Neural Pascal
A Language for Neural Network Programming

H.Peter Gumm
Dept. of Computer Science
SUNY at New Paltz
New Paltz, N.Y. 12561
gummp@snynewba.bitnet

Ferdinand B. Hergert
Siemens – Corporate Research
and Development – ZFE IS INF2
D-8000 Munich 83
hergert@ztivax.uucp

Abstract
Neural Pascal is an extension of object-oriented Pascal, designed to allow easy specification and simulation of neural networks. Syntactically, just a handful of extensions to Pascal had to be added. Mainly they are syntax for the declaration of neurons, links and nets; commands to build (and change) the net topology; generators to iterate through all elements of a linear datatype.

1. Introduction

Neural network models are frequently visualized as graphs with nodes representing neurons and edges representing synapses between the neurons. Subclasses of neurons are distinguished by their functions or by their location within the network. Often these subclasses are arranged in layers with different layers containing neurons of different functionality. The algorithms driving such a network will take advantage of the structuring of the net, or of the properties of neurons when performing calculations or when updating states of neurons.
It seems highly desirable to translate this view of a network into an executable program as directly as possible with the actual translation of the code into memory accesses and pointer references shielded from the concern of the programmer. There is no data structure present in the repertoire of imperative languages, such as Pascal, that would be suitable to represent this kind of graph-like structure as needed for neural networks.
On the other hand we did not want to design a new language from scratch but rather extend a widely used language to simplify the use of the new features. In the design of Neural Pascal it turned out, fortunately, that we could get by with only few new data structures and a handful of associated new commands. These data structures and commands were built on top of Turbo Pascal 5.5 which is an extension to standard Pascal mainly in its provision of object-oriented features. The functionality of NP's new data structures fits in nicely with an object- oriented approach. The new data types are objects with their private data and methods, and with the capability of inheriting properties and methods to subtypes.
Neuro Pascal is currently realized as a preprocessor that translates NP-code into Pascal source code, which is then compiled using Turbo-Pascal's efficient compiler. Program development takes place in a comfortable and intuitive integrated development environment that is similar in appearance and functionality to the development environment provided by Turbo Pascal.

2. The Language

Links and Ports:
Our underlying concept of a neural network is that of a directed graph whose nodes may be grouped into nodes of different types. A connection between nodes is called a 'link'. A link is a directed edge between two nodes. The main purpose of a link is to provide an access from one node to another provided there is a link connecting the two. Just as any other Pascal object, links are typed, so different link types may be required when connecting different kinds of neurons.
Links of the same type leading into one neuron are collected together into what is called 'ports' (see figure 1). A neuron which is the target for several links from other neurons will have to provide one or more ports to receive these links. By grouping incoming links into ports two aims are achieved: Firstly, strong typing is preserved by requiring links of different types to come in through separate ports , and secondly, network algorithms can appropriately handle the different kinds of connections or access exactly those nodes that are connected to a given neuron through a particular port. Access through a link is provided from the target neuron to the source neuron of the link. The reason for this lies in the fact that in updating a neuron N, all those neurons that are directly linked to N have to be consulted, i.e. for every link L arriving at N we have to access the node at the source of L. For the same reason, the type of the neuron at the source of a link is part of the type of the link.

```
TYPE
  inLink  = LINK FROM input
    weight : real
    BEGIN
      weight := random - 0.5
    END;
```
Fig. 2

Figure 2 gives an example of the definition of a link (`inLink`) that has as source an object (neuron) of type `input`. This link type has its own data field `weight` and as a matter of illustration we provided `inLink`s with an additional initialization code, which is automatically executed when such a link is created.
Figure 3 shows how neurons can be provided with appropriate ports for the links to hook into. This is done using NPs type constructor PORT. In this example the neuron of type `hidden` possesses two ports: port `entry` for links of type `inLink`, and port `lateral` to receive links of type `rigid`.

Creating a net topology:
A complete network will consist of many different kinds of neurons, grouped somehow according to their function in the net and connections between some of these neurons. In order to connect two neurons with a link, a SEW statement is supplied. SEW will create a link between the source neuron and the target neuron and hook the link into the designated port of the target neuron. The link type is inferred from the type of the port. At the same time the initialization code for the created link is executed.

```
hidden = OBJECT
  activation : real;
  entry    : PORT OF inLink;
  lateral : PORT OF rigid;
END;
```
Fig. 3

The network topology can thus be created dynamically. Accordingly, we provide the CUT statement for removing links again. CUT causes a link to be removed. Just as with SEW,

the user need not worry about storage allocation and reclamation. This is handled by the system. The syntax of SEW and CUT is given by:

```
SEW <source neuron> TO <port id> OF <target neuron>
CUT <source neuron> FROM <port id> OF <target neuron>
```

A network can now easily be modeled as an object consisting of groups of neurons. Even though the network topology can be changed dynamically, this will not be needed in many applications and it should be generally recommended to build the network topology by supplying the network object with an appropriate initialization procedure.

Accessing Neurons Through Links:
The purpose of links is to provide access paths to neurons. Given a link `li`, its source is referred to as "`li.?`". Typically, the same kind of processing is done with all links arriving at a given port. (This must be a rationale for creating an appropriate selection of ports). NP provides an iterator to do the same kind of processing with all links arriving at a given port: `ALL <linkId> TO <port> DO <statement>` will execute the `<statement>` with `<linkId>` running through all links arriving at `<port>`.
Suppose for instance that hidden neurons have been defined with a private method "Fire", this method could now be defined as given in figure 4.
An important point to observe is that this method is attached to just a neuron and does not need to know anything about the the topology of the network of which this neuron is a part.
A slight variation of the ALL statement is the WITH ALL statement :

```
WITH ALL <linkId> TO <port> DO <statement>,
```

which is equivalent to

```
ALL <linkId> TO <port> DO WITH <linkId> DO <statement>.
```

```
PROCEDURE hidden.Fire;
VAR
  sum : real;
BEGIN
  sum := 0;
  ALL li TO entry DO
    sum := sum + li.weight *
                 li.?.activation;
  ALL li TO lateral DO
    sum := sum - li.?.activation;
END;  { hidden.Fire }
```
Fig. 4

The order of processing the links in an ALL-statement is unspecified. The linkId need not be declared; its type is inferred from the type of the `<port>`. Essentially an ALL statement sets up a local block with the `<linkID>` having only the body of the ALL statement in its scope.

Inheritance:
In object oriented Pascal new types of objects can be defined by extending existing objects by new fields or methods. Since the instances of a type are called a "class" in object oriented parlance, the instances of the so extended type are said to form a "subclass". A class may have several different subclasses, and eventually the subclass hierarchy will form a tree. Since subclasses inherit all properties of their ancestor classes, objects of subclasses may appear at any place where an object of the parent class is expected.

```
TYPE
  genericLink = LINK FROM neuron;

  dendrite = LINK(genericLink)
    weight : real;
    BEGIN {init. code}
      weight := random - 0.5
    END;
```
Fig. 5

In NP, links are objects too, with the type of the neuron, where the link originates, is part of its type. Fig. 5 shows the definition of a generic link (`genericLink`) and a subclass (`dendrite`) derived from it.

3. The language environment

NP development takes place in an integrated environment with menu bars, popup windows, pulldown menus, quite similar to the environment Turbo Pascal 5.5 provides. The main menu bar provides choices for syntax checking or translating the original program into Turbo Pascal. The choice "Compile" creates an executable file by invoking the command line Turbo Pascal compiler. Some menu choices activate further pulldown menus for dealing with file handling or even to provide operating system shells, or to set switches and toggles. The environment is meant to be selfexplanatory to programmers familiar with Turbo Pascal or similar environments. The "translate" menu item shows the source code of the intermediate Turbo Pascal program in an editor window.

4. Planned Extensions

In the current version of NP links are always directed. If symmetrical links are required, we must use two links, one in either direction. The only problem with this approach is that data cannot be aliased between the two links. In the next version facilities for the creation of bidirectional links will be added.

References

1 (Borland International) *Turbo-Pascal 5.5. Object Oriented Programming Guide.* Scotts Valley, Ca. 1989.

2 H.P. Gumm, F.B. Hergert, *Neural Pascal (NP)*, Siemens Technical Report, INF2-ANN-5-89, 1989.

Figure 1