of ANNs, and its success to solve problems such as classification and functions approximation [24, 41]. Therefore, it will be very useful for the continuity of the research in ANN using NOP.

3.2 Backpropagation algorithm

Proposed in 1986 by Rumelhart et al. [42], the backpropagation algorithm is a learning process for MLP ANNs. The algorithm splits the training algorithm in two specific phases, executed sequentially for each training pair, in multiple rounds or epochs. This *algorithm* is very popular and widely used the training in MLP ANNs [43].

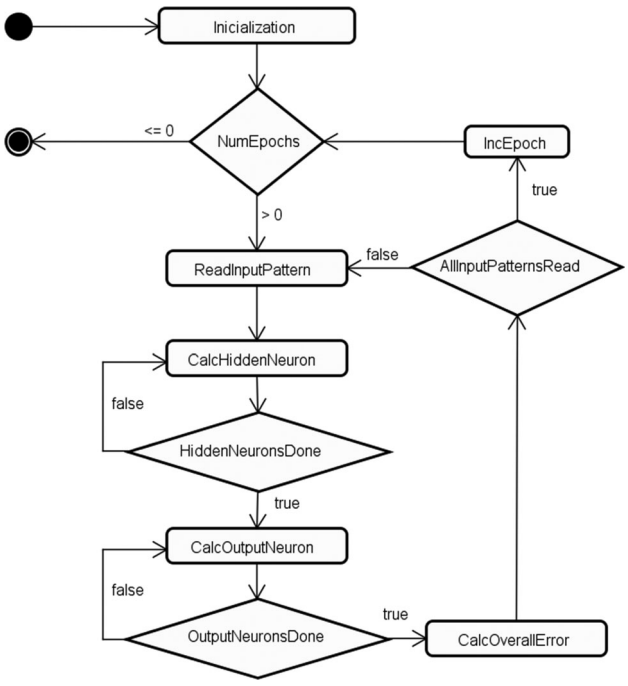

Fig. 12 UML activity diagram for a MLP training process: a feedforward-supervised algorithm, with the quantity of epochs (*numEpochs*) as the stop training condition

At first, the network is initialized with randomized weights. During the learning process, in the first phase (forward phase), values are presented at the inputs and multiplied by the weights in subsequent layers. Finally, the resulting value is given to the activation function and it generates the output of each layer.

In the second phase, as the learning procedure is supervised, the difference between the expected output and the output produced by the ANN generates an error. This phase (backward) uses the error obtained previously to change (adjust) the synaptic weights of all neurons in the ANN.

In architectures using hidden layers, there is no direct access to the values of errors for each of the neurons belonging to such layers. Thus, their adjustments values are based on the errors of the neurons of the next layer, multiplied by the synaptic weights connecting these neurons [2].

In order to illustrate the process of training of an MLP ANN, the next section provides an implementation using imperative programming. The implemented algorithm, as an example, is the same one used to train the MLP ANN for the IRIS Dataset problem, and the results are compared with the implementation using NOP, in the subsequent sections.

3.3 Example of MLP BP training using C language from the imperative paradigm

This section describes an imperative paradigm (IP) implementation of the BP training used in this work, based on Ludwig and Montgomery [44]. In the algorithm implementation, written in C programming language from procedural approach of IP, the function that performs the needed process for training was placed on the "main.c" file of the application. Figure 13 shows the main program, with the initial settings, loops and the calling function that perform the calculations.

The epochs (number of repetitions or rounds) were controlled by a simple loop, whereas another nested loop controls the reading of each input pattern. The weights are initiated with random small values. The activation function used, in the example, is the hyperbolic tangent. Figure 14 shows a snippet of source code that implements the function *calcNet()*.

The code displayed in Fig. 14 calculates the values of the hidden neurons and the error in a given training pattern. The first five lines (lines 10–14) are used to declare the

```
47 int main(void){
48     srand ( time(NULL) );
49     initWeights();
50     initData();
51     for(int j = 0;j <= numEpochs;j++){
52         for(int i = 0;i < numPatterns;i++){
53             patNum = rand() % numPatterns;
54             calcNet();
55             WeightChangesHO();
56             WeightChangesIH();
57         }
58         calcOverallError();
59     }
60     displayResults();
61     return 0;
62 }
```

Fig. 13 BP training algorithm using C programming language—code snippet: main() function of the application. Adapted from [36]

```
09 //// variables ////
10 int patNum = 0;
11 double errThisPat = 0.0;
12 double outPred = 0.0;
13 double hiddenVal[numHidden];
14 int trainInputs[numPatterns][numInputs];
15
16 void calcNet(void){
17   //calculate the outputs of the hidden neurons
18   int i = 0;
19   for(i = 0; i < numHidden; i++){
20     hiddenVal[i] = 0.0;
21     for (int j = 0; j < numInputs; j++){
22       hiddenVal[i] += (trainInputs[patNum][j] *
   weightsIH[j][i]);
23     }
24   hiddenVal[i] = tanh(hiddenVal[i]);
25   }
26   //calculate the output of the network
27   outPred = 0.0;
28   for(i = 0;i < numHidden;i++){
29     outPred += hiddenVal[i] * weightsHO[i];
30   }
31   //calculate the error
32   errThisPat = outPred - trainOutput[patNum];
33 }
```

Fig. 14 Snippet of the function *calcNet()*. Adapted from [36]

variables, representing the number of the training pattern, the error on a given pattern, the output of the ANN in a given pattern, and the array that represents the values of the hidden neurons and the training inputs.

Two nested for loops are used to calculate the value of each hidden neuron, and another for loop is used to calculate the output of the given input training pattern. At the end, the difference between the expected output and the current output is calculated. Another functions used in Fig. 14 are not showed here, but follow the same IP implementation logic.

In order to ensure the effectiveness of procedural implementation in C language, the FANN (*fast artificial neural network*) library was used [22]. In this context, the library was installed and configured with the same training parameters used in the proprietary solution. Figure 15

```
07 int main(){
08     const unsigned int num_input = 4;
09     const unsigned int num_output = 3;
10     const unsigned int num_layers = 3;
11     const unsigned int num_neurons_hidden = 5;
12     const float desired_error = (const float) 0.001;
13     const unsigned int max_epochs = 1000;
14     const unsigned int epochs_between_reports = 1000;
15     struct fann *ann = fann_create_standard(num_layers,
16 num_input, num_neurons_hidden, num_output);
17     fann_set_activation_function_hidden(ann,
18 FANN_SIGMOID_SYMMETRIC);
19     fann_set_activation_function_output(ann,
20 FANN_SIGMOID_SYMMETRIC);
21     fann_train_on_file(ann, "IRISDataSet.data",
22 max_epochs, epochs_between_reports, desired_error);
23     fann_save(ann, "IRISFANN_float.net");
24     fann_destroy(ann);
25 }
```

Fig. 15 Source code from an ANN MLP training with BP method, using the FANN library: 4 inputs, 3 outputs, 3 layers and 5 neurons in the hidden layer

shows the source code used to train the network using the FANN library.

In the code shown in Fig. 15, there is the adjustment of the parameters in lines 07–14. In line 15, there is the creation of ANN using library functions. Lines 17–20 show the command to set the transfer functions used (tanh). On line 21 the command for the ANN training previously created is displayed.

3.4 Considerations about the MLP BP imperative programming

As it can be seen, repetition blocks are used for the calculation in order to go through the arrays, generating a highly coupled code. In this context, it highlights the fact that the usual programming languages (such as C language used here) have no real facilities to develop optimized and decoupled code (or minimally couple to be precise). This makes it difficult to use available processing for not optimizing resources [12–14, 17, 45].

In addition, as it is written entirely sequentially, this code cannot be easily parallelized. This also makes it difficult to distribute code in the case of a system with parallelism, and it is particularly difficult to distribute code with fine granularity. This happens in the usual programming languages due to the structure and nature of execution enforced by their respective paradigms [15, 18, 46].

Furthermore, it is important that the process of distributable code development be agile and practical, since distribution is reliably necessary in certain contexts. However, the distribution itself is a problem because, from different perspectives, it may require strategies such as load balancing to avoid over communication, or fine-grained code distribution to take advantage of available processing resources [10, 34, 47].

That said, in order to overcome this shortcomings, the next section describes the implementation of an MLP ANN BP training using NOP, on the IRIS Dataset.

4 Implementation of the BP training algorithm using the NOP Framework

This section presents the implementation of the BP algorithm, in the training of an MLP ANN, using NOP.

Although the present work is based on the use of NOP Framework 2.0 and 3.0 for the implementations, the source codes are presented using LingNOP. With this language, the implementation of the algorithms is done in a declarative way, optimizing the programming process. Subsequently, through use of the NOP compiler [31, 32], the translation from NOP Language to NOP Framework can be performed (see Sect. 2.2 and the work of Ronszcka et al. [31] for further details).

4.1 Application development in NOP

The main structure of the application follows the MLP ANN implementation, changing only the number of generated structures for each problem. The configuration used for the IRIS Dataset has four inputs, five hidden neurons (one hidden layer) and three output neurons. The activation function used is sigmoidal (hyperbolic tangent). Figure 16 shows the diagram representing the relationship between the created entities for the NOP MLP ANN application.

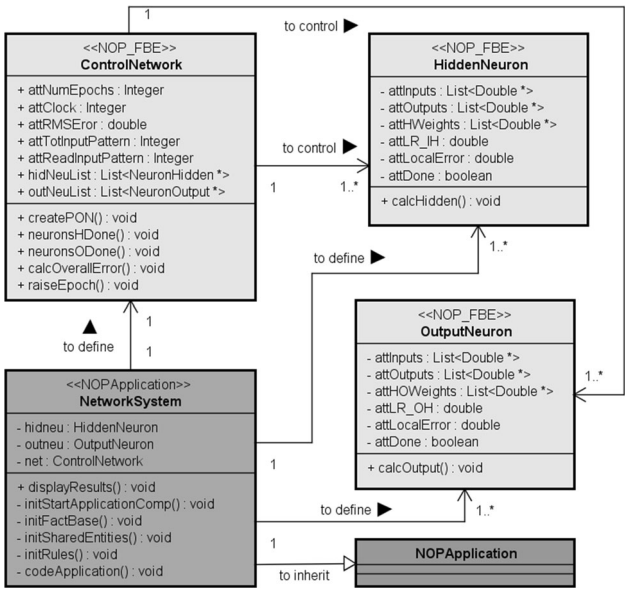

Fig. 16 Class diagram of the ANN proposed using NOP Framework 2.0

Three main sets of *Fact Base Elements* (*FBEs*) for the implementation of the BP training algorithm using NOP were defined: (1) *HiddenNeuron FBEs*, which implements each hidden neuron; (2) *OutputNeuron FBEs* that implements the output layer; and (3) *ControlNetwork FBE* that contains *Attributes* and *Methods* necessary to control the network itself.

Due to the use of the NOP Framework 2.0, it is necessary to derive the *NOPApplication FBE* (see Fig. 4). Thus, an inheritance class from the *NOPApplication FBE* was created, named *NetworkSystem*. This class contains the methods needed to run the proposed application in accordance with the principles of NOP and the NOP Framework 2.0.

The structure of the diagram is based on the main structure of the NOP Framework 2.0. Based on the entities defined in the interrelated models in Figs. 2, 4, and 15, the NOP application for the MLP ANN BP training was elaborated. Figure 17 shows the notification chain of the application using an activity diagram of UML as an alternative model.

In the diagram presented in Fig. 17, the openness of concurrency on indicates where NOP allows concurrency

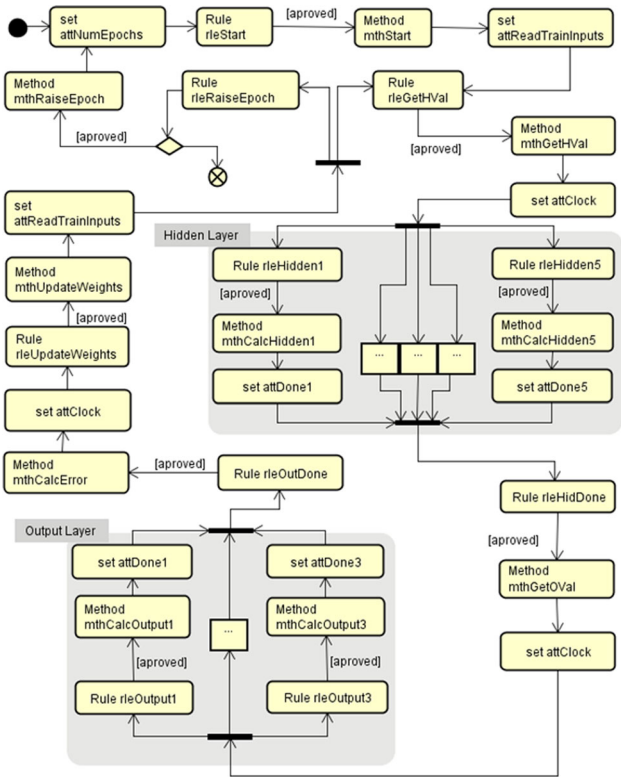

Fig. 17 UML Activity diagram: notification chain of the MLP ANN training, with BP algorithm, using NOP. Each activity is related to notify or execute a NOP entity. The concurrency is highlighted, in the activity diagram, both in execution of the neurons of the hidden layer, as in the neurons of the output layer

or parallelism, in a transparent way. This is highlighted in the diagram, both in the execution of the activities of the neurons of the hidden layer, as in the neurons of the output layer.

Computation in NOP is normally started by detecting a change in an *Attribute* of a *FBE*, which starts the notification chain. In this sense, the code that implements, using NOP Language [32], the *Rule* that represents the start of this notification chain is presented in Fig. 18.

As presented in Fig. 18, when the value of the *attNumEpochs Attribute* changes, the *Rule rleNetStart* verify, by means of the notification chain composed of pertinent *Premise* and *Condition* entities, if the value of the *Attribute* is greater than zero. If true, the *Method mthStart* is called, by means of the pertinent *Rule-Action-Instigation* entities, thereby performing the instructions designed by its source code.

The *Method mthStart* adjusts the *attReadTrainInputs Attribute* to zero (indicating that none of the input patterns of the training process in current epoch was read), thereby notifying the *Rule rleGetHVal*. This *Premise* allow the pertinent *Condition-Rules* verifies that, if the total of read inputs is smaller than the total training inputs, the *Method mthGetHVal* (Fig. 19) can be executed by means of the pertinent *Rule-Action-Instigation*, reading the next entry.

After reading each training input patterns and their proper attribution to the inputs of the hidden layer, the calculation process is triggered. For this control, the *attClock Attribute* is used to maintain the synchronization between the neurons (both hidden and output ones), as they work in a decoupled way, according to the NOP rules. As said, NOP implicitly decouples factual, logical, causal, and executional entities, allowing most of the process be done in parallel in terms of computational model. Figure 20 presents the *Rule rleHidden* responsible to describe this process.

As can be seen in Fig. 20, the *rleHidden Rule* is not defined with the reserved word *rule*, but with the reserved word *formRule*. In this case, the *Premises* and *Instigations* defined in this *Rule* do not use the *Attributes* and *Methods* of instances of *FBEs*, but rather the identification of the

```
85 rule rleNetStart
86   condition
87     subcondition sbcNetStart
88       premise preNetStart net.attNumEpochs > 0
89     end_subcondition
90   end_condition
91   action
92     instigation insNetStart net.mthStart();
93   end_action
94 end_rule
```

Fig. 18 Implementation, using NOP Language, of the *Rule rleNetStart*. In the logical–causal part, a *Premise* is defined in a *Condition*. In the facto-execution part, an *Instigation* is defined in a *Method*

```
85 rule rleGetHVal
86   condition
87     subcondition sbcNetGetValues
88       premise preNetGetValues net.attReadTrainInputs >
   net.attTotTrainInputs
89     end_subcondition
90   end_condition
91   action
92     instigation insNetGetValues net.mthGetHVal();
93   end_action
94 end_rule
```

Fig. 19 Implementation, using NOP Language, of the *Rule rleGetHVal*

```
85 formRule rleHidden
86  condition
87   subcondition sbcNHCalcHidden
88    premise preHidden HiddenNeuron.attDone == false and
89    premise preHiddenF Netwrok.attClock == 2
90   end_subcondition
91  end_condition
92  action
93   instigation insNHidden HiddenNeuron.mthCalcHidden();
94  end_action
95 end_formRule
```

Fig. 20 Implementation, using NOP Language, of the *Rule rleHidden*

FBE itself. In the translation process from NOP Language to the NOP Framework 2.0, a precompiler checks how many instances of *FBE* exist, and then allocates all the involved *Premises* and *Instigations* appropriately (in this case, five instances are considered). This process is called Formation Rules [31, 48].

In relation to the *Rule rleHidden*, it involves two *Premises*, nominated *preHidden* and *preHiddenF*. They, respectively, evaluate the *attClock Attribute* and the *HiddenNeuron* status, represented by the *attDone Attribute*. The *Method mthCalcHidden* of each neuron will be executed by means of *Condition–Rule–Action–Instigation*, if the conjunction between the two *Premises* is true. Figure 21 shows the code of the *Method mthCalcHidden*.

After the execution of calculations, the *Method mthCalcHidden* changes the *attDone Attribute* of *FBE HiddenNeuron* to true, thereby notifying the *Premises* involved in *Rule rleNeuronsHDone*. The true conjunction of these *Premises* initiates the process of calculation in

```
43 void NeuronHidden::mthCalcHidden() {
44   double sumOfProducts = 0.0;
45   sumOfProducts = (i0*w0)+(i1*w1)+(i2*w2)+(i3*w3);
46   attOutput = tanh(sumOfProducts);
47   attDerv = 1.0-(attOutput->getValue()*attOutput->getValue());
48   attDone = true;
49 }
```

Fig. 21 Implementation, using NOP Language, of the *Method mthCalcHidden*. The computation of each *HiddenNeuron* uses an auxiliary local variable to calculate the sum of the weighted inputs. In addition, the calculus of the output value and the derivative, to be used in weight adjust, is performed

each *FBE OutputNeuron* of the output layer, having as input the results of *FBE HiddenNeurons* in the hidden layer.

The methodology used in the implementation of the output layer is the same as the hidden layer. Thus, after the necessary calculations in each of the *FBE OutputNeurons* instances of the output layer, the notification of the *Rule preNeuronsODone* is performed.

In turn, the *Method mthNeuronsODone* disables the output neurons, adjusting the *attDone Attribute* of each instance of *FBE OutputNeuron* to false, in order to avoid unnecessary calls by the notification chain. Also, this Method executes the calculation of the overall network error. In addition, it changes the *attClock Attribute* to five (5), which is necessary to enable the next step of the BP algorithm. In this context, the *Rule rleUpdateWeights* becomes true, which invokes by notification the *Instigation* (via its *Action*) responsible for performing the *Method mthUpdateWeights*. This *Method* updates the synaptic weights and adjusts the *Attributes* attached to the *Rules rleRaiseEpoch* and *rleGetHVal*, thereby restarting the process.

As this work treats on the NOP Framework 3.0 in the implementation of BP training of an MLP ANN, Fig. 22 presents the implementation of the *Rules rleNetStart* (here named as *rleStart*) and *rleGetHVal* (displayed in Figs. 18, 19) using NOP Framework 3.0 syntax.

It is important to note that the implementation of an application can be performed directly in NOP Frameworks 2.0 or 3.0. However, as mentioned, the translation of the NOP Language to NOP Framework 2.0 can be accomplished through the compiler.

As can be seen in Fig. 17, the notification chain indeed initiates by changing the value of the *attNumEpochs Attribute*, which notifies the first pertinent *Rule*, by means of *Premises* and *Conditions*. Figure 23 shows the source code snippet that starts the NOP notification chain.

```
84 // implementation of Rule rleStart
85 mthStart = new MethodPointer<ControlNetwork>(this,
   &ControlNetwork::start);
86 RULE(rleStart, SingletonScheduler::getInstance(),
   Condition::SINGLE, this->core, mthStart);
87 PREMISE(preStart, this->attNumEpochs, 0,
   Premise::GREATERTHAN, Premise::STANDARD, false, this-
   >core, rleStart);
88 // implementation of Rule rleGetValues
89 mthGetValues = new MethodPointer<ControlNetwork>(this,
   &ControlNetwork::getValues);
90 PREMISE(preGetValues, this->attReadTrainInputs, this-
   >attTotTrainInputs, Premise::SMALLERTHAN,
   Premise::STANDARD, false, this->core, rleGetHVal);
91 RULE(rleGetHVal, SingletonScheduler::getInstance(),
   Condition::SINGLE, this->core, mthGetHVal);
```

Fig. 22 Source code of the *Rules rleStart* and *rleGetHValu* implementation, using NOP Framework 3.0

```
72 void NetworkSystem::codeApplication() {
73     net->attNumEpochs->setValue(1000);
74 }
```

Fig. 23 Implementation, using NOP Framework 3.0, which adjust the value of the *Attribute attNumEpochs*

4.2 Considerations about the MLP BP NOP implementation

Concerning to the MLP ANN coding, NOP allows algorithmic flexibility and an abstraction level in the form of FBEs, inherited from OOP. In addition, since NOP uses PD concepts such as ease of programming at high level and a representation of knowledge in rules like SBR, implementing to MLP ANN has become practice, taking source code readable and maintainable.

Another advantage is the mechanism of notifications, which is organized in autonomous and reactive entities that collaborate through occasional notifications. In this, the responsibilities of a program are divided among the objects of the model, allowing optimized and decoupled (or minimally coupled) execution, useful for the correct use of monoprocessing (NOP Framework 2.0), as well as for distributed processing (NOP Framework 3.0).

Moreover, in NOP the rule knowledge is threated by *Rule* entities (and its sub-entities such as *Condition*, *Premises*, and so on) whose inferences happen from *FBE* entities (and its sub-entities namely *Attributes* and *Methods*) thereby decoupling better the entities. An example is the *FBEs* of neurons (both *HiddenNeuron* and *OutputNeuron*), whose control is done independently, since the order of execution is controlled by the mechanism of inference and the notifications chain. That said, all neurons can therefore execute in parallel in a form of fine-grained parallelism (see Fig. 10).

5 Experimental results

For the IRIS Dataset, tests were conducted with 1000 epochs of training. The initialization of the weights was random, but the same for implementation with the NOP Frameworks 2.0, 3.0 and IP procedural C language, varying the values from 0 to 1.

In every execution of the training algorithm, both in the IP procedural C language implementations (proprietary and with FANN library) and with the NOP Frameworks 2.0 and 3.0, the average errors were about the same, validating that all implementations had the same results. However, the execution times in both NOP Frameworks were higher. This will be discussed below.

In the IRIS Dataset example, the training was conducted using 80% of the data samples, which were randomly

selected but the same to implementations in NOP Frameworks 2.0, 3.0 and in IP procedural C language. In each implementation, this resulted in 120 training patterns (40 for each kind of Flower), with four inputs (sepal and petal width and length) and three possible binary outputs. Figure 24 shows the results of the experiments.

The average execution time in 100 executions (1000 epochs each) was of 2980 ms to the implementation in NOP Framework 2.0. The NOP BP algorithm achieved good results using the factor of 0.1 as learning rate, reaching an overall error of 0.029, and a minimum error of 0.015 in one of the executions.

In the NOP Framework 3.0, processing times were higher. Even so, they achieved results equivalent to the Framework NOP 2.0 as compared to the average error in training. This is because the implementation is practically the same in both versions of the Framework. However, the goal of having a parallel, fully transparent code has been achieved. Figure 25 demonstrates the system monitor during the execution of the training.

Figure 25 displays the behavior of the processing cores during the execution of an ANN MLP training using the BP algorithm, with Framework NOP 3.0. In this context, even in a slower way, the process is being conducted through multiple processing cores, thus demonstrating the plausibility of using the NOP Framework 3.0 in the construction of parallel ANNs.

In time, all the experiments were conducted in a Personal Computer with a CPU type Intel® Core™ i7-3770 @ 3.50Ghz, with 32 Gb of RAM DDR3 @ 1358 MHz. The Operational System installed and used is Linux Ubuntu 16.04 LTS, 64-bit architecture.

In the implementation using IP procedural C language, both in the proprietary algorithm and FANN, the average execution time in the 100 executions (1000 epochs each)

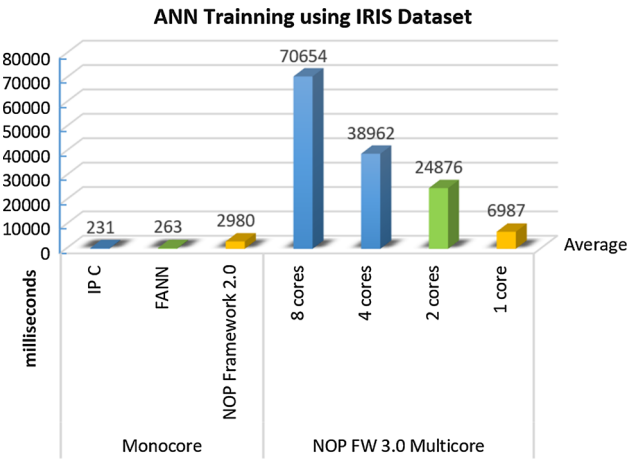

Fig. 24 Results of the experiments: MLP ANN training execution time in the IRIS database after 1000 training epochs. The result is the average of 100 executions of each application

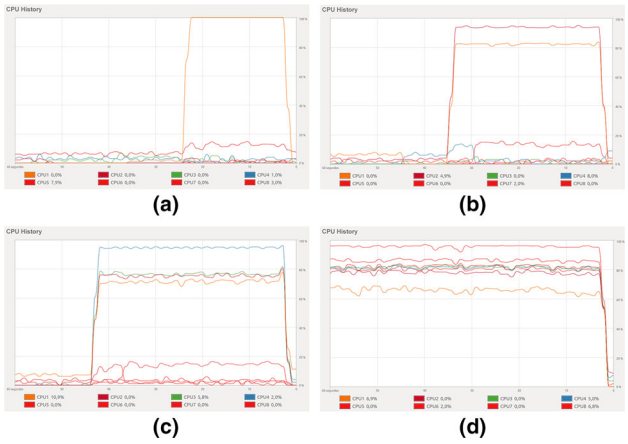

Fig. 25 Screenshot of CPU history executing ANN MLP training with BP algorithm, using the NOP Framework 3.0: **a** 1 core; **b** 2 cores; **c** 4 cores; **d** 8 cores

was 231 and 263 ms. Also with factor 0.1 as the learning rate, the average error was 0.038 in the 100 executions.

In both tests, with the IRIS Dataset and in the XOR training detailed in [23], the closeness in the values of the average overall error between the implementations (NOP Framework 2.0, 3.0, proprietary IP procedural C language and with FANN library) shows that the training performed using the NOP Framework is plausible. The difference in the execution times was expected, since the NOP Framework implementation makes intense use of OO techniques such as virtual methods and polymorphism that imply in slower execution time.

6 Conclusions and future works

This section discusses the NOP implementation of an ANN and presents future works.

6.1 Conclusions

An advantage in the use of NOP Framework 2.0 and 3.0 for implementing an ANN solution was the form of development. Firstly, NOP promotes a better organization of the solution for ANN than imperative programming (IP) due to its level of abstraction. In addition, differently than IP, NOP explicit the knowledge of each part of the ANN, which enhances the maintainability of the source code. With a clear view of each element of the ANN, the developers can easily adjust components according to their need.

Besides, NOP also allows the creation of decoupled and easy to distribute entities due to the notification nature. This decoupling level allows fine-grained level of parallelism and distribution in NOP systems as discussed in

[10, 16, 33]. Thus, the difficulties in the definition of decoupled and parallel programming structures from IP are suppressed by NOP programming. In NOP programming, the developer expresses the factual–executional and logical–causal knowledge through *FBEs* and *Rules*, since the decoupling happens naturally, by emergence, in construction time [18, 27].

Once defined the structures of the *FBEs* and *Rules*, after program start-up, notifications do not follow a sequential, systematic (step-by-step) execution pattern as in IP. In NOP, a sort of collaborative inference engine (emerged by notifier entities) defines when each action should be executed [17, 28]. This fact was observed by the need of control over the end of execution of each neuron. This was done in order to later carry out the calculation of the average of the entire network error.

The decoupling among entities, presented in NOP, is one factor that enables the parallelism in a, so-called, Neuro-NOP implementation. In this context, the implementation of ANN MLP in NOP presented in this article demonstrated the plausibility of developing a parallel code in a transparent way (see Fig. 25). Therefore, this paper presented the concept demonstration of the feasibility of a Neuro-NOP solution.

The results of the experiments, regarding the execution time of training of an MLP RNA in NOP, demonstrated that the code generated by NOP Framework 3.0 is slower than the codes in IP procedural C language and by the FANN library. This result is, in part, a reflection of the use of costly computational structures in the NOP Framework 3.0 implementation, as well as occurs in NOP Framework 2.0.

Another relevant issue regarding the multicore solution lies with the used execution model. Shared memory management, for example, can be considered as a bottleneck in the execution of the experiments. Since the BP training algorithm uses the result of all neurons calculus in the adjustment of the synaptic weights, the mutual exclusion (i.e., *mutex lock* and *unlock*) process must be widely used, causing such control to spend time processing [10, 49, 50].

In addition, the process of controlling execution of multiple processing cores may also be considered a bottleneck. In the experiments, the amount of neurons to be used is small compared to large ANNs, making the calculus, to be accomplished in each parallel neuron, to be performed in a quickly and simply way. In this context, the Operating System spent more time on the control of threads than on the processing of each neuron itself, in a parallel manner.

This last bottleneck is corroborated by the graph of results in Fig. 24. Processing times in the NOP Framework 3.0 are higher than in the NOP Framework 2.0, due to the allocation and control process of thread in multicore. Furthermore, as shown in the graph, the more cores are used, higher are the average processing time.

However, excluding the technical shortcomings presented in the implementation of the NOP Framework 3.0 itself, the transparency in the distribution of generated code is a positive point. In this way, such process can also be studied and applied in NOP materializations, both in software and hardware that can overcome these shortcomings. That said, it is suggested that NOP becomes relevant to the construction of ANNs, due the plausibility of the presented approach.

6.2 Future work in software

The prototypical NOP Programing Language and Compiler [31] was not used in this paper as well. However, preliminary results of this NOP materialization demonstrates that NOP allows fast processing even in mono-processed implementations presenting results with superior performance than the NOP Framework 2.0. This happens due to the avoidance of expensive data structures. Still, recent changes in NOP Compiler, allowed results batter than code written in usual C/C++ programs.

As matter of fact, the non-use of the NOP Language and Compiler in this paper is a factor that reflects the performance disadvantage of the ANN NOP implementation when compared with the ANN with traditional C language. Furthermore, the present article aims to demonstrate the use of Frameworks in the construction of a parallelizable ANN, in a transparent way, without taking account the execution performance. Of course, as much as the solution of the NOP Programming Language and Compiler evolves from its prototypical state, the Neuro-NOP will be implemented by means of this materialization.

In addition, a multi-threaded version of NOP Language and Compiler is in development. Actually, some first experiments of the Neuro-NOP in this context seem to be promising. Nevertheless, such a prototype is in an initial version and is very incipient for its use in this work.

6.3 Future work in hardware

Beyond the NOP materializations in software, new NOP materializations have been developed in hardware, such as NOP Digital Hardware (NOP-DH) Framework in VHDL, which was tested in FPGA. This NOP-DH Framework has allowed parallel implementation of NOP applications in hardware [35, 36]. Still, modifications on the NOP Language and Compiler, even in a limited version, already generate VHDL code directly.

In this future version of ANN in hardware, using NOP-DH Framework, OS-imposed bottlenecks will not be applicable. As its execution will take place directly inside

an FPGA chip, a differentiated access to the memory is used. Still, the process of controlling multiple parallel processes is done in a naturally way, according to the constructive origin of the reconfigurable hardware.

In addition, the notification-oriented computer architecture (NOCA), developed by Linhares et al. [16], is a new computer architecture solution that executes NOP software from program memory in sets of specialized NOP cores, such as *Premise* Cores and *Condition* Cores. In addition, NOCA is a viable alternative for software execution with parallelism, with compatible or better performance to Von Neumann model implementations [16]. Still, modifications on the NOP Language and Compiler, even in a limited version, already generate Assembly that executes in the NOCA developed in FPGA or in the NOCA simulator developed in software.

Actually, in all NOP materializations in hardware, the inherent decoupling present in NOP makes the parallelization transparent. This is very useful especially when the algorithms and/or systems present elements that inspire parallelism. In this context, ANNs are natural application candidates [51–54]. Therefore, in future works, it is believed that will be possible to implement parallelized Neuro-NOP algorithms in reconfigurable hardware in an quite easy and transparent way because the design would be made in NOP high-level language. The effectiveness of the implementation of ANNs in FPGAs is a fact, proven in works such as [55–57].

In this sense, the development of a new ANN definition language based on notifications is a goal of future works. This proposal would use the concepts of NOP to create a new language for describing ANNs, thus enabling the programming to be still more decoupled and transparently parallelizable for the ANN developers. In this new language, the basic NOP concepts will be used to model interneuron synaptic connections. This will enable to run the model in NOP-DH, as much as other NOP hardware solutions [16, 33].

Acknowledgements The authors would like to thank the original creators of NOP (Prof. J. M. Simão and Prof. P. C. Stadzisz) as well as the NOP research group particularly the developers of NOP Framework 2.0 (A. F. Ronszcka and G. Z. Valença). In addition, by funding researchers, the authors would like to acknowledge the Araucaria Foundation, Capes and UTFPR.

References

1. Nilsson NJ (2009) The quest for artificial intelligence: a history of ideas and achievements, 1st edn. Cambridge University Press, New York
2. Haykin S (2008) Neural networks and learning machines, 2nd edn. Pearson, Porto Alegre
3. Baptista D, Abreu S, Freitas F et al (2013) A survey of software and hardware use in artificial neural networks. Neural Comput Appl 23:591–599. https://doi.org/10.1007/s00521-013-1406-y
4. Park DC, El-Sharkawy MA, Marks RJ et al (1991) Electric load forecasting using an artificial neural network. IEEE Trans Power Syst 6:442–449. https://doi.org/10.1109/59.76685
5. Krose B, Van Der Smagt P (1996) An introduction to neural networks, 8th edn. University of Amsterdan, Amsterdam
6. Mellit A, Pavan AM (2010) A 24-h forecast of solar irradiance using artificial neural network: application for performance prediction of a grid-connected PV plant at Trieste, Italy. Sol Energy 84:807–821. https://doi.org/10.1016/j.solener.2010.02.006
7. Omondi AR, Rajapakse JC, Bajger M (2006) FPGA Neurocomputers. In: FPGA implementations of neural networks. Springer, Dordrecht
8. Lapuschkin S (2016) The LRP toolbox for artificial neural networks. J Mach Learn Res 1:1–5
9. Arvind August D, Pingali K et al (2010) Programming multicores: do applications programmers need to write explicitly parallel programs? IEEE Micro 30:19–33. https://doi.org/10.1109/MM.2010.54
10. Belmonte DL, Linhares RR, Stadzisz PC, Simão JM (2016) A new method for dynamic balancing of workload and scalability in multicore systems. IEEE Lat Am Trans 14:3335–3344. https://doi.org/10.1109/TLA.2016.7587639
11. Valerievich BA, Anatolievna PT, Alekseevna BM, et al (2017) Modern approaches to the development parallel programs for modern multicore processors. In: 2017 6th Mediterranean conference on embedded computing (MECO). pp 1–4
12. Gabbrielli M, Martini S (2010) Programming languages: principles and paradigms, 1st edn. Springer, London
13. Seiffert U (2004) Artificial neural networks on massively parallel computer hardware. Neurocomputing 57:135–150. https://doi.org/10.1016/j.neucom.2004.01.011
14. Gupta G, Sohi GS (2011) Dataflow execution of sequential imperative programs on multicore architectures. In: Proceedings of the 44th annual IEEE/ACM international symposium on Microarchitecture. pp 59–70
15. Borkar S, Chien AA (2011) The future of microprocessors. Commun ACM 54:67. https://doi.org/10.1145/1941487.1941507
16. Linhares RR, Simão JM, Stadzisz PC (2015) NOCA: a notification-oriented computer architecture. IEEE Lat Am Trans 13:1593–1604. https://doi.org/10.1109/TLA.2015.7112020
17. Simao JM, Stadzisz PC (2009) Inference based on notifications: a holonic metamodel applied to control issues. IEEE Trans Syst Man Cybern A Syst Hum 39:238–250. https://doi.org/10.1109/TSMCA.2008.2006371
18. Simão JM, Banaszewski RF, Tacla CA, Stadzisz PC (2012) Notification oriented paradigm (NOP) and imperative paradigm: a comparative study. J Softw Eng Appl 5:402–416. https://doi.org/10.4236/jsea.2012.56047
19. Simão JM (2005) A contribution to the development of a HMS simulation tool and proposition of a meta-model for holonic control. Ph.D. Thesis. CPGEI at UTFPR, Curitiba, PR, Brazil and CRAN at UHP, France
20. Oliveira RN, Roth V, Henzen AF et al (2018) Notification oriented paradigm applied to ambient assisted living tool. IEEE Lat Am Trans 16:7. https://doi.org/10.1109/TLA.2018.8327425
21. Mendonça ITM, Simão JM, Wiecheteck LVB, Stadzisz PC (2015) Método para Desenvolvimento de Sistemas Orientados a Regras utilizando o Paradigma Orientado a Notificações. In: 12th Congresso Brasileiro de Inteligencia Computacional—CBIC. Curitiba, Paraná, Brazil
22. Nissen S (2017) Fast artificial neural network library. http://libfann.github.io/fann/docs/files/fann-h.html. Accessed 12 Dec 2017

23. Schütz F, Fabro JA, Lima CRE, et al (2015) Training of an artificial neural network with backpropagation algorithm using notification oriented paradigm. In: 2015 Latin-America congress on computational intelligence, LA-CCI 2015. IEEE Comput. Soc, Curitiba, Paraná, Brazil, pp 1–6
24. Heaton J (2015) Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks, 1st edn. Heaton Research Inc, Chesterfield, MO
25. Fisher RA (1936) The use of multiple measurements in taxonomic problems. Ann Hum Genet 7:179–188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x
26. Vyas S, Upadhyay D (2014) Identification of Iris plant using feed forward neural network on the basis of floral dimensions. IJIRSET 3:18200–18204. https://doi.org/10.15680/IJIRSET.2014.0312062
27. Simão JM, Stadzisz PC (2008) Notification Oriented Paradigm (NOP)—a notification oriented technique to software composition and execution. Patent Number PI0805518-1, INPI, Brazil
28. Ronszcka AF, Banaszewski RF, Linhares RR et al (2015) Notification-oriented and rete network inference: a comparative study. In: Proceedings of 2015 IEEE international conference on systems, man, and cybernetics, SMC 2015. IEEE, Hong-Kong, China, pp 807–814
29. Simão JM, Tacla CA, Stadzisz PC (2009) Holonic control metamodel. IEEE Trans Syst Man Cybern A Syst Hum 2(39):1126–1139. https://doi.org/10.1109/tsmca.2009.2022060
30. Ronszcka AF, Valença GZ, Linhares RR et al (2017) Notification-oriented paradigm framework 2.0: an implementation based on design patterns. IEEE Lat Am Trans 15:2220–2231
31. Ronszcka AF, Ferreira CA, Stadzisz PC et al (2017) Notification-oriented programming language and compiler. In: Simpósio Brasileiro de Engeharia de Sistemas Computacionais. Curitiba/PR
32. Ferreira CA (2015) Linguagem e compilador para o Paradigma Orientado a Notificações (PON): avanços e comparações. M.Sc. Thesis. PPGCA at UTFPR, Curitiba, PR
33. Peters E, Jasinski RP, Pedroni VA, Simão JM (2012) A new hardware coprocessor for accelerating notification-oriented applications. In: FPT 2012–2012 international conference on field-programmable technology. IEEE Computer Society, Seoul, Korea, pp 257–260
34. Kerschbaumer R, Linhares RR, Simão JM et al (2017) Notification-oriented paradigm to implement digital hardware. J Circuits Syst Comput. https://doi.org/10.1142/S0218126618501244
35. Simão JM, Linhares RR, de Witt FA et al (2012) Paradigma Orientado a Notificações em Hardware Digital. 13p. Patent number BR 10 2012 026429 3, INPI, Brazil
36. Pordeus LF, Kerschbaumer R, Linhares RR et al (2016) Notification oriented paradigm to digital hardware. Sodebras 11:116–122
37. Ronszcka AF (2012) Contribuição para a concepção de aplicações no paradigma orientado a notificações (PON) sob o viés de padrões. M.Sc. Thesis. CPGEI at UTFPR, Curitiba, PR
38. Sudha N (2015) Multicore processor: architecture and programming. In: 2015 19th international symposium on VLSI design and test. IEEE, pp 1–2
39. O'Sullivan B, Stewart D, Goerzen J (2008) Concurrent and multicore programming. In: Real world Haskell. O'Reilly Media, New York
40. (2000) IEEE Standard for Information Technology—portable operating system interface (POSIX)—part 1: system application program interface (API)-Amendment J: Advanced Real-time Extensions [C Language]. IEEE Std 10031j-2000 0_1-88. https://doi.org/10.1109/ieeestd.2000.91855
41. Izeboudjen N, Bouridane A, Farah A, Bessalah H (2012) Application of design reuse to artificial neural networks: case study of the back propagation algorithm. Neural Comput Appl 21:1531–1544. https://doi.org/10.1007/s00521-011-0764-6
42. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323:533–536
43. Atanassov K, Krawczak M, Sotirov S (2008) Generalized net model for parallel optimization of feed-forward neural network with variable learning rate backpropagation algorithm. In: 2008 4th international IEEE conference intelligent systems, IS 2008, pp 1616–1619
44. Ludwig Jr O, Montgomery E (2007) Neural networks foundations and applications with C programs, 1st edn. Ed. Ciência Moderna
45. Raymond ES (2003) The art of unix programming, 1st edn. Addison-Wesley, Boston
46. Hughes C, Hughes T (2003) Parallel and distributed programming using C++, 1st edn. Addison-Wesley, Boston
47. Barreto WRM, Vendrami ACBK, Simao JM (2018) Notification oriented paradigm for distributed systems. In: Anais do Computer on the Beach. Florianópolis/SC
48. Pordeus LF (2017) Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações. M.Sc. Thesis. CPGEI at UTFPR, Curitiba, PR
49. Yun H, Gondi S, Biswas S, Corporation B (2016) BWLOCK: a dynamic memory access control framework for soft real-time applications on multicore platforms. Trans Comput 66:1247–1252. https://doi.org/10.1109/TC.2016.2640961
50. Akhter S, Roberts J (2006) Multi-core programming: increasing performance through software multi-threading. Intel press, USA
51. Pethick M, Liddle M, Werstein P, Huang Z (2003) Parallelization of a backpropagation neural network on a cluster computer. In: 15th IASTED international conference on parallel and distributed computing and systems. IASTED/ACTA Press, Marina Del Rey, California, pp 574–582
52. Gargouri A, Krid M, Masmoudi DS (2013) Hardware implementation of new bell-shaped pulse mode neural network with on-chip learning and application to image processing. Int J High Perform Syst Archit 4:132–143. https://doi.org/10.1504/IJHPSA.2013.055224
53. Lakshmi KP, Subadra M (2013) A survey on FPGA based MLP realization for on-chip learning. Int J Sci Eng Res 4:1–9
54. Rezvani R, Katiraee M, Jamalian AH et al (2012) A new method for hardware design of multi-layer perceptron neural networks with online training. In: Proceedings of 11th conference on cognitive informatics & cognitive computing, pp 527–534. https://doi.org/10.1109/icci-cc.2012.6311205
55. Abrol S, Mahajan R (2015) Artificial neural network implementation on FPGA chip. Int J Comput Sci Inf Technol Res 3:11–18
56. Nedjah N, da Silva FP, de Sá AO et al (2016) A massively parallel pipelined reconfigurable design for M-PLN based neural networks for efficient image classification. Neurocomputing 183:39–55. https://doi.org/10.1016/j.neucom.2015.05.138
57. Baptista FD, Morgado-Dias F (2017) Automatic general-purpose neural hardware generator. Neural Comput Appl 28:25–36. https://doi.org/10.1007/s00521-015-2034-5