Yet Another Neural Network Simulator

R. Boné, M. Crucianu, J.P. Asselin de Beauville

Equipe Reconnaissance des Formes et Analyse d´Images
Laboratoire d'Informatique
Ecole d´Ingénieurs en Informatique pour l´Industrie, Université de Tours
64, avenue Jean Portalis, 37200, Tours, FRANCE
{bone, crucianu, asselin}@univ-tours.fr

Abstract: YANNS (Yet Another Neural Network Simulator) is a new object-oriented neural network simulator for feedforward networks as well as general recurrent networks. The goal of this project is to develop and implement a simulation tool that satisfies the following constraints: flexibility, ease of use, portability and efficiency. The result is a simulator with the kernel implemented as a collection of C++ classes, and with two interfaces: a high-level network specification language and a Web-based graphical user interface. These interfaces provide the means for a painless presentation of the features of neural networks to students or engineers. The object oriented design provides a valuable software environment for the researchers who wish to develop and study new architectures and algorithms.

1. Introduction

Why build another general purpose simulation tool for neural networks? The last few years have witnessed the development and distribution of a large number of neural network simulators. We could mention Aspirin/MIGRAINE [1], GENESIS [2], PlaNet [3], RCS [4], SNNS [5], Xerion [6], etc. To this non-exhaustive list of freeware tools, one must add several commercial packages. It is not our purpose here to provide a complete review of the existing neural networks simulation tools, but we can highlight some common characteristics for most of these systems: they are written in the C language for only a few platforms (Unix or Windows) and provide a system-dependent interface. Because of the very specific data structures employed, very few of them can deal with general recurrent neural network models.

For the purpose of this introduction, we can only mention that no currently available simulation tool (to the best of our knowledge) is flexible and general enough to provide support for complex and detailed models of neural networks, in a full-blown object-oriented and multi-platforms approach.

The goal of this project was to create an efficient and flexible object-oriented neural network simulation environment for teaching, research and application development in the field of neural networks. As a consequence, we followed four major guidelines in the design of YANNS: flexibility, ease of use, portability, and efficiency.

2. Flexibility

The YANNS simulator consists of 3 main components: a simulator kernel, a description language and a graphical user interface. With flexibility in mind, we decided to develop the neural network simulation kernel in C++. This allowed us to obtain a high level of encapsulation and modularity. Since there is such a large overlap between the different possible neural network architectures, as well as between the various elementary components of each of these architectures, we believe we were not too ambitious when trying to describe most of the neurons and architectures we know about. The task was made easier by the very nature of neural networks: neurons and their interconnection are common features to almost all of the neural networks, and multilayer networks are only a particular case of general recurrent neural networks. A large part of the inheritance graph was very natural to conceive. Routines for building the neurons and the networks, for performing learning or test and for interacting with the simulator during the various steps of network development should be common, whatever the architecture or the training algorithm.

The resulting kernel software consists of approximately 10,000 lines of C++ code in which various C++ classes (Fig. 1) mirror the structure and function of neurons and networks.

The first top-level objects are the Networks. Networks are described by the neurons they contain, the lists of input and output neurons and possibly a specific structure as for the layered networks. All the learning algorithms are methods of these network classes.

A neuron is an object that has a net input, an external input, an output and a transfer function for mapping the net input to the output. Neurons are connected to one another by connections. Each neuron contains the lists of its predecessors and successors. Each connection may be weighted for a simple neuron or have a list of weights for FIR or IIR neurons ([7, 8]). The FIR and IIR neurons contain connections with delays (Fig. 2 and Fig. 3). To be general enough, we allow for non-consecutive delays (Δ).

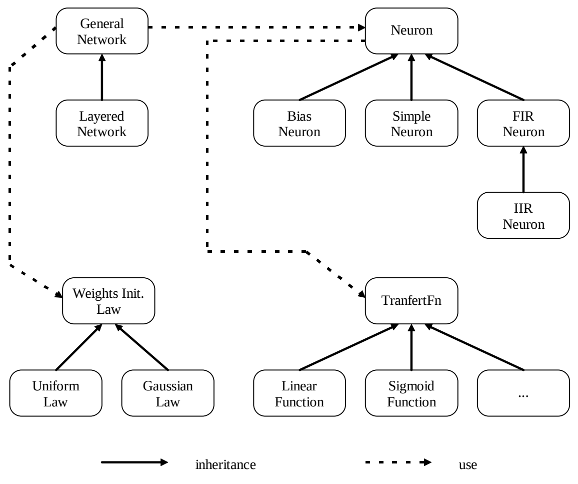


Fig. 1. Graph of the main classes.

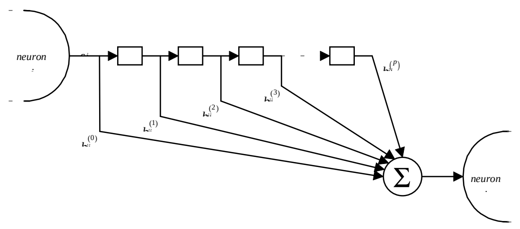


Fig. 2. FIR neuron

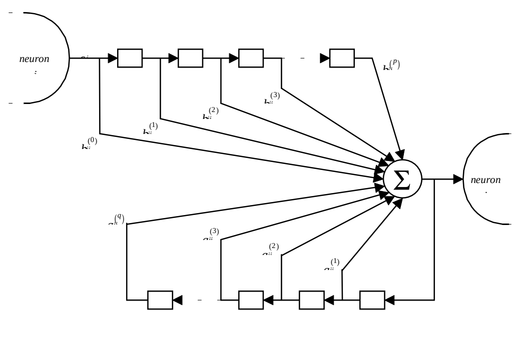


Fig. 3. IIR neuron

3. Ease of use

The building of a neural network is typically a very difficult phase of the simulation, especially for beginners. To make the simulator accessible to people with limited experience in programming or in the field of neural networks, we defined a specification language which is as simple and intuitive as possible.

In addition, the simulator uses a Web-based interface. An HTML interface with a few Java applets (e.g. for error display) is already in use. See Fig. 5 for an experiment definition form. We actually develop a fully graphical interface in Java, including graphical network specification tools and kernel control during the simulation run. We focus here on the high level network specification language.

This language provides all the network specification and control functions required. It allows one to specify the details of the architecture of the neural network, as well as the different algorithms and the parameters to employ. Various consistency controls are being performed by the parser and explicit error messages are available. Complex networks can be created quickly and easily, in a way well suited for inexperienced users who want to learn about neural network models with the help of the simulator.

In the next example, we describe a recurrent network with delays employing the Back-Propagation Through Time algorithm [9] for learning. The network is represented in Fig. 4.

The description language is powerful enough to allow us to specify a complex simulation in only a few lines of text.

A YANNS description file (see below) consists of a set of blocks, with one or more blocks for the definition of the neurons, followed by the blocks dedicated to the connections. A particular block describes the connections with the bias neuron, if it exists. The script block describes the common data set for the different learning algorithms. A subset from each data set can be selected.

```
NoNeurons=5;

Neurons [1..5] {
  TransfertFN=Linear;
}

Input=[1];
Output=[5];
```

```
Bias {
  target=[2, 3, 4];
}

Connections {
  source=[1];
  target=[2..4];
  type=Simple;
}

Connections {
  source=[2..4];
  target=[2..4];
  type=Simple;
}

Connections {
  source=[2..4];
  target=[5];
  type=FIR(0 2);
}

Script {
  Path="./";
  Learning="corpus.set";
  Stop="stop.set";
}

Learning {
  Algorithm=BPTT(0.01);
  NoMaxStep=50000;
  MinStopError=0.001
}
```

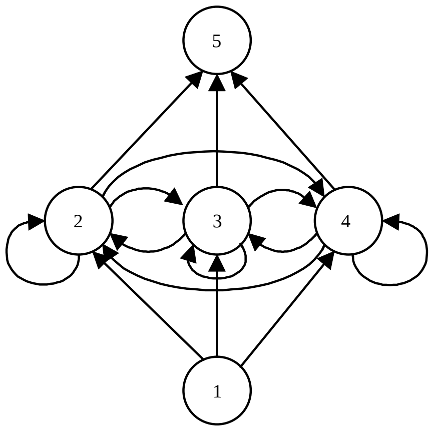


Fig. 4. A simple recurrent neural network described by the language.

Some tests performed with the help of a panel of students show that this language allows one to reduce significantly the developing time as compared to some others simulation tools. With his high level network specification language or the soon available graphical user interface, YANNS provides a complete set of learning algorithms and ways to initialize the network and control the learning:

- Back-Propagation for layered networks composed of simple, FIR and IIR neurons
- Back-Propagation Through Time for simple and FIR neurons
- RTRL and optimized RTRL
- Momentum term
- Weight decay
- Stochastic components
- Update mode: off-line or on-line
- Error function: quadratic or cross-entropy
- Stopping criterion: maximum number of iterations, minimum error on the stop set or on the learning set
- Transfer functions: linear, sigmoid, tanh, ...
- The user can specify the values for some of the weights and initialize the others randomly, from a uniform or Gaussian distribution
- Default values are available for all the parameters

4. Portability and efficiency

Several of the simulators mentioned earlier are limited to a single machine type because of the reliance on proprietary graphical systems. An easy way to develop a program for Unix, Windows and Mac platforms is to use the Java programming language. Unfortunately, this language is currently too slow, due to its nature and its youth.

We decided to use the best of Java, its ability to develop universal graphical interfaces, to add an interface which operates on any platform with a Java-compliant Web browser. But the kernel of YANNS, implementing all the learning algorithms, was developed in C++.

The Web-based interface also allows us to have control the simulator from distant machines. In the current version, the Web-based interface includes some Java applets in HTML files but most of the control is performed with the help of forms. Nevertheless, this interface allows us to use all the features of the kernel. Data files, network and experiment description files, or resulting files can be automatically transferred (an `ftp` access is required). We are now developing a pure Java interface, with increased flexibility.

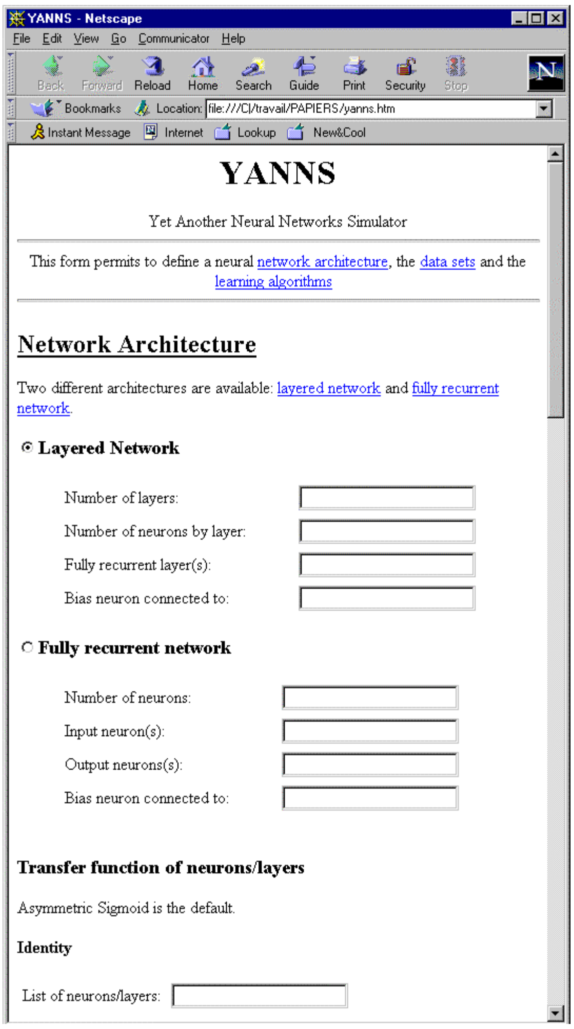
Even without a Web browser, the user can use the text interface to work with YANNS directly or to set up batch learning sessions.

The kernel of YANNS was written in a normalized version of C++. We compiled the kernel and the interpreter of the description language with numerous compilers and operating systems, with no trouble.

Taking into account the constraints of modularity and ease of use, it is necessary to keep the code as efficient as possible. This can be accomplished by using efficient data structures and numerical methods. These objectives are often inconsistent with modularity. However, our comparisons show that the learning times obtained with YANNS compare favorably to those of several other freeware simulators (when the network architecture and the learning algorithm are implemented by both).

5. Conclusion

YANNS encompasses a very large range of capabilities, allowing one to consistently describe feed-forward or recurrent neural network topologies. It already makes available numerous learning algorithms and offers a simple yet powerful description language. The kernel is written in C++. A Web-based interface is available and a full-blown graphical interface based on the Java language is now under development. We continue to follow our goals, that is developing a simulation tool available to the largest number of users, due to its ease of use and its portability, and giving them access to several recent developments in the domain of network architectures or learning algorithms.


YANNS - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: file:///C|/travail/PAPIERS/yanns.htm
Instant Message Internet Lookup New&Cool

YANNS

Yet Another Neural Networks Simulator

This form permits to define a neural network architecture, the data sets and the learning algorithms

Network Architecture

Two different architectures are available: layered network and fully recurrent network.

[✓ Layered Network]

Number of layers:

Number of neurons by layer:

Fully recurrent layer(s):

Bias neuron connected to:

Fully recurrent network

Number of neurons:

Input neuron(s):

Output neurons(s):

Bias neuron connected to:

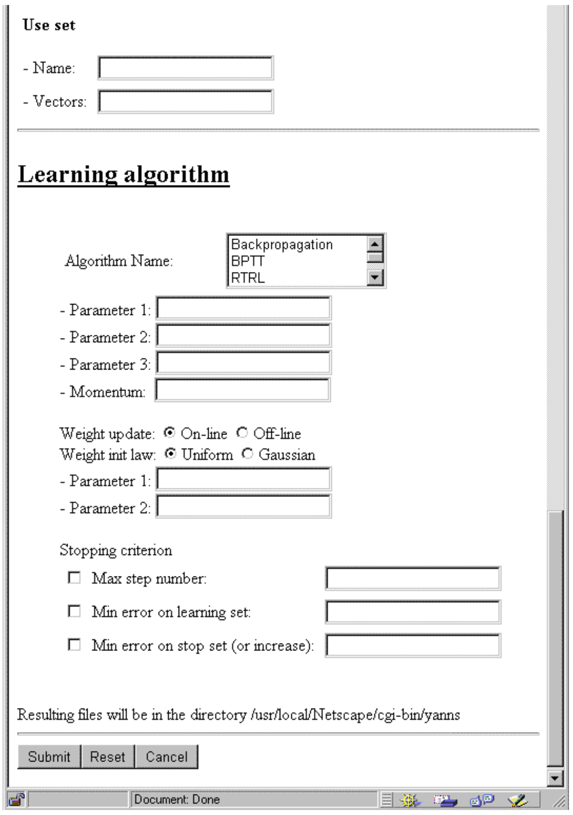
Transfer function of neurons/layers

Asymmetric Sigmoid is the default.

Identity

List of neurons/layers:

…



Use set

- Name:

- Vectors:

Learning algorithm

Algorithm Name: Backpropagation BPTT RTRL

- Parameter 1:
- Parameter 2:
- Parameter 3:
- Momentum:

Weight update: [✓ On-line] Off-line
Weight init law: [✓ Uniform] Gaussian
- Parameter 1:
- Parameter 2:

Stopping criterion

☐ Max step number:

☐ Min error on learning set:

☐ Min error on stop set (or increase):

Resulting files will be in the directory /usr/local/Netscape/cgi-bin/yanns

Fig. 5. The experiment definition form

6. Références

1. Leighton, R. and MITRE Corporation, *Aspirin/MIGRAINE*, pt.cs.cmu.edu.
2. Caltech, *GENESIS*, genesis.cns.caltech.edu.
3. University of Colorado, *PlaNet*, boulder.colorado.edu.
4. University of Rochester, *RCS* ftp.cs.rochester.edu.
5. University of Stuttgart, *SNNS*, ftp.informatik.uni-stuttgart.de.
6. University of Toronto., *Xerion*, , ftp.cs.toronto.edu.
7. Back, A.D. and A.C. Tsoi, *FIR and IIR Synapses, a New Neural Network Architecture for Time Series Modeling*. Neural Computation, 1991. **3**(3): p. 375-385.
8. Back, A., *et al. A Unifying View of some Training Algorithms for Multilayer Perceptrons with FIR Filter Synapses*. in *Neural Networks for Signal Processing IV: 4th Workshop*. 1994. Ermioni, Greece.
9. Rumelhart, D.E., G.E. Hinton, and R.J. Williams, *Learning Internal Representations by Error Propagation*, *Parallel Distributed Processing, Explorations in the Microstructure of Cognition*, ed. D.E. Rumelhart and J. McClelland. Vol. 1. 1986, Cambridge, MA: MIT Press. 318-362.