Object-Oriented Implementation of Neural Networks

Ivo Vondrak, Ph.D
Dept. of Computer Science, Technical University of Ostrava
tr.17.listopadu, Ostrava - Poruba, 708 33, Czech Republic
ivo.vondrak@vsb.cz

Abstract

Both Object-Oriented Programming (OOP) and Artificial Neural Networks (ANN) have really huge influence on the computer science. If the first one has changed approach in software design then the second one has opened new possibilities in artificial intelligence both in theory and in practice. The main aim of this paper is to demonstrate how these topics can be put together. It means how the OOP can help to model and simulate neural networks in such a way that one need not to think about implementation and he/she can concentrate only on correct usage of theoretical bases.

Introduction

Almost all processes of making software packages can be considered as a process of modelling. It means that the main objective of this procedure is to transfer real system under investigation into the computer. There were developed a lot of methods and programming techniques how this goal can be satisfied. Some of them prefer functional view on the reality, some of them deal mainly with the data or logic representation. Object-Oriented approach enables to put both above mentioned principles into one consistent environment. It means that it is possible to *reincarnate* real system into the computer model. To explain this claim the main properties of the OOP should be described as follows:

- *OOP is programming with Abstract Data Types* (ADT) what means an aggregation of related data elements together with all methods that may operate on that data type - *class*. An *object* is then an instance of this class.
- *OOP is programming with Inheritance* where inheritance is the creation of a new data type as an extension or specialization of an existing one.
- *OOP is programming with Polymorphism* which represents the concept that the same message can be interpreted in different ways by different receiving objects.

Hence, an object-oriented system can be described as a set of objects communicating with each other to achieve some results. Each object can be thought of as a small virtual computer with its own state (memory) and its own set of operations (instruction set). Computation is achieved by sending *messages* to objects. When an object receives a message it determines whether it has an appropriate operation, script, or *method* to allow it to respond to the message. The definition of the method describes how the object will react upon receiving the message. In OOP terminology, we refer to the collection of operations that define behaviour of an object as the *protocol* supported by the object. This concept is very near to the concept of the ANN paradigm where the nets are created from neurons communicating with each other by the signals passed by connections (axons). How the neural net can be reincarnated into computer model will be described in the following chapters.

Problem Decomposition - Analysis

An object-oriented solution to the problem should simulate the objects in the real NN. SW objects should be constructed to represent the neurons, the interconnections between them, the layers of the connected neurons and the whole network (Fig.1).

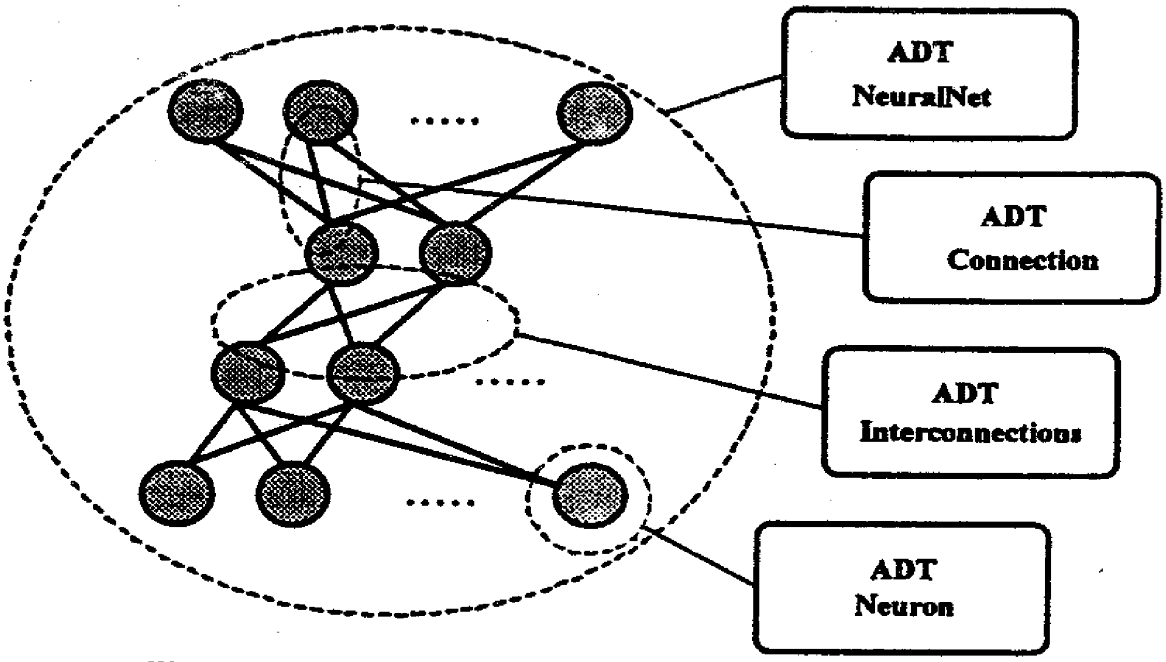

Fig.1 Decomposition of the ANN into the objects

These approach enables to find four main abstract data types represented by the following classes:

- class *Neuron* from which neurons will be created
- class *Connection* for the purpose to connect two neurons
- class *Interconnections* that represents the set (layer) of connections of the same type and behaviour
- class *NeuralNet*

Furthermore, operations on these objects (created from the above mentioned classes) would represent problem-domain tasks such as passing the signal, adaptation, self-organization, changing the topology and so on.

Object-Oriented Design

Some important boundaries have been already established for the classes based on the analysis what enables to define top-level class structure. The description of this structur reflects two basic types of relationships - *using and inheritance*. The using relationships (represented by double line) are shown on Fig.2.

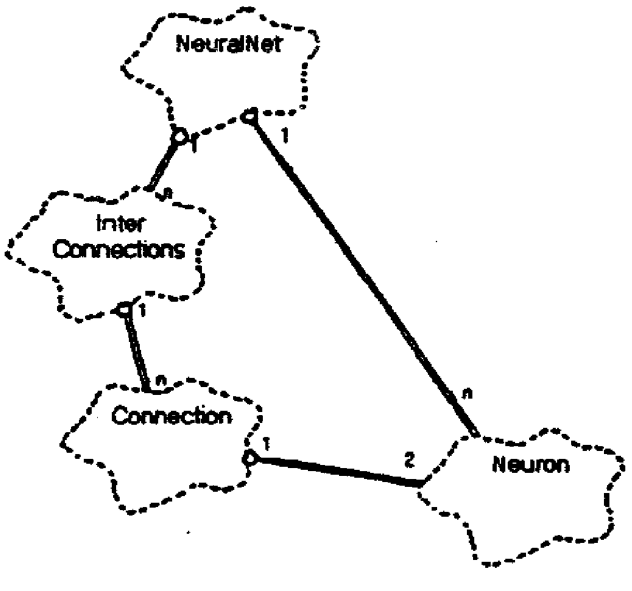

Fig.2 Class structure - using relationships

Along the lines of these using relationships their cardinalities are also expressed. For example, every neural net may contain *n* neurons, but every neuron is owned by exactly one neural net. This ownership enables neural net to have direct access to the excitation state of the neurons. On the other hand the ownership defined by the branch *neural net -> interconnections objects -> connections -> pairs of neurons* determines the topology of the ANN. Obviously models of ANN differ by various types of neurons, connections, topology of layers and so on. It means that the classification solved by the hierarchies of all above mentioned classes (ADT) will be necessary.

Hierarchy of Neurons

The explanation of the class hierarchy for neuron abstract data types will begin with the description of the general abstract model of the neuron (Fig.3).

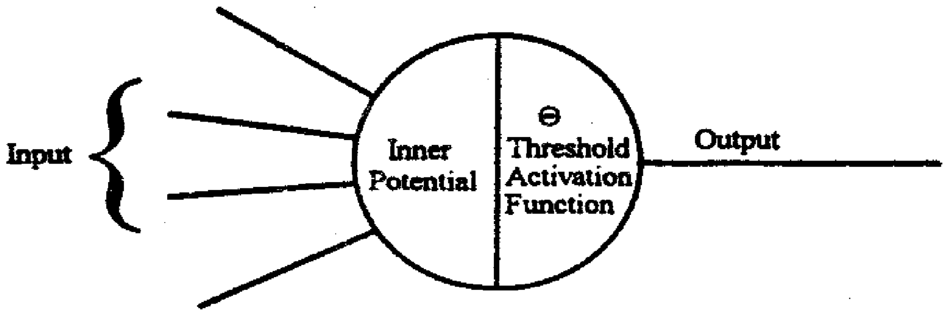

Fig.3 Abstract Neuron

This abstraction enables to define the abstract data type of the neuron as follows (presented code will be Smalltalk oriented and only the important messages will be described):

```
class:          Neuron
superclass:     Object
data elements:
        potential                    "inner potential"
        state                        "state of the excitation"
        threshold                    "threshold of the neuron"
        name                         "name of the neuron, usually represented by number"
message protocol:
        initialize: aName            "initialization of the neuron and setting a name"
        adjustPotential: aFloat      "add an input signal to the potential of the neuron"
        transfer                     "abstract method for the activation function"
        getState                     "returns the state of the neuron".
```

It is obvious that mainly the abstract method *transfer* will be defined later in the hierarchy according the type of the neuron. Its objective is to transfer input potential to the state of the excitation. If this abstract neuron is the base of the neuron's hierarchy then the whole hierarchy can be defined as follows:

```
Neuron
        BinaryNeuron
                BipolarNeuron
        KohonenNeuron
        RandomNeuron
        SigmoidalNeuron
                AdaptiveNeuron
                        IntervalNeuron
```

Binary neuron is represented be the two state neuron {0,1} with the following activation function:

```
transfer
        potential > threshold
                ifTrue: [ state := 1]
                ifFalse: [ state := 0]
```

The other neurons in the hierarchy reflect their properties, e.g. Kohonen neuron has implemented linear activation function where the inner potential is directly transfered into the state of the excitation. The subclasses based on the *SigmoidalNeuron* implement further data element *lambda* that represents the slope of their sigmoidal activation function and *error* that contains contribution of the neuron to the total error given by the output layer of neural network. This approach permits to implement the Parameters Adapting Back-Propagation (PAB) for all parameters of the neuron. It means that the class *AdaptiveNeuron* introduces new method *adjust* that can adapt neuron's threshold and slope of sigmoid based on PAB principles. The last type of neuron represented by the class *IntervalNeuron* implements the possibility to assign interval state of the excitation to the neuron via its data elements *minState* and *maxState.*

Hierarchy of Connections

The connections between neurons are important not only because of the necessity to pass the signal from one neuron to the second, but they represent the first level how the topology of neurons is defined. This abstraction enables to define the basic class *Connection* as follows (again only main methods will be described):

```
class:          Connection
superclass:     Object
data elements:
        first                       "first neuron from the couple of the neurons"
        second                      "second neuron"
        weight                      "weight of the interconnection"
message protocol:
        initialize                  "initialization of the connection"
        adjust: aFloat              "adjust a weight"
        passSignal                  "pass a signal from the first to the second neuron"
```

The method ***adjust*** simply updates old weight of the interconnection by the real value in this way:

```
adjust: aFloat
        weight := weight + aFloat
```

The signal is passed from the first neuron to the second one through method ***passSignal*** as follows:

```
passSignal
        second adjustPotential: ( weight * (first getState))
```

The whole hierarchy is very simple and it contains the following two classes:

```
        Connection
                IntervalConnection
```

Hierarchy of Interconnections Classes

The set of connections defining the part or the whole neural network represents the second and higher level of the topology definition. Again, each type of such set has its own abstract data type in the hierarchy of interconnections that begins with the base abstract class *Interconnections* defined as follows:

```
class:          Interconnections
superclass:     Object
data elements:
        connections                 "dynamic collection of the connections"
message protocol:
        initWeights                 "initialization of the weights of the connections"
        adjust                      "adjust weights of interconnections"
        passSignal                  "pass a signal between neurons"
```

```
add: aConnection                "add a connection"
remove: aConnection             "remove a connection from the collection"
```

The whole hierarchy reflects the particular properties of various types of the ANN's models and it is solved through the following classes:

```
Interconnections
    InterBAM
    InterHopfield
    InterMulti
        Grossberg
        InterBP
            InterBPInterval
        Kohonen
```

As was mentioned before the base is represented by the abstract class *Interconnections* that implements message protocol for the whole hierarchy. The others implement concrete different solutions for the methods *adjust* and *passSignal*. Furthermore, they introduce class methods responsible for the correct creation of the object with the appropriate connections. Passing the signal is similar for all classes in hierarchy and it can be solved by this code:

```
passSignal
    "Pass signal through interconnections."
    | neuron |
    connections do: [ :con |                "scan the connections"
        neuron := con getSecond.            "bind the second neuron"
        neuron initPotential.               "init its potential"
        connections do: [ :con |            "set potential by passing the signal"
            (con getSecond) = neuron
            ifTrue: [
                con passSignal
            ]
        ].
        neuron transfer                     "set state of the neuron"
    ].
```

Origin method is a little bit complicated because of the necessity to update each neuron only once while the previous code updates neurons several times and therefore the algorithm becomes inefficient. In spite of the presented algorithm that is similar for all classes the method *adjust* (responsible for the adaptation of the interconnections) differs through the hierarchy. In this paper only one example of the method that implements well-known *back-propagation* will be described as follows:

```
adjust
    "Adjust interconnections."
    | x xi d delta lambda|
    connections do: [ :con |                        "scan all connections"
        x  := con getSecond getState.
        xi := con getFirst getState.
        d  := con getSecond getError.
        lambda := con getSecond getSlope.
        delta    := -1 * learningRate * d * (x * (1 - x)) * lambda * xi.
        con adjust: delta.                          "adjust old weight"
    ]
```

Artificial Neural Networks

Class hierarchy of the ANN models is based on previously defined classes of interconnections. It means that layers of interconnections are put together to define the topology of neural networks. The abstract class *NeuralNet* that creates the beginning of the hierarchy of nets is defined in the following way:

```
class:          NeuralNet
superclass:     Object
data elements:
        inter                           "dynamic collection of the interconnections"
message protocol:
        initNet                         "initialization of the neural net"
        learning: aTrainingSet          "adaptation of the network"
        run: anInput                    "recall the information"
```

The classes for many types of ANNs are derived from this abstraction. The hierarchy owns these abstract data types that implement various paradigms:

```
NeuralNet
    BAM
        RandomBAM
    HopfieldNet
        BoltzmanMachine
    MultiLayeredNet
        BPNet
            PABNet
                IntervalPABNet
    KohonenMap
        CounterPropagNet
```

All object - neural networks - created from these classes provide management for sending messages between interconnection objects. For example *CounterPropagNet* consists of two objects: Kohonen and Grossberg layers that enable to implement the algorithm of Counter-Propagation, but there is no restriction to exchange the Grossberg layer by the Back-Propagation layer or by the PAB. This modularity is solved by the distribution of the responsibility for an adaptation and passing a signal to the appropriate objects. It means to adapt whole neural net means to adapt all interconnection objects and their adaptation is solved through connections that define them. Recalling of the information is implemented similarly. Unfortunately the source code for these classes is too large and it exceeds the possibilities of this paper.

Conclusions

Object-Oriented Programming becomes very effective tool for the software design and implementation in various fields of the human activity. The ANNs are not exception. Described class hierarchies show the main advantages of this approach, e.g. the direct reincarnation of the real nets into the computer model and the possibilities to reuse already written code. For example implementation of the interval based multi-layered network took approximately 30 minutes. Of course it is still possible to redefine or to extend hierarchies and adapt them for the solution of the concrete problem. It is possible to say that the reliability of the code is much higher than in case of some standard programming technology. This is solved mainly via above mentioned principle of the distributed responsibility in the information processing.

References

Vondrák I.: Object - Oriented Approach to the Neural Networks. Preprint, Institute of Mathematics , University of Leoben, Austria, 1992

Vondrak I.: Object - Oriented Design of Artificial Neural Networks, NEURONET'93, International Scientific Conference, Prague, Czech Republic, 1993