An Object-Oriented Neural Network Toolbox Based on Design Patterns

Christian Napoli(✉) and Emiliano Tramontana

Department of Mathematics and Informatics, University of Catania, Viale Andrea Doria 6, 95125 Catania, Italy
{napoli,tramontana}@dmi.unict.it

Abstract. Generally, the resolution of a problem by using soft-computing support requires several attempts for setting up a proper neural network. Such attempts consist of designing and training a neural network and can be a relevant effort for the developer. This paper proposes a toolbox that automates several steps for setting up a neural network, and provides high-level abstractions allowing a developer to choose classical network topologies and configure them as desired, as well as design a neural network from a scratch. A valuable aspect of our solution is given by the modularity of the whole design that builds on object-orientation and design patterns.

Keywords: Artificial intelligence · Soft-computing · Modularity · Software design · Software evolution

1 Introduction

The main characteristics of a neural network are the *topology* and the *working mechanism*. The topology characterises the neurons and their interconnections (synaptic links, feedbacks, delay lines, etc.), whereas the working mechanism consists of the algorithms used for training and, after that, to obtain coherent outputs. Such a working mechanism is chosen according to the topology of the trainee network. Different categories of topologies and training procedures have been proposed and are useful for categories of classification or prediction problems.

Generally, the best specific topology for solving a problem is unknown, therefore often the first training of a neural network is unsuccessful. In such a case, some policies of growth or pruning have to be applied, which involve the creation or elimination of one or more neurons. Moreover, each time a neuron is created or removed the network topology changes and a training phase is needed again (at least partially). Hence, while topology and training are functionally connected to each other, the set up for each should be made possible independently of the other. Hence, the developer should be able to choose abstractions and compose a novel behaviour by setting these up (Booch and Maksimchuk 2007).

As a solution to the above modularity requirement, this work presents an object-oriented toolbox that provides the user with a level of abstraction appropriate to design

G. Dregvaite and R. Damasevicius (Eds.): ICIST 2015, CCIS 538, pp. 388–399, 2015.
DOI: 10.1007/978-3-319-24770-0_34

neural networks for several classification and prediction problems. The proposed solution takes into account the important features of neural networks and supports their modular instantiation, hence providing reusable abstractions and modules where the training support can be reused and adapted to different topologies.

2 Neural Networks

A neural network is basically composed by a set of simple interconnected *functional units* called neurons. The topology resulting by the said connections allows us to have a specialised network performing some tasks such as: modelling physical phenomena (Bonanno et al. 2014c), performing predictions (Borowik et al. 2015), enhancing forecast correctness (Napoli et al. 2014a), filtering signals (Nowak et al. 2015; Gabryel et al. 2015) and images (Napoli et al. 2014b; Wozniak and Polap 2014; Wozniak et al. 2015a), etc. (Haykin 2004). A neural network processes data trying to emulate the human brain model, by partially reproducing the human connectome with the said functional units that we call neurons. Functionally, a neural network builds some knowledge out of training data and applies such a knowledge for further use on the same data domain. Hence, the field of neural networks gives us a different approach to problem solving with respect to standard algorithmic. This latter is based on a conventional set of instructions in well defined deterministic steps that are followed in order to reach the solution, while in general a neural network approach is not deterministic. In fact, it is not possible to program a neural network in order to perform a specific task without any need for training. Moreover, in the majority of cases, even the topology is unknown beforehand. Then, in order to use neural networks a user must be prepared to a try-and-check procedure that could require a certain amount of interactions.

Nevertheless, when a problem is successfully managed by a topology, such a topology could be used for a range of problems in the same domain. Moreover, a specifically trained neural network with such a topology could be reused to continue the computation on the same data domain. E.g. in order to predict the power generation of photovoltaic power plants the topology results similar among different plants, while each plant will use a specifically trained neural network that has learnt how to model the environmental conditions and variations on that specific site.

3 The Developed Toolbox

The presented toolbox is designed in order to be modular and robust to changes, and such properties have been achieved thanks to *design patterns*, i.e. software solutions that have been proven successful to solve recurrent problems in the field of object-oriented programming (OOP), described in terms of needed classes and their relationships (Gamma et al. 1994). In order to obtain a design for creating, training, using and managing neural networks, we have to take into account the big number of various topologies and training strategies available, the different sizes, and the different fields which the network can be applied to. The most important requirements are: (i) a neural network can have an arbitrary size and can use different kinds of neurons, (ii) neurons

are usually organised into layers, and connected to each other in a custom way, (iii) connections between neurons have their own characterisation, e.g. a connection could link two neurons or link a delay to a neuron and vice versa, and finally, (iv) neurons could use different activation functions according to the layer they belong to.

Out of the said requirements, we have designed the following main class hierarchies, and connected them according to three design patterns.

- Class `NeuralNetwork` represents a neural network consisting of some layers.
- Class Layer represents a layer, i.e. a given number of neurons, and a matrix of weights for the synaptic links to each neuron.
- Class Topology is a class that is meant to create neural networks consisting of layers that are interconnected in some specific way.
- Class TransferFunc is the root of a class hierarchy where each class provides an activation function (e.g. Tansig, Logsig and Radbas) and implements the proper methods for training and computation.

The following design patterns are used.

- *Factory Method* is used to organise the creation and interconnection of neurons inside layers, hence the initialisation decisions that allow to establish a given topology.
- *Decorator* is used to customise the interconnections between neurons.
- *Bridge* is used to let neurons choose one among several activation functions.

The said classes and patterns (see Fig. 1) are detailed in the following sections.

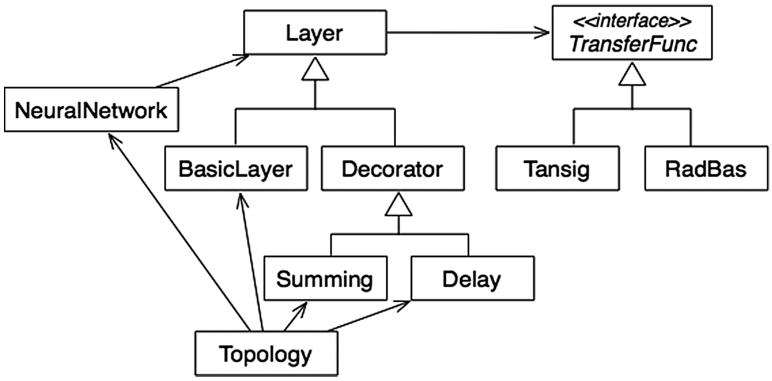

Fig. 1. The UML class diagram of the proposed toolkit

4 Details of Classes

4.1 Class Layer

The very core of the toolbox consists of class Layer and its subclasses. They are responsible to hold neurons and compute the output of such neurons. The main method is

compute() and implements the algorithm that performs the computation on the neurons according to input data (stored into field inputs), and stores output data into field outputs, accessible by means of method getOutputs(). Typically, a layer calls getOutputs() on another layer, therefore an instance of Layer holds a reference to another instance of Layer as field source. Additional methods of Layer are: trainStep() performing a single iteration of computation during training, initWeights() and setWeights(w) giving values to weights. Method initWeights() is called when Layer is instantiated, in order to initialise the synaptic weights and store them into field weights, such weights would be modified and recomputed during each training epochs by method trainStep() and then stored by using method setWeights(), taking as input the new weights and updating the weights matrix. Class Layer plays the role of *Component* for design pattern Decorator, hence several responsibilities can be additionally executed, and available as implementations in proper *ConcreteDecorator*s Summing and Delay (see Sect. 5.2 for details). Moreover, class Layer plays the additional role *Abstraction* for design pattern Bridge, in order to let the subclasses use different transfer functions (see Sect. 5.3 for details), and its subclasses have the additional role *RefinedAbstraction*. Therefore, Layer holds a reference to an instance of a class implementing a transfer function, i.e. interface *TransferFunc*, that will be used to compute the output of neurons. Having a separate class for computing the transfer function allows us to easily configure layers in order to use one among several available transfer functions.

4.2 Class TransferFunc

Since a neural network emulates the behaviour of its human counterpart, in the same manner an artificial neuron emulates the behaviour of a neural cell. The latter is a constantly stimulated cell that needs to reach some kind of threshold in order to activate itself. In an artificial neuron such a threshold is mathematically emulated by a so called *activation function* or *transfer function*. The two different names are used according to the point of view, i.e. whether we focus on the activity of one neuron only, or on the functionalities of the entire network of neurons. In the latter case, we call it a transfer function, since it will determine how the input data will change and interact while passing from a layer to the next. The neuron behaviour is then univocally identified by its activation function that gathers data from the linked neuron in the preceding layer and gives results to feed data into the neurons in the successive layer. A description both general and comprehensive of all activation functions is overwhelming since, a transfer function could be any function defined within a finite range and absolutely integrable, therefore any function in $L^1(\mathbb{R})$ (Gupta et al. 2004). Nevertheless, not all the functions in $L^1(\mathbb{R})$ are equally performant or reliable for a neural network. In general, in a feedforward neural network, a transfer function can be described as a function $f:\mathbb{R} \to \mathbb{R}$ so that the neuron returns an output in the form:

$$y_i = f\left(\sum_{j=1}^{N} wi_j x_j\right)$$

where index i identifies the computing neuron and index j the connected neurons from where the inputs x_j are coming. Each input is (in general) weighted before it is used to compute the output. It follows that when considering one input layer and two hidden layers, respectively of M and N neurons, then, the output will be in the form of

$$y_i = f_1\left(\sum_{j=1}^{N} w_{ij} f_2\left(\sum_{k=1}^{M} w_{jk} x_k\right)\right)$$

Then, it is possible to define the output of a neuron in the n-esime layer, recursively:

$$\begin{cases} y_{k_0}^{(0)} = u_{k_0} \\ y_{k_1}^{(1)} = f_1\left(\sum_{k_0} w_{k_0}^{k_1} y_{k_0}^{(0)}\right) \\ y_{k_n}^{(n)} = f_1\left(\sum_{k_{n-1}} w_{k_{n-1}}^{k_n} y_{k_{n-1}}^{(n-1)}\right) \end{cases}$$

so that a neural network becomes a function

$$N[\mathbf{u}] = N\left[\left(u_{k_0}\right)\right] = \left(y_{k_n}^{(n)}\right) = y.$$

For such a kind of topology a wide range of different transfer functions have been proven useful. Some excellent examples are the sigmoidal functions like

$$f(x) = \frac{A}{1 + e^{-k(x-u)}}$$

where A represents the maximum amplitude of the function, k is a fixed parameter and μ the value for the mean point of the sigmoid itself. Another kind of interesting activation function is represented by the radial basis function family. The most commonly used radial basis function is of gaussian type and is formalised as follows

$$\rho(x) = e^{-(\sigma x)^2}$$

on the other hand, a well enough approximation is used in the field of neural networks so that

$$f(x) = \sum_l w_l \rho\left(\|x - \mu_l\|\right) = \sum_l w_l e^{-\left(\sigma\|x-\mu_l\|\right)^2}$$

The latter function requires a different training procedure with respect to the previous one, it is indeed noticeable that also the set of parameters $\{\mu_l\}$, called centroids, must be adjusted during training.

It follows that different kinds of transfer functions require different training methods, also according to the number of fixed parameters. Moreover, certain transfer functions

could be applied to different kinds of topologies which, sometime, not only differ on the number of neurons and links, but also, or solely, on the kind of computation which are performed on different layers (Haykin 2009). A typical example is given by the comparison between a feed forward network using radial basis functions and a radial basis network, i.e. while they use the same transfer function on the first hidden layer (as the name suggests), they use the second hidden layer in a completely different way (that is the reason why the neuron in the two hidden layers of a RBF network are traditionally called pattern units and summation units) (Musavi 1992).

While this description is far from comprehensive or complete, it should give an insight on how the different choices regarding the training and operation of a neural network strictly depends on the topology and the activation function itself. It is then paramount to structure the code in an appropriate manner in order to both encapsulate the transfer function and the related responsibilities, while being still able to change the behaviour of other modules according to the chosen topology and functions.

4.3 Class Topology

The first recognisable feature of a neural network is trivially the topology. In the past decades, an entire zoology of different topologies has been developed, some topology categories are standard (see Fig. 2), others are highly customised. Therefore, not all the neural networks are organised similarly, and, even when some networks present topological similarities the functionalities are sometimes very different, moreover different networks are trained and used in different manners.

The encapsulation of the algorithms specifying topology details for a neural network is achieved by means of class Topology. Such a class implements methods that instantiate and interconnect Layers in a specific way, and also neurons with each others, hence creating a topology. Each given classical neural network topology has its own method, such as e.g. getFeedForward(), named after the related topology. All the knowledge on the topology is encapsulated into such methods and all other classes unaware of the topology issues.

Class Topology implements a variant of design pattern Factory Method, therefore class Topology plays role *ConcreteCreator*, and class Layer plays role *Product*. Typically, several instances of Layer are needed at once to form a neural network, hence references to such instances are inserted into an instance of NeuralNetwork. Moreover, each instance of Layer is given a reference to another instance acting as the preceding layer, the *ConcreteCreator*'s factory methods *inject dependencies* into Layer classes. In the following we will give some examples of the factory methods implemented within class Topology (e.g. getFeedForward(), getRecurrent(), getHopfield(), etc.), which create topologies by interconnecting layers in different ways.

Feed Forward Neural Networks. The feed forward topology is depicted in Fig. 2(a) and presents the easiest concept. It is implemented by means of subsequent layers, each layer communicates the output of each constituting neuron to the neurons of the next layer (Bonanno et al. 2014b). Therefore, to obtain the said interactions among instances of layers, a factory method dubbed getFeedForward(int l, int[] nl) is provided to instantiate a number l of Layers (l is the first input parameter), each having the number of

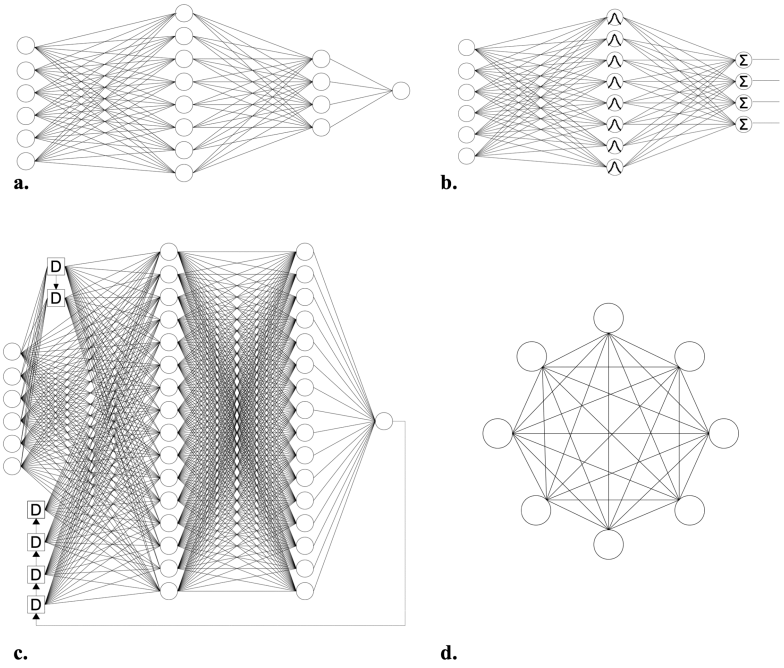

Fig. 2. Different neural network topologies: (a) feed forward, (b) radial basis, (c) recurrent, (d) Hopfield (1982). Note that networks (a), (b) and (c) are organised in layers, and belong to the same topology category, moreover, while (a), (b) and (c) share some topological commonalities their training and use is completely different.

neurons given as the i-sime position of array nl, and provide each Layer instance with a reference of the former Layer instance.

Radial Basis Neural Networks. Figure 2(b) shows a neural network using the radial basis configuration (Napoli et al. 2014c), in which each layer has its own purpose and functions in a different manner than another layer (Napoli et al. 2015). Therefore, a factory method getRadialBasis() provides the needed implementation for instantiating Layers and interconnecting them, similarly to getFeedForward(). Differently to the latter, getRadialBasis() sets the RBF transfer function for the instantiated layers, by providing the created instances of Layer with a reference to an instance of RadBas.

Recurrent Neural Network. Differently from the other categories, such a topology (Fig. 2(c)) presents feedbacks and delay lines (Williams and Zipser 1989; Bonanno et al. 2012). While a layer gets the outputs of a preceding layer in the same manner of a feed forward neural network, such a topology uses a different strategy to store delayed data and feed such data to the neurons linked to the related delay lines (Napoli et al. 2013b). For the related implementation, a layer is created by means of class Delay (appropriately storing data for a time step). Then such an instance of Delay is provided as reference to the other Layers.

The given examples are not comprehensive neither exhaustive of all the possible topology of a neural network. Such examples were intended to explain our design choices. Due to its high level of encapsulation and loose class coupling, our proposed toolbox can be easily expanded with other topologies and activation functions. Further application-dependent classes that implement custom topologies, functions and training methods can be also provided to our toolbox.

5 Details of Design Patterns

5.1 Factory Method

The main intent of design pattern Factory Method is to define an interface, *Creator*, and some classes, *ConcreteCreator*, to instantiate a class implementing a common interface *Product.* A *ConcreteCreator* encapsulates the algorithm selecting one among several implementations of *Product*, dubbed *ConcreteProduct*. As a result, client classes are unaware of the used *ConcreteProduct* class, and such classes can be freely changed with each other (Gamma et al. 1994).

We have used design pattern Factory Method, in our solution, to encapsulate the selection of the implementation of Layer that has to be instantiated, in such a way that client classes remain independent of the implementation. Moreover, each implemented *ConcreteCreator* will be responsible to provide instances of Layer with the reference to each other, and therefore *injects dependencies* into them.

A class playing as *Creator* and *ConcreteCreator* is Topology (see Fig. 1), which lets the client class obtain a NeuralNetwork. Additional topologies are implemented each as a factory method of Topology. This approach is far more flexible than connecting classes at design time to a specific implementation, e.g. giving a BasicLayer means to access type RadBas, which generates a tight coupling between such classes. Moreover, the number of dependencies for a class could expand, e.g. BasicLayer would have to handle both Tansig and RadBas. More generally, tight coupling generates undesired consequences and less reusable code.

5.2 Decorator

The main intent of design pattern Decorator is to have the possibility to add responsibilities to a class dynamically and with a more flexible mechanism than that provided by inheritance. The solution suggests the following roles for classes: *Component* defining the operations that all objects have, *ConcreteComponent* implementing the object to which functionalities are added, *Decorator* holding a reference to *Component*, *ConcreteDecorator* implementing an additional responsibility (Gamma et al. 1994). The abstraction provided by interface Layer let us keep a general representation of a layer, independent of its number of neurons and connections. The needed connections among neurons, possibly with delayed lines, are additional responsibilities that can be dynamically added to the basic behaviour of a layer. Accordingly, we have implemented the basic behaviour of a layer by class BasicLayer playing the role *ConcreteComponent* for the Decorator design pattern,

then the additional responsibilities are implemented as classes Summing and Delay playing as *ConcreteDecorator*s. Class Summing computes a weighted sum of the outputs, whereas class Delay stores the outputs for a time step. By means of the mechanisms provided by design pattern Decorator, the responsibilities of different classes can be accurately selected to extend the functionalities of instances of BasicLayer.

5.3 Bridge

The aim of design pattern Bridge is to decouple abstractions and implementations, in such a way that both can change independently. The solution provides role *Abstraction* defining the interface for client classes and holding a reference to *Implementor*, role *RefinedAbstraction* as a subclass of *Abstraction*, role *Implementor* defines the operations that the *Abstraction* uses, and its implementations are provided as role *ConcreteImplementor* (Gamma et al. 1994). While the other implemented patterns are concerned on the topology of the neural network in terms of layer sizing and connections (with special care for delays and feedbacks), the activation function for the neurons is selected by using the Bridge design pattern. The Bridge let us decouple the implementation of the different activation functions from the related abstractions. This behaviour is far more flexible with respect to the simple inheritance since this latter ultimately binds the implementation and the abstraction permanently in the code. The Bridge gives us means to separately modify, extend, manage and maintain the classes and methods related with the use of different activation function and, consequently, decoupling the topology of the network from its usage and the implemented functions. E.g., it is possible to have a simple multilayer topology with no feedbacks or delays: while this could be used for a feed forward neural network, the same topology could implement a radial basis neural network, although the use and training of those two network is completely different, as well as their purpose and management. It follows from the implementation of the Bridge pattern that the class Layer has to hold a reference to a *ConcreteImplementor*, therefore to an object of type TransferFunc, on the other hand it is possible in such a way to hide the implementation details to the client once again separating the concerns of the different classes (Tramontana 2013).

6 Related Works

Other object oriented solutions have been presented in projects and toolboxes for the implementation of neural networks. One of the most known is Neuroph (Ševarac 2012): a Java framework for creating, train and use neural networks with different topologies and training methods. This project, which begun in 2008, proposes an open source solution which is still evolving and expanding its potential. Such a software solution has increased its complexity during time, essentially because of the high degree of class coupling. While we agree on many of the identified abstractions, a better modularity could be achieved by using design patterns. Neuroph is mainly organised into three layers: one for the GUI and two for the neural network library and core. The library implements the supported neural

networks, while the core consists of two packages containing the base classes, which provide structure and functionalities to the created networks, and the utilities for training and management. We note that in it several coding practices should have been avoided and some of them are actually *smells*, for refactoring techniques (Fowler et al. 1999; Pappalardo and Tramontana 2013). It should be stated that creating, managing and training a neural network is a complex task affected by several difficulties. The UML class diagram of Neuroph shows a cyclical dependency of classes Neuron, Layer, and NeuralNetwork. Other cyclical dependencies can be found e.g. for classes NeuralNetwork and LearningRule, and for classes Neuron and Connection. Such a practice should be avoided in a well engineered software system. Moreover, tight coupling could be avoided by recurring to some design patterns, e.g. for the dependencies among classes Layer and NeuralNetwork, as well as for class Connections. Of course, it is not straightforward to refactor such a mature software. The approach presented in this paper aims at having a highly maintainable system which can be further enhanced, thanks to its modularity (Napoli et al. 2013a). Consequently, by using design patterns, while encapsulating the different needed abstractions we also successfully manage to separate the different concerns. The resulting system has classes that depend on interfaces and not implementations. This in turn makes it possible to include new features (e.g. new kind of topologies, transfer functions, training methods, etc.). Finally, out approach aims to build a system that can be easily ported into a distributed and parallel infrastructure (Bonanno et al. 2014a; Napoli et al. 2014d), making good use of Java multithreading and distributed supports.

7 Conclusion

This paper has shown the design of a toolbox that allows instantiating a neural network and configuring it as desired. The provided components automate some important design decisions, such as the topology, while also allowing the developer to configure the network as desired, as well as create a neural network from a scratch. Our solution has put a high relevance on the modularity issues, hence the proposed components are loosely coupled and can be easily be refined, by adding more classes to the provided classes hierarchies. The devised design choices, based on design patterns Factory Method, Decorator and Bridge, will easily grip on new available classes, without additional connecting code. In our future work, the modularity in our design will support us in distributing objects implementing parts of a neural network on a parallel and distributed infrastructure.

References

Booch, G., Maksimchuk, R.A.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Reading (2007)

Bonanno, F., Capizzi, G., Napoli, C.,: Some remarks on the application of RNN and PRNN for the charge-discharge simulation of advanced Lithium-ions battery energy storage. In: International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), pp. 941–945 (20–22 June 2012). doi:10.1109/SPEEDAM.2012.6264500

Bonanno, F., Capizzi, G., Sciuto, G.L., Napoli, C., Pappalardo, G., Tramontana, E.: A novel cloud-distributed toolbox for optimal energy dispatch management from renewables in IGSs by using WRNN predictors and GPU parallel solutions. In: International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), pp. 1077–1084 (18–20 June 2014) (2014a). doi:10.1109/SPEEDAM.2014.6872127

Bonanno, F., Capizzi, G., Coco, S., Napoli, C., Laudani, A., Sciuto, G.L.: Optimal thicknesses determination in a multilayer structure to improve the SPP efficiency for photovoltaic devices by an hybrid FEM - Cascade Neural Network based approach. In: International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), pp. 355–362 (18–20 June 2014) (2014b). doi:10.1109/SPEEDAM.2014.6872103

Bonanno, F., Capizzi, G., Sciuto, G.L., Napoli, C., Pappalardo, G., Tramontana, E.: A cascade neural network architecture investigating surface plasmon polaritons propagation for thin metals in OpenMP. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2014, Part I. LNCS, vol. 8467, pp. 22–33. Springer, Heidelberg (2014c)

Borowik, G., Wozniak, M., Fornaia, A., Giunta, R., Napoli, C., Pappalardo, G., Tramontana, E.: A software architecture assisting workflow executions on cloud resources. Int. J. Electron. Telecommun. **61**(1), 17–23 (2015). doi:10.1515/eletel-2015-0002

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)

Gabryel, M., Woźniak, M., Damaševičius, R.: An application of differential evolution to positioning queueing systems. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) Artificial Intelligence and Soft Computing. LNCS, vol. 9120, pp. 379–390. Springer, Heidelberg (2015)

Gamma, E., Helm, R., Johnson, R., Vlissiders, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)

Gupta, M., Lian, J., Noriyas, H.: Static and Dynamic Neural Networks: from Fundamentals to Advanced Theory. Wiley, New York (2004)

Haykin, S.: Neural Networks: a Comprehensive Foundation. Pearson, London (2004)

Haykin, S.: Neural Networks and Learning Machines. Pearson, London (2009)

Hopfield, J.: Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. **79**(8), 2554–2558 (1982)

Musavi, M.T.: On the training of radial basis function classifiers. Neural Netw. **5**(4), 595–603 (1992)

Napoli, C., Pappalardo, G., Tramontana, E.: Using modularity metrics to assist move method refactoring of large systems. In: IEEE International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS), pp. 529–534 (July 2013) (2013a). doi:10.1109/CISIS.2013.96

Napoli, C., Pappalardo, G., Tramontana, E.: Improving files availability for bittorrent using a diffusion model. In: 23rd IEEE WETICE Conference (WETICE), pp. 191–196 (23–25 June 2014) (2014a). doi:10.1109/WETICE.2014.65

Napoli, C., Pappalardo, G., Tramontana, E., Marszalek, Z., Polap, D., Wozniak, M.: Simplified firefly algorithm for 2D image key-points search. In: IEEE Symposium on Computational Intelligence for Human-like Intelligence (CIHLI), pp. 1–8 (9–12 December 2014) (2014b). doi:10.1109/CIHLI.2014.7013395

Napoli, C., Pappalardo, G., Tramontana, E.:"An agent-driven semantical identifier using radial basis neural networks and reinforcement learning. In: XV Workshop Dagli Oggetti agli Agenti (WOA 2014), Catania, Italy (25–26 Sepember 2014) (2014c). doi:10.13140/2.1.1446.7843

Napoli, C., Pappalardo, G., Tramontana, E., Zappalà, G.: A cloud-distributed GPU architecture for pattern identification in segmented detectors big-data surveys. Comput. J. bxu147 (2014) (2014d). doi:10.1093/comjnl/bxu147

Napoli, C., Pappalardo, G., Tramontana, E., Nowicki, R.K., Starczewski, J.T., Woźniak, M.: Toward work groups classification based on probabilistic neural network approach. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) Artificial Intelligence and Soft Computing. LNCS, vol. 9119, pp. 79–89. Springer, Heidelberg (2015)

Napoli, C., Pappalardo, G., Tramontana, E.: A hybrid neuro–wavelet predictor for QoS control and stability. In: Baldoni, M., Baroglio, C., Boella, G., Micalizio, R. (eds.) AI*IA 2013. LNCS, vol. 8249, pp. 527–538. Springer, Heidelberg (2013b)

Nowak, B.A., Nowicki, R.K., Woźniak, M., Napoli, C.: Multi-class nearest neighbour classifier for incomplete data handling. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.). LNCS, vol. 9119, pp. 469–480Springer, Heidelberg (2015)

Pappalardo, G, Tramontana, E.: Suggesting extract class refactoring opportunities by measuring strength of method interactions. In: IEEE Asia Pacific Software Engineering Conference (APSEC), pp. 105–110 (December 2013)

Ševarac, Z.: An open source software framework for neural network development. Neuroph. **11**(43), 40–44 (2012)

Tramontana, E.: Automatically characterising components with concerns and reducing tangling. In: QUORS workshop at Compsac. IEEE, pp. 499–504 (2013)

Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Comput. **1**(2), 270–280 (1989)

Woźniak, M., Połap, D., Gabryel, M., Nowicki, R.K., Napoli, C., Tramontana, E.: Can we process 2D images using artificial bee colony? In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) Artificial Intelligence and Soft Computing. LNCS, vol. 9119, pp. 660–671. Springer, Heidelberg (2015a)

Wozniak, M., Polap, D.: Basic concept of cuckoo search algorithm for 2D images processing with some research results: an idea to apply cuckoo search algorithm in 2D images key-points search. In: International Conference on Signal Processing and Multimedia Applications SIGMAP 2014, pp. 157–164, Setubal (2014). doi:10.5220/0005015801570164