NSK, an Object-Oriented Simulator Kernel for Arbitrary Feedforward Neural Networks*

Cédric Gégout•◇⋆ Bernard Girau• Fabrice Rossi◇⋆†

◇ Ecole Normale Supérieure de Paris
45 rue d'Ulm
75005 PARIS France

• Ecole Normale Supérieure de Lyon
L.I.P.
46 avenue d'Italie
69007 LYON France

⋆ THOMSON-CSF/SDC
DPR/R4
7 rue des Mathurins
92223 BAGNEUX France

Abstract

An object-oriented neural network simulator kernel is presented. It is based on a general mathematical model for arbitrary feedforward nets. We propose a C++ implementation of this model which satisfies the following requirements : expandability (allowing an easy implementation of a new neural model), portability and efficiency (the kernel does not increase significantly computation times for classic models, compared to a direct object-oriented implementation). Learning algorithms such as gradient-based ones can be written for arbitrary nets and are therefore directly available for every particular model.

1 Introduction

Due to the lack of mathematical theories about the capabilities of neural networks, the discovery of interesting properties of these networks strongly relies on computer simulations. One of the main practical problems we have to deal with when trying to experiment a new network model is the development of an implementation of this model in order to conduct experimentations.

Many simulation softwares have been developed in order to make the realization of complex simulations easier (e.g., [1, 2, 5, 8, 13, 14, 16, 17, 18, 19, 26, 27]). In general, these simulators are based on a computer science model, such as object-oriented programming or actor languages (see [6]). They do not include a formal mathematical model, except for instance in [15] which describes a functional approach to neural nets. The main problem is that these softwares have not been designed to allow an easy implementation of a new model : when we want to add a new neural model to the general framework, many things have to be redefined. These things include of course the model in itself but also some already implemented methods (e.g., backpropagation) which have only been implemented for isolated networks (e.g., Multilayer Perceptron).

We think that this lack of generality is due to a confusion between the backpropagation algorithm and the gradient descent learning method. The backprop is an excellent algorithm for computing the partial differentials of the output of a Multilayer Perceptron (MLP) with respect to its connection weights. It is not a training method. The "backpropagation algorithm" presented in [24] has in fact two building blocks : the backprop and the gradient descent method which is a classic optimization method for arbitrary real-valuated functions. With the help of this point of view, we have designed a mathematical model for arbitrary feedforward neural nets in which backpropagation can be used to compute differentials. This mathematical model is based on ideas coming from [3], but our approach aims to be more precise than this one. It is an extension of the classic MLP model.

With the help of this theory, we have created a software, the Neural Simulator Kernel (NSK), which simplifies the implementation process of a new model. The main idea is to provide a general object-oriented model for feedforward neural nets. A neural net consists of a large number of cells, called neurons, each with inputs, parameters and outputs. Some of these cells have their inputs or their outputs connected to the *outside*. Some cells have their outputs connected to the inputs of some others cells. In a feedforward net,

*In Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, New-Orleans, November 94. Available at
`http://apiacoa.org/publications/1994/tai94.pdf`

†Up to date contact informations for Fabrice Rossi are avaible at `http://apiacoa.org/`

there is no cyclic connection between cells. In such a net, the *input cells* which have inputs connected to the outside receive *global inputs*. They propagate through the network to the *output cells* which have outputs connected to the outside : the observed outputs form the *global output* of the neural net. The outputs of a neuron are a function of its inputs and its parameters.

All classic feedforward models are particular cases of the general one. This is true in particular for Multilayered Perceptrons, Radial Basis Function Nets (as defined in [22]), Wavelet Networks (as defined in [28]), etc ... Furthermore, if the functions computed by all the neurons in an arbitrary feedforward net are differentiable, then the global function is also differentiable and its differentials can be computed with the help of an extended backpropagation algorithm (see [3, 10]), which does not depend on the chosen neurons. Our model is therefore powerful enough to handle properly standard applications which generally use gradient-based learning methods. In fact, many neural learning algorithms, and especially the gradient-based ones, do not depend on the exact nature of the neural net they are working on. In our software, new training algorithms can be included without modifying the general model. When a new model is implemented, all classic algorithms are immediately available. There is a strict separation between training algorithms and neural models, but algorithms that are only available for a restricted set of networks can also be designed.

Moreover, implementing classic models with our general-purpose software does not reduce significantly their speed compared to a direct object-oriented implementation. In order to implement a particular model, two simple tasks are to be done :

- Setting constraints on the network structure.
- Implementing the local computation done by the new model of neurons.

With the help of code factorization given by inheritance, the implementation of the local computation is no more than a simple C++ translation of the mathematical relation between inputs, parameters and outputs. This translation is even further simplified by using general mathematical classes such as vector and matrix.

The remainder of the paper is organized as follows. Section 2 gives a summary of the mathematical expression of our feedforward model and describes a classic model in this framework. Section 3 describes the design of our simulator. Section 4 shows the implementation of MLP and gives some application examples, and section 5 gives some concluding remarks.

2 General Feedforward Neural Networks

In this section, a mathematical model of arbitrary feedforward neural networks is given. We present first the mathematical description of a neural net and then explain how to compute the function defined by this net. An extended backpropagation allows us to compute the differentials of this function.

2.1 Neuron

The building block of the general model is called a *neuron*. A good mathematical model for such a neuron is to handle it as a function ([15]).

Let I, P and O be vectorial spaces of finite dimensions on $\mathbb{R}$ (i.e., $I = \mathbb{R}^i$, $P = \mathbb{R}^p$ and $O = \mathbb{R}^o$). A **neuron** N is a function from $I \times P$ to O. When $x \in I$ and $p \in P$ are respectively the input and the parameter of the neuron, its output is $N(x, p)$.

The neuron N is **differentiable** when the partial functions $N_1(., p)$ and $N_2(x, .)$ are both differentiable.

2.2 Ordered Graph

An ordered graph is only a classic graph in which nodes and incoming edges are ordered.
More precisely, an ordered graph is a triplet $G = (\mathcal{N}, \mathcal{E}, (\leq_n)_{n\in\mathcal{N}})$ where :

1. $\mathcal{N}$ is a totally ordered finite set of **nodes**.
2. $\mathcal{E}$ is a part of N^2. $e = (i, j) \in \mathcal{E}$ is an **edge** connecting node i to node j.
3. $(\leq_n)_{n\in\mathcal{N}}$ is a family of orders. For each node $n \in \mathcal{N}$, $\leq_n$ is a total order on the set of its predecessors, i.e., $Pred(n) = \{i \in \mathcal{N} \mid (i, n) \in \mathcal{E}\}$.

Some definitions
Given the order $\leq_n$, an ordered sequence of the elements of $Pred(n)$ can be defined. Let $Pred(n)_k$ be the terms of this sequence.
The set of the successors of a node is defined as $Succ(n) = \{j \in \mathcal{N} \mid (n, j) \in \mathcal{E}\}$.
A *path* of length m from n to p in a graph is a sequence of nodes, $(n_i)_{0\leq i\leq m}$ verifying : $n_0 = n$, $n_m = p$ and $\forall i < m,\ (n_i, n_{i+1}) \in \mathcal{E}$.
A *cycle* is a non-zero length path from one node to itself. A graph is cyclic when it has at least one cycle.

Intuitive point of view
A graph is in fact a list of pairs. The first element of the pair is the node itself (i.e., $n \in \mathcal{N}$) and the second element is the possibly empty list of the predecessors of the node, i.e., the indices of the nodes which

are connected to the observed node by an edge. The orders are automatically defined with the help of the lists : the first element of one list is the smallest one of the corresponding set. A complete example is given in subsection 2.6.

2.3 Neural Net

Let $G = (\mathcal{N}, \mathcal{E}, (\leq_n)_{n\in\mathcal{N}})$ be a **non-cyclic** ordered graph. Let us assume that $\mathcal{N} = \{N^1, \ldots, N^n\}$ is a set of neurons. I^k, P^k and O^k are respectively the input, parameter and output spaces of the neuron N^k.

In what follows, we will not make any differences between the neuron N^k and its index k. For instance, I^{N^k} is another notation for I^k.
The graph G is a **neural net** if it fulfills the following condition :

$$\forall N^k \in \mathcal{N}, \sum_{N^i \in Pred(N^k)} \dim O^i = \dim I^k \qquad (1)$$

This condition allows us to identify[1] the product vectorial space $\prod_i O^{Pred(N^k)_i}$ with I^k. The input of the neuron N^k is well defined as the appending of the output of its predecessors in the graph, in the order which is locally defined by the graph.

2.4 Computation model

Input, output and parameter
Let In be the set of the input nodes, i.e., $In = \{N^k \in \mathcal{N} \mid Pred(N^k) = \emptyset\}$. Let Out be the set of the output nodes, i.e., $Out = \{N^k \in \mathcal{N} \mid Succ(N^k) = \emptyset\}$. None of these sets is empty because the graph is non-cyclic. The order on $\mathcal{N}$ can be restricted to In and Out. Let then In_i (resp. Out_i) be the ordered sequence of elements of In (resp. Out).

Let I, the **input space** of the neural net, be $\prod_k I^{In_k}$, let P, the **parameter space**, be $\prod_k P^k$ and let O, the **output space**, be $\prod_k O^{Out_k}$. Intuitively, we give an input to the net when we give an input to every *input neuron*. The parameters of the net are of course the "set" of the parameters of every neuron and the output of the net is obtained by gathering the outputs of every output neuron.

How to compute the output ? Intuitive point of view
The computation of the output of the net can be explained in a four steps process :

1. The parameter vector of the net is sliced and each sub-vector is dispatched to its neuron, according to the order on $\mathcal{N}$.
2. The input vector is processed with the same method as the parameter vector i.e., it is sliced and each sub-vector is dispatched to its corresponding *input neuron*.
3. In order to compute the output of a neuron, we wait for all its predecessors to compute their output. Then all these outputs are gathered to form a new vector, the input of the neuron. The output can then be computed by using the node function.
4. The outputs of the output neurons are appended together in order to obtain the global output vector.

With this method, we can view the neural net as a neuron.

Mathematical point of view
Let $i_1, \ldots, i_p$ be the indices of the neurons belonging to In. Let $o_1, \ldots, o_q$ be the indices of the neurons belonging to Out. Let $x = (x^{i_1}, \ldots, x^{i_p}) \in I$ and $p = (p^1, \ldots, p^n) \in P$ be respectively an input vector and a parameter vector. Let us define R^k the output of the neuron N^k recursively :

- If $N^k \in In$, then $R^k = N^k(x^k, p^k)$.
- If $N^k \notin In$, let C^k be $(R^{Pred(N^k)_1}, \ldots, R^{Pred(N^k)_q})$. Then, $R^k = N^k(C^k, p^k)$.

R^k is well defined due to condition 1 and non-cyclicity of the graph (see [10]).

Let R be $(R^{o_1}, \ldots, R^{o_q})$. Then we have $R \in O$. R is **the output of the net** and is a function of x and p. We have defined a new neuron *GN*, with input space I, parameter space P and output space O.

2.5 Extended backpropagation

Differentiable neural net
Let $G = (\mathcal{N}, \mathcal{E}, (\leq_n)_{n\in\mathcal{N}})$ be a neural net. It defines a "neural net neuron", *GN*. If every N^k in $\mathcal{N}$ is differentiable, then *GN* itself is differentiable (*GN* is obtained by composition of the neuron functions). Both differential dGN_1 and dGN_2 can be computed with an extended backpropagation algorithm. The end of this subsection gives some mathematical details about this extended backpropagation.

Some definitions

[1] These vectorial spaces are only isomorphic. In order to avoid cumbersome notation, we will always identify these spaces with the help of the canonical isomorphism. For instance, the vector $((a),(b)) \in \mathbb{R}\times\mathbb{R}$ is "equal" to the vector $(a,b) \in \mathbb{R}^2$.

We need first to extend the notion of predecessor. $Pred^\infty(N^k)$ is the subset of $\mathcal{N}$, verifying $N \in Pred^\infty(N^k) \iff$ there exists a path from N to N^k.

When N^k is an element of $Pred(N^j)$, its output is only a part of the input of N^j and more precisely the ith part of this input. Let $\partial_i N_1^j$ be the corresponding part of the jacobian matrix dN_1^i. This new matrix is obtained by extracting the columns corresponding to the inputs coming from node N^k. We have then : $dN_1^j = (\partial_1 N_1^j \ \partial_2 N_1^j \ \ldots \ \partial_l N_1^j)$.

Backpropagation
Let dR_k^j be the "backpropagated signal" coming from output node N^j and evaluated at node N_k. dR_k^j is technically a jacobian matrix, the differential of the output of the neuron N^j with respect to the output of the neuron N^k and is defined as follows :

- $dR_j^j = Id$, the identity matrix of N_k's output space.
- If $N^k \notin Pred^\infty(N^j)$, then $dR_k^j = 0$.
- If $N^k \in Pred^\infty(N^j)$, then $dR_k^j = \sum_{N^i \in Succ(N^k)} dR_i^j \partial_k N_1^i$

The following property is verified (see [10]):

$$dGN_2 = \begin{pmatrix} dR_1^{o_1} dN_2^{o_1} & \ldots & dR_n^{o_1} dN_2^{o_1} \\ \vdots & \ddots & \vdots \\ dR_1^{o_q} dN_2^{o_q} & \ldots & dR_n^{o_q} dN_2^{o_q} \end{pmatrix} \quad (2)$$

A similar property is also verified for dGN_1. In order to compute dGN_2, we compute $dR_j^{o_i}$ for each output neuron N^{o_i} and for each infinite predecessor N^j of N^{o_i}. This computation is done in a reverse order, i.e., from the output nodes and to the input nodes (whereas the output computation order is direct, i.e., from the input nodes to the output nodes).

Computational cost
The backpropagation algorithm has a lower computation time than a direct computation of the differentials of the output (this direct computation method is obtained by using the composed function derivation formula). We have proven that the extended backpropagation has the same time complexity than the standard backpropagation for a MLP ([10]).

In order to reduce computation time for MLP, we can gather neurons in a layer object. This method is useful if we have optimized matrix products (see subsection 3.2).

Our general model does not introduce loss of time compared to the classic MLP model.

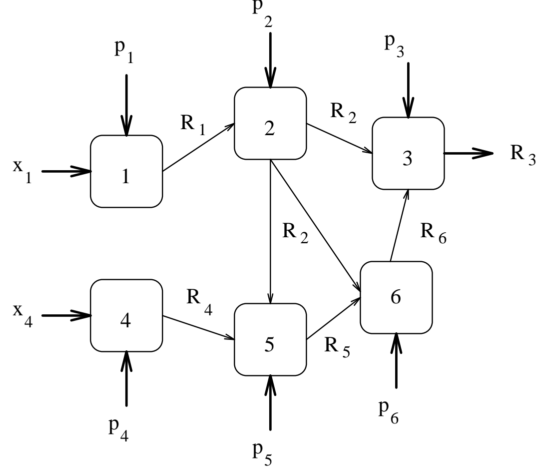

Figure 1: Output computation in a neural net

2.6 A complete example

We give here a simple example. Figure 1 shows a graphical representation of a neural net. The corresponding mathematical and simple representations of its graph are given hereafter :

1. The graph can be defined as the following list :
   $\{1,\}, \{2,[1]\}, \{3,[2,6]\}, \{4,\}, \{5,[4,2]\}, \{6,[2,5]\}$
2. $\mathcal{N} = \{1,2,3,4,5,6\}$, with the natural order.
   $\mathcal{E} = \{(1,2),(2,3),(2,5),(2,6),(4,5),(5,6),(6,3)\}$.
   As the predecessor sets $Pred(1)$ and $Pred(4)$ are empty, they do not need any order. Node 2 needs no order too because it has only one predecessor. For node 3, we have $2 \leq_3 6$. For node 5, we have $4 \leq_5 2$ and for node 6, we have $2 \leq_6 5$.

We have $In = \{1,4\}$ and $Out = \{3\}$. We have then $I = I^1 \times I^4$, $P = \prod_{i=1}^6 P^i$ and $O = O^3$. Let x be an input vector, i.e., $x = (x^1, x^4)$ and let p be a parameter vector $p = (p^1, \ldots, p^6)$. We have then :

1. $R^1 = N^1(x^1, p^1)$ because $1 \in In$ and $R^4 = N^4(x^4, p^4)$ because $4 \in In$.
2. $C^2 = R^1$ because $Pred(2) = \{1\}$. Therefore $R^2 = N^2(R^1, p^2)$.
3. $C^5 = (R^4 R^2)$ because $Pred(5) = \{2,4\}$ and $4 \leq_5 2$. Therefore $R^5 = N^5(R^4 R^2, p^5)$.
4. $C^6 = (R^2 R^5)$ because $Pred(6) = \{2,5\}$ and $2 \leq_6 5$. Therefore $R^6 = N^6(R^2 R^5, p^6)$.
5. $C^3 = (R^2 R^6)$ because $Pred(3) = \{2,6\}$ and $2 \leq_3 6$. Therefore $R^3 = N^3(R^2 R^6, p^3)$.

Let us now study the partial differentials of node N^3 with respects to the outputs of node N^2 and N^4 for instance. We have :

- $dR^3_4 = dR^3_5 \partial_4 N^5_1$,
- $dR^3_2 = dR^3_6 \partial_2 N^6_1 + dR^3_3 \partial_2 N^3_1 = dR^3_6 \partial_2 N^6_1 + \partial_2 N^3_1$.

We see in the second case how the local formula can get rid of a complex situation in which a node exercises a double (direct an indirect) influence on another one.

2.7 Input sharing

There is a difference between an input node and another one in the neural net : the former can not share its inputs whereas the latter can. For instance, two "inside" neurons can receive the output of one single neuron. On the contrary, two *input neurons* can not receive the same input from the "outside" (this is the exact meaning of the definition of I the input space of net).

This difference is useless and in fact cumbersome. We can easily get rid of it : we add before the input of the neural net a function F from the new vectorial input space NI to I. The output of the net is now $GN(F(x), p)$ for the input vector x and for the parameter vector p. The role of F is to dispatch the input vector to the *input neurons*. For instance, we can choose for F the function defined by $F(x) = (x, x)$. If we have two *input neurons*, F allows us to share a single input vector between the two neurons.

Introducing such a function does not change the backpropagation algorithm. We just have to change the final computation of dGN_1 (for details, see [10]).

Parameter sharing is also used in many neural applications. We can use the same method as for input sharing, especially since F will often have a linear form which allows computation optimization (with the help of the `Jacobian` method, see subsection 3.2).

2.8 MLP mathematical representation

We use here our model to describe the classic Multi Layer Perceptron model (see [24]). Let us recall the main features of a MLP :

- A MLP consists of several layers of neurons.
- Each neuron of a layer is connected to every neuron of the previous layer.
- In the first layer, every neuron has the same number of real valuated inputs and shares its input vector (i.e., the input vector of the net) with all neurons of the layer.

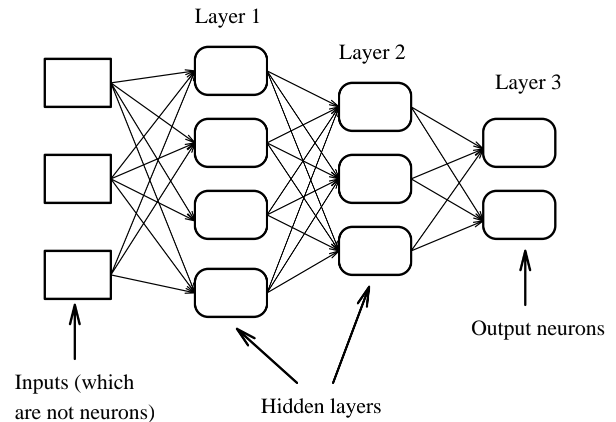

Figure 2: A multilayer perceptron

- Every neuron has a transfer function f and connection weights a_i describing the connection coming from previous layer neurons. It has a threshold b above which the neuron is "activated". If the input of a neuron is $(x_1, \ldots, x_p)$, then its output is $f(\sum_{i=1}^{n} a_i x_i + b)$.
- Usually, the neuron transfer function is the same for all neurons and is a sigmoid (e.g., $f(x) = \frac{1-e^{-x}}{1+e^{x}}$). MLP with a linear transfer function in the last layer can be used also.

Figure 2 shows a graphical representation of such a MLP.

Let us explain now how to describe a MLP with the language of our model :

- Let I, P and O be respectively $\mathbb{R}^q$, $\mathbb{R}^{q+1}$ and $\mathbb{R}$. The neuron function is defined as follows. If $x = (x_1, \ldots, x_q)$ and $p = (p_1, \ldots, p_q, o)$ are respectively the input and the parameter vectors, we have $N(x, p) = f(\sum_{i=1}^{q} x_i p_i + o)$. Of course, q depends on the layer of the neuron.
- The architecture of the graph is exactly the same as the one of the MLP (which is in fact a special kind of graph).
- The neural net uses the sharing system described above to share the same input between all the *input neurons*.

This simple example shows that the main difference between our model and a classic one lies in the meaning of a connection. In our model, a connection is only a wire that conveys a vectorial information from one neuron to another. In the MLP model, a connection

has an influence on the real information it conveys. There is a very strong relationship between parameters and connections : this is not general enough to handle complex neural structures.

3 Simulator Design

Many neurons, networks, algorithms, ... have already been implemented in NSK. But this section especially deals with the general design of the kernel and with the problems of expandability. It is intended to provide a general understanding of the programming method of NSK.

The simulator consists of the following parts :

1. General tools (Lists, Vectors, etc ...).
2. Neurons.
3. Neural nets.
4. Learning algorithms.

3.1 General tools

We have implemented simple numerical classes, `List`, `Vector` and `Matrix`. Some practical features of C++ were used in order to implement them in an efficient way.

Template classes
We use intensively parametrized classes (called template classes in C++, see [25]). This is a natural approach in order to design a list class for instance. The main idea is to allow users to experiment the effect of coding the connection weights of a MLP as double, as float or as fixed point real numbers for instance. This is a very important issue when we are trying to build a real hardware implementation of a network.

Derived classes and encapsulating classes
Derived classes allow to build a common interface for several different classes. In order to obtain the right behavior from a derived class, it must be manipulated with the help of a pointer (or a reference), as explained in [25]. This is rather dangerous because pointer copy semantics is not safe and introduces memory management problems (who will delete the object and when will it be deleted ?). In order to avoid such problems, we use an encapsulating class. The purpose of this class is to handle creation, access and deletion for the encapsulated class. We obtain then all the advantages of an object used without pointer and also the right behavior for the methods of the obtained class (for details about this method, see [9]).

We use a lazy semantics for the copy of numerical types : when we write for instance that `u=v`, where `u` and `v` are vectors, `u` becomes an “alias” for `v`. The content of `v` is really copied into `u` when we really need it (i.e., when we want to modify `u`).

As vector and matrix extractions are intensively used in NSK, these methods have been implemented with a lazy semantics and they do not make useless copies of extracted values. The time needed to extract a vector from another one is constant (it depends neither on the size of the extracted vector, nor on the size of the first vector). Of course, if we try to modify the extracted vector, a real copy is done.

3.2 Neurons

A neuron is only a special kind of function. In order to collect together the output function and its differentials, we have implemented a neuron as a class : `Neuron`. It is a template class and the template parameter is the underlying type involved in the input, parameter and output spaces of the neuron (i.e., the class `Neuron<T>` works with vectors of type `Vector<T>`).

If we look at the formula which defines the back-propagation algorithm, we observe that the jacobian matrices of a neuron are used only as right members of matrix products. Therefore, they do not have to be implemented as matrices, but only as a special template class : `Jacobian`.

The purpose of such a class is to allow easy optimization of these matrix products. Sometimes, the jacobian matrices of a neuron have only few non zero terms. Matrix products involving such matrices can be optimized in order to reduce their computation time. Optimizing some pseudo-matrix class derived from the `Matrix` class is not a good idea because we have to redefine all matrix operations, which is useless. On the contrary, by using a special `Jacobian` class that can be cast into `Matrix`, we only have to redefine a semi-product.

`Jacobian` is in fact an encapsulating class for the `JacobianRep` class. A derived class has to be implemented in order to add a new optimization method. A reference counting copy system is implemented by the `Jacobian` class. The `JacobianRep` objects are created outside of the `Jacobian` but deleted inside this class.

A neuron is then the following class :

```
template<class T> class Neuron {
protected:
  typedef Vector<T> V;
  virtual
```

```
    JacobianRep<T> *d_in(const V &,
                          const V &) const=0;
  virtual
    JacobianRep<T> *d_param(const V &,
                             const V &) const=0;
public:
  virtual V operator()(const V &,
                        const V &)const=0;
  Jacobian<T> diff_in(const V &,
                       const V &) const;
  Jacobian<T> diff_param(const V &,
                          const V &)const;
  ....
};
```

In order to implement a new neuron, we only have to redefine the virtual functions, the constructor and the destructor. Let us now describe the purpose of each method :

- `operator()` computes the output of the neuron. The first argument is the input vector and the second is the parameter vector.

- `d_in` and `d_param` compute the two differentials of the neuron as `JacobianRep<T> *`. The output of these functions can be accessed with the help of the public functions `diff_in` and `diff_param`. These functions translate the `JacobianRep *` into `Jacobian`.

3.3 Neural nets

Neural net implementation is based on a `Graph` class. This class is an implementation of the ordered graph described in section 2.2. It uses a list of triplets. Each triplet consists of a node, the list of its predecessors and the list of its successors. This class is of course a template class and gives all needed methods : edge and node insertion, *In* and *Out* computation, cyclicity checking, etc...

The `NeuralNet` class is obtained as a derived class from both `Graph` and `Neuron` classes. It uses a template class `VectorialFunction` to implement input sharing (and parameter sharing, see [10]). This class is intended to provide an implementation for differentiable functions from $\mathbb{R}^p$ to $\mathbb{R}^q$. A `VectorialFunction` has got a differential which can be computed as a `Jacobian`, in order to obtain optimized computation. We use the encapsulating class `VectorialFunction` for a real class `VectorialFunctionImp`.

The `NeuralNet` class implements the extended backpropagation in order to compute the differentials of the function computed by the net. It allows to view the output of a neuron in the net and the values of the backpropagated error term.

3.4 Learning algorithms

A general implementation of learning algorithms is rather difficult. We have designed here a model for supervised learning algorithms which is rather complete.

The "general" model
We think that the best paradigm for **supervised** neural learning is a functional one : we have a function F that we want to learn, an arbitrary norm $\|.\|$ on the vectorial space to which the function belongs and we are looking for a neural net N and a parameter vector for this neural net p that minimizes the approximation error $\|F(.) - N(.,p)\|$.

In this framework a learning algorithm is a method which allows us to find such a neural net and such a parameter. We will restrict ourselves to stepwise learning. In such a learning method, we start with an initial network N_0 and an initial parameter vector p_0. With a current guess (N_i, p_i) the algorithm is able to compute a new guess (N_{i+1}, p_{i+1}) which "converges" towards a minimizing pair (N_∞, p_∞).

A learning algorithm has got an internal state which can provide information of its former steps. Genetic algorithms have been implemented this way: the internal state contains a population of networks and each step makes an evolution of this population. The obtained guess is the best individual of the population (see [11]).

C++ implementation
As we are working in the "real" world, we can not stay as abstract as above. In order to learn a function, we must compute it. In most experiments, the function is in fact a set of input-output pairs. We have then a sampled function. This notion is implemented with the help of several classes :

- The `Sample<T>` class describes a sample. We can obtain the vectorial value of this sample (this value is a `Vector<T>`).

- The `SampleSet<T>` class describes sets of samples. Iteration through a `SampleSet` is allowed. We can get the nth sample of a set. It is also possible to extract a subset of samples.

- The `SampledFunction<T>` class describes sampled functions. Methods are given to obtain the sample set of the function and to compute its output as a function of a sample. It is of course an encapsulating class for the `SampleFunctionImp` class.

The norm is implemented as an `ErrorFunction`. This class allows the computation of a distance between a neuron function (and therefore a neural net which is a special kind of neuron) and a sampled function. It uses a `SampleSet` which is a subset of the `SampleSet` of the sampled function. This set can be dynamically changes while learning which allows to implement stochastical gradient descent and similar algorithms (see section 4). The distance between the neuron function (for a fixed parameter vector) and the sampled function depends only on the output of these functions computed for the samples belonging to the sample set of the `ErrorFunction`. The differential of the error function (considered as a function of the parameter vector of the neuron) can be computed by using the extended backpropagation.

A learning algorithm is then a `SupervisedAlgo`. It has an internal state which can be reset to an initial one. A method allows to compute one step of the algorithm. In order to learn and to compute the approximation error the `SupervisedAlgo` must be linked to a `SampledFunction`, to an `ErrorFunction` and to an initial `NeuralNet` with its initial parameter vector. After each step, we can look at the obtained `NeuralNet`, at the obtained parameter vector and at the remaining approximation error. During the learning process, we can modify the sampled function and/or its sample set.

Function Minimization

Many gradient based learning algorithms can be expressed as minimization algorithms for arbitrary real valuated functions. This is true for gradient descent (with or without momentum term), for delta-bar-delta, for quickprop, for Polak-Ribiere conjugate gradient and many other algorithms (see [7]). These algorithms can be expressed in a general way in C++ as a class `MinimizingAlgo`. This class work with a `ScalarFunction` which allows to handle differentiable functions from $\mathbb{R}^p$ to $\mathbb{R}$. We have implemented the eight different algorithms presented in [7]. They can be used to minimize arbitrary functions and they are important classic training algorithms for neural nets.

We have then implemented a special `ScalarFunction` which translates a `NeuralNet`, an `ErrorFunction` and a `SampledFunction` into a real valuated function (the function takes as input a parameter vector for the neural net and gives as output the approximation error with this parameter). Therefore we have the `GradientAlgo` class, derived from the `SupervisedAlgo` class, which allows us to use the `MinimizingAlgo` algorithms.

4 Application examples

4.1 How to handle gradient descent for MLP ?

MLP

The MLP mathematical model has been presented in subsection 2.8. We have just to implement a MLP neuron, which is in fact a pseudo linear neuron. The MLP is only a special kind of `NeuralNet` and can be derived from this class in order to add methods to simplify the insertion of layer of nodes (of course this is not mandatory but useful).

Gradient descent

The simple gradient descent algorithm has the following mathematical definition for an arbitrary vectorial function F from $\mathbb{R}^q$ to $\mathbb{R}$:

- p_0 is a first guess for the minimum of the function.
- For step $i+1$, the new guess is obtained through this way : $p_{i+1} = p_i - \epsilon_i \nabla F$, where ϵ_i is a sequence of strictly positive reals.

Therefore, it can be implemented as a `MinimizingAlgo` very easily with a constant ϵ parameter or with $\epsilon_i = \frac{1}{i+1}$ for instance.

ErrorFunction

The most commonly used error function is the quadratic error. Let us assume that the sample set is $S = \{s_i\}$ with $0 \leq i \leq n$. The *local* error between the function to learn F and the MLP N is $E(s_i, p) = \|F(s_i) - N(s_i, p)\|^2$, where $\|.\|$ is the euclidean norm in the output space. If the error function has got as sample subset $S' \subset S$, then we have $E(p) = \sum_{s_i \in S'} E(s_i, p)$. The choice of the subset is very important because if S' has only one element randomly chosen among the elements of S we have an on-line learning (also called stochastic backprop) and if $S' = S$ we have an off-line learning. We have to implement an extracting class `SelectSubSet` which allows us to choose a subset of S.

Conclusion

The MLP is implemented as a `NeuralNet`. The gradient descent algorithm is a `MinimizingAlgo`. The quadratic error function is an `ErrorFunction` and we have implemented some `SelectSubSet` classes. The `GradientAlgo` class must use all of these objects and a step of its minimizing process goes through the following stages :

1. Selecting a current sample subset with the help of the `SelectSubSet` object.

2. Building a `ScalarFunction` using the neural net, the error function, the sampled function and the sample subset.

3. Calling the `MinimizingAlgo` to optimize the `ScalarFunction`.

4.2 Geometrical parametrization of MLP

In a recent article we have proposed a new method to reduce training time for multilayer perceptrons [23]. This method is based on a geometrical view of the parameters of MLP neurons. As explained in subsection 2.8, the output of a neuron in a MLP is $f(\sum_{i=1}^{n} a_i x_i + b)$ for the vectorial input $x = (x_1, \ldots, x_n)$. Let $a = (a_1, \ldots, a_n)$ be the connection vector. The output of the neuron is then $f\big((a|x) + b\big)$. In order to achieve a more stable learning process, we introduce two parameter vectors, the dilatation vector $d = (d_1, \ldots, d_n)$ and the translation vector $t = (t_1, \ldots, t_n)$. The output of the neuron is now $f\big(d\big|(x-t)\big)$. The differences between this model and the classic MLP model are very simple, but it might be rather cumbersome to modify a classic program to handle the new model (or to handle both models). With NSK we only have to add a new class derived from `Neuron`. All the classic learning methods are immediately available. Testing our new model is then rather easy.

4.3 Other works

- Experiences with wavelet networks have also been conducted with NSK (see [12]).
- Genetic algorithms are studied as learning algorithms for neural networks (and also as general optimization methods) with NSK (see [11]).

5 Conclusion and future works

In this paper, we have presented an object-oriented simulator kernel. NSK is built on a general theory of feedforward neural nets and has therefore strong mathematical bases. It is easily expandable because of its highly modular organization. It has been designed to offer rather good runtime performances and allows implementation of optimized computing. The C++ framework allows rather simple notation for mathematical algorithms and the notion of derived classes turns the adding of a new model into a very simple task.

NSK consists of approximatively 15000 lines of code and has been implemented with the Sun C++ compiler on a Sparcstation 10. We are currently modifying it in order to obtain code compatible with the GNU C++ compiler : unfortunately, unlike ANSI C, C++ is not yet really portable, but the general availability of GNU C++ on virtually every UNIX based workstation will greatly simplify the adaptation process. D.S. NSK is still under development. We are working on an extension of the feedforward model in order to deal with recurrent networks as described in [20]. We also plan to include a code generator for parallel hardware : our goal is to provide an automatic parallelization tool that can produce C code for MIMD machines from a NSK neural net. This automatic parallelization will take two different aspects : parallelization of the neural computation, using advanced parallel algorithms for task graphs (our graph-based point of view is therefore well adapted, see [4]), and parallel computation by partitioning the sample set of the `SampledFunction` to learn, as in [21]. Of course, a simulation environment for NSK will be very useful, but the first aim is to provide a kernel and almost stand-alone tools (such as the `Vector` or `MinimizingAlgo`) rather than an environment.

References

[1] M. Azema-Barac, M. Hewetson, M. Recce, J. Taylor, P. Treleaven, and M. Vellasco. Pygmalion : Neural network programming environment. In *Proc. Int. Neural Network Conf.*, volume II, pages 709–712, 1990.

[2] A. Bartoli, G. Tononi, G. Buttazo, and P. Dario. Braintracer : a software package for the simulation of complex neural networks. In *Proc. Int. Neural Network Conf.*, volume II, pages 713–716, 1990.

[3] Léon Bottou. *Une Approche théorique de l'Apprentissage Connexioniste ; Applications à la reconnaissance de la Parole.* Thèse de doctorat, Université d'Orsay, 1991.

[4] Hervé Bourdin, Bernard Girau, and Loïc Prylli. Parallelisation interne des réseaux d'opérateurs. Technical report, Ecole Normale Supérieure de Lyon, Oct. 93.

[5] Pim H.W. Buurman, Wim J. M. Philipsen, and John van Spaandonk. PLANNET : a New Neural Net Simulator. In O. Simula T. Kohonen, K. Mäkisara and J. Kangas, editors, *Artificial Neural Networks*, pages 1481–1484. Elsevier Science B.V., 1991.

[6] Benoît Derot, Philippe Escande, and Catherine Moulinoux. Nacre : a neuron-oriented programming environment. In *Proc. Neuro-Nimes*, pages 183–200, 1989.

[7] Michel Fombellida and J. Destiné. Méthodes heuristiques et méthodes d'optimisation non contraintes pour l'apprentissage des perceptrons multicouches. In *Neuro-Nîmes*, pages 349–366, 1992.

[8] L. Fuentes, J.F. Aldana, and J.M. Troya. Urano : an object-oriented artificial neural network simulation tool. In *Proc. Int. Workshop on Artificial Neural Networks*, volume 686, pages 364–369. Springer-Verlag, 1993.

[9] Cédric Gégout, Bernard Girau, and Fabrice Rossi. Spécifications de NSK. Technical report, Thomson-CSF/SDC, Août 1993.

[10] Cédric Gégout, Bernard Girau, and Fabrice Rossi. A General Feed-Forward Neural Network Model. Technical report NC-TR-95-041, NeuroCOLT, Royal Holloway, University of London, May 1995. Available at `http://apiacoa.org/publications/1995/neurocolt1995.pdf`.

[11] Cédric Gégout and Fabrice Rossi. Continuous Parameter Optimization with Genetic Algorithms. Application to Neural Network Initialization. Technical report, Thomson-CSF/SDC, Nov. 1993.

[12] Bernard Girau. Réseaux neuronaux dérivés des ondelettes : Application à l'approximation de fonctions. Raport de maîtrise, Sept. 93.

[13] Gregory L. Heileman, Michael Georgiopoulos, and William D. Roome. A General Framework for Concurrent Simulation of Neural Network Models. *IEEE Trans. on Software Engineering*, 18(7):551–562, July 1992.

[14] Phil Kohn, Jeff Bilmes, Nelson Morgan, and James Beck. Software for ANN training on a ring array processor. In *Advances in Neural Information Processing Systems*, volume IV, pages 781–788. Morgan Kaufmann, 1991.

[15] P.W.M. Koopman, L.M.W.J. Rutten, M.C.J.D. van Eeckelen, and M.J. Plasmeijer. Functional descriptions of neural networks. In *Proc. Int. Neural Network Conf.*, volume II, pages 701–704, 1990.

[16] J.-F. Leber and G. S. Moschytz. An Interactive Object-Oriented Neural Network Simulator Applied to the Recognition of Acoustical Signals. In *IEEE Int. Symp. On Circuits and Systems*, volume 6, pages 2937–2940, San Diego, May 1992.

[17] Alexander Linden and Christoph Tietz. Combining Multiple Neural Network Paradigms and Applications using SESAME. In *IJCNN*, volume II, pages 528–534, Baltimore, June 1992.

[18] Edmond Mesrobian and Josef Skrzypek. A Software Environment for Studying Computational Neural Systems. *IEEE Trans. on Software Engineering*, 18(7):575–589, July 1992.

[19] Jacob M.J. Murre and Steven E. Kleynenberg. The MetaNet Network Environment for the Development of Modular Neural Networks. In *Proc. Int. Neural Network Conf.*, volume II, pages 717–720, 1990.

[20] Olivier Nerrand, Pierre Roussel-Ragot, Léon Personnaz, Gérard Dreyfus, Sylvie Marcos, Odile

Macchi, and Christophe Vignat. Feedback neural networks for non-linear adaptive filtering. In *Proc. Neuro-Nîmes'91*, pages 753–756, November 1991.

[21] Hélène Paugam-Moisy. On a parallel algorithm for back-propagation by partitioning the training set. In *Proc. Neuro-Nîmes*, pages 53–65, 1992.

[22] Tomaso Poggio and Federico Girosi. Networks for approximation and learning. *Proc. IEEE*, 78(9):1481–1497, September 1990.

[23] Fabrice Rossi and Cédric Gégout. Geometrical Initialization, Parametrization and Control of Multilayer Perceptrons : Application to Function Approximation. In *Int. Conf. on Neural Networks*, volume I, pages 546–550, Orlando (Florida), June 1994. IEEE.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986.

[25] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1992.

[26] Gregory L. Tarr, Setven K. Rogers, and Kabrisky Matthew. Effective Neural Network Modeling in C. In O. Simula T. Kohonen, K. Mäkisara and J. Kangas, editors, *Artificial Neural Networks*, pages 1497–1500. Elsevier Science B.V., 1991.

[27] Peter Wilke. Simulation of neural networks in a distributed computing environment using neurograph. In *Proc. Int. Workshop on Artificial Neural Networks*, volume 686, pages 394–398. Springer-Verlag, 1993.

[28] Qinghua Zhang and Albert Benveniste. Wavelet networks. *IEEE Trans. On Neural Networks*, 3(6):889–898, November 1992.