A Generic Neural Network Framework Using Design Patterns

by

Stefan van der Stockt

Submitted in partial fulfilment of the requirements for the degree
Master of Science (Computer Science)
in the Faculty of Engineering, Built Environment and Information Technology
University of Pretoria, Pretoria

October 2007

A research publication of

C I R G

Computational Intelligence Research Group

Visit the research group online at

cirg.cs.up.ac.za

A Generic Neural Network Framework Using Design Patterns

by

Stefan van der Stockt
E-mail: stefanvd@za.ibm.com

Abstract

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. This task is even more daunting for a developer of computational intelligence applications, as optimising one design objective tends to make others inefficient or even impossible. Classic examples in computer science include 'storage vs. time' and 'simplicity vs. flexibility.' Neural network requirements are by their very nature very tightly coupled – a required design change in one area of an existing application tends to have severe effects in other areas, making the change impossible or inefficient. Often this situation leads to a major redesign of the system and in many cases a completely rewritten application. Many commercial and open-source packages do exist, but these cannot always be extended to support input from other fields of computational intelligence due to proprietary reasons or failing to fully take all design requirements into consideration.

Design patterns make a science out of writing software that is modular, extensible and efficient as well as easy to read and understand. The essence of a design pattern is to avoid repeatedly solving the same design problem from scratch by reusing a solution that solves the core problem. This pattern or template for the solution has well-understood prerequisites, structure, properties, behaviour and consequences. CILib is a framework that allows developers to develop new computational intelligence applications quickly and efficiently. Flexibility, reusability and clear separation between components are maximised through the use of design patterns. Reliability is also ensured as the framework is open source and thus has many people that collaborate to ensure that the framework is well designed and error free.

This dissertation discusses the design and implementation of a generic neural network framework that allows users to design, implement and use any possible neural network models and algorithms in such a way that they can reuse and be reused by any other computational intelligence algorithm in the rest of the framework, or any external applications. This is achieved by using object-oriented design patterns in the design of the framework.

Keywords: artificial intelligence, artificial neural network, computational intelligence, software engineering, design pattern, taxonomy, incremental learning, sensitivity analysis, sensitivity analysis incremental learning algorithm, neural network library, CILib.

Supervisor : Prof. A. P. Engelbrecht
Department : Department of Computer Science
Degree : Master of Science

If people do not believe that mathematics is simple, it is only because they do not realise how complicated life is.

– *John Louis von Neumann*

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

– *Martin Fowler*

Acknowledgements

I would like to express my sincere thanks to the following people for their assistance during the production of this dissertation:

- The Lord God Almighty, for giving me the capability and insight to be able to complete this study.
- My parents, Etienne and Charlene, for their continuous support while researching this topic.
- Professor A. P. Engelbrecht, my supervisor, for his insight and motivation.
- My IBM managers, Dion Harvey and Ashraf Davids, for their understanding and support.
- My friends and work colleagues, whose help and support meant a lot.

This dissertation was produced with the use of the following open source and freeware software tools. Special thanks to the authors of these superb software packages:

Typesetting using LaTeX 2$_\varepsilon$
Bibliographic references maintained using BibTeX
Operating system Linux

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

If the human brain were so simple that we could understand it, we would be so simple that we couldn't.

— *Emerson M. Pugh*

Man has always been intrigued by the notion of intelligent computers, ranging from the ideas of science fiction writers who dreamed of having conversations with artificial beings that could reason and even feel, to the realists who wanted a computer that could solve problems that they themselves could not. The Turing test [111] was proposed by Alan Turing as a well-defined test for a machine's capability to demonstrate thought. The test assumes two parties, a human and a machine, that are engaged in conversation with another human via a chat program. The machine is said to pass the test if the human cannot reliably distinguish which one of the two parties is the machine and which one is the human. While a machine as described by Turing has still not been developed, extraordinary progress has been made in artificial intelligence in the last 50 years. Today the field is so immense that it has been split into subfields such as artificial life [114], robotics [13], expert systems [55], case based reasoning [96], among others.

One such subfield is computational intelligence (CI). The CI field consists of four areas, namely neural networks (NN) [50], evolutionary computation (EC) [5], [37], swarm intelligence [7] and fuzzy systems [120]. Neural networks is used as a collective name for topologies and algorithms that are based on biological neural systems found in living animals and humans. Evolutionary computation is a field where scientists simulate

the way that natural organisms evolve and adapt over time to overcome environmental constraints. Swarm intelligence is the study of the dynamics in a natural social system such as birds in a flock, a school of fish or a colony of ants. Finally, fuzzy systems involves reasoning about problems and their solutions in such a way that the values of 'true' and 'false' are not binary – a statement's logical validity can be a real value in fuzzy logic as opposed to 1 or 0 in Boolean logic, the reason being that uncertainties are being reasoned about.

The field of CI has enjoyed a lot of success, particularly in recent years. Neural networks have been used to solve diverse problems such as classification problems found in data mining [99], biochemistry [81], least cost routing in the telecommunications industry [45], scheduling of airline crew [70], spam prevention in email [9], circuit design [124], intrusion detection for computer systems [95], [102], game logic [109], [116], financial applications [18], and medical research. An interesting medical example [1] used a radial basis function network for phosphene localisation. The system involves an artificial retina implant that translates images into signals which are used to stimulate the optic nerve, which in turn generates phosphenes at different locations of the patient's visual field. It is shown that neural networks show promising results in helping to rehabilitate blind patients.

Even with all these useful applications it is not easy to find applications or code libraries that provide users with *all* the functionality they want. Commercial and open-source applications exist that allow users to train and apply certain CI algorithms. The main problems with these commercial packages are that they are often difficult or impossible to extend or modify, and that they don't always allow users to easily integrate different areas of CI (such as neural networks and swarm intelligence for example).

In 2001, Edwin Peer started an open source project called the Computational Intelligence Library, or CILib, as part of a Masters study [85]. The aim of CILib is to fully integrate the areas of CI into one coherent library where code could be reused efficiently and any CI algorithm could interface with any other algorithm. At the time of writing, CILib included a vast amount of knowledge in the fields of particle swarms, evolutionary computation, game theory and ant colony optimisation to name but a few. As a result of this need for integration of all CI components, there is also a fair number of data struc-

tures and algorithms that are implemented generically. These allow any CI algorithm to interface with other components by providing a standard class hierarchy and enforcing the same set of tools such as containers and types to increase interoperability.

This dissertation discusses the design and implementation of a generic neural network framework in CILib that allows users to design, implement and use any possible neural network architectures and algorithms in such a way that they can reuse and be used by any other CI algorithm components in the rest of CILib (or any external[1] application). This is achieved by using object-oriented design patterns in the design of the framework which maximises its reusability and extensibility.

1.1 Motivation

There are two ways that people can obtain neural network applications – they can write their own implementation, or reuse other implementations. Both of these options face the same fundamental challenge during design time of having requirements that are very difficult to articulate in a complete and precise way. This section discusses the fundamental dilemmas that neural network developers face. A short discussion on the use, availability and capability of existing neural network packages and libraries is also given.

1.1.1 Design Dilemmas

The implementation of a neural network model involves the same basic steps as with the construction of any other application. This includes deciding what the application's key requirements are, designing a solution that will satisfy all the requirements, and developing the application. It is advantageous to follow a methodology as part of a software development lifecycle. Numerous such methodologies exist, of which extreme

[1] As will be seen in chapter 4, CILib is completely open and external applications can reuse these components.

programming (XP)[2] and IBM's Rational Unified Process (RUP)[3] are two well-known examples.

A person who wants to implement a specific neural network model or a framework that supports quite a couple of neural network topologies and algorithms merely has to find out what the exact requirements are, design an application that addresses these requirements, implement it and test it.

This is where the design and implementation of neural network models turns out to be very illusive.

Typical neural network implementations are mostly custom written ad-hoc applications developed by the users who need them. These applications have as many features as their creators decided to incorporate. A designer would implement a neural network application based on a set of requirements that is considered to be complete. Later during a refactoring phase it would frequently be the case that requirements are expanded or contracted. If the underlying structure of the neural network needs to change (like a change in topology (neurons and weights), learning algorithm, data sources, or any other logic), the designer has to change the program source code. Frequently one would find that the original application design could not cater for the required change, as it was never a design requirement. The effect of this is usually devastating to a neural network implementation design. Often this situation leads to a major redesign of an existing system, or in many cases a completely new application.

Researchers such as Kazmierczak, Senyard and Sterling [61] agree that one of the big problems of developing neural networks in an ad-hoc manner using 'trial and error' and 'build and fix' approaches is that success is difficult to repeat. This means that developers who employ such methods have to start from scratch everytime a new network has to be developed, both from a development code base as well as problem solving experience.

Challenges such as those mentioned above occur because all components in a neural network are functionally very tightly coupled. Most modern applications have a very loosely coupled design. An example is a word processor. If one changes the font rendering or line breaking components in a word processor, the spell checker will still work correctly,

[2]See http://www.extremeprogramming.org for more information on XP.

[3]See http://www.ibm.com/software/awdtools/RUP for more information on RUP.

as these changes do not affect the spell checker functionality. In such an application it is easy to define levels of abstraction and to separate the application functionality cleanly into those layers.

In any neural network model implementation this is not always the case.

Each component of a neural network implementation is greatly dependent on the slightest detail of one or more other components. A few examples of such dependencies are included below. This list is by no means exhaustive. To make matters worse, it frequently happens that trying to meet one requirement invalidates one or more of the others. This is due to the tight coupling found in neural network systems. The details of what these challenges entail can be seen in chapters 2 to 6.

- **Net input signal:** The net input signals of neurons can include many types such as product units or summation units in the case of feedforward networks. Other network topologies use different types of net input signals, such as self organising maps that use distance metrics like Euclidean distance. The choice of net input signal directly influences network output, weight training algorithms, architecture selection algorithms, the type of data that a neuron can process, as well as the choice of activation function. A change in net input signal has significant consequences to the operations of other components, such as changes in learning algorithm mathematics, architecture selection, network evaluation, data sets and data types throughout the network, active learning algorithms and network topology. Developers need to design their applications to be flexible enough to facilitate these changes, yet also provide an efficient system.

- **Activation functions:** Related to the net input signal is the activation function of a neuron. The type of function makes a profound difference in how one would, for example, train a network with gradient descent optimisation. It may also influence the heuristics that are used in certain active learning schemes. Some neuron types such as the bias unit in feedforward networks give a constant output, while others such as input neurons need to emit what they receive (perhaps adjusted with a scaling factor). Output may also be restricted to certain ranges. Adaptive activation functions as discussed in [27] also allow these functions to be learned dynamically, which in turn may influence the entire system.

- **Neuron interpretation:** Some network types such as feedforward networks require bias units while other networks with the same physical topology layout do not require them. The designer has to keep track of the bias units and the 'real' neurons in these cases.

- **Recurrent neurons:** Recurrent neurons and layers of neurons require a history of their previous output, while most other neurons do not. If the designer decides to incorporate recurrent functionality in a neural network implementation during a refactoring stage, it may be very difficult or inefficient to change the design to achieve this.

- **Topologies:** Neurons need to be connected together to form a network. Many types of connection schemes exist, ranging from fully connected topologies to bipartite topologies to partially connected topologies, as well as other types of connection schemes. Weight connections can also potentially span from any neuron to any other neuron, either unidirectional or multidirectional. Furthermore, *ensembles* of topologies are possible, which allow multiple topologies to be used together and their results combined in a multitude of ways. *Modular* neural networks are similar in that many types of neural network topologies are combined into a single coherent network. Both ensemble and modular neural network members can be considered as neurons on a high level. Designers of a neural network implementation need to consider these points in their designs.

- **Weights:** The data types of weights may not always be the same, even in the same network topology. These can vary from being discreet, real, binary, complex or even vector values to name but a few. As weights values are the inputs to net input signal functions in neurons, this dependency needs to be enforced (especially in a dynamic environment that changes during runtime). Past weight values also need to be kept track of in many applications. In some neural learning schemes it may be required that the training algorithm be specified down to the individual weight connection level.

- **Growing/pruning:** Adding neurons or weights to, or removing neurons or weights

from a layer of neurons (either at runtime or statically) needs to involve integrity checks such as:

  - adding and removing the weights associated with the relevant neuron,
  - making sure that training algorithm implementations will still work on the increased/decreased number of neurons (changes in the mathematics, array sizes, error functions involving neuron outputs, and other details),
  - making sure the data source still has the correct number of inputs and outputs for the network topology, that patterns are still configured correctly and that any components that use patterns are still in working order.
  - clients that are loosely coupled to a neural network system need to be informed in some manner when a change to a topology occurs. An effective way to do this is publish/subscribe (sometimes called an *observer*), but the neural network system must provide this functionality.

  If growing/pruning functionality is added to the network design after the topology's implementation, it may be very difficult or impossible to implement efficiently.

- **Learning strategies:** There exists a multitude of ways to train any specific neural network model. These include learning strategies such as gradient descent based training algorithms, evolutionary algorithms, particle swarm optimisation, leapfrog optimisation, among others. Neural network topology design needs to take these different learning methods into account. Learning approaches may vary, as is the case for batch (offline) and stochastic (on-line) training.

- **Data sets:** There are many different collections of patterns in neural network models such as training sets, generalisation sets, validation sets and candidate sets. This leads to many possible ways of distributing data into these sets (such as random distributions, K-fold clustering, specific assignments, and many others).

- **Sources of data:** Data sources of neural networks may include many different types such as flat files, databases, network locations, generated data, web services, XML files, application output, and others. The system needs to make an abstraction here so that changes in data sources do not influence the rest of the system.

- **Pattern presentation and ordering:** Different types of pattern presentation exist. Broadly speaking, there are a multitude of ways that patterns can be ordered in data sets. Active learning and passive learning techniques are also possible and need to be taken into account. All these techniques require information from various sources across the neural network model in order to calculate how to perform its role. A neural network framework needs to be flexible enough to allow these components to be changed without affecting the rest of the system.

- **Measurements:** Measurements (such as a network's training error) need to be recorded when running simulations. Many of these measurements are common to the majority of neural network types, and some measures are dedicated to certain models only. More than one measurement can be active simultaneously. Simulation engines need to perform integrity checks as well as allow many active measures at any given time. New measurements must be easy to add without disrupting the rest of the system and measures should be reusable.

- **Stopping conditions:** Many different stopping conditions exist, of which some might be relevant to certain neural network models only. More than one stopping condition might be active simultaneously as well. More stopping conditions must be easy to add and reuse.

- **Parallel computing:** Some simulations can be very large and time consuming. Running simulations in parallel can speed execution time up substantially. However, the logic of distributing, collecting and correlating results needs to be included in the simulation manager. Exceptions such as machines rebooting, network failure, timeouts, and other problems also need to be handled. The neural network components themselves need to take clustering into account, which is easy to do via design patterns and object-orientation if the requirement is taken into account early. Retro-fitting a neural network implementation to support parallel computing may not be feasible in many cases.

- **Integration and integrity:** There are many complexities that arise when integrating different neural network components that were written by different developers using different programming models, including challenges such as:

  - Data integrity when different components use different data structures to represent information. An example would be a component that reads data from a source and returns a linked list of values, and a component that performs training on a neural network that accepts an *array* of values.
  - A composite application (i.e. integrating many separate components to form a larger application) needs to be tested thoroughly as components are not always guaranteed to work together – a common development model is needed.
  - Development time is increased as a developer would need to learn how a new component works, as well as learning a different programming model in many instances.

  In short, unnecessary time and effort is spent trying to adapt different programming models to work together, which might lead to development errors, slow performance, lack of flexibility, difficult maintenance and applications that cannot be extended easily (or at all).

- **Use cases:** Neural networks need to be trained before they are useful in applications such as financial forecasting, routing and others. Developers need to decide what type of neural network to use, which learning algorithm gives the best results (which will be problem specific) and finally provide an application component that provides the functionality of the trained neural network. If developers are not careful, they need to develop a neural network implementation for *each* type of neural network (say a_1 types), as well as each learning algorithm (say a_2). This could mean up to $a_1 \times a_2$ separate neural network implementations need to be developed, plus another application that uses the trained neural network.

  A generic neural network framework will allow developers to develop solutions much faster, as various neural network types can be tested with various different learning algorithms by merely configuring existing components. Certain components may also be reusable in a final user application, alleviating the need to rewrite the neural network functionality.

These are but a few of the dilemmas facing a neural network designer. Often, designers would rather write a new application to implement a specific neural network rather than modify an existing implementation. Modifying an existing implementation is often more time consuming and difficult due to the nature of the type of problems mentioned above.

Many researchers have started to address the challenges of neural network development and show how existing software engineering principles can address neural network development. Dart, Senyar and Sterling [14] present a unified framework in the form of an extention to the capability maturity model (CMM) [52], which is known as the neural network process model (NNPM). CMM is a world-renowned framework designed to guide developers in improving the manner in which software solutions are developed. CMM consists of 5 maturity levels, where each level addresses certain aspects of the development process. The first level consists mostly of from ad-hoc development processes, and process capability progresses cumulatively through levels to the fifth level, which significantly raises the maturity level by addressing and optimising issues such as testing strategies, separation of environments into development testing and production environments, performance management, reference models for developed implementations, amongst many other aspects. NNPM aims to provide a neural network development-specific specification of CMM to aid neural network developers to implement neural network applications more concisely and based on sound software engineering practices.

Kazmierczak, Senyard and Sterling [61] propose methods that can be used to both build neural network applications based on requirements, as well as solve specific problems. These methods also focus on the validation and verification of learning, allowing a more structured approach to building and using neural networks to solve problems. Emphasis is placed on the fact that there must be separation between the generic and application specific components of a neural network.

There are also frameworks which aid in the specification and development of specific neural network models. Schikuta [97] shows how Rumbaugh's object modelling technique (OMT) [94] can be used for the specification and implementation of neural networks using software engineering principles. Essentially, OMT is used to model the desired behaviour of a neural network model and, using software engineering tools, is used to generate a

neural network application outline in either the C or C++ language.

1.1.2 Neural Network Packages

This section gives a short overview of some well-known neural network packages and libraries. An exhaustive overview is not given, as the aim of this dissertation is not to provide a catalogue of available neural network libraries and packages. Rather, some well-known examples are listed, their main capabilities are expressed, and challenges and shortcomings that these packages have are highlighted.

There are a multitude (literally thousands) of neural network packages available to developers today. These can be classified into two main categories, namely *commercial* systems and *open-source* or *freeware* systems. These systems can again be classified into those that merely provide a library of neural network models that developers can use in their own projects, or applications that use proprietary neural network libraries to perform tasks such as data mining, or stock price trend analysis.

Some examples of well-known neural network libraries include the Stuttgart Neural Network Simulator (SNNS) [121] developed by the University of Stuttgart. SNNS includes about 20 types of neural network models and related training models that were developed using the C language. Another example is the neural network toolbox for MATLAB [16] that enables MATLAB users to train and use a host of predefined neural network models and algorithms. A well-known Java neural network package is Java Object Oriented Neural Engine (JOONE) [76] which consists of a modular system for constructing neural network models.

While these systems are all useful in their own right, they all tend to be designed and constructed with merely neural networks as a field of computational intelligence in mind. Connecting neural network models to other areas of CI such as particle swarm optimisation and evolutionary computation is in most cases very difficult or inefficient to do. While some libraries allow interactions with other CI algorithms, these algorithms typically reside *outside* of the neural network library, creating the need for integration. The results of such integration methods may include:

- Slow performance as there may be no efficient way of connecting a particular neural network library to a custom CI implementation or existing CI library.

- A high probability of integration errors, as developers have to manually integrate a neural network library with a CI library or custom CI implementation to obtain the desired result. This may happen, for example, if the two libraries use different data representation (such as weight value sequences), data structures, amongst other aspects.

- Long development cycles occur, as developers will in most cases have to start from scratch if they need to integrate a neural network model with a new CI implementation. Subsequently, the same amount of work will have to be performed for every new CI algorithm, as each new CI algorithm will have its own methods of operation that will have to be integrated.

- Closely related to the previous point, is the fact that the neural network library and other CI libraries most probably will not share a common base framework (or metamodel). This may severely limit the possibilities of reuse, the level of abstraction that is supported (in terms of neural networks, examples would be to have access to weights on a neuron- or layer level), and possible interaction patterns between the libraries.

- As the neural network and external CI algorithms are in different libraries or implementations, it will be difficult and inefficient to try and run simulations in a parallel computing environment.

Furthermore, it is not always possible or easy to extend or modify a given neural network model that resides in a library. This is mainly due to the dilemmas discussed above, and sometimes for legal reasons too. Also, most neural network libraries typically allow neural network models to be *trained*, but provide no way to use them in custom applications. In these cases developers have to rely on their own means to incorporate a neural network engine in their applications.

This dissertation aims to present and extend a framework called CILib that allows developers to easily connect neural network implementations to CI implementations *that reside in the same framework* in a structured way using design patterns. This allows algorithms to be developed very quickly, easily and error-free.

1.2 Scope

The aim of this dissertation is to lay a foundation of typical relationships and interactions found in neural networks today, and show how developers of neural network implementations can avoid the design problems and dilemmas as discussed above. This is achieved by extending and reusing a CI framework called CILib that aids developers to write CI algorithms in an efficient and reusable manner. The aim of this dissertation is not to create a neural network library. Rather the design of a *framework* for such a library is presented. Some complete neural network implementations are developed as examples of how the framework can be used. This dissertation has the following core objectives:

- Present a conceptual breakdown of existing neural network models, taking into account their architecture, data requirements, training methods and operating characteristics.
- Briefly discuss object-oriented design patterns and show how patterns can be used to design an effective generic neural network framework, using the conceptual breakdown as requirements.
- Give a brief overview of CILib's objectives, benefits and how the system works.
- Discuss the design and implementation of a generic neural network framework in CILib that uses design patterns and how this framework addresses the core requirements set by the conceptual breakdown of neural network models.
- Show some advanced implementations in the neural network framework, such as modular and ensemble networks, active learning, architecture selection, as well as external learning algorithms for neural networks such as other CI techniques. A new performance metric is also added to the framework. These provide good examples of how existing components can be reused to easily create completely different neural network models by merely extending existing interfaces. As the CILib metamodel is used, these new components are also fully reusable.
- Illustrate how to easily create neural network implementations in a controlled manner by using the framework as guidance. In particular it shows how the framework

guides development and greatly reduces errors by detecting incorrect configurations. Reuse of existing neural network components by extending their functionality or create new components that can be used by the existing system is also shown.

Note that the entire software development lifecycle, software engineering methodology, use cases, and the application of the framework to iterative problem solving are not in scope – only the actual technical design of the generic neural network framework is covered with emphasis on design decisions, benefits and trade-offs.

1.3 Dissertation Layout

The remainder of this dissertation is organised as follows:

- **Chapter 2:** A brief overview of neural networks is given and a conceptual breakdown is presented. This breakdown states the functional requirements of typical neural network models known today.
- **Chapter 3:** A discussion of the design patterns used in CILib and the generic neural network framework is presented. An overview is given of the problem that each pattern addresses along with an example of where it can be used to ease neural network implementation design.
- **Chapter 4:** The layout and general operation of the CILib framework is discussed.
- **Chapter 5:** The implementation of the generic neural network framework is discussed. Continuous reference to chapter 2, 3 and 4 shows how the framework addresses the key requirements listed in these chapters.
- **Chapter 6:** Some advanced implementations are discussed, including topics such as active learning, modular and ensemble networks, network growing and pruning as well as training neural networks using external training approaches such as PSO. The ease of extending a neural network system with a complicated metric such as receiver operating characteristics (ROC) is also presented. Throughout the chapter, the reuse and efficiency of the framework and CILib is highlighted.

- **Chapter 7:** The dissertation conclusion is given and further work ideas are discussed.

Chapter 2

Artificial Neural Networks

> *Computational properties of use to biological organisms or to the construction of computers can emerge as collective properties of systems having a large number of simple equivalent components (or neurons). The physical meaning of content-addressable memory is described by an appropriate phase space flow of the state of a system. A model of such a system is given, based on aspects of neurobiology but readily adapted to integrated circuits.*
>
> — *Extract from [50] by J.J. Hopfield, 1982, as one of the pioneers of modern neural networks.*

An *artificial neural network* is, as the name suggests, a *network* of connected *artificial neurons*. An *artificial neuron* in turn is a single processing element that accepts one or more inputs and produces an output signal – most neurons generate signals in the range (0,1). A fully functional artificial neural network is constructed by connecting these artificial neurons together. Many types of topologies are possible depending on how these artificial neurons are connected together. A topology that is often used, called a *layered* topology, is shown in figure 2.1 to illustrate how artificial neurons can be connected.

As this dissertation always refers to *artificial* neural networks (as opposed to the biological neural networks that inspired scientists such as Hopfield), the term 'artificial' is omitted for convenience. This chapter gives a brief conceptual overview of neural networks and all the components that make up a neural network system. The aim of

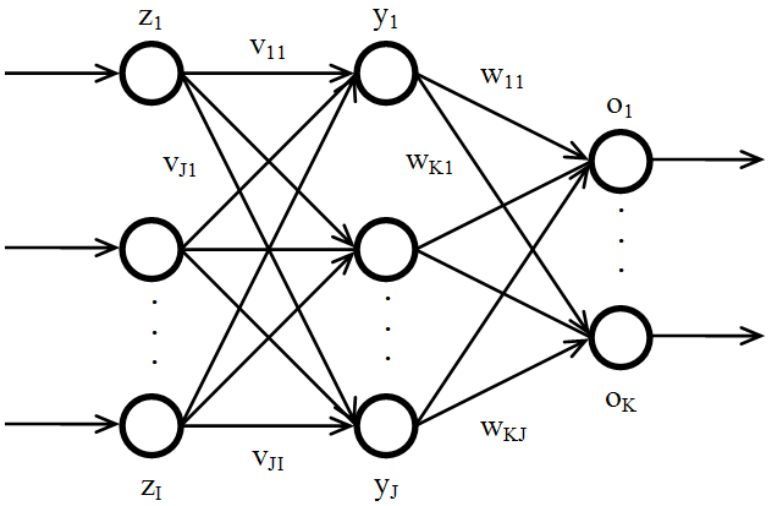


Figure 2.1: A network of connected neurons

this overview is to give a broad understanding of the *structure* of neural network systems as opposed to an exhaustive overview of all the known possibilities for each section. The focus is on highlighting the main components of a neural network system, as well as common high-level relationships between these components. For example, in section 2.2.3 which considers active learning, the characteristics of the active learning system are discussed instead of a full listing of all the known approaches to active learning.

What is a neural network system and what are all of the components that a developer needs to consider? Fiesler [33] states that a neural network system can be described by a four-tuple, namely its topology, its constraints, its initial state and its transition function. The topology describes the network's layers and neurons (i.e. its *frame*) and the interconnection scheme between layers and neurons. Network constraints describe the ranges of weights and transfer functions, while the network's initial state allocates starting values to these components. Lastly, transition functions describe the methods to reach successive network states, including neuron transfer functions, learning rules, clamping functions as well as changes to network topology (called an *ontogenic* function).

Taking this into account, any neural network system can fundamentally be divided into three main components. The rest of this chapter gives an overview of neural networks by discussing how a neural network system can be described by these three components,

which are

1. a *topology component* that encompasses the neuron and weight connection frameworks,

2. a *data component* that outlines different ways that data can be presented to a neural network, and

3. a *learning component* that stipulates various ways in which a neural network can be trained.

Developers can use this breakdown as a requirements specification to construct neural network implementations.

2.1 The Topology Component

A neural network topology comprises of the different connection schemes and neuron types that together form a neural network. The topology defines the relationships between neurons by means of their weight connections. This connectionist approach can be formally defined by using Graph Theory[1] as stated in [34]. Neurons are viewed as vertices and the weight connections are edges between the neuron vertices. This approach is discussed in more detail in chapter 5.

Formally (see [34]), a neural network topology consists of a framework of neurons and its interconnection structure (or weight connection framework). Note that topology and architecture are used interchangeably in this dissertation.

2.1.1 Neuron Framework

The neuron framework consists of one or more layers of *neurons* that are interconnected by *weight connections*.

[1] See [19] for a full overview of Graph Theory

The Neuron

A *neuron* is a single processing unit inside a neural network. A typical neuron consists of many components and processes as can be seen in figure 2.2:

- a number of input weight connections (the total number of input weights is known as the *fan-in* of the neuron),
- a number of output weight connections (the total number of output weights is known as the *fan-out* of the neuron),
- a *net input signal* that consolidates the input weights and input values,
- an *activation function* to generate an output signal based on the net input signal,
- a *scaling and limiting* phase may also be found (optional),
- a *competition* phase where neuron output is determined based on external factors such as other neuron outputs (optional).

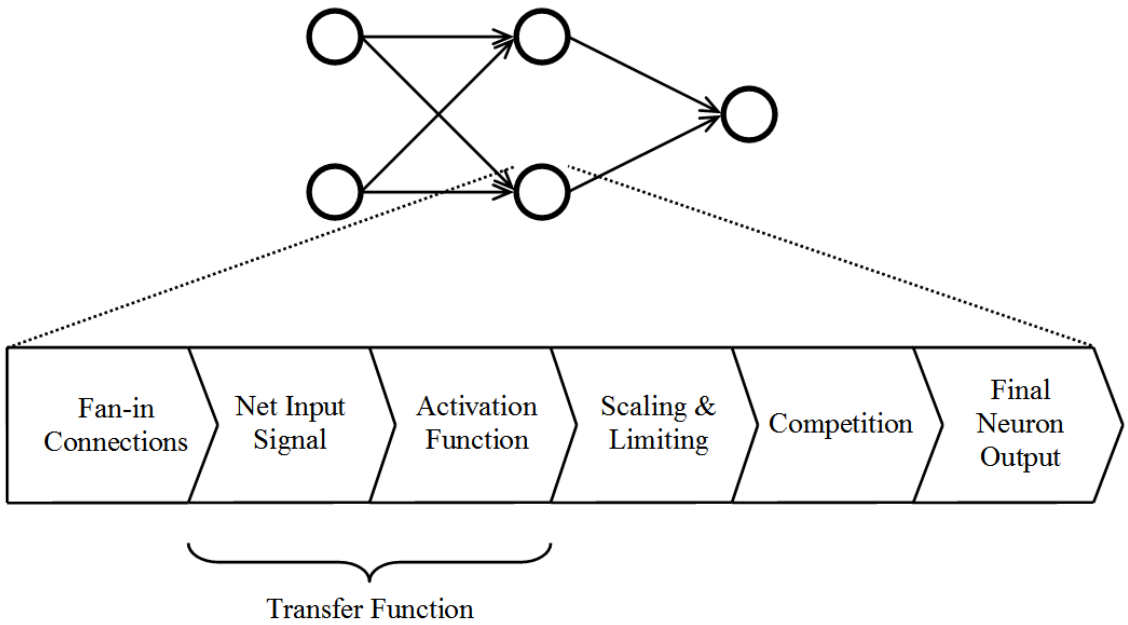


Figure 2.2: Processes in a neuron

The input weights of a neuron consists of all the weight connections that provide information to the neuron and is an important concept as the output of a neuron is computed by directly using these weight values. The *degree*, d_n, of a neuron n is the sum of the fan-in, d_n^{IN}, and the fan-out, d_n^{OUT} of the neuron. That is,

$$d_n = d_n^{IN} + d_n^{OUT} \tag{2.1}$$

The *net input signal* of a neuron is a mathematical transformation that takes the input weight strengths and the output of all the respective input neurons to compute a net neuron signal. Duch and Jankowski [20] give a full taxonomy of net input signals. These include:

- The inner product net input signal is defined as $net = I(\mathbf{z}_p, \mathbf{w}_n) = \mathbf{w}_n \cdot \mathbf{z}_p$ where $\mathbf{z}_p$ is the input vector and $\mathbf{w}_n$ is the input weight vector of neuron n.
- The distance-based net input signal defined as $net = D(\mathbf{z}_p, \mathbf{t}_p) = \|\mathbf{z}_p - \mathbf{t}_p\|$ where $\mathbf{z}_p$ is the input vector and $\mathbf{t}_p$ is the target vector.
- A combination of distance-based and inner product, such as $net = A(\mathbf{z}_p, \mathbf{w}_n, \mathbf{t}_p) = \alpha \mathbf{w}_n \cdot \mathbf{z}_p + \beta \|\mathbf{z}_p - \mathbf{t}_p\|$.

Net input signal output can be either scalar or vector values, although scalar values are mostly used. The distance-based function is not restricted to the Euclidean norm, so other norms such as quadratic distance functions can also be used. In most cases the net input signal is unbounded and activation functions are used to impose bounds on the neuron's output. The most widely used net input signal (known as the summation unit) is probably the inner product of the input vector $\mathbf{z}_p$ and the weight vector $\mathbf{w}_n$ to give the net input signal of the neuron, namely

$$\begin{aligned} net &= \mathbf{w}_n \cdot \mathbf{z}_p \\ &= \sum_{i=1}^{d_n^{IN}} z_{p,i} w_i \end{aligned} \tag{2.2}$$

where d_n^{IN} is the total number of input weights to the neuron. Another widely used net input signal is the *product unit* as discussed in [21], [57] and [74]:

$$net = \prod_{i=1}^{d_n^{IN}} {z_{p,i}}^{w_i} \tag{2.3}$$

A good example of a distance-based net input signal is the Euclidean distance metric, which is commonly used in unsupervised networks, defined as

$$net = \sqrt{\sum_{i=1}^{d_n^{IN}} (z_{p,i} - w_i)^2} \tag{2.4}$$

The choice of net input signal may have a profound affect on the learning algorithm. For example, as explained in section 2.3.1, gradient descent weight updates depend on the derivative of the net input signal. Thus using equation 2.2 or equation 2.3 makes a dramatic difference in both neural network topology as well as the learning component.

The purpose of an *activation function* is to transform a neuron's net input signal into the neuron's output (or firing strength). The combination of the net input signal and the activation function is known as the neuron's *transfer function* as can be seen in figure 2.2. The value of the activation function is influenced by *net* as well as the threshold value θ, also known as the *bias*.

There are two main types of activation functions, namely

- hard activation functions that include discrete functions such as the step or multi-step functions, and
- soft activation functions that include continuous functions such as sigmoid and radial functions.

Probably the most widely used activation function is the sigmoid function which is defined as

$$f(net - \theta) = \frac{1}{1 + e^{-\lambda(net-\theta)}} \tag{2.5}$$

where λ determines the steepness of the respective function; usually $\lambda = 1$. The range of the sigmoid is $(0, 1)$. A function that shares the shape of the sigmoid function, but has a greater range of $(-1, 1)$, is the hyperbolic tangent function, defined as

$$f(net - \theta) = \frac{e^{\lambda(net-\theta)} - e^{-\lambda(net-\theta)}}{e^{\lambda(net-\theta)} + e^{-\lambda(net-\theta)}} \tag{2.6}$$

and is approximated by

$$f(net - \theta) = \frac{2}{1 + e^{-\lambda(net-\theta)}} - 1 \tag{2.7}$$

A frequently used activation function is the step function, which implements a form of discrete output. The multi-step function can produce any number of discrete output values, while the step function produces a binary output of either β_1 or β_2, as defined by

$$f(net - \theta) = \begin{cases} \beta_1 & \text{if } net \geq \theta \\ \beta_2 & \text{if } net < \theta \end{cases} \tag{2.8}$$

A simple linear activation function is defined as

$$f(net - \theta) = \beta(net - \theta) \tag{2.9}$$

A gaussian function is defined as

$$f(net - \theta) = e^{-(net-\theta)^2/\sigma^2} \tag{2.10}$$

where $net - \theta$ is the mean and σ^2 is the variance of a Gaussian distribution.

A very interesting activation function used in *wavelet networks* [123] uses wavelet functions in the calculation of the hidden unit output. Notice that there is no separate net input signal calculation, only the activation of hidden unit y_j:

$$y_j = \psi\left(\frac{\mathbf{z}_p - \mathbf{v}_j}{a_j}\right), j = 1, \ldots, J \tag{2.11}$$

where J is the total number of hidden units in the wavelet network, $\psi(\cdot)$ is a wavelet function, $\mathbf{v}_j$ and a_j respectively are the translation parameter and dilation parameter of $\psi(\cdot)$. The information of $\mathbf{v}_j$ and a_j is stored as weight values in the neural network, where $\mathbf{v}_j$ is the input to hidden layer weights (see figure 2.1). A commonly used wavelet function, known as the inverse Mexican-hat function, is given by

$$\psi\left(\frac{\mathbf{z}_p - \mathbf{v}_j}{a_j}\right) = (\gamma_j^2 - I)e^{-(\frac{\gamma_j^2}{2})} \tag{2.12}$$

where

$$\gamma_j^2 = \left\|\frac{\mathbf{z}_p - \mathbf{v}_j}{a_j}\right\| = \sqrt{\sum_{i=1}^{I}\left(\frac{z_{i,p} - v_{ji}}{a_j}\right)^2} \tag{2.13}$$

In all the activation functions mentioned, the value of θ is used to determine what corresponding output value will be generated for a given value of *net*. This can be

clearly seen in the step function, where the value of θ categorically decides the value of the activation function from β_1 to β_2.

More complex activation functions exist, such as proposed by Zurada [126] and extended by Engelbrecht *et al.* [27]. Essentially, the same sigmoid function as defined in equation 2.5 is used, but a λ parameter is kept for each hidden and output unit. There is also a ϱ parameter which controls the range of the function. The definition of the sigmoid function now becomes $f(net - \theta, \lambda, \varrho) = \frac{\varrho}{1+e^{-\lambda(net-\theta)}}$. Several learning rules exist that allow the network to learn the optimal λ and ϱ values.

For a taxonomy and discussion on the different types of activation and output functions, see [20].

The terms *scaling and limiting* refer to two optional operations on neuron outputs after their transfer functions have been evaluated. Not all neural network topologies support these operations and they should only be used if their effects are well understood. *Scaling* is used to transform the output of an activation function to a desired range and offset. For example, the range of the sigmoid function in equation 2.5 is $(0, 1)$, but the neuron's final output can be in the range $(-1, 1)$ or $(5, 10)$ depending on the scaling function used. The level of scaling is dictated by the activation functions of the neurons that use the current neuron's output, as these neurons will receive the scaled values as input. *Limiting* is used to decide what the final output of a neuron will be. These can be logical constructs such as stipulating the conditions under which a neuron will keep its previous output value rather than using a newly computed value.[2]

Competition is another optional operation that some network topologies require. An example of this is the linear vector quantiser-I developed by Kohonen [63], where the algorithm requires a 'winning' neuron to be selected based on output values. Only the weights of the winning neuron are updated (see algorithm 2.4). The concept of a 'winning' neuron can also be used in classification problems. Consider, for example, a problem that has six classification outcomes. A feedforward neural network with six output units utilising competition can be used to represent the problem – the winning neuron represents the class that a particular input vector belongs to.

[2]This is sometimes called *clamping*.

Neuron Layers

Most neural networks have a layered topology that typically consists of multiple layers. There are exceptions where no layering is present, but such cases can be regarded as a single layered network as stated in [34]. A layer consists of one or more neurons. In the context of layers, three types of neurons can be distinguished:

- input neurons, which receive external input from data sources,
- hidden neurons, which receive input from other neurons in the network, and
- output neurons, which produce the network's output.

In a similar way, input-, hidden- and output layers contain only input-, hidden- or output neurons respectively. The total number of neurons, N, for all layers, L, is

$$N = \sum_{l=1}^{L} N_l \tag{2.14}$$

where N_l is the number of neurons in layer l.

Neurons in the same layer are not ordered as each neuron has equal importance. Exceptions do exist such as self organising maps [63] [64] (also refer to section 2.3.2) where the relative position of neurons are important. From a programmatical perspective, ordering of neurons is important in cases such as when training is performed by learning algorithms such as a particle swarm optimiser (PSO), as a particle represents all the network's weights in a single linear vector. This is discussed in more detail in section 2.3.1. These weights need to be extracted and inserted into the network in the exact same order every time. The order in which input and output vectors are mapped to an application is also important.

2.1.2 Weight Connection Framework

The weight connection framework specifies the way in which all neurons are linked together to form a neural network. Fiesler [34] specifies four connection types are specified based on a layered topology:

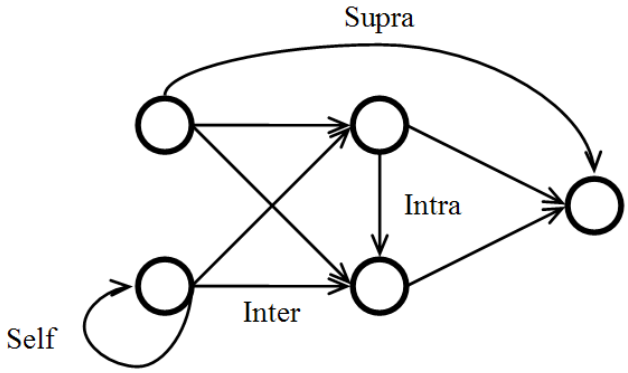


Figure 2.3: Types of weight connections

- Inter-layer connections connect neurons in two adjacent layers (i.e. neuron layer numbers differ by one).
- Intra-layer connections connect neurons in the same layer.
- Supra-layer connections connect neurons in layers that are not adjacent (i.e. neuron layer numbers differ by more than one).
- Selfconnections connect neurons to themselves.

These weight types are illustrated in figure 2.3.

Each connection has a *strength* or *weight* component associated with it. This weight value reflects the importance or influence of the source neuron to the neuron it is connected to. The choice of net input signal and activation function determines which weight values indicate strong or weak influence. Assuming a dot-product net input signal for example, weights that are close to zero indicate that the source neuron has a low influence. Weights with zero values might be removed at a later stage by a process called network *pruning*, which falls under *architecture selection*. Architecture selection involves adding more weights and/or neurons, as well as removing irrelevant and redundant weight connections and/or neurons from the topology. Refer to section 2.3.5 for more information on architecture selection approaches.

Typically, the data types of weight connections are floating-point values. However, this might not always be the case. There are a multitude of connection schemes such as:

- Binary weights can be used where values can be either 0 or 1. Austin and Buckle [2] used a binary weighted neural network is used to recognise features in images. Binary weighted networks have been used to map Boolean functions [17].

- Fuzzy weights are used to build fuzzy number based neural networks. Training of fuzzy neural networks is discussed in [32]. Nauck and Kruse [79] show how to use fuzzy neural networks for function approximation based on supervised learning.

- Complex numbers can be used as weight values. Rattan and Hsieh [89] use complex-valued weights and biases to perform non-linear complex principle component analysis on complex-valued input data.

- Rosen-Zvi and Kanter [92] show how to train networks that have weights with discrete values. Nousis *et al.* [87] show how the use of discrete activation functions can greatly reduce the complexity of neural network hardware implementations.

It is clear that weight values and network input data are not restricted to floating-point values. This fact becomes even more apparent when one considers the fundamental purpose of weight values – a weight is an input to a net input signal calculation. Thus connection weights, net input signals and activation functions have dependencies on each other and together have an influence on all aspects of the neural network system. Examples of aspects that are sensitive to changes in net input signal, activation function or weight values include the calculation of network output, neural learning algorithms, architecture selection, error metrics, objective functions, data sources and the interpretations of network results. This relationship is of key importance in the construction and use of any neural network framework.

Another characteristic of a connection is *symmetry*. A connection can be either unidirectional or multidirectional. A *unidirectional* connection can only transfer information in one direction. A *multidirectional* connection can transfer information in more than one direction. In the multidirectional case a *symmetric* connection (see figure 2.4) has the same weight value in all directions, while an *asymmetric* connection (see figure 2.5) has different weight values for each direction. Unidirectional connections are asymmetric by definition.

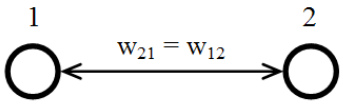


Figure 2.4: Symmetric weight connection

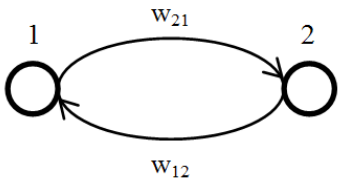


Figure 2.5: Asymmetric weight connection

Fiesler [33], [34] discusses the notion of *high-order* weight connections. High-order weights basically act as a splicing function that combines input weight values in a defined manner that can then become a single input to a neuron. A good example of this is the functional link neural network [41], [54], which is effectively a feedforward neural network with an expanded input layer that includes one or more functional high-order units h_l. Each functional unit h_l takes the original network input vector, $\mathbf{z}_p$, with dimension I as input and its output is a function $h_l(\mathbf{z}_p)$. The output of these functional units is then used as the input to the hidden layer.

Lastly, the initial weight values in a neural network can be initialised in a number of ways. The initial value of the weight vector is very important – initial positions that are close to a minimum will result in faster convergence in many learning algorithms. If gradient descent based approaches are used, then positions on a flat area will result in slower convergence speeds. Also, Horne *et al.* [53] show that too large initial values can saturate neurons prematurely. Well-known initialisation strategies include random initialisation, specific predetermined values and the inverse fan-in rule. The inverse fan-in rule [115] states that the input weights of a neuron should be initialised in the range $[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ where n is the fan-in of the neuron. Network types such as the learning vector quantiser developed by Kohonen [63] initialise weight values to the first input pattern's values.

2.2 The Data Component

Artificial neural networks are used to solve problems. Problems are typically represented as a data set of samples D that satisfy the constraints of the given problem. Examples of data sets include inputs and outputs to an unknown function or a collection of attributes that are related to a specific outcome or classification. This function can be expressed as

$$F_D : Z \rightarrow O \tag{2.15}$$

where Z is the set of input vectors with dimension I and O is the set of output (or target) vectors with dimension K. Thus F_D represents the mapping between input and target data. Sometimes, O is not known or unavailable as is the case for unsupervised networks such as the SOM.

The goal of a neural network is to approximate F_D as well as possible. This is achieved by using a neural learning algorithm to train the network's weight vector $\mathbf{W}$ in such a way that the network output matches F_D for a data set D to within a given error range. This neural network function can be expressed as

$$F_{NN}(D, \mathbf{W}) : Z \rightarrow O \tag{2.16}$$

where D is a data set sampled from F_D. It is not always possible for F_{NN} to approximate F_D precisely, which means that in most cases $\|F_{NN} - F_D\| \neq 0$. The aim of neural learning is to minimise this difference until a suitable accuracy level is found, for example $\|F_{NN} - F_D\| < \tau$. The smaller the value of τ, the more accurate the network becomes.

In most real-world applications the function F_D is not known explicitly – the network has to approximate the problem as well as possible based on the data, for both known and unknown examples (see section 2.3 for a discussion on generalisation performance). A single example from a data set is called a *pattern*. Thus a data set D can be regarded as a set of input-target pairs such that $D = \{d_p = (\mathbf{z}_p, \mathbf{t}_p) \,|\, p = 1, \cdots, P\}$ where $\mathbf{z}_p$ is the input vector with dimension I and $\mathbf{t}_p$ is the target vector with dimension K for pattern d_p. Note that for certain neural networks the target $\mathbf{t}_p$ can be null as is the case for the self organising map (SOM) [63], in which case the network learning algorithm needs to establish relationships between patterns using a metric.

Patterns are utilised in types of learning problems such as function approximation and classification. The goal for function approximation problems is to find a function F_{NN} that approximates F_D. For example, for the patterns $(z = 3, t = 9)$ and $(z = 5, t = 25)$ the function $F_{NN} = z^2$ is an appropriate function as it maps the relationship between z and t very well.

Neural networks can also be used to solve classification problems by trying to classify the input vector $\mathbf{z}_p$ as belonging to an associated target class $\mathbf{t}_p$. Consider the problem of determining what type of Iris flower one is dealing with when given only the attributes sepal length, sepal width, petal length and petal width. By defining the input pattern $\mathbf{z}_p = (a, b, c, d)$ where a, b, c and d correspond to the respective attributes mentioned and $t \in \{\text{Iris Setosa, Iris Versicolour, Iris Virginica}\}$[3], the neural network can classify a given pattern $\mathbf{z}_p = (4.9, 3.1, 1.5, 0.1)$ as belonging to class $t =$ Iris-setosa, for example.

It is important that any data set that is used for training be an accurate representation of the problem. This includes aspects such as having enough training data as well as making sure that the training examples are uniformly distributed across the problem space in a manner that accurately reflects the problem. If this is not the case, the neural network will not be able to approximate the problem to an acceptable accuracy. While the network might succeed in learning the training data, generalisation performance will most probably be adversely affected.

2.2.1 Data Considerations

There are several considerations mentioned in [25] regarding data sources for neural networks. It is crucial that these aspects be taken into account to ensure good performance of any neural network. These aspects are

- missing values and how to handle them,
- statistical outliers,
- coding of non-numeric values,
- scaling and transformation,

[3] t is usually coded numerically

- noise injection, and
- pattern sampling and presentation.

These aspects are discussed in more detail in the next subsections.

2.2.1.1 Missing Values

For a multitude of real-world problems it is very common to find that some patterns contain missing values. There are many ways to deal with this situation depending on the network type and data type. These vary from statistical solutions to changing the network architecture itself.

Approaches in [25] include removing the incomplete pattern, replacing missing values with the average value of the parameter (for the previous two cases this might bias training due to information loss), adding additional input units to indicate which patterns are incomplete or simply doing nothing as in the case of SOM (as the SOM algorithm has a way to manage this).

2.2.1.2 Outliers

Statistical outliers are patterns that deviate substantially from the data distribution. Examples of outliers include the value 47 in the set $\{3, 2, 5, 4, 7, 47\}$ and one green dot in a collection of red and blue dots. The problem that outliers present is that they result in large errors when they are evaluated by the neural network. When using gradient-based learning methods, this large error leads to large weight updates which drive the network weights towards the outliers, thus adversely affecting any learning achieved up to that point.

Approaches to deal with outliers include removing them from the training set before training, removing them during training or using an objective function that is resistant to outliers such as Huber's objective function [51]. Approaches to remove outliers during training are discussed in [101] and [40].

2.2.1.3 Non-numeric Values

Input to a neural network from a data set needs to conform to the type of data that neuron net input signal calculations expect. In most neural networks these values need to be numeric. If non-numeric values are present in the data source, they need to be transformed into numerical values.

According to [25], a good way to accomplish this is to present the nominal categories as binary input units. Another approach is to use one input unit and map the nominal values to numerical values. The drawback of this approach is that the range could be interpreted as floating-point values.

It is important to note that data patterns are the input to net input signal and (occasionally) activation function calculations. This entails that the *types* of data can also be anything from real, discrete, complex, or fuzzy-valued (among others).

2.2.1.4 Scaling and Transformation

An important part of data preparation is scaling. Input values should (but do not have to) be scaled to the active domain of the activation function – that is the range at which the activation function is most sensitive to change. For example the active domain of the sigmoid function is $[-\sqrt{3}, \sqrt{3}]$ [25]. This helps to prohibit activation function values that fall in inactive domains of the function, thereby causing ineffective small weight updates. There are a multitude of scaling and transformation methods available that can be used to scale values in problematic data sets where, for example, the minimum and maximum values are not known, or different measuring units are used.

For bounded activation functions, the target values for each pattern need to be scaled to fit into the activation function's active range. A commonly used linear scaling function is:

$$t_s = \frac{t_u - t_{u,min}}{t_{u,max} - t_{u,min}}(t_{s,max} - t_{s,min}) + t_{s,min} \tag{2.17}$$

where t_u is the unscaled value, t_s is the scaled value, $t_{u,min}$ and $t_{u,max}$ are the minimum and maximum values of the unscaled range respectively, and $t_{s,min}$ and $t_{s,max}$ are respectively the minimum and maximum values of the scaled range.

For classification problems the target values are usually scaled to the range $[0.1, 0.9]$

if sigmoid activation functions are used, as the function asymptotically approaches 0 and 1. This is a general guideline for any function with asymptotes.

2.2.1.5 Noise Injection

By injecting noise sampled from a normal distribution that has a zero mean and small variance, it is possible to generate more training data using the same input vectors with new target values. The new target vector, $\mathbf{t}$, should conform to the signal-plus-noise model as mentioned in [25], namely $\mathbf{t}_p = \mu(\mathbf{z}_p) + \zeta_p$ where p is the index of the current pattern, $\mu(\mathbf{z})$ is the unknown function to be approximated and ζ is noise with a zero mean sampled from a uniform distribution. Increased accuracy and reduced training time can also be achieved by injecting noise as this smooths the target function [82]. Noise injection was also used by Engelbrecht [24] to generate new patterns around decision boundaries and so gain better performance.

2.2.1.6 Pattern Sampling and Ordering

Data needs to be presented in different ways to different types of neural networks. In literature there are mainly four classes of data sets, namely

- a *training* set, D_T, containing all patterns used for training of the network,
- a *validation* set, D_V, containing all patterns used to validate neural learning during training,
- a *generalisation* set, D_G, that is used after training to test network performance on completely unknown examples. This is sometimes called a *test set* as well, and
- a *candidate* set, D_C, of all patterns from a data set that are available to the network, but don't form part of D_T, D_G or D_V. This set is mostly empty, except when pattern sampling is done by many neural learning algorithms such as active learning (see section 2.2.3).

There are many methods that govern how a data set could be distributed among D_T, D_G, D_V and D_C depending on how a particular neural network system is set up.

Examples include random sampling, performing fast clustering on the data to group similar patterns together and then distributing these evenly over the desired sets, using only data that lies close to decision boundaries as training data, and K-fold cross-validation where the data set is divided into K parts with $K-1$ parts assigned to D_T and *one* part assigned to D_V. When using K-fold cross-validation, one has to run K experiments, making sure that each of the K sets is used as the generalisation set once. See [77] for a discussion on how to use K-fold cross-validation with neural networks. Another very important approach to sampling is active learning (discussed in section 2.2.3), which allows a neural network to have control over which patterns to include in D_T. The training set is thus sampled dynamically and previously unused patterns are assigned to D_C.

Lastly, the order in which patterns are presented to the network (referred to as *pattern ordering*) is important in many scenarios. When training a neural network using stochastic gradient descent optimisation (see section 2.3.1), patterns from D_T are typically presented in a random order to prevent biasing the network as a result of pattern order. An approach in [83] called selective presentation involves dividing the training set into 'typical' and 'confusing' patterns (i.e. far or near to decision boundaries) and having the network train on these patterns in alternating steps. The increased complexity training approach in [10] lets the neural network learn easy patterns first and increases complexity over time. Other approaches include ordering training patterns based on a distance metric such as Hamming or Euclidean distance, as is done in [11].

It is important to note that, while sampling and ordering might influence each other, they are seen as separate components. For example, a neural network implementation might be set up to use active learning to dynamically choose which training patterns to include in D_T, yet might also use a pattern ordering scheme such as 'easy patterns first' or random sampling to determine in what order the selected patterns are presented to the network.

2.2.2 Fixed Set Learning

Fixed set learning, also known as passive learning, involves using a fixed set of training patterns for neural learning, just as the name suggests. The network has no control

over which patterns from the problem domain are included in training set D_T, which is constructed offline before training. The logic is illustrated in figure 2.6.

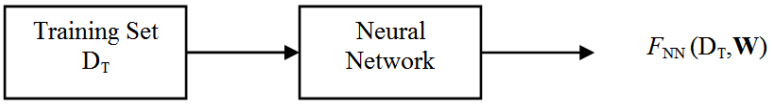


Figure 2.6: Logic behind Fixed Set Learning

Once this training set is constructed, it remains unchanged for the duration of training. This approach is simple to understand and implement, but does have a number of drawbacks:

- In order to accurately approximate a problem, 'enough' training patterns need to be included in the training set that represents the problem domain 'adequately'. Creating concise training sets presents a huge problem, especially if prior knowledge of the problem domain is not available.
- Lange and Männer [71] show that there exists a critical training set size. Including more patterns after this size has been reached does not improve generalisation performance. This maximum efficient size will need to be calculated manually every time a new training set is constructed.
- Redundant training patterns only increase training time, as no real gain is achieved.
- Too large training sets might impact generalisation performance, as well as training time.
- Patterns might not be equally distributed, thus biasing the network.

Although fixed set learning is easy to implement due to the lack of communication with the network, it may result in inefficient pattern selection. Much better results can be obtained if the network has the ability to decide which patterns from the problem domain need to be included in the training set in a dynamic manner.

2.2.3 Active Learning

Cohn, Atlas and Ladner [12] define *active learning* as follows:

> *"Active learning is any form of learning in which the learning algorithm has some control over what part of the input space it receives information from."*

By using an active learning approach, a neural network can dynamically determine which patterns will contribute the greatest to neural learning. The network can exploit its current knowledge of the problem domain to select the most *informative* patterns to be included in the training set D_T – that is, patterns that have a strong influence on network output will be selected. Thus the network has the ability to create and update its training set dynamically, instead of simply receiving the same training data passively from a statically defined training set. This may lead to faster training times and better generalisation performance as shown in [12] and [122]. The logic is shown in figure 2.7.

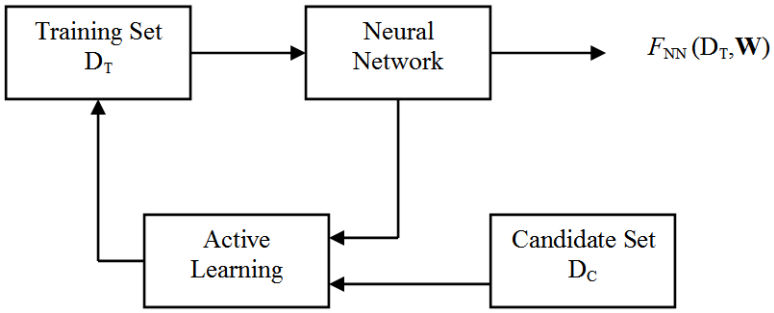


Figure 2.7: Logic behind Active Learning

The two main approaches in active learning include *incremental learning* and *selective learning*. Incremental learning starts off on a small initial training set that is a subset of the candidate set, D_C. At certain specified intervals, called selection intervals, informative patterns are selected from D_C by using one or more evaluation criteria. These patterns are removed from D_C and added to the training set D_T. Thus the training set is the union of all previously selected augmentation subsets, while the candidate set has these subsets removed every time.

Selective learning works similar to incremental learning in the sense that new subsets of informative patterns are selected in each selection interval. Instead of removing pat-

terns from D_C and adding them to the training set D_T, the size of D_C never decreases. The set D_T is reconstructed completely from all of the available patterns in D_C, keeping none of the previous interval's patterns. Training sets are thus totally new subsets after every selection interval.

In both incremental and selective learning, the selection interval may be triggered by events such as:

- A specified number of epochs has been reached.
- A certain percentage decrease in error has occurred.
- There is too little decrease in error per epoch.
- Bad generalisation performance is experienced, determined by using the generalisation factor $\varepsilon_V/\varepsilon_T$ where ε_V is the validation error and ε_T the training error [122], [91].

An important point here is that active learning needs to obtain information about the network from a variety of sources. In [122] and [91], information about the network error of all candidate set patterns is needed. In [23], sensitivity analysis information is used to determine which patterns from the candidate set need to be included in the training set. In short, active learning may need to extract information from any part of the neural network system. This is an important aspect of neural networks when considering the design of an application, as the various locations of information must be available to the active learning component.

2.3 The Learning Component

Essentially, the purpose of a neural network is to find a solution to approximate a problem that is expressed by samples in a data set. In most real-life scenarios, these input to output vector mappings are the only information available to define problems.

The purpose of neural learning is to modify the neural network topology in such a way that the neural network is able to solve a specific problem to a specified level of

accuracy, given examples of input values and, if applicable, corresponding output values. Changes in the topology may involve changing elements such as:

- network *weights*,
- neuron *activation function* and other neuron parameters[4],
- *pruning* the network by removing redundant weights and neurons, and
- *growing* the network as needed by adding more weights and neurons.

Learning algorithms thus involve modifying components in the network topology in an intelligent manner. There are four main learning paradigms that can be used to train neural networks:

Supervised learning is achieved by having the network compare outputs to associated targets to determine the error it made on pattern evaluation. This knowledge is used to adjust weights to minimise overall error.

Unsupervised learning involves having the network determine by itself what features and knowledge can be extracted from data – no external teacher oversees its learning.

Reinforcement learning involves a teacher that tells the network whether its actions are 'good' or 'bad.' This form of learning works on a penalty/reward basis.

Hybrid learning involves combining elements of multiple learning paradigms in one architecture for example radial basis function networks.

Learning in a neural network requires that the network topology is updated in the manner dictated by a learning algorithm. In the case of weight updates, the learning algorithm requires inputs to be presented to the network, after which the appropriate weight updates are performed based on the network's output. This process continues until an acceptable error is reached. However, the *order* in which pattern presentation and subsequent weight updates occur can be different in many optimisation algorithms. The two main approaches are

[4]See [25] for a discussion on adaptive sigmoid functions as an example.

- *Stochastic learning*, where weights are updated after each pattern presentation. Patterns are selected from the training set by a chosen pattern sampling mechanism as stated in section 2.2.1.6. Random presentation is a good choice as this prevents any bias occurring due to pattern ordering. Stochastic learning tends to be slower than batch learning as topology updates occur more often.

- *Batch learning*, where topology change information is accumulated until the end of an epoch, after which the topology is updated only once. Batch learning typically performs faster per epoch due to the fact that topology updates are only applied after all patterns have been presented.

The next sections present an overview of some well-known learning algorithms and optimisation methods that can be used to perform neural learning. The examples listed in each category are by no means exhaustive, as there are probably an endless number of different variations and approaches on how neural networks can be trained. The purpose of these examples is to provide insight into how completely different types of approaches to neural learning exist, even inside the same learning paradigm.

2.3.1 Supervised Learning

The main goal of this learning paradigm is to train a neural network up to a point where it can solve problems to a specified level of accuracy, given examples of input and output. Specifically, given a data set that expresses the function $F_D : Z \rightarrow O$, a neural network F_{NN} and a threshold $\tau > 0$, the network should be trained until $\|F_{NN} - F_D\| < \tau$. The smaller the value of τ, the more accurately the network approximates F_D. The problem is presented to the network as a set of patterns $D = \{d_p = (\mathbf{z}_p, \mathbf{t}_p) \,|\, p = 1, \cdots, P\}$, where d_p denotes a single pattern. Patterns should ideally conform to the signal-plus-noise model as mentioned in section 2.2.1.5.

Neural learning is performed on the patterns in D_T. The input vector, $\mathbf{z}_p$, from pattern $d_p \in D_T$ is presented to the network and the output vector, $\mathbf{o}_p$, is compared to the target vector, $\mathbf{t}_p$, to obtain a pattern error, $\varepsilon_p = \sum_{k=1}^{K} \frac{(\mathbf{t}_p - \mathbf{o}_p)^2}{K}$. Training is performed on the patterns in D_T while the output of D_V is typically used to determine if overfitting occurs or not. One such complete iteration over D_T and D_V is known as an *epoch*.

A well-known method for calculating the total error for each epoch is the sum squared error (SSE), defined as

$$\varepsilon_{D_T} = \sum_{p=1}^{P_T} \varepsilon_p \tag{2.18}$$

where P_T is the total number of patterns in the data set D_T. Similar errors can be defined for D_G and D_V. The SSE can be misleading when compared to the SSE from other simulations that have a different number of training patterns in D_T. This is very important to note in active learning scenarios where the size of D_T changes often and dynamically. An unbiased error estimate is the mean squared error (MSE), defined as

$$\varepsilon_{D_T} = \frac{\sum_{p=1}^{P_T} \varepsilon_p}{P_T} \tag{2.19}$$

The MSE error estimate takes the number of patterns into account and effectively computes the average error per pattern in the set. During or after an epoch, depending on whether stochastic or batch learning is used (see section 2.3), the network weights are updated by an optimisation method. The MSE error estimate is used as an objective function that needs to be minimised by the optimisation method, but other error estimates may also be utilised. Another well-known statistical accuracy measure is the *correlation coefficient*, defined as

$$r = \frac{\sum_{p=1}^{P} o_{k,p} t_{k,p} - \frac{1}{P} \sum_{p=1}^{P} o_{k,p} \sum_{p=1}^{P} t_{k,p}}{\sqrt{\sum_{p=1}^{P} o_{k,p}^2 - \frac{1}{P}(\sum_{p=1}^{P} o_{k,p})^2} \sqrt{\sum_{p=1}^{P} t_{k,p}^2 - \frac{1}{P}(\sum_{p=1}^{P} t_{k,p})^2}} \tag{2.20}$$

The correlation coefficient calculates the correlation between outputs and targets over all patterns to give an indication of the relationship between the approximated function F_{NN} and the true function F_D. Values closer to one indicate accurate approximations.

The choice of error estimate may have an impact on the learning equations of many optimisation methods (for instance gradient descent that is discussed in the next section). A few well-known optimisation methods are listed below. Only a brief discussion is presented for each method.

Gradient Descent Optimisation

Gradient descent (GD) is one of the most well-known and popular optimisation methods in neural computation. GD uses gradient information to move the weight vector along

the negative gradient of the training error estimate ε_{D_T} in weight space. The basis for GD is the error gradient with respect to the weights, defined as

$$\frac{\partial \varepsilon_{D_T}}{\partial \mathbf{w}_n} = \frac{\partial \varepsilon_{D_T}}{\partial net_n} \frac{\partial net_n}{\partial \mathbf{w}_n} \tag{2.21}$$

where net_n is the net input signal and $\mathbf{w}_n$ is the weight vector for neuron n. The values for $\frac{\partial \varepsilon_{D_T}}{\partial net_n}$ and $\frac{\partial net_n}{\partial \mathbf{w}_n}$ depend on the net input signal and the activation function that are used. Also note that, as the activation function f_n depends on the net input signal net_n, the chain rule gives

$$\delta_n = \frac{\partial \varepsilon_{D_T}}{\partial net_n} = \frac{\partial \varepsilon_{D_T}}{\partial f_n} \frac{\partial f_n}{\partial net_n} \tag{2.22}$$

where δ_n is known as the *error signal* for neuron n. The calculation of δ_n is different for every choice of ε_{D_T}, net_n and f_n due to the difference in the differential equations. Weights are updated by using the form

$$\mathbf{w}_n(t) += \Delta \mathbf{w}_n(t) + \alpha \Delta \mathbf{w}_n(t-1) \tag{2.23}$$

where $\Delta \mathbf{w}_n = -\eta \delta_n \frac{\partial net_n}{\partial \mathbf{w}_n}$, α is the momentum factor and η is the learning rate. Momentum is needed when stochastic learning is used and assists in 'smoothing out' weight changes that fluctuate between positive and negative updates (thus undoing the previous step's learning). The learning rate controls the size of each step's update. The error signal δ_n as well as the weight updates for the hidden layer weights are computed similarly. Refer to [25] and [80] for a full discussion on GD and details on any derivations.

LeapFrog Optimisation

LeapFrog was developed by Snyman [103] in 1982 as a method for unconstrained optimisation. Further improvements are discussed in [104]. It is based on the motion of a particle in an multi-dimensional conservative force field. The objective is to conserve the total energy of the particle which consists of its potential and kinetic energies. The potential energy of the particle represents the function to be minimised. Thus for a neural network the training error function (such as the MSE) represents the potential energy.

The LeapFrog method tracks the movement of the particle and adapts the weights of the network at intervals to reduce its potential energy in a suitable manner. A summary of the algorithm is given in algorithm 2.1.

1. Set $\mathbf{w}_0$ randomly, let $\Delta t = 0.5$, $\delta = 1$, $m = 1$, $\delta_1 = 0.001$ and $\epsilon = 10^{-5}$. Also initialise $i = 0$, $j = 2$, $s = 0$, $p = 1$ and $k = -1$.
2. Compute acceleration $\mathbf{a}_0 = -\nabla\varepsilon(\mathbf{w}_0)$ and velocity $\mathbf{v}_0 = \frac{1}{2}\mathbf{a}_0\Delta t$, where $\varepsilon(\mathbf{w}_0)$ is the MSE for $\mathbf{w}_0$.
3. Set $k = k + 1$ and compute $\|\Delta\mathbf{w}_k\| = \|\mathbf{v}_k\|\Delta t$.
4. If $\|\Delta\mathbf{w}_k\| < \delta$ go to 5, else set $\mathbf{v}_k = \delta\mathbf{v}_k/(\Delta t\|\mathbf{v}_k\|)$ and go to 6.
5. Set $p = p + \delta_1$ and $\Delta t = p\Delta t$.
6. If $s < m$ go to 7, else set $\Delta t = \Delta t/2$ and $\mathbf{w}_k = (\mathbf{w}_k + \mathbf{w}_{k-1})/2$, $\mathbf{v}_k = (\mathbf{v}_k + \mathbf{v}_{k-1})/4$, $s = 0$ and go to 7.
7. Set $\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{v}_k\Delta t$.
8. Compute $\mathbf{a}_{k+1} = -\nabla\varepsilon(\mathbf{w}_{k+1})$ and $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{a}_{k+1}\Delta t$.
9. If $\mathbf{a}_{k+1}^T\mathbf{a}_k > 0$ then $s = 0$, else $s = s + 1$, $p = 1$ and go to 10.
10. If $\|\mathbf{a}_{k+1}\| \leq \epsilon$ then stop, else go to 11.
11. If $\|\mathbf{v}_{k+1}\| > \|\mathbf{v}_k\|$ then $i = 0$ and go to 3, else $\mathbf{w}_{k+2} = (\mathbf{w}_{k+1} + \mathbf{w}_k)/2$, $i = i + 1$ and go to 12.
12. Restart: If $i \leq j$, then $\mathbf{v}_{k+1} = (\mathbf{v}_{k+1})/4$ and $k = k + 1$, go to 8, else $\mathbf{v}_{k+1} = 0$. $j = 1$, $k = k + 1$ and go to 8.

Algorithm 2.1: Outline of the LeapFrog algorithm.

Particle Swarm Optimisation

Particle swarm optimisation (PSO) models the social dynamics of natural systems such as birds in a flock or schools of fish and was first devised by Kennedy and Eberhart in 1995 [62]. In broad terms, PSO works by grouping individuals called *particles* together to form a *swarm*. All the particles have the same dimension (in standard PSO) and are *'flown'* through a search space of the same dimension. Each particle's position, $\mathbf{x}_i$, represents a possible solution to the problem at hand. Each particle also has a velocity, $\mathbf{v}_i$, associated with it. Optimisation is achieved by having each particle's position modified based on its own best experience (cognitive component) as well as the experience of its neighbours (social component) based on a performance measure $\mathcal{F}$. A particle's neighbours may include the whole swarm or certain subsets thereof. A particle's velocity at time t is

updated as follows using experience from both its own best position as well as that of its neighbours:

$$\mathbf{v}_i(t) = \mathbf{v}_i(t-1) + \rho_1(\mathbf{x}_{pbest_i} - \mathbf{x}_i(t)) + \rho_2(\mathbf{x}_{gbest} - \mathbf{x}_i(t)) \tag{2.24}$$

where $pbest_i$ is the index associated with the personal best performance of particle i and $gbest$ is the index associated with the global best performance of all particles in the swarm. Using the velocity update in equation 2.24, a particle's position at time t is updated as follows:

$$\mathbf{x}_i(t) = \mathbf{x}_i(t-1) + \mathbf{v}_i(t) \tag{2.25}$$

The result is that particles still search large areas of the search space around the solution while converging on an optimum point. The performance measure, $\mathcal{F}$, that is used to determine the effectiveness of a particle's position is known as the *fitness function*. This fitness function is thus the objective function to be optimised that characterises the problem and is problem dependant. An outline of the standard PSO algorithm is given in algorithm 2.2.

When using PSO to train neural networks, the position vector $\mathbf{x}_i(t)$ of each particle i represents the weight vector $\mathbf{W}_i(t)$ of a neural network i at time t. During the fitness evaluation of each particle position $\mathbf{x}_i(t)$, this weight vector $\mathbf{W}_i(t)$ needs to be obtained from the particle and inserted into a network topology to be able to calculate the fitness of the specific network. The PSO performance measure, $\mathcal{F}$, comprises of the chosen objective function for the neural network. The fitness calculation itself can take many forms and is dependant on the type of network that is to be trained. In the case of a feedforward neural network, the fitness value of a particle may be the MSE of all patterns in D_T, but a fitness value is not restricted to the MSE alone (see section 2.3.1). The fitness function may also be a combination of error metrics over D_T, D_G and D_V for example.

The weights of all neural networks are then updated simultaneously by adjusting the positions of all particles using equation 2.25 (as is the case when using standard PSO – different PSO approaches may use different update strategies). Consult [113] on how to train neural networks cooperatively and [56] for a discussion concerning the

Initialise the position $\mathbf{x}_i$ of each particle randomly.
Initialise the velocity $\mathbf{v}_i$ of each particle to zero.
Set $pbest_i = i$ for each particle.
Set $gbest$ = index of best $pbest_i$.
Set the current iteration $t = 0$.
repeat
    for each particle i
        Evaluate $\mathcal{F}$ for particle i using position $\mathbf{x}_i(t)$.
        If $\mathcal{F}(\mathbf{x}_i(t)) < \mathcal{F}(\mathbf{x}_{pbest_i}(t))$ then $pbest_i = i$, $\mathbf{x}_{pbest_i} = \mathbf{x}_i(t)$.
        If $\mathcal{F}(\mathbf{x}_i(t)) < \mathcal{F}(\mathbf{x}_{gbest}(t))$ then $gbest = i$, $\mathbf{x}_{gbest} = \mathbf{x}_i(t)$.
        Modify particle i's velocity using eq. 2.24.
        Modify particle i's position using eq. 2.25.
    end for
    $t = t + 1$.
until Convergence or stopping criteria are met.

Algorithm 2.2: Outline of the PSO training algorithm.

training of product unit networks using PSO. For an excellent discussion on training neural networks using PSO, as well as performance comparisons against various other optimisation techniques, refer to [112].

Evolutionary Computation

Evolutionary computation is based on the evolution of organisms in nature and has been used in AI as early as by Barricelli [5] in 1954 and by Fraser in 1957 [37]. The process of evolution has the aim of improving the ability of individuals to survive as time passes. The characteristics of organisms are encoded as *genes* in a *chromosome*. After reproduction, the chromosomes of offspring reflect those of both parents. Natural selection is a process by which individuals who are more able to adapt to a changing environment have a better chance of survival. In this way, evolution is an optimisation process that ensures that only the fittest individuals survive.

Set generation $g = 0$.
Initialise the population C_g of individuals.
repeat
    Evaluate the fitness $\mathcal{F}_i$ of each individual i in C_g.
    Select parents using selection operators.
    Generate offspring and perform cross-over.
    Perform mutation on offspring.
    Select the new generation C_{g+1}
    $g = g + 1$.
until stopping conditions are met.

Algorithm 2.3: Outline of the general evolutionary computation algorithm.

Evolutionary computation is a CI paradigm that endeavours to mimic the process of evolution and natural selection in order to solve optimisation problems. A potential solution to a problem is coded as a chromosome and evolutionary processes are used to effectively search through chromosome space. There are many types of evolutionary algorithms such as genetic algorithms [49], genetic programming [37], [5], evolutionary programming [36], evolutionary strategies [90], differential evolution [88], and co-evolution [15]. Each algorithm has its own representation of chromosomes, which may include bit strings, trees or real numbers to name but a few. An instance of a chromosome is called an *individual* and all the individuals in an experiment are called the *population*. Iterations of evolutionary steps result in many different *generations* of a population.

The performance measure $\mathcal{F}$ that is used to indicate how well individuals are doing is called a *fitness function*. The individuals in a population will typically have different fitness values, resulting in some individuals being 'more fit' than others (i.e. the 'more fit' chromosome represents a better solution to the problem). The fitness value is used by *selection operators* to determine which individuals have a higher probability of being selected to produce new offspring. The fitness is also used to decide which individuals will survive to the next generation. Typical selection operators include random selection (thus disregarding fitness), proportional selection based on the fitness value, tournament

selection where the best individual wins, rank based selection that uses the rank of the fitness values rather than the fitness values themselves, and finally elitism that involves selecting the best individuals to survive to the next generation.

Reproduction consists of mainly two processes, namely cross-over and mutation. *Cross-over* is the process of combining the chromosome information of two parent individuals to form an offspring individual containing elements of both parents. *Mutation* randomly variates gene values of a chromosome to form a new individual. As an example, a chromosome that consists of n bits is mutated by 'flipping certain bits' to form a new individual. In both cross-over and mutation, the fitness of the offspring will be either better, equal or worse than that of the previous generation's individuals. It is up to the developer of the evolutionary algorithm to combine the correct reproduction operators with the right selection operators in order to solve a problem.

The general outline of an evolutionary algorithm as stated in [25] is shown in algorithm 2.3. Note that specific types of algorithms might add, remove, or change the order of the steps in this outline depending on its specific design. The algorithm terminates when one or more stopping conditions are met, namely the maximum number of generations have been reached, the average fitness does not change significantly as generations progress or an acceptable individual has evolved.

EC follows a similar approach to train neural networks as PSO does. Each chromosome i in the population represents the neural network weight vector $\mathbf{W}_i(t)$ at time t. The performance measure $\mathcal{F}$ is defined similar to that of PSO, in that the neural network objective function over one or more datasets is used to indicate the fitness of a chromosome. Various selection, cross-over, and mutation operators may be used to create or change individuals in the population (as dictated by a specific EC algorithm). Evolutionary algorithms have been used with great success to train neural networks. Yao [119] shows how to use evolutionary computation to train neural networks. A good example of using EC to train a neural network is the Neuro Evolution of Augmenting Topologies (NEAT) algorithm [107]. Another good reference is [4], which contains a preliminary taxonomy and guide to literature about how to use EC approaches to train neural networks.

2.3.2 Unsupervised Learning

Unsupervised learning has a different objective than supervised learning in that the aim of unsupervised learning is to identify features and patterns in the data set *without* the aid of a teacher. In most cases, unsupervised learning algorithms typically produce results equivalent to a *clustering* of input space. Network weights are adapted in such a way that *'similar'* patterns are grouped together in output space (either the same neuron, or a group/neighbourhood of neurons). Similarity depends on the type of comparison metric that is used in the net input signal and/or activation functions of neurons such as distance based net input signals discussed in section 2.1.1 for example. The type of metric can also differ between network types.

Unsupervised learning is useful to solve problems where data sets of input-output pairs are not readily available or impossible to obtain. Kröse and Van der Smagt [66] give examples of the types of problems that unsupervised learning can solve:

- *Clustering* of input patterns into clusters where all the patterns in a cluster have common features that are not present in other clusters.
- *Vector quantisation* is used to discretise continuous spaces by having vectors from an n-dimensional space as input and a discrete representation as output.
- Mapping input vectors to a *lower dimension* than the original subspace while preserving as much variance of the input data in the output data as possible.
- *Feature extraction* to find patterns and characteristics present in the data set.

Most unsupervised networks have a two layer architecture that maps input patterns of dimensionality I to associated output neurons of dimensionality O. The following sections list some well-known unsupervised learning algorithms.

Hebbian Learning Rule

The Hebbian rule was introduced by the neuropsycologist Hebb in 1949 [47] as a very simple learning scheme. Hebb's hypothesis states that the ability of a neuron to fire is directly related to the ability of other neurons connected to it to fire (thus displaying

associative behaviour). If such a correlation is found to be present between two neurons, the weight that connects the neurons is strengthened. The weight change is calculated as

$$\Delta w_{ki} = \eta o_k z_i \tag{2.26}$$

where η is a constant learning rate, o_k is output unit k and z_i is input unit i. The weights are then updated by

$$w_{ki} \leftarrow w_{ki} + \Delta w_{ki} \tag{2.27}$$

Hebbian learning has the effect that input and output neurons that have strong correlations produce a large Δw_{ki} and thus a greater emphasis is placed on the connection between them.

Hebbian learning has a fundamental problem, namely that weight values are unbounded and tend towards infinity as more learning iterations are presented. There are many different ways to address this, such as introducing a *'forgetting factor'* to reduce weight values over time. Oja [84] developed such a rule called the principle component learning rule that is based on Hebbian learning. Principle component analysis (PCA) is a statistical technique that is used to reduce an H-dimensional subspace to an L-dimensional one where $L < H$ by extracting the principle components of the H-dimensional subspace. The weight changing method is identical to Hebbian learning, but with an added *forgetting factor*:

$$\Delta w_{ki} = \eta o_k z_i + \eta o_k^2 w_{ki}(t-1) \tag{2.28}$$

Learning Vector Quantiser

The learning vector quantiser (LVQ) was proposed by Kohonen [63] and is a very well-known unsupervised learning algorithm used for clustering patterns. In general terms, clustering can be defined as grouping all the data patterns in the set D_T (where $|D_T|$ is the total number of patterns in D_T) into C clusters where $|D_T| >> C$ and all patterns in cluster C_i are 'more alike' than any patterns in all other clusters C_j where $j \neq i$. Thus the aim of clustering is to produce groups (represented by output units) of similar input vectors as measured by a specific metric such as Euclidean distance.

Initialise all weights using a chosen initialisation scheme.
Initialise training parameters η and $\mathcal{N}(0)$ randomly.
repeat
  for each input pattern $\mathbf{z}_p$
    Calculate d_k for each output neuron o_k using eq. 2.4.
    Find the output neuron o_k with the smallest distance d_k.
    Update weights using equation 2.29.
  end for
  Update the learning rate, η.
  Update the neighbourhood radius, $\mathcal{N}(0)$.
  t = t + 1.
until Stopping criteria are met.

Algorithm 2.4: Outline of the learning vector quantiser training Algorithm

The LVQ architecture, illustrated in figure 2.8, facilitates clustering by utilising a two layer topology. Each output neuron o_k represents a cluster while each input neuron z_i corresponds to an attribute in the data pattern. The distance $d_{k,p}$ between the input vector $\mathbf{z}_p$ and the input weight vector $\mathbf{u}_k$ for each output neuron is used to determine the winning cluster. Any distance metric may be used, but Euclidean distance is mostly used to determine which output neuron is the winner for the current pattern, as defined in equation 2.4.

Note the usage of the 'competition' step as laid out in section 2.1.1. Once a winning neuron is determined, the weight values are adjusted as follows:

$$\Delta u_{ki} = \begin{cases} \eta(t)[z_i - u_{ki}(t-1)] & \text{if } k \in \mathcal{N}(t) \\ 0 & \text{otherwise} \end{cases} \tag{2.29}$$

where $\eta(t)$ is a learning rate and $\mathcal{N}(t)$ is the set of *neighbours* of the winning neuron – that is the weight vectors of neurons around (and including) the winning neuron. Algorithm 2.4 outlines the basic training approach for LVQ. Stopping criteria include:

- the maximum number of epochs is reached,

- the quantisation error, defined as $\varepsilon_T = \frac{\sum_{p=1}^{|D_T|} \|\mathbf{z}_p - \mathbf{u}_k\|_2^2}{|D_T|}$, is small enough,
- there is no more gain in training as weight updates are too small.

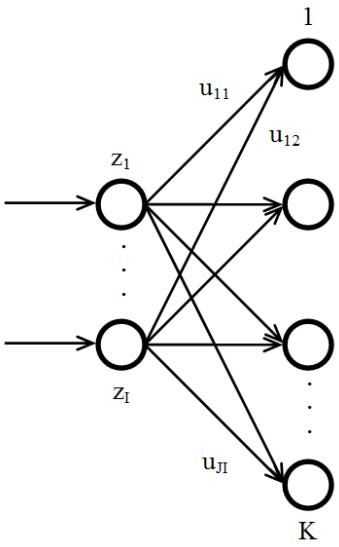


Figure 2.8: The LVQ architecture

Self-Organising Maps

The self-organising map (SOM) was developed by Kohonen [63] [64]. The SOM was largely motivated by the human cerebral cortex's abilities to cluster sensory inputs by using ordered maps.

A SOM is a multi-dimensional clustering method that is used to map I dimensional input vectors $\mathbf{z}_p$ to a discrete output space. The SOM, illustrated in figure 2.9, has a two-layer topology that creates clusters of input patterns. The output units of a SOM are arranged in an ordered grid, which is usually two-dimensional such as rectangular or hexagonaland and can follow any connection pattern. This grid is used to learn the probability density function of input space, while the original space's topological structure is maintained. The relationships between vectors in input space will be preserved in the grid.

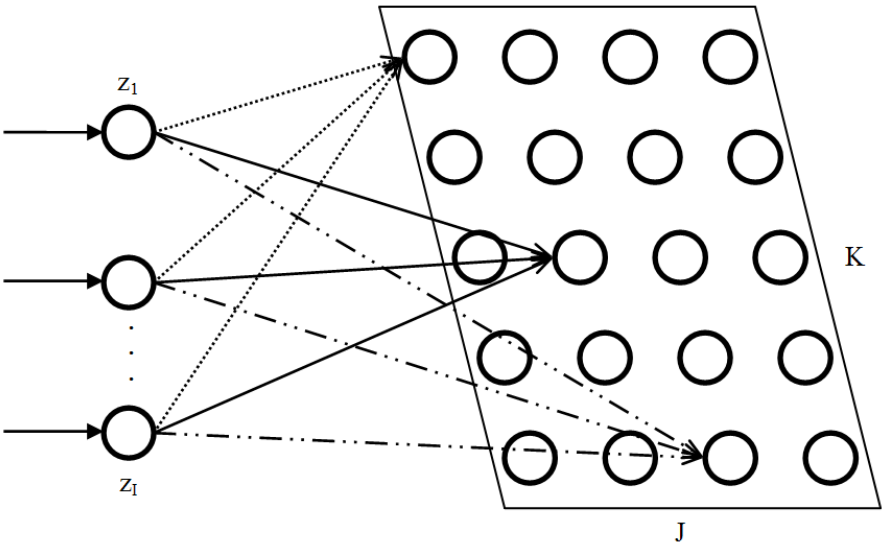


Figure 2.9: The SOM Architecture

There should be fewer neurons in the grid than there are training patterns, as the goal of the SOM is to map input space to a lower dimensional output space. All neurons in the grid, that is, each neuron kj in row k and column $j; \forall j = 1, \cdots, J, k = 1, \cdots, K$, are connected to the each input unit via an I dimensional weight vector $\mathbf{w}_{kj}$. These weights can be initialised in many ways, including random initialisation, using known codebook vectors, using the statistical principal components analysis of the input space, among others.

Algorithms 2.5 and 2.6 respectively show two of the different approaches to train a SOM, namely *stochastic* and *batch* learning. It is important to note that the two learning approaches assume the same type of topology (neuron structure and weight connections) and uses the same training data, but that the algorithm details are profoundly different. See section 2.3 for more details on the difference between stochastic and batch learning.

When using stochastic learning, the weights of each neuron are updated after each pattern presentation as follows:

$$\mathbf{w}_{kj}(t+1) = \mathbf{w}_{kj}(t) + h_{mn,kj}(t)[\mathbf{z}_p - \mathbf{w}_{kj}(t)] \tag{2.30}$$

where mn is the grid row and column of the winning neuron for input pattern $\mathbf{z}_p$. The winning neuron, called the *best matching unit* (BMU), is the neuron that is 'closest' to

Create a map of $K \times J$ neurons.
Initialise the weight vector of each neuron.
Set the current training epoch $t = 0$.
repeat
    for each input vector $\mathbf{z}_p$ in D_T
        Find the BMU using eq 2.31
        for each neuron kj in the grid
            Update $\mathbf{w}_{kj}$ using equation (2.30).
        end for
    endfor
    $t = t + 1$
until Stopping criteria are met.

Algorithm 2.5: Outline of the stochastic SOM training algorithm.

Create a map of $K \times J$ neurons.
Initialise the neuron weight vectors to the first KJ patterns in D_T.
Set the current training iteration $t = 0$.
repeat
    for each neuron kj in the grid
        Find all patterns $\mathbf{z}_p$ in D_T for which the neuron is the BMU.
        Add z_p to the current neuron's BMU winner list.
    end for
    for each neuron kj in the grid
        Update $\mathbf{w}_{kj}$ as the mean over the BMU list of the neuron.
    end for
    $t = t + 1$
until Stopping criteria are met.

Algorithm 2.6: Outline of the batch SOM training algorithm.

the input vector and is selected using a distance metric such as the Euclidean distance. That is,

$$\| \mathbf{w}_{mn} - \mathbf{z}_p \|_2 = \min_{\forall kj} \{ \|\mathbf{w}_{kj} - \mathbf{z}_p\|_2^2 \} \tag{2.31}$$

The function $h_{mn,kj}(t)$ in equation 2.30 is known as the *neighbourhood function*, which is used to select the neurons around the winning neuron that will also have their weights updated, as well as the magnitude of these updates. A condition for convergence of the SOM is that $h_{mn,kj}(t) \to 0$ as $t \to \infty$. The neighbourhood function is typically based on the distance between the coordinates of neurons as they are located on the map. A Gaussian neighbourhood function that is often used to implement $h_{mn,kj}(t)$ is given by

$$h_{mn,kj}(t) = \eta(t) e^{-\frac{\|c_{mn} - c_{kj}\|_2^2}{2\sigma^2(t)}} \tag{2.32}$$

where $\eta(t)$ is the learning rate, c_{mn} and c_{kj} are coordinates of neurons on the map, and $\sigma^2(t)$ is the width of the Gaussian kernel that defines the neighbourhood. Both $\eta(t)$ and $\sigma^2(t)$ need to be monotonically decreasing to ensure that $h_{mn,kj}(t) \to 0$ as $t \to \infty$.

Training continues until an accurate map is constructed. The quantisation error metric that is typically used to determine map accuracy is defined as

$$\varepsilon_T = \sum_{p=1}^{|D_T|} \|\mathbf{z}_p - \mathbf{w}_{mn}(t)\|_2^2 \tag{2.33}$$

Batch versions of the SOM training rule have been developed as stochastic training is very slow. Weight updates are only performed after all patterns have been presented. For each neuron in the grid, a list is kept of all the patterns in D_T for which it is the BMU. After an epoch, the weights of each neuron is assigned the mean over the patterns in its BMU list. An outline of the batch training algorithm as developed by Kohonen [64] is given in algorithm 2.6.

The effect of SOM training is to cluster similar input patterns together. After SOM training, whether using the batch or stochastic algorithm, the only network 'output' is the grid of neurons and their respective weight vectors $\mathbf{w}_{kj}$. An additional clustering step needs to be performed to locate cluster boundaries. A multitude of ways exist to accomplish this, of which the unified distance matrix (U-matrix) approach and Ward clustering are popular. The U-matrix expresses the distance between neighbouring weight vectors

for each neuron in the grid, with large values indicating cluster boundaries. The U-matrix can then be visualised using schemes like grey-scale colour coding.

Ward clustering starts by having each neuron form the centroid of its own cluster. Over time, the two clusters which are closest to each other are merged until either the optimal number, or the specified number of clusters are found. The Ward distance measure that is used to find the closest two clusters is given by

$$d_{rs} = \frac{n_r n_s}{n_r + n_s} \parallel \mathbf{w}_r - \mathbf{w}_s \parallel_2^2 \tag{2.34}$$

where r and s are cluster numbers, n_r and n_s are the number of neurons in the respective clusters, and $\mathbf{w}_r$ and $\mathbf{w}_s$ are the centroids of the respective clusters (i.e. the average of all weight vectors in the cluster).

A key design problem that researchers face when using SOM is the choice of the size of the grid. If the grid is too large, overfitting may occur. This may lead to a large number of small clusters with few patterns in each - in the extreme case only one pattern is assigned to each neuron. If the grid is too small, the variance between the weight vectors $\mathbf{w}_{kj}$ of any particular cluster may be found to be very high, thus not yielding any valuable differentiation. A good approach to address this problem is to introduce *architecture selection* as discussed in section 2.3.5 that grows the SOM. Training starts on a small grid and more neurons are added over time. See [25] for a discussion and algorithm for growing a SOM.

2.3.3 Reinforcement Learning

Reinforcement learning [6] can be described as the mapping from states in an environment to actions with the main objective being the maximisation of a reinforcement signal, or 'reward.' It is based on the approach taken to train animals, for example teaching dogs to be obedient or dolphins to perform tricks. In typical cases the learner has no prior knowledge about the problem space and has to discover which actions yield high rewards and which yield low rewards, or even penalties.

Neural networks that utilise reinforcement learning typically have two-layer architectures. The training set consists of input patterns and associated actions. The typical operation cycle starts by letting the network evaluate the environment via sensory inputs.

The network then decides what action to take, which is then evaluated by an external teacher. Positive or negative reward signals, r_p, are then generated by the teacher which is then used to update the weights using

$$\Delta w_{kj} = \eta(r_p - \theta_k)e_{kj} \tag{2.35}$$

where η is the learning rate, θ_k is the reinforcement threshold and e_{kj} is the eligibility of weight w_{kj} such that

$$e_{kj} = \frac{\partial}{\partial w_{kj}} \ln(g_j) \tag{2.36}$$

where $g_j = \text{Prob}(o_{k,p} = t_{k,p} \mid \mathbf{w}_k, \mathbf{z}_p)$. This approach is but one of many possible ways to train neural networks using reinforcement learning.

2.3.4 Hybrid Learning

As the name suggests, hybrid learning involves the combination of different learning paradigms in a single neural network model. The most common combination is to use elements of supervised and unsupervised algorithms together to permit the learning algorithm to find a more compact representation of the problem space. This allows for better generalisation performance and faster training times.

A typical example of a hybrid approach is the radial basis function network (RBFN) [78], which is a combination of the learning vector quantiser (LVQ) and gradient descent learning[5]. The first two layers of the RBFN form the LVQ consisting of I inputs and J outputs (which serve as the hidden units for the RBFN), as illustrated in figure 2.10. The hidden unit transfer functions are based on the LVQ output where the input vector's closeness to the hidden units are computed. The activation function of these hidden units can be given by several types of kernel functions, of which two examples include a Gaussian kernel,

$$y_j = e^{-\frac{||\mathbf{z}_p - \vec{\mu}_j||^2}{2\sigma_j^2}} \tag{2.37}$$

[5]Using LVQ and gradient decent learning for the RBFN is but one possibility and there are many variations of possible combinations of learning approaches.

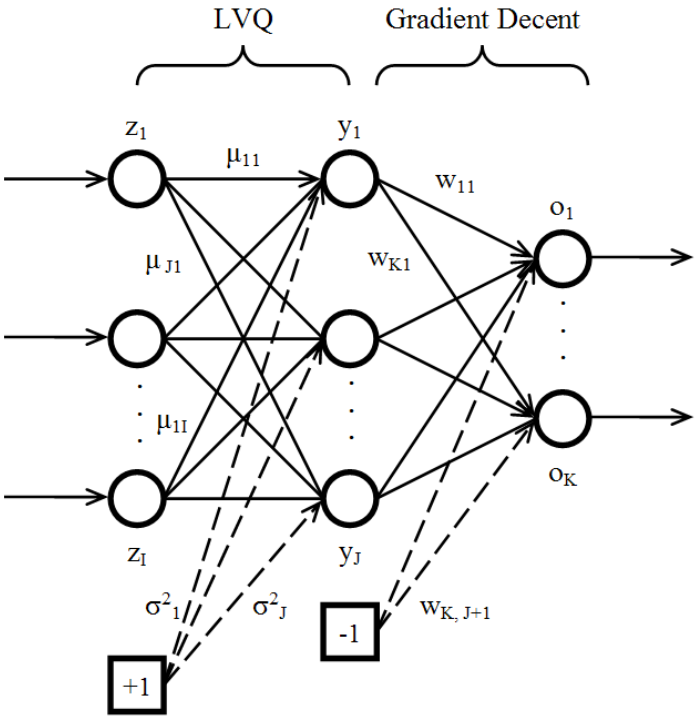


Figure 2.10: An example radial basis function network architecture

or a logistic kernel,

$$y_j = \left[1 + e^{-\frac{||\mathbf{z}_p - \vec{\mu}_j||^2}{2\sigma_j^2} - \theta_j}\right]^{-1} \tag{2.38}$$

The norm $||\bullet||$ in both equations is the Euclidean distance, σ_j is the standard deviation of the basis function, and $\vec{\mu}_j$ is the mean of the basis function.

The final layer of the RBFN is computed using a standard feedforward network with linear output units,

$$o_k = \sum_{j=1}^{J+1} w_{kj} y_j \tag{2.39}$$

Training of the specific RBFN discussed above is achieved in two steps: first the unsupervised weights between the first two layers are found and then the supervised training of the weights between the hidden and output layers follows. An outline of the basic algorithm is given in Algorithm 2.7 and is but one example of how two types of learning can be combined. The specific algorithm to training any particular RBFN in

Initialise all μ_{ji} to the average values of inputs in D_T.
Initialise all σ_j^2 to the variance over training set.
Initialise all w_{kj} to small random values.
repeat
    Perform LVQ epoch to learn the centroids $\overrightarrow{\mu}_j$.
    Set σ_j^2 for all winning y_j as the average of the Euclidean distances of $\overrightarrow{\mu}_j$ to the input patterns for which y_j was the winner.
until LVQ converges.
repeat
    Learn w_{kj} weights using $\Delta w_{kj} = \eta(t_k - o_k)y_j$
until gradient descent phase converges.

Algorithm 2.7: Outline of a RBFN training algorithm for figure 2.10.

general is directly related to the types of learning algorithms that are combined.

2.3.5 Network Architecture Selection

It has been shown that when given n network architectures that are trained on the same data set, the simplest network will give better generalisation performance on average [100], [110]. Network architecture selection is used to optimise neural network topologies and is commonly referrend to as the *growth* or *pruning* of a network. Architecture selection algorithms add or remove weight connections or neurons from the topology in order to select the architecture that gives the 'best' performance. Best performing architecture can be interpreted as the network that shows one or more of the following characteristics:

- The smallest number of neurons and weights for a desired accuracy level.
- The fastest convergence speed during training.
- The fastest evaluation time per pattern.
- The least overfitting of data during training.

- The least overall computational complexity.
- Noise in the data is not memorised.

Broadly speaking, the following architecture selection approaches can be followed:

- **Network growth:** A small network architecture is used and more neurons or weights are added during training, based on a construction algorithm. Examples of such algorithms are discussed in [38], [48] and [69]. The aim of growing a network is to start small and add more network elements only when needed.
- **Network pruning:** Training starts on an oversized network. Either during training or after convergence, the network is assessed and superfluous weights and/or neurons are removed. The relevance of each neuron or weight is determined and are removed if deemed irrelevant. A simple example of this is a weight connection that has a value close to zero - by removing it there is no real change to the network's behaviour, but the model is simpler and less computationally intensive. Irrelevant input parameters (i.e. those that don't affect the output of the network) can also be removed in this way in many network types.
- **Regularisation:** When regularisation is used to help facilitate architecture selection, the size and complexity of the network is added to the objective function to be minimised, for example

$$\varepsilon = \varepsilon_T + \Upsilon \varepsilon_C \tag{2.40}$$

where ε_T is a standard objective function such as MSE, ε_C is a penalty term which increases as network complexity increases and Υ controls the influence that the penalty term has. By minimising this new objective function, smaller network architectures are preferred. For an in-depth discussion on regularisation, see [42], [118].

There are many different approaches to perform architecture selection, some of which involve other fields of CI. These include intuitive techniques such as proposed in [44] that considers weight strength and activation frequency, evolutionary algorithms as found in [117], formal statistical hypothesis testing techniques as developed in [108] and sensitivity analysis techniques as done in [26] and [28].

Different architecture selection algorithms apply depending on the type of neural network that is used. A feedforward neural network is grown or pruned in a completely different manner than a SOM for instance. A full discussion on the different types of architecture selection algorithms is out of scope of this dissertation. The key point is that architecture selection algorithms can employ a variety of ways to grow or prune network topologies and that these algorithms all need access to any information that resides in the neural network system.

Architecture selection has serious implications for the design of neural network topology implementations that need to support architecture selection. The framework's topology component must be flexible enough to allow neuron and weight connections to be added or removed dynamically. Furthermore, any architecture selection algorithm needs to obtain information about the network from a variety of sources, for example the training error at time t, current weight values, neuron function parameter values, learning algorithm variables and any other metrics dictated by a particular architecture selection algorithm. The framework must allow any of this information to be accessible by architecture selection algorithms.

2.4 Example Neural Network Models

This section illustrates some of the typical neural network topologies graphically, and a brief overview of each one is provided. These include

1. a *layered* architecture (more than 1 layer),
2. an *unstructured* architecture (only 1 layer),
3. a *recurrent* architecture (more than 1 layer and at least one loop), and
4. a *modular* and an ensemble architecture.

These are definitely not the only types of topologies and merely serve to illustrate the diversity of neural network models. This in turn indicates how flexible a neural network framework needs to be.

2.4.1 Layered Networks

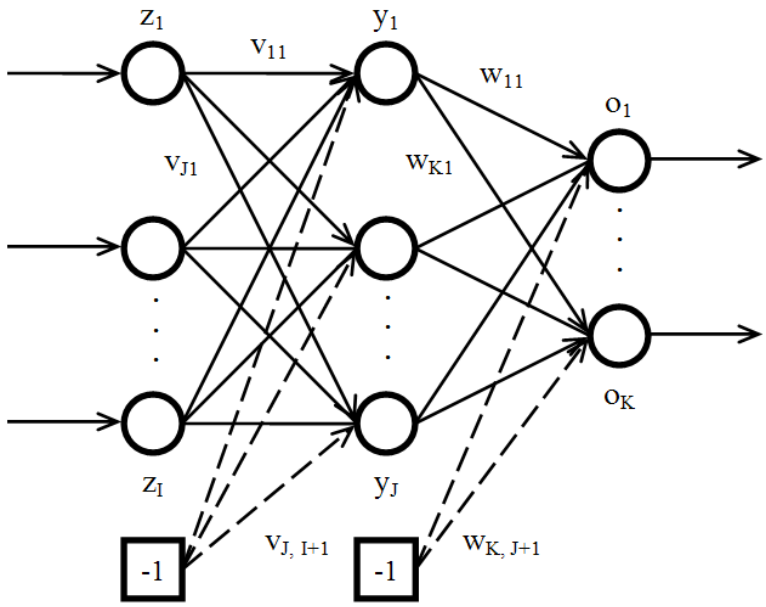


Figure 2.11: The feedforward network as a layered architecture example

The *layered network architecture* is used extensively in both supervised and unsupervised learning approaches. The most notable characteristics in a layered architecture are that it consists of *two* or more layers (including the input layer) and that it contains no loops. A single layered architecture is regarded as an *unstructured architecture* as described in section 2.4.2 Arguably, the most well-known example of layered network architecture is the *feedforward network* as illustrated in figure 2.11.

The output of a feedforward neural network is calculated as

$$\begin{aligned} o_{k,p} &= f_{o_k}(net_{o_k}) \\ &= f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(net_{y_j})) \\ &= f_{o_k}(\sum_{j=1}^{J+1} w_{kj} f_{y_j}(\sum_{i=1}^{I+1} v_{ji} z_i)) \end{aligned}$$

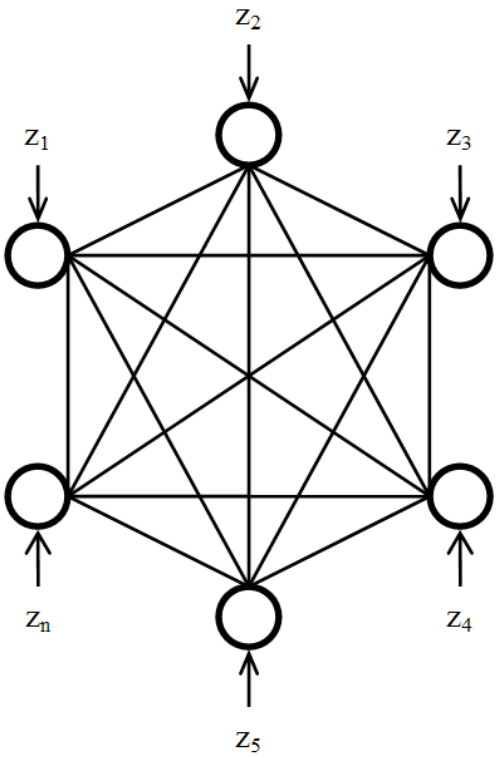


Figure 2.12: The Hopfield network as an unstructured architecture example

2.4.2 Unstructured Networks

An *unstructured network architecture* is a *single* layer of neurons. Thus the only possible weight connections between these neurons are intra-layer and self connections. Possibly the best example of an unstructured network is the Hopfield network [50] (shown in figure 2.12). This neural network is a form of recurrent neural network and serves as an associative memory.

The Hopfield network consists of only one layer of J neurons, where each neuron is connected to all other neurons except itself via symmetrical weight connections, i.e.

$$\begin{aligned} w_{ij} &= w_{ji}, \forall j, i \\ w_{ii} &= 0, \forall i \end{aligned}$$

Each neuron i utilises an inner product net input signal as defined in equation 2.2 and a step activation function as in equation 2.8 to produce an output value o_i that is

typically either 0 or 1, as defined in Hopfield's original paper [50]:

$$o_i = \begin{cases} 1 & \text{if } net > \theta_i \\ 0 & \text{otherwise} \end{cases} \tag{2.41}$$

where o_j is the output value of neuron j, $net = \sum_{j=1}^{J} w_{ij} o_j$ and θ_i is the threshold value of the activation function of the current neuron i. Sources such as [75] state that activation values can be -1 or 1 as well.

The state of the network is given by the vector $(o_1, o_2, \cdots, o_N)$ where N is the total number of neurons in the network. Evaluation of an input pattern consists of setting the network state to be the same as that of the pattern. Neurons are then evaluated in either a defined or random order and each neuron's output (or state) will either change or stay the same. Neurons are evaluated until the network reaches a stable state, that is, neuron outputs do not change anymore. Over time the network will converge to a new state which is the 'remembered' state for the input pattern. It is proven in [67] that the network will reach a stable state in at most $n2^n$ steps.

Each state of the network has an 'energy' function E associated with it, defined as

$$E = -\frac{1}{2} \sum_{i \neq j} w_{ij} o_i o_j \tag{2.42}$$

A Hopfield network is trained by finding the correct set of weights that will allow a set of inputs to converge on the correct remembered states respectively. E has been shown to be a monotonically decreasing function that has one or more minima [50]. The network is trained by lowering the energy of states that need to be remembered to coincide with these minima. The definition of E ensures that random pattern presentations will have the network converge on a state that is a local minimum of E.

2.4.3 Recurrent Networks

A layered network becomes a *recurrent network architecture* as soon as there are neurons with self connections or 'backward facing' inter- or supra connections (or loops) that cross layer boundaries.[6] Thus certain neurons can make use of information that is obtained

[6]Note that the Hopfield network is an *unstructured* network that also has recurrent connections, but it does not have multiple layers.

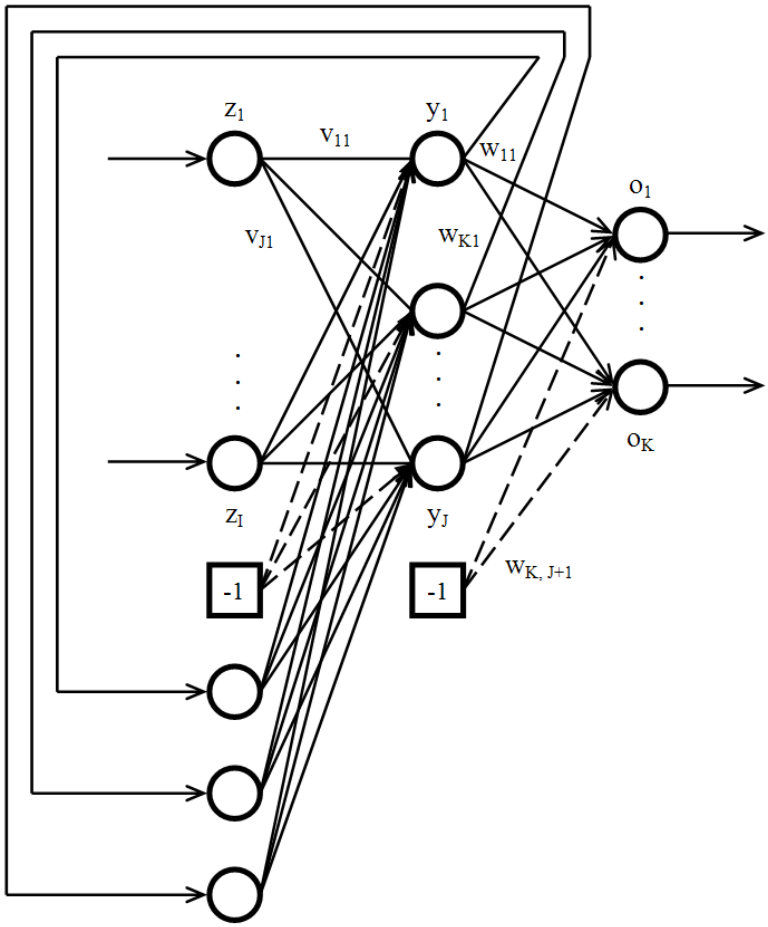


Figure 2.13: The Elman network as a recurrant network architecture example

from other neurons in later layers (or from itself, as is the case with self connections). This allows a neural network to learn temporal characteristics of the problem.

The Jordan network [59] and the Elman network [22] are good examples of simple recurrent networks. The Elman network (illustrated in figure 2.13) makes a copy of the hidden layer neurons in the input layer. Thus the hidden layer state of the previous pattern is also input to the network for the next pattern. The Jordan network works on the same principle, but makes a copy of the output layer neurons.

2.4.4 Ensemble And Modular Networks

This section discusses ensemble and modular network topologies as examples of advanced neural network topologies.

Ensemble Networks

An ensemble of neural networks [86] consists of a number of neural networks that are combined to solve the same problem, as shown in figure 2.14. The simplest example of an ensemble is a collection of neural networks that all share the same training data, the same topology and the same learning algorithm. More complex examples may vary the training sets between the networks in a variety of ways, use different learning strategies for each network, use different initial conditions per network or even use completely different types of networks to form part of the ensemble.

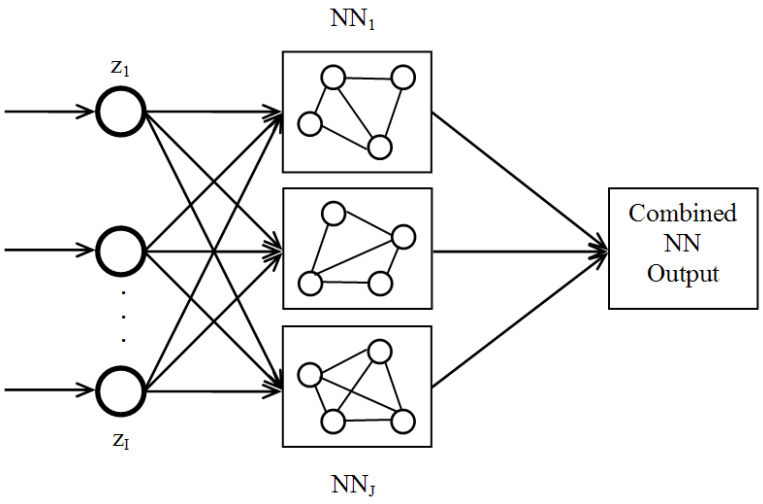


Figure 2.14: An Ensemble network architecture example

All the networks in the ensemble evaluate a pattern when it is presented. The output of these various networks has to be combined to give a single output value. A variety of ways of doing this exist, such as:

- Select the network in the ensemble that presents the lowest average generalisation error. In effect this is not really an ensemble, as n networks are trained and only

the best performing one is selected. This approach is simple, but disregards the inputs from other networks that might have performed better on certain input patterns.

- Calculate the average of the results of all networks to give a single answer, hopefully improving performance as a result. This approach is a true ensemble that uses n networks to evaluate a pattern.

$$F_{BEM} = \frac{1}{n} \sum_{i=1}^{n} F_{NN_i}(\mathbf{z}_p) \tag{2.43}$$

where F_{NN_i} is the output of network i. This is called the basic ensemble method (BEM) in literature [8].

- Use weighted values for the output of each network to minimise the MSE of the ensemble. This approach ensures that networks with high generalisation errors have a smaller effect on the value of the output of the ensemble than networks with lower generalisation errors, and so improving accuracy. This weighted average of all network outputs is computed as

$$F_{GEM} = \sum_{i=1}^{n} \alpha_i F_{NN_i}(\mathbf{z}_p) \tag{2.44}$$

where $\sum \alpha_i = 1$. This is called the generalised ensemble method (GEM) in literature [86].

- The dynamic ensemble method (DEM) [58] is based on the GEM method, but the weighting for each network is determined dynamically on a per-pattern basis instead of choosing static values (the α_i values in GEM). This dynamic weighting $\omega_i(t)$ is based on the certainty of the network's output – the 'closer' $y = F_{NN_i}(\mathbf{z}_p)$ is to the activation function boundaries, the higher the confidence $c(y)$ of the network is, where

$$c(y) = \begin{cases} y & \text{if } y \geq 0.5 \\ 1 - y & \text{otherwise} \end{cases} \tag{2.45}$$

Equation 2.45 assumes an activation function in the range (0,1) and can be readily adapted to other ranges. The confidence factor $c(F_{NN_i}(\mathbf{z}_p))$ can now be used to

rank each network against each other to produce a weighted average F_{DEM} over all networks, where F_{DEM} is defined as

$$F_{DEM} = \sum_{i=1}^{n} \omega_i(t) F_{NN_i}(\mathbf{z}_p) \tag{2.46}$$

where the $\omega_i(t)$ are defined as

$$\omega_i(t) = \frac{c(y)}{\sum_{j=1}^{n} c(y)} \tag{2.47}$$

and $\sum \omega_i(t) = 1$.

Modular Networks

The concept of modularity involves the division of a complex system into smaller, simpler units. These units accomplish the same original goal of the larger system by working together. Units may be similar or different to each other – the key feature of modularity is that there is *separation of concerns* by splitting functionality out into multiple modules.

Most neural network topologies have a monolithic structure – they typically have a set number of neurons, weights and layers and use the same training strategy across the entire topology. While these topologies are relatively simple to understand and implement, their performance degrades very quickly when large input dimensions are encountered [65]. Modular neural networks aim to make problems more manageable by using decomposition and replication to help solve them. Azam [3] defines a *modular neural network* (MNN) as follows:

> "A neural network is said to be modular if the computation performed by the network can be decomposed into two or more modules (subsystems) that operate on distinct inputs without communicating with each other. The outputs of the modules are mediated by an integrating unit that is not permitted to feed information back to the modules. In particular, the integrating unit decides both (1) how the modules are combined to form the final output of the system, and (2) which modules should learn which training patterns."

Zhao [125] discusses a *general model for MNNs* which is illustrated in figure 2.15. Input to the network is given to an *allocator* network, which distributes the pattern to

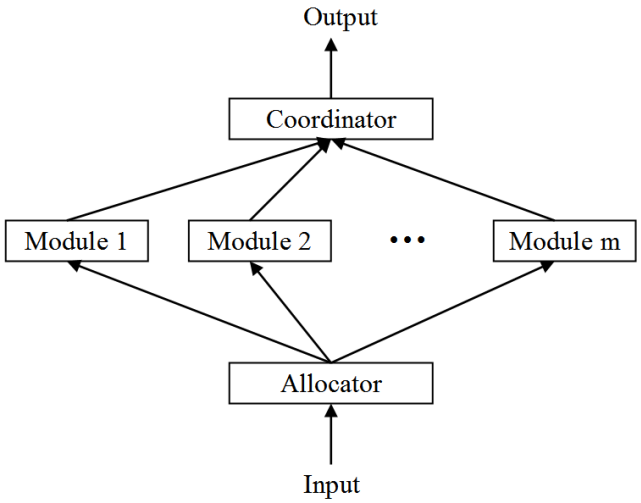


Figure 2.15: A general model for modular neural networks.

one or more modules. A *coordinator* is then used to combine the results from all the relevant modules (which can be either all or just a subset of all the modules) and give the final output of the network. The allocator and coordinator may themselves be neural networks. Based on the level of intelligence of the allocator and coordinator, there are various possibilities:

- An allocator can be so strong that it always selects only one module to perform the evaluation. This negates the need for a strong coordinator. In certain cases the coordinator may even be removed.
- If a weak allocator is used, then more than one or all of the modules will be selected to perform the task. A strong coordinator will be needed to combine the results from all the modules into a meaningful answer. Note that if the allocator is so weak that it may be removed, the network is effectively an *ensemble* network.
- In the extreme case where very strong modules are used, both the allocator and coordinator may be removed. This setup is known as a *one-class-one* network, as discussed in [68].

An important classification that can be made is whether the modules are trained *before* becoming part of the MNN, or *while* they are part of the MNN. Ensemble member

networks are typically trained beforehand, resulting in the fully trained network being incorporated into an ensemble architecture. MNN architectures may follow a similar approach, but may also allow the individual modules to train on different aspects of the problem. The learning techniques used may even differ between modules.

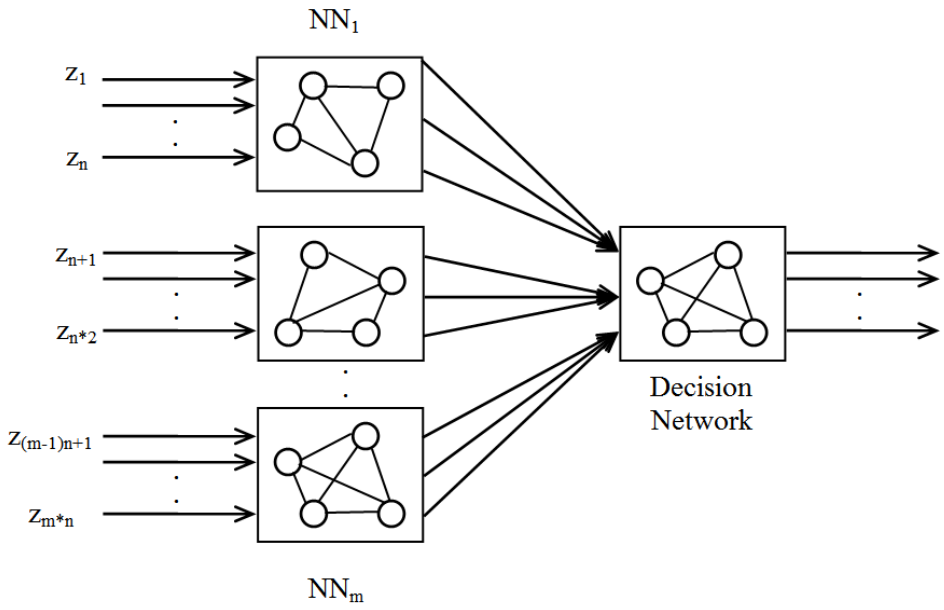


Figure 2.16: A two-layer MNN with a moderate allocator and strong coordinator.

Two example MNN architectures are discussed next. Figure 2.16 shows a two-layer MNN used for image classification [98]. The allocator (not shown in the figure) breaks an input pattern with a very large dimension of mn up into m smaller input vectors. Each of these vectors are then presented to individual modules, which allows different modules to work on different aspects of the problem. The coordinator is called the 'decision network' and is responsible for combining the output of the m modules into a classification result. It is shown in [98] that much higher generalisation performance is achieved by this MNN design over a conventional neural network architecture.

The 'NNTree' [125] is illustrated in figure 2.17 and is a possible MNN model that aims to fulfil the role of a decision tree. The difference is that neural networks are used for decision making instead of simple decisions. Each node in the tree is an expert network and all nodes have the same complexity. Figure 2.17 highlights four nodes in the tree

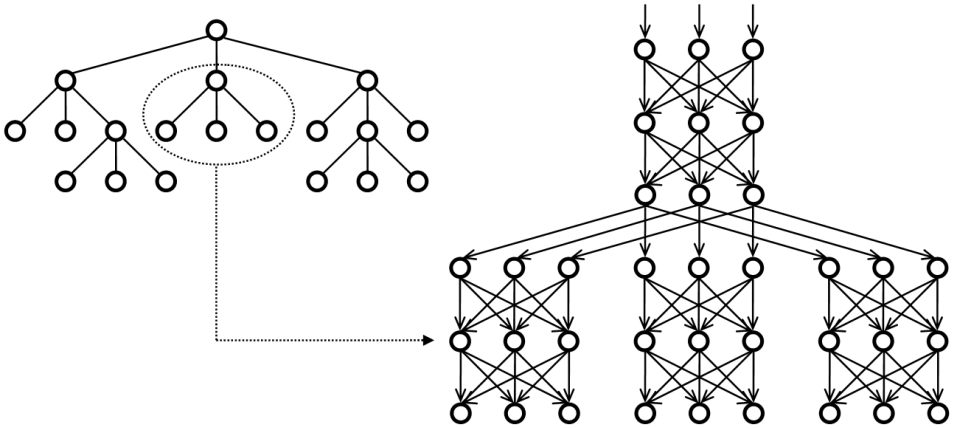


Figure 2.17: A NNTree with no allocator or coordinator.

and shows a enlargement of these nodes to illustrate how each node is a network module. Input is given to the root node for evaluation to produce an output vector $\mathbf{o}$. If the output unit o_i of the node is the maximum, the entire vector $\mathbf{o}$ is assigned to the i-th child and the process continues. If the node is a leafnode, the entire $\mathbf{o}$ is the output. In this way, a decision tree is traversed and the final output is both a specific node (representing a particular outcome) as well as an output vector. Compared to the general MNN model, the allocator of the NNTree is the tree itself and there is no coordinator, as only one 'leaf' node network is selected. It is shown in [125] that NNTrees are more efficient than traditional decision trees, as fewer nodes can be used to achieve higher recognition rates.

2.5 Summary

Neural network systems consist of a complex collection of components and algorithms. The structure of and operations on a neural network are embodied as three components namely topology, data, and learning. These components have complex interdependencies, and changes in one will almost certainly affect all the others.

The topology component comprises of the different connection schemes and neuron types that together form a neural network. A neuron is a single computation unit that comprises of a pipeline of processes and phases. These neurons are connected together via a multitude of ways to form a network with one or more layers.

The data component provides information about a problem that the neural network needs to approximate. This can be anything from function approximation to classification or clustering problems to name a few. Collectively the data set represents a function $F_D : Z \rightarrow O$ which maps input vectors from the set Z to target vectors in the set O. Sometimes, O is not known or unavailable as is the case for unsupervised networks. Data needs to be distributed and sampled into specific data sets, namely a training set D_T, a generalisation set D_G, a validation set D_V and a candidate set D_C (in the case of active learning). The order in which patterns are presented also needs to be defined. Active learning is introduced as a way for a neural network to decide which patterns are most informative to its learning, and then train on those patterns first.

Neural learning involves changing a network's topology to better solve a given problem. This includes aspects such as adaptive activation functions to update neuron functions, architecture selection to change the layout of the topology, and weight vector changes so that the neural network function F_{NN} approximates F_D better. Training is complete when $\|F_{NN} - F_D\| < \tau$ for a given τ. Neural network learning algorithms are divided into three main classes namely supervised, unsupervised and reinforcement learning. Hybrid learning methods consist of a combination of two or more of the above approaches. Neural learning can also be achieved by means of many types of optimisation algorithms, such as PSO, gradient descent, and EC.

A generic neural network framework needs to be very flexible in order to facilitate as many different types of neural models as possible. This chapter provided the base requirements for a neural network framework that is developed in chapter 5.

Chapter 3

Design Patterns

> *Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimise it.*
>
> — *E. Gamma, R. Helm, R. Johnson, J Vlissides (GoF)*

The success of every well-written software application depends greatly on its design. Every application has specific objectives that it has to address, yet developers often find that many of these objectives tend to be opposites of sorts. Optimising one objective tends to make others inefficient or even impossible to achieve. Classic examples in Computer Science include 'Storage vs. Time' and 'Simplicity vs. Flexibility.'

As recently as a decade ago, designing a good and maintainable application was more of an art than a well-defined methodology. Developers usually needed years of experience to build a well-designed application. In most cases the typical development cycle relied on trial and error along with many iterations of a methodology where requirements were reassessed and taken into account. The designs that resulted from such methodologies eventually did deliver the required functionality, but weren't necessarily extensible or reusable.

It was often the case that a specific application was working well, but its functionality could not be reused or extended due to its monolithic design and tight coupling between components. This lead developers to start thinking differently about application design. Hardware was getting cheaper and 'absolutely optimal code' was no longer the key requirement. Developers started to consider extensible applications to be more useful than a very fast monolithic design. The concept of 'write once, use many times' also became increasingly important. The rapid rise and general acceptance of the Java language and the J2EE standard by companies such as Oracle, IBM, BEA and virtually the entire IT industry confirms this.

In 1994 Erich Gamma, Richard Helm, John Vlissides and Ralph Johnson released their world renowned book called *Design Patterns: Elements of Reusable Object-Oriented Software* [39]. This book became one of the most well-known software engineering references ever written. The authors became known as the *Gang of Four* (GoF). In [39] the GoF show the important role that design patterns play in architecting complex applications. Design patterns made a science out of writing software that was modular, extensible and efficient as well as easy to read and understand. Since 1994, many authors such as Larman [73] have applied and expanded on the core GoF design patterns by incorporating them in application development lifecycles and methodologies.

The essence of a *design pattern* is to avoid repeatedly solving the same design problem from scratch by reusing a solution that solves the core problem. This pattern or *template* for the solution has well-understood prerequisites, structure, properties, behaviour and consequences. Design patterns allow developers to solve complex problems by using specific patterns to solve specific aspects of problems in a well-proven and predictable manner. Larman [73] defines a *pattern* as follows:

> "Most simply, a good pattern is a *named* and *well-known* problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussions of its trade-offs, implementations, variations and so forth."

An example is the iterator design pattern which abstracts the logic of looping over a collection so that it can be reused by any component that needs to iterate over a set of values. The prerequisites, behaviour and consequences of this pattern are well-known

and by using it, developers are assured that the looping operation in their code is working in a well-understood manner. Design patterns remove the uncertainty and doubt related to design. Their outcomes and consequences are well documented and understood, thus making design faster and more robust.

Design patterns also provide a conceptual language that developers can use to simplify the design process. Instead of thinking about the low-level details of everyday programmatic operations, developers can think in a concise fashion on a pattern level. By using a design pattern to address a design problem, designers never have to worry about the details of its implementation while trying to solve a complex problem – the chosen pattern stipulates precisely how to proceed and what the outcomes will be. Furthermore design patterns make it extremely easy for developers to discuss ideas and debug applications by allowing developers to 'speak the same design language.' If a designer refers to using an iterator pattern to pass over all elements in a composite structure and then update the element values using the visitor pattern, everybody knows what he/she is talking about.

The GoF present just over 20 design patterns that they describe as critical. These patterns form the core of the majority of program requirements and a developer could solve almost any design problem by using one or more of these patterns. The rest of this chapter focuses on the design patterns that were used in the design of the neural network framework for CILib. It is split into three sections corresponding to the three classes of design patterns defined by the GoF [39]. These are creational, behavioural and structural patterns. Each pattern is discussed by stating its name, the particular problem it aims to solve and the solution details, of which the latter is given in UML notation (see appendix B for an overview of UML).

For each pattern, an example that relates to the implementation of a neural network system is given as this will best describe why the pattern is used. This example merely serves as an illustration of how the particular pattern could solve a certain type of problem. The full details of a generic neural network framework are discussed in chapter 5.

3.1 Creational Patterns

Developers often run into several problems related to the creation of objects. These include scenarios where:

- The object to be built is extremely complex, involving the creation of many sub-components.
- The supertype of a specific product in a framework is known, but the exact type of the product is only known when subclasses of clients implement it – which constructor should the framework refer to when it needs to create products?
- A specific object is given and more instances are needed, but the type varies at runtime.

As the name suggests, creational patterns are used to create objects. Creational patterns offer developers a way to address problematic creation of objects in a standard, well-proven manner. The following sections give an overview of three well-known creational design patterns, namely the *builder*, the *prototype* and the *factory method*.

3.1.1 Builder

> *"Separate the construction of a complex object from its representation so the same construction process can create different representations" – GoF*

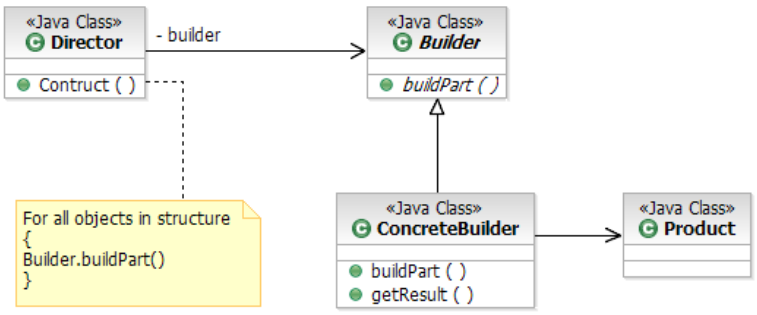


Figure 3.1: The Builder pattern

Most objects consist of members that consist of basic types and/or other objects. Sometimes these member objects themselves can be quite complex. The implication is that when such an object (called a *product*) is instantiated, all the member objects (called *parts*) need to be created as well. This includes setting up each individual part in the correct order, using the correct parameters and making sure that they can coexist with the other parts of the product.

The first difficulty in setting up the product in a simple manner by using the product's constructor is that the creation method is fixed in the product's constructor method. Changing this behaviour requires a change in the product's code. What if another instance of the product needs to be constructed where the only difference in creating it is the manner in which its parts are constructed? A good example of this would be a graph class that can be constructed in many types of base configurations such as directed or undirected graphs, simple graphs (no loops), complete graphs on n vertices, complete bipartite graphs, disconnected graphs, various different trees, among others.[1] A clean way to solve this problem is to use the builder design pattern.

The builder pattern (shown in figure 3.1) is used to split the construction process of a complex object from the representation of that object. It performs this role by supplying a separate interface that allows a client to create new products. The class that implements this interface is known as the *builder* and can be changed dynamically at runtime. By using the builder object, clients do not need to know how to create parts of products. When all parts are constructed by using the builder object's interface, a `getResult()` method is called to return the fully constructed product. To obtain different configurations of the same product object, developers merely need to define different builder objects to create parts differently.

The advantages of using the builder pattern include

- the ability to easily switch between different product creation implementations,
- a product's parts can be varied easily, and
- the ability to decide how to construct a complex object at runtime.

[1] These are just some types of graphs as noted in [29].

The builder pattern is well suited to be used in the construction process of a generic neural network topology. Recall from section 2.1 that a neural network topology houses information about neuron layers, neuron transfer functions, weight connections values, and the weight connection scheme. The type of neural network that is represented depends on the structure of the neuron layers in the component that implements the topology. The builder pattern can be used to delegate the construction of the neuron layers, weights and neuron functions to a topology builder object. To construct any of the needed parts, the client merely calls the appropriate method in the builder.

3.1.2 Prototype

"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype." – GoF

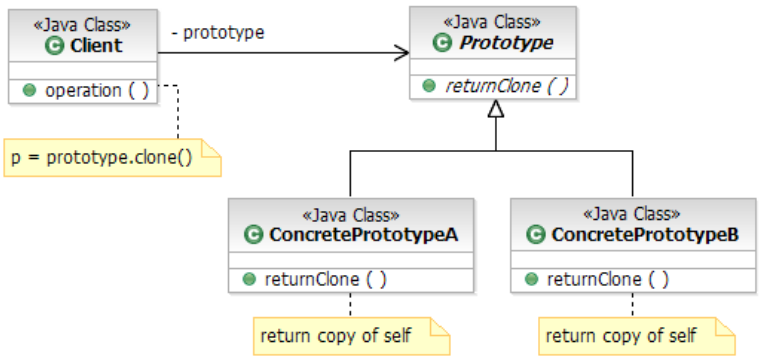


Figure 3.2: The Prototype pattern

The prototype pattern (shown in figure 3.2) is used in cases where an object is given and it is required that more objects of the same type be created, without knowing the type of the object. The prototype pattern allows developers to make deep copies of objects (copy the object's members instead of only referencing them). In most object oriented languages a new object is instantiated by creating a variable and assigning a new instance of a class to the variable using a constructor.

Consider a client application that needs to create many instances of certain types. A simple design would have the client application contain the instantiation logic of which

classes it needs to instantiate. When a new object is needed, the client simply calls the appropriate class constructor. The main drawback of this static approach is that the type of the object is explicitly stated in the client application code, which can yield the following design problems:

- If a new type of object needs to be added to the application, the client application code will need to be modified to allow this new type to be used. This means a lot of rework and direct code-level changes.
- Certain products are easy to construct, while others can be quite complex to initialise. The client application needs to be aware of each type and how to construct it. If there are many types, the client code can become quite complex.
- A possible solution to the previous point could be to use other creational design patterns such as a factory or builder to facilitate the creation of complex types. If the type hierarchy is very large, a parallel hierarchy will need to exist for the creational patterns, which can become messy.
- If all types are statically defined in the client application, it is difficult or even impossible to create new types based on runtime conditions.

The prototype pattern allows a developer to create an object in a dynamic way without knowing its type. A prototypical object can be assigned at runtime and more copies of it can be made at any time. By using the prototype pattern, developers can avoid rewriting code when more object types are needed. This results in reduced coupling between components as object creation and usage are cleanly separated.

A prime example of the prototype pattern in neural network implementations is the performance reporting mechanism. There are many types of performance metrics available such as MSE, SSE, ROC (receiver operating characteristics) and AUC analysis (area under ROC Curve) [30], distance based errors, percentage correctly classified patterns, among many others. All of these functions are created differently, yet they are used by the same component that evaluates an epoch. To avoid cluttering the neural network logic with details on which metric instance to create during each iteration of an epoch, a prototype error can be set and all future objects created by copying this prototype.

This allows more metric types to be added in future without having to rewrite the neural network component.

3.1.3 Factory Method

> *"Define an interface for creating an object, but let subclasses decide which class to instantiate. The factory method lets a class defer instantiation to subclasses." – GoF*

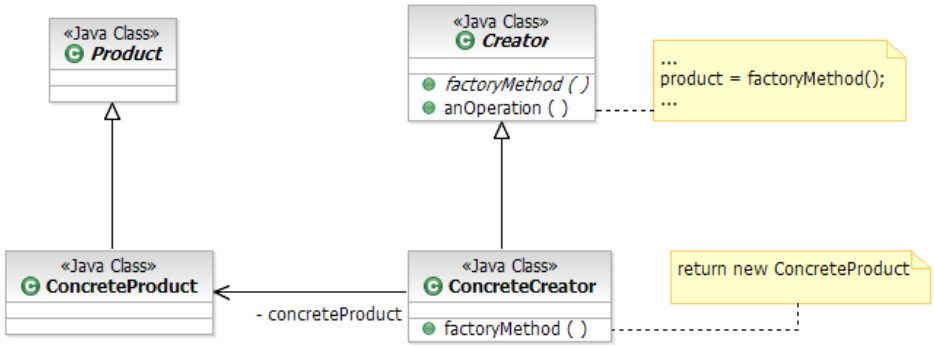


Figure 3.3: The Factory Method pattern

A framework is meant to provide a reusable shell that developers can use to build applications with. The framework defines high-level components, responsibilities as well as predefined relationships between components. In many cases the framework is responsible for *creating* many of these components as well.

The situation frequently arises that the framework itself needs to be responsible for the creation of a product it defines, yet this product is defined as abstract – the developer that implements a specific application using the framework has to implement the actual class that represents the product. The problem the framework developers face is how to create an instance of a product that they have no knowledge of as there are endless possible application-specific product implementations.

The factory method design pattern shown in figure 3.3 allows developers to remove the logic of *which* objects to create from the framework and lets the framework only be concerned with *when* to create the object. Subclasses of the abstract creator class can

then implement different `factoryMethod()` implementations that return different types of products. The factory method is useful in situations where a class does not know the type of the product object it needs to create, or when subclasses have the responsibility of deciding which type of product to create.

An example of the use of the factory method in CILib is given in section 4.1.6.

3.2 Behavioural Patterns

Behavioural patterns endeavour to describe both the relationships between objects and classes as well as the communication between objects. There are two main types of behavioural patterns:

- *Class* patterns use inheritance to distribute different behavioural characteristics between classes. An example is the template method pattern (see section 3.2.5).
- *Object* patterns distribute behaviour via object composition rather than inheritance. The chain of responsibility pattern (section 3.2.7) is a good example.

Behavioural patterns typically complement each other to provide clean application designs which promote loose coupling, better encapsulation and the ability to obtain synergy between patterns very easily.

3.2.1 Strategy

> *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." – GoF*

The situation frequently arises that in a particular implementation a certain component needs to be exchanged for another one. This can be due to a number of reasons, such as the need for a different implementation of the same logic or a different algorithm entirely. An example of this is a spell checker in a word processor. Many implementations of a spell checker can exist, such as implementations for different languages. The

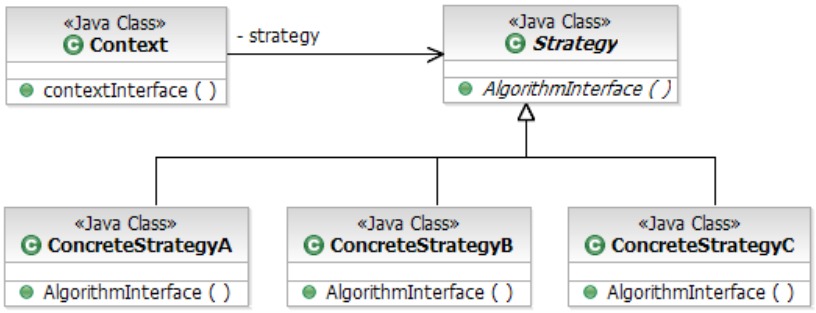


Figure 3.4: The Strategy pattern

same spell-checker can also be reused in other applications in the same suite such as a spreadsheet editor.

The strategy pattern, illustrated in figure 3.4, allows a developer to wrap the required functionality into coherent interchangeable units. By keeping the interface to the implementations the same, classes can reference specific methods and in this way access different versions of an algorithm. The advantages of using the strategy pattern are:

- Algorithm encapsulation is preserved,
- client classes are simpler,
- clients are easier to extend with additional functionality, and
- the behaviour of certain client classes can vary dynamically at runtime based on the strategy object it is using.

The strategy pattern plays an important role in developing generic neural network implementations. A top-level example is the interface between the neural network and its data sets. Best practises dictate that the data set as well as the operations on that data set are encapsulated in a single component. Yet different approaches exist that define how to distribute the data into the sets D_T, D_G, D_V and D_C (discussed in section 2.2.1.6). Different algorithms also exist such as active learning approaches (see section 2.2.3) that involve the redistribution of data patterns between the mentioned sets based on different algorithms.

By encapsulating all these responsibilities in a single component type using the strategy pattern, any neural network topology can use these different approaches. This requires that the strategy component's interface is rich enough to support all operations on data for example the ability to trigger an active learning update.

3.2.2 Iterator

"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." – GoF

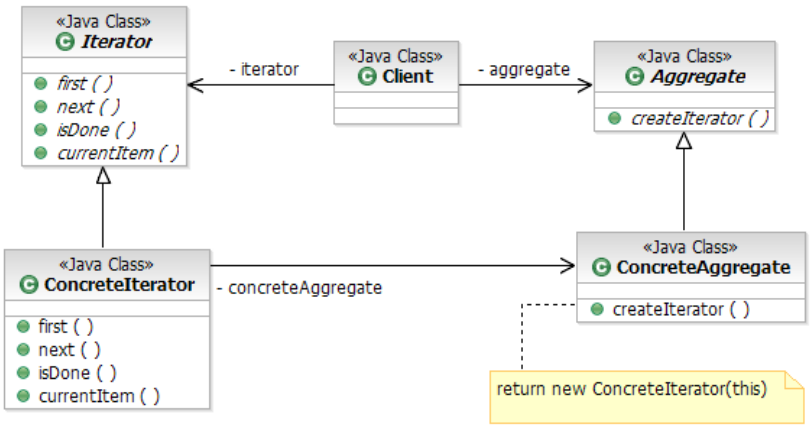


Figure 3.5: The Iterator pattern

One of the most common operations in application development is looping over a set of elements. Yet every time a developer needs to write a loop over such elements, issues such as loop conditions, loop order and accessing the current element has to be solved. All three these operations have to be done for every loop regardless of what type the elements are. The developer must also have knowledge of the underlying structure of the elements he is traversing, for example, a tree – two such traversals on a tree are breadth-first traversal and depth-first traversal. Furthermore, if there are many nested loops, the problem gets compounded and programming errors become common.

The iterator pattern (shown in figure 3.5) addresses this issue by abstracting the three mentioned fundamentals of any loop. By developing an iterator for a collection

of elements, the structure of the collection is decoupled from the client component that iterates over it. By specifying an iterator class for a type of traversal on the collection and encapsulating the three looping operations in the iterator, any traversal looks identical from a client component's perspective. As an example: if a client application needs to iterate over a tree, the client would not have to have its code changed to switch from breadth-first traversal to a depth-first traversal – by merely switching the iterator object, any new iteration can be used. This new iterator can also be loaded dynamically at runtime, giving even more flexibility.

The advantages of the iterator pattern are:

- Client components can use one interface to the iterator to specify many possible types of traversals on a collection.
- Existing traversals on a collection can be optimised to perform well and then reused by any client component. A developer never needs to reinvent the wheel when considering traversing a collection.
- More than one traversal on a collection can be active at any given stage. Each traversal's conditions are managed by its own iterator object. Different types of traversals can even be running simultaneously.
- The inner details of a collection are hidden from client components. This greatly simplifies client code.

The data component in a neural network system is a key candidate for the iterator pattern. This decouples the rest of the system from knowing how to traverse data sets such as D_T, D_G, D_V and D_C or keeping track of the state of these sets (as this state changes frequently in cases such as active learning). By using an iterator, the implementation of data structures can also be changed without affecting the rest of the system. Now it becomes possible to define data classes that use Java ArrayLists, basic arrays, dynamically generated data, or any other type of storage as data type, without having any client component have to know the details.

3.2.3 Visitor

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the element on which it operates." – GoF

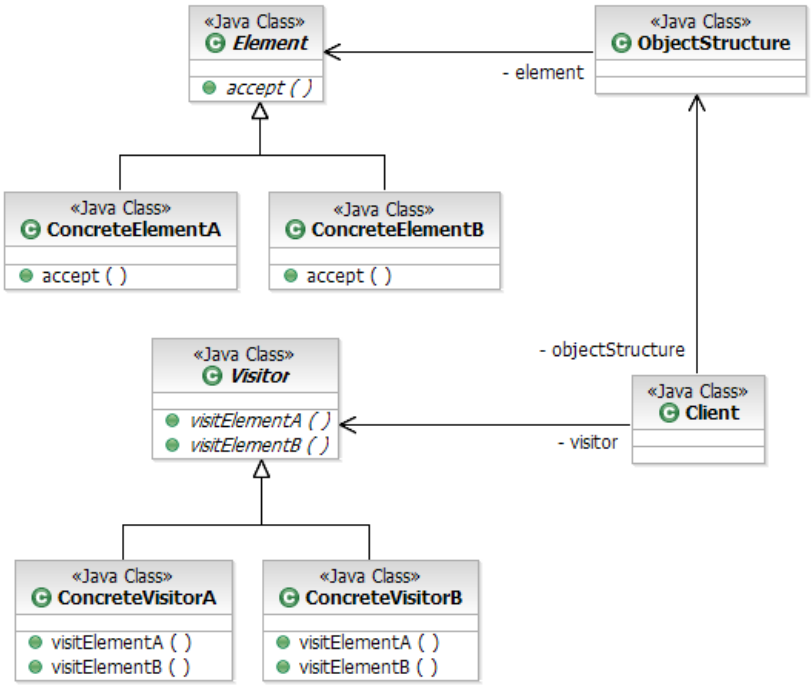


Figure 3.6: The Visitor pattern

The most common operations on the elements of a collection are to retrieve and set their values and is usually achieved by using getter and setter methods in Java. Yet in many cases a developer wants to perform an action that is applicable to the entire collection. Typical operations include getting the average value, setting each element to a random value, finding the sum over all values or extracting all the values from a complex collection such as a tree and returning it as a sequential list. The most obvious way to achieve these objectives is to write a method in the collection class that performs the requested operation. However, consider the following problems:

- If there are many such operations, the collection class will be overly complicated.

- When a new operation is needed, the collection class code will have to be modified.
- Should the underlying structure of the collection change, all operations may need to be updated to still work correctly.

The visitor pattern (see figure 3.6) decouples the operations applicable on a collection from the representation of the collection. The collection class never has to change when a new operation is added to it, as no operations are actually implemented in the class itself. By using a visitor object to update the elements (via the collection's interface), the implementation of the collection can change without affecting the visitors defined for it. This of course assumes that all information needed by the visitors is still available to them.

Visitors can be used to good affect in the topology of a neural network implementation. The topology can provide an `acceptVisitor()` method that traverses the underlying structure of neurons and weight connections, and allows the visitor to access each of these elements. This allows developers to perform operations such as random weight initialisations, weight value extraction, setting weights to a specific weight vector and many other operations that are applicable to neurons or weights in a neural network architecture. The visitor code is independent of the neural network topology. This can enable the use of weight initialising visitors in any type of architecture such as FFNN, SOM or Hopfield networks for example.

3.2.4 Observer

> *"Define a one-to-many relationship between objects so that when one object changes state, all its dependents are notified and updated automatically."*
> *– GoF*

One of the most important aspects to handle in application design is a one-to-many relationship between components. This situation occurs when a group of objects (called *observers*) all depend on the state of a central component (called the *subject*). If any change occurs in the subject, each of the observers needs to be updated. In a well understood system this responsibility can fall with the subject. The subject keeps a

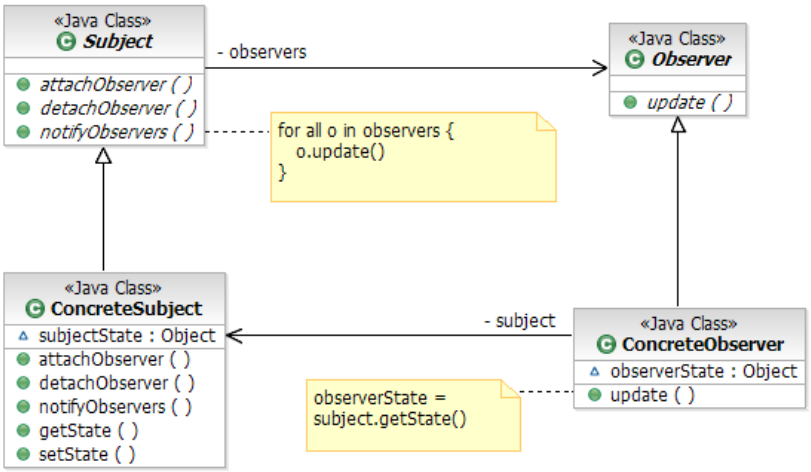


Figure 3.7: The Observer pattern

reference to each of its dependants and then merely updates these observers when the need arises. This is fine for a static system with few dependants, as the functionality to update these objects will not overly complicate the subject and will not change often. There are shortcomings to this approach:

- If more objects need to be added to the list of observers, code changes in the subject are required.
- The code to update observers will eventually start having a negative effect on the subject's performance if the list of observers becomes large.
- All the dependants are active all the time if the update code is added to the subject. It can be very tedious or even impossible to change the code to include only certain dependants at any given stage.

The observer pattern, illustrated in figure 3.7, allows a developer to extract the functionality of updating dependants (observers). The only functionality required in the subject is a `notify()` method. When any change that might affect dependants occurs in the subject, the `notify()` method needs to be called as well. This method iterates over a list of registered observers and informs each observer of the update and also allows the observer to take corrective action. This allows any number of observers to be dynamically

registered with the subject at runtime, without complicating the subject's code. The required functionality is already implemented for any possible number of observers. A client merely has to register the required observers with the components.

The topology component is the heart of any neural network implementation. A good generic topology component typically has the expressive power to implement any possible neural network architecture. Due to the nature of neural network systems, all other components such as the learning algorithm, the data source, error reporting mechanisms and the main neural network algorithm are dependent on the state of the topology component.

In order to ensure that all components with inter-dependencies work together in concert, the observer pattern is critical. One component may modify the topology state and in so doing invalidate other components. A prime example is when an architecture selection algorithm changes the network and other components (such as an error reporting system, a learning algorithm or components external to the system) are not aware of this change. The problem may not be picked up at all, leading to erroneous results. The observer pattern can be implemented in the topology class and any class that uses the topology can register itself as an observer. After any significant changes occur, the `notify()` method can be called and the change can be handled by all observers. In so doing, all components will be able to validate any changes and throw an appropriate exception if necessary.

3.2.5 Template Method

> *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." – GoF*

During the design of an application, it often happens that there is a great deal of common behaviour between certain components. Yet each of these components is self-containing – they don't explicitly reuse the functionality they share. An example of this is a collection of optimisation algorithms that follow the same high-level steps in optimising a problem, but use totally different techniques to achieve their goals. Essentially these algorithms share the same steps, but perform each step in different ways.

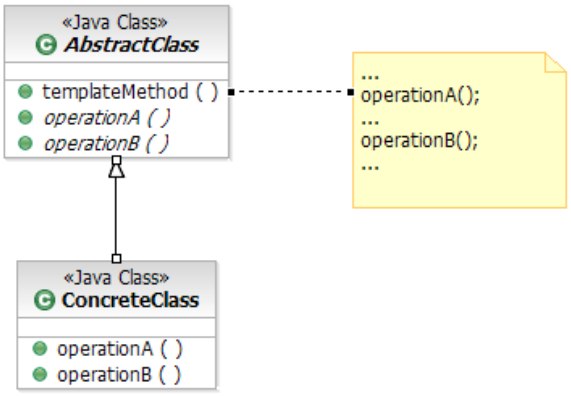


Figure 3.8: The Template Method pattern

The template method pattern is used to extract the behaviour that is common between components and present a template class as a skeleton or outline of steps. Subclasses of this template class redefine each step as they need to by overriding the inherited operations, as figure 3.8 shows. By using a template method, designers give the parent class control over the invocation of subclass operations, while subclasses provide the details of the operations.

Top-level application operations that are common to all neural network types such as constructing all the components, initialising the system, running experiments, performing one iteration of the experiment, handling stopping conditions, finalising operations after an experiment, and any other operations common to algorithms can be implemented using one or more template methods. This allows different subclasses to implement these operations. In chapter 4, CILib uses a template method to facilitate the above functionality, which enables any type of CI algorithm such as PSO, ACO, EC or NN to share the same base class and template methods.

3.2.6 Mediator

> *"Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently." – GoF*

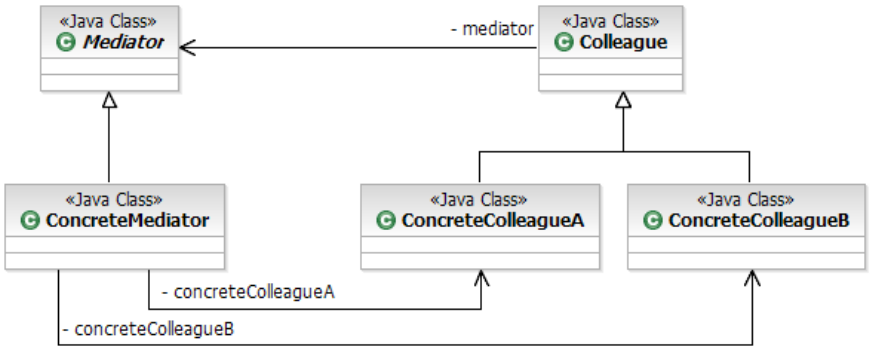


Figure 3.9: The Mediator pattern

Object-oriented programming promotes the separation of concerns into different components. Each component or class encapsulates only the logic that is needed to perform its intended objective. This enables developers to implement systems cleanly and effectively. It also introduces the need for these separate components to integrate with each other.

The easiest way to integrate two objects is to have them merely refer to each other. While this is a fast and simple solution, it has the drawback that it is not very flexible from a reuse perspective. If left unchecked, it might occur that objects communicate in complex and unstructured ways which becomes very difficult to understand. If an object refers to many other objects, it becomes very difficult or impractical to reuse it in other areas. Furthermore, the intended behaviour gets distributed among a set of objects, which becomes difficult and error prone to maintain.

The mediator design pattern is illustrated in figure 3.9 and acts as a broker between individual objects, called *colleagues*. The mediator pattern promotes loose coupling between colleagues and greatly simplifies the design of systems where many objects need to interact with one another. A mediator object is responsible for controlling and coordinating all interactions between its colleagues, yielding consequences such as:

- Subclassing is mostly limited to the mediator class. If a new system behaviour is desired, the colleague classes do not need to be modified – only the mediator class needs to be changed.

- Loose coupling between colleague objects is achieved. This makes it easy to change

colleague and mediator implementations without affecting other colleagues.

- Interaction between colleague objects is greatly simplified as all many-to-many relationships are replaced by one-to-many relationships, which are easier to understand and maintain. It also becomes easier to reuse an existing component.

- The complexity of object interaction is factored out of the respective colleague objects and is centralised in the mediator class. While this simplifies the entire system, good design of the mediator class is needed to ensure good performance, maintainability and ease of reuse.

The mediator pattern can be used in neural network implementations to help integrate all the disparate components. The network topology, data source and learning algorithms may be implemented in different components. A mediator class can be used to good affect to integrate these separate components in a way that promotes loose coupling. By doing this, it becomes very easy to reuse existing components to form completely different implementations by merely exchanging one component for another. An example of this is to exchange one learning algorithm for another while keeping all other components the same.

The mediator class itself can also be changed to construct completely different applications that consist of exactly the same colleague classes – the only difference is the manner in which components are invoked. For instance, a mediator implementation that only uses the data set D_T to train a network can be exchanged to train a network on D_T as well as validate learning using a validation set D_V. A further validation may also be performed after training by using D_G. Note that only the central mediator is changed, but that completely different behaviour is implemented.

3.2.7 Chain Of Responsibility

> *"Avoid coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it." – GoF*

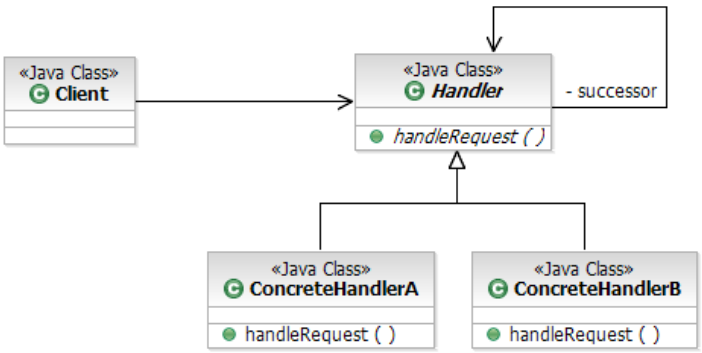


Figure 3.10: The Chain Of Responsibility pattern

It frequently happens that an object needs to pass a request to other objects to handle the request, either because the object cannot handle the request itself, or needs other objects to assist in handling the request. In certain circumstances it might be the case that the sender does not know which receiver should handle the request. The receiver may also not be directly accessible by the sender, which requires that the request be passed along a chain of objects (or intermediaries) before the intended receiver can handle the request. More than one object may also need to handle the request as it is passed along the chain. Another requirement may be that the receiver should be specified dynamically at runtime.

The chain of responsibility pattern (see figure 3.10) allows senders to be decoupled from the receivers of a request. The requester passes the request to a request *handler* object. This handler has a method that is known to the requester as well as links to other possible handler objects (if any). The handler then decides if it can handle the request or if the request needs to be passed on to other handlers. In this way, many handlers can help meet the request, but the original requester only knows about the initial handler object. As this handler can be a Java interface or abstract class in the purest form of the pattern, this handler can be set dynamically.

Consider the top-level application that is responsible for setting up a neural network system. A neural network implementation in the CILib framework is a prime example. The complete design principles and details of CILib are discussed in chapter 4. It is sufficient at this point to know that CILib has a simulator component that is used to

compose any CI algorithm dynamically at runtime using XML. The chain of responsibility pattern can be used here to delegate certain requests to lower-level objects. The way this works is to define the object structure in a tree-like representation using a combination of default constructor calls, getter and setter methods as well as method calls. After all objects have been created, the simulator uses the template method pattern to call an `initialise()` method on the top-level `Algorithm` object. This method is meant to be overridden to allow the particular `Algorithm` class to initialise itself. Yet the lower-level classes that make up the neural network implementations might also need to be initialised.

The chain of responsibility pattern allows the root `Algorithm` class to initialise itself and pass the initialise request to all of its member objects in a depth-first manner. In this way, the initialise request can be extended to the entire object tree. Note that this is not the same as the builder pattern in section 3.1.1 – the builder pattern is used to construct a complex object while the chain of responsibility pattern calls a method on each element of an already existing chain of objects.

3.3 Structural Patterns

The main objective of structural patterns is to realise larger *structures* in sets of classes or objects. Structural *class* patterns use inheritance and other methods to reconcile different classes' interfaces and implementations with each other. The adapter and facade patterns are good examples. Structural *object* patterns use object composition to dynamically build structures, allowing concepts like variable structures and object sharing to be utilised. The composite pattern is a good example of a variable structure, allowing applications to build complex tree-like structures during run-time.

3.3.1 Facade

> *"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."* – *GoF*

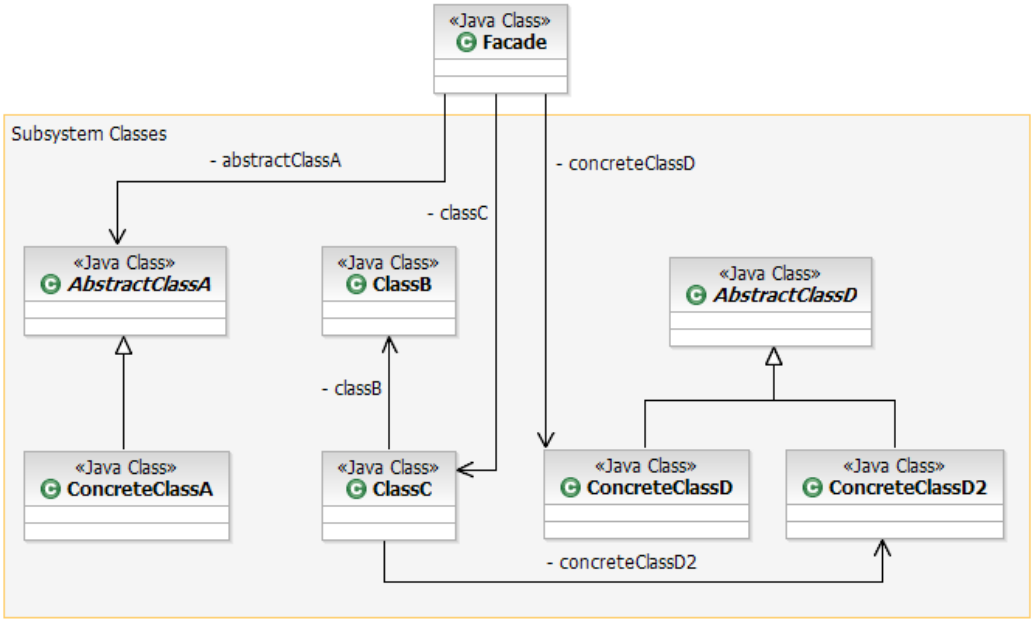


Figure 3.11: The Facade pattern

Most systems today are built using more than one component. This enables clean separation between functional areas which simplify implementation, maintainability, extensibility and reusability. Unfortunately it also complicates the usage of the system. All the components need to interact with one another in a certain manner and this functionality is included in their interfaces. The client application that uses the system doesn't need to be exposed to these functions – from an encapsulation perspective clients should not be able to see these inner details at all.

The facade pattern (shown in figure 3.11) is used to wrap all the components of a system and then present just one, smaller functional interface to the client application. The only interface the client needs to have access to is the facade component's. It is the facade component's role to delegate the requests to the correct components in the system.

The facade pattern can be used many times to varying degrees when implementing neural networks. At the top-level the entire neural network system can be implemented as a facade, which allows easy access to the functionality of all lower-level components. The client can now evaluate a single data pattern, train a neural network on data, obtain

the details of the neural network topology, or access any other network function from a single interface. The topology component can also utilise the facade pattern to hide details such as builder objects, neuron implementations, weight connections, and other implementation details and present a simple interface to manage a topology.

3.3.2 Adapter

> *"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." – GoF*

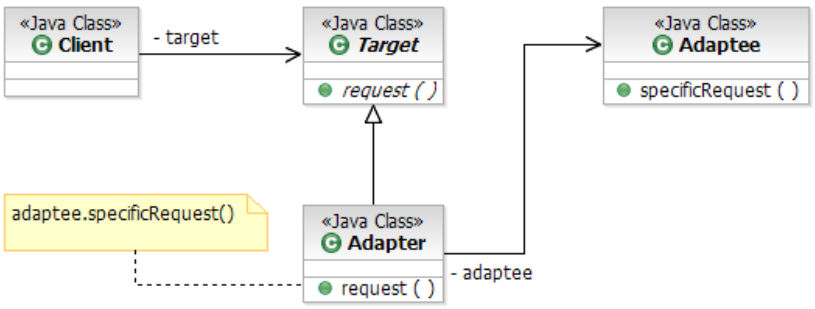


Figure 3.12: An Object Adapter

The main goal of the adapter pattern is the reuse of existing components. It achieves this goal by mapping operations of an existing component onto corresponding elements of a new component that wants to reuse the existing implementation. There are two main kinds of adapters in the object oriented programming model, namely the *class* adapter and *object* adapter. A class adapter uses multiple inheritance to adapt one class's interface to another. Multiple inheritance is not possible in Java, but this pattern can be realised through Java interfaces – the interface provides the desired requested interface and the class with existing methods provides the implementation. The interface's methods then merely forward calls to the correct existing methods. Effectively, an existing java class merely implements more methods of a new interface while the functionality is already provided by existing methods. This allows objects of the old class to be treated as instances of the new interface.

An object adapter relies on object composition (see figure 3.12). Rather than mapping interface methods to other class methods statically, the adapter class maps the requested methods to the matching interface of an *object reference*. This entails that the referenced object can be changed dynamically, thus allowing different implementations to be assigned to the adapter at runtime. Note that this is not the same as a class adapter which only maps methods in the same *class*.

Adapters can be used throughout CILib – a typical example would be when using other optimisation techniques such as PSO or EC and neural networks together as described in section 2.3.1. Each optimisation technique can only work on certain types of data objects, as specified by their interfaces. The neural network component may specify a different interface with different object types. By using adapters, the data objects and interfaces of the neural network can be mapped to the data objects and interfaces of the optimisation algorithm.

3.3.3 Composite

> *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." – GoF*

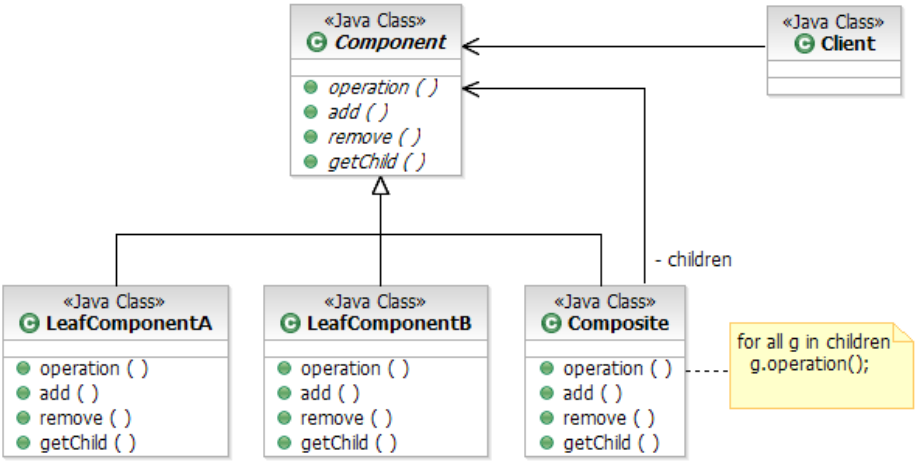


Figure 3.13: The Composite pattern

Consider a client application that has a requirement to create many different types of objects as well as collections of these objects. Furthermore, collections of objects may also contain other collections of objects, thus making this a recursive relationship. This relationship can be realised with many different classes, for example a class hierarchy to support different basic types and a class that acts as a container for these types using a storage data structure such as a Java `ArrayList`. Yet another class can be written to create collections of this container class to give the client application full recursive composition capability.

The disadvantage of this approach is that the client application now has three different types of classes to work with. In many applications this can become quite a problem as the types need to be treated differently depending on whether they are an isolated object, a container or a recursive container. In [39] the GoF illustrate this problem by using a graphical drawing tool's drawing interface. In a typical drawing tool, users can create single objects such as lines, circles, rectangles, images and so forth, but can also group these types to form *groups* of objects. The user treats a collection of objects as an object as well. This means that the client application needs to know how to perform the same actions on all three different object types, which increases complexity, duplicates effort and increases the chance of programming errors.

The composite design pattern allows developers to create type hierarchies that incorporate recursive composition containers *as part of the hierarchy.* This means that the class that acts as the container for the objects is part of the same hierarchy as the types it contains, thus allowing collections to be treated as a subclass of the same type. The client application can thus refer to the common base class and not be concerned whether the type is a basic type, or a collection of those types. By letting the collection class contain instances of the common base class, it may contain a mix of basic types and other collections thus allowing tree structures to be built. The composite pattern is illustrated in figure 3.13.

The advantages of using the composite pattern are:

- Primitive and composite objects are contained in the same hierarchy, allowing primitive objects to be grouped into collections.
- Recursive composition of objects is made possible and simple.

- Client application code is made simpler as objects and collections of these objects are treated uniformly.
- It is much easier to add new types. These new types will work with the existing primitive and composite types already defined.

The composite pattern has been used to good effect by CILib developers in chapter 4 as a way to build a common `Type` system.

3.4 Summary

This chapter discussed the main reasons why design patterns are needed to design systems efficiently. The advantages of using design patterns correctly include:

- **Flexibility.** Design patterns are aimed at giving more flexibility to applications, allowing components to be switched very easily and with very well-documented outcomes.
- **Reusability.** Design patterns allow well-written and proven components to be reused in other applications with little or no modification. This allows new applications to be built much faster, easier and with fewer errors, without 'reinventing the wheel.'
- **Extensibility.** A direct consequence of many design patterns is that more functionality can be added to a system at a later stage with minimal effort. More functionality is simply contained in a new component that can be used by existing applications.
- **Speed.** Development is made much simpler which in turn allows solutions to be developed faster due to clearly understood system behaviour and consequences. Reuse of well-written components also aids in developing new applications faster.
- **Reliability.** Design patterns, the problems that they are applicable to, their consequences and their relationships are clearly understood and documented. This

means that developers are never in the dark as to what the effect of any particular pattern will be.

- **Separability.** Design patterns break a large problem down into smaller components. Each of these components can then typically be solved by a specific design pattern. This approach clearly separates functionality into specific components, allowing components to be exchanged or modified without affecting the rest of the application too much.

- **Maintainability.** Application maintenance is much easier in a system that comprises of design patterns, as application functionality is clearly separated into specific components. Changes in one component have fewer unintended side-affects in other components when design patterns are used.

- **Dynamicity.** Many of the patterns discussed in this chapter allow application behaviour to be modified at *runtime*. This characteristic is absolutely critical in many applications today.

- **Communication.** Developers can share the same vocabulary when discussing application designs. Design patterns allow developers to talk about a problem at a higher level of abstraction than merely the programming language or modeling tool.

Design patterns are crucial in designing good object-oriented software. Good use of design patterns typically yield application designs that are smaller, simpler and better understood than designs that don't use patterns. Patterns allow design problems to be solved *methodically* using a template rather than being solved 'from scratch' each time.

The next chapter shows how design patterns can be used to solve common yet challenging problems that designers face when building CI implementations. The purpose and layout of the CILib library, how design patterns are used to solve complex problems and how the framework lays the basis for any CI implementation are discussed.

Chapter 4

CILib – Computational Intelligence Library

The hard and stiff breaks. The supple prevails.

— *Tau Te Ching*

The Computational Intelligence Library (CILib)[1] is an open source project that started off as the Masters degree topic of E. Peer in 2001 [85]. Since its inception, developers all over the world have started using CILib for research purposes. The aim of the CILib framework is to fully integrate the areas of computational intelligence (CI) such as swarm intelligence (particle swarm optimisation, ant colony optimisation, and others), evolutionary computation, neural networks, and fuzzy systems into one coherent library that allows any CI algorithm to interface with any other algorithm as requirements dictate. Examples include using the PSO algorithm to train a neural network as well as using an already trained neural network as the fitness function of a PSO. Note the duality of this example - *any* algorithm should be reusable by any other algorithm if a need for a specific type of interaction exists.

The framework should furthermore be easy to expand and be flexible to allow components to be reused efficiently. Similar to neural network systems, PSO and EA approaches also have a large number of design choices – PSO implementations have different types

[1]CILib can be found at http://cilib.sourceforge.net

of velocity and position update strategies, particle definitions, topology layouts, swarm layouts, and so forth. Evolutionary computation approaches have different chromosome and gene representations, population grouping choices, cross-over operators, selection operators, mutation operators among others. At the time of writing, CILib already included a vast number of implementations in the fields of particle swarms, evolutionary computation, game theory, ant colony optimisation, as well as custom data structures and foundation classes. All these implementations are easy to reuse and expand with new types of algorithms, many times without coding by merely reconfiguring components.

A more detailed breakdown of CILib's goals, as listed in [85], are listed below and reflect the advantages of using design patterns as listed in section 3.4:

- **Flexibility and reusability:** Design patterns should form the backbone of the architecture to allow for the creation of a framework capable of expressing all the complex interactions found in CI models. Furthermore, if a particular component implementation can be realised by combining the functionality of other existing components, the framework must allow this to be done easily and effectively.

- **Experimentation:** CILib should allow the easy configuration and execution of simulations on any specific CI problem. The framework should allow any component properties to be measured while allowing any combination of stopping conditions to be set. These simulations should be fully configurable at runtime without making any changes to CILib code.

- **Collaboration:** By having developers of different CI fields working together in the same framework, it becomes easier to integrate components from different fields. New developments in particular areas will also be accessible immediately to other developers without rewriting code.

- **Efficiency:** There must be a well-balanced trade-off between pure object-oriented design principles and improved performance. CI algorithms are generally very CPU intensive, but performance should not impact the design decisions too heavily. Parallel computing should also be supported and implemented for any application written in the framework.

- **Separability:** A clear abstraction between algorithms and problems must be defined to allow any problem to be solved by any suitable algorithm. No simulation measurements and stopping conditions should form part of the algorithm or problem components. This approach ensures that these components may be reused in many contexts.

- **Reliability:** As the project is open source, full code reviews will be performed frequently. Extensive unit testing on all components further enhances error-free code.

The rest of this chapter describes the implementation details of the CILib framework. These components form the base for the CILib implementation of a generic neural network framework as discussed in chapter 5.

4.1 Components in CILib

CILib defines a *framework* consisting of multiple components and predefined relationships between these components. This framework guides developers to easily implement specific CI implementations by reusing and extending existing implementations. By having all developers use the same framework components, design patterns and principles, the ability to meet CILib's goals becomes much easier. All CILib components either extend or directly support the top-level framework to form a particular CI implementation such as a PSO or a neural network. The high-level components that make up any CILib application are:

- **Algorithm.** Provides an implementation of actions that are common to all CI algorithms. These include stopping conditions, an event processor for algorithm events, common housekeeping tasks and thread handling. All algorithms extend and use this component.

- **Problem.** An interface used to represent the problem types present in CI. Together with sub-interfaces, it provides an interface to any given problem type.

- **Stopping condition.** Provide support for custom stopping conditions. In conjunction with the Algorithm component, multiple stopping conditions can be used simultaneously via the *observer* pattern.

- **Measurements.** Provides the ability to extract information from a running simulation. Multiple measurements can be used via the *observer* pattern and more measurements can be added easily.

- **Type.** A common type system aiding developers to easily integrate with other components. Type supports arrays of type components by using the *composite* pattern, and can be extended to include any other complex type (such as DNA strands for example).

- **Simulator.** Used to instantiate any configuration of algorithm, problem, stopping conditions, measurements and all supporting classes. Configurations are set via an XML file[2] and once instantiated, the simulation is run to completion.

It is important to note that CILib uses the component types above to both guide developers in using and extending the library, and to highlight erroneous configurations at runtime. The manner in which this is done is to use Java's inherent error detection capabilities when developers try to use incompatible objects/interfaces together. The framework of components outlined above gives structure to all CI implementations and in so doing almost completely eliminates incorrect configurations. The following sections discuss the framework in more detail.

4.1.1 Algorithm

The algorithm component (shown in figure 4.1) provides the basic shell for any CI implementation to run. This is done through the `Algorithm` class. This component is also responsible for handling common housekeeping tasks related to any running algorithms and making it possible to run simulations as Java threads (if needed in a clustered environment). Each subclass of `Algorithm` is associated with at least one `Problem` class.

[2]See http://www.w3c.org/xml for an overview of the XML specification.

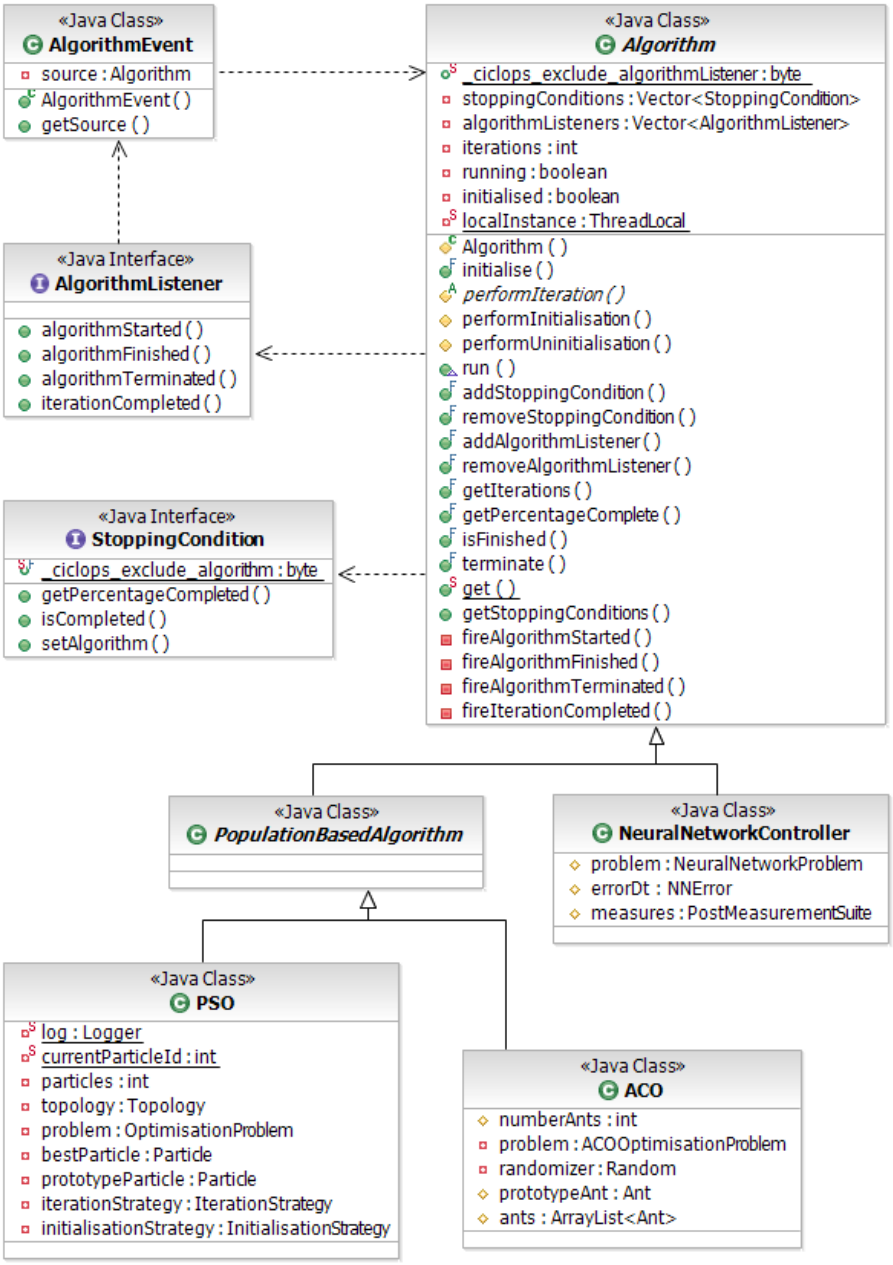


Figure 4.1: The Algorithm component

Many types of problems exist and a hierarchy of interfaces are used to define which algorithms are able to solve which types of problems. This is discussed in section 4.1.2.

The algorithm component is implemented as an abstract class `Algorithm` with the

bulk of the class providing the aforementioned functionality. The `run()` method is an instance of the *template method* pattern which handles the outline and general execution of algorithms. The `run()` method oversees calls to algorithm listeners and other housekeeping tasks as well, and also delegates a single iteration to the `performIteration()` method. The `performIteration()` method forms the main link between any CI implementation and the `Algorithm` class and needs to be implemented by subclasses (thus providing the particulars of a specific algorithm implementation).

The `Algorithm` class incorporates an event manager for handling any algorithm events. The *observer* design pattern is used to register any class that implements the `AlgorithmListener` interface as an observer. `Algorithm` notifies this list of listeners when the algorithm is started, finished, terminated early or has completed an iteration.

An important challenge that all CILib `Algorithm` subclasses face is the initialisation of themselves as well as any of their aggregated components. The `Algorithm` abstract class includes an empty `performInitialisation()` method so that any subclasses of `Algorithm` have the option to override this method to provide initialisation code if needed. This CILib framework always calls this method via the `initialise()` method in `Algorithm` by means of the *template method* design pattern[3].

4.1.2 Problem

The problem component defines the particular CI problem that needs to be solved. Every algorithm component has at least one problem component associated with it. This component is implemented as a hierarchy of interfaces with the `Problem` interface as its root. Each different type of problem is represented by a sub-interface which adds more functionality that is required for the type of problem as illustrated in figure 4.2. Only the `OptimisationProblem` interface and its sub-interfaces are shown in the figure to avoid cluttering the diagram. This type of hierarchical ordering is useful as it can limit what types of problems a particular algorithm can solve. This makes it easier to expand the library as there is a well defined structure. It also prevents errors that result

[3]At the time of writing, this was the official method to initialise components. Java 1.5 offers new ways using *Annotations* and CILib is poised to use this method in future. For more on Java 1.5, see [35].

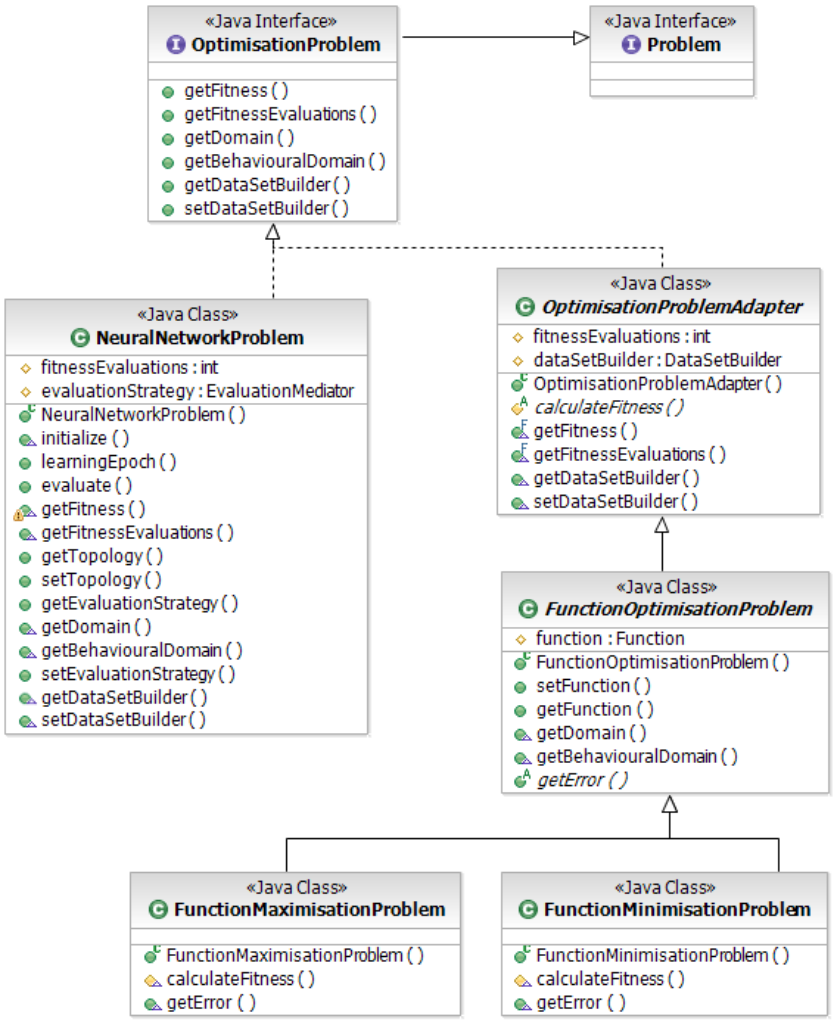


Figure 4.2: The Problem component

from incompatible algorithm-problem combinations, as the system will indicate incorrect object types. Each subclass of `Algorithm` refers to the specific type of problems that the algorithm can solve, which limits incompatible combinations.

4.1.3 Stopping Conditions

CI algorithms are typically iterative procedures where the aim is to optimise the solution to a problem as the algorithm progresses. Every algorithm needs certain conditions upon which it can terminate. Example stopping conditions include:

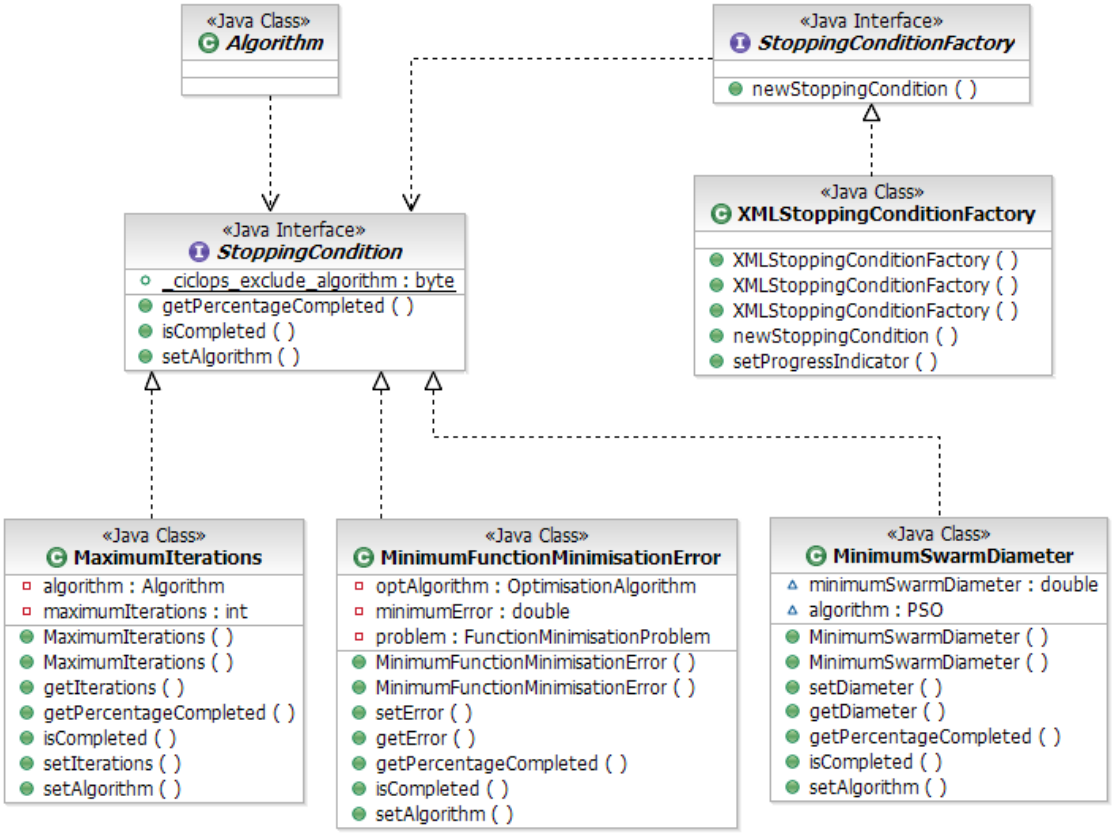


Figure 4.3: Example Stopping conditions

- A certain number of iterations has been reached.
- A certain amount of time has passed.
- The error estimate has reached an adequate accuracy.
- The error estimate does not improve significantly over time.
- The generalisation error estimate starts to become less accurate (i.e. overfitting occurs).

Many stopping conditions exist, of which certain ones are generic to all CI algorithms and the rest are specific to certain CI paradigms only.

A characteristic of CI algorithms is that more than one of these stopping conditions can be associated with an experiment simultaneously. For example, an experiment could

be set to end after 1000 epochs, stop if the error reaches a certain accuracy threshold, but also stop if the accuracy does not improve significantly over time. A typical design problem when implementing CI algorithms without the aid of CILib is that most implementations don't cater for all the possible stopping conditions, the ability to have multiple active ones or to easily extend the system with extra stopping conditions. CILib allows any number of stopping conditions to be added to any simulation easily and efficiently. Generic stopping conditions such as maximum iteration and maximum time can be reused between CI algorithms.

As mentioned before, many stopping conditions are generic, yet certain stopping conditions are only applicable to certain types of CI algorithms. An example of this is a stopping condition that is triggered when a PSO swarm's diameter is smaller than a specific value (see figure 4.3). This stopping condition would make no sense in other CI environments such as neural networks, ACO or EC. The CILib framework design ensures that an appropriate exception is thrown as soon as invalid combinations occur.

CILib includes the concept of stopping conditions as a standard part of the framework using the *observer* design pattern. The abstract `Algorithm` class includes an array of type `StoppingCondition` and the methods to add and remove any `StoppingCondition` objects. Thus any algorithm implemented in CILib has full support for multiple and simultaneous stopping conditions.

4.1.4 Measurements

All CI simulations are run with the express requirement of being measurable. Different researchers may also need to measure different things when running the same algorithm. For instance, one researcher may want to measure the training accuracy of an algorithm, while other researchers may be more interested in generalisation or time performance. The designer of an algorithm can't possibly think of all measurements that people may want to perform in future. As with stopping conditions, there may also be multiple measurements active at any given time, as well as the restriction that certain measurements can only work on certain problems.

The CILib framework allows multiple measurements to be added to a simulation via the *observer* design pattern. New measurements can easily be written by utilising

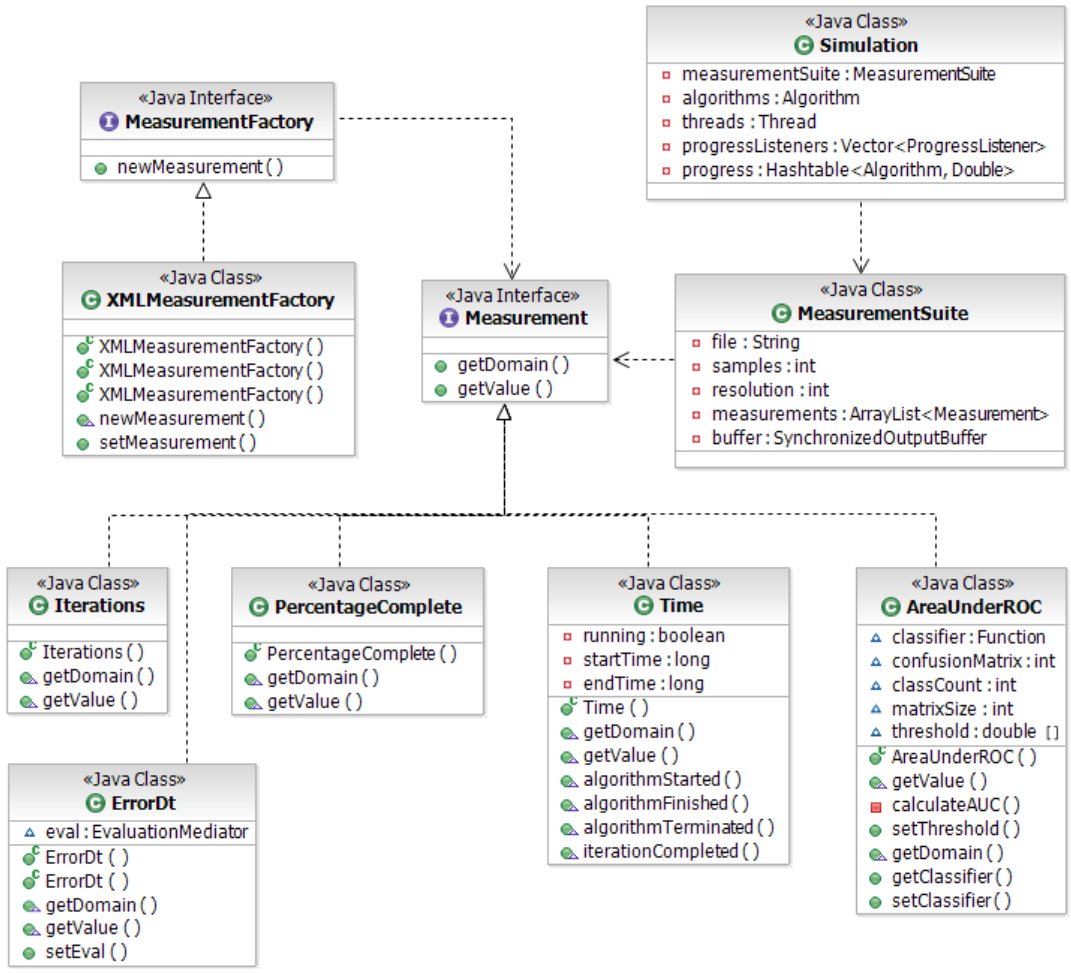


Figure 4.4: Measurement system

the `Algorithm` class's public interface to navigate the structures below. It remains the measurement's task to extract the needed information from the algorithm structure. This reinforces the fact that, as with stopping conditions, certain measurements are specific to certain types of algorithms, and this makes sense – one would for instance never invoke a *swarm diameter* measurement (applicable to PSO) on a neural network. The framework will highlight this as well, as it would actually be impossible to associate measurements to an algorithm that does not support the measurement – Java will not allow a variable name that is non-existent to be extracted from an algorithm. Some example measurement implementations are shown in figure 4.4.

An advantage of the `Measurement` classes is that `Algorithm` is not aware of any measurements being taken – the measurements only use the public interface of `Algorithm` to extract information. This means that there is a clean separation of measurement and algorithm logic, which makes both code sets easier to maintain and reuse. Measurements are added to experiments using the CILib simulator discussed in section 4.1.6. A sampling rate can be set which allows measurements to be taken at specified frequencies.

4.1.5 Type

The `Type` package in CILib serves as a common base for all data representations in any CI implementation. The reasons why `Type` is used include:

- **Conformity and standards:** Flexibility and reusability are drastically affected if developers can use any data types they wish. This would lead to interfaces not matching up without conversions via adapters, making interoperability difficult and time consuming. In an environment with n interfaces, there are $\frac{n(n-1)}{2}$ possible symmetrical connections and $n(n-1)$ possible asymmetrical connections[4]. By having a single data representation, the number of data format conversions between systems is drastically reduced.

- **Concise data:** Conversions between different representations may introduce errors as complex data structures could be converted incorrectly, leading to invalid experimental results. Development time and complexity would increase, as logic to convert to and from other representations needs to be included in the application.

- **Operational efficiency:** The native Java data types such as Double and Integer have extra functionality built-in (such as serialisability) that degrades their runtime performance. By developing a `Type` package that is designed to support only the needed functionality, performance at execution time is increased.

- **Efficient development:** The `Type` package allows developers to easily formulate and expand complex data structures in their applications. By using the *composite*

[4] [29] states that the number of edges in a graph is the degree of the graph divided by 2.

pattern, even complex data representations can be treated as `Type` objects. This allows complex structures to be built once, and reused by any application that supports `Type` objects.

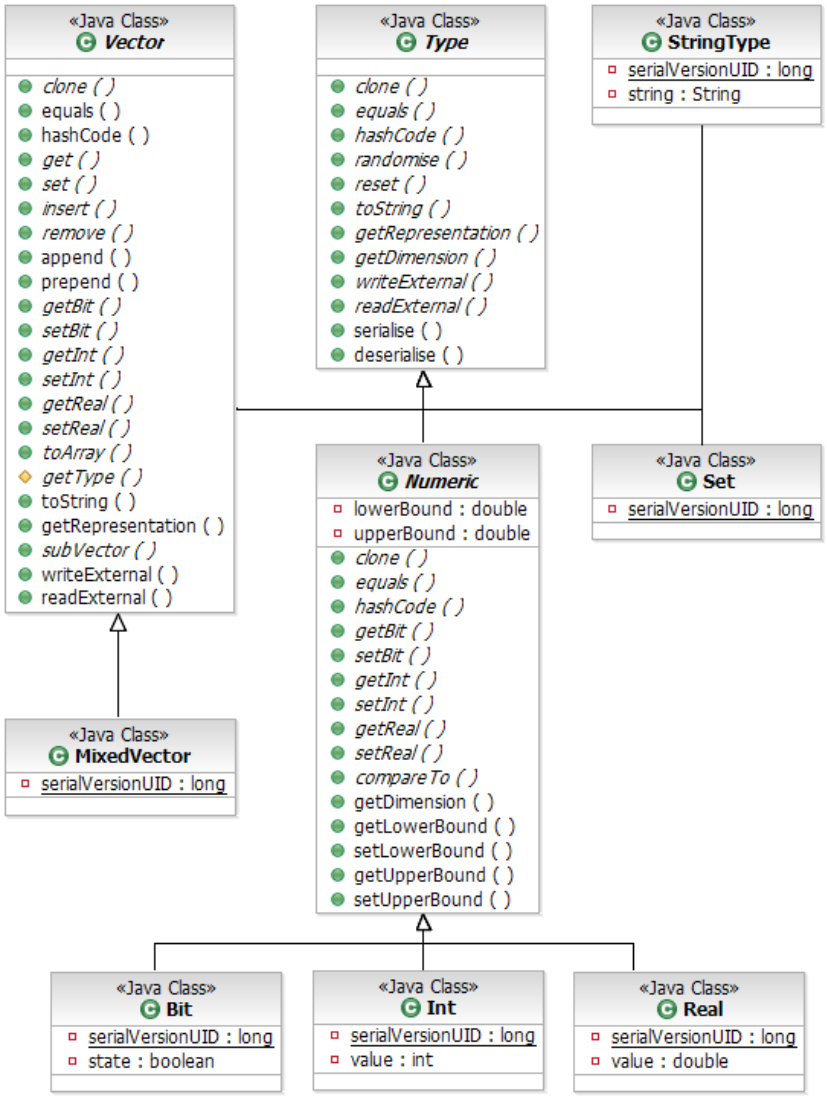


Figure 4.5: The CILib Type hierarchy

At the heart of the `Type` package is the `Type` abstract class as can be seen in figure 4.5. This class provides the base functionality required by all data types in CILib. This functionality is not specific to any particular data type and includes actions such as

more resource efficient `serialise()` and `deserialise()` method implementations. `Type` also specifies the basic interface to common actions such as `toString()` and `clone()` (conforming to the *prototype* pattern) that subclasses of `Type` need to implement.

All other data types are subclasses of `Type`. The `Numeric` abstract class adds all required functionality to represent numeric data types. This includes upper and lower bounds and `compareTo()` method that facilitates comparison. `Numeric` also defines methods to simplify the interpretation of one type as any other type in the `Type` hierarchy (i.e. custom built-in type casting). This provides native functionality to interpret a particular type as another type, but in a specific fashion as defined in the particular method. As an example, the `getReal()` method returns a `double`, the `getInt()` method returns an `int` and the `getBit()` method returns a `boolean`. This makes it very easy to work with `Numeric` data types as first order Java types and also makes conversion between `Numeric` types easy. There are three main `Numeric` subclasses, namely `Real`, `Int` and `Bit`. Using these methods, a `Bit` can easily be interpreted as a boolean value, or a numerical value of 1.0.

The `Vector` abstract class utilises the *composite* design pattern to allow the construction of any complex tree-like data structure via recursive composition. `Vector` is abstract and merely provides the interface for subclasses to provide the functionality. `MixedVector` extends `Vector` and incorporates an `ArrayList<Type>` object. As the array elements are all instances of `Type`, recursion is possible as `MixedVector` is a subclass of `Type` as well.

Two more data types that are implemented are `StringType` and `Set`. `StringType` is effectively a wrapper for the Java `String` type that allows the `Type` package to use `String` objects as components. This also enables the creation of complex non-numeric data structures by using a combination of `StringType` and `MixedVector`. A good example is when modeling DNA, which typically consists of long arrays of code strings (A, C, T and G). By using `MixedVector` and `StringType` together it becomes easy to build such long strings. The `Set` type is used to hold an unordered collection of `Type` objects.

The `Type` system is fully extensible if very complex data types need to be represented. As a further example, the modeling of biological processes such as RNA folding requires more information than what an array of strings could hold. This includes in-

formation such as which pairs form Hydrogen bonds, Anticodon locations, loops, among other information. RNA structures is just one example of complex data types that CI algorithms may need to manipulate. For more info on RNA structures and other biological processes, see [105]. By extending the base `Type` class and reusing elements such as `StringType` and `MixedVector` along with the extra information, it becomes easy to create a type called `RNA_Type` class that any application can reuse.

4.1.6 Simulator

Figure 4.6: Simulation factories

Once all the required classes for a given CI algorithm have been implemented in

CILib, one needs to combine them together in order to execute – on their own these classes are nothing more than components. A trivial way to achieve this is to create an executable class (i.e. a Java class with a `main()` method), instantiate all the objects correctly and run it. While this approach works fine for one simulation, it becomes untidy and tedious to maintain once many developers start using the system and dozens of simulations have been defined and configured. All simulation logic is then encapsulated in this component, which handles detailed Java object creation for algorithms and problems, handling simulation repetitions, as well as the configuration and capturing of measurement data.

Typically one would find that simulation logic above and beyond the algorithm's logic will be duplicated every time a new simulation needs to be created. This includes details such as running the algorithm 30 times for statistical analysis. Also, as `Algorithm` is designed to be run in parallel Java threads, the logic for this needs to be implemented for each simulation every time. For large CI problems, logic to run simulations in a multi-computer cluster environment is also needed. Redoing this implementation for every simulation drastically increases the likelihood of errors as well as increasing development time and effort.

For these reasons, CILib has a mechanism for configuring and executing simulations using a framework to handle all simulation related logic. It utilises XML object factories which allow `Algorithm`, `Problem`, `StoppingCondition` and `Measurement` classes to be constructed and configured by means of an XML document. The XML structure defines exactly what objects need to be instantiated as well as all the members for each object. As XML allows tree structures to be defined, this mechanism allows the 'objects inside object' to be defined as well.

As can be seen in figure 4.6, the `XMLObjectFactory` subclasses are connected with the corresponding Java interfaces to enable the creation of the relevant component classes. However, the objects returned by the factories need to be configured with the correct parameters in order to be executable. This is done via the XML document. Each class must provide a default constructor that sets all properties in the object to sensible defaults. All publicly accessible properties can be set by using the correct tags in the XML document. Below is an outline of what such an XML document looks like:

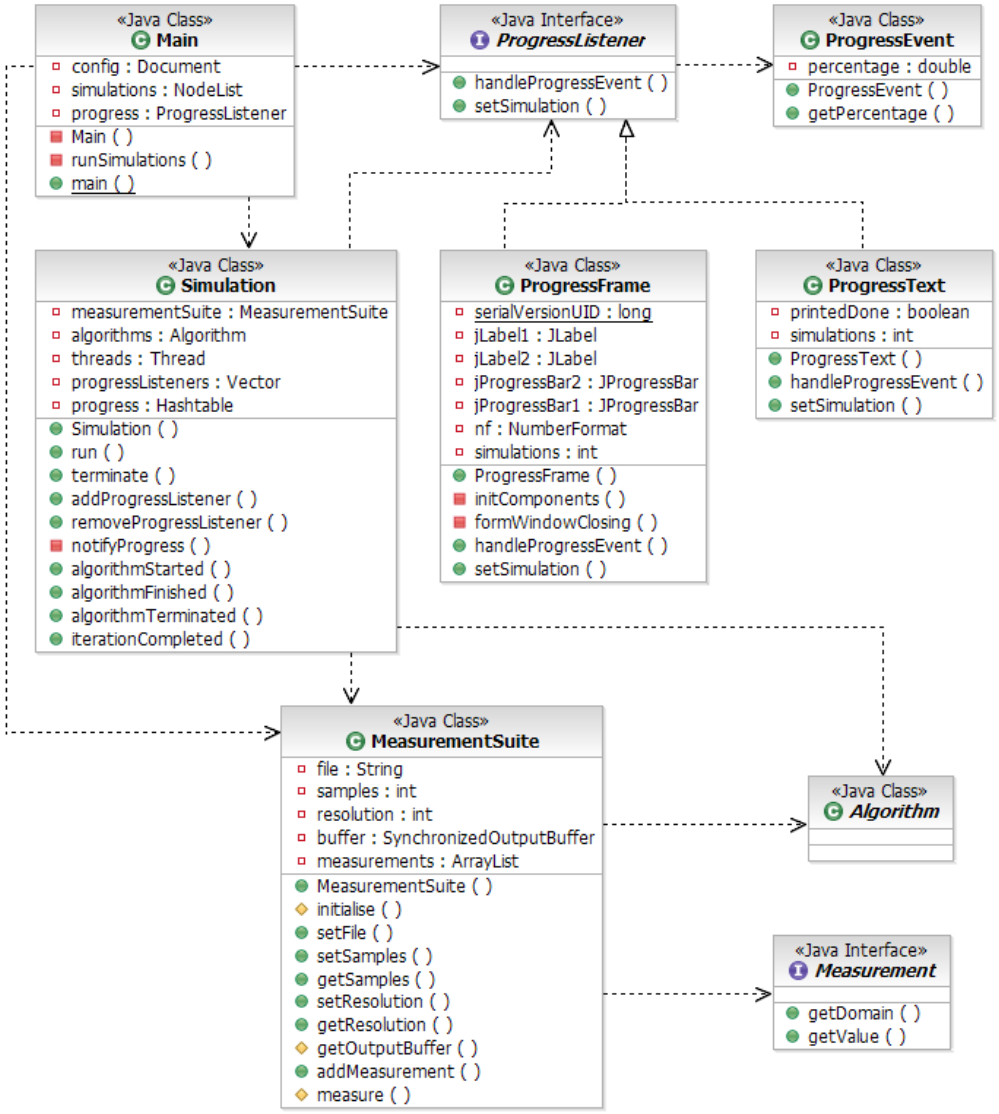


Figure 4.7: The CILib simulation component

```
<simulator>
  <algorithms>
    <algorithm id="anAlgorithm" class="...">
      <addStoppingCondition class="stoppingcondition.MaximumIterations" iterations="1000"/>
            (set all other algorithm properties here...)
    </algorithm>
  </algorithms>
  <problems>
    <problem id="aProblem" class="...">
```

```
            (set all problem properties here...)
    </problem>
  </problems>
  <measurements id="fitness" ..." resolution="10" samples="1">
    <addMeasurement class="..."/>
            (add more measurements here...)
  </measurements>
  <simulations>
    <simulation>
      <algorithm idref="anAlgorithm"/>
      <problem idref="aProblem"/>
      <measurements idref="fitness" file="c:/cilibOutput/fitness.txt"/>
    </simulation>
  </simulations>
</simulator>
```

In the above XML document it is clear that a *problem* is specified as well as an *algorithm*. *Measurements* are also defined using the appropriate tags, and *stopping conditions* are defined and added to the algorithm. These three components are then used to form a *simulation*. Multiple problems, algorithms and simulations can be defined in one XML document, each representing a different experiment run. This makes it easy to define for example an XML document that defines an algorithm, 15 problem cases, a set of measurements and 15 different simulations that solve each problem case using the same algorithm and measurements. The class layout of the simulation component is shown in figure 4.7.

4.2 Summary

CILib is a framework that allows developers to develop new CI algorithms quickly and efficiently. *Flexibility*, *reusability* and clear *separation* between components are maximised through the use of *design patterns*. *Reliability* is also ensured as the framework is open source and thus having many people *collaborate* to ensure that the framework is well designed and error free.

CILib defines an *algorithm* component that works on *problems*. By using an XML-based *simulation* engine to define and configure experiments, it becomes possible to use pre-defined *stopping conditions* and *measurements* to set up research experiments quickly, efficiently and robustly. Rework is also minimised as developers don't need

to rewrite execution functionality. A common *type* system is also used to further aid interoperability, robustness and efficiency.

The next chapter shows how CILib can be reused and extended to allow any neural network algorithm to be implemented in a generic fashion.

Chapter 5

Neural Network Framework in CILib

> *Computer Science is no more about computers than astronomy is about telescopes.*
>
> — *Edsger W. Dijkstra*

A conceptual breakdown of neural networks was presented in chapter 2. In this chapter, the aforementioned breakdown serves as the main requirement for the design and implementation of a *generic neural network framework* in CILib. The aim of this framework is to provide a base for developers to quickly and efficiently build implementations of neural network algorithms by reusing CILib functionality.

Many levels of reuse exist in this framework as can be seen in figure 5.1. The Java programming language serves as the base for CILib. The use of Java offers key benefits such as easy multi-threading, Java's famous 'write once, run anywhere' characteristic, convenient memory management, introspection capability for realtime dynamic class discovery, easy instantiation and invocation, and easy enabling of grid computing on an existing code base. A very good source to learn about Java is [35], which also states that Java is definitely not a bad language choice from a performance point of view – Java's performance used to be poor in earlier releases of the language compared to languages such as C++, but this is no longer the case in recent versions. Refer to [85]

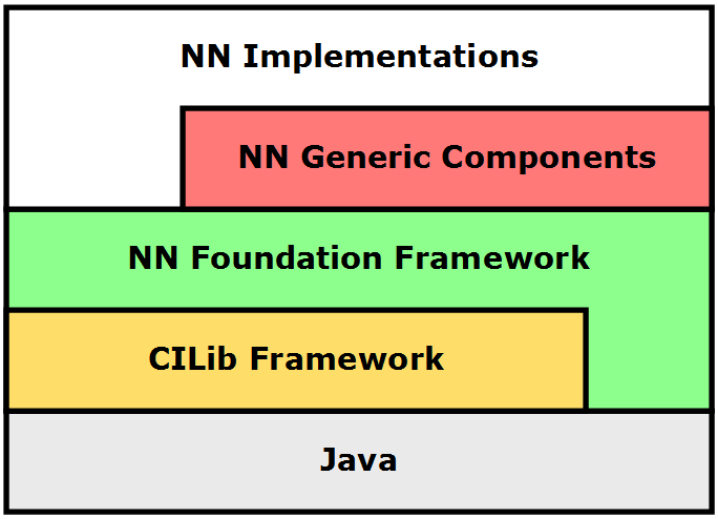


Figure 5.1: Architecture of the generic neural network framework in CILib

for a comparison of various Java compilers and other languages that shows concretely how Java provides adequate performance (and even outperforms GNU C++ by a fair margin in many cases.). The advantages of using Java to allow for quick and easy reuse of components far outweigh the potential minor loss of performance, especially when considering the fact that hardware is improving continuously whilst also becoming cheaper. One of the main goals that the University of Pretoria has for CILib is the ability to run CI simulations on a grid of more than a 1000 computers. CILib makes this goal very easy to achieve.

The *CILib framework*, as discussed in chapter 4, is built using Java and XML[1] technology as summarised in figure 5.1. CILib provides the base structure for all CI algorithms and lays the foundation for easy integration and reuse of CI components.

The *neural network foundation framework* is built on top of CILib and Java as depicted in figure 5.1. The foundation framework consists mainly of Java interfaces, abstract classes and a small number of predefined classes. These components serve as a *specification* for any neural network implementation to be built using CILib. The main characteristics of the foundation framework are:

- **Ease of use:** Enables the easy creation of custom neural network implementations

[1]See http://www.w3c.org/xml for an overview of the XML specification.

in CILib (that do not necessarily use the generic components discussed later in section 5.2).

- **Flexibility:** Facilitates the reuse of existing components to easily create new neural network algorithms.
- **Extensibility and reuse:** Allows existing neural network implementations to be extended easily and concisely, as the foundation framework addresses the base neural network relationships as defined in chapter 2.
- **Easy access:** Enables any application, whether internal or external to CILib, to access neural network components seamlessly. This is possible because the foundation framework supplies the required CILib interfaces to all neural network implementations by default.
- **Generic foundation:** Provides the base specification for the implementation of generic neural network components on top of the foundation framework, as discussed in section 5.2 to allow easy reuse.

The *neural network generic components* layer in figure 5.1 allows developers to develop reusable components that can be reconfigured and extended to represent different types of neural network models. A good example is a component that is capable of representing arbitrary topologies of neuron and weight connections. Generic components allow developers to quickly assemble new neural network implementations without having to write everything from scratch.

Finally, the top layer in figure 5.1 represents *neural network implementations*. This layer is required as the components of the neural network framework, as well as generic components, are not executable in their own right – the various elements need to be composed together to form any specific neural network model. In CILib, the main way of accomplishing this is the simulator component as discussed in section 4.1.6. The outline of the desired neural network implementation is described using XML notation and the application is built dynamically at runtime. Another way of defining an application is to define a Java class that instantiates all the required CILib objects and execute them.

This approach is more direct and of course voids all the advantages of using the native CILib simulator.

Neural network implementations can be built using a mixture of both custom foundation framework implementations and generic components. Neural network implementations that are based solely on the neural network foundation framework are intended to serve as 'stand-alone' custom neural network implementations. These applications leverage the foundation framework and are merely custom written neural network implementations that conform to the CILib specification as laid down by the foundation framework. One may find that a specific application in the foundation framework allows for the reconfiguration of parameters such as to reconfigure from two layers with five neurons each (i.e. a 5-5 architecture) to three layers with a 3-10-1 architecture. This is not considered to be a generic component, but a configurable custom application – a generic topology component would be able to, for example, specify any neuron topology that can be represented as a graph.

Applications can be implemented much faster through the reuse of generic components. The main goal of this approach is to be able to define new neural network topologies quickly and efficiently by using existing and well tested components. These components can be substituted for other more specialised components at later stages of development. An application that is built this way would consist of a mixture of various generic components and custom written components. Detailed examples of how this works are presented later in this chapter.

The rest of this chapter is divided into two parts, namely a discussion of the foundation framework and a discussion of generic neural network components. A breakdown is given on how the foundation framework extends CILib to facilitate neural network algorithms and implementations. After that, the use of generic components is discussed and how they compliment CILib to allow easy creation of neural network implementations using pre-built components. Throughout the discussion, reference is made to design patterns in chapter 3, how they aid in the solution and what advantages and consequences they provide.

5.1 The Foundation Framework

This section provides more detail on how the foundation framework stipulates the manner in which neural network implementations can be built using CILib.

As discussed in chapter 4, CILib is a framework that assists developers to write CI algorithms by stipulating the base infrastructure and relationships between components that all CI implementations have in common. CILib provides the functionality that every CI implementation needs, but that developers do not want to worry about every time they write new applications. A good summary of CILib is:

> CILib is a *framework* that allows researchers to design, implement and run CI *simulations* that consist of *algorithms* that work on *problems*. Simulations run until the algorithm completes as stipulated by *stopping conditions*. Simulations also allow *measurements* to be taken periodically.

However, CILib is a *framework* – it provides the base interfaces, abstract classes and pre-defined classes to allow algorithms, problems, measures, data types and stopping conditions to be defined. For each field in CI such as swarm intelligence, evolutionary computation, neural networks and fuzzy systems, the CILib framework needs to be extended to provide the required environment. The purpose of the foundation framework is to lay the groundwork for developers to implement neural network components that in turn can be used in neural network implementations. The foundation framework stipulates the base relationships between the building blocks of neural networks as found in chapter 2.

These include:

1. A *topology* component that serves as a container for neurons and weight connections. This component is responsible for providing an environment to represent specific neural network architectures.

2. A *data* component that encapsulates all aspects related to the use and manipulation of data sets and patterns.

3. A *training* component that is responsible for all neural learning that is considered

to be specific to the neural network system – other CI algorithms such as PSO access the system differently as discussed in section 6.2.

4. A *mediation* component is present to orchestrate the logical order of actions between the topology, data and training components for each type of neural network in a loosely coupled fashion. It also acts as a facade object to sub-systems, allowing access to lower level components of a neural network implementation via a single interface.

More detail about each component is given in the following sections. An example of a simple neural network implementation that is implemented using the foundation framework is also shown in section 5.1.5.

5.1.1 The Topology Component

As discussed in chapter 2, the *topology* of a neural network model consists of one or more layers of neurons and the weight connections that link these neurons together. Specifically, the neuron needs to support functions such as handling input weight connections, net input signals and activation functions (collectively called the transfer function), scaling and limiting of output values, and competition as discussed in section 2.1.1. The weight connection framework, as covered in section 2.1.2, must support multiple layers of neurons (either input, hidden or output layers), where inter-, intra-, supra- and self-connections are possible. Symmetrical and asymmetrical connections must also be taken into account, as well as the possibility of having higher-order connections. Depending on the type of network topology, some or all of these different concepts need be implemented in different manners to form specific neural network topology implementations, such as a Hopfield network, a SOM, a FFNN, a RBFN, an ensemble network, or a modular network. There are literally hundreds of known neural network topologies that need to be supported.

Furthermore, applications that utilise the topology component (such as the CILib simulator or a custom Java application) as well as other CILib components (such as a learning component) must have easy access to any relevant information of the topology without duplicating the structure. All aspects of the network topology should thus be

encapsulated in the topology component, yet be available to other components. Any changes to details inside the topology also has to be transparent to outside components. For example, if all the hidden layer neurons in a FFNN have their sigmoid activation functions changed to hyperbolic tangent functions (as discussed in section 2.1.1), this change should not require code changes in the mediation component or the data component. Of course, if the new configuration is invalid for some reason, a suitable exception must be thrown to highlight this. As a further expansion on the example above, if a particular learning component uses adaptive activation functions [27], the component may not work with hyperbolic tangent activation functions.

The `NeuralNetworkTopology` interface, as illustrated in figure 5.2, represents the concept of a topology in the foundation framework and uses the *strategy* design pattern to implement all the requirements listed above. As `NeuralNetworkTopology` is a Java interface, any class that implements `NeuralNetworkTopology` correctly can be regarded as a neural network topology. The implementing class may use any datastructure to represent the neurons and weights, based on the requirements of the required neural network model.

By implementing neural network topologies using the *strategy* design pattern, a number of architectural benefits are obtained:

- **Encapsulation:** Neural network topologies are encapsulated, meaning all the logic related to a specific implementation is contained in one place. This includes the types of neurons, weight interconnection schemes, the types of weights, and neuron functions. It also encapsulates the implementation details, such as the datastructures used to implement the topology. This cleanly separates the logic as well as the implementation from the rest of the application. It also becomes easier to define ensembles and MNN topologies by reusing existing implementations.

- **Client simplicity:** Components that use the `NeuralNetworkTopology` need not know the intricate details of how it is implemented – the client merely has to know how to call and use the component via its interface. Classes that implement `NeuralNetworkTopology` may provide more methods to further aid this.

- **Client extensibility:** Clients can easily exchange one topology for another one.

Consider an example where there is more than one implementation of the same type of topology, say a SOM – one implementation might be implemented using an extremely accurate (but slow) data type, while another implementation might only be accurate to the eighth decimal point, but executes fast. This flexibility allows developers to easily provide different implementations of the same topology model (perhaps using different programming models or data structures). Note that this may not make sense in all cases, such as exchanging a SOM topology for a Hopfield network in the same learning algorithm.

- **Dynamic variation:** If any future neural network implementation would ever require that a topology object be exchanged for another, or that a topology gets recreated *dynamically at runtime*, the strategy pattern makes it possible. This might, for instance, be useful in modular network environments where more network modules can be added dynamically, as is done with NNTrees in [125] (also see section 2.4.4).

The `NeuralNetworkTopology` interface specifies three methods that an implementing class should implement, namely `evaluate()`, `getWeights()` and `setWeights()`.

The `evaluate()` method must provide a mechanism to evaluate any given input pattern to produce a network output value. This mechanism must also be able to handle network types that do not produce explicit output values, for instance a SOM. The `evaluate()` method takes an `NNPattern` object as parameter and returns the output of the neural network (assuming the network model has output, otherwise a null value should be returned), where `NNPattern` is an interface that represents an input pattern. See section 5.1.2 for more information on the `NNPattern` interface.

As a topology is a container of neurons and weights, the base interface specifies the `getWeights()` and `setWeights()` methods that extract and insert weight values respectively, which allows any external component to access these weights. This can be used to train neural networks using external components such as PSO or an EA. The `getWeights()` and `setWeights()` methods also allow the network state to be saved to file and restored at a later stage.

It is important to note that `NeuralNetworkTopology` is only an interface and the class that implements `NeuralNetworkTopology` is responsible for providing the exact

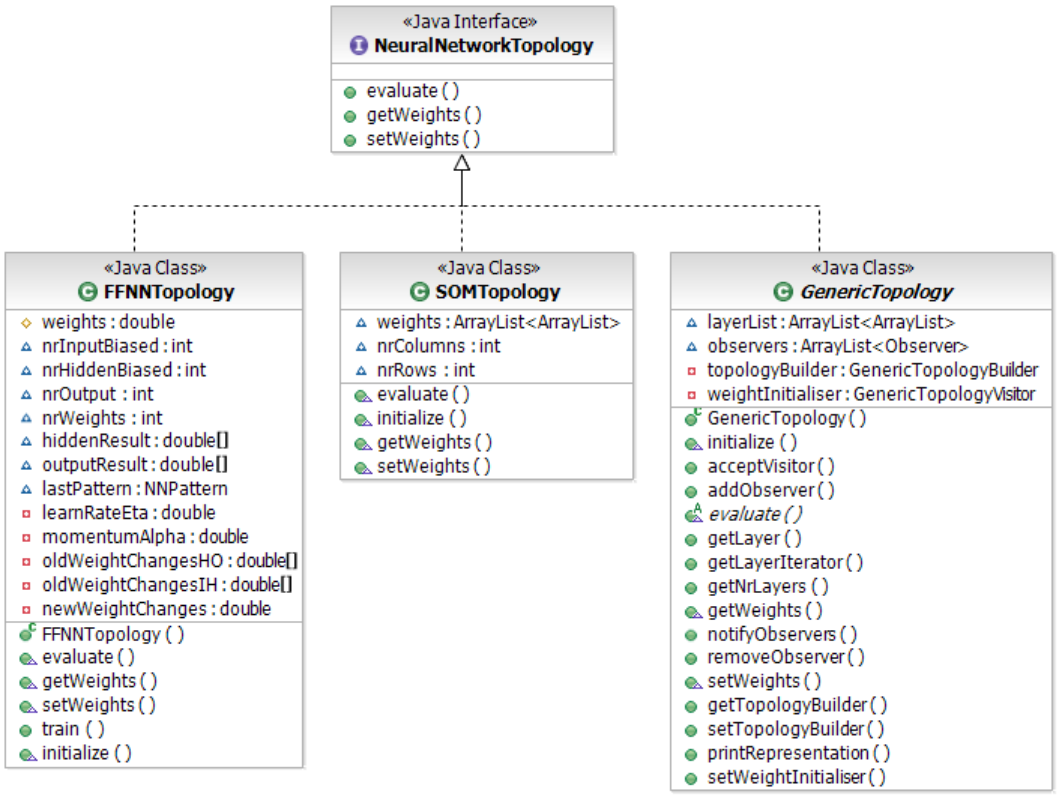


Figure 5.2: The `NeuralNetworkTopology` implementation

functionality of the specific topology it represents. Section 5.2.1 discusses the design and implementation of a generic topology implementation that has the capability of representing any topology that can be expressed as a graph.

5.1.2 The Data Component

Neural networks typically accept data in the form of input *patterns* that are presented to the network to generate some form of output (either an output pattern or a state change). As seen in section 2.2, a *pattern* can be defined as a single unit of data that is presented to a neural network. A data set D can be regarded as a set of patterns such that $D = \{d_p = (\mathbf{z}_p, \mathbf{t}_p) | p = 1, \cdots, P\}$ where $\mathbf{z}_p$ is an input vector (with dimension I) and $\mathbf{t}_p$ is its associated target vector (with dimension K) for pattern d_p. Depending on the network type, the target value is not always needed. Supervised networks typically accept

patterns with input and target values while unsupervised networks typically require input values only. As seen in section 2.1.2, the type of data in a pattern may vary, including types such as real, binary, complex, and discrete values. As data and weight values are merely inputs to net input signal and activation functions, any type that is supported by these functions may be present in data patterns (as well as weight values).

Data Patterns

A single pattern instance is represented by the `NNPattern` interface (see figure 5.3) which stipulates all the operations needed to access a pattern's information. These operations are the ability to enter or extract both input or target data values, cloning the pattern, and other housekeeping tasks. The reason why `NNPattern` is a Java interface is to allow multiple types of patterns to be defined. This approach allows researchers to easily create new types of pattern implementations, for instance the definition of a pattern with certain input and output values that are static, while other values are calculated dynamically. For example, consider a neural network in a financial application that accepts the Euro/Dollar exchange rate as one of the inputs. A subclass of `NNPattern` can look up the latest exchange rate, perhaps every five minutes, and update the relevant input value with the updated exchange rate. The complexity of this behaviour is completely masked from the rest of the neural network implementation.

The `NNPattern` interface encapsulates all aspects of a pattern, which means that the rest of CILib does not need to be aware of the complexity of the pattern's target – it may be static or calculated using a specific algorithm. `StandardPattern` in figure 5.3 implements `NNPattern` and supplies both input and target member variables of type `MixedVector` (Recall that `MixedVector` extends `Type` as discussed in section 4.1.5). Target values may be null, so `StandardPattern` may be used in cases where there is no target value (such as unsupervised learning). `MixedVector` allows any data type such as real, binary, complex, and discrete values among others to be used in `StandardPattern` instances, which means that `StandardPattern` should satisfy most neural network requirements in general.

One challenge with using `MixedVector` as data type for the input and output values of `NNPattern` is the construction of pattern copies. Examples of components that may

need to make copies of patterns are numerous. The learning component may need to make copies of a pattern to avoid accidentally changing the original pattern values.[2] A selective learning component (see section 2.2.3) may have to make copies of patterns for each new training subset as opposed to moving references around (or perhaps a history list of past training sets needs to be kept). A particular data component implementation may need to make copies of patterns to add noise to the copies as stipulated in section 2.2.1.5 and so doing increase the number of available patterns. These are just some examples of why patterns may need to be copied.

If, in the context of making copies of patterns, the constructor of the subclass of `NNPattern` is used along with the `setInput()` and `setOutput()` methods, the exact instances of `MixedVector` need to be provided to `setInput()` and `setOutput()` everytime a pattern needs to be copied. The code for instantiating the copy then also resides in the component that wants to make the copy. This code is also responsible for performing the copy process 'correctly,' as a deep copy needs to be made of the `MixedVector` objects inside the pattern object (as opposed to an object reference copy). Furthermore, as `NNPattern` is an interface, any implementing class may have any number of other member variables that also need to be copied correctly. Leaving the logic to make copies of patterns in external component code will overly complicate these components, as each type of pattern object needs to be catered for. Addition of new types of patterns will also require a code change in all components that have to make copies of patterns (to cater for the new type of `NNPattern` implementation).

The *prototype* design pattern, discussed in section 3.1.2, is used to create copies of `NNPattern` objects. The advantage of using the *prototype* pattern is that other CILib components do not need to know what type of pattern they are creating or how to make copies of it, merely that the pattern is a subclass of `NNPattern`. To copy a pattern, a component merely has to call the existing pattern object's `clone()` method, which returns a full copy of the pattern. The *prototype* pattern greatly simplifies the CILib design and reduces the need to modify existing component code when new types of `NNPattern` subclasses are added. Types are also assigned dynamically, meaning that the

[2]Some developers may even implement the `getInput()` and `getOutput()` methods of their respective classes that implement `NNPattern` to always make a copy upon request.

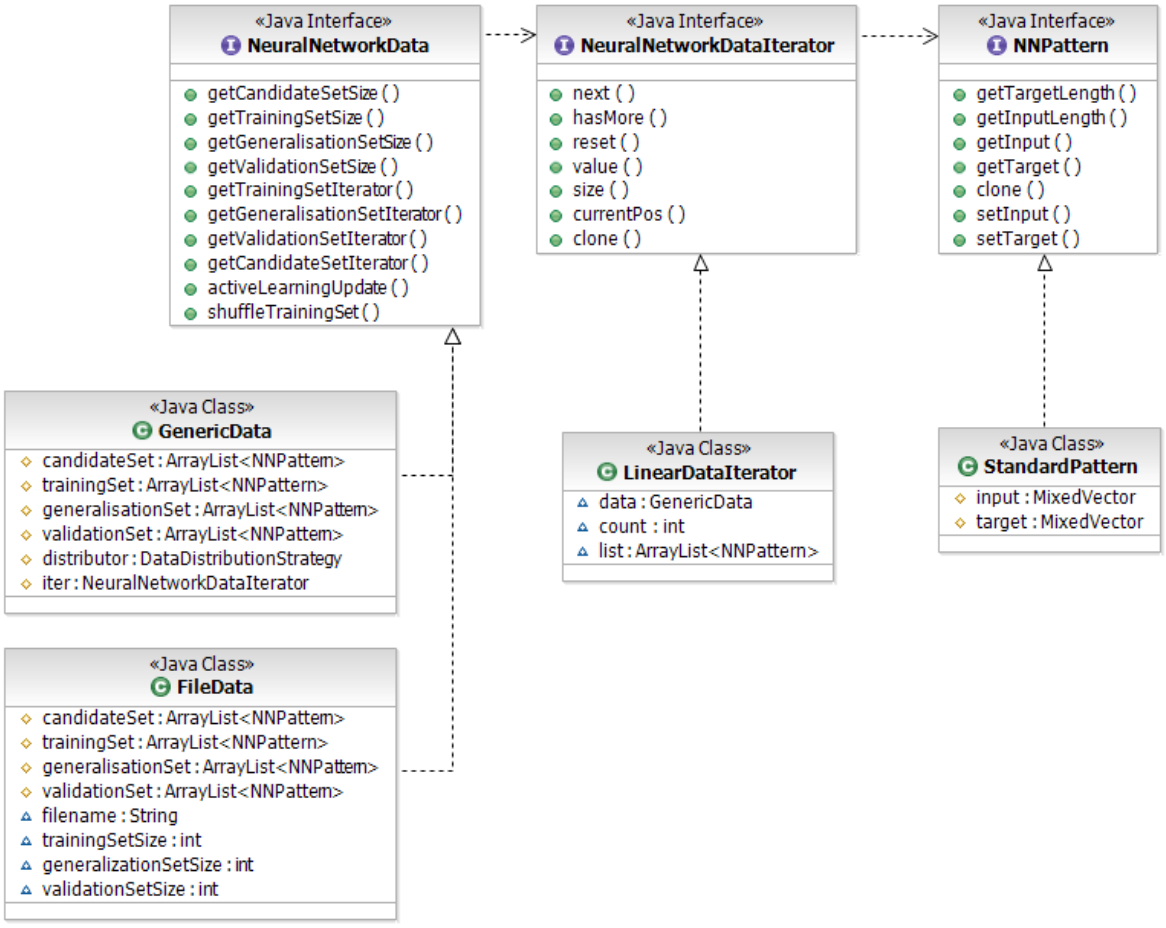


Figure 5.3: Architecture of the `NeuralNetworkData` component

specific subclass of `NNPattern` may be decided based on a runtime condition.

Pattern Organisation and Data Sets

Classes that implement `NNPattern` represent single instances of data patterns. A way is needed to organise patterns into collections of data sets containing patterns. With reference to section 2.2.1.6, patterns need to be organised into the data sets D_T, D_G, D_V and D_C. This organisation can occur using any distribution scheme as discussed in section 2.2.1.6. Pattern data also needs to be read in from data sources. Sources may include flat text files, XML files, relational databases, data from a network location such as a web service, data streams from other applications, dynamically generated function

output, or any other source of data.

The *order* in which patterns are presented to the network also needs to be defined. Various schemes exist (see section 2.2.1.6), such as selective presentation [83], increased complexity training [10], and random ordering. Furthermore, active learning approaches also need to be supported, which use complex algorithms to determine which patterns should be included in D_T (as discussed in section 2.2.3).

Components that use data sets and patterns should be shielded from the implementation complexity of data set manipulation, reading in data from sources, organising and distributing patterns into data sets, the algorithmic details of active learning, pattern ordering schemes and any other tasks related to the management of data. A component that wishes to use data should merely be able to obtain patterns upon request.

The `NeuralNetworkData` interface, illustrated in figure 5.3, lists the methods needed to implement a class that can provide patterns to the neural network system and act as a container for data sets of patterns. In particular, this class will need to read data from the source and distribute patterns to the D_T, D_G, D_V and D_C data sets. The `NeuralNetworkData` interface defines four access methods for the four data sets D_T, D_G, D_V and D_C. Developers are free to implement the details of the data component in any way, including how to perform reading patterns from data sources, distributing patterns among data sets, pattern ordering algorithms, and active learning algorithms (if any). The only requirement is that the return type of the four access methods is `ArrayList<NNPattern>`, which represents a standard way to represent a data set as a list of `NNPattern` objects.

Client components need to be shielded from the complexity of the `NeuralNetworkData` implementation. Probably the most often used functionality of the data component is the iteration over patterns in a data set. Client components merely want to obtain the next pattern upon request, without having to be concerned with the details of the chosen data structures and other implementation details. Clients may also want to iterate over more than one data set simultaneously (such as over D_T and D_V), and may even request multiple iteration counters on the same data set. The *iterator* design pattern from section 3.2.2 is utilised to allow client components to traverse data sets. The `NeuralNetworkData` interface provides methods to obtain a `NeuralNetworkDataIterator` for each of the data

sets D_T, D_G, D_V and D_C.

The advantages of using the *iterator* pattern on neural network data sets are:

- **Variation:** Different types of traversals can be implemented for any particular data set, such as a linear traversal, random traversal, or any other specific order based on a heuristic. The complexity of the traversal logic is hidden from the client, as well as the class that implements the `NeuralNetworkData` interface.
- **Multiplicity:** Many traversals may be active at any given moment as the state of each traversal is kept in the iterator. This is useful if many components of a neural network want to access data independently.
- **Encapsulation and abstraction:** Other components do not need to know how the data source is implemented to be able to traverse it. This means that if a data source implementation changes from, for instance, basic arrays to another data structure, the code of the other components does not need to be modified.
- **Conformity:** Other components can use a single interface to obtain a traversal over a specific data set. This interface is well-defined by the `NeuralNetworkData` interface. This consistency allows `NeuralNetworkData` to be implemented and used as a *strategy* very easily.

Components may use the functionality provided by `NeuralNetworkData` without knowing any implementation details. Many different implementations may be specified, with little or no change to other components if a different data component needs to be used. This means, for example, that whether fixed set learning or active learning is used, components such as the topology or learning components that use the data component should be completely unaffected. The *strategy* design pattern is used to encapsulate all aspects of a data source. Using the *strategy* pattern enables data components to be presented to other components in a consistent way. Components can also utilise different implementations of `NeuralNetworkData` with ease.

The `NeuralNetworkData` interface also directly supports active learning as discussed in section 2.2.3 via the `activeLearningUpdate()` method. Classes that implement

`NeuralNetworkData` have the option to provide the necessary functionality to facilitate active learning. The relevant mediator component is responsible for determining when to invoke this method, based on which active learning algorithm is used. Section 6.1 provides a more detailed look at the implementations of fixed set learning and two active learning algorithms as subclasses of `NeuralNetworkData`.

The final artifact of note is the `shuffleTrainingSet()` method. As the shuffling of training data sets is a very common action in most neural network models, the `shuffleTrainingSet()` method is included in the `NeuralNetworkData` interface for easier access by client components without the need for type casting. Section 2.2.1.6 lists a number of pattern ordering methods used to govern the order of training set patterns. These include the increased complexity training approach [10], selective presentation [83], ordering training patterns based on a distance metric such as Hamming or Euclidean distance [11], and random pattern ordering, among others. The `shuffleTrainingSet()` method allows subclasses of `NeuralNetworkData` to implement pattern ordering, using the *strategy* pattern, to effectively provide a new implementation for each type of ordering scheme. An alternative approach is to shift the logic for pattern ordering to an iterator. The iterator always works on the same ordered list, but performs ordering 'on the fly.' Performing ordering in an iterator effectively enables these ordering schemes to be reused across multiple data component implementations (at the expense of runtime performance as the iterator needs to continuously perform ordering calculations).

In section 5.2.2, a generic implementation of `NeuralNetworkData` is discussed.

5.1.3 The Learning Component

As discussed in section 2.3, the main purpose of neural learning is to train a neural network in such a way that the neural network is capable of approximating a defined problem to an acceptable level of accuracy. This entails changing the network topology such as changing one or more of the network's weights, but also other aspects such as changing neuron parameters (i.e. adaptive activation functions), pruning the network by removing weights and/or neurons, or growing the network by adding weights and/or neurons (as discussed in section 2.3.5).

In trying to implement neural learning in CILib, it soon becomes apparent that there

are two separate learning cases:

1. The desired optimisation algorithm is a stand-alone implementation in CILib, such as PSO or an EA.

2. The optimisation algorithm is a neural network specific algorithm and is *not* a stand-alone implementation in CILib. Examples include random search, non-generic gradient descent[3], as well as the training of a Hopfield network or a SOM.

This observation means that both approaches need to be supported – it is unreasonable to assume that *every* learning approach for *every possible* neural network model can be implemented as a CILib `Algorithm` subclass. Both approaches are discussed below.

Stand-alone CILib Algorithms

To support neural learning in cases where the desired optimisation technique is implemented as a stand-alone CILib component, it is necessary for the particular learning component to interact with the neural network system. This needs to be done in a defined manner, as there is no guarantee that the neural network system and the external algorithm both use the same data types. In particular, the external component will have to be able to map any information it needs from the neural network system to its own interface and vice versa, as well as handle the conversion of different data types if it is the case. This logic should ideally not be located in either the neural network or the external component, as it limits flexibility and adds unnecessary complexity to both components.

The *adapter* design pattern discussed in section 3.3.2 can be used to good effect to provide the glue between the neural network system and any external CILib component. An adapter has to be written for each *type* of neural learning goal, as the approach for neural network models such as SOM, Hopfield networks, feedforward networks or adaptive resonance theory [43] will vary. Different types of adapters have to be defined, as each of these networks has its own learning goal. For example, a possible goal for feedforward neural network training is to minimise the MSE objective function. This

[3]It is, however, possible to write a stand-alone implementation for gradient descent that extends the `Algorithm` class.

goal makes no sense for a SOM, and a different adapter will have to be written if an external learning component needs to train a SOM. A more detailed discussion on the use of PSO or an EA to train a feedforward neural network using the MSE objective function and an adapter is given in section 6.2.

The use of stand-alone algorithms entails that the neural network system is merely used to house the network topology (i.e. neuron layers and weight connections and how to compute pattern output), as well as the data source (i.e. the collection of patterns, how to read the patterns in from the data source, how to iterate over data sets). The manner in which the topology and data components are orchestrated to work together is handled by the mediation component by using the *mediator* design pattern, which is discussed in more detail in section 5.1.4.

Neural Network Specific Algorithms

There are cases where the learning algorithm is specific to neural networks and is thus not implemented as a generic optimisation algorithm in CILib (i.e. is not a subclass of `Algorithm`). Consider the training of a Hopfield network (see section 2.4.2) or the batch training of a SOM (see algorithm 2.6) as examples. Yet the CILib specification stipulates that an `Algorithm` class *must* be present, as it is the main component that is invoked during runtime. At first glance, the implementation of neural network specific algorithms seems straightforward – merely implement each learning algorithm as a new `Algorithm` subclass. This approach will technically work, but may prove inflexible in terms of reuse and composition of existing components.

Consider an example where several learning algorithms to train a FFNN are already implemented as subclasses of `Algorithm`. If a new architecture selection algorithm for a FFNN needs to be implemented that can be used alongside all the existing FFNN learning approaches, an immediate problem surfaces: how can the architecture selection algorithm be used in conjunction with the existing learning algorithms? An intuitive method is to write a new `Algorithm` implementation for *each* of the existing algorithms and have it include the architecture selection functionality. This is not a very clean approach at all, as copies of *all* the existing `Algorithm` implementations need to be made and modified to support the new functionality. When yet another learning algorithm (say

adaptive activation functions [27]) needs to be combined with all the existing algorithms, the number of components will be staggering (to support each combination of the three learning algorithms). This approach clearly does not scale well.

A better approach is to implement the new architecture selection and adaptive activation function components as *separate* `Algorithm` implementations. Secondly, an 'algorithm of algorithms' can be written that allows more than one of the existing algorithms to be combined, thus forming a collating top-level `Algorithm` class. This approach avoids the exponential explosion of implementing classes, but may become complex to manage as each of the existing algorithms must now be able to work on their own *and* as part of a composite algorithm mixture. Recall from the start of this subsection that the CILib specification stipulates that an application *must* have an `Algorithm` that works on a `Problem`. To manage the problem and algorithm hierarchies can become very tedious if this approach is followed. Lastly, many of these approaches, such as adaptive learning functions and architecture selection may be used alongside stand-alone CILib algorithms such as PSO or an EA. These existing CILib components need to be used as is, and it cannot be expected for these algorithms to be reimplemented as neural network specific versions of themselves.

The approach taken in this framework is to *decouple* the actual neural network specific learning algorithms from the CILib `Algorithm` class. The `TrainingStrategy` interface represents learning components that are not implemented as stand-alone CILib algorithms, and provides the core methods and relationships needed to build such learning components as described above. Using `TrainingStrategy` immediately solves the problem of integrating more than one `Algorithm` implementation. A predefined implementation of `Algorithm` called `NeuralNetworkController` is provided by the framework along with a single predefined problem component called `NeuralNetworkProblem`. These components are provided as part of the framework and never need to be changed, and their usage allows any combination of `TrainingStrategy` to be used together. More details about these two classes are given later in this section.

`TrainingStrategy` follows the *strategy* design pattern which allows developers to easily change learning algorithms by simply replacing one component for another in a neural network implementation. The advantages of using the *strategy* are:

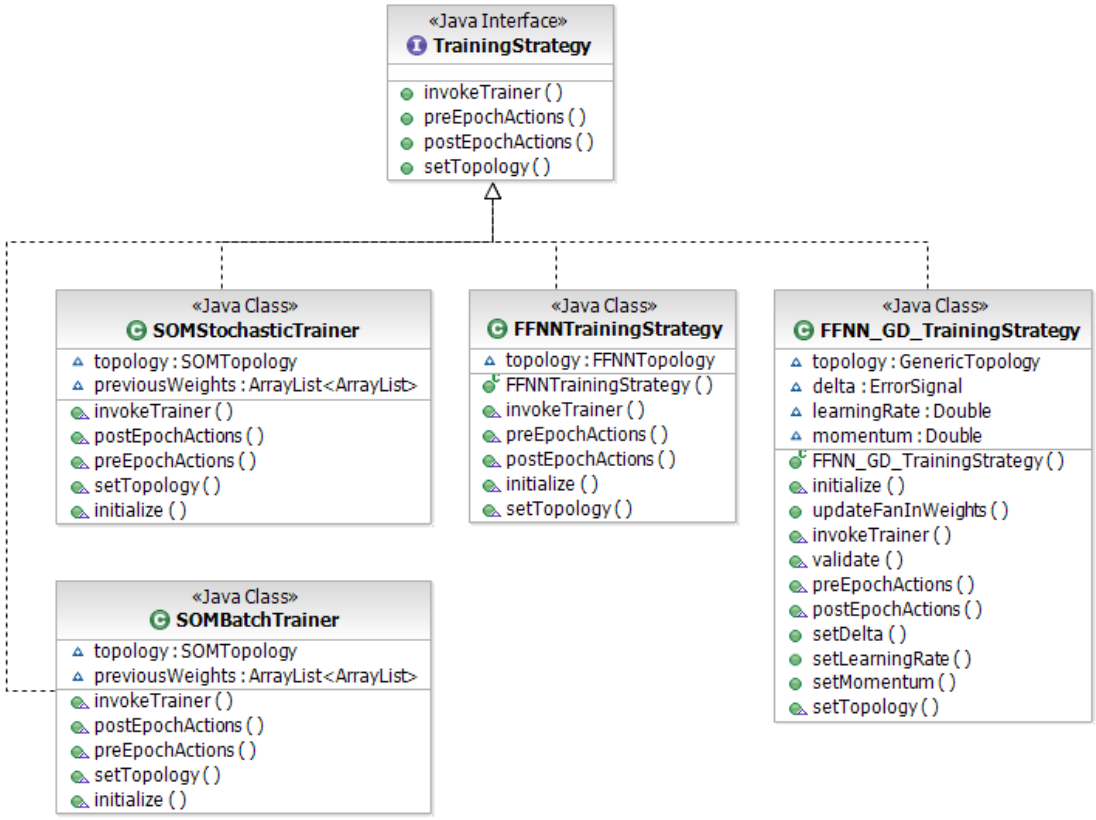


Figure 5.4: Training components specific to neural networks.

- **Encapsulation:** Learning algorithms are fully encapsulated in one component, which makes it easy to exchange implementations or other types of algorithms.
- **Extensibility:** More learning algorithm implementations can be added quickly and easily and may be used by the rest of the system immediately.
- **Abstraction:** Clients such as the mediation component are not concerned with how the neural network is trained – it merely informs the learning algorithm when to perform its role.
- **Concurrent usage:** If the functionality of two or more existing learning algorithms are needed by the same neural network implementation, it is easy for clients to use two or more simultaneously. An example is using gradient descent along with an architecture selection algorithm on the same topology. Another example

is if another `Algorithm` such as PSO is used, an architecture selection or adaptive activation function learning algorithm may still be used (and PSO is still the controlling `Algorithm` as CILib requires).

- **Dynamic modularity:** If a new type of neural network implementation requires that the type of learning algorithm needs to be changed dynamically at runtime based on certain conditions, the *strategy* pattern will allow this. A simple example of such an application is to use LeapFrog (see section 2.3.1) to train a neural network up to a certain point, and then to switch over to gradient descent for the final training stage. Another example is to use a metric to decide dynamically whether an architecture selection algorithm is needed, as well as dynamically deciding on which specific architecture selection algorithm to instantiate and use.

The `invokeTrainer()` method of `TrainingStrategy` provides the ability to invoke a single iteration of the particular learning algorithm. The `invokeTrainer()` method takes a parameter of type `Object`, which allows developers to be flexible in how they want to pass information to the training algorithm. A null value should be passed if this parameter is not used. The `preEpochActions()` and `postEpochActions()` methods are used to implement any actions that need to happen before or after a training iteration – `preEpochActions()` is meant to be called before the `invokeTrainer()` method begins and `postEpochActions()` is meant to be called after the `invokeTrainer()` method has been called. These methods are for learning algorithm specific actions, such as resetting variables, updating learning rates, or increasing counters. As they are part of the `TrainingStrategy` interface, these methods need to be implemented by the implementation developer – an empty implementation is the default if no action needs to be taken. The responsibility to orchestrate exactly how and when these three methods are invoked falls on the mediation component (which may use other design patterns such as the *template method* to outline training steps).

The `invokeTrainer()` method is responsible for obtaining any training related information from topology components, as well as modifying any topology information (such as weight values, neuron parameters, or architecture selection). As `TrainingStrategy` is a Java interface, developers are given full freedom in how they want to implement `invokeTrainer()`. With reference to figure 5.4, an object reference to a specific topol-

ogy implementation is included in each `TrainingStrategy` implementation. This means that these particular `TrainingStrategy` implementations are directly linked to particular topology implementations. This has the advantage of speedy execution, as well as ensuring that incorrect configurations do not occur (as Java will highlight the error). However, the learning algorithm is not generic anymore – it is tied to a specific topology implementation.

It is also sometimes possible to perform training via the `TrainingStrategy` interface only, in which case no direct link between a `TrainingStrategy` implementation and a specific topology exists – the training component is thus truly generic. A possible candidate for such an interface-only implementation is gradient descent. The neural network weight vector could be passed to the `invokeTrainer()` method via its `Object` parameter, perhaps as a list of layers. The resulting changed vector can then be returned and reinserted into the network topology without the `TrainingStrategy` implementation ever directly accessing the weight values inside the topology. Classes that implement `TrainingStrategy` are free to use only interfaces to obtain and update information, thus making the training strategies reusable across topology implementations.

The approach followed in this framework design is to allow all of the above implementation options. Developers have the option to build topology specific as well as generic 'topology independent' learning algorithm implementations. A possible concern is that a direct reference is made to a topology object, which means that there is a possible explosion of classes if learning algorithms have to be reimplemented for *each* topology implementation. This is addressed by the fact that most applications in CILib would use the generic `GenericTopology` implementation discussed in section 5.2.1, thus limiting the number of topologies substantially.

The `TrainingStrategy` interface adds a lot of flexibility when dealing with neural network specific algorithms. Recall from the start of this subsection that the CILib specification stipulates that an application *must* have an `Algorithm` that works on a `Problem`. In order to use the `TrainingStrategy` interface, a dedicated subclass of `Algorithm`, called `NeuralNetworkController`, is used that works on `NeuralNetworkProblem` classes. The `NeuralNetworkController` class is needed only when the neural network framework itself incorporates an optimisation algorithm and there is no other `Algorithm` implemen-

tation (say PSO) present.

The `NeuralNetworkProblem` class implements the `OptimisationProblem` interface to conform to the `Problem` interface of CILib. The `NeuralNetworkProblem` class is an example of the *facade* design pattern, which allows the `NeuralNetworkController` to access a neural network via the `Problem` interface. The `NeuralNetworkController` class is not needed if an external stand-alone algorithm (such as PSO) is to be used, as these algorithms will have their own `Problem` implementations. The `NeuralNetworkProblem` is only used with the `NeuralNetworkController` class. This is not a hard rule – if an application will benefit from using the `NeuralNetworkProblem` definition as a *facade*, it may of course do so. Apart from certain small housekeeping tasks related to `Problem`, the class acts as a wrapper for the mediation component it aggregates.

5.1.4 The Mediation Component

Sections 5.1.1, 5.1.2 and 5.1.3 respectively discussed the implementation of the neural network topology, data and learning components. These sections alluded to the fact that a mediation component is required to consolidate the various parts to form a viable neural network implementation. In particular, the mediation component has these design requirements:

- The neural network topology, data and learning components are all *separate* components. On their own, these components do not provide a full working neural network implementation, but merely provide the algorithms, data structures and approaches that together form the building blocks of a neural network implementation. Recall that the neural network topology, data and learning components all use the *strategy* design pattern. A mediation component is needed to combine the neural network topology, data and learning components into a coherent neural network implementation.

- The neural network topology, data and learning components need to be unaware of each other (i.e. they need to be loosely coupled). As many different types and implementations of each of these components are possible, their interactions

should not be controlled from within each component. This would lead to a *many-to-many* relationship between components which would limit flexibility, increase the likelihood of errors and severely increase code complexity. Performance will also suffer, as the code to manage interaction among objects will start to become more than the functional code. A way is needed to manage all object interactions centrally using *one-to-many* relationships.

- In addition to centralising object interactions using *one-to-many* relationships, the *logic and order* in which these interactions need to occur needs to be defined. For example, when using a FFNN and gradient descent training, a pattern from D_T is first presented to the network and the output is returned, which is then used to calculate the error, given the target value from the pattern. This error is then passed to the training component to perform weight changes. Another example is the order and location of the logic that determines when an active learning update should occur.

- There exists a need to capture and manipulate measurements and information that fall above and beyond the functionality provided by the neural network topology, data and learning components. These measurements and information need to be captured at specific times using specific inputs and outputs to the three mentioned components. Other information may also be used, such as total execution time or total pattern evaluations. All this information needs to be collated, manipulated and stored in a central location. Examples include the error the network made on a particular pattern, the total MSE metric over an epoch, variables and counters for active learning, and the total number of evaluations performed over all epochs (a useful measure when considering algorithm complexity), among other measures. This information must also be made available to external components such as CILib measurements (see section 4.1.4).

- If different neural network behaviour is needed, only the mediation component needs to be modified, and not *all* the components such as the topology, data or learning components. If the same neural network topology, data and learning component implementations are required (i.e. without any change to component

logic), but with totally different application behaviour, the mediation component must simply be exchanged for another mediation implementation. Alternatively, the mediation component itself must be flexible enough to be customisable. An example of this is mediator a that provides a FFNN that uses only the D_T set during training, and a mediator b that uses D_T and D_V for overfitting analysis, as well as D_G for generalisation performance analysis. Both a and b use the same components, but different logic is used in each mediator to provide different implementations.

- Similar to the previous point, there may be cases where the mediation component needs to remain unchanged, but one or more components in the mediator need to be exchanged for different implementations. For example, one data component may have to be exchanged for another data component implementation without having to change the mediator code. The central mediator component must allow loose coupling of components, i.e. support the *strategy* design pattern as used by the topology, data, and learning components.

- A neural network system still needs to be mediated after training is complete. A component is required that performs mediation between components such as the topology and data component, so that the trained neural network may be used in user applications, or as a fitness function in other CI algorithms. This needs to be decoupled from the CILib `Problem` interface, which is used for training only.

- External applications need to access the neural network framework as a single coherent system and not as a set of loose components. A component is needed that uses the *facade* design pattern (see section 3.3.1) to hide the complexity of the neural network framework and present any external system with a single interface that can provide all required functionality. In reality, the component still comprises of multiple components.

A single abstract class called `EvaluationMediator` is defined that uses the *mediator*, *prototype*, *strategy*, and *facade* design patterns to meet the requirements listed above. The `EvaluationMediator` class, illustrated in figure 5.5, lays the foundation for the implementation of subclasses that are responsible for composing working neural network implementations.

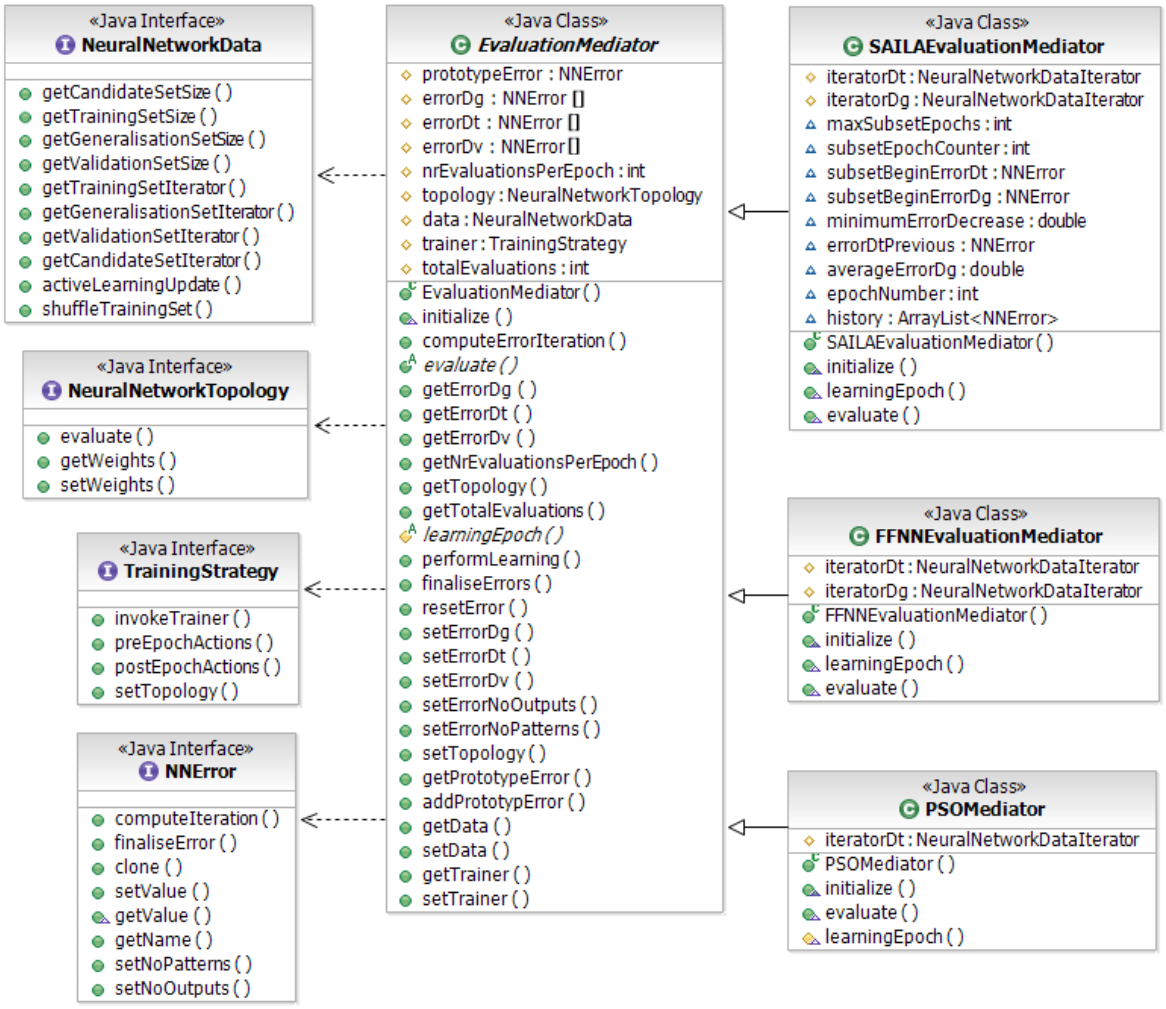


Figure 5.5: Neural network architecture using a mediator.

By using the *mediator* design pattern, a lot of flexibility is gained. Advantages of using the *mediator* pattern are the following:

- *Loose coupling* is achieved between the neural network topology, data and learning components by not letting these components refer to each other directly. Instead, subclasses of `EvaluationMediator` can implement any type of interaction between the components and act as a broker between the neural network topology, data and learning components by passing information between these components. In this way, *many-to-many* relationships between components are replaced with *one-*

to-many relationships, with the mediator sitting in the middle.

The topology, data, and learning components are not concerned about where they obtain input from, or where their output goes to. The mediator is responsible for providing each component with its needed information, and managing any output. For example, the topology component is not concerned about where it obtains `NNPattern` objects from, or what to do with the `MixedVector` network output. The data component provides `NNPattern` instances via an *iterator* and is not concerned about which component is asking for patterns. Learning components merely need to be invoked at the correct time, with any information that it might require (which is implementation specific). Using the *mediator* pattern decouples all components and turns interactions into *one-to-many* relationships.

- The *logic and order of interactions* between components can be defined using the *mediator* pattern. This means that for any particular mediator implementation (i.e. subclass of `EvaluationMediator`), information can 'flow' between the neural network topology, data and learning components in a particularly defined manner. The mediator is responsible for handling the interactions between components. Broadly speaking, the algorithms listed in chapter 2 give the outline of the flow of a neural network model – the logic, order and any variables such as counters are defined in the mediator, while the steps are delegated to components such as the topology, data or learning components.

  As an example, a FFNN that is trained using gradient descent needs to iterate over D_T, evaluate each pattern and obtain its output, calculate the error that the network made using the pattern target, use this error to change network weights, and finally report the MSE for each epoch. The *mediator* pattern provides the logic and ordered flow of information between components.

- *Limitted subclassing* is achieved as only the mediator needs to be extended to support new functionality, as opposed to each of the other components having to be extended otherwise. Each subclass of `EvaluationMediator` represents a new combination of neural network components to form a new type of neural network. Using different mediator implementations allows developers to leave existing neu-

ral network topology, data and learning components unchanged, yet achieve new functionality by merely replacing the mediator component. Without a mediator, the entire system would be a monolithic structure that would have to support all possible configurations.

As an example, consider a FFNN that is trained using gradient descent and fixed set learning. If a mediator is not used, and active learning should replace fixed set learning, it would cause code changes in the topology component, as the logic to evaluate a pattern would probably be implemented directly in the topology. The learning component code would also have to be changed, as aspects such as the number of patterns presented per epoch are now variable (due to active learning), which influences the MSE calculation in the learning component. Using the *mediator* pattern requires *no* changes in the topology, *no* changes in the learning component, and a new (but simpler) data component[4].

- After a neural network is trained, the *mediator* pattern can be used to integrate the various separate components of the network (such as the topology and data components) to form a usable application. In this way, the role of the mediation component is extended beyond neural learning scenarios and can aid in using trained neural networks in other CI algorithms. The use of a mediation component decouples the definition of a neural network model from any specific `Problem` component, which allows external learning algorithms such as PSO to reuse existing implementations of `Problem`.

The *mediator* pattern acts as a client for the *strategy* design pattern, which means that for each mediator implementation, any neural network topology, data and learning components (which all support the *strategy* pattern) can be exchanged without changing the mediator code. This is done by changing the mediator's references to `NeuralNetworkTopology`, `TrainingStrategy`, as well as `NeuralNetworkData` objects. This capability gives a lot of flexibility to the neural network framework:

[4] A small change in the mediator implementation may be needed if the mediator does not support active learning. The `activeLearningUpdate()` method of the data component needs to be invoked at the correct time.

- Data implementations may be exchanged easily, allowing details in the data component such as the data source, pattern distribution, and pattern ordering to be changed without having to change any code in the mediator. Active learning can also be supported in this way – merely replace a fixed set learning data component with a component that uses an active learning approach. This however requires that the mediator supports active learning (i.e. the mediator must know how and when to invoke the `activeLearningUpdate()` method of the data component which triggers active learning).

- Topology implementations are easy to exchange, as the `evaluate()`, `getWeights()` and `setWeights()` methods of the topology component merely get invoked on a different topology implementation. This gives great flexibility in allowing developers to exchange, say, a slow but accurate (to the 30th decimal point) FFNN topology with a faster implementation that is only accurate up to 8 decimal points. Other examples include exchanging topologies that use different types of activation functions, using SOM topologies that implement different neighbourhood functions (rectangular vs. hexagonal for instance), exchanging an Elman recurrent topology for a Jordan (see section 2.4.3) topology, or exchanging a singular topology for a modular or ensemble topology (see section 6.3.1).

- Different learning components may be used with the mediator. From the perspective of neural network specific learning algorithms, the *strategy* pattern allows learning components to be exchanged with ease. More than one component may be used simultaneously as well, for example using an architecture selection component as well as a weight adjusting component. Note that the strategy is not used to support CILib stand-alone algorithms such as PSO – there is a cleaner and simpler solution for this via the use of adapters, as discussed in section 6.2. The main difference between learning components that use the *strategy* and those that do not, is that the learning component is no longer invoked from within the mediator, as the external algorithm is directly responsible for neural network learning. Both approaches require interaction with the mediation component.

The abstract `learningEpoch()` method can be considered the main method for controlling the flow of a neural network model. The `learningEpoch()` method is implemented in subclasses to facilitate all learning-related aspects of a particular neural network model. The `learningEpoch()` has to control the iteration over the desired data sources, make sure patterns are presented to the network and output retained (if applicable), invoke the `TrainingStrategy` class at the right time and pass it the correct information, make sure error metrics are computed correctly and at the right location, invoke active learning at the appropriate time, and any other actions that need mediation. In short, all the logic and interactions for brokering the various components of a neural network model is defined here.

The `learningEpoch()` method is flexible enough to fulfil neural network training requirements. If the `NeuralNetworkController` is used, the `performIteration()` method of `NeuralNetworkController` calls the `performLearning()` method in the mediator via the reference in `NeuralNetworkProblem`, which calls the `performLearning()` method in the mediator. The `performLearning()` method in the mediator performs a number of household tasks (such as incrementing counters) before calling the `learningEpoch()` method, and the *template method* design pattern (see section 3.2.5) is used to achieve this. The `learningEpoch()` method is abstract and subclasses of `EvaluationMediator` provide the exact functionality. In the case where the training component is another `Algorithm` such as PSO or EA, the `learningEpoch()` method still defines the outline of the logic for calculating the required error metric over a data source, as discussed in section 6.2.

Subclasses of `EvaluationMediator` inherit functionality as discussed in detail below. Subclasses merely need to make use of this functionality at the appropriate times. Subclasses only need to implement two abstract methods, namely `evaluate()` and `learningEpoch()` as seen in figure 5.5.

The `evaluate()` method takes an `NNPattern` object as input and returns the output of the neural network as type `MixedVector` (recall from section 4.1.5 that `MixedVector` is a CILib `Type`). It does so by forwarding the request to a topology object. A `MixedVector` object is returned, as the output of a neural network is an array in most instances, and `MixedVector` is the standard data type used in CILib. Using `MixedVector` makes it

easier for other CI algorithms to access neural network functionality. If no output is returned (as is the case with SOM), a null value is returned.

The abstract `EvaluationMediator` class provides the base implementation for error reporting metrics, which are represented by the `NNError` interface (as seen in figure 5.5). These may be used by the application for training purposes, or may be extracted as measurements using the CILib measurement system discussed in section 4.1.4. The `NNError` interface provides methods to allow the metric value to be set directly to a particular value using `setValue()`, as well as to allow the metric to be computed over an epoch by using the `computeIteration()` and `finaliseError()` methods. The final metric value is obtained by using the `getValue()` method. Three arrays of type `NNError` are provided to keep track of errors on D_T, D_G and D_V respectively. These arrays are `errorDt`, `errorDg`, and `errorDv` and are all of type `NNError[]`. Arrays are used so that more than one error metric may be specified per data source at the same time, such as having the MSE and classification accuracy metrics be associated with the algorithm simultaneously.

A challenge with using an array to house error metrics is that the arrays for D_T, D_G and D_V have to be kept in sync (i.e. they must all house the same type of metrics in the same order). Each metric in each array must be reset/recreated by the mediator before an epoch commences. The mediator cannot use the error metric class constructors to create the objects, as these metrics are assigned dynamically based on the XML setup document provided to the CILib simulator (see section 4.1.6). Furthermore, each array needs to be treated and manipulated as a whole. This means that all the metrics in the array need to be created, updated and finalised (i.e. the `finaliseError()` method) as a single unit. An example of finalising a metric is obtaining the MSE by dividing the accumulated error total (or SSE) for an epoch by the number of patterns and output units (see equation 2.19).

To solve the problems of dynamic selection and creation of error metrics, the *prototype* design pattern discussed in section 3.1.2 is used to define the prototype array `prototypeError` for the three error arrays. Each time a metric has to be constructed in a mediator, the prototypical instances are used to make copies of pre-configured error metrics. The *prototype* pattern decouples the logic of creating the error metric from

the implementation of the mediator, allowing error metrics to be assigned dynamically without having to explicitly use constructors. Any metrics may now be used with a mediator, and not only those that have their constructors supported.

The `EvaluationMediator` class also provides helper methods to perform common actions on each error metric in an entire array at the same time. If the helper methods are not used, developers would have to manually iterate over each error array and manually call operations of the `NNError` interface on each metric in the array. Helper methods take care of the iteration and merely forward method calls to each metric in an array. Any of the `errorDt`, `errorDg`, and `errorDv` arrays may be passed as parameter to these helper methods. Example methods include the `resetError()` method used to recreate each error metric in the array using the prototypical instance `prototypeError` (which is a `NNError[]`), the `computeErrorIteration()` method that invokes the `computeIteration()` method of each of the metrics in a particular array, the `finaliseErrors()` method which invokes the `finaliseError()` method of each metric in a particular array, and various getter and setter methods (for each error array) that respectively exports or imports an entire error array.

Lastly, the `EvaluationMediator` abstract class acts as a *facade* to the rest of CILib, allowing any external components to access a neural network using a simplified interface. The `evaluate()` and `performLearning()` methods are good examples, as these methods hide the details of network pattern presentation and network learning respectively. External applications merely need a reference to an `EvaluationMediator` implementation and invoke a required method. Furthermore, easy access via a single interface is granted to all the subcomponents of the neural network system, including error metrics, the neural network topology, data, and learning components. The `getTopology()` and `setTopology()` methods provide access to the `NeuralNetworkTopology` implementation, which in turn allows elements of the network topology such as weights and/or neurons to be accessed. Similarly, the `getData()` and `setData()` methods access the data component and the `setTrainer()` and `getTrainer()` methods provide access to the learning component.

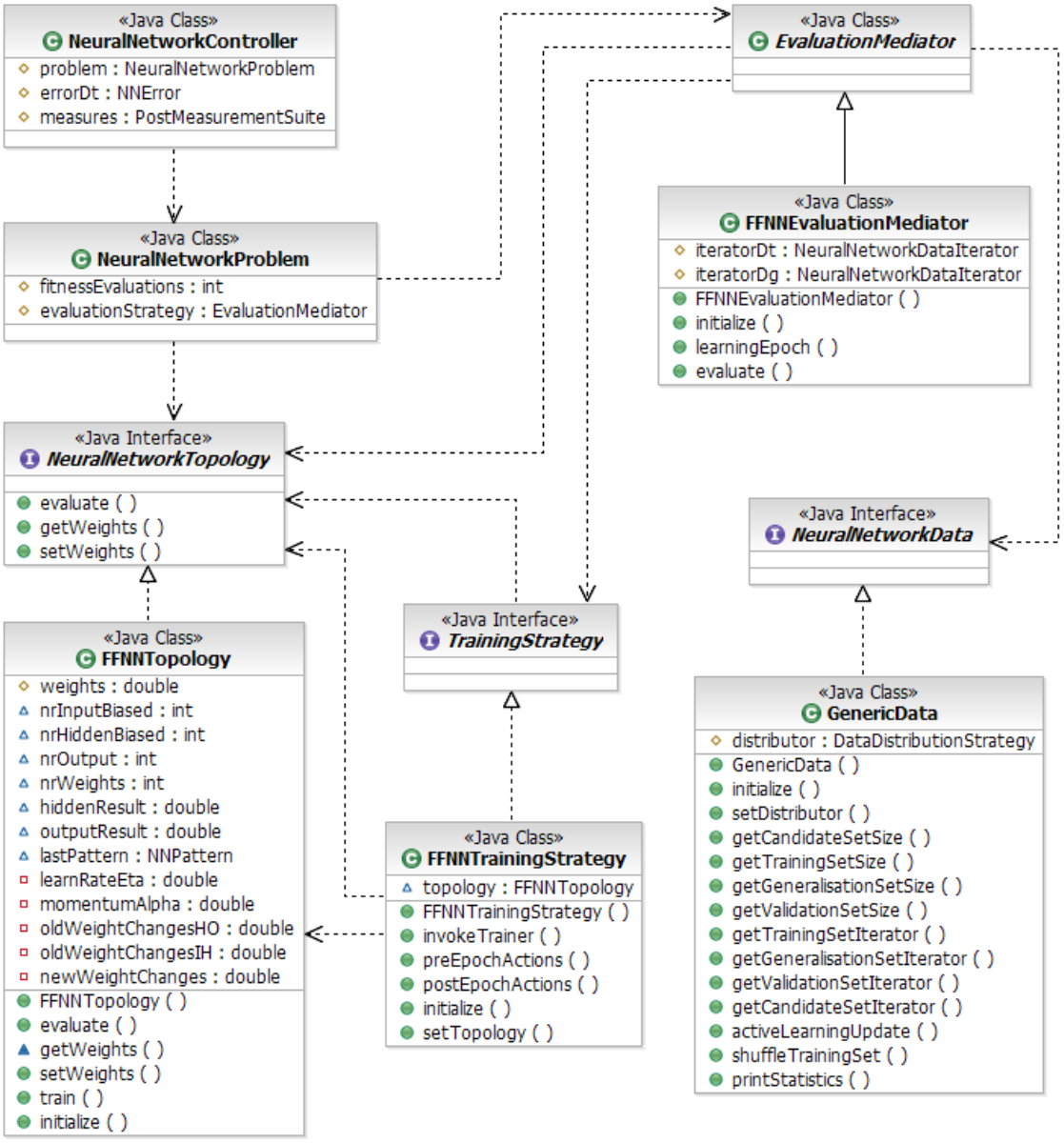


Figure 5.6: FFNN implementation in the foundation framework

5.1.5 An Example implementation

Figure 5.6 shows the layout of a stand-alone implementation that is developed using the foundation framework. In this example, a simple feedforward neural network (as depicted in figure 2.11) is implemented that uses gradient descent as the weight optimisation

algorithm. The main design goals of this neural network are simplicity and speed. The `FFNNTopology` class extends `NeuralNetworkTopology` and acts as the main container for weights and neuron functionality. The characteristics of the neural network model are hard-coded in very efficient code to ensure speedy processing. The characteristics of the implementation include:

- A 3-layer architecture, with linear activations for input units and sigmoid activation functions (see equation 2.5) for hidden and output units.
- The number of neurons in each layer can be configured.
- Weight values can be extracted and inserted via the class interface.
- Java basic types are used as much as possible to ensure fast processing. Arrays that store the weights are optimised to be one long flat array, rather than supporting more types of network topologies (which would increase complexity and probably decrease performance and simplicity).
- A generic data implementation called `GenericData` is used, which is discussed in more detail in section 5.2.2.

To setup this neural network using the CILib simulator (see section 4.1.6), the following complete XML setup document must be created:

```
<simulator>
   <algorithms>
      <algorithm id="NNalgo" class="neuralnetwork.foundation.NeuralNetworkController">
         <addStoppingCondition class="stoppingcondition.MaximumIterations" iterations="1000"/>
      </algorithm>
   </algorithms>

   <problems>
      <problem id="NNProblemFSL" class="neuralnetwork.foundation.NeuralNetworkProblem">
         <evaluationMediator class="neuralnetwork.generic.evaluationstrategies.FFNNEvaluationMediator">
            <topology id="NNtopo" class="neuralnetwork.basicFFNN.FFNNTopology">
               <nrInputBiased value="5"/>
               <nrHiddenBiased value="4"/>
               <nrOutput value="3"/>
               <learnRateEta value="0.5"/>
               <momentumAlpha value="1.0"/>
            </topology>
            <data id="dat" class="neuralnetwork.generic.datacontainers.GenericData">
```

```
            <distributor class="neuralnetwork.generic.datacontainers.RandomDistributionStrategy">
               <file value="c:/data/IrisScaled.txt"/>
               <noInputs value="4"/>
               <percentTrain value="80"/>
               <percentGen value="5"/>
               <percentVal value="15"/>
               <percentCan value="0"/>
            </distributor>
         </data>
         <addPrototypError class="neuralnetwork.generic.errorfunctions.MSEErrorFunction" noOutputs="3"/>
         <addPrototypError class="neuralnetwork.generic.errorfunctions.ClassificationErrorReal"/>
         <trainer class="neuralnetwork.basicFFNN.FFNNTrainingStrategy">
            <topology idref="NNtopo"/>
         </trainer>
      </evaluationMediator>
   </problem>
</problems>

<measurements id="fitness" class="simulator.MeasurementSuite" resolution="10" samples="30">
   <addMeasurement class="neuralnetwork.foundation.measurements.ErrorDt"/>
   <addMeasurement class="neuralnetwork.foundation.measurements.ErrorDv"/>
   <addMeasurement class="neuralnetwork.foundation.measurements.RobelOverfittingRho"/>
</measurements>

<simulations>
   <simulation>
      <algorithm idref="NNalgo"/>
      <problem idref="NNProblemFSL"/>
      <measurements idref="fitness" file="c:/metrics/nnfitness.txt"/>
   </simulation>
</simulations>
</simulator>
```

The XML document above follows the template given in section 4.1.6 to set up a CILib experiment with an algorithm (with stopping conditions), problem and measurements to facilitate the training of a FFNN. The algorithm component is set up using the `<algorithms>` tag and specifies that `NeuralNetworkController` is to be used, as a neural network specific training algorithm is used. The algorithm is set to only stop after 1000 epochs. The problem is set up using the `<problems>` tag, which is set up to use `NeuralNetworkProblem` in conjunction with `NeuralNetworkController` as discussed in section 5.1.3. The problem is configured to use a mediator component (via the `<evaluationMediator>` tag) called `FFNNEvaluationMediator`. This mediator in turn requires that the `<topology>`, `<data>` and `<trainer>` tags be set (along with anyparticu-

lar parameters), as can be seen in the XML above. Two error metrics are also added, one MSE and another 'percent correctly classified.' Lastly, three measurements are added via the CILib `<measurements>` tag, namely `errorDt`, `errorDv` and `RobelOverfittingRho`.

The training component uses gradient decent optimisation and is implemented by the `FFNNTrainingStrategy` class, which implements the `TrainingStrategy` interface. Invocation calls to the `FFNNTrainingStrategy` class's `invokeTrainer()` method are forwarded to the `train()` method in `FFNNTopology`. The `FFNNTopology` class can access and modify its own members directly, which is much quicker than via the class interface. Note that, while this increases speed and enhances simplicity, it may appear as if the specific training algorithm is defined statically and is thus not flexible. Yet this behaviour is easy to change. By merely replacing the `FFNNTrainingStrategy` class with another subclass of `TrainingStrategy`, or using the `NeuralNetworkFunctionAdapter` as discussed in section 6.2, other training algorithms may also be used. These components do not *have* to use the `train()` method in `FFNNTopology` at all – they are free to use any means necessary to train the network. These may include other types of training algorithms such as Leapfrog, PSO or an EA (through the use of the `getWeights()` and `setWeights()` methods of the topology component), or any other training method.

An important point to note is that `TrainingStrategy` is an *interface*, which means that `FFNNTopology` might as well have implemented `TrainingStrategy` directly, having the `invokeTrainer()` method invoke the `train()` method locally, or even just renaming the `train()` method itself to `invokeTrainer()`. With this approach, developers could *still* use any other training strategy as discussed above – the mediator component just needs to be aware of which training component is the desired one.

The `FFNNEvaluationMediator` class outlines the logical structure of the feedforward neural network. The `learningEpoch()` method iterates over the D_T and D_V data sets and presents all training patterns, with training being invoked after every pattern evaluation (i.e. stochastic learning as discussed in section 2.3). The training set is shuffled after every epoch. The `evaluate()` method takes an `NNPattern` object and returns the network output. The mediator then passes the output to the list of error metrics (along with the associated `NNPattern` object). The error value is also coordinated by the mediator and is passed to the training components (as gradient descent is used, which requires

an error metric for learning). Figure 5.6 does not show the error metrics, but they are added dynamically using the inherited functionality from `EvaluationMediator`.

The `GenericData` class represents a standard data set of `NNPattern` objects and is discussed in more detail in section 5.2.2. The distribution of patterns into the sets D_T, D_V and D_G is done when the data is read in from a file. For interest, more advanced `NeuralNetworkData` implementations are discussed in chapter 6 and any of these may of course also be used here.

In summary, this is a very simple example of a custom neural network implementation that gives a better view on how the foundation framework operates. By using the foundation framework, even a simple system such as this has a high degree of flexibility and reuse:

- Integration with the rest of CILib's algorithm, problem, type system, measurements, stopping conditions, and simulation capabilities is provided automatically. The CILib framework is merely extended to provide a specific FFNN implementation.

- It is easy to change the neural learning algorithm, either in a static way before execution or dynamically during execution.

- The management of data is made easier. It is easy to vary data sources, use different data presentation schemes such as fixed set learning, active learning such as the sensitivity analysis incremental learning algorithm (see section 6.1.2) or dynamic pattern selection (see section 6.1.3), or to use differrent ways of distributing patterns across D_T, D_V and D_G.

- `EvaluationMediator` objects can be changed easily to completely change the nature of the neural network implementation. A basic feedforward training scheme that uses fixed set learning and no overfitting measures, for example, can be reconfigured to use active learning with one or more overfitting measures. This is done using the same topology, data and learning components in both configurations – the components are merely mediated differently by a new mediator object to produce a new type of implementation.

- The topology can be easily exchanged for another topology implementation instead of changing topology code everytime different functionality is required. An example is that if a different transfer function for neurons is needed, no code changes need to be performed – a different topology object merely has to be specified. If this new topology object has already been developed before, it means that object reuse is achieved as opposed to code changes.

- If a different training strategy is needed, the `TrainingStrategy` object merely has to be replaced. It is also possible to remove it completely and use a `PSO` or `EA` implementation to train the network. Once again, this requires no code changes. See section 6.2 for a full discussion of using PSO or an EA to train a neural network.

Many of the points listed above illustrate the need to use generic components. For example, it is easy to change the data or topology components for the application, *assuming that other implementations have already been written.* It would be easier and more efficient to reuse an existing data or topology component and merely reconfigure it, rather than writing a new implementation.

The next section illustrates the concept of generic components, such as a generic data component that is able to provide four data sets, namely D_T, D_G, D_V and D_C that have been populated using any defined distribution algorithm (such as a fixed distribution as dictated in a specific list, random pattern distribution, or K-fold cross-validation) with data patterns coming from any source such as a file, XML files, network service, database, function output, or any other source.

5.2 Generic Components

This section discusses generic components that developers can reconfigure to represent different aspects of neural network models – with minimal or no coding. A good example of such a component is the `GenericTopology` class which is capable of representing any neural network topology that can be expressed as a graph. The `GenericTopology` class is discussed in more detail in section 5.2.1.

Another good example of a generic component is the `GenericData` class. This component allows developers to read data from any source and distribute patterns among

four distinct data sets, namely D_T, D_G, D_V and (if needed) D_C as discussed in section 2.2. A discussion of the `GenericData` class is given in section 5.2.2.

Architecturally, all neural network generic components are built on top of the foundation framework, as is illustrated in figure 5.1. Generic components offer developers a set of predefined implementations of common neural network functionality (such as generic topology or data components) that give benefits such as:

- **Generic implementations save time:** Researchers can develop generic implementations of components that are used regularly by many neural network applications. By developing these components in a flexible way, developers save time as they do not need to rewrite existing functionality when writing new applications.
- **Component quality assurance:** By reusing a generic component, application integrity is improved and testing time is reduced. By using a proven, reliable generic component, developers are assured that this specific part of the neural network implementation is working as designed.
- **Flexible design with minimal coding:** The generic components framework provides base implementations for components that are needed by most typical neural networks. Developers can easily build new types of neural network implementations by configuring existing generic components and merely adding custom components to complete the application.
- **Modularity:** Developers have the ability to use a generic component initially and swap it out for a specific custom component at a later stage in conjunction with the mediation component as discussed in section 5.1.4, without affecting the rest of the application. This concept of modularity and interoperability is key to making neural network implementations simple, quick and easy to develop.
- **Efficiency:** If the design of a particular component is done well, it is very efficient to reuse that same component rather than spending more time and effort to 'reinvent the wheel.'

In short, generic components are a collection of well-tested implementations in the foundation framework that allow developers to easily reuse or extend them. It is clear

that generic components are *implementations* of the foundation framework rather than an extention to the framework. Of course one of the main objectives of generic components is that they are *generic*, i.e flexible, configurable, extensible, and reusable. Thus there are Java interfaces that need to be implemented before a component will be able to support new functionality. An example (which will be discussed in more detail below) is a generic topology builder object that needs to be defined for a particular type of topology (such as a SOM) before that topology can be constructed.

The rest of this section discusses the `GenericTopology` and `GenericData` classes, how these components are implemented, and how they can be used to solve complex design problems.

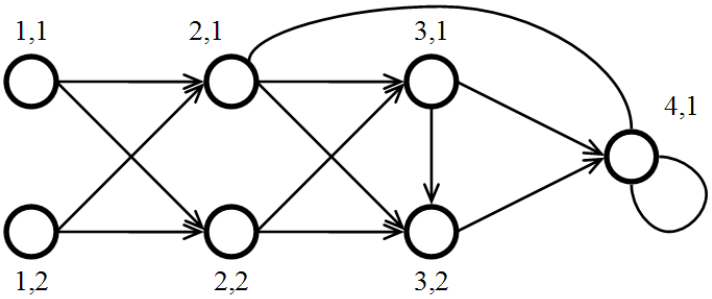
5.2.1 A Generic Topology Implementation

The `GenericTopology` class implements the `NeuralNetworkTopology` interface from the foundation framework. As mentioned in chapter 2 and in [34], a neural network topology can be implemented as a mathematical graph using a matrix representation[5]. Figure 5.7 shows a neural network topology viewed as a graph along with the matching connectivity matrix that represents the weight connections between neurons. The rows and columns in the matrix both contain all the neurons in the network, with weights from the same layer ordered together. The tuple (l, n) should be interpreted as neuron n in layer l.

The graph in figure 5.7 is read as follows: the neuron in column c has as its input all the connections to the neurons in the rows where the entries contain a $\bullet$ (i.e. is not empty), where $1 \leq c \leq N$ and N is the total number of neurons in the network. For example, the column of neuron $(3, 2)$ has as its inputs all the outputs of neurons $(2, 1)$, $(2, 2)$ and $(3, 1)$, as these rows all have non-null entries.

The graph in figure 5.7 can express all four types of weight connections as discussed in section 2.1.2. Entries on the diagonal indicate self-connections (a weight connection's start and end neuron is the same) and any entries below the diagonal are either inter, intra (if they fall in the same layer) or supra layer connections that originate from later layers. An example is the neuron $(2, 1)$ that has a connection from neuron $(4, 1)$ in its

[5] See [29] for a full discussion on using matrices to implement mathematical graphs.



	1,1	1,2	2,1	2,2	3,1	3,2	4,1
1,1			•	•			
1,2			•	•			
2,1					•	•	
2,2					•	•	
3,1						•[1]	•
3,2							•
4,1			•[2]				•[3]

[1] Intra layer connection
[2] Supra layer connection
[3] Self connection

Figure 5.7: Weights connections represented as a graph

input. Entries above the diagonal are either inter, intra or supra layer connections that originate from earlier layers. Note that the weight connections are directional – entries from a neuron in column (a, b) that has a neuron in row (x, y) as input are not the same as the connection from a neuron in column (x, y) to a neuron in row (a, b), except on the diagonal where $(a, b) = (x, y)$. This allows for full asymmetric and symmetric weight connections in neural network models as discussed in section 2.1.2.

As can be seen in figure 5.7, each layer of neurons is cleanly separated into bands when taking the column number into consideration. Each of these bands have a number of neurons in it, for example, in figure 5.7 there is a band consisting of two columns $(2, 1)$ and $(2, 2)$ which together represent the second layer in the figure. Grouping neurons using these bands is very useful to be able to implement the weight matrix efficiently. By replacing the • symbols in the matrix with weight values, it becomes possible to

calculate the output of a neuron in a column c by merely using the non-null entries in the column as the weight vector in the activation function of c. The immediate problem with this approach is that a typical weight matrix is very sparsely populated, which impacts on the efficiency of calculating the activation value programmatically.

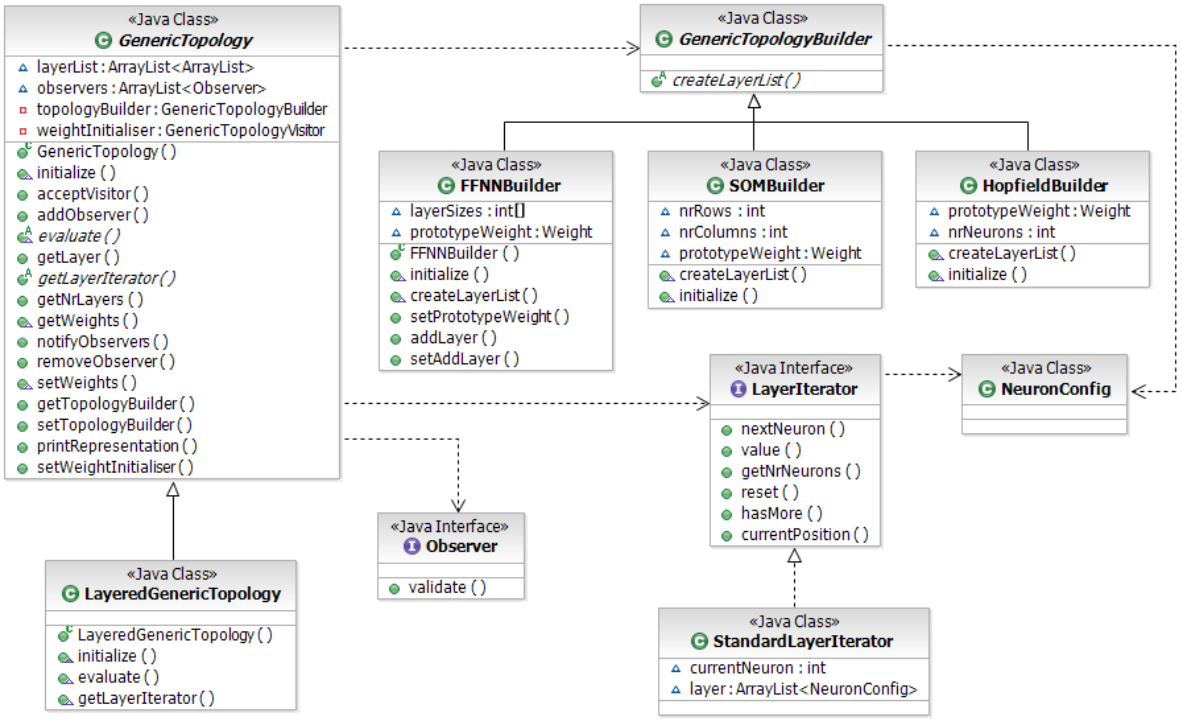


Figure 5.8: The generic neural network topology system

The way in which the `GenericTopology` class implements neurons and weights is to use the connectivity matrix approach on a logical level, but to avoid the sparse matrix problem by implementing the graph in an efficient way using `ArrayList<NeuronConfig>` objects. Each column in figure 5.7 is represented by a `NeuronConfig` object (see figure 5.9) which is used to hold all information a neuron needs to compute its output value.

At this point, a way of representing a weight connection matrix in Java has been shown, but the problem of dealing with a sparse matrix in an efficient way still needs to be addressed. The solution used is that the `NeuronConfig` class merely ignores empty entries – only non-null weight values and neuron links are stored. Doing this of course prohibits the network from using matrix coordinates to determine the inputs of a neuron

in column c by using row indices (as described earlier). For this reason, the `NeuronConfig` object also keeps a list of references to other `NeuronConfig` objects that indicate the source neuron for each input weight of c. This object holds information such as

- `currentOutput` of type `Type`, used to store the output of the neuron;
- `input` of type `NeuronConfig[]` which is a list of all input neurons;
- `inputWeights` of type `Weight[]` which is the corresponding weight values for `input`;
- `patternInputPos` of type `int`, used to indicate which location in the pattern is used by this neuron;
- `patternWeight` of type `Weight`, the corresponding weight for `patternInputPos`;
- `timeStepMap` of type `boolean[]`, used to indicate whether to use input[i]'s current output (a value of zero) or timestep t-1 output (a value of one)(this is used mostly in recurrent architectures with self connections);
- `Tminus1Output` of type `Type` which is used to keep track of the neurons's previous output. This is needed in cases where *limiting* (also known as *clamping*) is used to revert to the neuron's previous output value as opposed to using the newly computed value (see section 2.1.1); and
- `isOutputNeuron` of type `boolean` which is used to indicate to the topology if this is an output neuron.

The list above shows that `NeuronConfig` uses a fan-in centric approach, which means that each neuron is responsible for knowing where all its inputs come from and what the weight values are – all information about the neuron is contained in a single `NeuronConfig` object.

Note that every neuron in the topology has to separately keep track of 'internal' weight connections (connections from other neurons) and 'external' weight connections (input from a pattern). Logically, there is no difference between an input from a pattern and an input from another neuron, but on a programmatic level these inputs have to

be treated differently. It is important to make this distinction when considering modular network topologies, where the pattern weights specifically need to be set by the encapsulating network. More detail on this is given in chapter 6.

Weight values for each weight is stored in a `Weight` class. The value stored in `Weight` is an instance of CILib's `Type` component as laid out in section 4.1.5, which effectively allows any type of weight to be used (i.e. real, complex, binary, vector, among others), as discussed in section 2.1.2. The `Weight` class is needed to store more information such as past values of the weight, and may be extended to store other information that any particular neural network model may require. Gradient descent is a good example of an algorithm that requires past weight changes to be remembered. If `Type` were to be used directly instead of `Weight`, other components would have to keep track of the old weight values, effectively copying the topology structure. This in turn is error prone, tedious to construct and maintain, and is also inefficient.

Topology builders (discussed below) cannot possibly check for *each* possible type of `Weight` object configuration (as weights can contain any `Type` object), which presents a challenge as the builder is responsible for setting up the weight framework itself. The *prototype* design pattern is used by the `Weight` class, which allows topology builders to create new instances of `Weight` without knowing any implementation details about the `Weight`. Using the *prototype* pattern simplifies the builder class dramatically and allows it to be expanded with more `Weight` implementations without having to change the builder code. If the *prototype* pattern is not used, all instances of `Weight` would have to be instantiated using the `Weight` class's constructor. As the class type of `Type` is only set at runtime via a CILib XML document, the *prototype* pattern makes it possible to avoid using the constructor of `Weight` and simplify the builder code.

All the columns located in the same band in figure 5.7 are grouped together into a single `ArrayList<NeuronConfig>` object, where each column represents one neuron by using a `NeuronConfig` object. A variable of type `ArrayList<ArrayList<NeuronConfig>>` called `layerList` contains each of these layers to give the desired matrix structure. A `LayerIterator` interface is also provided and is used by `GenericTopology`, its subclasses and other classes to iterate over neurons in a layer. The *iterator* design pattern is used here to make it easier for components external to `GenericTopology` to gain access to

and modify the neuron structure in an efficient manner.

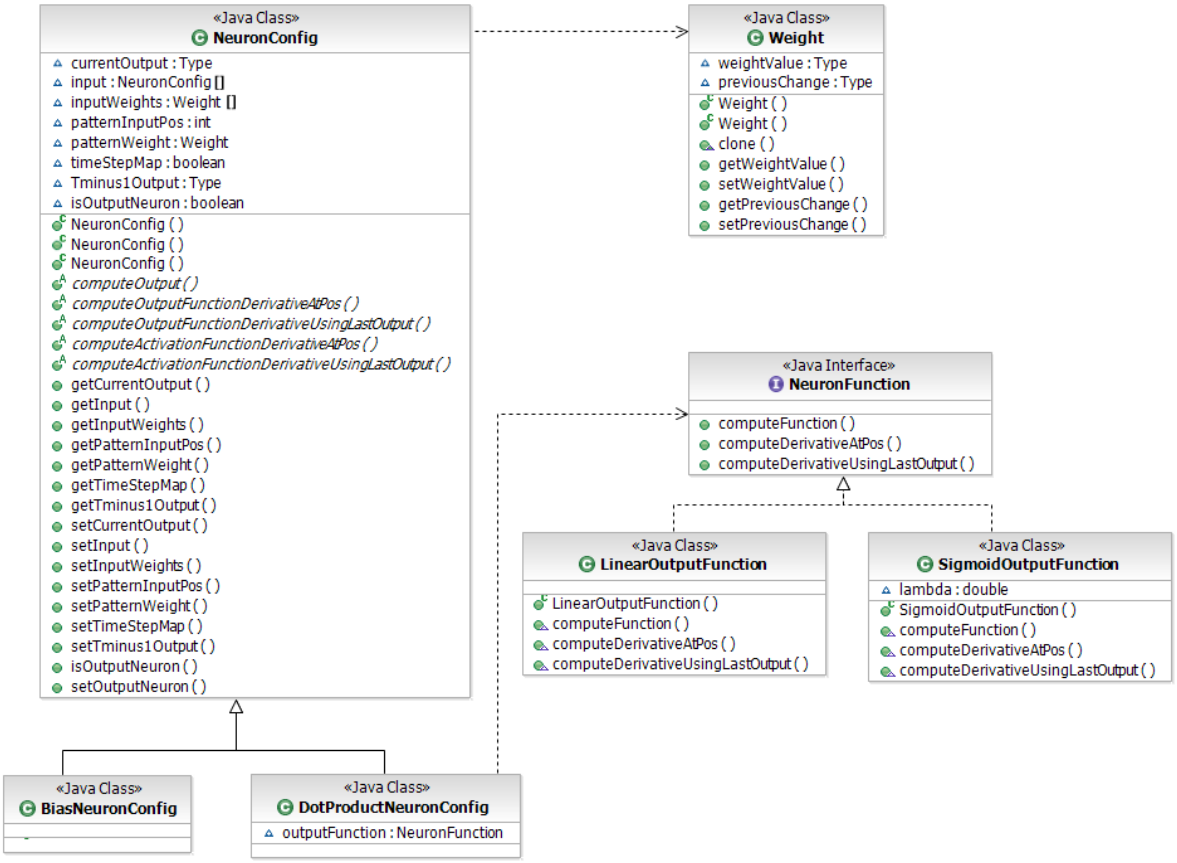


Figure 5.9: layout of `NeuronConfig` and its sub-components.

With reference to the class `NeuronConfig` in figure 5.9, notice that the abstract methods start with 'compute' . The class member variables merely provide the information needed to compute the output of a neuron as opposed to the *type* of neuron that is represented (i.e. the transfer function details). Depending on the choice of net input signal and activation function as discussed in section 2.1.1, the mentioned abstract methods have to be implemented differently. Subclasses of `NeuronConfig` can implement these methods in any way to represent neuron types such as bias units, summation units, product units, SOM units, Hopfield units, or any other type of neuron.

The `GenericTopology` class uses the *visitor* design pattern discussed in section 3.2.3 to allow more actions to be added to the topology. Typical actions include the ability

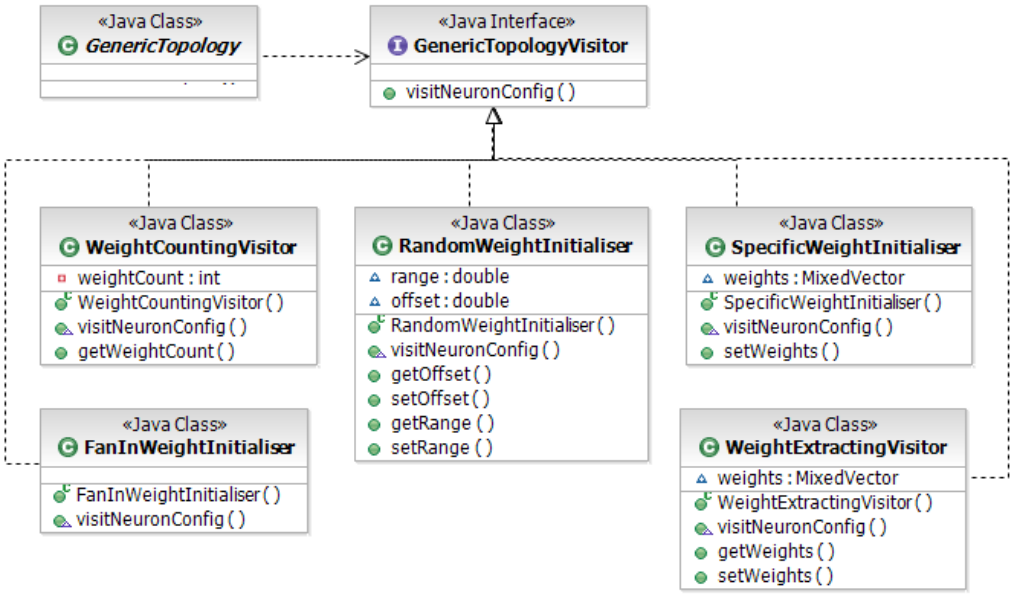


Figure 5.10: Various `GenericTopologyVisitors` used by `GenericTopology`.

to extract or set weight values using a `MixedVector`, initialising weights randomly using range and offset, initialising weight values using the inverse fan-in rule (i.e. to the range $[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$) as discussed in section 2.1.2, extracting weights from the topology in a certain order, and counting the number of weights. Using a visitor decouples the operations on the topology from the representation of the topology, allowing more visitors to be added with ease and without complicating the `GenericTopology` class. The rest of the class can also use visitors to good effect, such as the `initialise()` method using a specified visitor to initialise weights and the `getWeights()` and `setWeights()` methods using visitors to perform their work.

Lastly, the `GenericTopology` class makes provision for the use of the *observer* design pattern, as discussed in section 3.2.4. The *observer* pattern allows any component that is interested in knowing when an event in `GenericTopology` occurs to be notified and to take action. An example of when the observer pattern needs to be used is when a growing/pruning component is used. When the growing/pruning component changes the topology size by adding or removing neurons or weights, other registered components can be notified that the topology layout has changed. Relevant components in this example are the training component and the data component, as they typically have dependencies

on the topology size. These components in turn have to decide if they can compensate for the change and continue execution, or throw an exception. For example, if PSO is used as the learning algorithm, then the size of the topology directly influences the PSO particle size. Similarly, the data component needs to be able to decide if execution can continue if the input or output vector size has changed (such as when input or output neurons have been removed or added). Any number of observers can be added or removed dynamically at runtime, and the code to define more observers is decoupled from the `GenericTopology` class, making it simpler and more robust.

The `GenericTopologyBuilder` Class

The first significant inhibitor of using the `GenericTopology` class is the complexity of constructing the layers of `NeuronConfig` objects, the `Weight` object connections between neurons, and setting up the type of each neuron correctly (net input signal, activation function, initial output value, whether it is an output neuron or not, linking patterns to the neuron and setting it to be a self-connection or not). Fortunately, all neural networks can be interpreted as having at least one layer of neurons and thus the `layerList` variable of type `ArrayList<ArrayList<NeuronConfig>>` is always valid.[6] The `GenericTopologyBuilder` abstract class allows developers to define their own subclasses that are responsible for setting up the connectivity matrix for each different type of neural network model.

The `GenericTopologyBuilder` utilises the *builder* design pattern to build neuron topologies. The parts that need to be constructed are the layers of `NeuronConfig` objects and their `Weight` objects. By varying the builder object used, the same `GenericTopology` class can represent any neural network topology (that can be expressed as a graph of course).

The main advantages of using the *builder* design pattern in this context are:

- **Easy component construction:** The logic of how to construct each type of neural network topology is encapsulated in one component. This logic includes

[6] See section 2.1.1 as well as [34] for a discussion on why all networks can be seen as having at least one layer of neurons.

setting up the network's net input signals and activation functions for each neuron in each layer, the weight connections between neurons, the number of layers, the relationship of neurons to input patterns, which neurons are output neurons and initial output values for each neuron. Using a builder object for each type of network cleanly separates each type of topology into separate components, making it easy to maintain or extend the system.

- **Flexibility:** A high degree of flexibility is achieved as it is easy to switch between different existing builder subclasses. This flexibility allows different types of topologies to be represented, without the need to change any topology code. The same `GenericTopology` class can be used to represent diverse network architectures as laid out in chapter 2 such as SOM networks, feedforward networks, Hopfield networks, Elman recurrent networks, functional link networks, LVQ networks, radial basis function networks, among others. Each type of topology is built using a different builder class specific to the type of network, that only needs to be defined once. Figure 5.8 shows examples of three builder objects for a SOM, feedforward network and Hopfield network respectively.

- **Separability:** Maintaining each topology is very easy, as different builder objects are used for each topology type – changes in one builder do not affect the others.

- **Dynamic construction:** The type of topology can be configured dynamically at runtime by providing the respective builder. This type of behaviour might be useful in scenarios involving growing or pruning of a network topology, or in ensembles.

- **Reuse:** A builder for a type of topology can be defined once and reused in many applications. This means that developers never have to 'reinvent the wheel' with respect to datastructures for neurons and weights each time they want to construct a new neural network implementation.

A segment of an XML setup document for the CILib simulator is shown below that utilises `GenericTopology`, `GenericTopologyBuilder` and `FanInWeightInitialiser` to setup a FFNN topology. More variables may be added of course, such as a list of observers that may be added to `GenericTopology`. The `FFNNgenericTopologyBuilder` class is a

specific implementation of a FFNN with sigmoid activation functions and defines exactly what types of neurons are to be created in each layer. A more generic implementation of a topology builder may of course be written that is capable of being configured to any degree.

```
<topology id="NNtopo" class="neuralnetwork.generic.LayeredGenericTopology">
  <topologyBuilder class="neuralnetwork.generic.topologybuilders.FFNNgenericTopologyBuilder">
     <prototypeWeight class="neuralnetwork.generic.Weight">
        <weightValue class="type.types.Real" real="0.5"/>
        <previousChange class="type.types.Real" real="0.0"/>
     </prototypeWeight>
     <addLayer value="5"/>
     <addLayer value="10"/>
     <addLayer value="3"/>
  </topologyBuilder>
  <weightInitialiser class="neuralnetwork.generic.topologyvisitors.FanInWeightInitialiser"/>
</topology>
```

5.2.2 A Generic Data Implementation

Another good example of a generic component is the `GenericData` class, as illustrated in figure 5.11. This class implements the `NeuralNetworkData` interface to provide a generic data source for neural network implementations. The main characteristic of `GenericData` is that it provides the four different data sets as discussed in section 2.2, namely a training set D_T, a generalisation set D_G, a validation set D_V, and a candidate set D_C. The class uses `ArrayList<NNPattern>` objects to represent patterns in each of these data sets. Using `NNPattern` allows any type of data pattern to be represented, as discussed in section 5.1.2.

`GenericData` allows patterns to be read in from any source such as flat text files, XML files, relational databases, data from a network location such as a web service, data streams from other applications, dynamically generated data, or any other source of data. The task of reading in and populating these data sets is not done in the `GenericData` class itself, but is delegated to a `DataDistributionStrategy` class (illustrated in figure 5.11). Classes that implement this interface read in patterns using the `java.io` package's `BufferedReader` object, allowing any data source that is implemented as a `Reader` to be used. Notice that `DataDistributionStrategy` does not specify that the

`BufferedReader` class *must* be used – developers are free to use any means necessary in the implementing classes. See [35] for more information on `java.io` and input streams.

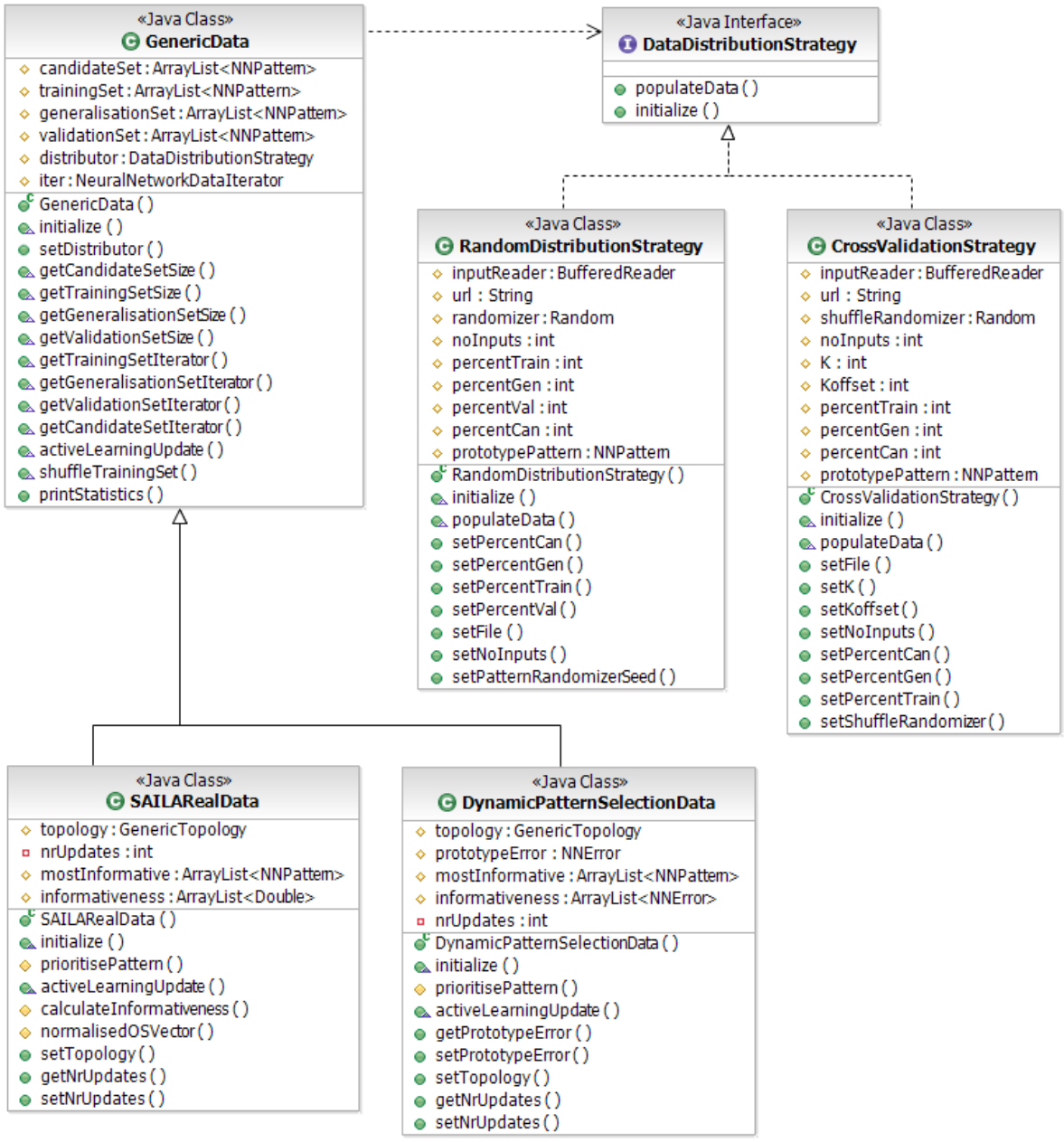


Figure 5.11: Generic data classes for neural networks

The `DataDistributionStrategy` interface uses the *strategy* design pattern to exter-

nalise and encapsulate the process of reading and populating data sets, allowing developers to be flexible in how they want to populate the four data sets in `GenericData`. Classes that implement `DataDistributionStrategy` need to provide a `populateData()` method that distributes data among D_T, D_G, D_V and D_C using a specific approach, such as static pattern distribution (i.e. the first 500 patterns to D_T, the next 200 to D_G and the last 200 to D_V), random pattern sampling based on percentages per data set, K-fold crossvalidation as discussed in section 2.2.1.6, or any other conceivable method of pattern distribution. Figure 5.11 illustrates `RandomDistributionStrategy` and `CrossValidationStrategy` as two examples of `DataDistributionStrategy` subclasses. As discussed in section 5.1.2, the *prototype* design pattern is used to allow distribution strategies to create new `NNPattern` instances without knowing the exact type of pattern used.

The `DataDistributionStrategy` allows developers to reuse the `GenericData` implementation by decoupling it from the data source type and the manner in which data sets are populated. An example of how to use this component in an XML simulator setup document is given as part of the XML in section 5.1.5.

5.3 Summary

This chapter outlined how the CILib framework in chapter 4 can be extended to support neural network implementations as outlined in chapter 2. This task is aided by design patterns as discussed in chapter 3. Using a design pattern to solve a problem yields advantages such as having a named problem/solution pair, a description on how to reuse the pattern, the consequences of using the pattern, recommendation on other patterns that complement the pattern, as well as advice and considerations on how to implement the pattern. This information was used to good effect on many occasions to solve many complex design challenges related to the development of a reusable generic neural network framework. The main advantages of using design patterns in the generic neural network framework are:

- Ease of use is dramatically enhanced as the framework gives developers exact guidance on how to construct a neural network implementation inside the CILib frame-

work.

- The system is extensible as more components can be added to the framework, allowing many different types of neural network components to be constructed and used inside the same framework.
- Flexibility is increased as any component can be replaced with a different component with ease, in most cases without code changes. This enables reuse of existing components to be very simple.
- Neural network information such as weights, neuron function parameters, learning algorithm parameters, data set information, and any metrics defined on a network can be easily accessed by external applications in a defined way.

The *topology* of a neural network includes neurons, the layers they are organised in, as well as the weight connections between them. It was shown that neural network topologies can be represented using the *strategy* design pattern which gives advantages such as encapsulation, simpler and more extensible mediation components and dynamicity. Furthermore, a generic topology implementation is discussed that is capable of representing any topology that can be expressed as a mathematical graph. This makes it easy for developers to use a well tested, existing topology implementation to easily and quickly define a neural network model. This implementation uses patterns such as the *strategy* which allows implementation to be switched easily, *builder* which allows multiple topology types to be defined once and reused, *iterator* to allow external components to access the individual neurons and weights of a topology, *visitor* to add more operations to the class dynamically, *observer* to notify all interested parties when events occur, and the *prototype* to create new `Weight` objects without knowing their type.

The *data* component is responsible for providing patterns to train a neural network model. Patterns are represented by the `NNPattern` interface, which allows any type of pattern to be defined. The *strategy* design pattern is used again to allow implementations to be varied. The *iterator* pattern is used to iterate over any of the mentioned data sets, allowing more than one iteration to be active at the same time and many types of iterations to be defined. Active learning is also supported. A generic data implementation is also developed that is capable of reading any patterns from any source and distribute

patterns among D_T, D_G, D_V and D_C using any distribution method. The *prototype* design pattern is used to allow patterns to be created without clients knowing its exact type.

The *learning* component is supported for both stand-alone and neural network specific optimisation techniques. Specialised algorithms that are specific to neural networks (such as Hopfield or SOM), and are not implemented as stand-alone algorithms, are catered for by the `TrainingStrategy` interface as well as the `NeuralNetworkController` and `NeuralNetworkProblem` classes. The *chain of responsibility* pattern is used to carry initialisation requests from `Algorithm` through the object structure to lower levels.

Mediation is required between the topology, data and learning components as these are not applications in their own right. Neural network models also have variables and concepts that stretch across these components and need to be represented. The `EvaluationMediator` abstract class provides the basic infrastructure that subclasses can use to implement mediator classes to represent specific neural network implementations.

The next chapter focuses on the implementation details of various components that facilitate active learning, using external learning algorithms to train neural networks, the provision of support for ensemble and modular networks, as well as an advanced metric for classification algorithm performance analysis.

Chapter 6

Advanced Neural Network Implementations

Nothing is a waste of time if you use the experience wisely.

— *Auguste Rodin*

The main objective of the generic neural network framework discussed in chapter 5 is to enable developers to rapidly create new neural network implementations in CILib. The emphasis is on reconfiguring and extending existing components to implement a desired neural network implementation quickly and efficiently. Ideally, developers only need to write code for components that do not exist yet rather than continuously rewriting the same logic for each new application.

The aim of this chapter is to show how easy the generic neural network framework is to use and extend. Various different types of neural network implementations and components are discussed, including

- active learning approaches such as the sensitivity analysis incremental learning algorithm (SAILA) and dynamic pattern selection (DPS);
- neural learning with other CI algorithms such as PSO or EC;
- ensemble network implementations;

- modular neural network topologies and how various parts of the topology can be trained with different algorithms;
- network pruning and growing algorithms; and
- a measure of classification performance, namely receiver operating characteristics (ROC) analysis, is implemented generically and used.

For each of the above, it is shown how most of the neural network implementation functionality is already defined in CILib. The new functionality that is required reuses and extends the existing components to build an entirely new neural network model. In addition, the new components are built in a generic way so that they in turn may be reused in future implementations as well.

The rest of this chapter discusses the details and implementation of the various components and algorithms outlined above.

6.1 Extending The Data Component

This section discusses the working and implementation of two active learning algorithms, namely the sensitivity analysis incremental learning algorithm (SAILA) by Engelbrecht [23] and dynamic pattern selection (DPS) by Röbel [91]. These two algorithms have the same goal of growing the training set to only include those patterns that the network can learn the most from. However, the algorithms achieve this goal in completely different ways – SAILA uses sensitivity information of a pattern's output with respect to its inputs to measure its informativeness, while DPS searches for the pattern with the largest error. Yet even with this drastically different behaviour, it is very simple to implement in the neural network framework in CILib using the same base components.

Fixed set learning is also mentioned briefly for comparison with the two active learning algorithms.

6.1.1 Fixed Set Learning

Fixed set learning (FSL), also known as Passive learning, was discussed in detail in section 2.2.2. In FSL, there is no pattern selection mechanism other than presenting the

Initialise weights and training parameters.
Construct D_T, D_V and D_G using sampling techniques from section 2.2.1.6.
repeat:
    Train the NN on D_T
    Shuffle training set D_T (optional).
until NN convergence is reached.

Algorithm 6.1: Outline of the FSL algorithm.

entire training set D_T to the neural network every epoch. In most FSL implementations the patterns in D_T are shuffled after every epoch to prevent bias. A pseudocode outline of the basic FSL algorithm is given in algorithm 6.1 – this serves as a base of comparison against SAILA in section 6.1.2 and DPS in section 6.1.3.

FSL uses the `GenericData` implementation as outlined in section 5.2.2 with no modification. Data is read into D_T once during initialisation and this set is used for the duration of neural learning.

6.1.2 Sensitivity Analysis Incremental Learning Algorithm

The sensitivity analysis incremental learning algorithm (SAILA) was developed by Engelbrecht [23] as a method to facilitate active learning. Typical supervised learning algorithms for multilayer neural networks involve training on a fixed set of patterns. FSL has a very interesting dilemma. On the one hand the data set needs to contain enough information (i.e. an adequate distribution across the search space) to solve the specific problem (such as a function approximation or classification problem). On the other hand, training time will be adversely affected if there are too many patterns in the training set. Uninteresting or duplicate patterns may also affect generalisation performance and lengthen training time [72], [122]. Worse, these redundant patterns might not be sampled uniformly across the space, thus over-emphasising areas in the search space, and thus biasing the learner.

Active learning tries to address these problems by using the network's current knowledge of the problem to select patterns from the available data that will yield the highest

decrease in training and generalisation error. As stated in section 2.2.3, Atlas *et al* define active learning as *any form of learning in which the learning algorithm has some control over what part of the input space it receives information from.* The learner uses current attained knowledge to select patterns from the candidate set, D_C, that are most likely to lead to a maximum decrease in error and adds these patterns to the training set, D_T.

SAILA conforms to the definition of active learning by using pattern sensitivity information to select those patterns in D_C that are most likely to solve the learning problem. The first-order derivatives of a pattern's output with respect to its input is indicative of the influence that pattern has on the approximated function's output – patterns with the highest influence or *informativeness* cause the largest weight changes during training, which in turn leads to faster conversion to a good solution.

The rest of this chapter illustrates how pattern informativeness is calculated for any given pattern, followed by a discussion of the SAILA algorithm.

Pattern Informativeness

Pattern informativeness is defined as follows [23]

Pattern Informativeness: *Define the informativeness of a pattern as the sensitivity of the NN output vector to small perturbations in the input vector. Let* $\Phi^{(p)}$ *denote the informativeness of pattern* d_p*. Then,*

$$\Phi^{(p)} \doteq \| \mathbf{S}_o^{(p)} \| \tag{6.1}$$

where $\mathbf{S}_o^{(p)}$ *is the output sensitivity matrix for pattern* d_p*, and* $\| \bullet \|$ *is any suitable norm.*

Engelbrecht [23] suggests the max norm,

$$\Phi^{(p)} \doteq \| \mathbf{S}_O^{(p)} \|_\infty = \max_{k=1,\cdots,K} \{ | S_{o,k}^{(p)} | \} \tag{6.2}$$

where $S_{o,k}^{(p)}$ is the sensitivity of a single output unit o_k to small perturbations in the input vector $\mathbf{z}$, with K being the total number of output units. The output sensitivity vector in equation 6.1 and equation 6.2 is defined as

$$\mathbf{S}_o^{(p)} = \| S_{oz}^{(p)} \| \tag{6.3}$$

where $\mathbf{S}_{oz}^{(p)}$ is the output-input layer sensitivity matrix for pattern $\mathbf{z}^{(p)}$. Assuming differentiable activation functions, each element $S_{oz,ki}^{(p)}$ of this matrix is computed as

$$S_{oz,ki}^{(p)} = \frac{\partial o_k}{\partial z_i^{(p)}} \tag{6.4}$$

For sigmoid activation functions (see equation 2.5), equation 6.4 (for a single pattern d_p) can be simplified to

$$\begin{aligned} S_{oz,ki} &= \frac{\partial o_k}{\partial z_i} \\ &= \frac{\partial o_k}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial z_i} \\ &= f'_{o_k} \sum_{j=1}^{J} w_{kj} f'_{y_j} v_{ji} \\ &= (1 - o_k) o_k \sum_{j=1}^{J} w_{kj} (1 - y_j) y_j v_{ji} \end{aligned} \tag{6.5}$$

Each element $S_{o,k}^{(p)}$ of the output sensitivity vector $\mathbf{S}_o^{(p)}$ can be calculated by using one of the following suitable norms suggested in [23], namely the sum-norm,

$$S_{o,k}^{(p)} = \| \mathbf{S}_{oz}^{(p)} \|_1 = \sum_{i=1}^{I} | S_{oz,ki}^{(p)} | \tag{6.6}$$

or the Euclidean-norm,

$$S_{o,k}^{(p)} = \| \mathbf{S}_{oz}^{(p)} \|_2 = \sqrt{\sum_{i=1}^{I} (S_{oz,ki}^{(p)})^2} \tag{6.7}$$

where I is the number of input units.

To summarise, a pattern $\mathbf{z}^{(p)}$ is considered informative if any number of output units are sensitive to perturbations in the input values of $\mathbf{z}^{(p)}$. Thus the larger $\Phi^{(p)}$ is, the more informative $\mathbf{z}^{(p)}$ is. $\Phi^{(p)}$ is computed by calculating the output-input sensitivity matrix $\mathbf{S}_{oz}$ of $\mathbf{z}^{(p)}$ for each input z_i and each output o_k. The sensitivity matrix $\mathbf{S}_{oz}$ is then used to calculate the sensitivity vector $\mathbf{S}_o$. The element of the vector $\mathbf{S}_o$ with the highest absolute value is set as $\Phi^{(p)}$.

The SAILA Algorithm

As an active learning algorithm, SAILA uses pattern informativeness as the measure to decide which patterns to include in the training set D_T. At specific *selection intervals* the pattern informativeness of all the patterns in the candidate set D_C is calculated. The pattern(s) with the highest informativeness are then removed from D_C and added to D_T by using the $\mathcal{A}^+_{SAILA}$ operator defined as

$$\mathcal{A}^+_{SAILA} = \{p \in D_C \mid \Phi^{(p)}_\infty = \max_{q=1,\ldots,P_C} \{\Phi^{(q)}_\infty\}; \forall q \in D_C, \text{not yet selected}\} \tag{6.8}$$

where P_C is the size of D_C at the current selection interval and $\Phi^{(p)}_\infty$ is defined in equation 6.1. Algorithm 6.2 gives a pseudo-code outline of the SAILA algorithm.

There are four design parameters that need to be considered when setting up SAILA, as discussed in [23]:

- **Initial training set size:** The initial training set consists of only one pattern, selected from the D_C using equation 6.8. This is to study SAILA under conservative conditions – more than one pattern may be selected if the researcher wishes to do so.

- **Subset selection criteria:** The SAILA operator in equation 6.8 chooses the most informative pattern.

- **Subset size:** The number of patterns to include in D_T after each selection interval is set to be one, thus being very conservative. More than one pattern can be included in the subset and the number of patterns to be selected may even be varied over time.

- **Interval termination criteria:** SAILA attempts to train on the current D_T until maximum gain is achieved before adding more patterns from D_C. Four termination criteria are used to prevent the learner from spending too much time training without sufficient gain or overfitting the current D_T. These are [23]:

  1. The number of epochs that the neural network is trained on a particular training subset is limited to 100. This ensures that the network does not train on the current D_T indefinitely.

Initialise weights and training parameters.
Set subset increase size = 1 (default value).
Construct initial D_T from D_C using equation 6.8.
repeat:
    repeat:
        Train the NN on D_T
    until a termination criterion is triggered.
    Compute the subset D_S to add to D_T:
        For each $\mathbf{z}^{(p)} \in D_C$, compute S_{oz} using equation 6.5.
        Compute $\mathbf{S}_o^{(p)}$ for $\mathbf{z}^{(p)} \in D_C$ using equation 6.3
        Compute $\Phi^{(p)}$ for each $\mathbf{z}^{(p)} \in D_C$ using equation 6.1.
        Let $D_S = \mathcal{A}^+_{SAILA}$ (equation 6.8).
        $D_T \leftarrow D_T \cup D_S$, $D_C \leftarrow D_C - D_S$
until NN convergence is reached.

Algorithm 6.2: Outline of the SAILA algorithm.

2. If there has been sufficient gain on the current D_T, a new subset is selected and added to D_T. A selection interval is triggered when the error on either D_T or the validation set D_V decreases by more than 80% (this number is configurable).

3. If the average decrease in error on either D_T or D_V is too small, a selection interval is triggered. This prevents the learner from training on a set D_T with too little gain. The threshold is set to 0.001 initially, and is divided by 10 as the order of magnitude of the error on D_T or D_V decreases. This threshold is configurable and other approaches may be taken to update it.

4. If the error ε_G on D_G increases too much, it is a sign that overfitting on the current D_T might be occurring. A selection interval is triggered as soon as $\varepsilon_G > \overline{\varepsilon}_G + \sigma\varepsilon_G$, where $\overline{\varepsilon}_G$ is the average error on D_G over all epochs and $\sigma\varepsilon_G$ is the standard deviation. Other overfitting measures may be used as well.

Initialise weights and training parameters. Set subset increase size = 1.
Construct D_T from D_C, using the pattern with the largest error to start.
repeat:
    repeat:
        • Train the NN on D_T
    until $\rho > 1.0$
    Compute the subset D_S to add to D_T:
        For each $d_p \in D_C$, compute the error on D_p.
        Let d_{p_+} be the pattern with the largest error.
        Let $D_S \leftarrow D_S \cup d_{p_+}$, $D_T \leftarrow D_T \cup D_S$, $D_C \leftarrow D_C - D_S$
until NN convergence is reached.

Algorithm 6.3: Outline of the DPS algorithm.

6.1.3 Dynamic Pattern Selection

The *dynamic pattern selection* algorithm (DPS) was developed by Röbel [91]. Röbel states that the aim of DPS is to effectively select the training set D_T during training by continually validating the generalisation properties of the neural network.

Formally, the aim of DPS is to incrementally grow the training set $D_T \subset D_C$ in such a way that the training error ε_T is minimised to allow the network to converge faster to F_D within a specified level of accuracy, in other words $\|F_{NN} - F_D\| < \tau$. The two main questions that DPS need to address are: when should new patterns be added to D_T and which patterns should be chosen?

Röbel's answer to deciding when new patterns should be added is to increase the training set by a single pattern when the generalisation properties of the network becomes poor. In other words when the validation error ε_V of the network is worse than the training error ε_T, the training set should be increased. For example, when using the MSE metric, this requirement can be measured easily by using the *generalisation factor,* ρ, defined as

$$\rho = \frac{\varepsilon_V}{\varepsilon_T} \tag{6.9}$$

For good generalisation performance, it is required that $\rho \leq 1.0$.

The process of selecting a pattern from D_C to add to D_T should have as little overhead as possible. Röbel suggests adding that pattern from D_C that gives the highest error when evaluated by the network. Algorithm 6.3 gives an outline of how DPS works.

6.1.4 Implementation

SAILA and DPS are implemented by a combination of data source and mediator constructs. Both the `SAILARealData` and `DynamicPatternSelectionData` classes extend the `GenericData` class, as can be seen in figure 5.11. These classes provide all the functionality of the respective algorithms. The `activeLearningUpdate()` method is used by client classes such as mediators to extract a pattern from D_C and insert it into D_T. The `prioritisePattern()` method is invoked by `activeLearningUpdate()` to determine which pattern to include in D_T. In `SAILARealData` this method proceeds to calculate the pattern informativeness of all patterns in D_C as an `ArrayList<Double>` array according to the algorithm and approach in section 6.1.2. The most informative pattern is then selected to move from D_C to D_T. For `DynamicPatternSelectionData`, the method calculates the error (specified as a prototype, so any error metric may be used) on all patterns in D_C, stores it in a member of type `ArrayList<NNError>` and chooses the highest one, as outlined in section 6.1.3.

The use of `GenericData` has many advantages. All the basic functionality from `GenericData` is reused and extended to facilitate the two new algorithms. It allows any implementation of `NNPattern` to be reused, as this is the type that `GenericData` uses. SAILA and DPS data sources are also decoupled from any specific target, as `DataDistributionStrategy` can be reused to read in data from any source. It also allows SAILA and DPS to be used by any neural network implementation that is configured to work with `NeuralNetworkData`.

Both `SAILARealData` and `DynamicPatternSelectionData` respectively implement all the logic of *how* to do an active learning update, but not *when* to perform it. These mediators need to determine the context in which to call `activeLearningUpdate()`, taking many different aspects regarding the state of epoch execution into consideration, as based on the respective algorithm. SAILA has four interval termination criteria as discussed in section 6.1.2, while DPS has only one, namely the *generalisation factor* ρ. This

information needs to be stored in each mediator and is calculated while the network is being trained. The two different mediator implementations need to be implemented (one for SAILA and one for DPS) that loosely follow the same structure, but use vastly different algorithmic approaches. Of course, it seems that the mediator component itself is a prime candidate for the *template method* design pattern, which stipulates that only one mediator implementation, say `ActiveLearningTemplateMediator`, is defined. The respective algorithmic steps in algorithms 6.2 for SAILA and 6.3 for DPS respectively that coincide logically (i.e. termination criteria, subset selection, and training of the network) are then delegated to subclasses of the template `ActiveLearningTemplateMediator` class. While this approach seems to be a cleaner approach, each type of active learning algorithm is still a subclass of a mediator abstract class (`ActiveLearningTemplateMediator` in this example), so little benefit is gained by using the *template method* pattern for just SAILA and DPS. If many different active learning algorithms reuse the proposed `ActiveLearningTemplateMediator` class, it may be worthwhile adding the complexity of the *template method* design pattern to reduce the rework in having a separate mediator for each active learning algorithm.

6.2 Extending The Learning Component

This section provides more detail on how other stand-alone CI algorithms such as a PSO or an EA can be used to train neural networks. Recall from section 2.3.1 that a PSO and an EA both use a similar approach to train a neural network.

When using PSO to train neural networks, the position vector $\mathbf{x}_i(t)$ of each PSO particle i represents the weight vector $\mathbf{W}_i(t)$ of a neural network i at time t. When using an EA, each chromosome i in the population represents the neural network weight vector $\mathbf{W}_i(t)$ at time t. The performance measure, $\mathcal{F}$, for both PSO and EA comprises of the chosen objective function for the neural network. During the fitness evaluation of each particle or chromosome, this weight vector $\mathbf{W}(t)$ needs to be obtained from the particle or chromosome and inserted into a network topology to be able to calculate the fitness of the specific network. The fitness calculation itself can take many forms and is dependant on the type of network that is to be trained. In the case of a feedforward

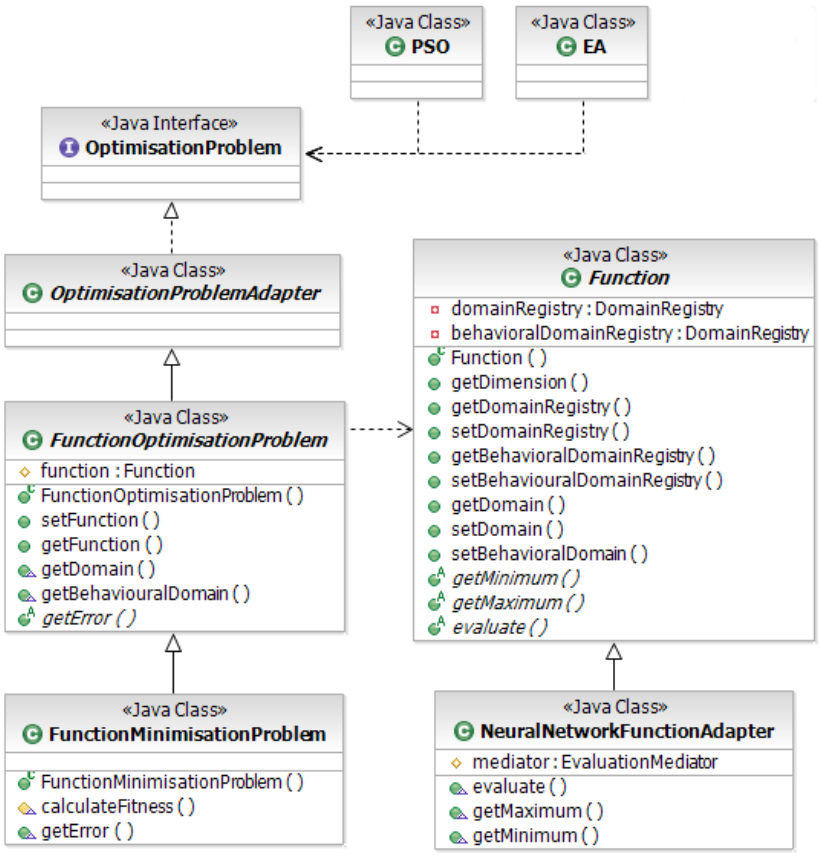


Figure 6.1: Using an adapter with CILib `Algorithm` implementations.

neural network, the fitness value may be the MSE of all patterns in D_T, but a fitness value is not restricted to the MSE alone (see section 2.3.1). The fitness function may also be a combination of error metrics over D_T, D_G and D_V for example.

In the case of PSO, the weights of all neural networks are then updated simultaneously by adjusting the positions of all particles using equation 2.25 (as is the case when using standard PSO – different PSO approaches may use different update strategies). An EA approach may use various selection, cross-over, and mutation operators to create or change individuals in the population (as dictated by a specific EA algorithm).

In order to use a PSO or an EA to train a neural network, some integration needs to be performed to allow the PSO or EA to interpret a neural network error metric as a performance measure, $\mathcal{F}$. In particular, the neural network mediator needs to be

instructed to compute a desired error metric over one or more data sets. This result must then be returned to the PSO or EA as the fitness of the particle or chromosome respectively. The PSO or EA system provides the `Algorithm` implementation that CILib requires, which needs the neural network error metric to be translated into a function that can be used as a performance measure $\mathcal{F}$ by a PSO or an EA. Both PSO and EA use subclasses of `Function` to represent performance measures, which means that `NNError` metrics need to be interpreted as `Function` interpretations. The `Function` class is illustrated in figure 6.1.

The *adapter* design pattern is well suited to act as a broker between the PSO or EA system and the neural network system. By using the *adapter* design pattern, a `NeuralNetworkFunctionAdapter` class that implements the CILib `Function` interface can obtain `NNError` metrics from a neural network mediator and convert it to a format that a PSO or EA algorithm can understand. The `NeuralNetworkFunctionAdapter` class is illustrated in figure 6.1. The `NeuralNetworkFunctionAdapter` class's `evaluate()` method maps the inbound parameter `in` (which represents the information in a PSO particle or EA chromosome) to the neural network topology's weight vector via the `NeuralNetworkTopology` class's `setWeights()` method. The topology reference needs to be obtained by using the mediator interface. The code for this is shown below.

```
public Double evaluate(Object in) {
  mediator.getTopology().setWeights((MixedVector)in);
  mediator.performLearning();
  return mediator.getErrorDt()[0].getValue();
}
```

As the `NeuralNetworkFunctionAdapter` class extends the `Function` interface, any `FunctionMinimizationProblem` can now accept the `NeuralNetworkFunctionAdapter` class as the function to be minimised. This entails that any subclass of `Algorithm` that has the capability to work on `FunctionMinimizationProblem` objects (of which PSO and EA are examples), can now be used to perform neural learning. Any 'suitable' neural network mediator may be used. Suitable mediators include mediators that can return valid `NNError` metrics and that do not explicitly invoke other training methods such as gradient descent. After the weight vector is set by the `NeuralNetworkFunctionAdapter`

class, the mediator class's `performLearning()` method is called, which in turn invokes the `learningEpoch()` method (using the *template method* pattern as discussed in section 5.1.4). The `learningEpoch()` method of the mediator class iterates over various data sets such as D_T, D_G or D_V, obtains the output of each pattern, and calculates the relevant `NNError` metrics that have been set up in the mediator. Recall from section 5.1.4 that the mediator component uses the *prototype* pattern to allow any number of `NNError` metrics to be dynamically assigned to a mediator. At the end of the epoch, the error is finalised as explained in section 2.3.1. The code below shows the `learningEpoch()` method of a mediator that returns the error over D_T only.

```
protected void learningEpoch() {
  //recreate metrics using prototype pattern
  resetError(errorDt);
  setErrorNoPatterns(errorDt, data.getTrainingSetSize());
  iteratorDt = data.getTrainingSetIterator();

  //iterate over each pattern in training data set
  while(iteratorDt.hasMore()){
      MixedVector output = topology.evaluate(iteratorDt.value());
      this.nrEvaluationsPerEpoch++;
      computeErrorIteration(errorDt, output, iteratorDt.value());
      iteratorDt.next();
  }
  finaliseErrors(errorDt);
}
```

An example of an XML setup document for the CILib simulator that can be used to set up a PSO that trains neural networks using the `NeuralNetworkFunctionAdapter` is shown below. The `<algorithm>` tag is used to specify the PSO as training algorithm and all its related parameters. The `<problem>` tag is where the neural network is declared as a function to be minimised by being interpreted as a `FunctionMinimisationProblem`. This allows the `NeuralNetworkFunctionAdapter` to be used as the `Function` parameter for the CILib problem.

```
<simulator>
  <algorithm id="msa-pso" class="pso.PSO" particles="30">
    <topology class="entity.topologies.VonNeumannTopology"/>
    <prototypeParticle class="pso.particle.StandardParticle">
      <velocityUpdateStrategy class="pso.velocityupdatestrategies.StandardVelocityUpdate" />
      <inertiaComponent class="PSO.StandardInertia">
         <inertia value="0.6" />
      </inertiaComponent>
      <socialComponent class="PSO.StandardAcceleration">
         <acceleration value="1.4" />
      </socialComponent>
      <cognitiveComponent class="PSO.StandardAcceleration">
         <acceleration value="1.4" />
      </cognitiveComponent>
    </prototypeParticle>
    <addStoppingCondition class="stoppingcondition.MaximumIterations" iterations="150" />
  </algorithm>

  <problem class="problem.FunctionMinimisationProblem">
    <function class="functions.NeuralNetworkFunctionAdapter" />
      <evaluationMediator class="neuralnetwork.generic.evaluationstrategies.FFNNEvaluationMediator">
         <topology id="NNtopo" class="neuralnetwork.generic.LayeredGenericTopology">
            <topologyBuilder class="neuralnetwork.generic.topologybuilders.FFNNgenericTopologyBuilder">
               <prototypeWeight class="neuralnetwork.generic.Weight">
                  <weightValue class="type.types.Real" real="0.5"/>
                  <previousChange class="type.types.Real" real="0.0"/>
               </prototypeWeight>
               <addLayer value="5"/>
               <addLayer value="10"/>
               <addLayer value="3"/>
            </topologyBuilder>
            <weightInitialiser class="neuralnetwork.generic.topologyvisitors.FanInWeightInitialiser"/>
         </topology>
        <data id="dat" class="neuralnetwork.generic.datacontainers.GenericData">
          <distributor class="neuralnetwork.generic.datacontainers.RandomDistributionStrategy">
            <file value="c:/data/IrisScaled.txt"/>
            <noInputs value="4"/>
            <percentTrain value="80"/>
            <percentGen value="5"/>
            <percentVal value="15"/>
            <percentCan value="0"/>
          </distributor>
        </data>
        <addPrototypError class="neuralnetwork.generic.errorfunctions.MSEErrorFunction" noOutputs="3"/>
      </evaluationMediator>
  </problem>
```

```
  <measurements id="measurements" class="simulator.MeasurementSuite" samples="1" resolution="25">
    <addMeasurement class="measurement.single.Fitness" />
    <addMeasurement class="measurement.single.Solution" />
    <addMeasurement class="measurement.single.FitnessEvaluations" />
  </measurements>

  <simulation>
    <algorithm idref="msa-pso" />
    <problem idref="msa" />
    <measurements idref="measurements" file="test5-results-msa-pso-with-gaps.txt" />
  </simulation>
</simulator>
```

6.3 Extending The Topology Component

This section looks at some advanced configurations and operations in the topology component of a neural network implementation. Modular and ensemble network implementations are discussed, followed by a brief look at how growing and pruning of neurons and weights in topologies can be achieved.

6.3.1 Modular and Ensemble Networks

Modular neural networks (MNN) and ensembles of networks were discussed in section 2.4.4. It was shown that an ensemble network can be treated as a special kind of MNN in which no allocator is present. Bearing this in mind, there are design decisions that need to be addressed when implementing MNN topologies using CILib:

- A MNN is a set of network modules that are in turn connected to form a larger network structure. This network may also be seen as a module in yet another MNN, so recursive composition has to be defined for more than two levels. This entails that recursion needs to be supported on the neuron level.

- Modules need to communicate with other modules by receiving input and providing output. Both inputs and outputs can be multidimensional and need to be addressed when interpreting modules as neurons in the top-level network. Some transformation will need to take place to convert neuron inputs from the top-level

network to network inputs that the module can understand. Weight values between modules also need to be mapped to the correct level inside the modules.

- As each module is a neural network in its own right, each module needs to be constructed using a builder object. All the modules in turn need to be composed into the top-level network.

- Training may need to be performed on modules in the MNN. The framework needs to support this in an easy to use and efficient manner.

The `GenericTopology` discussed in section 5.2.1 is designed to directly support the construction and use of MNN topologies. Recall that the `GenericTopology` uses the `NeuronConfig` class to represent processing elements in a network. In typical neural network topologies, these are neurons such as the bias unit, summation units, product units, SOM units, or any other type of neuron. The `NeuronConfig` class provides various abstract methods of which `computeOutput()` is one, as can be seen in figure 5.9. Subclasses need to implement these methods to provide the behaviour of the desired neuron type. Weights are represented by the `Weight` class which uses the CILib `Type` package to represent weight values.

As alluded to above, MNN implementations have at least two logical layers in the network topology consisting of the various modules and the top-level MNN topology. The implementation of the individual modules is directly supported by the `GenericTopology` class as these are merely standard neural network topologies. The entire topology is contained in a single `GenericTopology` object instance. The top-level MNN topology also needs to be implemented as a topology with 'neurons' that are in fact neural networks themselves (see figures 2.16 and 2.17). The `GenericTopology` class can implement this type of topology directly if an implementation of `NeuronConfig`, say `NeuronConfigMNN`, is able to represent an entire network topology (i.e. a module) as a neuron (see figure 6.2).

Each `NeuronConfigMNN` object in the top-level MNN topology needs to act as a wrapper for a module, where each module is an independent instance of `GenericTopology`. The `computeOutput()` method of `NeuronConfigMNN` has the task of converting a pattern evaluation request from the top-level MNN topology to a call to the wrapped

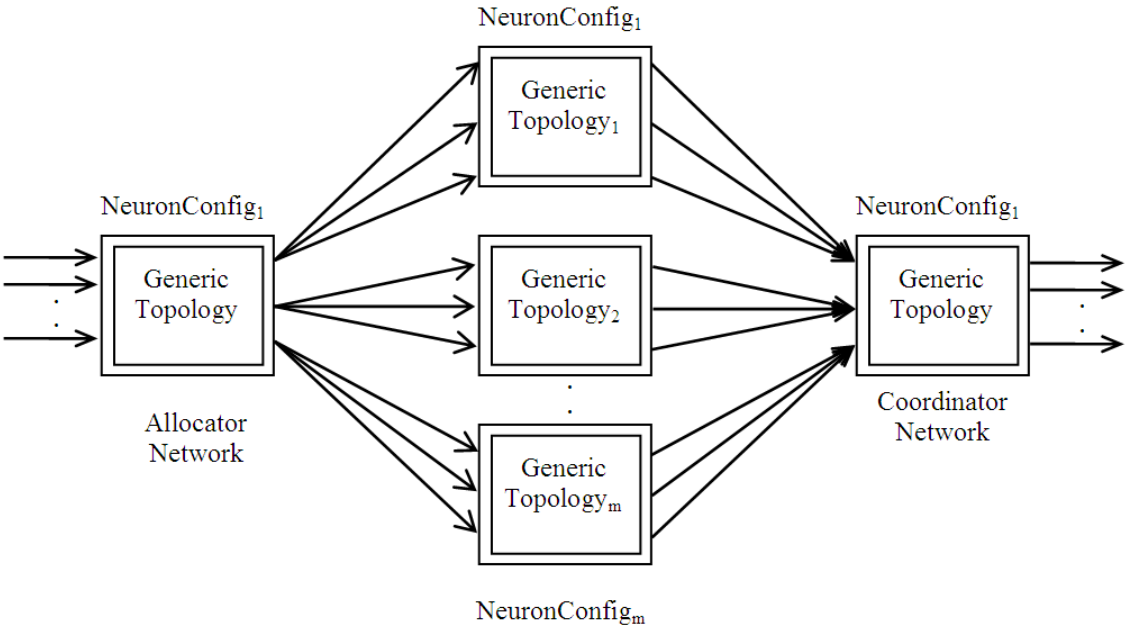


Figure 6.2: `NeuronConfigMNN` wrapping `GenericTopology` instances in a MNN topology.

`GenericTopology` instance's `evaluate()` method. Bear in mind that a module may receive inputs from multiple other modules. An example is the coordinator as discussed in section 2.4.4 which has to combine the outputs of all the other modules in the MNN. A module may also receive only part of the output of another module as input, such as an allocator that has, say, a outputs and a module that needs b inputs from the allocator, where $a > b$. The `computeOutput()` method of `NeuronConfigMNN` has the responsibility to integrate and transform these various inputs and weight values to be usable by the encapsulated `GenericTopology`.

The *adapter* pattern is used to integrate `NeuronConfigMNN` and `GenericTopology`. The `NeuronConfig` class provides a number of member variables that implement a processing unit such as a neuron. The full list can be seen in the UML in figure 5.9. The ones that are applicable to the desired wrapper are:

- `input` of type `NeuronConfig[]` which is the list of other `NeuronConfig` processing units that provides input to this unit,
- `inputWeights` of type `Weight[]` which is the corresponding weight values for the

`input` variable,

- `currentOutput` of type `Type` which is the output of this processing unit, and
- `patternInputPos` and `patternWeight` are also needed if input to the module is directly from a pattern.

Firstly, it should be noticed that the `NeuronConfigMNN` object's `currentOutput` may be multi-dimensional, as the wrapped topology's output is not restricted to being one-dimensional. Similarly, each `Weight` object in the `inputWeights` list may be multi-dimensional, depending on the topology.

The top-level MNN topology neurons use `input` as well as `inputWeights`, and possibly `patternInputPos` and `patternWeight` (if input is received directly from the pattern) as variables, but the wrapped `GenericTopology` object's `evaluate()` method expects a `NNPattern` object. This conversion needs to be done in the neuron's `computeOutput()` method. To do this, the output of each module listed in the `input` variable needs to be consolidated into a single `MixedVector` object called `inputMNN`, which will in turn be used to create a `NNPattern` object.

The top-level MNN weights also need to be taken into account by the topology that is wrapped by each `NeuronConfigMNN` object. The `inputWeights` variable (and possibly `patternWeight` if it is used) also needs to be consolidated into a single `MixedVector` object called `weightsMNN` which represents the weight connections between modules. This `weightsMNN` variable is then used to set the weights inside the topology accordingly.

An example of the `computeOutput()` method of `NeuronConfigMNN` is given below. The code iterates over all the `NeuronConfigMNN` objects in the `input` variable, as these represent other modules in the MNN. Not all of the outputs are relevant though (i.e. only a subset of the outputs of another module are needed), so only those with non-null weight connections need to be included in the `inputMNN` vector. The `PatternWeightSetVisitor` extends `GenericTopologyVisitor` and has the responsibility to set the weights (specifically the `patternWeight` objects in each `NeuronConfig` in the wrapped topology) correctly and throw an exception if there are any problems such as a dimension mismatch. Once the `inputMNN` vector is finalised, an instance of `NNPattern` needs to be created that acts as the vessel for the `inputMNN` vector to be presented to the topology. Once

this is done, the `evaluate()` method of the wrapped `GenericTopology` is invoked and the output is returned as the output of the module. This output can be used by other `NeuronConfigMNN` modules. The final output of the MNN itself is also a CILib `Type` object. By setting the boolean `isOutputNeuron` to true in the relevant modules, the output of the MNN is collated in the top-level `GenericTopology` class in the same manner as for any other topology.

```
public Type computeOutput(NeuronConfig n, NNPattern p) {
  MixedVector inputMNN = new MixedVector();
  MixedVector weightsMNN = new MixedVector();

  for (int i = 0; i < this.input.length; i++){
    MixedVector tmpInput = ((MixedVector)input[i].getCurrentOutput());
    MixedVector tmpWeight = ((MixedVector)inputWeights[i].getWeightValue());

    //copy only relevant weights and inputs to vector
    for (int j = 0; j < tmpInput.size(); j ++){
      if (tmpWeight.get(j) != null){
        inputMNN.add(tmpInput.get(j));
        weightsMNN.add(tmpWeight.get(j));
      }
    }
  }
  PatternWeightSetVisitor v = new PatternWeightSetVisitor(weightsMNN);
  topology.acceptVisitor(v);
  NNPattern pat = new StandardPattern();
  pat.setInput(inputMNN);

  return topology.evaluate(pat);
}
```

From a performance point of view, note that the code above essentially only copies references into the new variables `inputMNN` and `weightsMNN`. If the topologies are static (i.e. no neurons or weights are added or removed), these variables may be configured

beforehand during an initialisation step. This may improve performance in networks with a large number of weights on the MNN level network.

As for any implementation that uses the `GenericTopology` class, the entire MNN structure needs to be built using the *builder* design pattern. There are 2 levels of construction:

- build the top-level MNN topology using an appropriate builder, and
- build each module using an appropriate builder.

The latter case is very easy – each module is a 'typical' neural network topology such as a FFNN. The existing `GenericTopologyBuilder` builders such as those in figure 5.8 can be reused as they are. No modification is needed as the fully constructed `GenericTopology` instance is merely wrapped in an appropriate `NeuronConfigMNN` object, as shown in figure 6.2.

The MNN topology itself is a new kind of topology and thus requires a new builder implementation that extends `GenericTopologyBuilder`, say `MNNTopologyBuilder`. The layout of the `NeuronConfigMNN` objects has to follow the appropriate architecture layout, such as the structures in section 2.4.4. The wrapped `GenericTopology` member variable of each `NeuronConfigMNN` object is then initialised by invoking an appropriate builder as explained above. A list of builder objects has to be provided to the `MNNTopologyBuilder` when it is constructed, which can be done using the CILib XML interface. Note that weights for each module can be set using the `GenericTopologyVisitor` functionality provided by `GenericTopology` such as random weights or a specific weight vector, as illustrated in figure 5.10.

6.3.2 Architecture Selection

This section briefly discusses how architecture selection operators such as growing and pruning of neurons and weight connections can be used with the `GenericTopology` class. Growing and pruning of network elements form part of the ontogenic function which is part of the neural learning paradigm, as discussed in section 2.3.5.

Recall from section 5.2.1 that `GenericTopology` uses a graph-like approach to implement neurons using the `NeuronConfig` class. Every column in the graph's connectivity

matrix in figure 5.7 constitutes a single neuron that is represented by a `NeuronConfig` object. A fan-in centric view of the neuron is taken, meaning that a `NeuronConfig` object is responsible for obtaining the information it needs to calculate its own output. All the non-null entries in the connectivity matrix indicate the input neurons for the object. The `GenericTopology` class also provides a `LayerIterator` interface which can be used to gain access to and modify the neuron and weight connections structure.

Recall from figure 5.9 that each `NeuronConfig` object holds information such as

- `currentOutput` of type `Type`, used to store the output of the neuron;
- `input` of type `NeuronConfig[]`, which is a list of all input neurons;
- `inputWeights` of type `Weight[]`, which is the corresponding weight values for `input`;
- `patternInputPos` of type `int`, used to indicate which location in the pattern is used by this neuron;
- `patternWeight` of type `Weight`, the corresponding weight for `patternInputPos`;
- `timeStepMap` of type `boolean[]`, used to indicate whether to use input[i]'s current output (a value of zero) or timestep t-1 output (a value of one)(this is used mostly in recurrent architectures with self connections);
- `Tminus1Output` of type `Type`, used to keep track of the neurons's previous output. This is needed in cases where *limiting* (also known as *clamping*) is used to revert to the neuron's previous output value as opposed to using the newly computed value (see section 2.1.1); and
- `isOutputNeuron` of type `boolean`, used to indicate to the topology if this is an output neuron.

The above variables are not always used and default values can be used when constructing new objects. The *prototype* pattern (which is supported by `NeuronConfig`) will be very useful to allow any growing component to create copies of configured neurons. Using the *prototype* means that any particular architecture selection algorithm is

independent of the type of weights, activation functions, and other neuron details – a pre-configured prototypical instance is used to create more copies without the algorithm having to know how to create neuron objects.

In order for the `GenericTopology` class to support growing and pruning functionality, the following needs to be taken into account:

- it must be possible to add or remove `NeuronConfig` objects in any layer in a seamless fashion without invalidating the topology model, and
- it must be possible to add or remove `Weight` objects from specific `NeuronConfig` objects without invalidating the topology model.

The `LayerIterator` interface makes it extremely easy to navigate to any point in the `GenericTopology` network structure. Since the `layerList` variable and each layer in `GenericTopology` is implemented as a Java `ArrayList`, it is extremely easy to add or remove elements in any location. `NeuronConfig` uses a fan-in centric approach, which means that each neuron is responsible for knowing where all its inputs come from and what the weight values are – all information about the neuron is contained in a single `NeuronConfig` object.

To add or remove a weight is a simple operation. The relevant `NeuronConfig` object must first be located in the layer. Note that Java arrays are used for the `input` and `inputWeights` variables for performance reasons as the functionality to calculate a neuron's output is used very frequently – object dereferences (such as those in the `ArrayList` class) are kept to a minimum by using first class Java variables. A manual resizing of the arrays will be needed before an addition or after the removal of a weight. To remove a weight, the relevant `Weight` object must be found in the `inputWeights` array of the neuron object and removed, along with the matching `NeuronConfig` object in the `input` array (i.e. the associated input from another neuron for the specific weight). Similarly, to add a weight, a new `Weight` object must be added to the `inputWeights` array and a reference to the correct `NeuronConfig` object must be added to `input` at the same index.

To add a new neuron to a layer, a `NeuronConfig` object needs to be constructed to represent the new neuron. As mentioned above, the *prototype* pattern is very useful here

to avoid manual instantiation of neurons each time. Using the *prototype* pattern also makes it possible to reuse a particular growing algorithm with any type of neuron. The new object is then added at any location in the desired layer (which is an `ArrayList` object). The final step is to update any other `NeuronConfig` objects in the topology that need to have the new neuron as part of their input by adding relevant weights.

To remove a neuron follows a similar approach. The relevant `NeuronConfig` object needs to be located in the layer, a reference must be kept, all the other `NeuronConfig` objects in the topology that refer to it must be updated by having the reference and weight removed from `input` and `inputWeights` variables, and finally the `NeuronConfig` object itself must be removed using the `ArrayList` interface.

When a structural change is made to a `GenericTopology` object, it may possibly invalidate other components in the application. Examples include `NeuralNetworkData` implementations that need to be modified if input neurons are removed, neural learning components that need to be notified if structural changes occur, `NNError` objects that need to be informed if more or fewer output units are present, and PSO particles' lengths will change if any weights are removed. The *observer* pattern is very important in this situation. It allows components such as those mentioned above to register themselves with the `GenericTopology` as observers. Whenever a change in the topology occurs, the components on this list are notified. They may in turn inspect the topology and decide whether to continue or raise an exception.

The paragraphs above highlight *how* to perform the operations related to growing and pruning a network, but not *where*. As growing and pruning falls under the neural learning paradigm, the functionality may be implemented as a subclass of `TrainingStrategy` as discussed in section 5.1.3. Different approaches should each have their own subclass of `TrainingStrategy` for their implementations. Subclasses of `EvaluationMediator` need to orchestrate how and when architecture selection algorithms need to be invoked, as well as provide any required information via the `Object` parameter of the training strategy's `invokeTrainer()` method.

6.4 Extending CILib Functionality

This section looks at extending the CILib framework with the receiver operating characteristics (ROC) measurement, which is aimed at measuring classifier performance. Its outcome is a simple result but its derivation involves complex algorithmic calculations. It is shown how easily a measure such as ROC can be implemented and reused by any CILib implementation.

6.4.1 Receiver Operating Characteristics (ROC)

ROC is a measure of how well a classifier performs given a classification problem. ROC analysis has been used as early as 1989 to compare machine learning algorithms [106]. ROC analysis has a number of advantages as a classification performance measure:

- It has been shown that ROC is not biased by skewed class distributions or unequal classification costs [31].
- ROC analysis makes it easy to visualise classifier performance.
- An inherent characteristic of the ROC surface is that it can show whether a classifier performs better or worse than random guessing (something that an accuracy metric alone cannot show conclusively).
- ROC curves give researchers a method to assess continuous classifier performance across all chosen threshold values at once (as a specific threshold or boundary needs to be set for a continuous classifier, such as viewing all values above 0.5 as 1 and all values below 0.5 as 0)[1]. The same classifier system (such as a trained neural network for instance) can have radically different conservative or liberal behaviour depending on the selected threshold value.
- The performance of different classifiers can be compared at different threshold values.

[1]The concept and effect of a threshold is discussed in subsection 6.4.1.3

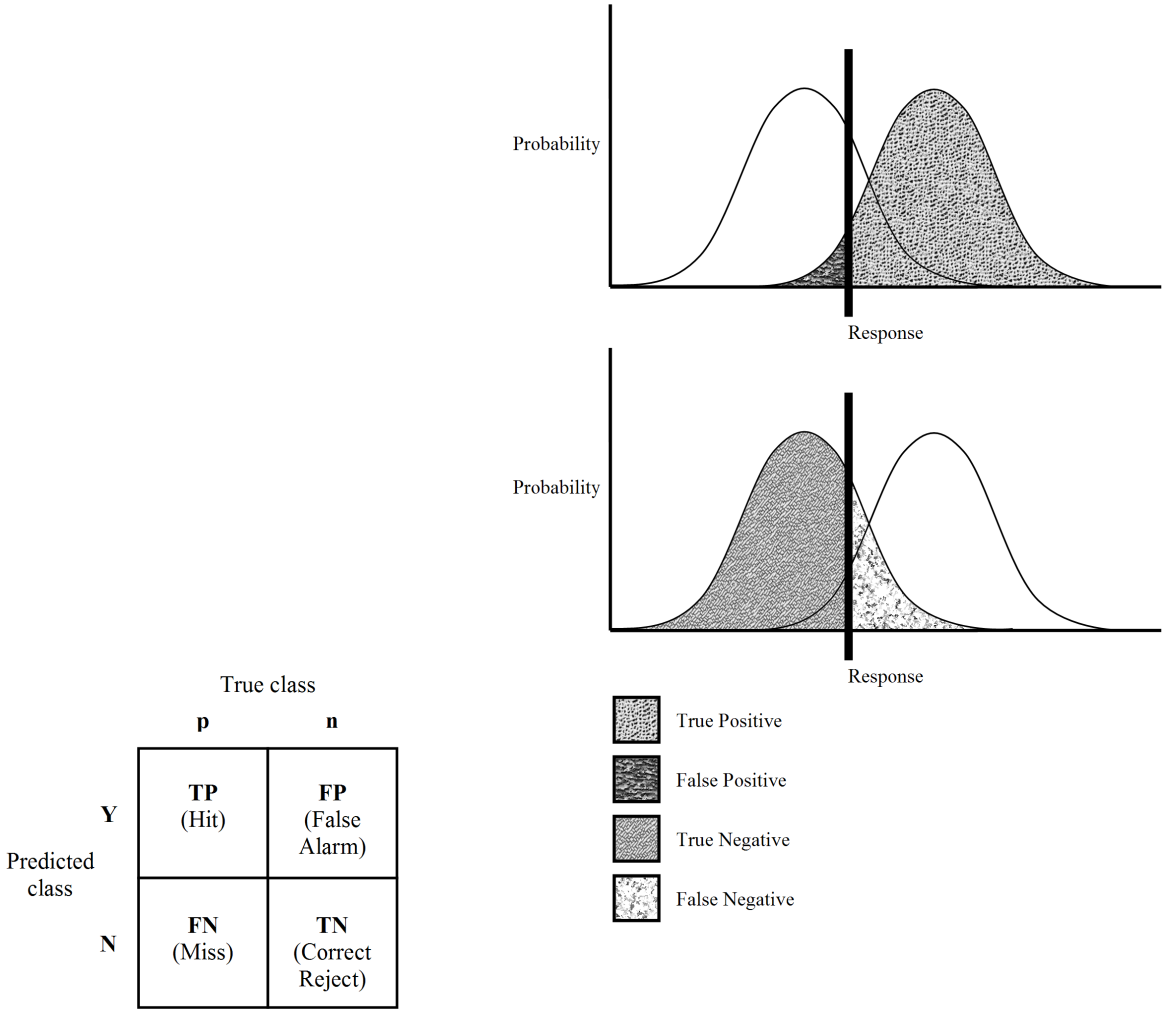


Figure 6.3: An example of how a confusion matrix is determined.

6.4.1.1 Classifier Performance

As discussed in section 2.2, the aim of a classifier is to map an input vector $\mathbf{z}_p$ to a particular class, where there are at least 2 or more classes. Once a classifier is associated with a data set, there are four possible outcomes from any single pattern classification (based on two classes): a positive pattern that is classified as positive is called a *true positive* (TP); a negative pattern that is classified as positive is called a *false positive* (FP); if a negative pattern is classified as a negative it is called a *true negative* (TN); if

a positive pattern is classified as a negative it is called a *false negative* (FN). The TP and TN together indicate all correct decisions, while the FP and FN together represent the errors made.

The relationship between TP, FP, TN and FN is illustrated in figure 6.3 in a *confusion matrix*. In each of the graphs, the right bell curve is the distribution of positive patterns, while the left curve is that for negative patterns. The dark vertical line in each graph denotes the classification threshold value – all values to the left of this line is associated with the negative class and all values to the right of this line is associated with the positive class. For discrete classifiers (i.e. classifiers that give a categorical class output such as 0 or 1, or 'positive' and 'negative') the threshold value can be fixed, but for continuous classifiers (classifiers such as neural networks that produce output that falls in a range, say $[0, 1]$), the threshold value can be changed. For example, if the threshold value is set to 0.6, all classifier output greater than 0.6 are regarded as positive classifications and all output below 0.6 as negative.

There are a number of metrics that can be calculated from these TP, FP, TN and FN, such as:

$$\begin{aligned} TP_{rate} &= \frac{\text{Positives classified correctly}}{\text{Total positives}} \\ &= \frac{TP}{\mathcal{P}} \end{aligned} \tag{6.10}$$

$$\begin{aligned} FP_{rate} &= \frac{\text{Negatives classified incorrectly}}{\text{Total negatives}} \\ &= \frac{FP}{\mathcal{N}} \end{aligned} \tag{6.11}$$

$$\text{Accuracy} = \frac{TP + TN}{\mathcal{P} + \mathcal{N}} \tag{6.12}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{6.13}$$

where $\mathcal{P}$ is the total positive patterns and $\mathcal{N}$ is the total negative patterns in the data set as in figure 6.3.

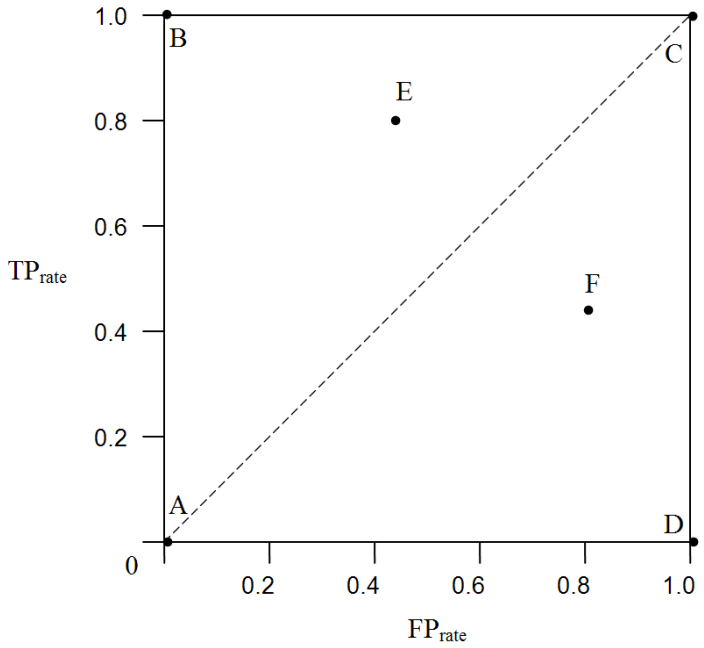


Figure 6.4: An example of an ROC graph

6.4.1.2 ROC Graphs

An ROC graph plots the TP_{rate} in equation 6.10 against the FP_{rate} in equation 6.11 to give a visual representation of benefits (a high TP_{rate}) and costs (a high FP_{rate}). Every classifier[2] produces a (TP_{rate}, FP_{rate}) pair that can be plotted on the ROC surface. An example of an ROC surface is shown in figure 6.4.

There are some interesting points in figure 6.4. Point A represents the strategy of never classifying a pattern as belonging to the positive class – all negative patterns are mapped correctly and no positives are correct. This is the most 'conservative' part of the ROC surface. The opposite (or 'liberal') end is located at point C, which represents the strategy of *always* issuing a positive classification. This approach never classifies a positive pattern incorrectly, but always classifies all negative patterns wrong. The dashed line from (0,0) to (1,1) represents random guessing – the TP_{rate} and FP_{rate} are always the same. Any classifier that falls on this line is no better than random guessing,

[2]This is for a given threshold value, say 0.5. More detail on this is provided in section 6.4.1.3.

and any classifier that falls below this line performs worse than random guessing.

Point B represents perfect classification – all positive patterns are classified as positive and all negatives are mapped to the negative class. Informally, the 'closer' a point is to point B, the better it performs. Point D is the opposite of B and illustrates the strategy of mapping all positive patterns to the negative class and all negatives to the positive class – this is the 'worst' possible classifier. Note however, that if the class labels were reversed, the classifier would be relocated to point B. Thus the ROC surface is symmetrical around the dashed line. This means that, although point F performs worse than random guessing, it can be transformed to have the same performance as point E, as E and F are symmetrical points around the dashed line.

It is important to note that the concept of 'best' classification is relative to the scenario that the classifier has to solve. Certain applications require more conservative classifiers that only map instances to the positive class if there is very strong evidence, while other applications need more liberal classifiers that issue positive classifications with very little evidence. Interestingly, a classifier that issues continuous output (such as a neural network) can provide both a liberal or a conservative classifier, depending on the chosen threshold value. The required threshold value is determined by using ROC curves and is discussed in the next section.

6.4.1.3 ROC Curves

Discrete classifiers issue discrete outputs – a pattern is mapped to one of two classes with a certainty of either 1.0 or 0.0. This in turn produces a single confusion matrix and thus a single (TP_{rate}, FP_{rate}) pair that gives a single ROC point. Yet certain classifiers such as neural networks produce continuous output, where a pattern's classification is expressed in a certain range, say (0.0, 1.0), with 0.0 being a negative classification and 1.0 being a positive classification. The problem is to decide where the boundary (or threshold) lies where all output below the threshold is considered to be negative and all those above considered positive.

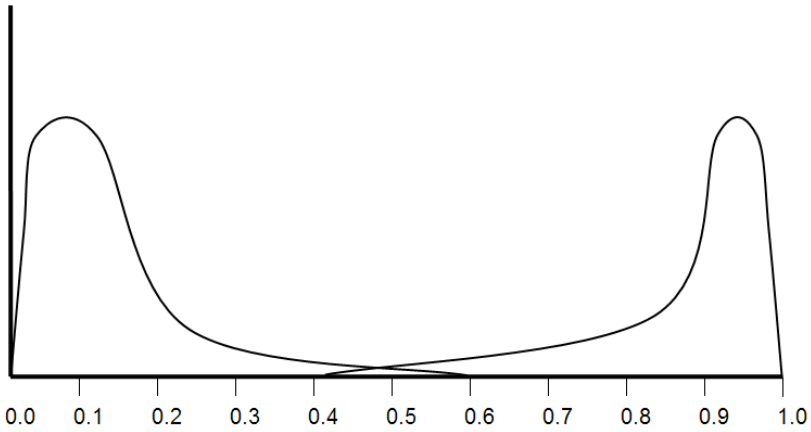
Each choice of threshold produces a different point in ROC space (i.e. a different (TP_{rate}, FP_{rate}) pair), implying (rather intuitively) that each choice of threshold changes how the same classifier algorithm maps patterns to classes. As an example, the extreme

Inputs: D_T = training set,
$f(i)$ = classifier output, *min* and *max* is the range of $f(i)$,
increment = the value to increase the threshold t by each iteration.
for t = *min* **to** *max* **by** *increment* **do**
FP = 0
TP = 0
for $i \in D_T$ **do**
    if $f(i) \geq t$ **then**
        if i is a positive example **then**
            TP = TP + 1
            else FP = FP + 1
        end if
    end if
end for i
Add $(\frac{FP}{n}, \frac{TP}{p})$ to ROC curve
end for t

Algorithm 6.4: A Conceptual method for drawing an ROC curve.

threshold of 1.0 with an output range of $[0.0, 1.0]$ will ensure that *all* patterns are classified as negative (i.e. below the threshold). This maps to point A in figure 6.4. A threshold of 0.0 in the same system will have all patterns mapped to the positive class, which will result in point C in figure 6.4. By varying the threshold from the minimum to the maximum of the range of the classifier and computing the confusion matrix for each threshold, a curve can be drawn in ROC space. This is called an *ROC Curve*. A pseudo code algorithm, given in [30], is listed in algorithm 6.4 that shows the logic of how to generate an ROC curve.

The optimal classifier threshold corresponds to the point (0,1) on the ROC surface which makes no false positives and no false negatives. The question is what threshold value comes closest to this optimum point? By drawing a ROC curve of various possible threshold values, it is easy to see visually which point is 'closest' to the point (0,1), as is



Threshold	TP	FP	FP_{rate}	TP_{rate}
1.0	0	0	0	0
0.9	500	0	0	0.5
0.8	700	0	0	0.7
0.7	870	0	0	0.87
0.6	940	0	0	0.94
0.5	970	20	0.02	0.97
0.4	1000	60	0.06	1
0.3	1000	140	0.14	1
0.2	1000	290	0.29	1
0.1	1000	450	0.45	1
0.0	1000	1000	1	1

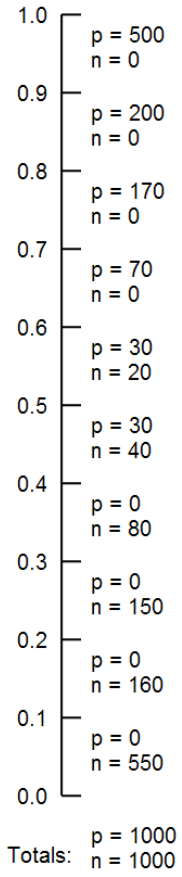


Figure 6.5: The effect of moving the threshold on TP_{rate} and FP_{rate}.

shown in figure 6.7. The solid line represents a curve that is arguably the best, as it comes the closest to point (0,1). The curve illustrated by the fine dots shows a poor performing classifier, as its average performance is only slightly better than random guessing (the line in the figure). The ROC curve illustrated by the broken line performs worse in the 'conservative' area of the ROC surface than the solid line curve, but performs better in the 'liberal' part of the graph. While ROC curves are good to visually inspect which classifiers give the best performance at certain points on the graph, it is difficult to compare classifiers accurately for the entire curve. A solution to this is the area under curve (AUC) metric that is discussed next.

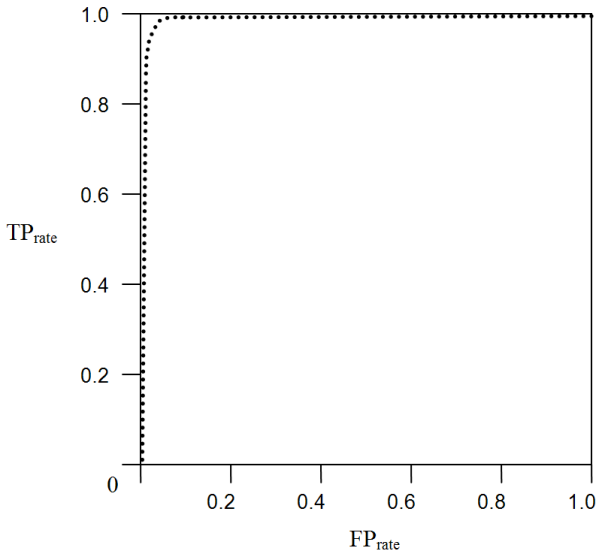


Figure 6.6: The ROC curve generated in figure 6.5.

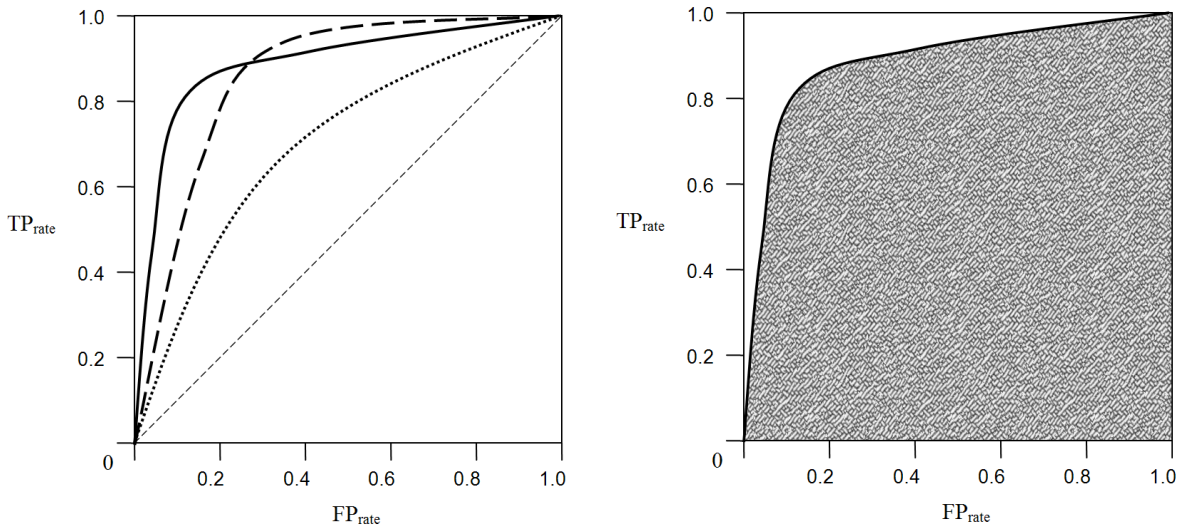


Figure 6.7: Various ROC curves and the AUC of the solid curve.

6.4.1.4 Area Under Curve (AUC)

ROC curves give a good visual overview of how a classifier performs given different threshold values. It becomes very difficult to compare different classifiers against each other, as curves need to be compared against other curves. Fortunately, the ROC surface

is unit sized (its length and breadth is always 1.0), which implies that its total surface always lies in the range $[0.0, 1.0]$. The *area under curve (AUC)* metric takes this fact into account and provides a way to compare the performance of classifier using a single *scalar* value.

ROC curves are monotonically increasing functions that start at point $[0, 1]$ and end at $[1, 1]$ [60]. Thus, if classifier A has a larger AUC than classifier B, it implies that, on average, the ROC curve of A must lie closer to the point (0,1) than the curve of B. Note that random guessing is illustrated by the line from $[0, 1]$ to $[1, 1]$, which gives an AUC of 0.5. Thus no realistic classifier should ever yield an AUC less than 0.5. The AUC of a classifier is equivalent to the probability that a randomly chosen positive pattern will be ranked higher than a randomly chosen negative instance, which is the same as the statistical Wilcoxon test of ranks [30]. An example of an ROC curve and its corresponding AUC is given is figure 6.7.

6.4.1.5 ROC and AUC for Multi-class Problems

Up to this point in the discussion, only problems with two classes were considered. For problems with two classes, the confusion matrix is 2×2 as illustrated in figure 6.3. For problems with $\mathcal{K}$ class labels, the confusion matrix becomes a $\mathcal{K} \times \mathcal{K}$ matrix, with the diagonal representing $\mathcal{K}$ benefits and the $\mathcal{K}^2 - \mathcal{K}$ off-diagonal entries showing the errors made. This makes ROC curves a bit more complex for multi-class classification. Fortunately the AUC can still be regarded as a scalar value in multi-class problems, which makes the AUC an especially valuable metric.

Various ways to calculate AUC from multiple ROC curves are discussed in [30]. A very intuitive method to calculate AUC, called the *class reference formulation*, produces $\mathcal{K}$ different ROC graphs using 2×2 confusion matrices. For each class c_i, an ROC graph is plotted using c_i as the positive class P_i, and all other classes as the negative class N_i:

$$P_i = c_i \tag{6.14}$$

$$N_i = \bigcup_{\forall j, j \neq i} c_j \in C \tag{6.15}$$

where C is the set of all classes. The corresponding AUC calculation is given by

$$AUC_{total} = \sum_{c_i \in C} AUC(c_i) \cdot p(c_i) \tag{6.16}$$

While this is an easy way to extrapolate ROC analysis to multiple dimensions, it violates the desirable ROC property of being insensitive to skew class distribution [30]. Hand and Till [46] take a different approach by drawing all combinations of ROC curves for all the classes in set C. The AUC is then derived based on the fact that the AUC is equivalent to the Wilcoxon test of ranks:

$$AUC_{total} = \frac{2}{|C|(|C|-1)} \sum_{\{c_i,c_j\} \in C} AUC(c_i, c_j) \tag{6.17}$$

where $AUC(c_i, c_j)$ is the AUC of the 2-class ROC curve of classes c_i and c_j as discussed in previous sections. There are $\frac{|C|(|C|-1)}{2}$ such pairs. This formulation is insensitive to class distribution and is thus a more desirable calculation for multi-class AUC.

6.4.2 Implementation

The CILib implementation of the ROC measure is extremely simple and has already been illustrated in figure 4.4. Recall from section 4.1.4 that CILib provides a built-in measurement system that can be used by the CILib simulator or developers' own applications. Specific implementations should extend the `Measurement` interface.

The `AreaUnderROC` class follows these guidelines to implement ROC analysis. The `getValue()` method invokes the `calculateAUC()` method, which proceeds to calculate various ROC points based on separate confusion matrices, plots a ROC curve based on these points, and finally calculates the area under curve (AUC) metric using equation 6.17. A single scalar value is returned as the performance measure of the classifier. Any component that is required to act as a classifier is regarded as a `Function`, either directly (as a subclass of `Function`) or via the *adapter* design pattern.

Notice in figure 4.4 that the `threshold` member variable is of type `double[]`, meaning that any number of threshold values may be set. This is required as each threshold value generates a new ROC point, and all of these ROC points in turn are used to plot the ROC curve. The number of classes must also be set in the `classCount` variable, as equation

6.17 requires this. Any reasonable number of classes and threshold values may be set, taking into consideration that a too fine granularity will severely impact execution time due to the complexity of ROC analysis.

6.5 Summary

This chapter investigated various advanced use cases that a generic neural network framework should be able to support. While a complete overview was not given in each case, enough information on what such an implementation might entail was given. The discussions also focus on how existing components are reused and how the new functionality may in turn be reused by other components.

Chapter 7

Conclusion

The future is called "perhaps," which is the only possible thing to call the future. And the only important thing is not to allow that to scare you.

— *Tennessee Williams, Orpheus Descending, 1957*

This chapter provides a summary of this dissertation in section 7.1, and provides ideas for future work and research in section 7.2

7.1 Summary

This aim of this dissertation was to discus the design and implementation of a generic neural network framework in CILib that allows users to design, implement and use any possible neural network architectures and algorithms in such a way that they can reuse and be reused by any other CI algorithm in the rest of CILib (or any external application). This was achieved by using object-oriented design patterns in the design of the framework which maximises its reusability and extensibility.

It was discussed that developers that need neural network implementations could either write their own implementations, or reuse other implementations. Typical design dilemmas that neural network developers face were discussed. It was shown that neural network requirements are very tightly coupled – a change in one component tends to have severe effects in other components, making the change impossible or inefficient.

Often this situation leads to a major redesign of the neural network system and in many cases a completely rewritten application.

Challenges that were highlighted included the net input signal and activation functions of neurons, recurrent neuron functionality, neuron and weight topologies, types of weight values, architecture selection, neural learning, data sets, data sources, pattern presentation and ordering, simulation measurements and stopping conditions, parallel computing considerations, integration integrity, as well as use cases that do not involve training (i.e. applications of neural networks). It was shown why components in a neural network are functionally very tightly coupled and that great case and a good background study is needed in order to decide how to separate functionality into separate modules.

A few well-known neural network packages and libraries were listed, their main capabilities were expressed, and challenges and shortcomings that these packages have were highlighted. The main shortcoming was shown to be that most neural network libraries and packages tend to be designed and constructed with merely neural networks as a field of computational intelligence in mind. While some libraries allow interactions with other CI algorithms, these algorithms typically reside *outside* of the neural network library. This creates the need for integration and results in inefficiencies and extended development times. It was argued that integrating such specialised neural network libraries with other areas of CI such as particle swarm optimisation and evolutionary computation was in most cases very difficult or inefficient to do.

A conceptual breakdown of neural networks was given in chapter 2, which showed that a neural network model essentially consists of a topology model, a data model and a learning model. The topology describes the neuron as a single computation unit that comprises of a pipeline of processes and phases. These neurons are connected together via a multitude of possible interconnection schemes to form a network with one or more layers. It was shown that neuron topologies can be regarded as a graph, which greatly aided in the design of a generic neural network topology implementation. The data component provides information about a problem that the neural network needs to approximate. Various considerations such as pattern sampling, pattern ordering and sources of data needed to be taken into account. It was shown that neural learning involves changing a network's weight vector so that the neural network approximates

the problem to a desired accuracy. Various learning paradigms were discussed such as supervised, unsupervised, reinforcement and hybrid learning. Network architecture selection, ensemble and modular networks were also briefly discussed.

'Designing object-oriented software is hard, and designing reusable object-oriented software is even harder' as quoted by the GoF [39] summarised the aim of using design patterns in application design. Design patterns allow developers to solve complex problems by using reusable solutions in a well-proven and predictable manner with known outcomes and advice on how to use the pattern in new scenarios. Design patterns offer advantages such as flexibility, reusability, extensibility, speed, reliability, separability, maintainability and dynamicity. An overview of the design patterns that were used in the neural network framework was given in chapter 3.

CILib is a framework that allows developers to develop new CI implementations quickly and efficiently. Flexibility, reusability and clear separation between components are maximised through the use of design patterns. Reliability is also ensured as the framework is open source and many people collaborate to ensure that the framework is well designed and error free. CILib allows researchers to design, implement and run CI simulations that consist of algorithms that work on problems. Simulations run until the process completes as stipulated by stopping conditions and also allow measurements to be taken periodically. Furthermore, simulations can be set up dynamically using an XML interface. A brief overview of the CILib framework was given in chapter 4.

A layout of how the CILib framework could be extended to support neural network implementations was given. This task was aided by the use of design patterns in chapter 3 and the conceptual breakdown of neural networks in chapter 2. It was shown that neural network topologies can be represented using the *strategy* design pattern which gave advantages such as encapsulation, simpler and extensible clients, and dynamicity. The data component is responsible for providing a source of data to train a neural network model. It was shown how any type of data pattern could be supported. It was discussed how the learning component is supported for both stand-alone and specialised optimisation techniques, making it easy for external algorithms to train neural network models. This was done using the *strategy* and *adapter* design patterns respectively. Mediation is required between the topology, data and learning components as these are

not applications in their own right. Neural network models also have variables and concepts that stretch across these components and need to be represented. It was shown how the *mediator* pattern can be used to good effect when integrating various neural network components into a coherent system.

Furthermore, a generic topology implementation was discussed that is capable of representing any topology that can be expressed as a mathematical graph. This generic implementation makes it easy for developers to use a well tested, existing solution to easily and quickly define new neural network models. The generic implementation uses a whole host of design patterns to aid in flexibility, efficiency and extensibility. A generic data implementation is also developed that is capable of reading any patterns from any source and distribute them among D_T, D_G, D_V and D_C using any distribution method. Both these implementations are fully extensible and can be configured (without code changes) to provide different topology or data implementations respectively.

Some advanced implementations in the neural network framework were shown, including active learning approaches such as the sensitivity analysis incremental learning algorithm (SAILA) and dynamic pattern selection (DPS), neural learning with other CI algorithms such as a PSO or an EA, ensemble network applications, modular neural network topologies, network pruning and growing algorithms and a measure of classification performance, namely receiver operating characteristics (ROC) analysis. The emphasis is on reconfiguring and extending existing components to implement a desired neural network application quickly and efficiently. Ideally, developers only need to write code for components that do not exist yet rather than continuously rewriting the same logic for each new application.

7.2 Future Work

Some ideas for future work and research include:

- **XML builder for `GenericTopology`.** Everytime a new topology type has to be represented using `GenericTopology`, a new builder class has to be defined to construct it correctly using building blocks like `NeuronConfig` and `Weight`. It is possible to define neural network topologies as XML structures [93], similar to the

way they are represented using objects in `GenericTopology`. A generic XML-based builder implementation can be defined that can build *any* type of topology that can be represented in XML.

Following this line of thought, it seems feasible that a builder class can be defined that is capable of building topologies as stated using a formal language for neural network specifications. Kazmierczak, Senyard and Sterling [61] show that the Z language[1] can be used to specify neural network topologies (as well as other aspects of neural network development). Fiesler [33] also defines a formal mathematical language for neural network definition.

- **Graphical neural network models.** As a neural network topology can be defined using XML, it is possible to design the topology *graphically* and save the output in the desired XML format for the XML builder. This will greatly aid the previous point as well, as topology definition can be done using a graphical user interface, which avoids the user having to manually edit XML definition files.

- **Reusable subcomponents of `EvaluationMediator`.** Currently, a new mediator implementation has to be written for every new type neural network implementation. Yet many aspects of each application are rewritten, such as the logic for an epoch over D_T vs. an epoch over D_G. Conceptually, these operations are the same but different data sets are used. A taxonomy of reusable and composable subcomponents of `EvaluationMediator` can be built that makes it easy to reuse pockets of application logic effectively. The *decorator* design pattern (not discussed, but see [39]) may well prove useful here.

- **More implemented neural network models.** This dissertation looked at the *framework* for neural network application construction. Yet the framework is designed to reuse and extend existing implementations. More base implementations need to be added, making it easy for developers to build neural network models by configuring them as opposed to coding them.

[1] Refer to http://www.comlab.ox.ac.uk/archive/z.html for information on the Z standard.

- **Tighter CILib integration.** It is starting to emerge that more and more CILib implementations would like to reuse the functionality provided by classes such as `NeuralNetworkData` and its generic implementation `GenericData`. There are plans to refactor this functionality to be part of the base framework instead of being part of the neural network framework only.

Bibliography

[1] C. Archambeau *et al.* Phosphene Evaluation in a Visual Prosthesis With Artificial Neural Networks. *Adaptive Systems and Hybrid Computational Intelligence in Medicine*, 2001.

[2] J. Austin and S. Buckle. The Practical Application of Binary Neural Networks. Technical report, Advanced Computer Architecture Group, University of York, 1994.

[3] F. Azam. *Biologically Inspired Modular Neural Networks.* PhD thesis, Virginia Polytechnic Institute and State University, 2000.

[4] K. Balakrishnan and V. Honavar. Evolutionary Design of Neural Architectures – A Preliminary Taxonomy and Guide to Literature. Technical report, Iowa State University, 1995.

[5] N.A. Barricelli. Esempi Numerici di Processi di Evoluzione. *Methodos*, pages 45–68, 1954.

[6] A.G. Barto, R.S. Sutton, and C. Watkins. Neuronlike Adaptive Elements That Can Solve Difficult Problems. In *IEEE Transactions on Systems, Man and Cybernetics*, volume vol. 13, pages 834–846, 1983.

[7] G. Beni and U Wang. Swarm Intelligence in Cellular Robotic Systems. *NATO Advanced Workshop on Robots and Biological Systems*, 1989.

[8] C.M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[9] Z. Chuan, L. XianLiang, H. Mengshu, and Z. Xu. A LVQ-based Neural Network Anti-spam Email Approach. In *ACM SIGOPS Operating Systems Review archive*, volume vol. 39, pages 34–39. 2005.

[10] I. Cloete and J. Ludik. Increased Complexity Training. In *New Trends in Neural Computation*, pages 267–271. Springer-Verlag, 1993.

[11] I. Cloete and J. Ludik. Delta Training Strategies. In *IEEE World Conference On Computational Intelligence, Proceedings Of The International Joint Conference On Neural Networks.*, volume vol. 1, pages 295–298, 1994.

[12] D. Cohn, L. Atlas, and R. Ladner. Improving Generalization With Active Learning. *Machine Learning*, pages 129–145, 1994.

[13] J.J. Craig. *Introduction to Robotics*. Pearson Prentice Hall, 2005.

[14] P. Dart, A. Senyard, and L. Sterling. Towards the Software Engineering of Neural Networks: a Maturity Model. In *Proceedings of the 2000 Australian Software Engineering Conference*, page 45, 2000.

[15] R. Dawkins and J.R. Krebs. Arms Races Between and Within Species. In *Proceedings of the Royal Society of London*, volume 205, pages 489–511, 1979.

[16] H. Demuth and M. Beale. *Neural Network Toolbox: For Use With MATLAB*. The Mathworks Inc., 1997.

[17] V. Deolalikar. Mapping Boolean Functions With Neural Networks Having Binary Weights and Zero Thresholds. Technical report, Information Theory Research Group, HP Laboratories, 2001.

[18] A. Diekmann and S. Gutjahr. Prediction of the Euro-Dollar Future Using Neural Network – A Case Study for Financial Time Series Prediction. In *IDEAL '98*, 1998.

[19] R. Diestel. *Graph Theory*. Springer-Verlag New York, electronic edition 2000 edition, 1997,2000.

[20] W. Duch and N. Jankowski. Survey of Neural Transfer Functions. Technical report, Department of Computer Methods, Nicholas Copernicus University, 1999.

[21] W. Durbin and D.E. Rumelhart. Product Units: A Computationally Powerful and Biologically Plausible Extention to Backpropagation Networks. *Neural Computation*, vol. 1:133–142, 1989.

[22] J.L. Elman. Finding Structure in Time. *Cognitive Science*, vol. 14:179–211, 1990.

[23] A.P. Engelbrecht. *Sensitivity Analysis of Multilayer Neural Networks*. PhD thesis, University of Stellenbosch, South Africa, 1999.

[24] A.P. Engelbrecht. Data Generation Using Sensitivity Analysis. In *International Symposium On Computational Intelligence*, 2000.

[25] A.P. Engelbrecht. *Computational Intelligence, An Introduction*. John Wiley and Sons, 2002.

[26] A.P. Engelbrecht and I. Cloete. A Sensitivity Analysis Algorithm for Pruning Feedforward Neural Networks. *Proceedings of the IEEE International Conference in Neural Networks*, vol. 2:1274–1277, 1996.

[27] A.P. Engelbrecht, I. Cloete, J. Geldenhuys, and J.M. Zurada. Automatic Scaling Using Gamma Learning for Feedforward Neural Networks. Technical report, University of Stellenbosch, South Africa, 1995.

[28] A.P. Engelbrecht, L. Fletcher, and I. Cloete. Variance Analysis of Sensitivity Information for Pruning Feedforward Neural Networks. *Proceedings of the IEEE International Joint Conference in Neural Networks*, 1999.

[29] S.S. Epp. *Discrete Mathematics With Applications*. Brooks/Cole Publishing Company, 2nd edition, 1995.

[30] T. Fawcett. ROC Graphs: Notes and Practical Considerations for Researchers. Technical report, HP Laboratories, 2004.

[31] T. Fawcett and F. Provost. Analysis and Visualization of Classifier Performance: Comparison Under Imprecise Class and Cost Distributions. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining.*, 1997.

[32] T. Feuring. Learning in Fuzzy Neural Networks. *Proceedings of IEEE International Conference on Neural Networks*, 1996.

[33] E. Fiesler. Neural Network Classification and Formalization. *Computer Standards and Interfaces*, vol. 16, 1994.

[34] E. Fiesler. Neural Network Topologies. *Handbook of Neural Computation*, 1996.

[35] D. Flanagan. *Java In A Nutshell.* O'Reilly, 5th edition, 2005.

[36] L.J. Fogel, A.J. Owens, and Walsh M.J. *Artificial Intelligence Through Simulated Evolution.* Wiley, 1966.

[37] A. Fraser. Simulation of Genetic Systems by Automatic Digital Computers. *Australian Journal of Biological Sciences*, vol. 10:484–491, 1957.

[38] B. Fritzke. Incremental Learning of Local Linear Mappings. *Proceedings of the International Conference of Artificial Neural Networks*, 1995.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[40] T.D. Gedeon, P.M. Wong, and D. Harris. Balancing Bias and Variance: Network Topology and Pattern Set Reduction Techniques. In *From Natural To Artificial Neural Computing*, pages 551–558. 1995.

[41] J. Ghosh and Y. Shin. Efficient Higher-order Neural Networks for Classification and Function Approximation. *International Journal of Neural Systems*, vol. 3:323–350, 1992.

[42] F. Girosi, M. Jones, and T. Poggio. Regularisation Theory and Neural Network Architectures. *Neural Computation*, vol. 7, 1995.

[43] S. Grossberg. Competitive Learning: From Interactive Activation to Adaptive Resonance. *Cognitive Science*, vol. 11, 1987.

[44] M. Hagiwara. Removal of Hidden Units and Weights for Back Propagation Networks. *Proceedings of the 1993 International Joint Conference on Neural Networks*, vol. 1:351–254, 1993.

[45] J. Häkkinen, M. Lagerholm, C. Peterson, and B. Söderberg. A Potts Neuron Approach to Communications Routing. *Neural Computation*, vol. 10, 1998.

[46] D.J. Hand and R.J. Till. A Simple Generalization of the Area Under the ROC Curve to Multiple Class Classification Problems. *Machine Learning*, vol. 45:171–186, 2001.

[47] D.O. Hebb. *The Organisation of Behaviour*. Wiley, 1949.

[48] Y. Hirose, K. Yamashita, and S. Hijiya. Back Propagation Algorithm Which Varies the Number of Hidden Units. *Neural Networks*, vol. 4:61–66, 1991.

[49] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[50] J.J. Hopfield. Neural Networks and Physical Systems With Emergent Collective Behaviour. *Proceedings of the National Academy of Science USA*, vol. 79:2554–2558, 1982.

[51] P.J. Huber. *Robust Statistics*. John Wiley and Sons, 1981.

[52] W.S. Humphrey. *Managing the Software Process*. Addison-Wesley, 1989.

[53] D.R. Hush, J.M. Salas, and B. Horne. Error Surfaces for Multi-Layer Perceptrons. In *International Joint Conference on Neural Networks*, volume vol. 1, pages 759–764, 1991.

[54] A. Hussain, J.J. Soraghan, and T.S. Durbani. A New Neural Network for Nonlinear Time-Series Modelling. *Neuro Vest Journal*, pages 16–26, 1997.

[55] J.P. Ignizio. *An Introduction to Expert Systems: The Development and Implementation of Rule-based Expert Systems*. McGraw-Hill, 1991.

[56] A. Ismail and A.P. Engelbrecht. Training Product Unit Neural Networks With Particle Swarm Optimizers. Technical report, Department of Computer Science, University of the Western Cape, South Africa; Department of Computer Science, University of Pretoria, South Africa, 1999.

[57] J. Janson and J.F. Frenzel. Training Product Unit Neural Networks With Genetic Algorithms. *IEEE Expert*, pages 26–33, 1993.

[58] D. Jiménez. Dynamically Weighted Ensemble Neural Networks for Classification. Technical report, University of Texas, San Antonio, TX, USA, 1998.

[59] M.I. Jordan. Attractor Dynamics and Parallelism in a Connectionist Sequential Machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 531–546, 1986.

[60] T. Kanungo, D.M. Gay, and R.M. Haralick. Constrained Monotone Regression of ROC Curves and Histograms Using Splines and Polynomials. *ICIP*, 1995.

[61] E. Kazmierczak, A. Senyard, and L. Sterling. Software Engineering Methods for Neural Networks. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, pages 1530–1362, 2003.

[62] J. Kennedy and R.C. Eberhart. Particle Swarm Optimization. In *Proceedings of the IEEE International Conference of Neural Networks*, volume vol. 4, pages 1942–1948, 1995.

[63] T. Kohonen. *Self-Organising Maps*. Springer Series In Information Science, 1995.

[64] T. Kohonen. *Self-Organising Maps*. Springer, second edition, 1997.

[65] T Kohonen, G. Barna, and R. Chrisley. Statistical Pattern Recognition With Neural Networks: Benchmarking Studies. In *Proc. IEEE International Conference On Neural Networks.*, pages 61–67, 1988.

[66] B. Kröse and P. Van der Smagt. *An Introduction to Neural Networks*. University of Amsterdam, 1996.

[67] R. Kruse, D. Nauck, and F. Klawonn. Neuronale Netze und Fuzzy-Systeme. *Vieweg*, 1996.

[68] S.Y. Kung and J.S. Taur. Decision-based Neural Networks With Signal/image Classification Applications. In *IEEE Transactions On Neural Networks*, volume vol. 6, pages 170–181, 1995.

[69] T.Y. Kwok and D.Y. Yeung. Constructive Feedforward Neural Networks for Regression Problems: A Survey. Technical report, The Hong Kong University of Science and Technology, 1995.

[70] M. Lagerholm, C. Peterson, and B. Söderberg. Airline Crew Scheduling Using Potts Mean Field Techniques. In *European Journal of Operational Research*, 1998.

[71] R Lange and R. Männer. Quantifying a Critical Training Set Size for Generalization and Overfitting Using Teacher Neural Networks. In *International Conference on Artificial Neural Networks*, volume vol. 1, pages 497–500, 1994.

[72] S. Lange and T. Zeugmann. Incremental Learning From Positive Data. *Journal of Computer and Systems Sciences*, vol. 53:88–103, 1996.

[73] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Interactive Development*. Prentice Hall, 2005.

[74] L.R. Leerink, C. Lee Giles, B.G. Horne, and M.A. Labri. Learning With Product Units. *Advances in Neural Information Processing Systems*, vol. 7:537–544, 1995.

[75] D.J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 6th edition edition, 1995-2003.

[76] P. Marrone. *Java Object Oriented Neural Engine – A Complete Guide*. http://www.joone.org, 2007.

[77] D. Michie, D.J. Spiegelhalter, and C.C. Taylor. Machine Learning, Neural and Statistical Classification. Technical report, University of Leeds, University of Strathclyde, MRC Biostatistics unit Cambridge,, 1994.

[78] J. Moody and C.J. Darken. Fast Learning in Networks of Locally Tuned Processing Units. *Neural Computation*, vol. 1, 1989.

[79] D. Nauck and R. Kruse. Neuro-Fuzzy Systems for Function Approximation. Technical report, Faculty of Computer Science, Otto-von-Gueriche-University of Magdeburg, Germany, 1999.

[80] N.J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers Inc, 1998.

[81] Y. Ofran and B. Rost. Predicted Protein-Protein Interaction Sites From Local Sequence Information. Technical report, Columbia University, 2003.

[82] S. Oh, R. Reed, and R.J. Marks II. Similarities of Error Regularization, Sigmoid Gain Scaling, Target Smoothing and Training With Jitter. *IEEE Transactions of Neural Networks*, vol. 6:529–538, 1995.

[83] N Ohnishi, A. Okamoto, and N. Sugiem. Selective Presentation of Learning Samples for Efficient Learning in Multi-Layer Perceptron. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, volume vol. 1, pages 688–691, 1990.

[84] E. Oja. A Simplified Neuron Model as a Principle Component Analyzer. *Journal of Mathematical Biology*, vol. 15:267–273, 1982.

[85] E.S. Peer. A Serendipitous Software Framework for Facilitating Collaboration in Computational Intelligence. Master's thesis, University of Pretoria, South Africa, 2004.

[86] M.P. Perrone and L.N. Cooper. *When Networks Disagree: Ensemble Methods for Hybrid Neural Networks*. Chapman-Hall, 1993.

[87] V.P. Plagianakos, G.D. Magoulas, N.K. Nousis, and M.N. Vrahatis. Training Multilayer Networks With Discrete Activation Functions. Technical report, University Of Patras Artificial Intelligence Research Center, Patras, Greece, 2000.

[88] K.V. Price, R.M. Storn, and J.A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer Verlag Berlin and Heidelberg GmbH and Co. K, 2005.

[89] S. Rattan and W. Hsieh. Complex-valued Neural Networks for Non-linear Complex Principle Component Analysis. Technical report, University of Brittish Columbia, 2004.

[90] I. Rechenberg. *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien des Biologischen Evolution*. Fromman-Hozlboog Verlag, 1973.

[91] A. Röbel. The Dynamic Pattern Selection Algorithm: Effective Training and Controlled Generalization of Backpropagation Neural Networks. Technical report, Institut für Angewandte Informatik, Technische Universität Berlin, 1994.

[92] M. Rosen-Zvi and I. Kanter. Training a Perceptron in Discrete Weight Space. *Physical Review E*, vol. 64, 2001.

[93] D.V. Rubtsov and S.V. Butakov. XML-Based Format for Trained Neural Network Definition. *Academic Open Internet Journal - www.acadjournal.com*, vol. 4, 2001.

[94] J. Rumbaugh *et al. Object-Oriented Modelling and Design*. Prentice Hall, 1991.

[95] J. Ryan, M. Lin, and R. Miikkulainen. Intrusion Detection With Neural Networks. In *Advances in Neural Information Processing Systems*, volume vol. 10, 1998.

[96] R. Schank. *Dynamic Memory: A Theory of Learning in Computers and People*. Cambridge University Press, 1982.

[97] E. Schikuta. A Software Engineering Approach to Neural Network Specification. In *Third Annual SNN Symposium on Neural Networks: Artificial Intelligence and Industrial Applications*, 1995.

[98] A. Schmidt and Z. Bandar. In *IASTED International Conference Computer Systems and Applications*, 1998.

[99] C. Shah. Automatic Organisation of Text Documents in Categories Using Self-Organising Map (SOM). Technical report, IEEE Regional Student Paper Contest, 2002.

[100] J. Sietsma and R.J.F. Dow. Creating Artificial Neural Networks That Generalise. *Neural Networks*, vol. 4, 1991.

[101] P. Slade and T.D. Gedeon. Bimodal Distribution Removal. In *New Trends in Neural Computation*, pages 249–254. Springer-Verlag, 1993.

[102] R. Smith. Network-based Intrusion Detection Using Neural Networks. Technical report, Rensselaer Polytechnic Institute, 2002.

[103] J.A. Snyman. A New and Dynamic Method for Unconstrained Optimization. *Applied Mathematical Modelling*, vol. 6:449–462, 1982.

[104] J.A. Snyman. An Improved Version of the Original LeapFrog Dynamic Method for Unconstrained Optimization: LFOP1(b). *Applied Mathematical Modelling*, vol. 7:216–218, 1983.

[105] E.P. Solomon, L.R. Berg, and D.W. Martin. *Biology*. Thomson Learning, Brooks/Cole, 7th edition, 2005.

[106] K.A. Spackman. Signal Detection Theory: Valuable Tools for Evaluating Inductive Learning. In *Proceedings of the sixth international workshop on Machine learning*, pages 160–163. Morgan Kaufmann Publishers Inc., 1989.

[107] K.O. Stanley and R. Miikkulainen. Efficient Evolution of Neural Network Topologies. Technical report, University Of Texas at Austin, 2002.

[108] J.M. Steppe, K.W. Bauer, and S.K. Rogers. Integrated Feature and Architecture Selection. *IEEE Transactions on Neural Networks*, vol. 7:1007–1014, 1996.

[109] G. Tesauro. Neurogammon: A Neural Network Backgammon Program. In *IJCNN Proceedings III*, pages 33–39, 1990.

[110] H.H. Thodberg. Improving Generalisation of Neural Networks Through Pruning. *International Journal of Neural Systems*, vol. 1, 1991.

[111] A.M. Turing. Computing Machinery and Intelligence. *Mind*, 59:433–460, 1950.

[112] F. Van den Bergh. *An Analysis of Particle Swarm Optimizers*. PhD thesis, University of Pretoria, South Africa, 2001.

[113] F Van den Bergh and A.P. Engelbrecht. Cooperative Learning in Neural Networks Using Particle Swarm Optimization. Technical report, Department of Computer Science, University Of Pretoria, 2000.

[114] J. Von Neumann. The General and Logical Theory of Automata. *Cerebral Mechanisms in Behaviour - The Hixon Symposium*, 1951.

[115] L.F.A. Wessels and E. Barnard. Avoiding False Local Minima By Proper Initialization of Connections. *IEEE Transactions on Neural Networks*, vol. 3:899–905, 1992.

[116] J. Westra. Evolutionary Neural Networks Applied in First Person Shooters. Master's thesis, University Utrecht, 2007.

[117] D. Whitney and C. Bogart. The Evolution of Connectivity: Pruning Neural Networks Using Genetic Algorithms. *Proceedings of the IEEE International Joint Conference on Neural Networks*, vol. 1:134–137, 1990.

[118] P.M. Williams. Bayesian Regularisation and Pruning Using a Laplace Prior. *Neural Computation*, vol. 7, 1995.

[119] X. Yao and Y. Liu. Making Use of Population Information in Evolutionary Artificial Neural Networks. *IEEE Transactions on Systems, Man and Cybernetics*, 1998.

[120] L.A. Zadeh. Fuzzy Sets. *Information and Control*, 1965.

[121] A. Zell *et al.* Stuttgart Neural Network Simulator. Technical report, University of Stuttgart, University of Tübingen, 1998.

[122] B.T. Zhang. Accelerated Learning by Active Learning Example. *International Journal of Neural Systems*, vol. 5(no. 1):67–75, 1994.

[123] Q.J. Zhang and K.C. Gupta. *Neural Networks for RF and Microwave Design.* Artech House, 2000.

[124] Q.J. Zhang *et al.* Electromagnetic Based Modeling of Embedded Passives Using Neural Networks. In *IPC International Conference on Embedded Passives*, 2004.

[125] Q. Zhao. Modelling and Evolutionary Learning of Modular Neural Networks. Technical report, University of Aizu, Japan, 2001.

[126] J. Zurada. Lambda Learning Rule for Feedforward Neural Networks. In *Proceedings of the IEEE International Joint Conference in Neural Networks*, 1992.

Appendix A

Acronyms And Symbols

A.1 Acronyms

ACO	Ant Colony Optimisation
AI	Artificial Intelligence
AUC	Area Under Curve
BMU	Best Matching Unit
CI	Computational Intelligence
CILib	Computational Intelligence Library
DPS	Dynamic Pattern Selection
EA	Evolutionary Algorithm
EC	Evolutionary Computation
FFNN	Feedforward Neural Network
FN	False Negative
FP	False Positive
GD	Gradient Descent
LVQ	Learning Vector Quantiser
MSE	Mean Squared Error
MNN	Modular Neural Network
NEAT	Neuro Evolution of Augmenting Topologies
NN	Neural Network

PCA	Principle Component Analysis
PSO	Particle Swarm Optimisation
PU	Product Unit
RBFN	Radial Basis Function Network
ROC	Receiver Operating Characteristics
SAILA	Sensitivity Analysis Incremental Learning Algorithm
SOM	Self-Organising Map
SSE	Sum Squared Error
TN	True Negative
TP	True Positive
XML	Extensible Markup Language

A.2 Symbols

θ	Bias
$\mathbf{W}$	Weight vector for the entire neural network
$\mathbf{w}_n$	Weight vector for the neuron n
I	Number of inputs in the input layer
J	Number of hidden units in the hidden layer
K	Number of outputs in the output layer
z_i	i-th input unit
y_j	j-th hidden unit
o_k	k-th output unit
d_p	Denotes a single pattern
$\mathbf{z}_p$	Input vector for pattern d_p, dimension I
$\mathbf{t}_p$	Target vector for pattern d_p, dimension O
v_{ji}	Hidden weight connection to neuron j from neuron i
w_{kj}	Weight connection to neuron k from neuron j
f_{o_k}	Activation function for output neuron o_k
f_{y_j}	Activation function for hidden unit y_j
F_{NN}	Neural network output function
F_D	Function described by data in data set D
D_T	Training set
D_G	Generalisation set
D_V	Validation set
D_C	Candidate set
ε_T	Error on D_T
ε_G	Error on D_G
ε_V	Error on D_V
α	Momentum
α_i	Weights for ensemble network coordinator
η	Learning rate
$\Phi^{(p)}$	Informativeness of pattern d_p

Appendix B

Unified Modeling Language

B.1 UML Notation

The aim of this chapter is to brief the reader on the UML notation used in this thesis. Figure B.1 is a typical UML diagram as found in the rest of this thesis and will be used for the discussion. Based on the figure, the UML constructs used include:

- **Abstract and Concrete Classes.** Classes are represented as boxes with 3 fields namely the name, member variables and class methods. `EvaluationMediator` is an abstract class as can be seen on the diagram by its names being written in *italics*. `NeuralNetworkController` is an example of a concrete class as its name is not in italics. All classes have a circle with a 'C' before the name.

- **Class Methods and Members.** Member variables of a class are represented in the area beneath the class name, while methods are listed beneath the member variables. Method names in *italics* indicate abstract methods and thus can be found in abstract classes only. For notational convenience the members section, methods section or both sections can be collapsed, thus reducing the area used on diagrams (see `EvaluationMediator`). The colour of the icon of an element indicates member visibility – green means a member is public, yellow indicates protected while red indicates private. Blue triangles indicate inherited members and methods.

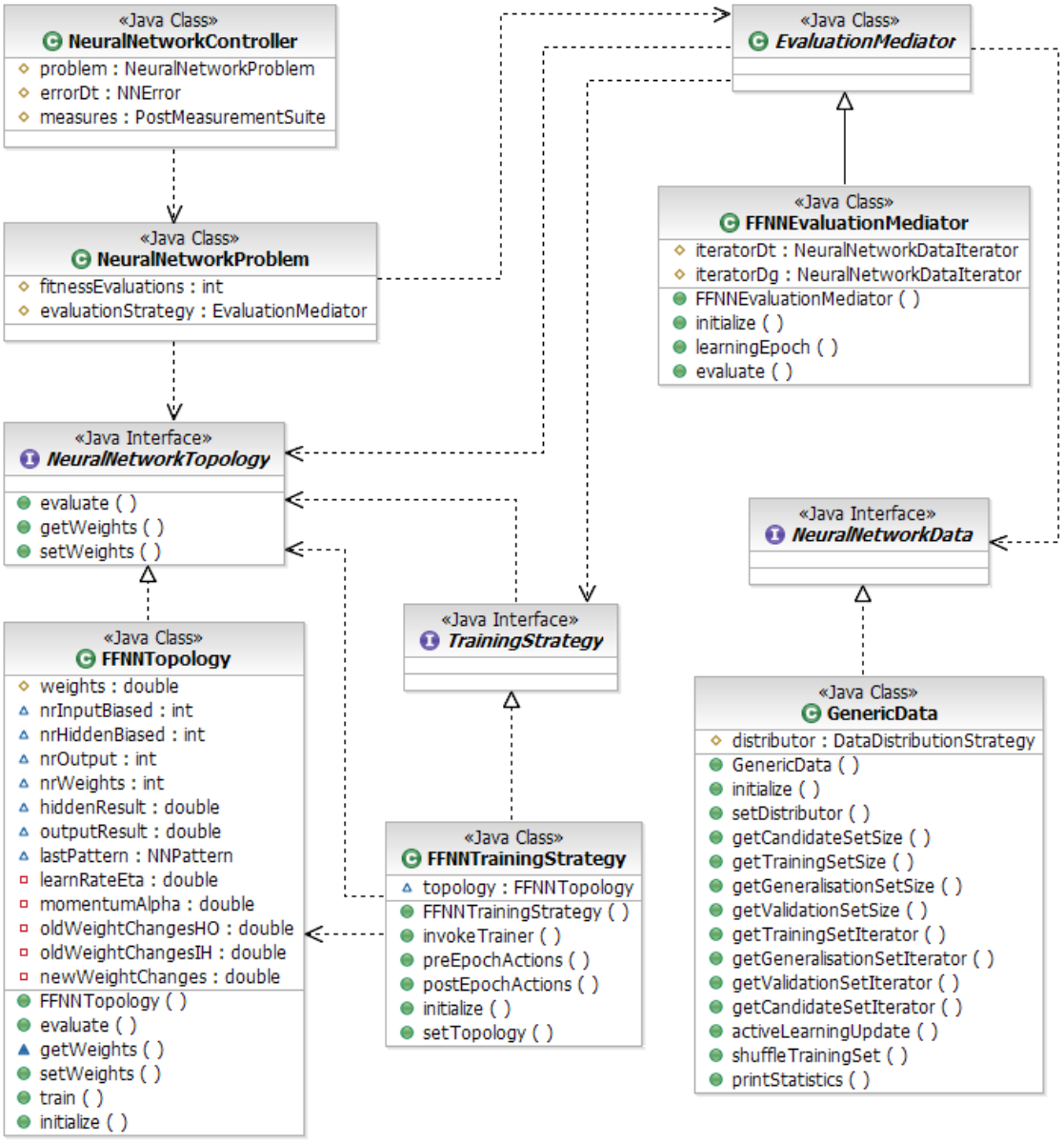


Figure B.1: FFNN application in the foundation framework

- **Interfaces.** Similar to classes, interfaces consist of boxes with the interface name and interface methods. There are no member variables for an interface. Interfaces have a circle with an 'I' before their name. The methods section can also be collapsed for notational convenience as is the case with classes.

- **Object References.** Object references are represented as dotted lines ending in an arrow leading from a class to another class or interface.

- **Class Inheritance and Interface Implementation.** Class inheritance is represented by a solid line leading from the superclass to the subclass. On the line there is also a triangle that forms an arrow pointing in the direction of the superclass. An example of inheritance can be seen between `EvaluationMediator` and `FFNNEvaluationMediator`.

  Interface implementation follows the exact structure of class inheritance, but the lines are dotted rather than solid. The triangle on the line points to the interface. An example can be seen in figure B.1 where `GenericData` class implements the `NeuralNetworkData` interface.

- **Other Notation.**

  Any reference to Java source code throughout the thesis is written in `typewriter` font. This allows the reader to easily distinguish between names of concepts and the actual code implementations.