KBCC Java Library Tutorial for the LNSC Laboratory

Laboratory for Natural and Simulated Cognition

Director: Dr. Thomas R. Shultz

Author: Jean-Philippe Thivierge

Last update: 03/01/22

KBCC Java Library Tutorial for the LNSC Laboratory

Foreword

Welcome to the first edition of the KBCC Tutorial ! This work is presented by the members of the Laboratory for Natural and Simulated Cognition (LNSC) at McGill University, Canada. The goal of this tutorial is to give the reader all the tools necessary to prepare, run, and analyze simulations in the KBCC library using the Java 2.0 programming language. If you are new to the KBCC code library, I hope you will enjoy using it and will appreciate its completeness and flexibility when applied to your own research and application endevours. This tutorial will guide you through the process of getting acquinted with the library, and ultimately becoming a skilled user. If you are a familiar user, I guarantee you will find this tutorial to be not only a great reference in the classes and methods of the KBCC library, but also a great friend in finding tips and tricks to reach a variety of goals.

Who should use this tutorial?

The current tutorial will guide the reader through using the KBCC Tutorial assuming a certain background. It is recommended that reader possess at least an intermediate level of comprehension of the Java 2.0 programming language, including a solid base of object oriented programming. As well, users should be familiar and comfortable with a programming environment in the Java language, such as JBuilder^{tm} or Forte^{tm}. These two applications are available commercially from Borland and Sun Systems respectively.

We will assume no strong mathematical background of the reader. However, readers who do possess skills in Calculus will undoubtly be advantaged in their comprehension of the inner workings of the algorithms involved in the KBCC library. We will also assume no background in neural networks. Here again, readers who understand neural networks will be advantaged in learning the KBCC library.

In a nutshell, this tutorial is intended for programmers who wish to learn how to use the KBCC library to run their own neural network simulations. Researchers and engineers in many domains

employ neural networks in order to analyze data, make predictions on future events, or more fundamentally study neural network algorithms. This library puts all the power of neural networks to the finger tips of programmers, with a minimum amount of code to generate. Enjoy your journey through the elements of the KBCC library, and remember this simple advice: try it yourself, explore, and have fun !

System requirements

Recommended system requirements for a typical Java development environment (e.g., JBuilder) include 256 MB of RAM. These environments can still be used with less memory, but you might experience delays in the display of items such as auto-completion. The minimun recommended screen resolution is 1024x768. This is mainly because development environments feature a number of split screens that each take up space on your display screen.

Memory requirements for the code library itself vary according to the size and complexity of the tasks you intend to run. Small tasks of 10 inputs or less and a limited number of patterns should not require any extra memory. However, very large tasks (>100 inputs) will require more RAM from your machine and more PCU time to complete. For instance, a serie of 100 DNA analysis networks each with 240 inputs and 1000 patterns completed in about five days on a Dell machine with a 1.7 xion chip and 1G of RAM. We often suggest that developers interested in these large implementations dedicate a machine for the purpose of running simulations.

Symbols

The following symbols will be used throughout this tutorial:

This is the symbol for an important point that is crucial for your code to run well. It should never be ignored.

The magnifier symbol announces additional information that complements the text, but is not essential to the basic understanding. These bubbles can be skipped upon the first reading without affecting comprehension.

Java code will be represented in the following font:

```
FunctionalUnit sampleUnit
```

1. Basic elements of the library

This chapter covers:

- the XOR problem
- the DataSet
- the FunctionalUnit
- the Analyzer
- the Statistic
- data encoding

Various simulations in the library all contain a number of elements in common which constitute the basic classes for the entire library. In this chapter, we will provide some basic descriptions of these elements. First, we will define a basic problem called the XOR problem that is implemented in every tutorial openly available from the library. Second, we will define basic data handling stragety through the introduction of the DataSet class. This is the class that stores incoming and outgoing data in training a network. Third, we will define the basic FunctionalUnit type that encapsulates virtually every function in the library. Fourth, we introduce the Analyzer, a class that serves to calculate various statistics during training time. Fifth, we talk about statistics in the library, and define how you can use commonly available ones or put your own together. Finally, we discuss the issue of data encoding, in a section entended mostly for new users of neural networks.

The XOR problem

The problem we will used to illustrate basic simulations in Backprop and Cascade Correlation tutorials is the classic boolean XOR (exclusive-or) problem. The input and target values of this problem are presented in Table 1.

Table 1

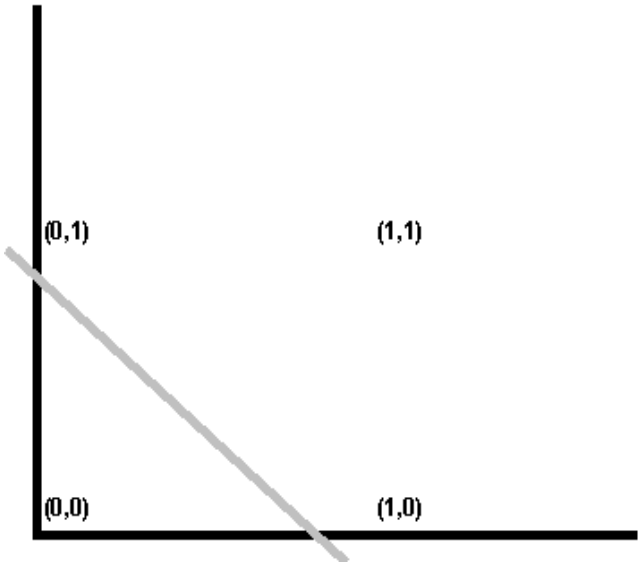
OR and XOR Functions

Node 1	Node 2	Output activation	
		OR	XOR
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

The network is trained to return 1 if the input bits are different, and 0 otherwise. This problem poses a particular challenge to neural networks. In order to solve a classification problem, a network will try and divide the input space into a number of hyperplanes that best represent the classification imposed by the output. If the problem was a simple OR problem, it could be solved easily by a single division of the inputs, represented as coordinate points in a Cartesian map (see Figure 1).

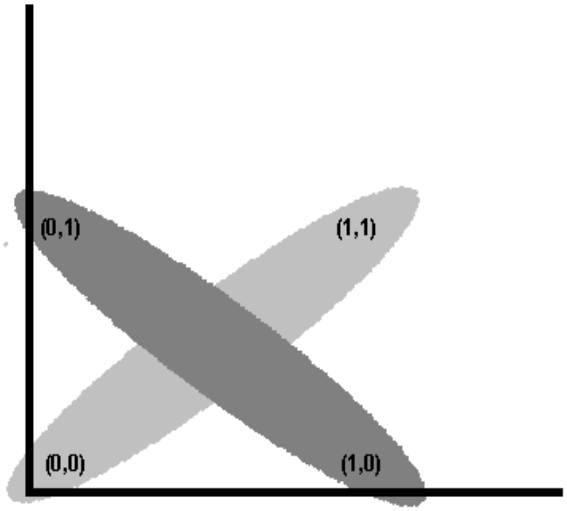
Figure 1

Partitioning input space for the OR problem



In the case of the XOR problem, the network is unable to reach a solution using a single straight line to divide the inputs (see Figure 2). In order to solve this problem, the network has to create an internal representation of the problem. This intermediate representation must shift some of the elements in the original input space in order to accomodate partitioning. For instance, a possible solution might include shifting the element at position (1,1) to a point closer to the element at (0,0), so that a straight line can be drawn between the two categories present in the problem.

Figure 2
Partitioning input space for the XOR problem



The DataSet

The DataSet structure is based on a standard hash table, mapping keywords to various objects. A hash table is a basic data structure that maps string names to objects of various types. Similar structures are avaible from many of the most common programming languages, include CMAPs for C++ using MFC, and a-lists for the LISP language. The hash table operates storage without consideration of the order in which elements are placed in the hash table. This enables quick retrieval times of objects. This property of hash tables makes them a great tool to handle large banks of data such as the ones we will require in dealing with neural network simulations. An simple example for use of a hash table could be to store students' names with their grades at a recent exam. To store an item, you would associate a student's name with his score, and store it in the hash table. Retrieving

the score of a student would require a simple table lookup with the name of a student. The lookup would return the score associated with the name of the particular student.

In the KBCC library, the DataSet is the structure of choice for storage for a great amount of information directly related to the running simulations. The two most important function of this class are getData and setData, which respectively retrieve and store data in a DataSet object. DataSet always retrieves and stores data of type Object, which is the most basic type for any Java objects. This storage strategy has some pragmatic implications for data handling. When storing data, getData downcasts the argument to store to a type Object. "Downcasting" means that it now thinks of your argument as type Object, and nothing else. This makes it easier to store and retrieve in the hash table, because you don't have to associate the type with every stored object. At retrieval time, however, setData will return your stored object as a type Object. This is unfortunately not very good if you expect a type Double to be able to do computations with. So you have to explicitely perform the opposite strategy as the hash table does at storage time. Because the hash table downcasted you object, you must now upcast it. Admitedly, this can sometimes be tricky, because you have to remember what was the type of your object prior to storage. If you don't upcast to the right type, you will more than likely raise an exception at run time. This exception might show up as something like Java.lang.CastException:]]]D, for instance. This means that the stored object cannot be upcasted to the type you provided. All of this might seem a little nebulous, so let's look at some code. First, here's the parameters for the setData method:

```
[String name] [Object object]
```

The first argument is a name that identifies the object to be stored. The name of a student, for instance, could be passed here as a string. The hash table will only store a single entry per string name. This means that if you call setData using the same name twice, DataSet will overwrite the first call with the second. Upon retrieval time, DataSet will return the last value you associated with the string name. The second argument to setData receives an object of the general type Object. As mentionned, the class Object encapsulates all other class types in the Java language. Practically speaking, this means that virtually any object type can be passed as second argument. DataSet simply

casts that argument down to the Object type. Let's take a concrete example. To store a student's grade with his name, you can simply do the following. First, create a new object of type DataSet:

```
DataSet grades = new DataSet();
```

DataSet contains a number of alternative constructors. In one of these, you can specify the initial size of the table to be built. In most cases, you don't have to worry about the final size of the table when creating a new DataSet object. This is because DataSet grows as necessary in size to accomodate new data. Once the object is created, you can simple call setData as many times as necessary to store all the students' grades:

```
grades.setData("JP Thivierge", 8.9);
grades.setData("Tom Shultz", 9.2);
grades.setData("Fred Dandurand", 9.4);
```

In the KBCC library, there are specific string names already defined for most common types of data that will be inputted or outputted of networks. All of these names are stored under a dedicated class in the library named DataNames. This class contains exclusively public String members that define different types of data. Here are some frequently used examples:

```
public static final String PATTERN_COUNT = "PatternCount";
public static final String INPUT_PATTERNS = "InputPatterns";
public static final String OUTPUT_PATTERNS = "OutputPatterns";
public static final String DERIVATIVES = "Derivatives";
public static final String SECOND_DERIVATIVES = "SecondDerivatives";
public static final String TARGET_PATTERNS = "TargetPatterns";
public static final String ERROR_PATTERNS = "ErrorPatterns";
```

Members of the DataNames class are designed to set a standard for commonly used data name String' s in the library. You are *required* to use them when available. If you intend to build upon the existing library, notify the build master of new DataNames to add.

To use a member of DataNames, you can simply call it inside the setData method:

```
someData.setData(DataNames.INPUT_PATTERNS, inputPatterns);
```

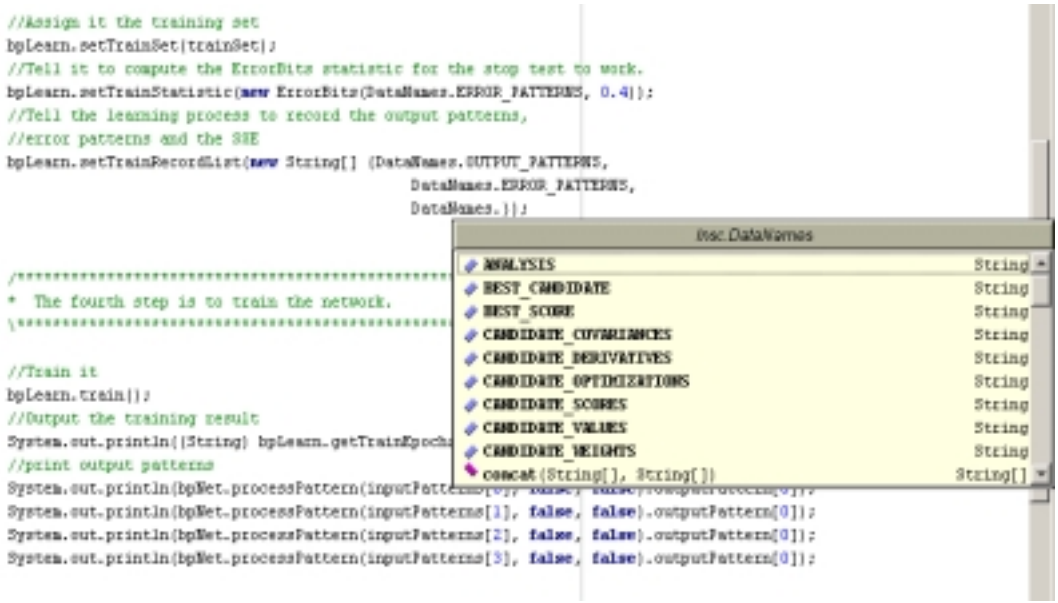


Figure 3. Example of an auto-completion tool taken from JBuilder[tm]

It comes in handy to use the auto-completion tool available in most environments to search the DataNames class. This tools enables the use to view all the members of a class, including methods and fields. See Figure 3 for an example of an autocompletion menu. For a view of the primary values that can be stored in a DataSet during a simulation, refer to the String members of the DataNames class. The keywords contained in this class will give the developer an idea of the type of information he could require to run and analyze a simulation.

DataSet objects are an essential part of running and analyzing simulations in the KBCC library. In what follows, we will elaborate on the multiples uses of DataSet in your experiments. First, objects of type DataSet are used to store what will be going into the network, namely the training and testing patterns. This is a rather standard procedure in the library. Training and testing patterns are stored in a data set. This data set is always defined by three main elements, namely input patterns, target patterns, and pattern count. The first element, input patterns, refers to the values that are inserted in the network. Because there is typically a number of inputs nodes, input patterns typically take the form of vectors. The latter is defined as a single input pattern. Each element of the vector is associated with a particular input pattern. In this optic, it is crucial that the length of each vector match exactly the number of input nodes in the network to train. View Figure 4 for a schematic view of this.

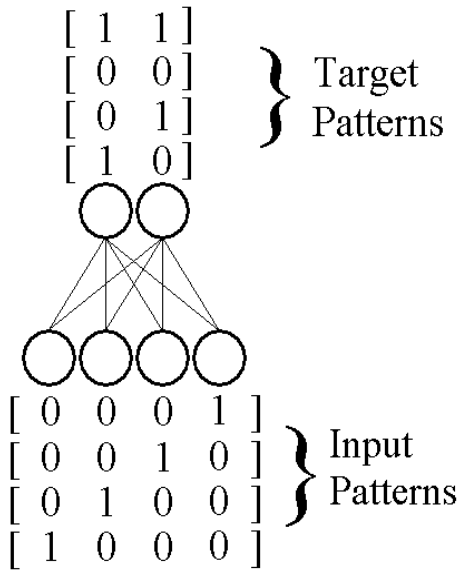


Figure 4. Schematic display of network input and target patterns stored in a DataSet.

The second element, target patterns, deals with the values that act as a teacher signal in training. Target patterns are used in supervised learning, which is the main purpose of the algorithms in the KBCC library, including Cascade-correlation and Backpropagation. As shown in Figure 4, the lenght of target patterns must necesarily match the length of output nodes in the network to train. As

another contraint, the number of target patterns must match the number of input patterns, and each target pattern in an ordered array of target patterns must correspond to an input pattern of same index in an array in input patterns.

The third element, referred to as pattern count, serves to tell the system how many patterns are going to be presented in total. This is a single number that must reflect exactly the number of patterns contained in input patterns and target patterns. In short, input patterns, target patterns, and pattern count are the basic elements that define any data to train using a neural network in the KBCC library. It is the user's responsability that they are defined and that they remain logical in terms of number of patterns, and correspondance with network architecture.

A second use of DataSet in the library that is quite useful is to store data relevant to a simulation, including network training error, speed of performance, internal activations, etc. DataSet can be used to monitor the performance of a network during training for various statistics such as sum squared error and number of epochs, and during testing for error on new data.

The Functional Unit

The most fundamental aspect of the library to remember is that everything that is a function is encapsulated by the FunctionalUnit basic class. Functions come in various flavours when dealing with neural networks. First, there are basic one-to-one functions that serve as activation functions. The following Table gives a description of the most widely used of these:

Table 2

Description of simple units

Type	Class Name	Example of use
Gaussian	GaussianUnit	RBF networks
Linear	LinearUnit	Hidden-output connections
Sin	SinUnit	Hidden unit activation
Hyper-tan	TanhUnit	Hidden unit activation
Threshold	ThresholdUnit	KBANN networks

Functional units are at the very basis of the KBCC library. They are used systematically for everything that is a function. At the basic level, the interface class FunctionalUnit defines the framework for dealing with multi-variate, real-valued functions. This means that FunctionalUnit can serve to implement functions with varying types of inputs and outputs in R. We will now present various uses of FunctionalUnits in the KBCC library, which will demonstrate the omni-potence of this class.

As mentionned, the FunctionalUnit interface can be used to define activation functions of neural networks. An activation function typically acts on a single unit (input, hidden, or output). An activation function takes the summed product of weights times input as a single value, and returns a single value. The function typically serves as a threshold function on its input. To implement this, the class FunctionalUnit is extended by AbstractSimpleUnit, a class that takes only one input and returns only one output. The framework for the AbstractSimpleUnit enables the user to specify a function, its derivative, and its second derivative using the corresponding methods. In addition, two properties of the basic function can be adjusted, namely the offset (origin of the function) and factor (spread of the function).

Second, at a broader level, FunctionalUnits define any type of networks in the library, including multi-layer networks and Cascacade-correlation networks. Thinking of a network as a function facilitates a lot of computations, for instances when it comes to recruiting sub-networks in KBCC, or pruning entire networks in Optimal Brain Damage. The way networks are integrated into FunctionalUnits may not be of immediate importance for the novice who is still experimenting with various aspects of the library. However, it is a good thing to keep in mind as it will becomes important for many future developments. In a sense, a neural network in the library can be viewed as one big FunctionalUnit composed of a number of connected smaller FunctionalUnits.

Eventually, developers of the library intend to redefine the weights connecting FunctionalUnits as FunctionalUnits themselves. In fact, weigths *w* simply provide a multiplicative factor for some given activation input *a*: $\sum_{ij} w_i a_j$. Abstracting weights into FunctionalUnits would provide us with networks that are essentially big building blocks of layers and weights.

Any type of network is in itself considered a differentiable function, and also has as basic type the FunctionalUnit interface. Table 3 presents the basic network types available from the library.

Table 3

Description of general network types

Type	Class Name	Example of use
Cascaded networks	CascadeNetwork	CC and KBCC networks
Multi-layered networks	MultiLayerNetwork	Backprop networks
General networks	WeightBasedNetwork	any*

*note: this class is still in the works, and intends to implement a general framework for any type of network with weighted connections of layers, including self-organizing networks and recurrent networks.

Analyzer

The analyzer interface offers a framework for analyzing networks during or after training. An example of an analyzer is the functionality implemented for pruning networks (see chapter 10). A typical analyzer required the following:

- a network of basic type FunctionalUnit
- a DataSet containing DataNames of elements to record during a given training epoch
- a DataSet containing DataNames of elements to record during a given testing epoch

Note that the values used for the two DataSets may optionnally be set to null if nothing is to be recorded.

Table 4

Description of statistics

Statistic/Properties	Type	Name (default)	Input Name	Input Type	Statistic	computeStatistic DataNames
AbstractStatistic*	N/A	N/A	N/A	N/A	N/A	
ErrorBits	Integer	ErrorBits	User defined	double[][]	$y = \sum_{i,j} \left(\lvert x_{i,j} \rvert > ScoreThreshold \; ?1 : 0 \right)$	
SumSquared	Double	SumSquared	User defined	double[][]	$y = \sum_{i,j} x_{i,j}^2$	
MeanSquared	Double	MeanSquared	User defined	double[n][m]	$y = \frac{1}{n \cdot m} \sum_{i,j} x_{i,j}^2$	SumSquared
TRS	Integer	TRS	User defined, user defined	double[p][], double[p][]	$y = \sum_{i=1}^{p} \left(wif(\vec{a}_i) = wif(\vec{b}_i) ?1 : 0 \right)$ [1]	
MeanVector	double[]	MeanVector	User defined	double[p][]	$\vec{y} = \frac{1}{p} \sum_{i=1}^{p} \vec{x}_i$	
CovarianceMatrix	double[][]	Covariance	User defined, user defined	double[p][], double[p][]	$y_{i,j} = \frac{1}{p} \sum_{k=1}^{p} \left(a_{i,j} \cdot b_{i,j} \right) - \vec{mv}(A)_i \cdot \vec{mv}(B)_j$ [2]	Means
FrobeniusNorm	double[][]	FrobeniusNorm	User defined	double[n][m]	$y = \sum_{i,j} x_{i,j}^2$	

1- Input type double[][] stands for array of vectors or matrix.

* These classes are abstract and can' be directly instantiated, they must be derived.

Statistic

The statistic interface provides some basic functionality for computing a statistic on a given array of data. Table 4 provides the complete list of statistics implementing this interface.

We have already seen in past chapters how to record sum squared errors during learning as well as during testing on a new data set. The popular sum squared error, for instance, is based on an interface class named Statistic, which is designed to compute a given statistic, and return a data set containing some values stored

under specific keys. The main advantages of the Statistic interface are it's simplicity and its flexibility. This is due to the fact that Statistic does not specify anything about what gets computed. This is done when extending the abstract implementation of Statistic AbstractStatistic for your own purposes. If you want to define a new statistic, you must extend the AbstractStatistic class. At that point, you will be mainly interested in the computeStatistic method. Here are the arguments required for computeStatistic:

[DataSet dataSet] [String[] recordList]

First, a DataSet is passed containing some data. For instance, to compute sum squared errors, the DataSet object could contain a double[][] of output patterns stored under DataNames.OUTPUT_PATTERNS, and a double[][] of target patterns stored under DataNames.TARGET_PATTERNS. Then, you must provide some names of stuff to record when computing the statistic. For instance, DataNames.SUM_SQUARED_ERROR could be included as second argument to computeStatistic for a statistic that records sum squared errors. When processing a certain data set to compute a statistic, computeStatistic is expected to store values corresponding to the recordList in a new DataSet that is returned just before the method returns.

If you are building new Statistics, there is a current trend in the library we wish to maintain. An extensive class of the new Statistic can be implemented that implicitely stores a number of elements if requested. Then, for more basic users, this new class is derived and more basic and user-friendly statistics are created. This enables for quick use of most commonly used statistics.

Data Encoding

The first consideration for running a network on any given problem is encoding. There are a number of different possible encoding schemes, the most popular of which are unary, binary, and continuous. Unary encoding consists in assigning a bit for every possible value of a input pattern. For example, suppose a problem of recognizing the DNA bases A, C, T, and G. Encoding these letters using unary bits could result in:

A = 0 0 0 1

C = 0 0 1 0

T = 0 1 0 0

G = 1 0 0 0

The assignment of the "1"'s is purely arbitrary, as long as every letter value gets its own specific assignment position. Binary encoding is somewhat similar to unary, but can be considered more compressed:

A = 0 0

C = 0 1

T = 1 0

G = 1 1

In this case, the idea is to find the least amount of bits that can represent variations in the input space. For 4 possible values of inputs and 2 possible values per encoding bit, the number of bits *n* necessary per pattern follows $4 = 2^n = 2^2$. Finally, continuous encoding is usually defined by the data set itself. It refers to values in R that can take any values between a given range. Percentages of red cell concentrations in blood, for instance, are continuous values. Output patterns follow the same encoding schemes as input patterns. To date there are no formal tools in the KBCC library for performing data encoding of a data set.

Analyzing network performances

In his PhD thesis, Waugh (1995) proposes the following measures for neural networks:

1- classification accuracy on the train set

2- size of the final solution, in terms of nodes and connections

3- speed of learning on the train set, in epochs, number of phases, or PCU time

4- explanation ability of the final theory

5- ability of the network to stand up to partial corruption

A number of possible statistics exist for (1) in the library. A summary of these is presented in Table 5:

Table 5
Statistics used to analyze network classification accurarcy

Data to collect	Java classes or methods
- Sum squared error	- SSE
- Mean squared error	- SumSquared
- Relative Information score	- MeanSquared
- ROC curves	- ROCCurve
	- ErrorBits

Testing networks for generalization is usually performed using a separate data set composed of examples never seen by the learning system. The most widely accepted way to split a data set into training and testing sets is by *n*-fold cross-validation. In this technique, the original data set is split into *n* independent subsets, and a total of n networks are trained. Every time a new network is initialized, it is trained on *n*-1 subsets, and testing on the n^{th}. A popular value for *n* in the litterature is 10. The KBCC library contains a method in the Tools class to generate cross-validation sets from a data bank. This method takes two arguments: (1) a data set containing entries for DataNames.INPUT_PATTERNS, DataNames.TARGET_PATTERNS, DataNames.PATTERN_COUNT; (2) an integer specifying the number of folds in which the data is split. In return, the method returns an object of type DataSetCollection from which the individual folds can be retrieved as follows:

```
DataSet trainData[i] = (DataSet) crossValidationData.getData("TrainSet", i);
DataSet testData[i] = (DataSet) crossValidationData.getData("TestSet", i);
```

One advantage of running networks using cross-validation is that they are more easily analyzed afterwards by statistical tests. When a number of networks are trained using n-folds, they can be treated statistically as independent, and analyzed accordingly. For paired independent samples, pairwise t-tests are among the simplest methods (see for instance Shultz and Rivest, 2000). For added robustness, it is recommended to train a number of n-fold simulations. In Thivierge and Shultz

(2002), for instance, 10 independent simulations were set up to each run a 10-fold simulation, resulting in a final count of 100 networks. By using more data, significance of the results can be more convincing.

2. Multi-layer networks

As opposed to Cascade-correlation networks, Multi-layer networks in the KBCC library have a fixed architecture, which means that they are unable to grow as they go through the learning process. Also, Multi-layer networks do not have any cross connections between layers. This means that the input layer as well as every hidden layer is fully connected to the layer directly above it, but not to any other layers. Finally, another difference between Cascade-correlation networks and Multi-layer networks is that the latter can have more than one hidden node per layer. Cascade-correlation, on the other hand, creates a new layer for every new hidden unit recruited. As in Cascade-correlation, Multi-layer networks have bias units. In particular, all layers below the output have a bias going into the next layer. For instance, the input layer has a bias unit going into the first hidden layer, and the first hidden layer has a bias connecting it to the layer above it. If the network only has a single hidden layer, the hidden layer bias connects directly to the output layer. If the network has more than one hidden layer, each hidden layer has a bias connecting it to the next hidden layer, and the last hidden layer has a bias connecting it to the output layer. Each bias connection has a default activation of 1, which remains fixed throughout the learning process. The following figure depicts a typical Multi-layer network with 2 hidden layers:

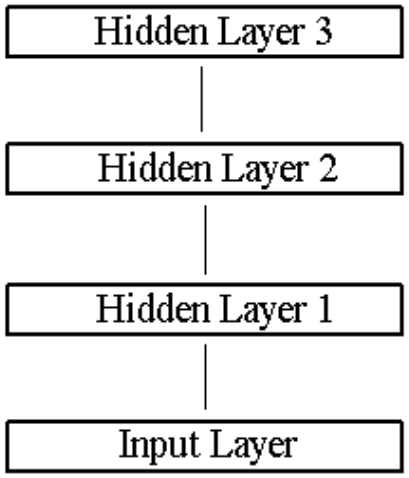


Figure 1. Architecture of a Multi-layer network with 2 candidate recruits.

As with Cascade networks, the first step to running a Multi-layer network is to define a train set containing the input patterns, the target patterns, and a count of the number of patterns in the set. For a discussion of this process, please review chapter 2 on running a simple simulation. There is no difference between defining a train set for a Cascade network and for a Multi-layer network. Once the train set is defined, the next step involves creating a new network of type

MultiLayerNetwork, and initializing it with the following constructor:

```
MultiLayerNetwork(int newInputCount, int[] newHiddenCount, FunctionalUnit
sampleHiddenUnit, int newOutputCount, FunctionalUnit sampleOutputUnit)
```

The first argument of the contructor, newInputCount, defines the length of the input patterns. The second argument, newHiddenCount, sets the number of hidden units of the new network created. This number will remain fixed throughout training. For instance, a single hidden layer of 4 units translates as:

```
new int[] {4}
```

Here is example of a initialization of three layers of hidden units, with 2 units on the first layer, 4 units on the second layer, and 6 units on the third layer:

```
new int[] {2, 4, 6}
```

Notice that, in this case, the number of nodes in each layer must be defined separately. For the third argument, sampleHiddenUnit, the user must give an example of the type of hidden units the network will have. All possible types of units are derived from the basic class FunctionalUnit. There are a number of different types of units readily available from the library. These include the logistic, gaussian, sin, and hyper-tan. For more on these units and on the FunctionalUnit class, please jump to chapter 7. For example, here is a logistic unit (also called sigmoid) with a span of one and an offset of -0.5:

```
new LogisticUnit(1.0, -0.5)
```

The fourth argment of the MultiLayerNetwork constructor, newOutputCount, is the number of outputs the network will have. This number should match exactly the number of target values per pattern in the task presented to the network. For example, if a task has four target values, the network should have four output nodes. This is true for all types of networks in the KBCC library. The last argument is another FunctionalUnit, this time representing the type of units at the level of the output units. The type of functional unit passed here doesn't have to match the one passed for the hidden units. For instance, there are no restrictions against using logistic units at the hidden level and sin units at the output level. Often times, it is wise to adapt the type of units used to the nature of the task at hand. For more on this topic, please refer to chapter 7.

That's it ! We have defined the necessary elements to obtain a multi-layer network. Here is an example of such a network with two linear input units (??), two sigmoid hidden units of range 1.0 and offset -0.5, and a single sigmoid output unit of range 1.0 and offset zero:

```
MultiLayerNetwork bpNet = new MultiLayerNetwork(2, new int[] {2},
                          new LogisticUnit(1.0, -0.5), 1, new LogisticUnit());
```

Once a network has been created, the next step is to define what criteria will stop the training of

the network. Will it be a set number of epochs? Or maybe you want the network to reach a certain minimum of classification error ? The stopping test to be created is of type GenericTest. This interface class encapsulates all tests performed on a training or testing session. The possible tests are of the following type:

- DisjunctiveTest: creates a test that is a combination of other tests, using an OR. This means that, out of a number of possible tests, the first one that passes makes the whole DisjunctiveTest exit.

- IntervalTest: checks whether a number is inside a certain interval.

For the purposes of training a Multi-layer network, a disjunctive test will do the work. Here, for instance, we can specify that we want the learning process to iterate until a certain error is reached or a certain number of epochs has passed, which ever comes first. In this way, a network that is incapable of converging will still terminate its learning, and a network that quickly converges to a solution will terminate its learning right away. To implement this disjunctive test, we must create a new instance of a DisjunctiveTest. The constructor to the DisjunctiveTest requires two parameters.

```
DisjunctiveTest(GenericTest[], String[])
```

First, you must provide an array of tests to perform. Each of these take the form of a GenericTest. Second, you have to pass as second argument to the constructor of DisjunctiveTest a String array that names each of the tests defined in the first argument. For instance, if in the first argument we define a first test for the error and a second test for the number of epochs, the second argument could receive a String array containing “Minimum error reached” for the case where the error test passes, and “Maximum epochs reached” for the case where the maximum number of epochs were reached. All this will become a lot clearer with an example. But first, we need to have a closer look at the constructor for the IntervalTest class:

```
new IntervalTest(String, Number, Number)
```

Remember that the goal of the interval test is to check whether a given number falls between a certain interval of values. To do this, the constructor of the class requires a value to check. This value is defined by the first parameter. Using DataNames, it is possible to retrieve a large number of statistics from the training process. The error bits, for instance, are retrieved by the keyword STAT_ERROR_BITS. For more on using DataNames, please consult chapter 2.

It is possible to have an interval test with an open interval (i.e., an interval where one of the values is infinity). This is done by setting one of the interval values to null. Be careful however not to set both interval values to infinity !

Once we put everything together, the final result for our stop test could be something like this:

```
GenericTest stopTest = new DisjunctiveTest(new GenericTest[]
                        {new IntervalTest(DataNames.STAT_ERROR_BITS, null,
                        new Integer(0)),
                        new IntervalTest(DataNames.EPOCH, new Integer(1000), null)},
                        new String[] {"SUCCESS", "FAILED"});
```

In this particular case, we want training to stop if the error bits have dropped to zero, or if the epochs have reached 1000. Whichever of these events come first will stop the learning process. To help us analyze the learning process, the system is told to output "success" if the first test passes (zero error), or "failed" if the second test is the one that ends training (1000 epochs have been reached).

Once we have established our stopping criterions, the next step consists in defining a learning method that will train the network towards its goal. The fact that the KBCC library is fully modular enables the user to select networks and learning schemes independently. Many learning schemes are currently available in the library, including the classes BatchErrorCorrectionLearning, CascadeCorrelationLearning, CascadeErrorCorrectionLearning, ErrorCorrectionLearning, as well as a number of specific Cascade Correlation learners. For the purposes of setting up a backprop simulation, we will restrict our current demonstration to the BatchErrorCorrectionLearning. This learning scheme applies the basic delta rule and error backpropagation principles that made backprop famous. Batch learning means that the network stores all patterns in a learning epoch before adjusting its weights. The main advantage of batch learning is that it offers a partial solution to the problem of the moving target. Batch learning restricts the oscillation of weights from pattern to pattern by postponing weight updates until all patterns in a data set have been seen.

To create the BatchErrorCorrectionLearning scheme, one simply has to create a new object of type `BatchErrorCorrectionLearning`. The constructor for this object has the form:

```
BatchErrorCorrectionLearning bpLearn = new
        BatchErrorCorrectionLearning(MultiLayerNetwork, Optimizer, GenericTest, boolean);
```

As always, let's take each argument at a time. The first of these is an instance of a MultiLayerNetwork on which the learning will be applied. In our case, we will fill in this argument with the network already defined as bpNet. For the second argument, an Optimizer is required. The optimizer is the core of a learning algorithm. It is the means by which our network will adjust its weights in order to reduce error at the output level. In the specific case of our backprop simulation, the optimizer's purpose is to implement the delta rule necessary to perform weight updates. In the KBCC library, this optimizer goes by the name of Backprop. Once we have set our network and optimizer in place, the essence of the batch learner is defined. Only a couple more items are necessary. The third argument to the learner is a stop test that tells it when to stop training. The learner is basically implemented as an infinite loop (e.g., while (true) or for (;;)). Without a stop test, it will go on training indefinitely. Thus, the third argument of the learner receives the generic test we have previously defined as stopTest, which tells the network to stop if it has minimized its error, or has reached a given number of epochs. This stop test will be

evaluated every time weights are updated in the training process. Finally, the last argument unables the user to define which type of training error it wants the network to compute and record. There are two options here: sum squared error (SSE) or mean squared error (MSE). Both of these convey the same information, but under different formulas:

$$SSE = \sum_i (t_i - o_i)^2$$

$$MSE = \frac{1}{n}\sum_i (t_i - o_i)^2$$

Put in words, mean squared error averages the error over all patterns. For mean squared error, set the last argument in BatchErrorCorrectionLearning to true. Conversely, for sum squared error, set this argument to false.

It is quite possible to compute both of these values if so needed. One way to do this would be to compute the sum squared errors throughout training, and obtain the mean squared errors afterwards. Suppose you have stored the sum squared errors in double[] SSE, here's how to obtain mean squared errors (MSE) for a given SSE_i using your initial training set:

```
MSE[i] = SSE[i] / ((Double[][])
        trainSet.getData(DataNames.INPUT_PATTERNS)).length;
```

Now that our learning scheme has been defined, we are almost ready to start training the network. But before we do so, a minimum of two elements are necessary to set in the our bpLearn, namely a train set, and an error statistic that will serve to compute the stop test. First, we must tell the network to train using a specific data set. This is fairly straighforward using the setTrainSet method in BatchErrorCorrectionLearning:

```
setTrainSet(DataSet);
```

If you are not familiar with data sets, please review Chapter 2 on setting up a simple simulation. Finally, we must tell our learner what type of error statistic will be used to test for a stopping criteria. This is done using the setTrainStatistic method in BatchErrorCorrectionLearning:

```
setTrainStatistic(Statistic);
```

A statistic is a very basic class of objects describing a number of measures (for more on Statistic and its inhereted members, see Chapter ??). For the purposes of our current simulation, we want the learner to use the ERROR_BITS statistic. Unlike sum squared error, this statistic is not interested in an exact figure of the discrepancy between target and output vectors. Rather, it is concerned with a more broad measure of data fitting. Error bits will consider that an output bit has been well classified if it is within a given range of its corresponding target bit. This range can be defined by a single threshold value applied at both positive and negative ends of the range.

Let's take an example to illustrate. Be a network with sigmoid units at the output level scaled between [0, 1]. These units are trained to approximate some target bits of range [0, 1] as well. Image that an output bit returns a value of 0.18, and that the target value is 0.0. Is that a misclassification? Well, this depends on how strict we are about error in the system. If we set an error threshold of 0.3, the output bit would fall under the threshold and be classified as a hit. However, if our error threshold is set below the given output value (say, 0.1), then that given output would be classified as a miss. For the purposes of our current example, we will set a threshold that is not restrictive at all, with a value of 0.4. Our ErrorBits statistic thus looks like this:

```
ErrorBits(DataNames.ERROR_PATTERNS, 0.4));
```

which results in our setTrainStatistic method being called as:

```
bpLearn.setTrainStatistic(new ErrorBits(DataNames.ERROR_PATTERNS, 0.4));
```

Finally, if you are ready to train the network, this can be performed using the simple method train(), which takes no arguments. However, prior to calling this method, a number of other options can be called through various methods (??name all of them??). One of them consists in defining exactly what values get recorded as the training rolls on. The method to achieve this is setTrainRecordList. This method takes a single String[] argument of names of data to record. A simple example will illustrate this:

```
bpLearn.setTrainRecordList(new String[] {DataNames.OUTPUT_PATTERNS,
                                         DataNames.ERROR_PATTERNS,
                                         DataNames.SUM_SQUARED_ERROR});
```

In this example, the learning process is told to record output patterns, error patterns, and sum squared error values. These values are stored in a data set that can retrieved when training is completed by calling getTrainDataCollection, as in:

```
Object[] SSEs = bpLearn.getTrainDataCollection(DataNames.SUM_SQUARED_ERROR);
```

As illustrated, the getTrainDataCollection method of bpLearn simply requires a DataName String that corresponds to a data element requested in setTrainRecordList. The only tricky part is to know how to upcast what is returned from getTrainDataCollection. Because the data storage in learning is stored using DataSet, which is a derivative of a native hash table in Java, everything is stored as an Object, the most general class for every instance in Java. Although the initial downcast is a no-brainer, upcasting the returned values necessitates knowledge of the specific types stored in the hash table. Sum squared error, for instance, is an array of type double. In this case, the proper upcast would be:

`(Double[]) SSE`	or, equivalently
`(Double) SSE[i]`	where i is an index of a given epoch (as mentionned earlier, SSEs are recorded at the end of an epoch).

For a complete list of data types and upcast, consult the chapter ?? on datasets.

Once training is completed, a typical thing to do is to test the network on novel data. For this, make sure to define a separate data set in the usual routine:

```
DataSet testSet = new DataSet();
testSet.setData(DataNames.INPUT_PATTERNS, double[][] someInputPatterns);
testSet.setData(DataNames.TARGET_PATTERNS, double[][] someTargetPatterns);
testSet.setData(DataNames.PATTERN_COUNT, double somePatternCount);
```

make sure to specifiy at least these three items every time you create a new train or test set: the input patterns, the target patterns, and the number of patterns in the set. In the case where you want to pass some unlabeled patterns through the network and use it as an expert system, you can drop the target patterns. To test the network using your novel data set, use the following method of the MultiLayerNetwork object bpNet:

```
DataSet someRecordedTestData = bpNet.processDataSet(DataSet testSet, String[] {/*names of elements to record*/});
```

Testing works similarly to training the network, in the sense that you must pass a data set and a list of data to record to the system, and you get a data set back containing the data you asked to have recorded. If you don't have a full data set to test, you can always just process a single pattern through the network in the following way:

```
bpNet.processPattern(double[] someInputPattern, boolean computeDerivative, boolean computeSecondDerivative);
```

From processPattern, you can obtain a double[] of output unit activations in the following way:

```
bpNet.processPattern(double[] someInputPattern, boolean computeDerivative, boolean computeSecondDerivative).outputPatterns[];
```

It is also possible to obtain first and second derivatives by setting the appropriate boolean values to true and calling either of:

```
bpNet.processPattern(double[] someInputPattern, boolean computeDerivative, boolean computeSecondDerivative).firstDerivatives[];
```

or

```
bpNet.processPattern(double[] someInputPattern, boolean computeDerivative, boolean computeSecondDerivative).secondDerivatives[];
```

The goal of this chapter was to get you started on running, analyzing, and testing simple backprop networks. For more indept look at aspects of the KBCC library that can help with alternative experimental setups, please consult corresponding chapters. If you wish to experiment with the concepts and elements presented in this chapter, the library contains a BPTutorial class that sets up

a simple simulation on the XOR problem using the techniques we presented here.

3. Cascaded networks

In this chapter, we set up a simple simulation in Cascade-Correlation (Fahlman & Lebiere, 1989). It is recommended that the reader follow the steps presented and implement the simulation described. Be warned that this chapter assumes some knowledge of the basic functioning of the Cascade-correlation algorithm as described by Fahlman and Lebiere (1989). If you are unfamiliar with this algorithm, it is important that you refer to the last chapter of this tutorial, where you can find a link to a basic slide show tutorial set up by Thomas Shultz. Once you are familiar enough with how the algorithm proceeds to learn, you are ready to read on.

The problem we will use to illustrate a basic simulation is the classic boolean XOR (exclusive-or) problem. A full description of this problem is available from the previous chapter. Here are the necessary definitions to run a CC simulation on the XOR problem:

- a DataSet with corresponding input and target values
- a Cascade Network
- a Learning process
- a recordlist of values to output during training

First, a training set is created by a double[][] array containing the real or binary values to enter the network. A matching double[][] array of output patterns is also created.

The input and output arrays are added to a DataSet object, which acts like a hash table, by matching keywords to corresponding values. The input and target patterns are set under the keywords “InputPatterns” and “TargetPatterns” respectively, using the setData method.

IMPORTANT: the input and target arrays must be of same lenght in x as double[x][y]. The y value, on the other hand, will determine the number of input and output nodes in the network, and do not have to be of same size.

A number of Java classes are given to faciliate the streaming of input and target patterns to your program. Remember however that pre-processing of your data may be necessary.

Once a data set is created, the next step is to create a new Cascade Network that will be trained to the data set. This procedure simply involves creating a new object of the class CascadeNetwork, with the following parameters in the constructors:

```
[int input count] [int output count] [FunctionalUnit sampleUnit]
```

The first two parameters are already known from the training set. The third parameter defines the type of units that will compose the network. This tutorial initiates a cascade network with logistic units. The optional parameters for the initialization of a new instance of LogisticUnit are

[double newFactor] [double newOffset]

For problems with values ranging from [0,1], the offset of the sigmoids must be shifted to -0.5. This is the typical sigmoid unit found in a large number of neural networks. Other types of units can be found in the library, including linear units and hyper tan units.

Besides logistic (sigmoidal) units, other types of units can be used depending on the type of problem you are training the network on. Unpublished lab research by Francois Rivest (2002) has shown that networks benefit more from sine units when trained on problems that present fairly constant repeating patterns. The problem that was used was the classic two-spiral problem of Fahlman. In this problem, networks must learn to discriminate between two interlocking spirals. Networks that used sine activation units at the hidden unit and/or output level learned the problem with higher accuracy than network with logistic activation units.

After a network is created, the next step involves creating a learning process by which the network will be trained. In this regard, a number of learning processes are available, including batch learning for multi-linear feed forward networks (BatchErrorCorrectionLearning) and cascade-correlation learning (CascadeCorrelationLearning). The learning process used in the tutorial is the FahlmanCascadeCorrelationLearning. The construction parameters for creating a new instance of class FahlmanCascadeCorrelationLearning are as follows:

[CascadeNetwork newNetwork] [int newCandidateCount] [FunctionalUnit newSampleCandidate]

The first parameter involves the network created at the previous step. The second parameter defines the number of candidate units to be placed in the pool at the beginning of each input phase (if you are uncertain about the terms “candidate pool” and “input phase”, you may want to review the Cascade-Correlation Tutorial). As for a lot of parameters concerning neural networks, there are no hard and fast rules for setting the number of candidates. Note however that training times will increase as the number of candidates increases. The third parameter in the initialization of the learning process involves a sample of the type of units used for candidates.

After the learning process is initialized, we assign it a training set created at step one, using the setTrainSet method. Finally, a number of keywords can be added to the recordList, which records statistics on the training of the network as the process evolves. For the purposes of this tutorial, we promted the recording of the output patterns, error patterns, and sum squared errors.

The next step is to train the network. During training, the process will typically output the following:

- command line arguments (available in JBuilder)
- stopping reasons
- Error bits
- Sum squared error
- phases
- epochs
- output weights
- candidate scores
- winning unit installed into the network (index and name)
- candidate weights

Let's take a look at these one by one. First, the command line arguments in Jbuilder are always present at the top of every output window. These commands are sent directly to the Java Virtual Machine in order to compile and run your code. If you chose not to use the Jbuilder environment to run your code, or if this environment is not available on the machine you are using, you can always resort to using the command line directly. To do so, simply bring up a command prompt (In Windows: Start -> Run -> cmd), and write there that exact line (providing your java files are in the same folders). From my personal experience, this can sometimes be useful when lending your code to collegues that are on Unix operating systems. For the rest of us, however, this is of rather little importance. The second information that will automatically be outputted to the your output window will consist in a stopping reason for the first output phase. Most commonly, this reason will show up something like "Stagnant", "Time out", or, ultimately, "Success". Third, the error bits will be outputted. Error bits indicate the number of output values that the network has misclassified. Remember that networks output continuous values, and bits are typically discrete values. In order to determine if the network has classified a network correctly, its output values must thus be thresholded. By default, the threshold is set to 0.4. If you wish to modify it to make it more or less rigourous, you may do so in the following way:

```
ccLearn.setTrainStatistic(new ErrorBits(DataNames.ERROR_PATTERNS, 0.4));
```

A value of 0.4 is used as the default, and this value assumes that the output sigmoids range from [0, 1]. Given this assumption, the mid-point activation of an output unit is thus 0.5. If the activation falls below this value, the network can be though to attempt to classify the pattern as a 0. On the other hand, if the activation is above the mid-point 0.5 value, the network can be though to attempt to classify the pattern as a 1. As a matter of precaution, if the value is too close to 0.5, we assume that the network is still uncertain as to whether the pattern belongs to either the 0 or 1 category. By taking a value of 0.4 for the threshold of the error bits, an output must have a value below 0.4 to fall into the 0 category, and a value above 0.6 to fall into the 1 category. By setting a smaller threshold, say, 0.2, you can force force a network towards more precise output activation values. Conversely, by setting a higher threshold, say, 0.5, you can relax the precision of classification. Training is deemed successful when the error bits falls to zero, indicating that the network can classify all instances of the problem correctly. Fourth, besides the

error bits, the training process of a CC network also outputs the sum squared error of the classification. The preceeding chapter on training multi layer networks explains in more elaborate details what the sum squared error statistic consists of. Fifth, the training process also outputs information about the phases taken on by CC networks and, sixth, the phase at which they stop. Typically, it is expected that both input and output phases come to a stop at around 100 epochs. If, in your simulation, the results are drastically different from this and don't reflect any deep changes in the base classes, you may want to revise some of the fundamental aspects of your setup. Seventh, information about the connection weights will be outputted. The weights of a CC network are divided into two main categories according to the layers they feed: input-side and output-side weights. Input-side weights feed the hidden nodes exclusively, while output-side weights feed the output weights exclusively. Note that input-side weights are outputted as "candidate weights". Eighth, the scores of the various candidates at the end of each input phase are outputted. Based on covariance calculations, the candidate with the highest score gets installed into the architecture of the network at the end of the input phase. This winning candidate is indicated by the output of the "Winning unit". The index as well as the name of this winning unit are provided.

Once the training process reaches an end, we can output some of the special statistics recorded using the `getTrainDataCollection` method. This method takes as argument a `DataName.String` describing a statistic to retrieve. You can retrieve a number of items in this way. For the sake of example, say we needed to output the hidden units' derivatives. First, it is essential that `DataNames.SUM_SQUARED_ERROR` was originally part of the `ccLearn.setTrainRecordList` String[] of arguments. Once training comes to an end, you can retrieve the saved statistic in the following way:

```
Double[] SSE = (Double[])
ccLearn.getTrainDataCollection(DataNames.SUM_SQUARED_ERROR);
```

If you are not familiar with the DataSet structure or casting procedures, please refer back to chapter 1.

4. Knowledge-based networks

This chapter covers:

- KBCC networks
- RBCC networks

In the previous chapters, we have explored the basics of Backpropagation and Cascade Correlation networks. One serious drawback of these algorithms is that they always start without any knowledge. This is represented by the fact that their initial weights are assigned random values. Engineers, as well as psychologists, are very favourable to the idea of having networks that start with some background knowledge when faced with a given problem. From the engineer's point of view, a network that can benefit from prior knowledge can potentially learn a problem faster and more accurately. From a psychologist's point of view, these networks are perhaps more apt at modelling human cognition, because it is well established from empirical studies that humans can transfer knowledge from one task to another with some appreciable benefits. Finally, by recording how networks transfer and combine previously acquired knowledge, we could gain some insight on the learning strategies used to solve a given task.

In this chapter, we review two techniques of knowledge transfer based on the Cascade Correlation algorithm. It is essential that the reader be familiar with this algorithm as well as the way in which it is implemented in the Java library. If this is not your case, please refer back to chapter 3. The first algorithm we cover is the Knowledge-based Cascade Correlation (KBCC), described in full in Shultz and Rivest (2001). Secondly, we will cover a new algorithm called Rule-based Cascade Correlation. A full formal description of this algorithm has yet to be published.

Knowledge-based Cascade Correlation

The reader will be happy to know that the way in which KBCC networks are ran is essentially the same as for CC networks. This is mainly due to the way in which the Java library was set up. As we have discussed in a previous chapter, both single units and full networks have for base class FunctionalUnit. This class specifies any type of function that takes a certain number of inputs and

returns a certain number of outputs. In the case of simple units (e.g., sigmoid, sin, tan, gaussian, etc.), the function takes a single input and returns a single output. In the case of full networks (e.g., CC networks, multi-layered networks, etc.), the function can take many inputs and return many outputs.

In the previous chapter on CC networks, we set up a pool of candidate units that the network could recruit into its topology in order to solve a given problem. In describing the basic version of CC, these units consisted in single units. However, pre-trained full networks can also be included in the candidate pool of CC networks in a very similar way. The only difference resides in the way this candidate pool is created. Let's take a simple example where a KBCC network is trained on a 2-bit parity (XOR) problem. In the candidate pool of this network, we will be include both single logistic units and pre-trained networks have solved the XOR problem. A full code version of this simulation is available from `KBCCTutorial.java`. As for standard CC networks, the first step is to create a cascaded network:

```
CascadeNetwork kbccNet = new CascadeNetwork(2, 1, new LogisticUnit());
```

The parameters of this constructor are the same as for CC networks, and can be found in the previous chapter. In this case, we create a network with 2 inputs, 1 output, and input-output connections consisting in logistic units.

The next step is where KBCC really begins. First, we must create a candidate pool. This candidate pool will specify the nature of the candidates, and how many we want to include. The Java object that will hold these definitions is of type `CandidateDescription`. In defining a candidate pool, a different object of type `CandidateDescription` will be created for each different type of candidate to be included in the pool. For instance, if a pool is to be composed of sigmoid units and pre-trained CC networks, a total of two objects will have to be created to accommodate each of these candidates. This can be easily accomplished in Java by creating an array of objects of type `CandidateDescription`:

```
CandidateDescription[] pool = new CandidateDescription[2];
```

Once we have created our objects, let's fill them with the necessary information. To do this, we must fill in the constructor of CandidateDescription for each of the objects we have created. Here are the parameters for a commonly used CandidateDescription constructor:

[String: name] [integer: number of copies to include in the pool] [FunctionalUnit: sample of the function] [boolean: will the first example be directly connected?]

We will now take these parameters one by one and see how they work. First, you must provide a name that defines the type of function you are including in the pool. For instance, if you are including sigmoid units, an appropriate name would be: "sigmoid". This name will be printed out when you run simulations to let you what was recruited at the end of a given input phase. The only important thing is to use a name that will be meaningful to you. The second argument to the constructor is an integer that specifies how many functions of a given type you want to include in the candidate pool. For instance, if you are including sigmoid units, you may put a value of 5 here, stating that you want a total of 5 sigmoid units to be included in the pool of candidates. Be aware that this creates multiple copies of a given FunctionalUnit. If you pre-trained some CC networks on a given task and want to include five *different* networks, you must pre-train them separately. Otherwise, if you put a value of 5 for this parameter, you will be placing 5 copies of the same network in your pool of candidates. When these networks are trained in input phase, the only thing that will vary from one to the other is the input-side weights connecting these networks to the layers under it. The values of these weights is generated randomly at the start of every input phase.

Fahlman recommended that the number of copies of each element to include in a pool should be around eight (8). Keep in mind that more copies will tend to slow down training.

The third argument of the constructor gives the system an example of the actual function to be used in the candidate pool. For instance, let's suppose you pre-trained a CC network named ccNet on a given task, and you now want to include it in the candidate pool of a KBCC network. The third argument will then be simply set to ccNet. Because the expected argument is of the basic type FunctionalUnit which encapsulates all functions, you can pass virtually any function here. The only

constraint is that the function be differentiable. By definition, most commonly neural networks embed differentiable functions. The only possible exception is certain cases of rule-based networks, as we will see shortly. For purposes of this discussion, you usually don't have to bother thinking about a network being differentiable or not. In fact, if it is composed of a set of differentiable simple functions (e.g. sigmoids) joined together by weights, it is necessarily differentiable. To check this, there is a function in FunctionalUnit which checks for you whether a function is differentiable or not provided this information was given when the function was created. This function is as follows:

```
someFunction.isDifferentiable();
```

`isDifferentiable` returns true is the function is differentiable, and false otherwise.

An important point to consider about passing fully pre-trained networks to a KBCC network is that the weights inside that subnetwork will never be retrained. This is due to the fact that when a sub-network is passed to the candidate pool of a KBCC network, it is down-casted to its basic `FunctionalUnit` type, and basically becomes a "black box" function that takes a number of inputs and returns a number of outputs.

In the past, some lab members have suggested that we enable KBCC networks to retrain the weights inside recruited hidden units. There are some pros and cons to performing this, which is a discussion we won't go into for the time being. This option has never been integrated into the Java library, and it is left as a possible path of exploration for future developments.

There is one last point I would like to make concerning the recruitment of pre-trained networks. One great advantage of the way the library is set up is that there is no need to worry about the number of inputs/outputs of a KBCC network versus the size of pre-trained candidate networks. Suppose, for instance, that you are training a KBCC network with 20 inputs and 5 outputs. It is possible that your candidate pool contain a pre-trained network composed of 5 inputs and a single output. KBCC can seamlessly integrate this network into its topology without the use having to worry about this

mismatch in size, and without having to code any extra specifications when setting up the simulation.

Finally, the last argument to `CandidateDescription` is a boolean value termed "first directly connected". When you recruit a pre-trained network, KBCC will basically evaluate it's usefulness in solving the task at hand. If a sub-network is particularly useful, it would be best to have the weights linking the inputs of KBCC to the input of this sub-network set to 1. This would insure that the inputs of KBCC arrive at the sub-network unaltered, in this way taking advantage of the fact that this given sub-network is particularly useful at solving the task. When you set the "first directly connected" option to true, KBCC treats the first copy of a given `FunctionalUnit` in the pool a little differently from the rest of its "brothers" by connecting its input weights in that described fashion. Practically speaking, this means that, if a given sub-network is very good at solving a given problem, it's first copy has a very high chance of being recruited. As usual, all other copies have their input-side weights initialized randomly. It is recommended that the "first directly connected" option is set to true for pre-trained networks, and false for simple units that don't contain any prior knowledge.

Let's now integrate all we have learned so far into creating a pool with 4 pre-trained CC networks and 4 sigmoid units. First, a pool must be created with 2 objects of type `CandidateDescription`:

```
CandidateDescription[] pool = new CandidateDescription[2];
```

Then, we must initialize each of the pool elements according to their desired content:

```
pool[0] = new CandidateDescription("sigmoids", 4, new LogisticUnit(), false);
pool[1] = new CandidateDescription("ccNets", 4, ccNet, true);
```

Once your pool is created, you can relax. The rest is admittedly quite simple. All you have to do is create the learning scheme for your KBCC network to train. The most common constructor for performing this receives the following arguments:

```
[CascadeNetwork mainNetwork][CandidateDescription[] pool]
```

The first argument expects your KBCC network, and your second argument receives your pool of candidates. An example may not be necessary here, but here's one anyway, using the Frobenius norm:

```
FrobeniusCascadeCorrelationLearning kbccLearn =
FrobeniusCascadeCorrelationLearning(kbccNet, pool);
```

Et voila! You are ready to train your KBCC network:

```
kbccLearn.train();
```

At this point, you can refer back to the section on training and analysing CC networks. There are some data of particular interest that you may want to analyse when running KBCC simulations. In particular, you may wish to pay close attention to the number and type of recruited candidates.

KBCC learners (e.g., `FrobeniusCascadeCorrelationLearning`) offer the more advanced user a number of options to adjust various aspects of the basic algorithm. One option of particular interest is the following:

```
kbccLearn.setBeginInInputPhase(boolean switch);
```

Typically, CC and KBCC networks always start and end in output phase. However, with this option set to true, it is now possible to force the network to begin in input phase. The results of this alteration are still under investigation.

Another option of interest in the learner consists in asking the network not to layer the new candidates on top of each other. In other words, it tells the network not to cascade new hidden recruits, but rather to put them on the same layer. Here's how it works:

```
kbccLearn.setDontCascade(boolean switch);
```

RBCC Networks

Rule-based networks are a derived from KBCC networks. The only fundamental difference is in the content of the candidate pool. In KBCC networks, this candidate pool is typically composed of pre-trained networks. In RBCC, however, the candidate pool is composed of rules that are shaped in the form of networks. In this section, we will review the process of creating rules and using them in simulations with RBCC.

To illustrate how to create a new rule, we will create a new function and call it `createRule()`. This function will take no argument and return a rule-based network of type `FunctionalUnit`. The rule itself is created with a series of `String`s specifying various horn clauses. This is better illustrated with an example. Imaging we wanted to create a 2-bit XOR rule called "rule".

```
String[] rule = new String[] { "rule :- not(1), not(2)",    //clause 1
                               "rule :- 1, 2" };            //clause 2
```

Clauses are indicated by the symbol ":-". The first clause states that the rule should return true if the inputs 1 and 2 are both not true. The second clause states that the rule should also return true if the inputs 1 and 2 are both true. When analyzing the various clauses, you can think of them as being "OR'd" together: clause 1 OR clause 2.

Once you have specified your various clauses, you must explicitely indicate to the system the order in which the rules will be presented, using a `String[]` once again:

```
String[] inputOrderWanted = new String{"1", "2"};
```

Remember that a network will be created with this rule. `inputOrderWanted` says that the first input to this network should be considered "1", and the second input "2". It sounds simple enough for now, but things may get more complicated with more intricate rules, so don't overlook this seemingly obvious step.

Besides specifying what to expect at the input, we must also specify the output. In the case of our example, the output is simply the result of the rule. Once again this is specified using a `String[]`:

```
String[] outputOrderWanted = new String{“rule”};
```

You should know that it is possible to combine various rules at the output, but we'll keep things simple for now.

Once we have created the rule, we now wish to have it embodied in a network. For this purpose, we will use a class called `KBANNFactory` that generates a network based on a rule. Here is the skeleton of the constructor we will use to this purpose:

```
[String[] rule] [double omega] [AbstractSimpleUnit sampleInputUnit] [AbstractSimpleUnit sampleHiddenUnit] [AbstractSimpleUnit sampleOutputUnit] [String[] inputOrderWanted] [String[] outputOrderWanted]
```

We will now consider each of these arguments in turn. The first argument expected by the `KBANNFactory` is the `String[]` that describes the rule in terms of Horn clauses. The second argument is a value termed omega (Ω). We will not review the details of this value. Suffice it to say that this represents the weight values in the network, and that a value of 0.4 is recommended (see Shavlik & Towell, 1993). The third, fourth, and fifth arguments receive a sample of input, hidden, and output units respectively. Remember that sub-networks recruited by KBCC must always be a differentiable function, and that this is the case if their activation functions are differentiable. In the case of creating networks from rules, we could create perfect rules by using step functions (see Figure 1). However, we can't do that because this function is not differentiable at all points along *x*. To remediate to this problem, we can rely on sigmoid units that have been somewhat “swashed” in order for them to come close to the step function (see Figure 2). In this way, of course, we don't get perfect rules, but we can approximate them conveniently and effectively. In order to generate a sigmoid unit with these properties, we must increase the beta parameter, which has a default value of 1.0

```
LogisticUnit sigmoid = new LogisticUnit();
sigmoid.setBeta(2.0);
```

Optimal values for beta are still to be determined by experimentation. Another important aspect of defining activation units is that rules must typically receive inputs of -1 for negative values, and +1 for positive values. This affects the way we define the offset and factor of the sigmoids in our network. Given input data that varies from -1 to +1, the offset must be of -1.0, and the factor of 2.0:

```
LogisticUnit sigmoid = new LogisticUnit(2.0, -1.0);
```

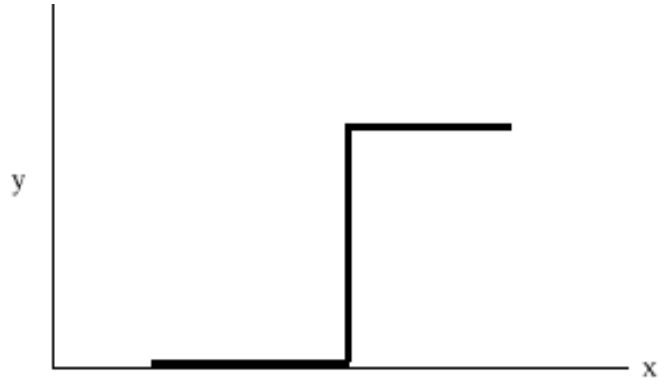


Figure 1. Step function that could be used as activation function to generate perfect rules.

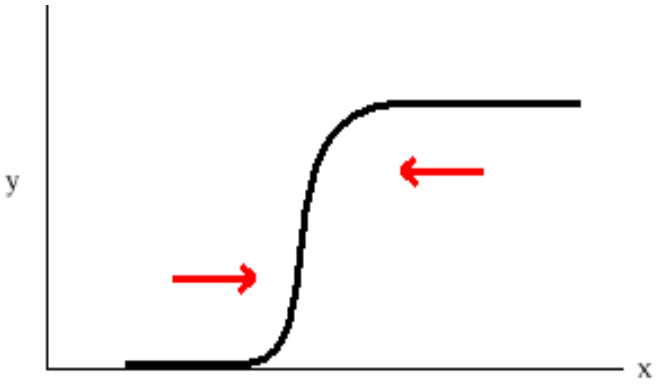


Figure 2. Squashed sigmoid function used to approximate a step function.

Input to rules always receive discrete values of -1 (negative) and +1 (positive). Do NOT use other ranges such as [0,1] to define input to rules, or the system will not work.

The sixth argument provided to KBANNFactory consists in the inputOrderWanted we specified, and the last argument expects the outputOrderWanted also specified above. Here's an example that creates a network based on the 2-bit XOR, where the rule has been previously defined as "rule".

```
KBANNFactory kbannFactory = new KBANNFactory(rule,
                                             4.0,
                                             new LinearUnit(2.0,-1.0), //input unit type
                                             new LogisticUnit(2.0,-1.0, 0.0, 2.0), // hidden unit type
                                             new LogisticUnit(2.0,-1.0), // output unit type
                                             inputOrderWanted,
                                             outputOrderWanted);
```

Once the KBANNFactory has been created, we can generate one or many networks in the following way:

```
FunctionalUnit ruleBasedNetwork = kbannFactory.createUnit();
```

This network can then be used directly in the candidate pool of a KBCC network.

10. Pruning

This chapter covers:

- Near-zero
- OBD

Pruning neural networks is a common technique whose purpose is to reduce the size of a network and ameliorate its generalization to new data. A number of technique to perform the pruning of connections and nodes are available from the litterature, and the KBCC library offers two of the most commonly used. The first, near-zero pruning, prunes weights whose absolute activation falls below a certain threshold. The second, Optimal Brain Damage (LeCun et al., 1990), calculates a saliency of the wieghts based on an estimation of the Hessian matrix. In the current chapter, we will review how each of these methods can be implemeted in a Cascade-correlation network. Thus far in the development of the library, pruning can only be used for cascaded networks. Future developments will make this feature available to other types of networks, including Backpropagation.

A) Near-zero pruning

In the KBCC library, the near-zero pruning is derived from the Analyzer class. This means that, if need be, pruning can be called at the end of each phase (input and output). If this is not necessary to your simulation, pruning can also be done at the end of the training phase. The following code instructs the network to prune its output weights after training, and calculate the resulting sum squared error:

```
//Train a KBCC network
kbccLearn.train();

//prune it
CCNearZeroPruning pruning = new CCNearZeroPruning(CCNearZeroPruning.OUTPUT, 2.0);
pruning.runAnalyzer(kbccNet);

//Calculate resulting sum squared error
DataSet testData = kbccNet.processDataSet(trainSet,
                        new String[] {DataNames.SUM_SQUARED_ERROR,
                        DataNames.OUTPUT_PATTERNS, DataNames.ERROR_PATTERNS});

SSE SSEss = new SSE(DataNames.SUM_SQUARED_ERROR);
DataSet SSEdata = SSEss.computeStatistic(testData,
                        new String[] {DataNames.SUM_SQUARED_ERROR});

Double sumSquared = (Double) SSEdata.getData(DataNames.SUM_SQUARED_ERROR);
```

Let's take a closer look at the CCNearZeroPruning class. When calling a new instance of this class, the constructor requires two arguments: the type of weights you want to prune and the threshold for pruning. The type of weights to prune is stored inside the class itself, under final integers that are called as:

CCNearZeroPruning.OUTPUT
CCNearZeroPruning.TOPMOST
CCNearZeroPruning.OTHERS
CCNearZeroPruning.COMPLETE

In this way, you have the option of pruning the output weights, the topmost layer of weights (topmost hidden layer of the network), all other weights (OTHERS), or everything (COMPLETE). The output weights refer to weights that connect to the output layer, whether they start from the input layer or from hidden layers. The TOPMOST refer to the weights connecting the latest recruited candidate (*topmost* candidate) to the output weights of the network. The OTHERS weights, refer to the weights connecting candidates together. The following figure illustrates this:

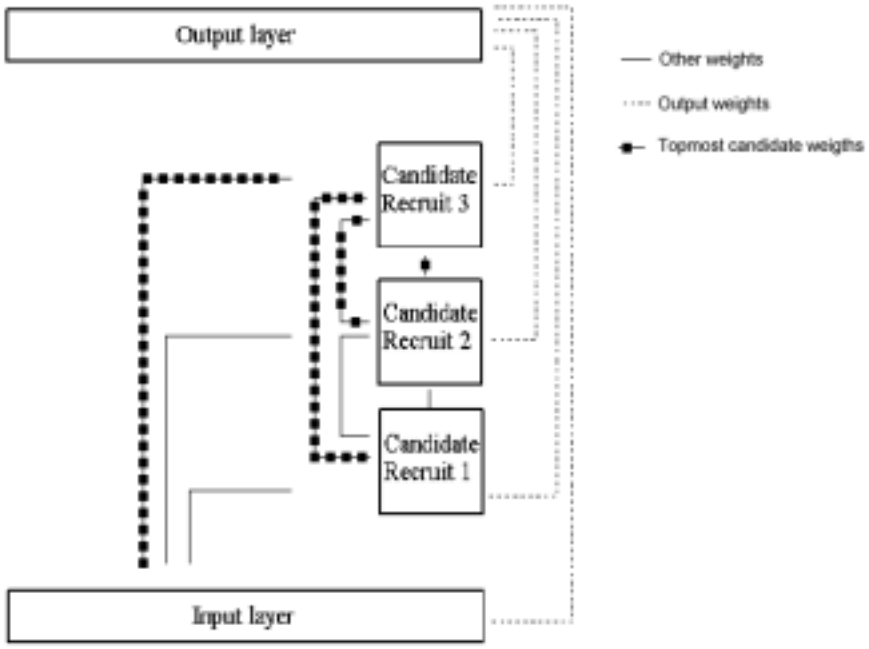


Figure 1. Pruning designations.

The second argument taken by the contructor of the CCNearZeroPruning class is a threshold value of type double. There are no hard and fast rules for setting this value. At the current point of research, the best strategy is still to run a number of networks, look at the weight values, and set a threshold that would seem to prune some of the low weight value while retaining the highest values. Once the constructor has been called, the next and final line simply calls the runAnalyzer() method, which takes a Cascade network as argument, and returns a data set containing either:

```
int PrunedOutputWeightCount
int PrunedTopMostWeightCount
int PrunedOthersWeightCount
int PrunedCompleteWeightCount
```

depending on the type of weights affected by pruning. Do not use the other version of runAnalyzer(). It has not yet been implemented in the library.

The runAnalyzer() method can only take as argument a network of type CascadeNetwork. Otherwise, an error will be raised. This is because pruning has not yet been implemented to other types of networks. Also note that FrobeniusCascadeCorrelationLearning must be set as the learning method of the network. For a discussion of the Frobenius norm, please refer to chapter 3. This entire warning also applies to Optimal Brain Damage.

The code snippet above illustrates how to prune a network after it has finished training. Because CCNearZeroPruning is derived from the class Analyzer, it is also possible to prune weights at the end of every phase. This is done by initializing the pruning *prior* to calling train() on the learning process.

The pruning example presented above illustrated a case when pruning was performed at the end of the learning period. Because CCNearZeroPruning is derived from the class Analyzer, it is also possible to prune a network during training. The user has the choice of pruning during a phase or at the end of it by adjusting the setAnalyzer() method in the learning process. Here is the arguments required by this method:

```
setAnalyzers(Analyzer newOutputPhaseAnalyzer, int newOutputPhaseAnalyzerInterval,
Analyzer newEndofOutputPhaseAnalyzer, Analyzer newEndofInputPhaseAnalyzer);
```

The first argument requires an Analyzer to be ran during the output phase. If you want to prune a network during the output phase, you would place your instance of the CCNearZeroPruning class in place of this argument. The second argument defines an interval at which to prune weights in output phase. A value of 2, for instance, means that the weights selected by the pruning scheme will be re-evaluated for pruning at every second epoch of the output phase. If you want to prune a network during the output phase and you have set an Analyzer as first parameter, you have to fill in this value with a relevant number. Otherwise, if you have not set an Analyzer as first parameter and you are not interested in pruning weights during the output phase, you can simply set this value to zero without any consequences. If you wish to prune weights at the end of the output phase, the third argument is where you would place your pruning Analyzer. In this way, weights selected by the Analyzer will be pruned only at the end of the output phase, when the network is ready to either end training or switch to input phase. Finally, if you wish to prune weights at the end of the input phase, you will place your pruning Analyzer as the fourth argument of the setAnalyzer() method. This will ensure that the weights selected by the Analyzer are pruned at the end of the input phase, just before the network switches to output phase. If you

fill in an Analyzer for all of the arguments of the setAnalyzer() method, the network will test for weight pruning during the output phase, at the end of the output phase, and at the end of the input phase. However, if you do not wish to prune weights at all these times, you can set one or more of the Analyzer arguments (the first, third, and fourth arguments) to null. To illustrate how the setAnalyzer() method works, here's an example of a KBCC network where the topmost weights are pruned at the end of the input phase:

```
CCNearZeroPruning pruning = new CCNearZeroPruning(CCNearZeroPruning.TOPMOST, 2.0);
kbccLearn.setAnalyzers(null, 0, null, pruning);
kbccLearn.train();
```

Pruning weights only at the end of the input phase is achieved by setting all the Analyzer arguments of the setAnalyzer() method to null, except the fourth argument, `newEndofInputPhaseAnalyzer`, which receives the pruning Analyzer.

To prune weights during training, you have to call the setAnalyzer() method *before* the train() method. However, to prune weights after training, you must call the runAnalyzer() method on the Analyzer instance *after* training.

It is important to understand that you should usually match the type of weights you select for pruning and the time in the training process when that pruning occurs. Output weights are typically pruned either during the output phase or at the end of the output phase, when the network is ready to either end training or switch to input phase. Pruning of the topmost weights, on the other hand, typically occurs at during the input phase or at the end of it. It is easy to understand, for example, that there is no point in pruning input weights during the output phase, because these weights are only modified during input phase. A proposed strategy for online pruning consists in pruning output weights at the end of the output phase, and topmost weights at the end of the input phase, because the topmost weights are the weights of the candidate unit that has just been installed in the network. Pruning of all weights can be reserved for a network that has finished training.

B) Optimal Brain Damage

Optimal Brain Damage (LeCun, 1993) calculates a saliency score on the weights of a network. This score is meant to classify weights according to their contribution to the reduction of the output error. Weigths with a high saliency help reducing the error considerably, while weights with a low saliency do not contribute significantly to reducing the error of the network. The class that performs the Optimal Brain Damage is CCOBD. This class is derived directly from CCNearZeroPruning, and is used in a similar way.

Pruning of weight OTHERS is not implemented yet in the library. Trying to use it will raise an Unsupported Operation exception. Also note that the warning in CCNearZeroPruning also applies to Optimal Brain Damage.

The CCOBD constructor requires the following parameters:

```
CCOBD(int newPruningMode, double newThreshold, DataSet newTrainSet)
```

The first two arguments are similar to the ones used for the CCNearZeroPruning: a pruning mode defining the type of weights to be prunes and a threshold of type double under which weight saliencies are pruned. The third parameter is the data set used to train the network. As mentionned in chapter 2, this data set must contain keywords for the inputs, targets, and pattern count. The following code prunes the topmost weights of a Cascade network at the end the training:

```
CCOBD = new CCOBD(CCNearZeroPruning.TOPMOST, 2.0, trainSet);
pruning.runAnalyzer(ccNet);
ccLearn.train();
```

Notice that the first argument of the constructor, which defines the type of weights to be pruned, is still derived from the CCNearZeroPruning. This is because CCOBD is derived from that class, and there was no need to override this argument. As with near-zero pruning, there are no precise rules available to set the pruning threshold.

As for near-zero pruning, Optimal Brain Damage can be applied online as well as after the training. The setAnalyzer() and runAnalyzer() methods of the learning process and pruning process respectively are used in the same way for Optimal Brain Damage as for near-zero pruning.

Having explored both techniques of pruning, the natural question to ask is: “which technique is better?”. Unfortunately, there are no definitive answer to this question, or even to the question of which weights are better candidates for pruning. A partial answer can be obtained from Waugh(1995) who compared various techniques of pruning in Cascade-correlation and found no significant difference in either in terms of final size of the network or generalization error. This research is not definitive, however, and further studies are underway.

10. Extra tools

This chapter covers:

- Linear algebra
- Network Serialization

Network Serialization

Training KBCC networks with large data sets (e.g., from repositories such as UCI and Proben) can be computationally very demanding, and a single simulation can take days to complete even on a powerful machine. One partial solution to this is to train some sub-networks on another machine, save them in a file, and have KBCC retrieve them at learning time. To perform this is quite straightforward. Here's some simple code to serialize networks out to file:

```
java.io.FileOutputStream outFile = new FileOutputStream("some_file_name.data");
java.io.ObjectOutputStream out = new ObjectOutputStream(outFile);
out.writeObject(some_network);
```

Don't forget to import the `java.io.*` package at the top of your class. Now here's some simple code to serialize networks from a file back into a program:

```
java.io.FileInputStream file = new FileInputStream("some_file_name.data");
java.io.ObjectInputStream in = new ObjectInputStream(file);
MultiLayerNetwork bpNet = (MultiLayerNetwork) in.readObject();
```

For certain types of applications, you may want to save or retrieve a number of networks. In this case, a good advice is to index your files with iterative values so that they can be saved and retrieved using a loop. Here's an example of retrieving a number of files using this method:

```
int totalNetworks = 10;
java.io.FileInputStream[] file = new FileInputStream[totalNetworks];
java.io.ObjectInputStream[] in = new ObjectInputStream[totalNetworks];
MultiLayerNetwork[] some_nets = new MultiLayerNetworks[totalNetworks];

for (int i = 0; i<10; i++) {
        file[i] = new FileInputStream("some_file_name" + i + ".data");
        in[i] = new ObjectInputStream(file[i]);
        someNetwork[i] = (MultiLayerNetwork) in[i].readObject();
}
```

Remember that serialization deals strictly in terms of basic Object types. Casting networks is thus a necessity here. Another important point is that when serializing a network out to file, a new file will be created with the name you have provided. This class will be written in machine language and cannot be modified in any way without compromising the retrieval of the network later on. When performing simulations, it is a good idea to systematically save networks and learners after training is completed. If you want to retrieve any new information later on and networks have not being saved, it will be impossible, and you will have to run the entire program all over.

LinearAlgebra

The class LinearAlgebra offers a number of methods for performing common operations on vectors and matrices. It is a very complete and useful class for doing anything from matrix rotations to multiplications, to adding vectors, to extracting columns. Another useful set of functions in the LinearAlgebra class is the toString() methods. These will take as argument a 1D, 2D, or 3D array and returned a tabbed string that can be used to output vectors and matrices as text.

12. Resources

Besides this tutorial, a number of extra resources on the KBCC library, as well as on Cascade-Correlation and Knowledge-Based Cascade-Correlation, are available through journal articles, books, and the World Wide Web.

There are a number of publicly available data set for experimenting with learning systems. One of the most widely used is certainly the UCI machine learning repository, accessible from:

http://www1.ics.uci.edu/~mlearn/MLRepository.html.

This repository contains a number of data sets for various real-life problems such as DNA splice-junction determination and myocardial infarction classification. For a PowerPoint tutorial of the learning process of a Cascade-correlation network, consult Thomas R. Shultz's page at:

http://www.psych.mcgill.ca/perpg/fac/shultz/Recent_Publications_files/cc_tutorial_files/frame.htm

This is a great introduction the main concepts of Cascade Correlation. A number of publications on Cascade-correlation and its KBCC extensions are available from:

http://www.psych.mcgill.ca/perpg/fac/shultz/Recent_Publications_files

The most recommended articles on CC and KBCC algorithms are, repectively, the following:

Shultz, T. R., & Rivest, F. (2001). Knowledge-based cascade-correlation: Using knowledge to speed learning. Connection Science, 13, 43-72.

Fahlman, S.E., Lebiere, C. (1989). The cascade-correlation learning architecture. Advances in Neural Information Processing Systems, 2, 525-532.

In general, some recommended journals to consult on neural networks include:

IEEE Neural Networks: www.ieee.org

Connection Science: http://www.tandf.co.uk/journals/tf/09540091.html

Neurocomputing: http://www.elsevier.com/locate/neucom

Neural Computation: http://mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=31

For a more or less complete listing of journals, consult the following:

http://www.emsl.pnl.gov:2080/proj/neuron/neural/journals.html