REIHE INFORMATIK
2/95

MONNET: a software system for modular neural networks based on object passing

R. Lange und R. Männer
Universität Mannheim
Container B6
D-68131 Mannheim

Abstract:
Modular neural networks integrate several neural networks and possibly standard processing methods. Tackling such models is a challenge, since various modules have to be combined, either sequentially or in parallel, and the simulations are time critical in many cases. For this, specific tools are prerequisite that are both flexible and efficient. We have developed the MONNET software system that supports the investigation of complex modular models. The design of MONNET is based on the object oriented paradigm, the environment is C++/UNIX. The basic concepts are dynamic modularity, object passing, scalability, reusability, and extensibility. MONNET features flexible and compact definition of complex simulations, and minimal overhead in order to run computationally demanding simulations efficiently.

Also appeared as:

R. Lange and R. Männer. MONNET: A software system for modular neural networks based on object passing. In S. Rogers, editor, *Applications of Artificial Neural Networks IV*, volume 1965, pages 232–243, Bellingham, WA, 1993. SPIE — The International Society for Optical Engineering.

Contact:

- Dipl. Phys. Rupert Lange
  Dept. Computer Science V
  University of Mannheim
  B6
  68131 Mannheim
  Germany

phone	+49-621-292-5707
fax	+49-621-292-5756
e-mail	lange@mp-sun1.informatik.uni-mannheim.de

- Prof. Dr. Reinhard Männer
  Dept. Computer Science V
  University of Mannheim
  B6
  68131 Mannheim
  Germany

phone	+49-621-292-5758
fax	+49-621-292-5756
e-mail	maenner@mp-sun1.informatik.uni-mannheim.de

1 Introduction

Neural networks (NN) supplement standard processing tools, particularly in the field of signal and image processing. The main advantage of NN with respect to their function is their learning and generalization capability and the robustness of their mapping against noise in the input data. From the implementation point of view the parallelism of NN is most important. Our group works in the field of parallel neural computers and on hybrid (opto-electronical) feed forward NN. For the latter, a hybrid single layer feed forward NN is being implemented. It is integrated transparently into the MONNET system so that it can be used and tested in a comfortable software environment. The optical realization exploits, additionally to the parallelism of NN, that NN can be operated with reduced accuracy of synaptic efficacies and neuron states. We investigate the processing capabilities of several NN that are constrained in this way in order to develop a NN model that is well suited for our optical implementation. Since many questions concerning NN cannot be answered by analytical treatment, simulations of NN are required. Simulations for research are mainly implemented in software to provide high flexibility, because there are many different types of NN. Each NN can have several parameters. Additionally, NN have to be combined with standard methods for pre- and postprocessing. Hence simulations can involve many different components. To manage such complex *experiments*, specific tools for neural engineering are prerequisite, that are both flexible and efficient. The latter is crucial because experiments either involving large NN or scanning high dimensional parameter spaces are computationally demanding.

There are several software systems for neural network simulations. Most of the systems, that partly integrate special hardware, emphasize the flexible construction of arbitrary NN and the visualization of the NN while running the simulation. Parameters of the NN like weights, neuron states, and thresholds can be studied interactively. Examples for this class of systems are the Aspirin/MIGRAINES [1] and the SNNS [2] simulator. They are used at our department to check NN on specific training sets. SNNS can be extended by writing new algorithms in C. Both Aspirin/MIGRAINES and SNNS lack support for experiments with advanced pre- and postprocessing or experiments scanning parameter spaces. Other simulators have a modular structure similar to MONNET, allowing several NN and standard methods to be combined. Examples are the Sesame [3, 4] and MUME system [5]. They are more fine grained than MONNET with typical modules being NN layers instead of entire NN. Extensibility is supported only within the framework of existing modules, with site oriented C language extensions similar to the systems described above, but there is no systematic way to add new modules or data types. Beyond the scope of NN simulators, we use the Khorus [6] image and signal processing system, which is very general and flexible. The logical structure combining modules is similar to MONNET. Khorus does not implement neural methods, but adding new modules is supported by several tools. Extension to new types of module I/O is not supported, though for the field addressed the framework provided seems to be general enough to enclose new data types. For each module a pro-

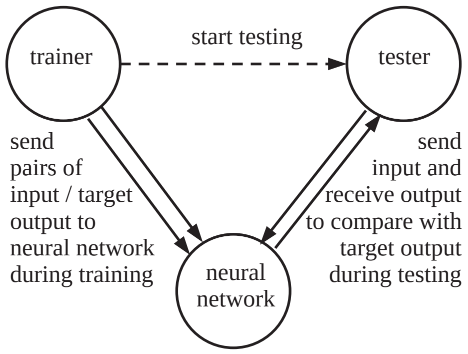
trainer
start testing
tester
send pairs of input / target output to neural network during training
send input and receive output to compare with target output during testing
neural network

Figure 1: MONNET modules

gram is started, exchanging data by means of shared memory, pipes, and sockets, and allowing Khorus to distribute modules over different machines. Khorus is a very flexible and comprehensive system (expressed in 90 MB binaries) and it provides a graphical programming language. However, by starting a program for each module, efficiency suffers for many modules.

None of the above simulators satisfies our demands regarding efficiency and/or functional scope. Main deficiencies are the lack of a consistent framework to extend the simulators and difficulties of running complex simulations combining several NN and pre- and/or postprocessing stages. We have devised the MONNET system to overcome these deficiencies. MONNET focuses on flexible definition of and efficiency running complex experiments. The next section makes clear how MONNET is designed to fit our demands. Then we pick the main components of the implementation to illustrate MONNET to be a both flexible and efficient system. A case study follows to give an idea how experiments are defined.

2 Design

The concept of MONNET is that of separate processing modules exchanging data, see Figure 1 (page 2). The modules process the input they receive from other modules; they itself produce new output after they have received sufficient input. Modules have different channels to handle logically distinct I/O. The type of I/O is not restricted, and arbitrary channels can be linked, provided that the interfaces fit. An experiment is assembled from modules, and running the experiment means passing I/O between modules repeatedly. This process is priority and event driven. Some concepts behind MONNET are similar to that of data flow computers [7, 8]. The problem that data flow systems cannot be realized efficiently on a control flow architecture arises only for fine grained (instruction level) data flow systems; it does not apply to the very coarse grained MONNET system.

2.1 Logical structure

The main processing modules of MONNET are *nodes*. Nodes send and receive *objects* through their *channels*, and the channels are connected by *links*. Links may provide additional simple processing, see below. That is, nodes pass objects to other nodes via links. Both new nodes and new links can be added in order to extend the functional scope of MONNET, as well as new objects to allow new types of data to be transferred.

Objects are the data items passed in MONNET. The object framework is generic to hold data types of any complexity, examples are scalars or vectors (input to a NN) and lists (entire training sets). Objects are organized in a reasonable manner to allow abstract data types so that logically similar objects can be processed identically. In order to check whether the output of a source is compatible to the expected input at the destination, objects provide their type, e.g. `FloatVector 4` for a vector of floating point values with 4 elements. Objects provide binary I/O to save and load interim results and standardized text I/O to exchange data with external applications. There is a type of I/O that saves the above object type information together with the object data in order to handle unknown objects on reading. For the same purpose there is a systematic way to create new objects from the type information read in. Finally, to allow systematic expansion of read-only data for writing purposes, object duplication is provided, see 2.2 below.

Processing in MONNET is mainly accomplished by nodes. Any function can be realized in one node. A node can e.g. represent one neuron or an entire NN or any standard algorithm, as a Fourier transform. Nodes communicate by passing objects, where several outputs might be produced by one input, or vice versa. Nodes can have any number of channels to send and receive objects, and the type of I/O of a node is not restricted because the object framework is generic. Nodes can be operated in different *modes* in order to be generic processing devices, e.g. the number of neurons or the learning rate define the mode of a NN node. The mode of a node does not change while running an experiment, but the *state* of a node clearly can, e.g. the synaptic efficacies of a NN during training. So for experiment definition one must pass parameters to each node, to define its mode and possibly an initial state. The number of channels and the type of their I/O objects are defined by the mode of the node. Therefore, as the mode, they are fixed during experiment execution. As with objects, there is a framework that nodes have to comply with. First, to save or load, create, and identify nodes they comply with the object framework. Hence nodes are a subclass of objects. Then, for checking the experiment setup, information about the channels of a node must be provided. First, there must be means to test whether a channel is for input or output, or both. Second, the type of the objects passed through the channel must be provided. For running the experiment, there are means to check whether a node has new output, to send the output, and to receive input.

Nodes can be combined by connecting their channels by links. The output of a node is sent to the target nodes by these links. Any two channels of arbitrary nodes can be connected, though consistency of the data sent and that expected by the target node must be checked. The link used mostly just connects channels and does not modify the objects transferred; but other links can be added that have transfer functions to provide simple processing. Hence the output of a link is not necessarily identical to its input. The 'simple' above means that links yield exactly one output for each input and do not have an internal state. However, a link may be operated in different modes. If a link provides processing capabilities, it typically provides non-reversible processing, e.g. disturbance of input by noise or digitizing a function in a vector. The modes in these examples define the form and degree of disturbance in the first or the range to be scanned and the number of elements in the resulting vectors in the latter example. Additional processing by links was introduced to reduce overhead by avoiding very simple nodes. As nodes, links are a subclass of objects and — like nodes for their channels — links must provide the type of their input and output and means to transfer objects. Transfer means to receive and process and send the object passed.

Control of such an MONNET experiment is based on priorities and events. Nodes are autonomous devices in that they control themselves. They depend only on their internal state and the I/O they receive and send. Explicit experiment control can be accomplished by interchanging objects. An example is a node that tests the performance of a NN mapping on a testing set that has been sent to the node, see Figure 1 (page 2) and the case study below. It starts testing the connected mapping after receiving a signal. As described above, nodes signal if they have new output. These nodes are called *active*. Generally, multiple nodes are active at the same time, and because objects are passed and processed sequentially, priorities are required for the nodes. Such priorities provide implicit experiment control. This is useful to avoid control signals. An example is passing objects through a pipeline of nodes, where in the simplest case the nodes send one object for each object received. Such a pipeline can be controlled implicitly by increasing priorities in the direction the objects are passed. To run an experiment, the following is performed. In experiment initialization, the nodes and links are created and specified (mode and initial state) and the initial activities of the nodes are polled and recorded. Then proper execution starts, during which the following is repeated until there is no active node left: the active node with highest priority gets control to send its output(s), the objects are passed via the links to the target nodes which get control to handle the input and possibly get active themselves.

Extensibility is a cardinal concept of MONNET. New objects, nodes, and links can be added to MONNET provided that the new component complies with the corresponding framework. These frameworks are terse and do not restrict the functional scope of MONNET in principle, see the implementation section below. Any useful component can be implemented. However, to keep the system compact and reduce development efforts, external tools are used together with MONNET if feasible. An example is displaying and printing plots of the results produced. For that, standardized I/O as described above is important.

2.2 Efficiency

MONNET focuses on efficiency to be suited for computationally demanding simulations: either large simulations due to large NN or numerous simulations due to exhaustive search in the parameter space of the experiment. The efficiency of MONNET is based on keeping the cost of object passing small by means of object references, and second by reducing the number of transfers by exploiting scalability, as explained now.

Execution of a network of nodes and links requires many data transfers. The data transferred can be very complex, in which case copying objects is a huge overhead if the target node or link guarantees to access the object read-only. Because of that, objects are transferred by reference rather than by value in order to minimize the cost of the transfers. Since objects support duplication, transfers by value are possible if required.

To minimize the overhead of the MONNET system the number of nodes and links and the number of object transfers have to be kept as small as possible. This can be achieved with scalable objects, nodes, and links. Examples are, that a single vector is transferred instead of many scalars, and that a NN can be represented by a single node with only one channel for all input values, regardless of the number of neurons in the NN. This leads to experiments with few nodes and links. There are typically less than 100, even for rather complex simulations. Scaling up an experiment, e.g. doubling the length of signals processed, hence does not affect the number of transfers. Since the cost for one transfer is constant, this is true for the absolute overhead, too. This means that the relative overhead vanishes while scaling up, since then the computation time in the nodes and links generally increases rapidly. The efficiency of MONNET draws strongly on this scalability feature.

In addition to the above the ability to save and load any object transferred is provided. First, this is important to keep interim results. Second, the internal state of nodes can change during execution of the experiment, and reaching some state may require considerable computation, e.g. a NN training. Hence, although reproducible, it is useful to provide means to save and load the internal states of nodes. In MONNET, both entire nodes as well as parts of nodes can be accessed for read and write, and since they are objects they can be transferred and saved or loaded as normal objects can.

2.3 User interface

Since the definition of experiments is illustrated in the case study below we sketch the user interface only briefly in this section. An experiment is assembled of nodes and links. The user can pick any number of the provided nodes and can connect arbitrary channels by the links provided. The experiment setup is defined as text in a natural and terse manner and typically fits onto 1 or 2 pages. Each link and node in the experiment requires to write one line. This text experiment description is processed by the MONNET interpreter. Faults in the specification of nodes or links (mode and initial state / mode) and faults connecting channels to links and links to channels are reported. Errors are, if for a channel-to-link connection the channel is a pure input channel, or for a link-to-channel connection the channel is a pure output channel, or if the passed objects are not compatible. The cost of interpreting is very small, so trying another experiment setup by changing the specification of nodes or links or replacing nodes or links can be done in seconds, e.g. changing the number of hidden neurons in a back propagation NN or replacing the entire NN by a Perceptron NN.

To ensure save parameter changes macros are provided. For example, a signal may be processed in several stages. Then the length of the signal is a relevant parameter for several nodes and links and can be defined as a macro (symbol) to be altered at one place. Additionally, one can reuse previous experiments as subexperiments, without inserting all the text. All this is accomplished by a macro processor that is passed first during the interpreting process. It provides more capabilities, but essentially defining symbols and including files is required. The usefulness of macro processing is illustrated in the case study below.

3 Implementation

In this section, the implementation of the MONNET system is explained. We cannot go into the details, but the concepts shown should suffice to give an idea that MONNET meets the requirements worked out in the previous section and how this is accomplished.

3.1 Object oriented environment

The environment employed is C++/Unix. The choice of an object-oriented implementation is obvious when considering the design guidelines. The frameworks described above for objects, nodes, and links simply correspond to abstract base classes `Object`, `Node`, and `Link` in C++, with `Node` and `Link` being a subclass of `Object`. Proper objects, nodes, and links are then implemented as classes derived from the corresponding base classes. The required methods (e.g. file I/O for objects, object I/O for nodes and links) are defined as virtual functions in the `Object`, `Node`, and `Link` classes. MONNET draws strongly on the concept of virtual functions. This polymorphism together with the concept of inheritance simplifies the development of new components either. Although using an OOP language for this project has not been a must it proved to be very useful in respect to yield both a flexible and efficient system.

Two basic components had to be implemented due to the choice of C++ as programming language. As described, object references are passed instead of values. In C++ there is no garbage collection, so that the programmer has to throw away pointer-referenced data that is allocated dynamically. To leave that to the compiler built-in pointers to objects are transparently replaced by a custom `ObjectPtr` class that provides reference counting and frees data automatically if no reference is left. The other basic component is to store the actual C++ class hierarchy. For that, the name of each class and that of its parent(s) is registered. For each class “A”, this allows to determine during runtime, whether “A” is derived from, a base class of, or neither of both of any other class. This is prerequisite to check the assignment compatibility of objects during experiment initialization. Second, a pointer to a function that returns a pointer to a new object of class “A” is registered. This provides a systematic way to create new objects of class “A” by name, especially from type information read in, either during experiment initialization or file I/O. Having object creation (by name) centralized is also useful to hide dynamic loading of code into the program, due to new objects, nodes, and links.

3.2 Kernel

We will only summarize the implementation of the kernel. Several stages are passed during the initialization in order to interpret the experiment description and setup internal data structures required prior to running the experiment. As mentioned, the very first step is macro processing, provided by a standard Unix preprocessor, namely **`m4`**. This is followed by a custom preprocessor **`scan`**, that converts the user experiment definition (see the case study below) into the kernel experiment description format. **`m4`** and **`scan`** are called prior to the proper MONNET kernel **`network`** from the **`monnet`** script. That way, the user interface is separated from the kernel.

The kernel consists of several stages. First, **`read`** reads in the descriptions of the nodes and links of the experiment; this full text descriptions a stored for diagnostic purposes. Then, during **`build`**, the nodes and links defined in the experiment are created dynamically from the heap, and specified as described under 2.3. The so-called external check **`xcheck`** then checks that the connection of node channels by links is correct, followed by **`setup`** to setup internal data structures representing the topology of the nodes and links. Then, during the internal check stage **`icheck`**, control is passed to each node by its **`Check`** function to let it check whether it is embedded correctly, meaning that no mandatory channels are left unconnected. The last stage during initialization is to poll the initial activities, i.e. recording which nodes request to send output. Then, the proper execution **`run`** of the experiment starts, with repeatedly selecting the active node of highest priority, passing its output(s) to the target nodes via the links, and giving the target nodes control to handle their input(s). The execution stops when there is no active node left.

Experiment initialization, being structured in several stages, seems to be rather complicated, but it is not, neither concerning the amount of code required nor with respect to interpreting time. The latter depends on the number of nodes and links in the experiment, but interpreting time is less than a second for the experiments run so far. The execution time of typical experiments is predominated by the contributions of the nodes and links, so that the overhead due to the MONNET system is negligible.

3.3 Objects

The object framework is realized as an abstract C++ class `Object`. The non-virtual functions must not be supplied by the developer of new objects, they rely on the class registration (that provides class names and creating objects by name) and the virtual functions shown. The friend function `RegisterObject` performs the class registration described above and sets the class id `_Object` to a unique value; `_Object` is returned by `Object::ClassID`. The name of a class is provided by the (non-virtual) `Class` function, which picks the registered name using (virtual) `ClassID`. The return value `DString` is a custom class for strings and equivalent to `char*`. Member function `Specification` returns the specification of an object as a string. Together with `Class`, this gives the type information mentioned earlier, e.g. `FloatVector 4`. `NewSpecifcation` changes the specification. Object duplication is provided by `Copy`, so that `anyobject.Copy()` points to a new object identical to `anyobject`. For raw object I/O, i.e. without type information, there are `Write` / `Read` for binary and `Print` / `Scan` for text I/O. The latter pair is called for the standard C++ I/O operators << and >> resp. The raw I/O functions `Write` / `Read` / `Print` / `Scan` are virtual to be overloaded in derived classes, whereas the following functions `Save` / `Open` / `Export` / `Import` for typed I/O are not. They rely on the corresponding raw function, `Class` / `NewObjectByName` (the latter provided via class registration), and `Specification` / `NewSpecification` for output and input of objects resp. As an example, `Export` first writes the class name `Class` and the specification string `Specification` on the leading type information line and then calls `Print` to write out the proper data. Besides utility functions, the interface of the C++ class `Object` then is as follows; except for function `ClassID`, zero return values indicate that the operation failed.

```
class Object
{
friend void RegisterObject();              // class registration
public:
virtual int ClassID();                     // identifying class of objects
DString Class();                           // identifying class of object by name
virtual DString Specification();           // identifying specification of objects
virtual int NewSpecification(DString& nS); // changing specification of
objects
virtual Object* Copy();                    // copy object (duplication)
// I/O without class and specification information.
virtual int Write(ostream& os);           // binary
virtual int Read(istream& is);            // binary
virtual int Print(ostream& os);           // called by standard output operator <<
virtual int Scan(istream& is);            // called by standard input operator >>
// I/O with class and specification information,
int Save(ostream& os);                    // binary
int Open(istream& is);                    // binary
int Export(ostream& os);                  // text
int Import(istream& is);                  // text
};
```

3.4 Nodes

The node framework is also realized as an abstract C++ class `Node`, where `Node` is a subclass of `Object`. Hence the `Object` class defines the framework for node type information, node duplication, and node I/O, which is not repeated here, see above instead. The new member functions group into 3 categories. First, there are functions called during experiment initialization, where the components are created and the syntax of the experiment is checked. `InChannel` / `OutChannel` / `InOutChannel` provide the type of the channel, whether it receives input, sends output, or both. `ChannelClass` / `ChannelSpecification` provide type information about the objects passed through the channel. `Check` has to confirm that the node is connected to its environment correctly. Second, there are functions called while running the experiment. `NewOut` tells whether the node requests to send new output, with the `NewOut` with argument checking on one specific channel. Input to the node from elsewhere is passed to the node via `In`, and it sends its output via `Out`. Third, a systematic way to access members of the node (by name) is provided by `MemberClass` / `MemberSpecification` / `GetMember` / `SetMember`, with obvious meanings. Node `AccessNode` employed in the case study below relies on these. Besides utility functions, the interface of the C++ class `Node` is as follows; `NodeID`, `ChannelID`, and `LinkID` actually are integer ids (`typedef int`) to identify nodes, channels, and links.

```
class Node: public Object
{
friend void RegisterNode();                // class registration
// from Object
public:
virtual int ClassID();                     // identifying class
// new, for experiment initialization
virtual int InChannel(ChannelID c);        // does channel accept input?
virtual int OutChannel(ChannelID c);       // does channel provide output?
virtual int InOutChannel(ChannelID c);     // does channel both accept input
and provide output?
virtual int ChannelClass(ChannelID c);     // class of transferred Objects
virtual DString ChannelSpecification(ChannelID c); // specification
virtual int Check(NodeID id);              // check that Node is embedded correctly
// for running the experiment
virtual int NewOut();                      // is there any new output?
virtual int NewOut(ChannelID c);           // is output of channel c new?
virtual void In(ChannelID c, ObjectPtr input); // input to Node: receive Object
virtual ObjectPtr Out(ChannelID c);        // output from Node: send Object
// for accessing the node or its members
virtual int MemberClass(DString &member);
virtual DString MemberSpecification(DString &member);
virtual ObjectPtr GetMember(DString &member); // read member
virtual void SetMember(DString &member, ObjectPtr value); // write member
};
```

3.5 Links

As for nodes, the link framework is realized as an abstract subclass `Link` of `Object`. New member functions fall in the same categories, experiment initialization and running the experiment. The third category, member access, is not necessary for links since they do not have an internal state. So for experiment initialization, we have analogous to `ChannelClass` / `ChannelSpecification` for class `Node` the pairs `FromClass` / `FromSpecification` for input and `ToClass` / `ToSpecification` for output of the link. During experiment execution, only `Transfer` is called; that function receives, processes, and sends the object passed. The `Link` class reads as follows.

```
class Link: public Object
{
friend void RegisterLink();                // class registration
// from Object
public:
virtual int ClassID();                     // identifying class
// new, for experiment initialization
virtual int FromClass();                   // class of Link input Objects
virtual DString FromSpecification();       // specification of Link input Objects
virtual int ToClass();                     // class of Link output Objects
virtual DString ToSpecification();         // specification of Link output Objects
// for running the experiment
virtual ObjectPtr Transfer(ObjectPtr from); // transfer Object
};
```

4 Case study

This section shows first a simple experiment to train and test a NN on a given data set. Two extensions of the experiment are then made. Together, this should give an idea how much effort it takes to define an experiment and to reuse and extend it. For the text experiment description, `#` starts a comment (to the end of the line) and specification strings (mode for nodes and links and possibly initial state for nodes) are bracketed with {}.

4.1 Train and test NN

The basic experiment is a standard setup, see Figure 2 (page 11). A NN "mapping" is trained on a given training set. The length of the training is defined in epoches, i.e. the number of cycles through the training set. Then, to check the success of the NN training, the mapping of the NN on the training set is tested. In detail, the following happens. The input part of the training set is stored in the file "inputPart" in text format, with one pattern vector (with 8 elements here) on each line. This file is read in by the node named "inputPart" of type `FileIO`. The vectors read in are sent to the `Trainer` node "trainer" and to the `Tester` node "tester". Analogously, the output part of the training set is read in and sent to "trainer" and "tester", but through different channels. The training is then started by reading in a control vector by the `FileIO` node "control". This vector defines (in epoches) the control points of the training, i.e. after how many epoches the training has to be interrupted for testing purposes. The last control point gives the total length of the training. The control vector is sent to "trainer", "tester", and the `Perceptron` node "mapping". Node

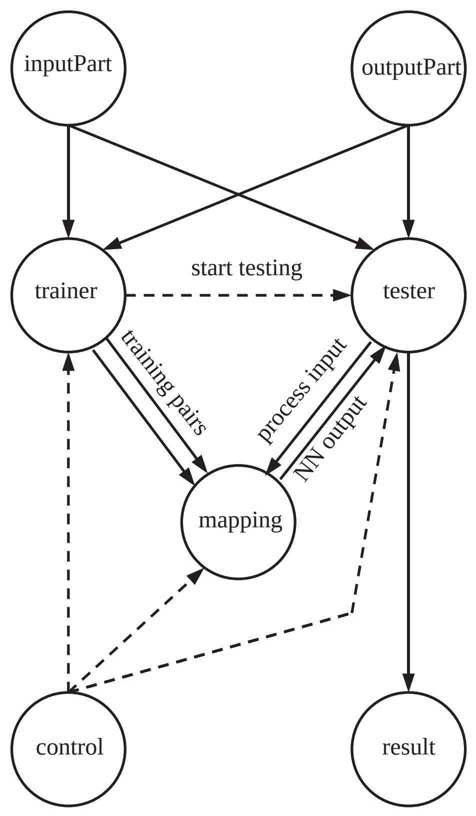

Figure 2: base experiment

"trainer" obviously requires the control vector. Node "tester" requires the control vector to set up internal data structures, since the result vector has the same length. The control vector is sent to "mapping" on its reset-channel, meaning that the synapses and thresholds are set to zero in the `AdaTron` case, what is necessary if you read in multiple control vectors for several training runs (useful for `BackPropagation` with random initialization of the weight matrix). After "trainer" has received the control vector, it sends the entire training set (2 lists) to "mapping" (which is necessary in this particular case since the AdaTron NN requires global information on the training set). "trainer" then starts training the NN by sending the input / target output pairs to "mapping" repeatedly. "mapping" itself adjusts its weights and thresholds according to the AdaTron [9] learning rule. When the trainer has reached a control point, it sends a signal (object `Pulse`) to "tester" which then computes the mean square error of the NN mapping on the training set (the mean absolute error is available on another channel of `Tester`). For that, the input parts are sent to "mapping" and the process results sent back by "mapping" are compared with the target outputs. Training is then continued until the next control point has been reached. After the last control point, “tester” sends the result vector to `FileIO` node “result” to be written into file “result”. If there is another control vector left, the training is repeated for the new control vector, giving an corresponding additional result vector.

The text description (file “base_experiment”) of this experiment follows; the notation for a node is

```
nodeType nodeName -t priority -s {specification string: mode and initial state of node}
```

and for a link

```
linkType fromNodeName.channel toNodeName.channel -s {specification string: mode of link.}
```

The character # introduces a comment (to the end of the line).

```
# monnet experiment : train and test neural network on given data set

# parameters
define(N1,8)                                            # mapping input dimension
define(N2,3)                                            # mapping output dimension

# training set is read in from files inputPart and outputPart
FileIO inputPart -t 20 -s {open inputPart -type scpr -class FloatVector -spec N1}
FileIO outputPart -t 20 -s {open outputPart -type scpr -class FloatVector -spec N2}

# control vector gives length of training in epochs and control points
FileIO control -t 10 -s {open control -class FloatVector}

# trainer
Trainer trainer -t 20
DirectLink inputPart.0 trainer.0 -s FloatVector         # training set input part
DirectLink outputPart.0 trainer.1 -s FloatVector        # training set output part
DirectLink control.0 trainer.2 -s FloatVector           # control vector

# trainable mapping
AdaTron mapping -t 40 -s {N1 N2}
DirectLink trainer.0 mapping.2 -s {Object FloatVector} # actual pair input part
DirectLink trainer.1 mapping.3 -s {Object FloatVector} # actual pair output part
DirectLink control.0 mapping.4                          # reset mapping
DirectLink trainer.5 mapping.5 -s ObjectPtrList         # training set input part
DirectLink trainer.6 mapping.6 -s ObjectPtrList         # training set output part

# tester for training set
Tester tester -t 40
DirectLink inputPart.0 tester.0 -s FloatVector          # inputs
DirectLink outputPart.0 tester.1 -s FloatVector         # target outputs
DirectLink control.0 tester.4 -s FloatVector            # control vector
DirectLink trainer.3 tester.5                           # control point reached
DirectLink tester.2 mapping.0 -s FloatVector            # input sent to mapping
DirectLink mapping.1 tester.3 -s FloatVector            # mapping output

# print results
FileIO result -s {open result -mode write -type scpr}
DirectLink tester.6 result.0                            # error on training set
```

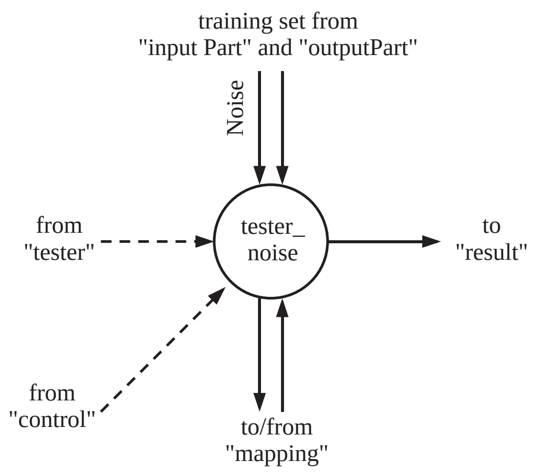

Figure 3: extension 1

In this example, if you perferred the original perceptron you would just write Perceptron instead of AdaTron on the line where you define the "mapping" node and everything would work the same way, i.e.

```
Perceptron mapping -t 40 -s {N1 N2}.
```

4.2 Extension 1: input disturbance

We now extend the experiment to additionally check the robustness of the neural mapping against noise in the input neurons, see Figure 3 (page 13). First, the initial experiment is included. Then, an extra `Tester` node "tester_noise" is required, which gets disturbed instances of the input parts together with the original target outputs of the training set. The disturbance of the input parts is accomplished by the `Noise` link. It applies additive white noise with a range [-0.5,0.5] to the vectors read in by "inputPart" and transfers them to the new "tester_noise". Experiment execution is mainly as above, but now, the performance of the NN "mapping" on the disturbed input / target output pairs is computed additionally. For that, at each control point, the original `Tester` node "tester", which is started by "trainer", starts itself the testing on the new testing set by sending a signal to "tester_noise" after it has finished testing on its own testing set. The mean square error of the NN on the additional testing set is written out into the same file as above, giving a second line for each training. That is, the extension to a new testing set simply requires one new `Tester` node, one `Noise` link to disturb the input parts, and some `DirectLink` links to connect the `Tester` correctly.

The experiment description reads as follows:

```
# extend base experiment to test on an additional testing set
```

```
include(base_experiment)                          # inserts base experiment

# parameter RANGE defines the range of the additive white noise applied
define(RANGE,0.5)                                 # mean square error of 0.2

# tester for testing set: input part of training set is disturbed by white noise
Tester tester_noise -t 40
Noise inputPart.0 tester_noise.0 -s {-noe N1 -op add -wnmean 0 -wnrange RANGE}
DirectLink outputPart.0 tester_noise.1 -s FloatVector # target outputs
DirectLink control.0 tester_noise.4 -s FloatVector    # control vector
DirectLink tester.5 tester_noise.5                # control point reached
DirectLink tester_noise.2 mapping.0 -s FloatVector    # input sent to mapping
DirectLink mapping.1 tester_noise.3 -s FloatVector    # mapping output

# print results
DirectLink tester_noise.6 result.0                # error on testing set
```

To check for different values of the range of the white noise simultaneously (or some other sort of noise, e.g. flipping in the case of binary inputs) one can add serveral Tester nodes this way.

4.3 Extension 2: weight disturbance

The next extension is to additionally check the robustness of the neural mapping against reduction of the accuracy of the synaptic efficacies, see Figure 4 (page 15). First, this requires to access the weight matrix at each control point. Second, the accuracy of the synaptic weights and the thresholds is reduced by snapping (rounding) these values onto (few) predefined values. Third, a dummy NN has to be initialized with the constrained synapses / thresholds. Finally, the mapping of the constrained NN on the training set has to be tested. This extension is implemented with three additional nodes. First, the new `AccessNode` node "pick" is required to access the synaptic matrix and the thresholds of the original NN node "mapping". At each control point, "pick" picks the weight matrix and the threshold vector and sends them to the new `TestNN` node "constrained". `TestNN` is a dummy NN that receives its weight and threshold values on two channels and snaps these values onto a predefined set of target values. The relative magnitude of these target values is given by the mode of `TestNN`. The absolute values are derived from this parameter "OMEGA" by scaling "OMEGA" so that its maximum matches the actual maximum of the weights and thresholds resp. To test the performance of the "constrained" `TestNN` an additional `Tester` node "tester_constrained" is used, which is connected analogously to the `Tester` node "tester_noise" in the previous extension. The testing of "tester_constrained" is started by "tester_noise" after that has finished its testing. The mean square error of the constrained NN "constrained" on the training set is written out on the same file as above, giving a third line for each training.

The experiment description reads as follows:

```
# extend base experiment to test on an additional neural network
# with reduced accuracy of its weights
```

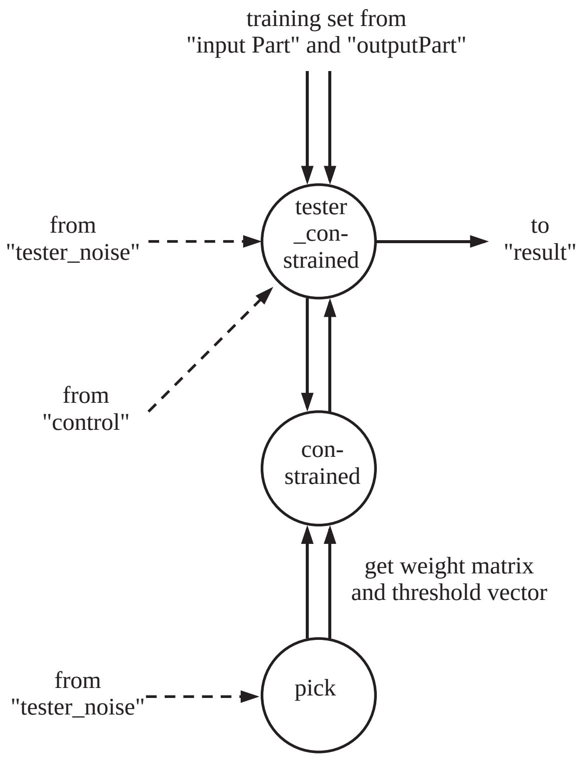

Figure 4: extension 2

```
include(extension_1) # inserts base experiment

# parameter OMEGA defines the relative digitization of the weights
define(`OMEGA',`{1 2 4}')

# additional constrained mapping with constrained weights and thresholds
TestNN constrained -t 80 -s {N1 N2 -omega OMEGA}

# pick the weights/thresholds at each control point, send them to the constr. mapping
AccessNode pick -t 50 -s {mapping weight threshold}          # weight and threshold
DirectLink tester.5 pick.0                                   # at each control point
DirectLink pick.1 constrained.2 -s {FloatMatrix -expand}     # send them to the
DirectLink pick.2 constrained.3 -s {FloatVector -expand}     # constrained mapping

# test constrained mapping on training set
Tester tester_constrained -t 40
DirectLink inputPart.0 tester_constrained.0 -s FloatVector   # test on training set
DirectLink outputPart.0 tester_constrained.1 -s FloatVector  # target outputs
DirectLink control.0 tester_constrained.4 -s FloatVector     # control vector
DirectLink tester.5 tester_constrained.5                     # start test after rnn
DirectLink tester_constrained.2 constrained.0 -s FloatVector # c. mapping input
DirectLink constrained.1 tester_constrained.3 -s FloatVector # c. mapping output
```

```
# print result of constrained mapping on training set
DirectLink tester_constrained.6 result.0                 # mean square error
```

In this example, the weight matrix and the threshold vector are duplicated prior to passing them to the `TestNN` node "constrained". This means, that the values of the weights and thresholds of the original NN "mapping" are not modified, so that the training process is not influenced. Object duplication is expressed in the "-expand" flag in the specification (mode) of the corresponding DirectLink links

```
DirectLink pick.1 constrained.2 -s {FloatMatrix -expand}
DirectLink pick.2 constrained.3 -s {FloatVector -expand}
```

If one omits this duplication flags, another interesting experiment is performed, where the accuracy of the weight matrix and the thresholds is repeatedly reduced during the training. In that case, the new `Tester` "tester_constrained" could be ommited, since it would then test an identical NN as the original `Tester` "tester".

5 Conclusions

Starting in August 91, a prototype of the MONNET kernel was finished in January 92. The system was ready to start serious experiments in August 92. Our experience since then proves MONNET as a valuable tool to investigate NN. Experiments with NN and standard methods can be defined very compactly, within 1 or 2 pages. Parameterized nodes and links allow generic experiments that can readily be reused. Varying parameters or even replacing a module can be done without effort. The system is efficient — for the experiments carried out the overhead of the system was negligible (<5%). Experiment definition is reasonably easy and save if using macros. The functional scope of MONNET can be widened, in principal to any extent, by adding new objects, nodes, and links. The strict object-oriented design guarantees extensibility of the system, eases the implementation of nodes and links, and is responsible for the kernel to be compact.

For batch processing MONNET can meet all demands, but some useful nodes and links are not implemented yet. It still lacks graphical I/O of objects, similar to the systematic binary and text I/O provided by the `FileIO` node. The user interface is not graphical and — more severe — debugging is rather low leveled. Extension for interactive processing and changing an experiment during execution by adding or removing nodes an links is considered but not thoroughly checked.

We are working on recursive subexperiments, based on a new node `Experiment` that represents an entire experiment. Besides the subexperiment description file parameters of the sub-nodes can be passed to this Experiment node. We call this a softwired experiment since a subexperiment description is passed to Experiment that can represent any network of nodes and links. Additional hardwired subexperiments, in which a networks of nodes and links are compiled into new nodes, are considered as the next stage to allow most efficient subexperiments. A meta language to define nodes and links is planned. The structure of new components should be defined in this language, at least the number and type of channels for nodes and the type of objects passed for both nodes and links. These definitions could serve a twofold purpose. First, they allow to check experiment syntax without actually creating the nodes and the links, a prerequisite to implement a separate interactive graphical experiment editor. A sophisticated experiment editor could additionally allow dummy experiments for semantic checks. Second, templates for the C++ implementation of the nodes and links could be generated from the abstract definitions. This relieves the programmer from some annoying though not difficult tasks. Finally, the modular data driven structure of MONNET experiments forces the idea of distributed processing: using special neuro-hardware or other LAN workstations, perhaps by means similar to that employed by the Khorus system (see 1.).

The MONNET system is not restricted to the field of NN. It is more an extension of concepts established so far to achieve modular programming. Procedures have been the first ansatz to have algorithms provided in libraries. The object oriented paradigm extends that to provide algorithms together with data structures in object classes. The MONNET system adds control to the modules, having algorithms, data structures, and control together, and provides means to assemble large applications from these software integrated circuits. That way, MONNET can be useful for complex software engineering in other fields.

References

[1] Russell R. Leighton. *The Aspirin/MIGRAINES Software Tools - User's Manual Release V5.0.* The MITRE Corporation, December 1991.

[2] Andreas Zell, Niels Mache, Tilman Sommer, and Thomas Korb. Design of the snns neural network simulator. In *Östreichische Artificial-Intelligence-Tagung*, volume 287 of *Informatik-Fachberichte*, pages 93–102, Berlin, Germany, 1991. Springer Verlag.

[3] Alexander Linden and Christoph Tietz. Combining multiple neural network paradigms and applications using sesame. In *Proceedings of the International Joint Conference on Neural Networks*, volume II, pages 528–534, Los Alamitos, CA, June 1992. IEEE Computer Society Press.

[4] A. Linden, Th. Sudbrak, and Ch. Tietz (eds.). *The SESAME Handbook — Programmers Version.* German National Research Center for Computer Science (GMD), Sankt Augustin, Germany, 1992.

[5] Marwan Jabri, Edward Tinker, and Laurens Leerink. *MUME - An environment for multi-net and multi-architectures neural simulation.* Systems Engineering and Design Automation Laboratory, University of Sydney, Sydney, Australia, 1992.

[6] The Khorus Group. *Khorus Manual - Vol. I User's Manual and Vol. II Programmer's Manual.* Department of Electrical and Computer Engineering, University of New Mexiko, Albuquerque, NM, 1992.

[7] Subrata Dasgupta. *Computer Architecure - A Modern Synthesis*, volume 2, chapter 9. John Wiley & Sons, New York, NY, 1989.

[8] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*, chapter 10. McGraw-Hill Book Company, New York, NY, 1985.

[9] J.K. Anlauf and M. Biehl. Properties of an adaptive perceptron algorithm. In R. Eckmiller, G. Hartmann, and G. Hauske, editors, *Parallel Processing in Neural Systems and Computers*, pages 153–156, Amsterdam, Netherlands, 1990. Elsevier Science Publishers B.V. (North-Holland).